“Um Método de Testes de Integração Para Sistemas Baseados em Componentes”

Josiane Aparecida Cardoso

Trabalho Final de Mestrado Profissional em Computação

Instituto de Computação
Universidade Estadual de Campinas

"Um Método de Testes de Integração Para Sistemas Baseados em Componentes"

Josiane Aparecida Cardoso

Campinas, 20 de Fevereiro de 2006

Banca Examinadora:

- Profa. Dra. Eliane Martins (Orientadora)
  Instituto de Computação – UNICAMP
- Prof. Dr. Mario Jino
  Faculdade de Engenharia Elétrica e Computação – UNICAMP
- Profa. Dra. Cecília Mary Fischer Rubira
  Instituto de Computação - UNICAMP
- Prof. Dr. Ricardo de O. Anido (Suplente)
  Instituto de Computação – UNICAMP

FICHA CATALOGRÁFICA ELABORADA PELA

BIBLIOTECA DO IMECC DA UNICAMP

Bibliotecária: Maria Júlia Milani Rodrigues – CRB8a / 2116

Cardoso, Josiane Aparecida

C179m Um método de testes de integração para sistemas baseados em componentes / Josiane Aparecida Cardoso -- Campinas, [S.P. :s.n.], 2006.

Orientadora : Eliane Martins

Trabalho final (mestrado profissional) - Universidade Estadual de Campinas, Instituto de Computação.

1. Engenharia de software. 2. Reuso. 3. Arquitetura de redes de computador. I. Martins, Eliane. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

(mjmr/ic)

Título em inglês: A method of integration testing for system based on components

Palavras-chave em inglês (Keywords): 1. Software engineering. 2. Reuse. 3. Computer network architectures

Área de concentração: Engenharia de Computação

Titulação: Mestre em Computação na Área de Engenharia de Computação

Banca examinadora: Prof. Dr. Mario Jino (FEEC/UNICAMP)

Profa. Dra. Cecília Mary Fischer Rubira (IC/UNICAMP)

Prof. Dr. Ricardo de O. Anido (IC/UNICAMP)

Data da defesa: 20/02/2006

Programa de Pós-Graduação: Mestrado Acadêmico em Computação

“Um Método de Testes de Integração Para Sistemas Baseados em Componentes”

Este exemplar corresponde à redação do Trabalho Final devidamente corrigido defendido por Josiane Aparecida Cardoso

e aprovado pela Banca Examinadora.

Campinas, 20 de Fevereiro de 2006.

Profa. Dra. Eliane Martins
(Orientadora)

Trabalho Final apresentado ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Computação na área de Engenharia de Computação.

TERMO DE APROVAÇÃO

Trabalho Final Escrito defendido e aprovado em 20 de fevereiro de 2006, pela Banca Examinadora composta pelos Professores Doutores:

Cecília Mary Fischer Rubira

Profa. Dra. Cecília Mary Fischer Rubira
IC - UNICAMP

Mario Jino

Prof. Dr. Mario Jino
FEEC - UNICAMP

Eliane Martins

Profa. Dra. Eliane Martins
IC - UNICAMP

À minha querida
mamãe que Deus levou tão cedo
mas que sempre se fez presente em
todos os momentos da minha vida.

Agradecimentos

Gostaria de agradecer a Deus por me ajudar em todos os momentos dessa longa jornada do mestrado (e da vida) e me permitir alcançar mais essa conquista.

À minha querida e amada mamãe Maria que partiu tão cedo mas que sempre esteve olhando por mim nos momentos em que mais precisei.

Ao meu querido e amado papai José que com seu jeitinho todo especial me ajudou a chegar até aqui.

À minha orientadora Eliane Martins pelo auxílio durante a escrita desse trabalho, pelas idéias e críticas sempre construtivas e por saber ser compreensiva e humana em todos os momentos em que precisei e que não foram poucos.

Ao meu irmão José Mário que embora ausente sei que sempre torceu pelo meu sucesso.

Aos amigos que conquistei na UNICAMP Camila, Patrick, Claudinha, Mozart e também a Maria Teresa do CPqD sempre dispostos ajudar.

À minha companheira de chá no CPqD, Flávia sempre disposta a minha fazer companhia para uma pausa e escutar meus comentários de cansaço, tempo, etc.

Enfim, às minhas queridas sobrinhas Ana Beatriz e Maria Vitória cujo sorriso e inocência me traziam ânimo para continuar.

Resumo

O desenvolvimento baseado em componentes vem sendo cada vez mais utilizado hoje em dia, pois permite que um sistema seja construído através da reutilização de software.

Um problema associado à reutilização de componentes é o fato de que um componente implementado para um determinado contexto pode ser reaproveitado em um outro contexto com especificações diferentes. Esse tipo de problema pode comprometer a construção de um novo sistema de software confiável baseado em componentes. Falhas devem ser encontradas não só nos componentes, mas na integração dos componentes.

No contexto de testes de integração, estratégias têm sido propostas considerando apenas abordagens estáticas com base no diagrama de classes do sistema para se definir uma ordenação para os testes de integração minimizando o número de *stubs* necessários.

Esse trabalho propõe um método de testes de integração para sistemas baseados em componentes que se baseia na arquitetura do sistema considerando uma abordagem dinâmica sem a necessidade do código fonte do componente.

Abstract

Nowadays the development based on components is being more frequently used since it allows a system to be constructed through the reuse of software.

A problem associated to the reuse of components is that one component implemented for a specific context may be reused in another context with different specifications. This kind of problem may jeopardize the construction of a new reliable software system based on components. Faults must be found not only in the components, but in the integration of the components as well.

In the context of integration testing, strategies have been proposed considering only the structural approaches based on the class diagram of the system to define an ordering for the integration testing, minimizing the number of *stubs* needed.

This work proposes a method of integration testing for system based on components, which bases itself on the architecture of the system, considering a behavioral approach without the need for the component source code.

ÍNDICE

1 CAPÍTULO 1

INTRODUÇÃO 1

1.1 Motivação e Objetivo 5

1.2 Organização do Trabalho 6

2 CAPÍTULO 7

FUNDAMENTOS PARA O DESENVOLVIMENTO BASEADO EM COMPONENTES *(DBC)* 7

2.1 Noção de Componentes 7

2.2 Desenvolvimento Baseado em Componentes 11

2.3 Processo de Desenvolvimento Baseado em Componentes 13

2.4 Arquitetura de Software Baseado em Componentes 15

3 CAPÍTULO 21

VISÃO GERAL DE TESTES DE SOFTWARE 21

3.1 Princípios da Atividade de Teste de Software 21

3.2 Fases do Teste de Software 24

3.2.1 Teste de unidade 25

3.2.2 Teste de Componente 26

3.2.3 Teste de Integração 26

3.2.4 Teste de Sistema 26

3.3 Processo de Teste de Software 26

3.4 Teste de Componentes 30

3.4.1 Perspectiva do Cliente 30

3.4.2 Perspectiva do Fornecedor 31

3.5 Melhoria da Testabilidade 31

4 CAPÍTULO 33

TÉCNICAS DE TESTES DE INTEGRAÇÃO 33

4.1 Análise de Dependência 35

4.2 Integração Big Bang 36

4.3 Integração Ascendente (Bottom Up) 37

4.4 Integração Descendente (Top Down) 40

4.5 Integração por Colaboração 44

5 CAPÍTULO 47

TRABALHOS RELACIONADOS 47

5.1 Kung et. al 49

5.2 Tai e Daniels 49

5.3 Le Traon et. al 52

5.4 Briand et. al 54

5.5 Travassos e Oliveira 57

5.6 Lima e Travassos 60

5.7 Stafford et. al (Chaining) 62

6 CAPÍTULO 65

UM MÉTODO DE TESTES DE INTEGRAÇÃO PARA SISTEMAS BASEADOS EM COMPONENTES 65

6.1 Processo de Desenvolvimento de Sistemas Baseados em Componentes com Reuso de Componentes 67

6.2 Processo de Teste de Sistemas Baseados em Componentes 69

6.2.1 Teste de Sistema 70

6.2.2 Testes de Integração 71

6.2.3 Testes de *COTS* 71

6.2.4 Teste de Componentes 72

6.2.5 Teste de Regressão 72

6.3 Proposta do Método de Teste de Integração para Sistemas Baseados em Componentes 73

6.3.1 Obter a Arquitetura do Sistema 75

6.3.2 Determinar Dependências Estruturais dos Componentes 75

6.3.3 Determinar Dependências Comportamentais dos Componentes 77
6.3.4 Determinar o Fator de Influência (*Afetados por)* 83
6.3.5 Determinar a Ordem dos Componentes para os Testes de Integração 85
6.3.6 Determinar os *Stubs* Necessários 90
6.3.7 Considerações Finais 90
7 CAPÍTULO 95
ESTUDO DE CASO 95
7.1 Aplicação do Método 96
7.2 Resultados Obtidos 98
CAPÍTULO 8 101
CONCLUSÕES 101
7.3 Contribuições 101
7.4 Trabalhos Futuros 102
REFERÊNCIAS BIBLIOGRÁFICAS 103
APÊNDICE 1 – MATRIZES DE DEPENDÊNCIAS E CÁLCULO DE FI. 112
APÊNDICE 2 – CÁLCULO DE FI E FIT. 118

Índice de figuras

FIGURA 2.1: NOTAÇÃO UML 2.0 PARA UM SISTEMA BASEADO EM COMPONENTES. 10
FIGURA 2.2: EXEMPLO - ESTILO ARQUITETURAL EM CAMADA. 17
FIGURA 3.1: PARALELISMO ENTRE AS FASES DO PROCESSO DE DESENVOLVIMENTO E DAS FASES DO TESTE DE SOFTWARE. 27
FIGURA 3.2: FASES DO PROCESSO DE TESTE. 28
FIGURA 3.3: DOCUMENTAÇÃO DE TESTES - NORMA IEEE 829. 29
FIGURA 4.1: INTEGRAÇÃO *BOTTOM UP*, CONFIGURAÇÃO DO PRIMEIRO ESTÁGIO. 38
FIGURA 4.2: INTEGRAÇÃO *BOTTOM UP* - CONFIGURAÇÃO DO SEGUNDO ESTÁGIO. 38
FIGURA 4.3: INTEGRAÇÃO *BOTTOM UP,* CONFIGURAÇÃO FINAL. 39
FIGURA 4.4: INTEGRAÇÃO *TOP DOWN* - REPRESENTAÇÃO *DEPTH-FIRST* OU *BREADTH-FIRST*. 40
FIGURA 4.5: INTEGRAÇÃO *TOP DOWN*, CONFIGURAÇÃO DO PRIMEIRO ESTÁGIO. 41
FIGURA 4.6: INTEGRAÇÃO *TOP DOWN,* CONFIGURAÇÃO DO SEGUNDO ESTÁGIO. 42
FIGURA 4.7: INTEGRAÇÃO TOP DOWN, CONFIGURAÇÃO FINAL. 43
FIGURA 4.8: EXEMPLO DE CONFIGURAÇÃO PARA A TÉCNICA DE INTEGRAÇÃO POR COLABORAÇÃO. 45
FIGURA 5.1: EXEMPLO DE ORD *(OBJECT RELATION DIAGRAM).* 48
FIGURA 5.2: ORD (OBJECT RELATION DIAGRAM) - ASSOCIAÇÕES REMOVIDAS. REPRESENTAÇÃO DOS NÚMEROS DE NÍVEL PRINCIPAL. 50
FIGURA 5.3: REPRESENTAÇÃO DAS ASSOCIAÇÕES DO NÍVEL PRINCIPAL DE NÚMERO 1. 51
FIGURA 5.4: ORD (OBJECT RELATION DIAGRAM) - ARESTAS REMOVIDAS. REPRESENTAÇÃO DOS NÚMEROS DE NÍVEL PRINCIPAL E SECUNDÁRIO. 51
FIGURA 5.6: ESTRATÉGIA APLICADA INICIALMENTE AO VÉRTICE C7. 53
FIGURA 5.7: RESULTADO FINAL PARA A ESTRATÉGIA APLICADA INICIALMENTE AO VÉRTICE C7. 54
FIGURA 5.8: ORD (OBJECT RELATION DIAGRAM) - ESTRATÉGIA DE BRIAND ET. AL [36]. 56
FIGURA 5.9: APLICAÇÃO RECURSIVA DO ALGORITMO DE TARJAN [56] - ESTRATÉGIA DE BRIAND ET. AL [36]. 57
FIGURA 5.10: MODELO EXEMPLO. 59
FIGURA 5.11: TABELA DE PRECEDÊNCIA PARA CÁLCULO DE FI E FIT – 1ª. INTERAÇÃO 59

FIGURA 5.12: TABELA DE PRECEDÊNCIA PARA CÁLCULO DE FI E FIT- 2ª. INTERAÇÃO 59
FIGURA 5.13: INTERAÇÃO DESSA ESTRATÉGIA - CÁLCULO DE FI. 62
FIGURA 5.14 : CADEIAS (CHAINS) 63
FIGURA 5.15 : NOTAÇÃO UML - DEPENDÊNCIAS ESTÁTICAS E A CORRESPONDENTE MATRIZ DE DEPENDÊNCIA. 64
FIGURA 5.17: CHAINS 64
FIGURA 6.1 - VISÃO GERAL DO PROCESSO PARA O DESENVOLVIMENTO BASEADO EM COMPONENTES COM REUSO DE COMPONENTES [46]. 69
FIGURA 6.2 - PROCESSO DE TESTE GENÉRICO ADAPTADO PARA O *DBC* [24]. 70
FIGURA 6.3: VISÃO GERAL DO MÉTODO DE TESTES DE INTEGRAÇÃO PARA SISTEMAS BASEADOS EM COMPONENTES. 74
FIGURA 6.4: DIAGRAMA DE CLASSES EXEMPLO. BASEADO NO *ORD* DA *FIGURA 5.1*. 75
FIGURA 6.5: RELATÓRIO EXEMPLO DE DEPENDÊNCIAS ENTRE MÉTODOS. MÉTODOS OBTIDOS DA *FIGURA 6.4*. 81
FIGURA 6.6: MATRIZ DE DEPENDÊNCIA COMPORTAMENTAL DO EXEMPLO *(FIGURA 6.4)*. 82
FIGURA 6.7: MATRIZ DE DEPENDÊNCIA DO EXEMPLO (FIGURA 6.4) - COMPONENTES *AFETADOS*. . 84
FIGURA 6.8: MATRIZ DE DEPENDÊNCIA DO EXEMPLO – CÁLCULO DE FIT *(1ª. INTERAÇÃO)*. 86
FIGURA 6.9: MATRIZ DE DEPENDÊNCIA EXEMPLO – CÁLCULO DE FIT *(2ª INTERAÇÃO)*. 88
FIGURA 6.10: MATRIZ DE DEPENDÊNCIA EXEMPLO – INTERAÇÕES PARA O CÁLCULO DE *FIT* 89
FIGURA 6.11: COMPARAÇÃO DE RESULTADOS OBTIDOS – TRABALHOS ESTUDADOS. 91
FIGURA 6.12: RESULTADOS OBTIDOS – MÉTODO PROPOSTO. 93
FIGURA 7.1: ARQUITETURA DO CONTROLADOR DE CALDEIRA A VAPOR. 95
FIGURA 7.2 : TRECHO DO RELATÓRIO DE DEPENDÊNCIAS GERADO PARA O SISTEMA CONTROLADOR DE CALDEIRA DE VAPOR (*DEPENDENCY FINDER*). 97
FIGURA 7.3: RESULTADOS OBTIDOS – SISTEMA CONTROLADOR DE CALDEIRA DE VAPOR. 100

1 Capítulo

Introdução

A atividade de teste sempre fez parte do processo de desenvolvimento de software e, quando bem realizada, torna-se uma forma de avaliar e agregar qualidade ao produto, reduzindo os custos e o re-trabalho. No entanto, na maioria das empresas os testes nem sempre são tratados com a devida seriedade. O que se testemunha são testes feitos muitas vezes pelo próprio analista de requisitos ou desenvolvedor, sem planejamento ou análise de cobertura, de forma estanque após o término do desenvolvimento e pouco antes da entrega do produto.

A criação de um produto de software envolve definir processos que agem sobre um conjunto de dados separados. Um produto de software pode ser liberado contendo falhas não detectadas durante a realização dos testes independentemente da linguagem de programação utilizada no desenvolvimento. Apesar de parecer simples à primeira vista, a atividade de teste exige um bom planejamento e controle durante a execução para ser bem sucedida [61].

Um dos problemas constantemente citados quando se discute a atividade de teste é o alto custo. Considerada como sendo uma das atividades mais onerosas do processo de desenvolvimento, chegando a consumir 50% dos custos [39], a atividade de teste consiste em uma atividade dinâmica que visa revelar falhas e consiste em executar o programa alvo com um conjunto de dados de entrada e determinar se ele se comporta como o esperado [19]. Além disso, o conjunto de informações obtido durante os testes é de fundamental importância para as atividades de manutenção, depuração e estimação de confiabilidade [39].

Por esse motivo a atividade de teste tornou-se, pouco a pouco, um tema de grande importância com a necessidade de adaptação de métodos práticos que assegurem a qualidade dos produtos finais, a fim de torná-los confiáveis e de fácil manutenção.

Nesse texto o uso dos termos: falha *(fault ou bug),* erro*(error), defeito(failure)* seguirá de acordo com o padrão IEEE (IEEE STD. Standard Glossary of Software Engineering Terminology, padrão 610.12/1990).

Uma falha *(fault)* em um programa é um passo, processo ou definição de dados incorretos, como por exemplo, uma instrução ou comando incorreto, introduzidos no software pelo desenvolvedor. Um engano *(mistake)* é uma ação humana que produz um resultado incorreto, como por exemplo, uma ação incorreta tomada pelo programador e podem ser cometidos na especificação e no código do sistema. Um erro *(error)* resulta de uma falha ativada em tempo de execução e constitui uma diferença entre um valor corrente e o valor esperado, ou seja, qualquer estado intermediário ou resultado inesperado na execução de um programa constitui um erro. Um defeito *(failure)* direciona o programa a produzir resultados diferentes do esperado nas fronteiras do sistema.

Conhecer os aspectos da atividade de testes como suas limitações, objetivos, formas de criação de conjunto de testes e de aplicação de técnicas de teste de acordo com o momento do projeto, facilita o trabalho de planejamento e controle.

Desenvolvem-se rapidamente novos sistemas de software em diversas áreas, possibilitando ao consumidor uma grande diversidade de escolha, porém, em função desta rapidez, a confiabilidade e a qualidade dos softwares tendem a cair.

O investimento na atividade de testes ao longo de todo o processo de desenvolvimento é cada vez maior e está se tornando um dos itens mais estudados no contexto de aprimoramento da qualidade de software.

Convém ressaltar, que esse investimento quando feito de forma equilibrada, diminui a necessidade de esforço muito grande e menos efetivo no final do desenvolvimento.

A nova visão da qualidade de software está voltada para prevenção de falhas através da melhoria do processo de desenvolvimento de software. Essa nova abordagem deixa de lado a idéia de que a atividade de teste é mero coadjuvante no processo de desenvolvimento.

É fundamental a definição de um processo de desenvolvimento, em paralelo à definição de um processo de teste, para que sejam possíveis a extração de medidas das atividades realizadas e artefatos produzidos nos projetos. Somente dessa maneira será possível a obtenção dos indicadores que, juntamente com uma equipe de testes independente e capacitada, possibilitarão a análise e comparação dos resultados dos projetos de uma empresa. Assim, estará garantida a melhoria do processo de desenvolvimento e, conseqüentemente, a melhoria da qualidade do produto entregue ao cliente.

Como argumenta Pressman [55], o processo de desenvolvimento de software envolve uma série de atividades, sendo que, mesmo com uso de métodos, técnicas e ferramentas de desenvolvimento, ainda podem permanecer falhas no produto, os quais podem ocasionar dificuldades e custos adicionais para o seu aperfeiçoamento. Esses custos associados à correção das falhas encontradas nos produtos de software têm impulsionado a adoção de atividades de verificação, validação e testes *(VV&T)* como uma atividade de garantia de qualidade durante todo o processo de desenvolvimento de software.

É importante esclarecer que os testes são usados para encontrar falhas num sistema de software [19], nunca para mostrar a sua total ausência. O máximo que as técnicas de testes conseguem fazer é mostrar a ausência de certos tipos de falhas no sistema, mas nunca garantir que esse esteja correto.

Essas atividades visam diminuir o custo das falhas, considerando o fato que na medida em que a falha é descoberta mais tarde no ciclo de desenvolvimento, o custo de sua correção cresce exponencialmente. Nenhum desenvolvedor ou analista, por mais experiente que seja, está imune a falhas de codificação e projeto.

Uma das maneiras de garantir a qualidade de um produto de software é testá-lo, para certificar-se de sua conformidade aos requisitos especificados e aos padrões adotados.

O teste de produtos de software envolve, basicamente, quatro etapas: o planejamento dos testes, o projeto de casos de teste, a execução dos testes e coleta resultante, e a avaliação de dados [19, 55]. Essas quatro etapas devem ser desenvolvidas ao longo do próprio processo de desenvolvimento e podem ser consideradas como uma atividade incremental e concretizada em três fases de teste: os testes de unidade, os testes de integração e os testes de sistema.

Essas fases tradicionalmente caracterizam o teste de programas procedimentais, no qual o teste de unidade é intraprocedimental e o teste de integração é interprocedimental. Algumas variações são consideradas para o teste de programas desenvolvidos com base no contexto de orientação a objetos *(OO)*, como discutidos por Vincenzi [3], pois há uma divergência em relação a menor unidade a ser testada, podendo ser um componente, uma classe ou um método.

Segundo Vincenzi [3], na tentativa de reduzir os custos relacionados à atividade de teste de software, têm sido propostos técnicas e critérios que auxiliam na condução e avaliação do teste de software. A diferença entre essas técnicas está na origem da informação que é utilizada para

avaliar ou construir conjunto de casos de teste sendo que, cada técnica, possui uma variedade de critérios para esse fim. Esses critérios podem ser utilizados tanto na geração de conjunto de casos de teste, quanto para a avaliação e adequação desses conjuntos.

As técnicas de testes são, em geral, classificadas em funcional, estrutural e baseadas em defeitos. Do ponto de vista comportamental do sistema, tem-se o teste baseado em máquinas finitas de estados, muito utilizado na área de protocolos de comunicação e no contexto de *OO* [3].

É importante notar que nenhuma das técnicas de teste é completa, no sentido de que nenhuma delas é, em geral, suficiente para garantir a qualidade da atividade de teste. Desta forma, técnicas de teste devem ser vistas como complementares umas às outras e a questão que se coloca é como utilizá-las de forma que as vantagens de cada uma delas sejam bem exploradas em conjunto com outras. Por isso, essas técnicas devem ser aplicadas em conjunto para assegurar um teste de melhor qualidade [32].

As técnicas e métodos baseados em orientação a objetos surgem trazendo um enfoque diferente dos métodos tradicionais e têm sido cada vez mais utilizadas, principalmente, devido ao seu potencial para reutilização de componentes de software, diminuindo assim, o custo de produção de sistema. Para que o reuso seja bem sucedido, é fundamental que o componente utilizado seja confiável.

Componentes reutilizáveis devem ser testados várias vezes durante o desenvolvimento, toda vez que forem utilizados em um novo contexto, e a cada vez que sofram alguma alteração [39].

Conforme destacado por Harrold [39], testar sistemas baseados em componentes de software, inclui: (i) o teste de programas orientados a objetos; (ii) o desenvolvimento de processos de teste efetivos, (iii) a demonstração da eficácia de critérios e (iv) estratégias de teste. Essas atividades constituem atualmente as principais direções para a área de testes.

Atualmente, a maioria das organizações desenvolvedoras de software ainda está em processo de transição para o paradigma orientado a objetos e, à medida que mais e mais organizações adotarem tal paradigma, maior será a demanda por técnicas, critérios e ferramentas que apóiem os testes de programas orientados a objetos e de sistemas baseados em componentes [39].

1.1 Motivação e Objetivo

Diversas técnicas e critérios de testes vêm sendo propostos para apoiar a seleção de um conjunto de testes e dar indicações de quão bem testado está o produto de software. No contexto de desenvolvimento baseado em componentes, ainda são poucas as evidências sobre a eficácia das técnicas e critérios propostos.

Pode ser observado que componentes de software herdam muito das características do paradigma orientado a objetos, mas a noção de componentes transcende a noção de objetos.

Reuso em orientação a objetos, em geral, representa o reuso de bibliotecas de classes considerando uma linguagem de programação específica. Componentes de software podem ser reutilizados sem o conhecimento da linguagem de programação ou ambiente no qual foram desenvolvidos, caracterizando uma forma de reuso mais genérica.

A idéia de componentes está muito relacionada com o conceito de ocultação de informação ou *encapsulamento* [8]. Assim sendo, o desenvolvimento baseado em componentes, os quais, em geral, fazem uso extensivo das características de orientação a objetos, tem motivado uma reavaliação das estratégias de teste quanto à adequação nesse novo contexto.

A arquitetura de um sistema baseado em componentes fornece o contexto de utilização de componentes neste sistema. Os componentes a serem integrados podem ter sido implementados sem o conhecimento deste sistema. As especificações da arquitetura de um sistema e de um componente permitem ao desenvolvedor decidir a respeito da reutilização de um componente pronto, chamados componentes de prateleira ou *COTS*, ou da implementação de um novo componente.

Um problema associado à reutilização de componentes é o fato de que um componente implementado para um determinado contexto pode ser reaproveitado em um outro contexto com especificações diferentes. Este tipo de problema pode comprometer a construção de um novo sistema de software confiável baseado em componentes. Falhas devem ser encontradas não só nos componentes, mas na integração dos componentes. Como a conectividade entre os componentes é o ponto chave para os sistemas baseados em componentes e uma integração imperfeita pode resultar em despesas significantes, os testes de integração tornam-se fundamentais. Sendo assim, propor um método de teste de integração para sistema baseado em

componentes que favoreça o cliente/usuário do componente e contribua para a obtenção de um processo de testes completo objetivam a realização deste trabalho.

1.2 Organização do Trabalho

Esse trabalho está divido em capítulos organizados da seguinte Forma:

Capítulo 2 - Apresenta alguns fundamentos teóricos relacionados a componentes bem como ao desenvolvimento de sistemas baseados em componentes *(DBC)* e arquitetura de software e processo de desenvolvimento baseado em componentes.

Capítulo 3 – Descreve uma visão geral de testes de software bem como alguns princípios da atividade de testes, as fases do testes de software e o processo de testes de software.

Capítulo 4 – Apresenta algumas técnicas de testes de integração incremental e não incremental.

Capítulo 5 – Apresenta alguns trabalhos estudados que formam a base para o método de testes de integração para sistemas baseados em componentes, proposto nesse trabalho.

Capítulo 6 – relacionado ao método proposto nesse trabalho composto de um exemplo para a descrição do método bem como algumas considerações em relação aos trabalhos estudados e o método proposto.

Capítulo 7 – Apresenta o estudo de caso no qual o método proposto foi aplicado e os resultados obtidos pelo mesmo.

Capítulo 8 – Conclusões, Contribuições, Trabalhos Futuros.

2 Capítulo

Fundamentos para o Desenvolvimento Baseado em Componentes *(DBC)*

Esse capítulo apresenta os fundamentos básicos relacionados a componentes de software que serão de fundamental importância para o método de testes de integração que será proposto.

A *Seção 2.1* apresenta os conceitos de componentes, interfaces e conectores. O objetivo de uma metodologia ou processo de desenvolvimento de software é sistematizar as atividades de construção de programas, distribuindo a sua complexidade em: definição do problema, desenvolvimento do sistema e manutenção [55]. O uso de processos disciplinados de desenvolvimento reduz o número de falhas introduzidas no sistema [60].

Baseando-se neste contexto a *Seção 2.2* descreve as características de um desenvolvimento baseado em componentes bem como alguns dos princípios de um processo de desenvolvimento baseado em componentes seguido pela *Seção 2.3* que descreve um processo genérico de desenvolvimento baseado em componentes.

A importância da arquitetura fica ainda mais clara no contexto do desenvolvimento baseado em componentes. Sendo assim, uma vez que na composição de sistemas, os componentes precisam interagir entre si para oferecer as funcionalidades desejadas e a arquitetura do sistema facilita a escolha de um padrão para a integração, a *Seção 2.4* descreve os conceitos relacionados à arquitetura de sistemas baseados em componentes.

2.1 Noção de Componentes

Uma das maiores vantagens na utilização de componentes de software é a criação de um mercado, que possibilita que desenvolvedores de software possam comprar os componentes e assim implementar os sistemas de uma maneira mais rápida [10].

Componentes comercializados são em sua maioria produtos padronizados, trazendo consigo todas as vantagens do processo de montagem de componentes permitindo realizar personalizações [7]. O usuário final possui, então, a alternativa de escolha dos componentes para a implementação do seu sistema [64].

No contexto da orientação a objetos, um componente pode ser entendido como uma unidade de teste básica, podendo corresponder a uma classe num sistema ou a um método específico de uma classe. Um componente pode ser entendido como um sistema de componentes [54].

Segundo Szyperski [7], um componente de software é uma unidade de composição com especificações contratuais de interfaces e explícita dependência de contexto. Um componente de software pode ser desenvolvido independentemente de ser utilizado por terceiros para composição. O que torna alguma coisa um componente não é uma aplicação específica e nem uma tecnologia de implementação específica; sendo assim, qualquer dispositivo de software pode ser considerado um componente, desde que possua uma interface definida. Esta interface deve ser uma coleção de pontos de acesso a serviços, cada um com uma semântica estabelecida.

Sistemas baseados em componentes atêm-se ao princípio da divisão e conquista de um gerenciamento complexo que consiste em dividir um grande problema em pequenas peças, resolver essas pequenas peças e então se construir uma solução mais elaborada sob um alicerce mais simples. Uma vez que esta solução vem sendo utilizada há muitos anos, em que se difere a estrutura baseada em componentes? A principal diferença é que um componente segue o princípio de objetos no qual conforme a combinação de funções os dados relacionados são encapsulados em uma única unidade [30].

Em geral, quando se utiliza um componente de software, o desenvolvedor só tem acesso à sua especificação e à interface de acesso que deve ser utilizada para a obtenção de determinada funcionalidade. O código de um componente, em geral, não é disponibilizado. Com isso, mantendo-se a mesma interface de acesso, a implementação de determinada funcionalidade pode ser alterada sem causar maiores problemas aos projetos que utilizam o componente cuja funcionalidade foi alterada.

Componentes de software podem ser reutilizados sem conhecimento da linguagem de programação ou do ambiente no qual foram desenvolvidos, caracterizando uma forma mais genérica de reuso [8].

Um componente pode assumir vários formatos, cada um relativo a uma etapa do ciclo de desenvolvimento [30]:

1. *Padrão:* formato imposto ao componente para que ele se adapte a um determinado modelo de componentes, facilitando a posterior montagem da aplicação. Exemplos de padrões de componentes são: OMG Corba [44] e Sun Enterprise JavaBeans [57].
2. *Especificação:* componente como abstração arquitetural, representando uma unidade de composição separada do restante do sistema, com funcionalidades e interfaces bem definidas. É parte da arquitetura e independente de implementação.
3. *Implementação:* componente que pode ser executado por um computador, podendo ser integrado a outro software. Esse é o conceito utilizado para os componentes de prateleira, ou *COTS*.

A especificação de um componente ocorre durante a definição da arquitetura de um sistema na qual o componente implementado ou *COTS* será integrado.

Apesar da ausência de um consenso geral sobre o conceito de componentes, um aspecto muito importante é sempre ressaltado na literatura: um componente deve *encapsular* dentro de si seu projeto e implementação, ou seja, impedir que outros componentes tenham acesso aos seus dados, e oferecer interfaces bem definidas para o meio externo. A dissociação entre especificação e implementação, possibilita a interação com outros componentes [45].

O propósito da separação de um componente em duas partes, *interfaces* e *implementação*, são propiciar a flexibilidade para que o componente possa ser conectado ou substituído por outros componentes. Com essa separação, é possível numa aplicação baseada em componentes, modificar a implementação do componente ou, até mesmo, substituí-lo por outro componente com uma especificação semelhante, sem prejudicar as interações e a funcionalidade disponível. Qualquer dependência de implementação possibilita a perda da substituição do componente [42].

A reutilização de código por substituição ou a conexão entre componentes já existentes, proporcionam redução no tempo de desenvolvimento e simplificam a manutenção pela facilidade de localização de suas operações.

Um componente apresenta suas dependências, *interfaces requeridas*, com a descrição de suas operações, e seus serviços oferecidos, *interfaces providas*. Uma interface provida identifica um

ponto de acesso aos serviços que o componente prevê para o ambiente onde é utilizado. Uma interface requerida identifica um ponto de acesso aos serviços que o componente requer no ambiente. Em sua implementação um componente deve conter as classes ou funções capazes de executar as operações providas em suas interfaces [63].

Em UML 2.0, as interfaces providas são representadas por círculos preenchidos, e as interfaces requeridas, por semicírculos. Uma conexão ilustra uma dependência entre os componentes *(Figura 2.1).*

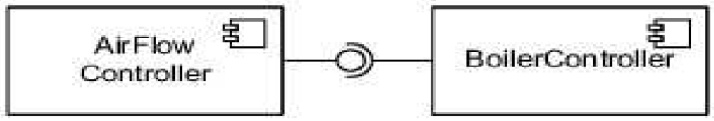


Figura 2.1: Notação UML 2.0 para um sistema baseado em componentes.

As interfaces correspondem às portas dos componentes e a comunicação com outros componentes é feita através dos *conectores.* Conectores possuem um conjunto de características que os tornam especialmente atraentes para tratar aspectos relacionados com a interação e a comunicação entre os componentes [29]. Dentre essas características destacam-se:

- Conhecimento das interfaces e referências dos componentes por eles conectados;
- Capacidade de examinar e manipular o conteúdo de requisições e respostas antes de encaminhá-las aos seus destinos finais;
- Possibilidade de controlar o direcionamento das requisições e respostas para os componentes.

Os componentes podem existir em diferentes níveis de abstração, desde uma simples sub-rotina de bibliotecas até uma aplicação completa, como o *Excel*, da *Microsoft*. A seguir cinco diferentes níveis de abstração de componentes são identificados [37]:

1. *Abstração Funcional:* O componente implementa uma única função, como uma função matemática, por exemplo, e a interface provida é a própria função.
2. *Agrupamentos Casuais:* O componente é uma coleção de entidades não relacionadas adequadamente, que podem ser, por exemplo, declarações, dados e funções, entre outras. A interface provida consiste em nomes de todas as entidades do agrupamento.

3. *Abstrações de Dados:* O componente representa uma abstração de dados ou uma classe em uma linguagem orientada a objetos. A interface provida consiste em operações para criar, modificar e acessar a abstração dos dados.

4. *Abstrações em Clusters:* O componente é um grupo de classes relacionadas que trabalham em conjunto. Essas classes relacionadas são chamadas, às vezes, de *framework* e a interface provida é a composição de todas as interfaces providas dos objetos que constituem o *framework*. De acordo com Wirfs-Brock e Johnson [67], os *frameworks* são um projeto de subsistema constituído de um conjunto de classes abstratas e concretas e das interfaces entre elas. Detalhes específicos do subsistema de aplicações são implementados com o acréscimo de componentes e o fornecimento da implementação concreta das classes abstratas nos *frameworks*. *Frameworks* raramente são aplicações propriamente ditas.

5. *Abstração de Sistema:* O componente é um sistema inteiramente fechado. Reutilizar abstrações de nível de sistema é, às vezes, chamado de reuso de *COTS*. A interface provida é chamada de API (*application programming interface* - interface de programação de aplicações), que é definida para permitir que os programas acessem os comandos e as operações do sistema.

Componentes de software possuem representação lógica e binária, ou física. Essa última é representada sob a forma de linguagem de máquina, ou de representações intermediárias, que podem ser executadas em uma máquina virtual como os *byte-codes* de Java. Um componente lógico é uma representação, em nível de projeto, de um pacote que deve estar bem separado de seu ambiente e de outros pacotes, encapsulando características e mantendo funcionalidades internas [7].

2.2 Desenvolvimento Baseado em Componentes

O desenvolvimento baseado em componentes, ou a engenharia de software baseada em componentes, emergiu no final da década de 90 como uma abordagem baseada no reuso para o desenvolvimento de sistemas de software. Sua motivação foi devido à frustração de que o

desenvolvimento orientado a objetos não tinha conduzido a um extensivo reuso, como originalmente foi sugerido.

As classes de objetos individuais eram muito detalhadas e específicas e tinham que estar associadas a uma aplicação em tempo de compilação ou quando o sistema estivesse conectado. O conhecimento detalhado das classes era necessário para sua utilização, e isso, geralmente, significava que código-fonte precisava estar disponível, apresentando problemas difíceis para a comercialização de componentes. Apesar das primeiras previsões otimistas, nenhum mercado significativo para os objetos individuais foi desenvolvido.

Os componentes são mais abstratos que as classes de objetos e podem ser considerados provedores de serviços que podem operar sem a necessidade de quaisquer outros dispositivos *(stand-alone)*. Quando um sistema precisa de algum serviço, ele requisita um serviço a um componente, sem se preocupar a respeito de onde este componente está sendo executado ou com a linguagem de programação utilizada para desenvolver esse componente [22].

Um sistema baseado em componentes é composto por componentes que interagem entre si para fornecer as funcionalidades desejadas e vem conquistando muita atenção da comunidade de engenharia de software como uma nova perspectiva para o desenvolvimento de software, ao invés daquela baseada em blocos monolíticos, onde os componentes são tão acoplados que não é possível exercitá-los separadamente e que, até bem pouco tempo, era a estratégia de desenvolvimento da maioria dos produtos de software disponíveis no mercado [8].

O desenvolvimento baseado em componentes permite que uma aplicação seja construída pela composição de componentes de software que já foram previamente especificados, construídos e testados, o que resulta em ganho de produtividade e qualidade no software produzido.

Obstáculos associados à abordagem de componentes estão relacionados à busca e seleção dos componentes para reuso. A dificuldade de localização de componentes está associada à existência de padrões de repositório e mecanismos de busca que permitam aos potenciais usuários selecionar componentes que supram suas necessidades [51]. A dificuldade de seleção está associada às deficiências apresentadas pelos mecanismos de descrição de componentes, ou seja, em especificar o que os componentes fazem e como interagem.

Componentes, em geral, são desenvolvidos para reuso, mas para que a reutilização seja possível é preciso que esses componentes sejam adaptáveis. O projeto de um componente deve

ser conduzido de tal forma que seja genérico, para que se torne adaptável a vários propósitos e não somente ao propósito ao qual foi projetado. Compatibilizar um componente originalmente incompatível pode se tornar uma atividade extremamente difícil ocasionando em mais um obstáculo para abordagem de desenvolvimento baseado em componentes.

2.3 Processo de Desenvolvimento Baseado em Componentes

Segundo Jacobson [23], um processo de desenvolvimento de software além de ser um conjunto de etapas, métodos, técnicas e práticas criadas para o desenvolvimento e manutenção de softwares e seus artefatos associados (planos, documentos, modelos, código, casos de teste, manuais, etc.) é composto por boas práticas de engenharia de software que conduzem o desenvolvimento do software, reduzindo os riscos e aumentando a confiabilidade nos sistemas.

O objetivo de um processo de desenvolvimento de software é sistematizar as atividades de construção de programas, distribuindo sua complexidade em três grupos de atividades gerais [22]: (i) definição do problema; (ii) desenvolvimento do sistema e (iii) manutenção.

Com a popularização do desenvolvimento baseado em componentes *(DBC),* a necessidade de novos processos voltados para esse paradigma é uma realidade. Isso acontece porque os processos de desenvolvimento tradicionais não são totalmente adequados ao desenvolvimento de sistemas baseados em componentes. Mais especificamente, esses processos devem conter fases e métodos que também ofereçam técnicas que permitam o empacotamento de componentes com o objetivo específico de serem reutilizados [22]. Os métodos também devem auxiliar na definição de como os componentes devem ser conectados uns com os outros para atender aos requisitos especificados, isto é, auxiliar na construção da arquitetura do software [30].

Em um processo de desenvolvimento baseado em componentes é possível enfatizar dois aspectos distintos:

1. Desenvolvimento para reuso, que enfatiza a produção de componentes propícios a serem reutilizados;

2. Desenvolvimento com reuso, que consiste na composição de sistemas a partir de componentes já existentes.

Apesar dessas duas abordagens de desenvolvimento, metodologias de desenvolvimento de software atuais não são adequadas para sistemas baseados em componentes. Metodologias mais adequadas devem contemplar os dois aspectos desse tipo de desenvolvimento [55]:

- a *engenharia de domínio* guia a construção dos componentes para que possam ser reutilizados;
- a *engenharia de aplicação* determina as fases para um desenvolvimento ascendente (*bottom up):* seleção, adaptação e testes de componentes, além das fases tradicionais (análise, projeto, implementação e testes).

De acordo com Pressman [55] a metodologia ideal para o *DBC* deve ser focada na *engenharia de aplicação* e composta pelo menos pelas seguintes fases:

1. *Fase de análise tradicional*.
2. *Fase de projeto* arquitetural do sistema.
3. *Fase de seleção dos componentes* a serem incorporados à arquitetura definida na fase anterior e que podem ser adquiridos de uma biblioteca própria, de terceiros ou serem produzidos de acordo com a necessidade do novo software.
4. *Fase de qualificação*, na qual os componentes são examinados, testados e avaliados.
5. *Fase de adaptação*, na qual o componente é adaptado para as necessidades específicas do software. Fase aplicada somente para os componentes que apresentaram problemas na fase de adaptação.
6. *Fase de composição*, na qual os componentes são integrados à arquitetura estabelecida para a aplicação.

Algumas metodologias para engenharia de aplicação têm surgido na literatura, como é o caso do Catalysis [15], Kobra [9], UML Components [30].

2.4 Arquitetura de Software Baseado em Componentes

Arquitetura de Software surgiu como uma evolução natural das abstrações de projetos, na busca de novas formas de construir sistemas de software maiores e mais complexos [43].

Além de apresentar os conectores e as interações entre componente e conectores, uma arquitetura software apresenta diferentes visões [22] de um sistema, assim como a arquitetura de uma casa possui a planta do encanamento, da parte elétrica e da divisão dos cômodos.

Diferentes visões são necessárias para especificar diferentes estruturas do sistema de software. Uma visão pode, por exemplo, ser necessária para especificar a comunicação entre processos e *threads* [1], de um sistema, outra visão pode especificar fluxos de dados.

Na medida em que o tamanho e a complexidade dos sistemas de software têm aumentado, torna-se crucial definir adequadamente a organização desses sistemas. A busca de formas adequadas de organização, além de produzir sistemas de software de mais alta qualidade, pode permitir o reuso de seus componentes em outros sistemas e facilitar a manutenção do sistema. Para que o reuso seja possível e eficiente, as funções e interfaces dos componentes devem estar bem definidas. A manutenção é facilitada pela organização do software, que facilita sua compreensão e, dada uma interface bem definida, facilita também a substituição de componentes.

Uma arquitetura de software apóia também questões importantes de projeto, tais como: a organização do sistema como uma composição de componentes, as estruturas de controle globais, os protocolos de comunicação, a composição dos elementos do projeto e a designação da funcionalidade dos componentes do projeto. Uma definição da arquitetura nos dá uma clara perspectiva de todo o sistema e do controle necessário para o seu desenvolvimento [22].

A definição de uma arquitetura de um sistema depende da definição dos requisitos do sistema a ser construído.

Com base nos requisitos, pode-se iniciar um processo de desenvolvimento de sistemas de software baseado em componentes, onde uma de suas fases é a definição da arquitetura com a

especificação de componentes. Nessa fase pode ser tomada a decisão sobre integrar componentes prontos ou desenvolver um novo componente.

De acordo com a sua natureza os requisitos de software podem, freqüentemente, ser classificados em três categorias:

1. *Requisitos funcionais*, que representam os comportamentos que um sistema deve apresentar diante de certas ações de seus usuários, como o sistema deve reagir a entradas específicas e como deve se comportar em determinadas situações. Em alguns casos, os requisitos funcionais podem também explicitamente declarar o que o sistema não deve fazer.
2. *Requisitos não-funcionais,* que quantificam determinados aspectos do comportamento do sistema. São restrições sobre os serviços oferecidos pelo sistema. Entre eles destacam-se restrições de tempo, restrições sobre o processo de desenvolvimento, padrões, entre outros.
3. *Requisitos de domínio,* que se originam do domínio da aplicação do sistema e refletem características desse domínio. Podem ser *requisitos funcionais* ou *não funcionais.*

Uma arquitetura de um sistema de software possui propriedades arquiteturais, que são derivadas dos requisitos de software do sistema e influenciam, direcionam e restringem todas as fases do ciclo de vida do software. Em geral, essas propriedades arquiteturais são derivadas dos requisitos de software não-funcionais tais como [22]:

- *Modificabilidade:* característica que define a capacidade do sistema de se adaptar a alterações de requisitos ou mesmo a inclusão de novos requisitos;
- *Reusabilidade:* característica que define o grau de reuso do componente, isto é, define quão genérico e independente da aplicação este componente é, para que possa ser reutilizado em diversas aplicações;
- *Desempenho:* tempo de resposta ou tempo de processamento de uma requisição que deve ser compatível com a realidade e necessidade do cliente;
- *Tolerância a Falhas:* capacidade do sistema de reagir e recuperar-se diante de situações excepcionais.

A *modificabilidade*, por exemplo, depende fortemente de como o sistema foi modularizado, pois isto reflete as estratégias de encapsulamento do sistema. A *reusabilidade* de componentes

depende do nível de acoplamento dos componentes do sistema. O *desempenho* depende da complexidade de comunicação entre os componentes e, especialmente, da distribuição física destes componentes. A *tolerância a falhas*, por sua vez, só é possível através da aplicação de técnicas de redundância de software e/ou hardware e tratamento de exceções.

Para garantir que uma propriedade arquitetural seja preservada durante todo o desenvolvimento do sistema utiliza-se um *estilo arquitetural* adequado para descrever a propriedade [52]. Um estilo arquitetural descreve características de uma classe de arquiteturas de software, através de tipos de componentes, conectores e regras de formação da topologia da arquitetura [34].

Um exemplo de estilo arquitetural é o estilo em camadas *(Figura 2.2),* que pode promover a *modificabilidade* e a *reusabilidade*. A camada de mais alto nível é a que faz a interface com o usuário do sistema, e a de mais baixo nível é responsável pela persistência das informações do sistema e comunicação com o sistema operacional. Entre estas duas camadas podem existir outras intermediárias.

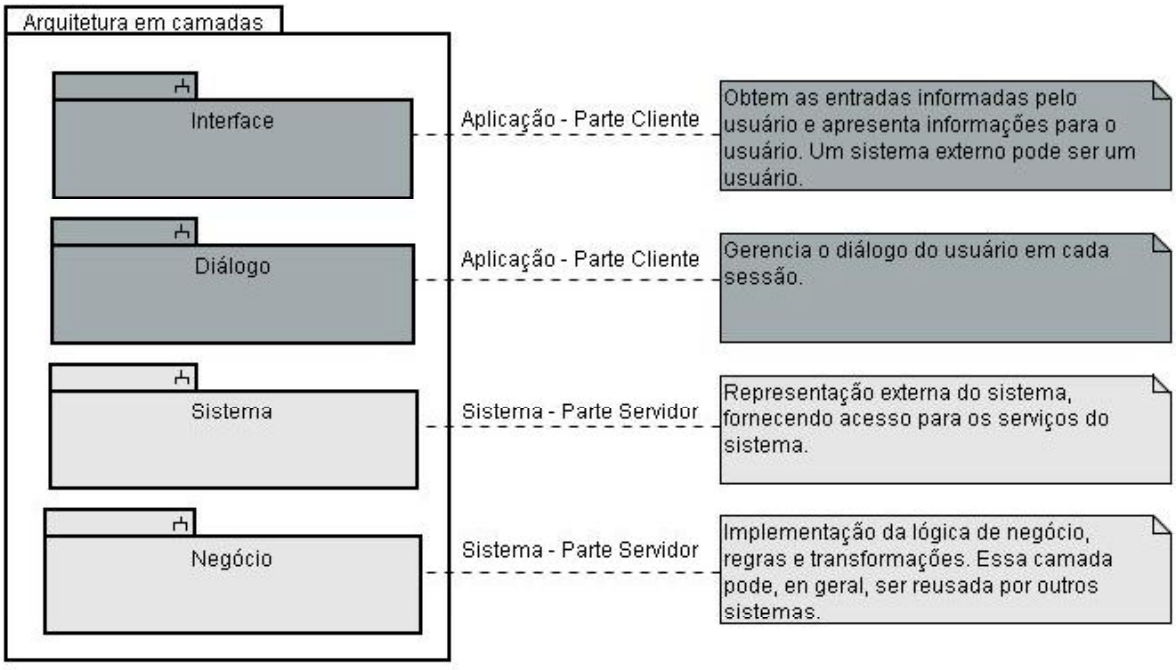


Figura 2.2: Exemplo - Estilo arquitetural em Camada.

Durante a escolha do estilo arquitetural a ser adotado, o arquiteto de software pode inclusive reutilizar arquiteturas de sistemas anteriores, baseado nas semelhanças entre as suas restrições e os seus requisitos de qualidade. Seguindo essa tendência, pesquisadores estudam uma forma de agilizar o desenvolvimento do software [35].

Segundo Stafford et. al [27, 28], o advento das linguagens formais de descrição da arquitetura *(architectural description linguages - ADLs)* torna possível antecipar a correção de uma falha durante os estágios de desenvolvimento. *ADL´s* capturam informações sobre os componentes do sistema e também como esses componentes estão interconectados. Algumas *ADL´s* também capturam informações sobre os possíveis estados dos componentes e sobre o comportamento entre os componentes envolvidos na interação de um determinado componente. Não são considerados nesse nível o comportamento e a manipulação de dados internos ao componente. Alguns exemplos de *ADL's* são *Rapide* [49], *Wright* [48] *e Acme* [12].

Ainda segundo Stafford et. al [25, 27] muitas questões permitem respostas em nível de arquitetura através da análise de uma descrição formal da arquitetura do sistema, tais como:

1. Caso esse componente esteja sendo usado por outro sistema, quais outros componentes também são requeridos?
2. Caso esse componente troque informações através de repositórios compartilhados, com quais outros componentes ele é capaz de trocar informações?
3. Caso esse componente sofra alguma alteração, que outros componentes poderão ser afetados?
4. Caso esse componente sofra alguma alteração, qual é o mínimo conjunto de testes que devem ser executados novamente?
5. Caso a especificação fonte para um componente seja verificada fora do *workspace* de modificação, quais outras especificações fontes devem também ser verificadas fora do *workspace*?
6. Caso ocorra uma falha, qual é o conjunto mínimo de componentes que devem ser inspecionados durante o processo de depuração?

As três primeiras questões são baseadas na arquitetura e é possível respondê-las exclusivamente através da análise da descrição formal do sistema. As questões 4, 5 e 6 são questão a respeito da implementação que serão respondidas com o auxílio de uma análise à nível

de arquitetura. Isso requer a habilidade de mapear certos elementos da implementação atual do sistema, tais como, uma linha de código ou um caso de teste, para a versão atual da arquitetura.

A arquitetura de um componente de software descreve a composição dos componentes e os relacionamentos entre eles. Esses relacionamentos necessitam que as interfaces entre os componentes sejam consideradas [64].

O desenvolvimento da arquitetura inicia-se com a elaboração de um projeto das interfaces do componente e, posteriormente, sua implementação. Sendo assim a arquitetura representa um conjunto de interfaces que representam conexões a serem preenchidas por componentes [64].

Para construir um sistema de software confiável e baseado em componentes que possibilite o reuso de componentes, deve-se definir uma arquitetura de software bem organizada e baseada em um estilo arquitetural que promova a *reusabilidade* e a *confiabilidade*. Essa arquitetura deve conter a especificação dos componentes, conectores *(Seção 2.1)* e suas interações. A especificação de um componente deve conter suas interfaces providas e requeridas, a operação que deve implementar [11], quais condições excepcionais pode gerar e quais condições excepcionais podem tratar, sendo que estas duas últimas condições podem ser geradas dentro do próprio ou em outros componentes do sistema [63]. A especificação de um conector deve definir quais componentes são interligados por ele. A interação entre componentes e conectores define a *configuração de componentes* de uma arquitetura de software. A configuração é a base para que o sistema estabeleça as ligações entre os componentes e conectores.

3 Capítulo

Visão Geral de Testes de Software

Desenvolvimento de sistemas é um exercício para resolução de problemas [68], uma vez que um programa é um modelo que pretende representar a solução para um problema do mundo real escrito em linguagem de máquina. Por ter tal característica, faz parte do processo de desenvolvimento determinar se esta representação da solução é válida e correta.

Esse capítulo abordará os aspectos fundamentais e alguns conceitos envolvendo a atividade de teste, através de revisão bibliográfica. A *Seção 3.1* apresenta os princípios bem como os objetivos e as limitações e objetivos da atividade de teste.

A seguir, na *Seção 3.2,* descrevem-se as fases que compõem uma atividade de testes. Na *Seção 3.3* serão apresentadas as fases de um processo de testes.

Como de acordo com Harrold et. al é possível analisar as questões que envolvem testes de sistemas baseados em componentes por duas perspectivas: a perspectiva do desenvolvedor e a do cliente, na *Seção 3.4* descrevem-se essas perspectivas relacionadas ao teste de sistemas baseados em componentes bem como as principais características e dificuldades relacionadas.

Finalmente, na *Seção 3.5* descrevem-se alguns conceitos relacionados a *testabilidade.*

3.1 Princípios da Atividade de Teste de Software

Adaptando a citação de Glen Myers [19], pode-se dizer que a atividade de teste é o processo de revisar especificações, projetos e programas com a intenção de descobrir uma falha. Alguns dos itens que são comuns a todos os autores e pesquisadores do assunto "atividade de teste" e que descrevem os fundamentos e princípios desta atividade, estão relacionados abaixo:

- A atividade de teste não prova a ausência de falhas, apenas mostra sua existência.

- Bons casos de teste são aqueles que encontram falhas no sistema até então não descobertas.
- Bons casos de teste são projetados levando em conta os requisitos do projeto.
- Um critério que pode ser utilizado para a determinação do esforço a ser gasto na atividade de atividade de teste é verificar qual o grau de severidade das conseqüências advindas do seu mau funcionamento.
- A probabilidade de encontrar uma falha em uma determinada parte do sistema é proporcional ao número de falhas já encontradas nesta parte.

A maioria dos autores concorda que os programas devem, preferencialmente, ser testados por pessoas não envolvidas no processo de desenvolvimento, ou seja, por uma equipe independente. Pode haver também a interação dos desenvolvedores com a equipe independente, justificando as decisões tomadas durante o projeto. Esta abordagem ajuda na revisão do projeto.

Segundo Myers [19], o objetivo da atividade de teste é o processo de executar um programa com a intenção de descobrir uma falha através de um bom caso de teste. Um bom caso de teste pode ser definido como aquele que tem uma alta probabilidade de revelar uma falha no software ainda não descoberta e, um caso de teste bem-sucedido é aquele que revela uma falha ainda não descoberta no sistema. Caso nenhum defeito ocorra durante os testes, isso não significa que o sistema não contenha falhas.

Os objetivos acima implicam em uma mudança drástica de ponto de vista. Pois eles apontam, contrariamente ao ponto de vista comumente defendido, de que um teste bem-sucedido é aquele em que nenhuma falha é encontrada [55].

Segundo Binder [54], existem algumas atividades que, embora não sejam atividades de teste, são atividades importantes para prevenir e remover falhas em um sistema, e, portanto podem ser usadas em complemento aos testes. Por exemplo:

- A verificação de uma análise ou modelo de projeto através da alteração de sintaxe ou simulação.
- O exame detalhado da documentação ou do código em inspeções, revisões, ou ensaios.
- A análise estática de código através do uso de um tradutor de linguagem ou verificador de código, mais especificamente, compilar um código não é testar.

- O uso de analisadores dinâmicos para identificar despejo de memória ou problemas similares.
- Depuração *(debbuging),* através de testes bem sucedidos que direcionam o *debbuging*.

Sistemas podem ser vistos como um componente uma vez que descrevem o relacionamento dos elementos de entrada com os elementos de saída. O processo de testes é utilizado para verificar se o componente realiza suas atribuições de forma confiável, ou seja, se para uma determinada entrada, a saída obtida corresponde à esperada.

Uma completa validação do componente em qualquer estágio do ciclo de vida pode ser obtida através da execução do processo de teste para cada valor de entrada possível. Caso cada etapa tenha sido bem-sucedida, o componente foi verificado; senão, uma falha foi encontrada. Sabe-se, entretanto, que esse método de teste conhecido como teste exaustivo é impraticável devido às restrições de tempo e custo para realizá-lo. Na maior parte dos casos, o domínio da função (conjunto de dados de entrada possíveis) é infinito, ou quando finito grande o bastante para fazer o número de testes requeridos inviável [59].

Visando reduzir o número potencialmente infinito de testes do processo de testes exaustivos para um número possível, deve-se encontrar um critério para selecionar elementos representativos do domínio da função. Esses critérios devem refletir tanto a descrição funcional quanto a estrutura do componente.

O subconjunto de elementos selecionado para ser usado no processo de testes é chamado de conjunto de dados de teste ou *"test set"*. No entanto, o problema é encontrar um conjunto de dados de teste adequado, que seja grande o suficiente e composto por todos os valores válidos do domínio do sistema e suficientemente pequeno para que se possam testar elementos de cada tipo de entrada do conjunto de teste.

Como as falhas existentes no software são desconhecidas, a escolha dos dados é feita com base em *critérios,* os quais servem para definir o conjunto finito de elementos a serem exercitados durante os testes. Esses elementos são parte do modelo de testes [59], o qual, na maioria dos métodos existentes, podem representar a estrutura ou a especificação do software.

Quando o modelo representa a estrutura de um software, os critérios de testes usados são ditos estruturais ou caixa branca. Já nos critérios de teste ditos funcionais, ou baseados na especificação ou ainda caixa preta, o modelo de testes representa o comportamento esperado do

software e suas características operacionais, tais como desempenho e confiabilidade *(reliability)* que é a garantia de que o sistema estará funcionando pelo tempo exigido por uma operação.

De acordo com Maldonado et. al [31] dada à diversidade de critérios de teste que têm sido estabelecidos e reconhecidos, estudos teóricos e empíricos de critérios de teste fornecem subsídios para o estabelecimento de estratégias de baixo custo e alta eficácia e são de extrema relevância para a escolha e/ou determinação de uma estratégia de teste, que em última análise passa pela escolha de critérios de teste, de forma que sejam combinadas as vantagens de cada um desses critérios.

Nos estudos teóricos procura-se estabelecer propriedades e características dos critérios de teste, avaliando os critérios de teste e analisando suas características, tais como, o custo de aplicação e a eficácia em revelar falha, comparando os critérios entre si. Nos estudos empíricos dados e estatísticas são coletados, os quais registram, por exemplo, a freqüência na qual diferentes estratégias de teste revelam a presença de falhas em uma determinada coleção de programas, fornecendo diretrizes para a escolha entre os diversos critérios disponíveis [65].

A realização de *estudos empíricos* intensificou-se nos últimos anos, procurando avaliar as diferentes técnicas e critérios de testes existentes, de modo a definir uma estratégia de teste confiável e econômica para a realização dos testes, em que o custo, eficácia e dificuldade de satisfação *(strength)* são fatores básicos para comparar a adequação de um critério.

O fator *custo* se reflete no esforço necessário para que o critério seja utilizado; em geral é medido pelo número de casos de teste necessários para satisfazer o critério. A *eficácia* refere-se à capacidade que um critério possui de detectar falhas. O fator *satisfação* refere-se à probabilidade de satisfazer um critério tendo sido satisfeito outro critério [2].

3.2 Fases do Teste de Software

A atividade de testes pode ser considerada como uma atividade incremental realizada em quatro fases: *teste de unidade, teste de integração, teste de sistema* e *teste de componente* [5]. Variações são identificadas no contexto de software orientado a objetos.

Uma estratégia de teste deve envolver tanto testes de baixo nível, que verificam se uma pequena parte do código funciona corretamente, quanto testes de alto nível, que validam funções do sistema relativas aos requisitos do cliente.

Com a divisão da atividade de teste em várias fases, o testador pode se concentrar em aspectos diferentes do software, em diferentes tipos de falhas e utilizar diferentes estratégias de seleção de dados de teste e medidas de cobertura de cada uma delas [62].

Para as fases que compõem as atividades de teste de unidade, testes de componentes e testes de integração podem ser necessário o uso de *drivers* e stubs cuja definição encontra-se na introdução da *Seção 4*.

3.2.1 Teste de unidade

O teste de unidade é um dos fundamentos principais do desenvolvimento de software e normalmente é uma atividade considerada um adjunto da etapa de codificação pois, visando uma sintaxe correta, inicia-se o projeto de caso de teste de unidade que se concentra no esforço de verificação da menor unidade de projeto de software e visa garantir que toda a funcionalidade tenha sido testada pelo menos uma vez.

O teste de unidade verifica se uma parte de código *(unidade)* executa adequadamente a sua funcionalidade, isoladamente do resto do sistema e, uma vez que uma unidade não é um programa individual, geralmente um *driver* e/ou *stub (Seção 4)* deve ser desenvolvido para cada unidade de teste. Além disso, na maioria das vezes, a adição de novas funcionalidades irá exigir mudanças nos testes de unidade para refletir a funcionalidade.

Durante esta fase utiliza-se muito a técnica de *teste estrutural* ou *caixa branca*. O teste estrutural enfoca a implementação e a estrutura da função, ou método, ou classe, ou componente. Essa fase pode ser realizada em paralelo para múltiplas unidades sob teste.

A maior resistência em se dedicar tempo para os testes de unidade embora este seja um grande aliado na localização e prevenção de falhas, é a necessidade de rapidez devido aos prazos curtos.

3.2.2 Teste de Componente

Um componente é a integração de diversas unidades com interfaces bem definidas. Nesta fase o componente é testado de acordo com a especificação das funcionalidades de sua estrutura. Também podem ser necessários *drivers* e *stub*s [5].

3.2.3 Teste de Integração

Sistemas de software são compostos por componentes que possuem interoperabilidade entre si. Os testes de integração são direcionados a descobertas de falhas na arquitetura do sistema e trata-se de uma técnica sistemática que constrói a estrutura de programa e, ao mesmo tempo, realiza testes para descobrir falhas associadas a interfaces [55], ou seja, é o caminho no qual o teste é conduzido para integrar os componentes no sistema.

Segundo Pfleeger [58], o teste de integração é o processo de verificar se os componentes do sistema, juntos, trabalham conforme descrito nas especificações do sistema.

3.2.4 Teste de Sistema

Depois que o software foi integrado, o sistema funciona como um todo, são realizados testes de sistema, cujo conjunto de testes são derivados de uma análise dos requisitos funcionais. O teste de sistema valida o software assim que ele é incorporado a um sistema maior [55].

O objetivo do teste de sistema é assegurar que o software e os demais elementos que compõem o sistema, tais como, hardware e banco de dados, combinam adequadamente e que a função/desempenho global seja obtida.

3.3 Processo de Teste de Software

Exceto para pequenos programas, os sistemas não devem ser testados como uma unidade isolada, *monolítica.* Os grandes sistemas são constituídos a partir de subsistemas, que são constituídos a partir de módulos e/ou componentes, que são compostos por procedimentos, métodos e funções. O processo de teste deve, por conseguinte, evoluir em estágios e a realização dos testes devem ser

de forma incremental, em conjunto com a implementação do sistema [22]. Segundo Pádua [66], um processo de teste possui duas grandes atividades:

1. *Preparação* onde são elaborados os planos e as especificações de teste, além da implementação dos casos de teste.
2. *Realização* onde os testes são executados e os seus resultados são analisados.

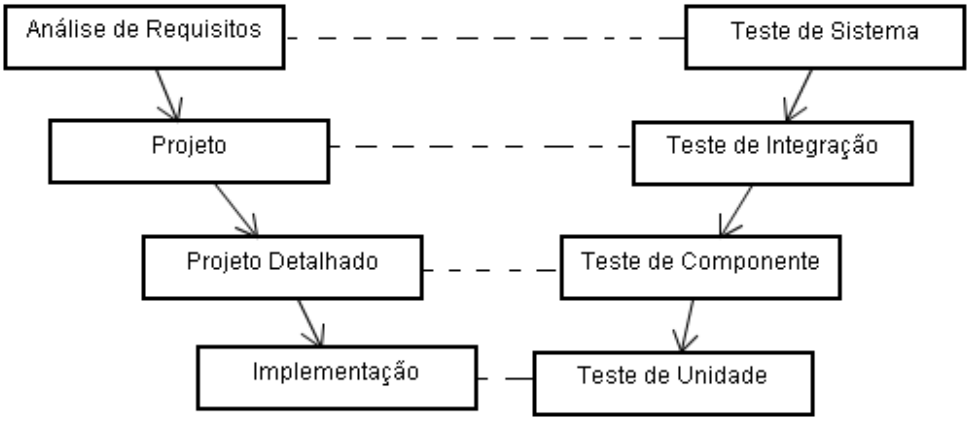


***Figura 3.1**: Paralelismo entre as fases do processo de desenvolvimento e das fases do teste de software.*

Uma vez que a implementação do sistema só é necessária na fase de realização do processo de testes, a fase de preparação dos testes pode ser realizada em paralelo à fase de desenvolvimento, como ilustra a *Figura 3.1*. O plano de testes de sistema é baseado na análise de requisitos, o plano de testes de integração, na arquitetura do sistema, e o plano de testes de componentes, no projeto detalhado.

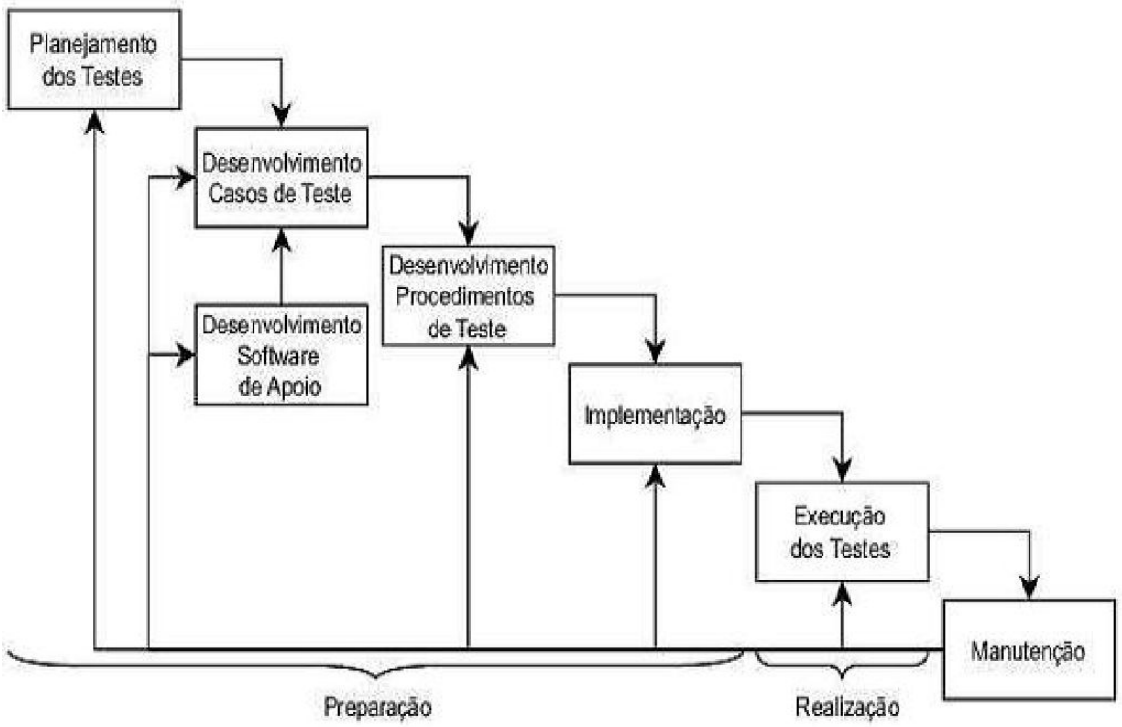


Figura 3.2: Fases do processo de teste.

A *Figura 3.2* mostra as fases de um processo de teste. Apesar da ilustração da notação do desenvolvimento em cascata, resultados de uma fase podem interferir na fase anterior. Cada uma das fases deve ser documentada: o conjunto de documentos recomendados pela norma IEEE 829 [54, 4] é ilustrado na *Figura 3.3*, de acordo com as fases nas quais são confeccionados. Essa norma considera o preenchimento dos documentos em uma *fase de registro,* porém, como essa fase ocorre em paralelo à *fase de execução* nesse trabalho foi considerada a unificação das fases [6]. A fase de preparação do processo de teste pode ser descrita em quatro passos:

1. Produz-se o documento de *plano de testes* que se refere ao *planejamento dos testes*, no qual são definidos quais serão os itens e aspectos a serem testados, as abordagens adotadas, os critérios de cobertura, os recursos necessários e o cronograma para a realização dos testes.
2. Produz-se o documento de *especificação de projeto de testes* que se refere ao *desenvolvimento dos casos de teste* e, se necessário, refere-se em paralelo ao *desenvolvimento dos softwares de apoio*. Nessa fase também se preenche o documento de *especificação dos casos de testes*, que contém as entradas e as saídas esperadas para cada caso de teste, além das

condições gerais para a sua execução e ocorre um refinamento do plano de testes, detalhando as funcionalidades e características a serem testadas.

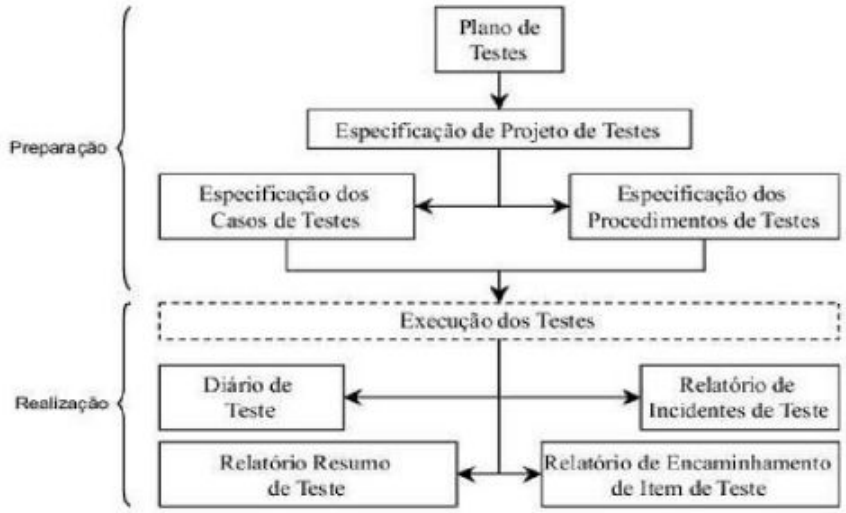


Figura 3.3: Documentação de testes - norma IEEE 829.

3. Produz-se o documento de *especificação dos procedimentos de teste* que se refere ao *desenvolvimento dos procedimentos de testes* no qual são descritos os passos para a execução de um conjunto de casos de teste.
4. Implementam-se os casos de teste e, se necessário, os softwares de apoio e prepara-se o ambiente de testes.

Na fase de realização os testes são executados e os seguintes documentos são produzidos:

- Diário de teste, com os registros cronológicos da execução;
- Relatório resumo dos testes, com a descrição resumida das atividades de teste e uma avaliação dos resultados;
- Relatório de incidentes de testes, registrando eventos ocorridos durante os testes que mereçam uma análise posterior;
- Relatório de encaminhamento do item de teste, registrando as correções de cada equipe responsável.

3.4 Teste de Componentes

Teste de componentes e de sistemas baseados em componentes envolve uma série de questões. Segundo Harrold et. al [40], é possível analisar a questão do teste de sistemas baseados em componentes de duas perspectivas: a perspectiva do cliente e a perspectiva do fornecedor.

3.4.1 Perspectiva do Cliente

Os clientes são aqueles que desenvolvem sistemas integrando suas aplicações e componentes desenvolvidos independentemente. Para auxiliá-los, existem diversas iniciativas de se adaptarem técnicas de análise e teste destinadas a programas tradicionais na análise, teste e manutenção de sistemas baseados em componentes. Entretanto, existem algumas questões que dificultam a adaptação de tais técnicas, tais como:

1. O código fonte dos componentes, em geral, não está disponível para os seus clientes. Técnicas e critérios de testes baseados na implementação, tais como critérios baseados em análise de fluxo de dados necessitam do código fonte para derivar os requisitos de teste. Quando o código fonte do componente não está disponível para o cliente, as técnicas de testes tradicionais não podem ser aplicadas no teste de sistemas baseados em componentes ou pelo menos um esquema alternativo deverá ser estabelecido entre as partes interessadas.

2. Em sistemas baseados em componentes, mesmo que o código fonte esteja disponível, os componentes e a aplicação do cliente podem ter sido implementadas em diferentes linguagens de programação. Desse modo, uma ferramenta de análise ou de teste que seja baseada em uma linguagem de implementação específica irá falhar durante os testes.

3. Um componente de software freqüentemente oferece mais funcionalidades do que a aplicação do cliente necessita. Com isso, sem a identificação da parte da funcionalidade que é utilizada pela aplicação, uma ferramenta de teste irá fornecer relatórios imprecisos. Por exemplo, critérios de teste estruturais avaliam o quanto determinado conjunto de teste é adequado em cobrir os requisitos de teste exigidos pelo critério (elementos estruturais do programa). Quando se deseja avaliar a adequação de determinado conjunto de testes em relação a um sistema baseado em componentes, os elementos estruturais que compreendem a parte não utilizada do componente devem ser excluídos na avaliação. Caso contrário, uma ferramenta

de teste irá produzir relatórios indicando baixa cobertura para o conjunto de teste, mesmo se tal conjunto teste exaustivamente a porção do código sendo utilizada [16].

3.4.2 Perspectiva do Fornecedor

O fornecedor implementa e testa o componente de software independentemente da aplicação que fará uso de tal componente. Ao contrário do cliente, o desenvolvedor tem acesso ao código fonte. Desse modo, testar o componente para o desenvolvedor é similar ao teste de unidade/integração tradicional. Entretanto, critérios tradicionais, tais como cobertura de comandos e de desvios condicionais, podem não ser suficientes para o teste de componentes devido à baixa capacidade de detecção de defeitos desses critérios [21].

Corrigir uma falha em um componente depois que esse já está no mercado, em geral, envolve um custo de correção muitas vezes maior do que se a mesma falha descoberta durante os teste de integração de um sistema não baseado em componentes, porque um componente pode estar sendo utilizado em muitas aplicações.

O desenvolvedor do componente deve testar efetivamente os componentes como uma unidade de software independente. Testar adequada e efetivamente componente de software independentemente do contexto de seu uso aumenta a confiança na qualidade do componente e reduz os custos dos testes dos componentes por parte de seus clientes.

Rosenblum [16] descreve uma abordagem para teste de unidade para componentes de software que é dependente do contexto da aplicação e, portanto, mais relevante para o cliente do que para o desenvolvedor. Outra abordagem desenvolvida por Harrold et. al [40], separa a análise e o teste da aplicação do cliente da análise e do teste do componente de software.

3.5 Melhoria da *Testabilidade*

Segundo o padrão IEEE 610.12-1990 (IEEE Standard Glossary of Software Engineering Terminology), tem-se que a *testabilidade* é uma indicação (i) do quanto um sistema ou componente facilita tanto o estabelecimento de critérios de teste quanto a realização de testes para determinar se esses critérios foram satisfeitos; e (ii) os requisitos são expressos de forma a

permitir tanto o estabelecimento de critérios de testes quanto a realização dos testes para determinar se estes critérios foram satisfeitos.

Em outras palavras, a *testabilidade* é uma qualidade do software que se refere à facilidade com a qual critérios de teste podem ser estabelecidos e satisfeitos. Um sistema testável requer menor esforço para a realização dos testes e melhora a eficácia dos mesmos, isto é, o seu potencial para encontrar falha [17]. De acordo com Binder [53], vários fatores são considerados para a *testabilidade* de um componente, tais como:

- A *representação* que se refere à qualidade da documentação do sistema. A facilidade de mapeamento entre a documentação do sistema e sua implementação assim como uma atualização regular desta documentação são fatores importantes para que os casos de teste tenham sempre os seus objetivos bem definidos refletindo os requisitos reais do sistema.
- A *implementação* no sentido de que localização das falhas é mais fácil em sistemas altamente coesos e com pouco acoplamento, pois facilitam os testes aumentando a *testabilidade*. Por outro lado, situações difíceis de serem exercitadas, tais como, otimizações ligadas ao desempenho, concorrência e tratamento de exceções reduzem a *testabilidade* do sistema.
- A *capacidade de teste embutido (Built-in-test)* que se refere à melhoria da *testabilidade.* Por exemplo, inclusão de métodos *set/reset* para a definição de um estado para o sistema e inclusão de assertivas que implementem a abordagem *Design-by-Contract* que representa o relacionamento entre uma classe e seus clientes como um acordo formal, expressando os direitos e obrigações de ambas as partes [38].
- O *conjunto de testes*, os quais devem ser bem documentados para que os custos com a manutenção dos testes possam ser reduzidos.
- A *ferramenta de testes* que permite a realização de um grande volume de testes a um custo reduzido.
- O *processo de desenvolvimento* que contribui com a qualidade do sistema e de sua documentação.

4 Capítulo

Técnicas de Testes de Integração

De acordo com Binder [54], teste de integração é a busca por falhas nos componentes que produzem falhas entre os componentes. Essa integração pode ser realizada de forma incremental ou não-incremental.

A *integração não-incremental* é executada através da abordagem *big bang* [55] *(Seção 4.1)* na qual combinam-se todos os componentes e testa-se o programa completo como um todo, não costuma ser eficaz. Dada a amplitude do teste do sistema como um todo, torna-se difícil isolar uma falha e, quando essas são corrigidas, surgem novas falhas e o processo continua de um modo aparentemente infindável [55].

A *integração incremental* é considerada mais eficiente, pois se implementa e testa-se o programa em pequenos segmentos, onde as falhas são mais fáceis de serem isoladas e corrigidas e as interfaces têm maior probabilidade de serem testadas completamente [55]. A integração incremental pode ser realizada seguindo-se através da arquitetura do sistema pela ordem ascendente ou descendente dos componentes.

Segundo Pfleeger [58], a complexidade que pode geralmente residir no componente, é freqüentemente transferida para a interface entre os componentes. Assim, o teste de unidade é menos difícil, ao passo que o teste de integração deve ser mais extensivo e fundamental.

Os testes de unidade e de integração devem ser combinados para que se possa testar efetivamente uma implementação.

Os planos de testes são normalmente baseados na dependência de implementação entre componentes, ou seja, são baseados conforme a necessidade de um componente em relação a outro; sendo assim, a análise da arquitetura do componente é necessária. Caso a arquitetura do sistema não seja especificada, então a definição do projeto de testes de integração deverá aguardar pela implementação de algum ou, no pior caso, de todos os componentes.

O processo de teste de integração freqüentemente revela falhas, omissões e ambigüidades nos requisitos e na arquitetura; portanto, definir a arquitetura do projeto o mais cedo possível é essencial.

Os testes de integração geralmente são projetados de acordo com o estilo arquitetural adotado pelo sistema. Por exemplo, se o sistema possui uma arquitetura em camadas, a técnica de teste de integração indicada é a de integração por camadas.

Integrar vários ou todos os componentes ao mesmo tempo é normalmente problemático. A remoção de falhas, nesse caso, é complicada, pois uma falha pode estar em qualquer interface.

Assim, a integração incremental seria a técnica mais eficaz, pois um componente é adicionado e, em seguida, sua interoperabilidade é testada. As interfaces são sistematicamente testadas e as falhas de interfaces são reveladas antes de serem propagadas. Desta forma, a remoção de falhas torna-se mais eficiente, pois essas são mais comuns em componentes adicionados ou modificados recentemente.

Embora razoavelmente simples, o desenvolvimento incremental requer disciplina e comprometimento. A seqüência dos componentes deve ser identificada usando cuidadosamente a análise de dependência dos componentes *(Seção 4.1)*. Em seguida os testes devem ser planejados e gerenciados seguindo esta seqüência.

Em geral, técnicas de integração fazem uso de *drivers* e/ou *stubs*. Os *stubs* são pedaços de software que devem ser construídos para simular ou substituir partes de software que ainda não foram desenvolvidas ou ainda não foram testadas, e que estejam subordinadas ao componente a ser testado e, portanto, são necessários para testar os componentes dependentes deles. Um *stub* é a implementação parcial de um componente [54]. É um componente usado como fachada para simular o comportamento de um componente real [4]. O teste de um componente A, que chama um componente B, mas B ainda não foi testado ou ainda não está pronto, 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. Um *stub realístico* é escrito para simular o comportamento do componente B em qualquer caminho de teste seja em relação ao componente A ou em relação a qualquer outro componente do sistema [69]. Um *driver* nada mais é do que um “programa principal” que aceita dados de

casos de teste, encaminha tais dados para o módulo a ser testado e retorna os dados relevantes [55].

Drivers e *stubs* representam *overhead,* ou seja, ambos são softwares que devem ser escritos mas que não são entregues com o produto final. *Drivers* e *stubs* mantidos simples, acarretam um *overhead* real relativamente baixo.

Infelizmente testar componentes em nível de unidade geralmente é muito custoso devido à dificuldade e ao tempo gasto na construção de *stubs* adequados para exercitar as unidades de forma satisfatória. Nesses casos, testes completos podem ser postergados até a etapa de teste de integração quando *drivers* e *stubs* são usados [55].

Uma vez que esse trabalho irá propor um método para testes de integração em sistemas baseado em componentes e a análise de dependência é de grande importância para os testes de integração, a *Seção 4.1* apresenta alguns conceitos relacionados à análise de dependência, em seguida as *Seções 4.2* a *4.6* apresentam resumidamente algumas técnicas de integração *incremental* e não-*incremental*.

4.1 Análise de Dependência

Análise de dependência envolve a identificação de elementos dependentes de um sistema. Refere-se a uma *técnica de redução* uma vez que um dado relacionamento entre alguns elementos forma um subconjunto de um sistema.

A análise de dependência tem sido largamente estudada e usada como base para otimização de programas, geração de casos de teste e depuração *(debbuging)*. Dependências podem ser identificadas baseando-se em informações sintáticas *(o que)* disponível nas especificações formais dos sistemas. Este tipo de análise geralmente ignora o estado da informação; sendo assim, conhecimentos semânticos *(como)* da linguagem podem ser incorporados a essa análise aperfeiçoando a precisão dos resultados [25].

Análise de dependência é uma técnica de análise para identificar e determinar vários tipos de dependências de código de programas. Análise de dependência quando aplicada no código do programa é baseada no relacionamento entre as instruções e as variáveis no programa. Atualmente estão sendo desenvolvidas técnicas para identificar e explorar relacionamentos de

dependência em nível de arquitetura. Relacionamentos de dependência em nível de arquitetura surgem da conexão entre os componentes e das restrições nas interações entre eles [69].

Segundo Binder [54], 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.

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 ascendente (*bottom up) (Seção 4.3).* As dependências explícitas entre componentes correspondem a interfaces que o componente necessita para sua correta operação, ou seja, as interfaces requeridas [54]. Dependências ocorrem onde um componente usa os serviços de outro componente.

4.2 Integração Big Bang

A técnica de integração *big bang* é normalmente usada devido às pressões do dia a dia e os testes são aplicados para demonstrar uma operabilidade mínima do sistema [54].

Essa técnica não necessita de nenhum *driver* e/ou *stub* sendo indicada nas seguintes situações: (i) para um sistema estável, testado detalhadamente em um outro momento e com poucas alterações, como por exemplo, manutenção e/ou inclusão de novas funcionalidades; (ii) para um sistema pequeno e bem estruturado cujos componentes passaram adequadamente pelos testes de unidade e os componentes são testados toda vez; (iii) para um sistema *monolítico* geralmente desenvolvido em linguagem de programação convencional (Basic, Fortran, Cobol); (iv) em implementações orientadas a objetos baseados em um projeto pobre ou sem nenhum projeto, não sendo possível testá-lo parcialmente.

Usar a técnica de integração *big bang* para um sistema que não se encaixe em nenhuma das situações citadas anteriormente poderá significar ganho de mais problemas do que propriamente resolução dos mesmos.

De acordo com Binder [54], quando a integração caminha sem dificuldades e o esforço de testes é pequeno a integração *big bang* torna-se a escolha correta. Porém, muitas falhas ocorrem na interface dos componentes do sistema e só são descobertas quando este é colocado em execução, frustrando o teste de integração *big bang.*

Uma vez que um conjunto de testes mínimos é aplicado, e por não se tratar de uma técnica de integração incremental, depurar e diagnosticar a falha, é difícil porque entrada do sistema é construída e o conjunto de testes é aplicado de uma só vez. Assim, todos os componentes são igualmente suspeitos de causar ou propagar a falha e não há quase nenhuma informação sobre a localização da falha.

4.3 Integração Ascendente *(Bottom Up)*

O teste de integração ascendente inicia a construção da arquitetura e os testes de integração pelos componentes localizados nos níveis mais baixos da arquitetura.

Os componentes são integrados de baixo para cima *(bottom up)* e o processamento exigido para os componentes do nível inferior ao nível sob teste, está sempre disponível. Desta forma, mesmo que haja dependência entre os componentes sob teste e os componentes do nível inferior ao nível sob teste, a necessidade de *stubs* é eliminada.

Na técnica de integração *bottom up* aplica-se análise de dependência nos componentes sob teste e primeiramente são testados os componentes com menor número de dependências. Para cada rodada de testes, ou seja, para cada nível sob teste haverá um nível superior composto de *drivers.*

A integração *bottom up* executa movimentando-se das *folhas* para a *raiz* na árvore de dependência. Uma árvore de dependência composta por *n* níveis possui *n* estágios de integração.

Em um primeiro estágio, codificam-se os componentes de mais baixo nível (*componentes de nível folha* na árvore de dependência) e os *drivers* para coordenar a entrada e a saída dos casos de

testes para esses componentes. O nível sob teste é testado por estes *drivers* como mostra a *Figura 4.1*.

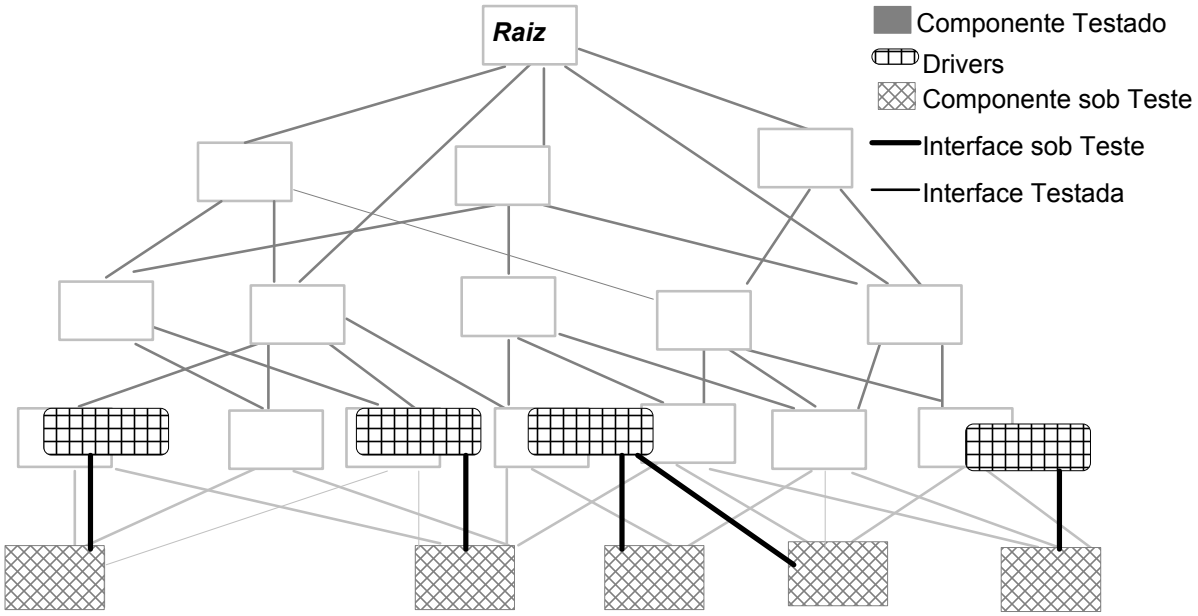


Figura 4.1: Integração *bottom up*, configuração do primeiro estágio.

Em seguida os *drivers* são substituídos pelos componentes codificados. Esses componentes possuem interfaces, ou seja, enviam mensagens ou passam essas mensagens como argumento para os componentes que já foram testados no estágio anterior.

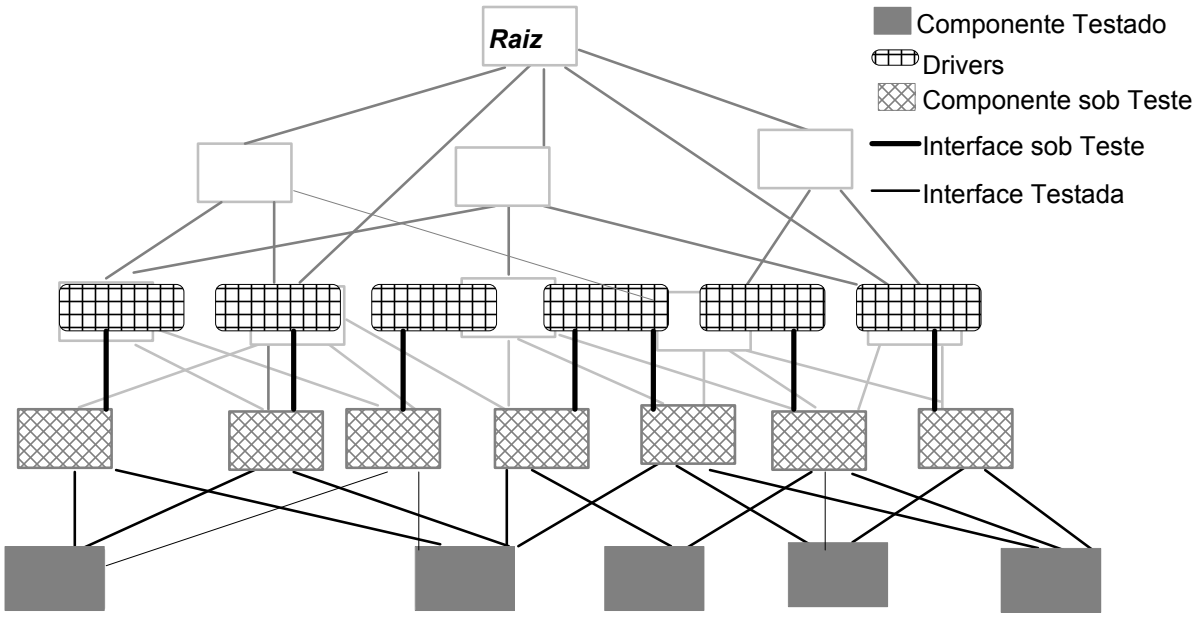


Figura 4.2: Integração *bottom up* - configuração do segundo estágio.

Prossegue-se testando sucessivamente todos os componentes dos níveis de baixo para cima até que se atinja uma configuração final, onde um *driver* é codificado para testar o nível mais alto na árvore de dependência, ou seja, a raiz *(Figura 4.3).*

Na medida em que a integração desloca-se para cima, a necessidade de *drivers* de teste diminui e, assim que os testes para um determinado nível são finalizados, os *drivers* no nível superior ao nível sob teste são substituídos pelos respectivos componentes e uma nova rodada de testes é iniciada.

Um *driver* pode ser reusado para testar diferentes componentes em diferentes níveis, portanto, deve-se evitar que um único *driver* mantenha um conjunto de testes para testar diferentes componentes num mesmo nível. Visando facilitar o reuso é recomendável que um *driver* seja criado para testar individualmente cada componente.

No desenvolvimento incremental de um sistema baseado em componentes, a integração *bottom up* é freqüentemente utilizada como apoio aos testes de unidade. Testa-se cada componente logo após o término de sua codificação e então o componente é integrado aos componentes já testados.

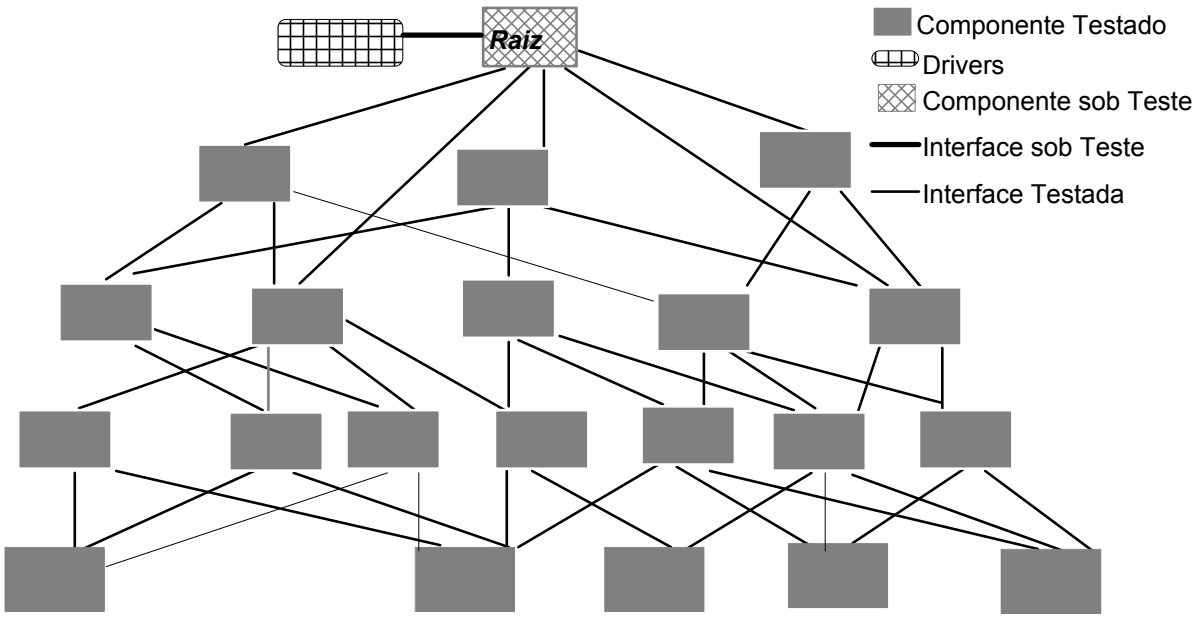


Figura 4.3: Integração *bottom up,* configuração final.

4.4 Integração Descendente *(Top Down)*

Um sistema desenvolvido segundo uma abordagem incremental permite modelar a estrutura de controle do sistema como uma árvore de dependência na qual os componentes do topo desta árvore mantêm responsabilidades de controle [54].

A integração *top down* é uma abordagem incremental à construção da estrutura de programa. Os componentes são integrados movimentando-se de cima para baixo através da hierarquia de controle da aplicação, iniciando-se do componente de controle principal. Os componentes subordinados ao componente responsável pelo controle são incorporados à estrutura de duas formas: em profundidade (*depth-first)* ou em largura *(breadth-first)*.

Em relação à *Figura 4.4*, *depth-first* integraria todos os componentes num caminho de controle principal da arquitetura. A escolha de um caminho de controle principal é bastante arbitrária e depende das características específicas da aplicação. Por exemplo, escolhendo-se o caminho à esquerda, os componentes C1, C2 e C5 seriam integrados primeiro. Em seguida, C8 ou (se necessário para o adequado funcionamento de C2) C6 seria integrado. Posteriormente os caminhos de controle centrais à direita seriam construídos.

A integração *breadth-first* incorpora todos os componentes diretamente subordinados a cada nível, movimentando-se horizontalmente ao longo da arquitetura. Examinando-se novamente a *Figura 4.4*, os componentes C2, C3 e C4 seriam integrados primeiro. Prossegue-se pelo nível de controle seguinte, então, os componentes C5, C6 e C7 seriam integrados e por último o componente C8 seria integrado.

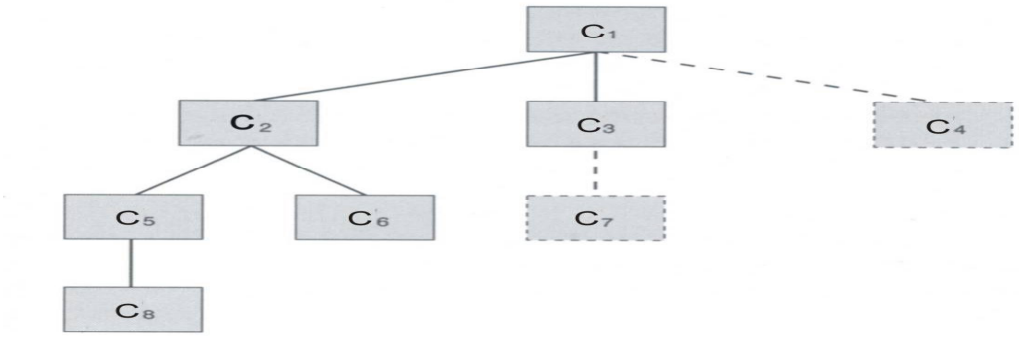


Figura 4.4: Integração *top down* - representação *depth-first* ou *breadth-first*.

Após modelar a estrutura de controle como uma árvore de dependência, em um primeiro estágio implementa-se o componente de nível mais alto da estrutura de controle, criam-se os *stubs* referentes às interfaces requeridas do componente e um *driver* para coordenar a entrada e a saída dos casos de testes para cada nível sob teste *(Figura 4.5).*

Para cada nível testa-se horizontalmente *(breadth-first)* cada componente. Os *stubs* são substituídos pelo respectivo componente implementado no nível que será testado e os novos *stubs* necessários para os testes do próximo nível (de cima para baixo) são criados *(Figura 4.6).* Prossegue-se sucessivamente dessa forma até que todas as interfaces requeridas, ou seja, todos os componentes dos níveis inferiores ao nível sob teste (de cima para baixo), no sistema tenham sido implementados e exercitados. O sistema estará totalmente integrado e o teste concluído, quando o último *stub* no nível mais baixo da estrutura de controle for substituído pela implementação do componente real *(Figura 4.7).*

A técnica de integração *top down* requer um único *driver* no nível mais alto da estrutura de controle. Um *stub* é necessário para cada componente do nível inferior ao nível sob teste na estrutura de controle.

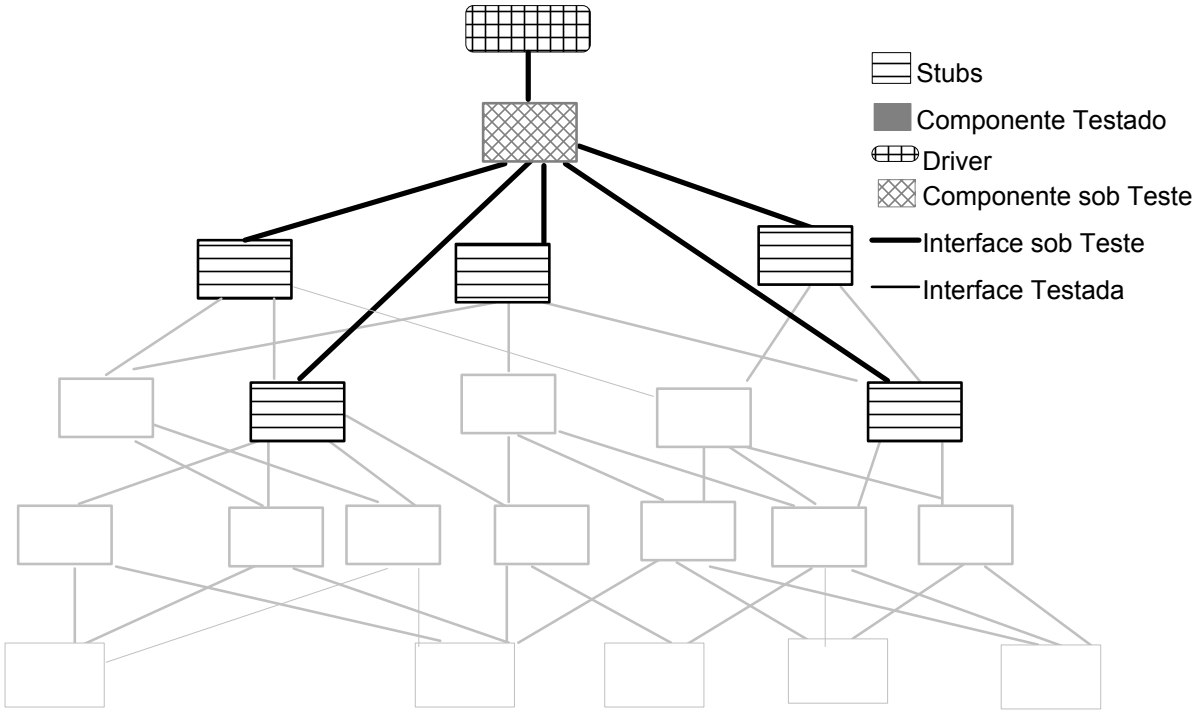


Figura 4.5: Integração *top down*, configuração do primeiro estágio.

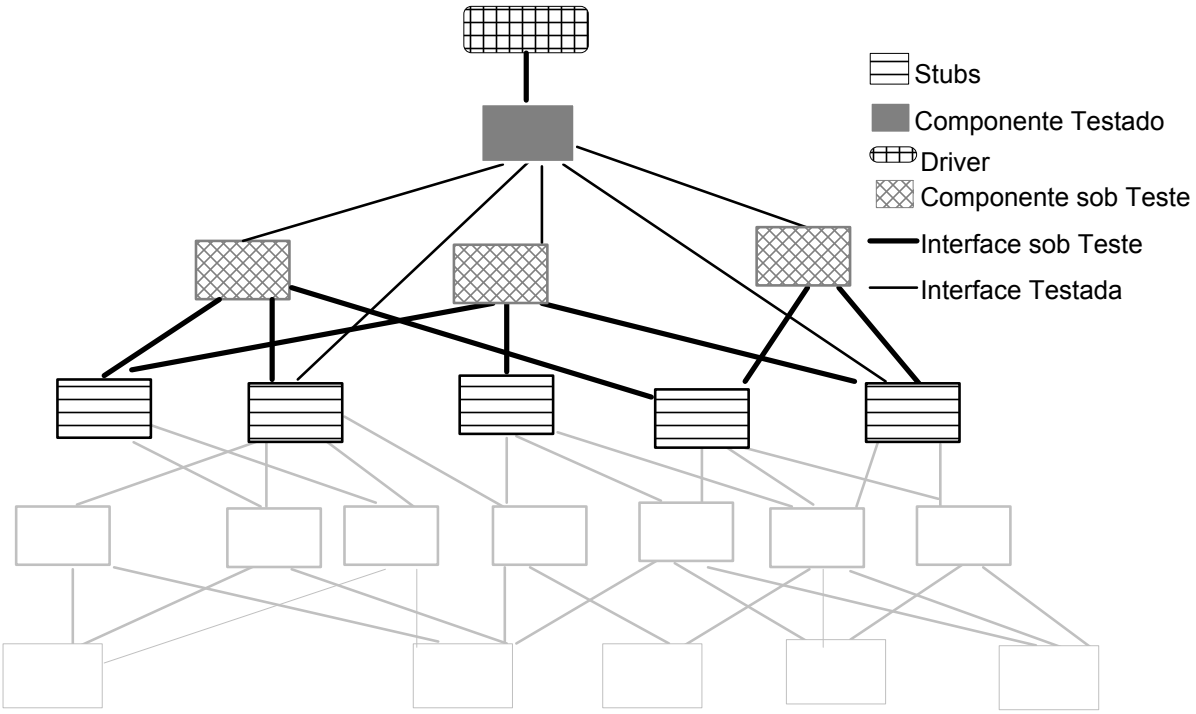


Figura 4.6: Integração *top down,* configuração do segundo estágio.

Convém ressaltar que a técnica de integração *top down* embora pareça relativamente fácil, na prática podem surgir muitos problemas. O mais comum desses problemas ocorre quando um componente requer uma interface de um componente que não pertença ao nível imediatamente inferior ao nível sob teste, desta forma, um processamento em níveis baixos na estrutura é solicitado para testar adequadamente níveis superiores.

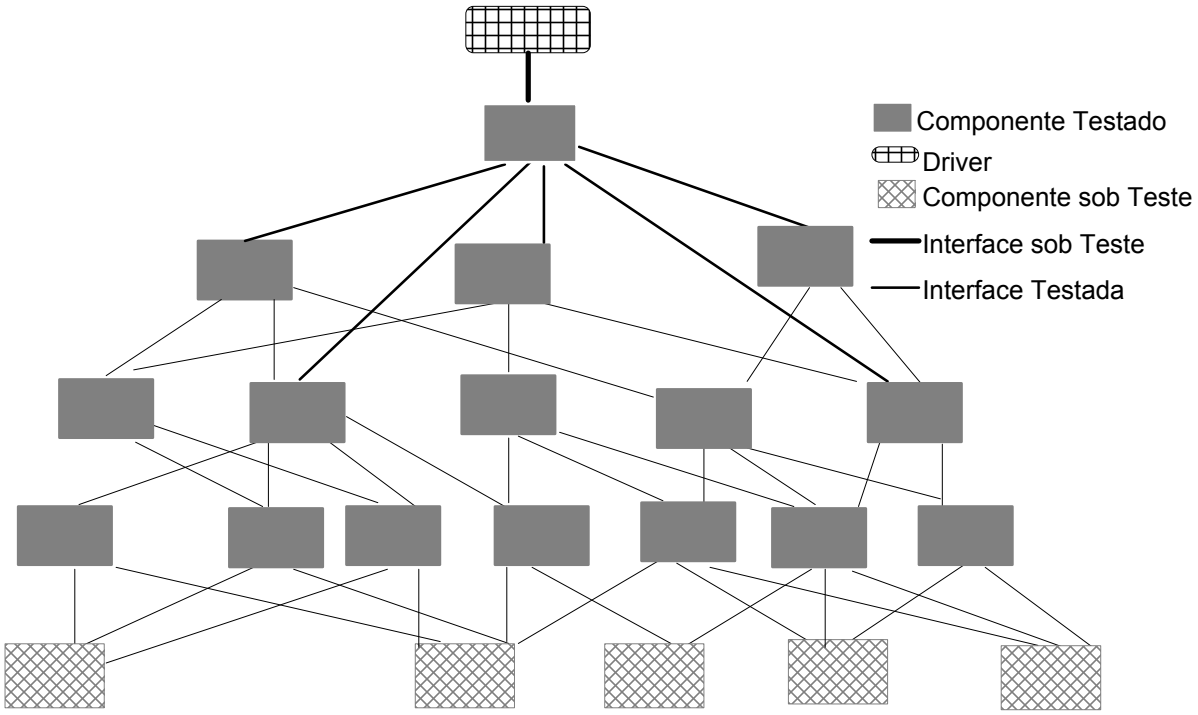


Figura 4.7: *Integração top down, configuração final.*

O analista, neste caso, pode retardar os testes até que os componentes reais sejam implementados, o que possibilita certa perda de controle sobre o que realmente está sendo testado em determinado nível, pois, ou testa-se a incorporação de um novo componente à estrutura, ou exercitam-se os testes específicos do nível sob teste. Isso pode levar a dificuldade na determinação das causas de falha.

Uma outra opção seria desenvolver *stubs* que executem funções limitadas que simulem o componente real. Isso pode levar a um significativo *overhead*, uma vez que os *stubs* tornam-se cada vez mais complexos.

Dependendo da dificuldade em aplicar esta técnica de integração para o sistema em questão o analista pode ainda optar por aplicar a técnica de integração de baixo para cima *(bottom up).*

Quando se faz uso da técnica de integração *top down* para alcançar resultado desejado, geralmente, (i) um grande número de *stubs* é codificado e (ii) alterar um componente em qualquer nível abaixo do nível sob teste pode acarretar alterações em componentes já testados anteriormente, ocasionando a necessidade de uma revisão na configuração de *driver* e *stubs.* Os

testes deverão ser executados novamente, elevando o custo para a manutenção e desenvolvimento do sistema sob teste.

Por outro lado, os testes de integração iniciam-se mais cedo logo que os componentes do primeiro nível (de cima para baixo) são codificados.

4.5 Integração por Colaboração

Na técnica de integração por colaboração seleciona-se uma ordem de integração conforme uma colaboração selecionada e suas dependências *(Figura 4.8).* Exercitam-se as interfaces entre as partes participantes da colaboração exercitando-se uma colaboração de cada vez e a integração é finalizada quando todos os componentes e interfaces forem exercitados.

Um sistema é composto por muitas colaborações, uma colaboração pode conter colaborações e um componente pode pertencer a mais de uma colaboração. Isto significa que testar todos os componentes não necessariamente significa testar todas as combinações de componentes que geram uma colaboração, desta forma algumas falhas de interface presentes em colaborações não exercitadas podem ter sido mascaradas.

A técnica de integração por colaboração requer um único *driver* conectado à interface da colaboração sob teste. A criação de *stubs* é necessária para substituir os componentes que não pertençam à colaboração sob teste e que possua uma interface provida com os componentes dentro da colaboração sob teste.

Na medida em que se exercitam os testes novas colaborações vão sendo configuradas. Essas configurações devem ser revistas e executadas novamente a cada alteração no sistema sob teste.

Dada uma colaboração, não é possível que um componente participante desta colaboração seja exercitado separadamente. Sendo assim, cada colaboração pode ser vista como um cenário *big bang* minimizado porque são testados conjuntos de componentes por colaboração de uma só vez.

Uma vez que nessa técnica de integração por colaboração o objetivo é testar uma funcionalidade fim a fim e um diagrama de colaboração representa uma chamada para um caminho ou um conjunto de chamadas para caminhos que são exercitados para uma

funcionalidade sob teste, todos os caminhos não são necessariamente modelados, neste caso, a colaboração especificada pode estar incompleta.

Por outro lado, na técnica de integração por colaboração é possível testar rapidamente todas as interfaces pelo menos uma vez com a aplicação de poucos testes. Além disso, o conjunto de teste de integração por colaboração pode, em geral, ser reusado ou expandido para o teste do sistema.

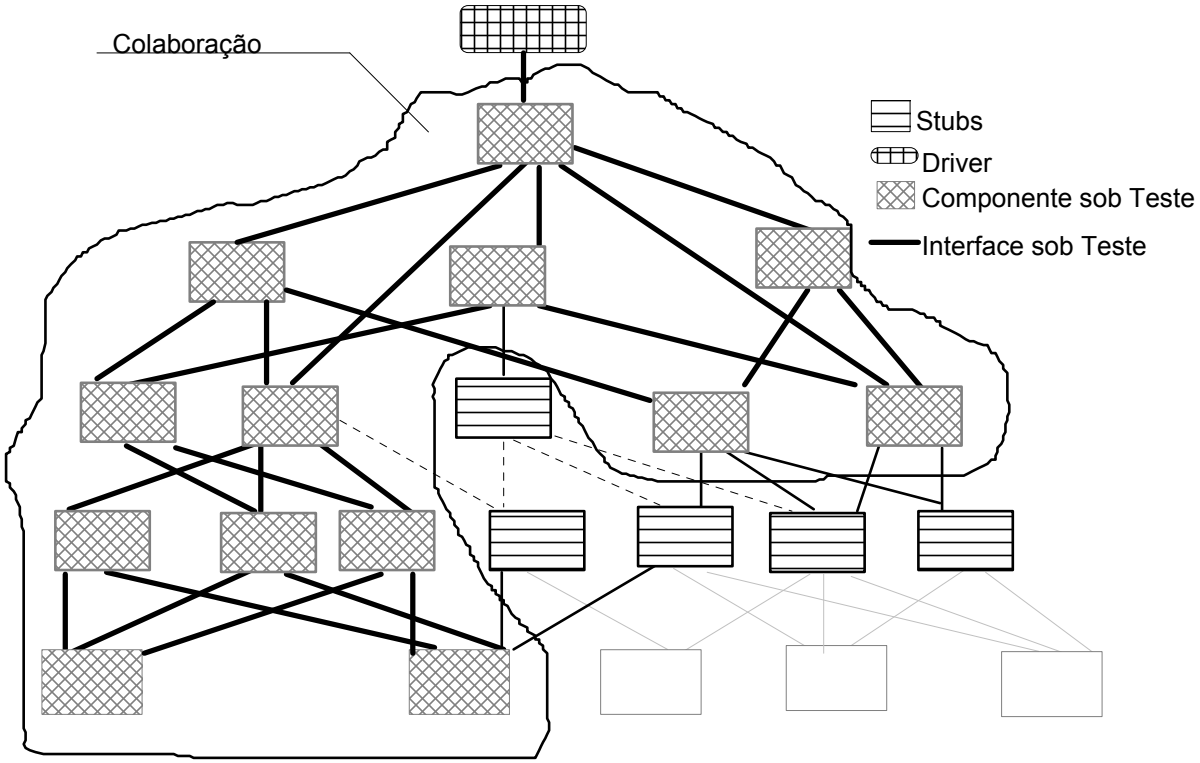


Figura 4.8: Exemplo de configuração para a técnica de integração por colaboração.

5 Capítulo

Trabalhos Relacionados

Os estudos referentes à ordem de integração, em geral, baseiam-se no diagrama de classes de um sistema orientado a objetos. Embora o método proposto (Capítulo 6) vise uma ordem de integração em um sistema baseado em componentes, vale lembrar que um componente pode ser entendido como uma unidade de teste básica, podendo corresponder a uma classe.

Alguns trabalhos relacionados foram estudados na busca de uma solução *(método)* para determinar uma ordem para a escolha dos componentes durante os testes de integração. O objetivo de todas as propostas estudadas é minimizar o número de *stubs* de teste a serem produzidos e conseqüentemente, minimizar o esforço e o custo durante os testes de integração.

Na presença de ciclos de execução surge a necessidade de um diagrama de classes que permita identificar quais relacionamentos são *quebráveis* e quais são *não quebráveis.* Para isso Kung et. al [13], Tai e Daniels [33] propõem e usam um diagrama extraído do diagrama de classes do sistema para representar os relacionamentos de herança *(I)*, agregação *(Ag)* e associação *(As).* Trata-se do diagrama normalmente chamado de ORD *(Object Relation Diagram).*

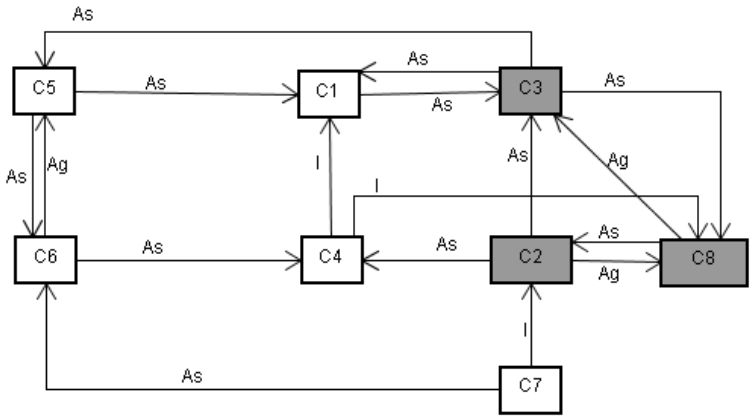
A *Figura 5.1* apresenta um *ORD* que contém exemplos de ciclos de dependência. Por exemplo, obtém-se um ciclo de dependência no conjunto A= {C2, C3, C8}.

Os trabalhos estudados baseiam-se no princípio da *quebra de ciclos* de algumas dependências cíclicas visando obter uma dependência acíclica entre os componentes. *Quebrar um ciclo*, no diagrama da *Figura 5.1*, consiste em remover uma aresta que conecte dois desses componentes do conjunto A; por exemplo, removendo-se a aresta que conecta o componente C8 ao C2 ocasiona a quebra do um ciclo de dependência representado no conjunto A.

No contexto dos trabalhos estudados, quando ocorre a remoção de uma aresta há a necessidade da substituição do componente *alvo*, ou seja, o componente para o qual a aresta removida estava direcionada, por um *stub*. Por exemplo, na *Figura 5.1,* ao se remover a aresta

{C8,C2} o componente C2 é substituído por um *stub* para os testes de integração que envolva o componente C8.

Quando um conjunto de componentes está incluído em um ciclo de dependências diz-se que eles pertencem a um cluster (componente fortemente conectado - *SCC-Strongly Connected Component).*



***As* –Associação *Ag* – Agregação *I* - Herança *Cx* - Classes**

Figura 5.1: Exemplo de ORD *(Object Relation Diagram).*

Uma importante questão é determinar qual critério de avaliação deve ser usado para determinar a escolha de qual aresta será removida para quebrar um ciclo de dependência.

Portanto, quando uma classe servidora é usada por várias classes clientes, serão necessários tantos *stubs* quantas classes clientes existirem, para minimizar o risco de uma falha não detectada. Sendo assim, o mais provável é que o número de *stubs* seja proporcional ao número de clientes das classes que serão substituídas por *stubs* [36].

As *Seções* a seguir apresentam algumas soluções estudadas na busca por uma ordem de integração.

5.1 Kung et. al

Kung et. al [13] estão entre os primeiros pesquisadores a apresentar uma solução para o problema dos ciclos de dependência e a discutir este tipo de problema nos testes de integração para sistemas orientados a objetos.

Os estudos mostram que quando não há ciclos entre as classes define-se a ordem de integração como se fosse executar uma ordenação topológica de classes baseando-se no grafo de dependência de classes. Quando há ciclos Kung et. al [13] argumentam que todo ciclo em um diagrama de classes contém ao menos uma associação.

A técnica proposta por Kung et. al [13] consiste em identificar, no diagrama *ORD (Figura 5.1)*, os *SCC's* e remover somente as arestas que representem associações *(As)* até não existirem mais ciclos. Caso haja mais de uma associação candidata, a escolha é feita de forma aleatória.

Relacionamentos que representam herança e agregação não são removidos nessa técnica por representarem, além do acoplamento de controle, acoplamento de dados e dependência de código. *Quebrar* um relacionamento de herança, por exemplo, pode ocasionar o desenvolvimento de um *stub* que implemente a herança.

A ordem definida para os testes de integração é dada ao se efetuar uma ordenação topológica no grafo de dependência após a remoção dos ciclos dos relacionamentos de associações; além disso, o problema relacionado à minimização de *stubs* não está explícito nessa técnica.

5.2 Tai e Daniels

O trabalho de Tai e Daniels [33] toma como base o trabalho de Kung et. al [13] permitindo apenas a eliminação de arestas que representem associações. Os autores propõem uma estratégia para a escolha de quais arestas do *ORD (Figura 5.1)* devem ser eliminadas para a quebra dos ciclos de dependência, ao invés de escolher aleatoriamente uma aresta.

A estratégia proposta em um primeiro passo ignora todas as associações e associa cada classe no diagrama de classes baseada em dependência de herança e agregação a um número de nível principal. Classes independentes são as classes pertencentes ao nível principal 1.

Visto que, neste passo só os relacionamentos de herança e agregação são considerados e estes não formam ciclos, aplica-se a ordenação topológica no diagrama. A *Figura 5.2,* derivada da *Figura 5.1* representa o diagrama *ORD* para primeiro passo dessa estratégia.

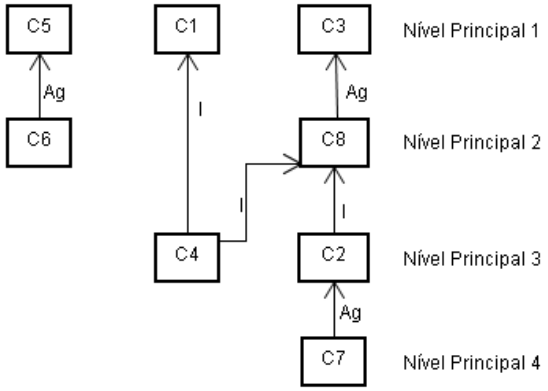


Figura 5.2: *ORD (Object Relation Diagram)* - Associações removidas. Representação dos números de nível principal.

Posteriormente, dentro de cada nível principal adicionam-se as arestas que representem apenas dependências de associação, definindo para cada classe um número de nível secundário. Esses números de nível principal e secundário são então usados para definir uma ordem de integração. Nesse passo há a possibilidade de surgirem ciclos de dependência que devem ser *quebrados* em uma determinada ordem possibilitando uma ordenação topológica no diagrama.

Para *quebrar* os ciclos define-se um peso para cada associação em cada nível principal, calculado a partir do número de arestas de chegada no vértice de origem somado ao número de arestas de saída do vértice destino. As arestas com peso maior são então selecionadas para serem removidas e conseqüentemente quebrar os ciclos.

A *Figura 5.3* representa as classes envolvidas em um ciclo no nível principal de número 1 e suas associações. O peso para as diferentes associações nesse ciclo seria: P(C3,C1)=2, P(C5,C1)=2, P(C1,C3)=4, P(C3,C5)=2. Portanto o peso maior pertence à associação entre as classes C1, C3, então, essa será a associação removida para *quebrar* o ciclo neste nível principal de número 1 e como os valores para os pesos restantes são iguais atribui-se para C1, C5, C3 os valores 1, 2, 3 respectivamente como sendo seus números de nível secundário.

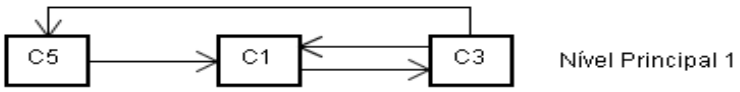


Figura 5.3: Representação das associações do nível principal de número 1.

A *Figura 5.4* possui além da representação dos números de nível principal, a representação das associações a serem removidas e a representação dos números de nível secundário; por exemplo, para a classe C5, C5-2 significa que o número de nível secundário da classe C5 é 2.

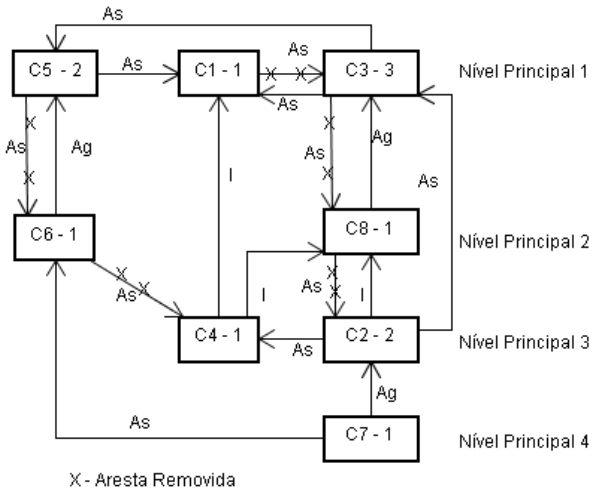


Figura 5.4: *ORD (Object Relation Diagram)* - Arestas Removidas. Representação dos números de nível principal e secundário.

Nível Principa	**Nível Secundári**	**Testa Classe(s) envolvida**	**Nível Principa**	**Nível Secundário**	**Testa Classe(s)**
1	1	C1 com o *stub* (C3,C1)	**3**	1	C4
	2	C5 com o *stub* (C6,C5)		2	C2
	3	C3 com o *stub* (C8,C3)	**4**	1	C7
2	1	C6 com o *stub* (C4,C6)			
	2	C8 com o *stub* (C2,C8)			

Figura 5.5: Representação dos *stubs* necessários em cada número de nível principal e menor.

Quebradas as associações obtém-se uma ordem final no diagrama e, como para cada aresta removida teremos uma classe alvo substituída por um *stub,* nesse exemplo, serão necessários cinco *stubs* específicos, ou seja, um para cada classe. A *Figura 5.5* representa a ordem dos testes de integração e o número de *stubs* necessários em cada nível principal e secundário.

A representação na *Figura 5.5 stub* (C3, C1), por exemplo, representa qualquer *stub* de C3 criado para os testes de C1.

5.3 Le Traon et. al

O trabalho de Le Traon et. al [69] identifica os *SCC's* usando uma adaptação do algoritmo de Tarjan [56]. Os *SCC's* podem ser triviais, quando são compostos de apenas um vértice, e não triviais quando são compostos por mais de um vértice.

As arestas candidatas à quebra de um ciclo são chamadas de ramo de dependência *(frond dependency)*, ou seja, uma aresta que parte de uma classe (origem) em direção a uma outra classe (destino) da qual a classe origem dependa direta ou indiretamente.

A proposta de Le Traon et. al [69] quebra os ciclos removendo as dependências que chegam da classe com o maior peso, no *SCC* considerado. O peso nessa estratégia é definido de maneira diferente de Tai e Daniels [33]: ele é a soma dos ramos de dependência de chegada e de saída para uma dada classe, dentro do *SCC* considerado. Em resumo a notação de peso é considerada especificamente nas classes identificadas como sendo ramos de dependência (que capturam alguns dos ciclos no qual a classe está envolvida). Para cada *SCC* não trivial, o procedimento acima é então chamado recursivamente.

O peso é calculado com base na identificação das arestas denominadas ramos de dependências e esta identificação depende da construção da árvore de busca em profundidade. O peso depende, então, de qual vértice deu início à busca.

Como exemplo de um primeiro passo dessa proposta de Le Traon et. al [69] a *Figura 5.6,* derivada da *Figura 5.1* representa um *ORD* no qual o vértice C7 foi o escolhido para ser o vértice inicial. Obtém-se o *SCC* composto pelas classes C5, C1, C3, C6, C4, C2 e C8. A ordem parcial

para os testes de integração seria: testa-se o *SCC* {C5, C1, C3, C6, C4, C2, C8} em seguida testa-se a classe C7 usando as classes C6 e C2.

Na *Figura 5.6* o valor do peso para cada vértice no *SCC* seria P(C5)=1, P(C1)=1, P(C3)=2, P(C6)=1, P(C4)=1, P(C2)=2 e P(C8)=2.

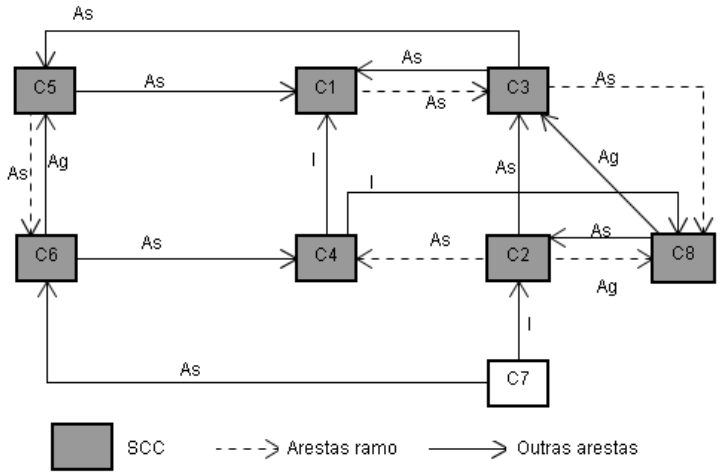


Figura 5.6: Estratégia aplicada inicialmente ao vértice C7.

Obtêm-se três vértices com o mesmo peso P=2. Num segundo passo escolhe-se o vértice C2, ou melhor, a associação C2, C8 para quebrar o ciclo, aplica-se o algoritmo de Tarjan [56] novamente e identifica-se um novo SCC envolvendo as classes C1, C3, C4, C5, C6, C8. Uma ordem parcial para os testes de integração seria:

1. *SCC* {C5, C1, C3, C6, C4, C2, C8} é testado;

    1.1 *SCC* {C5, C1, C3, C6, C4, C8} é testado usando o *stub* (C2, C8);

    1.2 C2 é testada usando as classes C4, C3 e C8

2. C7 é testada usando as classes C6 e C2.

Segue-se aplicando recursivamente o algoritmo de Tarjan [56] e para cada *SCC* seleciona-se a aresta a ser removida baseando-se em seu peso até que não existam mais ciclos. A *Figura 5.7* mostra os *SCC's* e as correspondentes arestas removidas, a ordem final para os testes de integração bem como os *stubs* necessários.

O resultado da aplicação da estratégia de Le Traon et. al [69] é não determinístico e, portanto, existem várias possibilidades de ordenação das classes. Essa estratégia apresenta dois pontos bastante diferentes de Kung et. al [13] e de Tai e Daniels [33]:

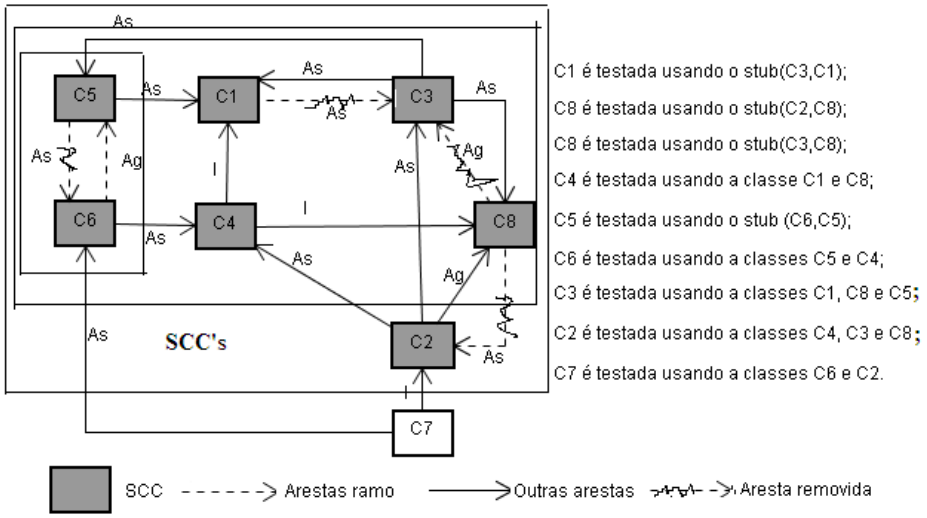


Figura 5.7: Resultado Final para a estratégia aplicada inicialmente ao vértice C7.

1. As arestas que representam herança e agregação podem ser removidas para a quebra dos ciclos de dependência;
2. O algoritmo minimiza o número de *stubs* realísticos *(Seção 4)* que devem ser implementados para simular todos os serviços que a classe original implementa, não se importando com qual cliente está sendo testado.

5.4 Briand et. al

Briand et. al [36] apresentam uma estratégia que utiliza algumas características das propostas de Le Traon et. al [69] e Tai e Daniels [33]:

- Usa o algoritmo de Tarjan [56] recursivamente para identificar os *SCC's*;
- Associa pesos às arestas que representam dependências de associação, como uma forma de estimar o número de ciclos no qual a associação está envolvida num *SCC*.

A proposta de Briand et. al [36] é determinística, pois qualquer alternativa de escolha gera o mesmo número de *stubs,* e minimiza o número de *stubs* específicos que são implementados para fornecer somente os serviços que cada classe cliente necessita individualmente, ao contrário de Le Traon et. al [69].

Essa proposta de Briand et. al [36] conduz à remoção de uma e apenas uma aresta de associação em cada *SCC* enquanto a estratégia de Le Traon et. al [69] conduz à remoção de todas as dependências que chegam na classe selecionada no *SCC.*

A proposta inicia-se com a aplicação recursiva do algoritmo de Tarjan [56] para identificar os *SCC's* não triviais. A cada passo dentro de cada *SCC* não trivial calcula-se um peso para cada dependência de associação usando uma versão modificada da definição de Tai e Daniels [33]. A aresta que deve ser removida é aquela que representa o maior peso, obtido através da multiplicação das arestas de entrada do vértice de origem pelas arestas de saída do vértice destino de cada aresta que represente uma associação no diagrama *ORD.* Esses passos devem ser seguidos até não existirem mais *SCC's* não triviais.

A *Figura 5.8* mostra um exemplo da aplicação em que se pode observar as arestas que foram removidas para a quebra dos ciclos, o nível de cada vértice, o nível para cada *SCC* bem como os *stubs* necessários. Nesse exemplo, após a aplicação inicial do algoritmo de Tarjan [56] são encontrados os *SCC's* não triviais {C1, C3, C5, C6, C4, C8, C2} e o *SCC* trivial {C7}. Obtém-se uma ordem parcial para os testes de integração; testa-se o *SCC* {C1, C3, C5, C6, C4, C8, C2}; testa-se C7 usando C6 e C2. O peso das arestas deste *SCC* {C1, C3, C5, C6, C4, C8, C2} é, então, calculado da seguinte maneira:

C8->C2=3 * 3=9 C2->C4=1 * 2=2 C2->C3=1 * 3=3 **C1->C3=3 * 3=9** C3->C1=3 * 1=3

C3->C5=3 * 2=6 C5->C1=2 * 1=2 C5->C6=2 * 2=4 C6->C4=1 * 2=2 C3->C8=3 * 2=6

No exemplo, a remoção da aresta C1->C3 ou a remoção da aresta C8->C2 é indiferente pois ambas geram o mesmo número de *stubs* já que elas possuem o mesmo peso. Ao se eliminar a aresta C1->C3 o algoritmo de Tarjan [56] deve ser aplicado ao *SCC* {C1, C3, C5, C6, C4, C8, C2}, que gera o *SCC* {C2, C3, C5, C6, C4, C8} e o *SCC* trivial {C1}. Novamente, uma aresta do *SCC* não trivial deve ser eliminada para quebrar os ciclos. Por fim, a seqüência de aplicações do

algoritmo de Tarjan [56] e, de acordo com a eliminação das arestas, resultam em uma ordem para os testes de integração mostrada na *Figura 5.9.*

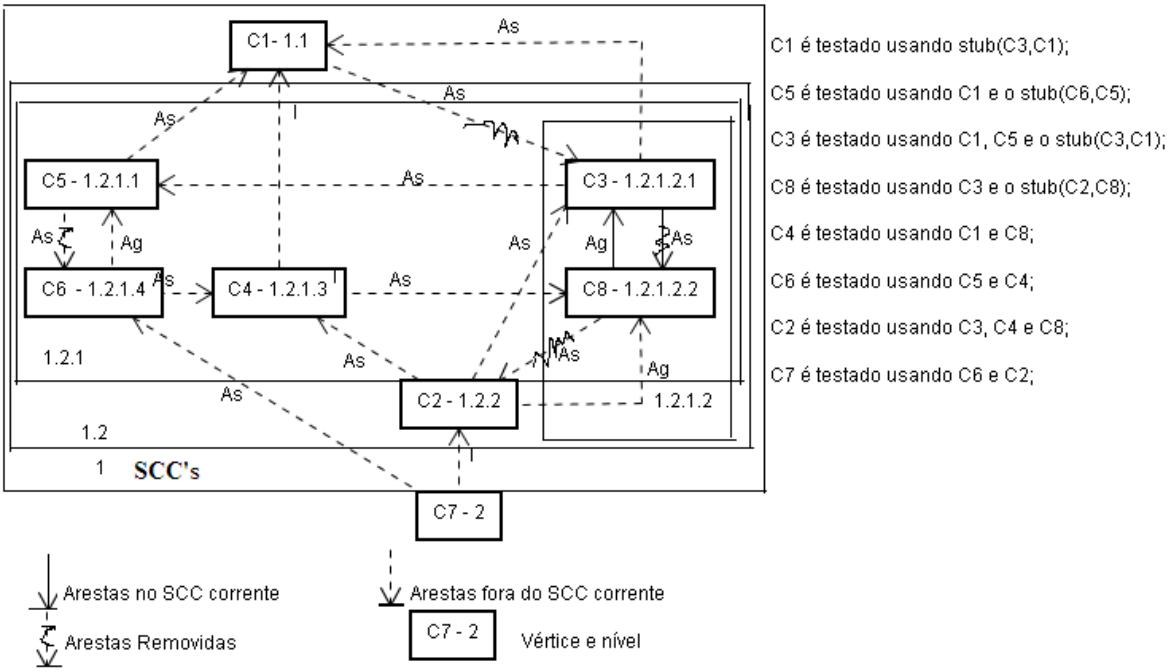


Figura 5.8: *ORD (Object Relation Diagram)* - Estratégia de Briand et. al [36].

ORD={C1, C2, C3, C4, C5, C6, C7}
1. {C1, C3, C5, C6, C4, C8,C2}: Aresta de maior peso C1->C3 = 9 1.1 {C1} 1.2 {C2, C3, C5, C6, C4, C8}: Aresta de maior peso C8->C2=9 1.2.1 {C3, C5, C6, C4, C8}: Aresta de maior peso C5->C6=4 1.2.1.1 {C5} 1.2.1.2 {C3, C8}: Aresta de maior peso C3->C8=1 1.2.1.2.1 {C3} 1.2.1.2.2 {C8} 1.2.1.3 {C4} 1.2.1.4 {C6}

1.2.2 {C2}

2. {C7}

Figura 5.9: Aplicação recursiva do algoritmo de Tarjan [56] - Estratégia de Briand et. al [36].

5.5 Travassos e Oliveira

Na proposta de Travassos [18] e Oliveira [20] o diagrama de classes de um projeto, descrito pela UML *(Unified Modeling Language),* é usado como base de entrada para todas as informações necessárias ao emprego da estratégia.

A partir de um conjunto de heurísticas que determinam critérios de precedência entre classes, poderá ser estabelecida uma lista ordenada de classes para a execução dos testes de integração.

Os critérios de precedência foram definidos com o propósito de verificar, com base na semântica estabelecida pela UML, quais as características determinantes para as classes serem 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 são:

- ***Herança:*** Garantir que a subclasse funcione de forma adequada significa, primeiramente, garantir que a superclasse (classes base) tenha sido testada. Quando a superclasse (classe base) for abstrata, deve-se testar primeiro a subclasse que seja menos acoplada.

  A análise de dependência em relação à classe base propicia uma análise indireta das dependências das subclasses, na medida em que a subclasse será testada após os testes da classe base.

- ***Assinatura dos Métodos:*** Testa-se primeiro a classe servidora e posteriormente a classe cliente.

- ***Agregação:*** Representa um relacionamento *todo-parte,* na qual as classes representam itens maiores formados por itens menores. Nesse trabalho, considera-se qualquer agregação como sendo composta, onde 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 deve ser atributo de outra. Sendo assim, será utilizada a navegabilidade para definir critérios de precedência quando a interface entre as duas classes ocorrer através de uma associação. Os casos em que houver navegabilidade bidirecional serão analisadas as possibilidades de propiciar a escolha da classe a ser ordenada primeiro.
- ***Classes de Associação:*** Uma classe de associação surge a partir da necessidade de colaboração entre duas outras classes. Desta forma, as classes que deram origem à classe de associação terão precedência de testes de integração sobre a classe derivada.
- ***Dependência:*** Nos testes de integração, a classe fornecedora terá precedência para ser testada.

Para viabilizar a aplicação dos critérios de precedência e estabelecer a ordem de prioridade entre as classes foram definidas duas propriedades:

1. O *fator de influência (FI)* de uma classe que 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.
2. 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.

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 no modelo, será possível estabelecer uma lista ordenada das classes para execução dos testes de integração. A partir do modelo representado na *Figura 5.10* obtém-se a *Figura 5.11* que representa a

precedência de uma classe em relação a outra no modelo. Por exemplo, a classe **Departamento** tem precedência somente sobre a classe **Escola** então FI=1.

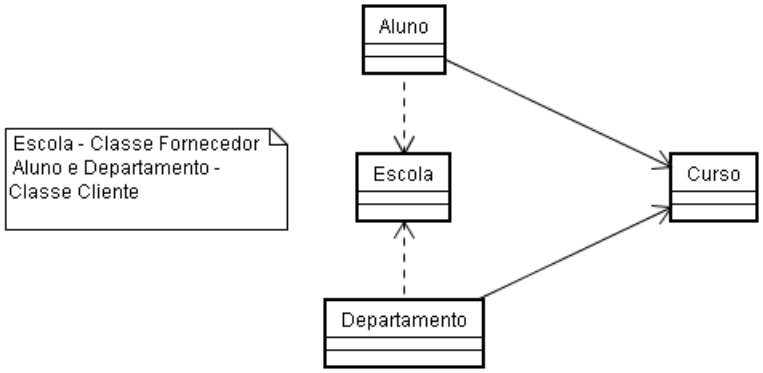


Figura 5.10: *Modelo exemplo.*

	Escola	Departamento	Curso	Aluno	FI	FIT
Escola					**0**	**2**
Departamento	**X**				**1**	**2**
Curso		**X**		**X**	**2**	**0**
Aluno	**X**				**1**	**2**

Figura 5.11: Tabela de precedência para cálculo de FI e FIT – 1ª. Interação

O *FIT* = 0 para a classe *Curso* indica o início da lista ordenada para os testes de integração. Para selecionar a ordem das próximas classes deverão ser recalculados os fatores de integração tardia *FIT* das demais classes desprezando o fator de influência das classes anteriormente selecionadas (*Figura 5.12).* Nesse exemplo, o resultado da lista ordenada para os testes de integração seria {Curso, Departamento ou Aluno; e Escola}.

	Escola	Departamento	Aluno	FI	FIT
Escola				**0**	**2**
Departamento	**X**			**1**	**0**
Aluno	**X**			**1**	**0**

Figura 5.12: Tabela de precedência para cálculo de FI e FIT- 2ª. Interação

5.6 Lima e Travassos

O trabalho proposto por Lima e Travassos [18] é uma complementação do trabalho proposto por Travassos [18] e Oliveira [20].

No trabalho proposto por Travassos e Oliveira [18] a aplicação das heurísticas em diagramas que não continham classes fortemente conectadas *(SCC)* mostrou-se eficiente. Entretanto, para diagramas contendo mais de uma classe com mesmo fator de integração tardia, representando ciclos de dependência entre as classes, as heurísticas não resultaram em um esforço de teste satisfatório, demandando um tratamento especial em situações de *deadlock.* Dessa demanda surgiu a necessidade de uma complementação das heurísticas e foram incluídos os seguintes critérios de precedência à proposta original:

- ***Cardinalidade:*** 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 o critério de precedência quando a cardinalidade representar a noção de opcionalidade (zero ou zero-para-muitos). Nesse caso, a classe com cardinalidade opcional deverá ser testada após a outra classe da associação.
- ***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. Essas classes têm seu teste de integração totalmente dependente da integração das demais classes do modelo.
- ***Inexistência de Fatores de Integração Tardia Nulos:*** Conforme exposto por Travassos e Oliveira [18] classes com *FIT*=0 são testadas primeiro. Entretanto, alguns modelos podem ser representados somente por classes com forte acoplamento, não existindo inicialmente classes com fator de integração tardia nulos. Nesses casos, a seqüência de testes de integração deve ser feita por meio das classes que possuam o menor *FIT* calculado.
- ***Tratamento de Deadlock:*** Um mesmo valor de *FIT* para mais de uma classe pode significar que essas classes possuem uma dependência entre si, existindo um ciclo. O tratamento dos

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

Para estabelecer prioridade entre as classes com mesmo valor de *FIT*, a estratégia respeita alguns critérios:

1. A classe selecionada deve gerar o menor número de *stubs* específicos necessários em comparação com o número de *stubs* para as outras classes com o mesmo valor de *FIT*.
2. No caso das classes necessitarem da implementação do mesmo número de *stubs*, deverá ser testada aquela cujos *stubs* apresentem a menor complexidade (medida pelo tamanho da classe, ou seja, pelo somatório do número de atributos e do número de métodos de cada *stub* [41]).
3. No caso de serem testadas algumas dessas classes com mesmo valor de *FIT,* diminuir o número de *stubs* necessários para testar as outras classes de mesmo *FIT*, indicando uma dependência interna, ou seja, classes que necessitem do mesmo *stub* para ser devidamente testada, esta classe deverá ser testada em primeiro lugar.
4. Caso alguma classe com mesmo valor de *FIT* apresente alguma associação com a navegabilidade obrigatória, esta classe deverá, preferencialmente, ser testada em primeiro lugar.

O diagrama *ORD da Figura 5.1,* por conter ciclos e por ter sido usado para a demonstração das outras propostas é usado também para demonstrar o emprego das heurísticas propostas nesse trabalho de Lima e Travassos [18].

A lista de classe não ordenada seria composta por {C1, C2, C3, C4, C5, C6, C7, C8} os valores dos fatores de influência seriam os apresentados na primeira coluna da *Figura 5.13.* A classe C7 possui *FI=0* e depende totalmente dos testes de integração das classes C2 e C8, que por sua vez são dependentes das demais classes, será removida da lista.

Obtém-se uma nova lista {C1, C2, C3, C4, C5, C6, C8} com os valores mostrados na segunda coluna da *Figura 5.13.* Observando a inexistência de fatores de integração tardia nulos,

as classes C1, C5 e C8 serão priorizadas conforme necessidade de dependência interna entre C5 e C1, pois ambas necessitam de um *stub* específico. Nesse momento, a lista de classes ordenadas para testes de integração será {C1, C5, C8}.

Classes	FI	FIT	FIT- 1A. Interação	FIT- 2A. Interação	FIT- 3A. Interacão
C1	3	3	3		
C2	1	1	7	2	0
C3	3	3	5	0	
C4	2	2	5	0	
C5	2	2	3		
C6	1	1	4	2	0
C7	0				
C8	2	2	3		

Figura 5.13: Interação dessa estratégia - cálculo de FI.

Calcula-se o *FIT*, segunda interação, para as classes restantes {C2, C3, C4, C6}. Calcula-se novamente para as classes com *FIT* diferente de 0 {C6, C2}, terceira interação, e finalmente obtém-se a lista ordenada de classes {C1, C5, C8, C4, C3, C6, C2, C7}.

A utilização da seqüência de ordenação das classes para os testes de integração encontrada seguindo as heurísticas apresentadas nesse trabalho, implicará na necessidade de implementação de dois *stubs* específicos: um *stub* de C3 para testar a classe C1 e outro *stub* de C3 para testar a classe C8, ou ainda um *stub* realístico de C3 para testar ambas as classes C1 e C3.

5.7 Stafford et. al *(Chaining)*

A proposta de Stafford et. al [27] baseia-se na arquitetura do sistema e considera tanto as dependências estruturais quanto as comportamentais para definir os componentes *que afetam* e os que *são afetados* por um componente alvo.

Para isso os autores desenvolveram uma técnica de análise de dependência arquitetural chamada encadeamento porta-a-porta *(port-to-port chaining)* e implementaram essa técnica em uma ferramenta chamada *Alladin* [26]. Nessa ferramenta, a representação da arquitetura consiste

num conjunto de células, onde cada célula representa um conjunto de relacionamentos que podem existir entre um dado par de elementos arquiteturais. Usa-se esse conjunto para o desenvolvimento de uma cadeia *(chain)* de elementos dependentes.

Encadeamentos (chain) representam relacionamentos de dependência em uma arquitetura. O *chain* individual une dentro de um *chain,* elementos associados de uma arquitetura que estão diretamente relacionados, enquanto um *chain* de dependências inclui associações entre elementos arquiteturais que estão relacionados indiretamente. Encadeamento *(chaining)* é o processo de aplicar algoritmos de dependência sobre uma descrição arquitetural a fim de produzir conjuntos de componentes e/ou elementos relacionados.

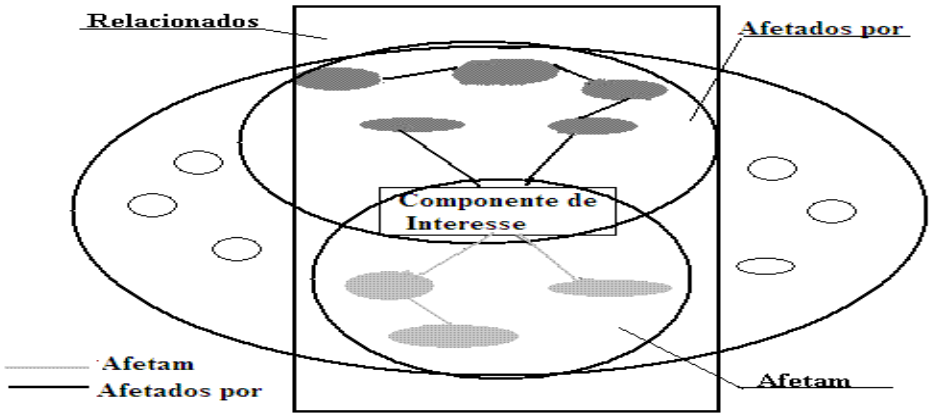
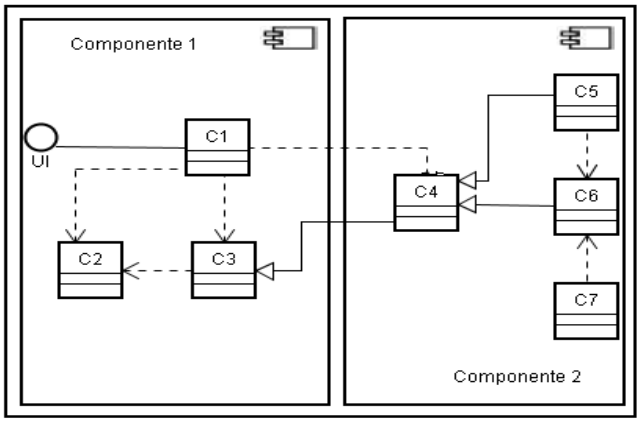


Figura 5.14 : *Cadeias (Chains)*

Essa técnica define três tipos de encadeamentos*:* afetados por *(affected-by),* afeta *(affects)* e relacionados *(related), (Figura 5.14).* Um relacionamento *afetado por* indica que o componente alvo da análise ou componente de interesse pode ser afetado pelos componentes conectados a ele no *chain.* Contrariamente, um relacionamento *afeta* contém um conjunto de componentes sobre o qual o componente de interesse pode potencialmente ter algum efeito. C*hains relacionados* são as uniões dos *chains que afetam* e dos *chains afetados.*

Alladin [26] usa uma matriz para representar os relacionamentos de dependência. A matriz é composta de *m* linhas por *n* colunas, onde *n* representa os componentes que dependem dos componentes de *m*. Por exemplo, se o componente *C1* é dependente do componente *C2*, então a célula na coluna *C1* e a linha *C2* detalham esse relacionamento. Em geral, para que a tabela de dependência possa ser construída para uma dada linguagem de descrição de arquitetura, é necessário entender os vários caminhos pelos quais os elementos arquiteturais podem se

relacionar pois a representação *C1*, *C2*, por exemplo, pode tanto representar uma conexão direta entre os componentes *C1* e *C2,* como conter uma combinação de relacionamentos entre *C1* e *C2*.



	C1	C3	C4	C5	C6	C7
C1			X			
C2	X	X				
C3	X		X			
C4				X	X	
C6				X		X
UI	X					

Figura 5.15 : Notação UML - Dependências estáticas e a correspondente Matriz de dependência.

Na matriz de dependência, as células com o *"X"* indicam que a classe pertencente à linha na matriz que contém este *"X"* mantém um relacionamento com a classe pertencente à coluna da matriz que contém esse *"X",* possibilitando a visualização do *chain* para as classes do exemplo em questão.

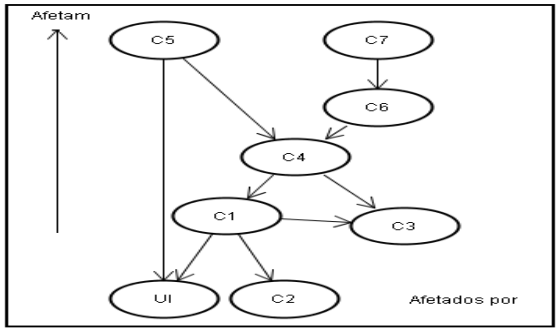


Figura 5.17: Chains

Seja *M* a matriz de dependência. Para M[C1,C2]="X" então C1∈ ao conjunto *Afetados por* de C2 e C2 ∈ ao conjunto de *afetados* de C1.

Por outro lado, uma vez que a C1 ∈ ao conjunto de *afetados por* de C2 e C2 ∈ ao conjunto de *afetados por* de C3 , então a C1 ∈ ao conjunto de *afetados por* de C3. De posse desses conceitos construiu-se o grafo de dependência *(Figura 5.17)* para o exemplo da matriz de dependência representado na *Figura 5.16.*

6 Capítulo

Um Método de Testes de Integração Para Sistemas Baseados em Componentes

Visando suprir a necessidade de um processo de desenvolvimento e testes de sistemas baseado em componentes de alta qualidade que concilie as vantagens do *DBC* e maximize o reuso de componentes prontos durante o desenvolvimento, estão sendo desenvolvidos um processo de desenvolvimento e um processo de testes como parte do projeto chamado *CompGOV* [46, 24], brevemente descrito como parte deste trabalho uma vez que o método proposto neste trabalho visa também a adequação ao projeto *CompGOV* [46, 24].

A característica importante para esses processos é o fato de eles serem genéricos, no sentido de serem constituídos de atividades gerais, comuns à maioria das metodologias de *DBC* atuais. Com isso, pretende-se que a proposta seja facilmente adaptável aos vários processos já utilizados atualmente, o que facilita sua utilização prática.

Desenvolvido por um consórcio de seis empresas e instituições de pesquisa de São Paulo e do Nordeste, o projeto *CompGOV* vem atuando em pesquisas para o desenvolvimento de componentes de software para o E-gov, e na construção de uma biblioteca virtual que possibilite o compartilhamento desses componentes cujo objetivo é otimizar os gastos públicos com a *Tecnologia da Informação* (TI), evitando investimentos redundantes. A intenção é que as áreas de TI de órgãos públicos ou privados possam visualizar com facilidade quais componentes estão disponíveis na biblioteca.

Pólos de tecnologia de Campinas (SP) e do Nordeste (Recife-PE e João Pessoa-PB) se unem para o desenvolvimento desse projeto *CompGOV*. O projeto é coordenado pelo Centro de Estudos e Sistemas Avançados de Recife (Cesar) e envolve o Centro de Pesquisas Renato Archer (CenPRA/MCT), a Ci&T Software, a Universidade Federal de Pernambuco (UFPE), a Universidade Federal da Paraíba (UFPB) e a Universidade Estadual de Campinas (Unicamp).

Todo produto gerado pelo projeto será público e livre, ficando à disposição da indústria brasileira para utilização dos componentes.

Dentro desse contexto do projeto *CompGOV* [24, 46] a proposta desse trabalho é apresentar um método de teste de integração baseado no processo genérico para o desenvolvimento de sistemas baseado em componentes com reuso de componentes [46], descrito na *Seção 6.1,* agregado a um processo genérico de testes de sistemas baseados em componentes, proposto em [24] e descrito na *Seção 6.2*.

Em geral, a realização dos testes de integração em sistemas baseados em componentes é uma atividade problemática, pois o fornecedor nem sempre disponibiliza o código fonte para o cliente/usuário do componente. Além disso, as descrições dos serviços providos e/ou dos serviços requeridos são em geral: (i) informais dificultando a conexão entre as interfaces providas e/ou requeridas do componente; (ii) a especificação do componente nem sempre contém informações suficientes e precisas que possam ser utilizadas de maneira concisa e (iii) os relacionamentos existentes entre os componentes são difíceis de serem identificados e gerenciados.

Esse trabalho considera que as fases referentes aos testes de unidade e aos testes de componentes tenham sido devidamente exercitadas e propõe um método que visa suprir as seguintes necessidades relacionadas aos testes de integração:

- Oferecer suporte na obtenção de uma ordem para os testes de integração dos componentes;
- Minimizar o número de *stubs* de teste a serem produzidos e, conseqüentemente, minimizar o esforço, e o custo durante os testes de integração;
- Garantir uma cobertura adequada ao teste do serviço provido pelo componente na medida em que o método permite uma visualização da dependência entre componentes no sistema.

Considera-se, para o método proposto, a ausência de código fonte e, desta forma, os passos para os testes de integração de sistemas baseados em componentes são descritos baseando-se na arquitetura do sistema. Arquitetura essa preferencialmente representada pelo diagrama de componentes, porém, pode-se utilizar o método proposto baseando-se em qualquer representação, diagramas e/ou documentos de especificação, capaz de demonstrar o relacionamento entre os componentes do sistema através de suas interfaces providas e requeridas.

Embora toda a descrição dos passos para o método proposto diga respeito a componentes convém lembrar que, conforme citado na *Seção 2.1,* um componente pode ter a granularidade de uma classe, de um conjunto de classes ou até mesmo de um sistema, composto de interfaces providas e/ou de interfaces requeridas.

Para que fosse possível propor um método para os testes de integração para sistemas baseados em componentes, vários trabalhos relacionados foram estudados (*Capítulo 5).* Pelo fato de se tratar de um refinamento de todas as outras propostas estudadas, algumas características dos trabalhos de Stafford et. al [27], Lima e Travassos [18] e Travassos [18] e Oliveira [20] foram selecionadas e adaptadas para compor o método proposto nesse trabalho.

Esse capítulo apresenta de forma resumida o processo de desenvolvimento [46] (*Seção 6.1)* e o processo de testes [24] (*Seção 6.2)* para sistemas baseados em componentes com reuso que fazem parte do projeto *CompGOV*.

Na *Seção 6.3* apresenta-se uma visão geral do método de teste de integração para sistemas baseados em componentes propostos nesse trabalho; as *Subseções*, *6.3.1* a *6.3.6*, apresentam e exemplificam cada passo referente ao método proposto.

Conclui-se esse capítulo com a *Seção 6.4* onde também se apresenta uma relação dos resultados obtidos pelos trabalhos estudados bem como o resultado obtido pelo método proposto.

6.1 Processo de Desenvolvimento de Sistemas Baseados em Componentes com Reuso de Componentes

O processo de desenvolvimento genérico descrito em [46] e, cuja *fase de preparação* dos testes pode ser realizada em paralelo, compõe-se de oito passos básicos descritos a seguir e representados no diagrama da *Figura 5.1*:

1. *Especificação de requisitos:* esse passo abrange basicamente sete atividades (i) análise e representação do domínio do problema; (ii) identificação inicial dos requisitos funcionais; (iii) por se tratar de um processo para reuso de componentes, já nessa fase devem ser

construídos os protótipos para os componentes prontos; (iv) refinamento dos requisitos funcionais; (v) especificação dos requisitos de qualidade; (vi) especificação das restrições de desenvolvimento; e (vii) definição das restrições de reuso dos componentes. Em geral, os requisitos podem ser vistos como os objetivos esperados para o sistema, assim como as condições e as capacidades necessárias para o alcance destes objetivos.

2. *Projeto arquitetural:* com base na especificação de requisitos, nesse passo são avaliadas as restrições arquiteturais e selecionada a arquitetura adequada. Os componentes são então posicionados no estilo arquitetural adotado. Devido ao tamanho e à complexidade dos sistemas o projeto arquitetural é importante para a visualização das falhas e o sucesso relacionado aos requisitos do sistema. Uma vantagem interessante desse projeto é a alta granularidade de reuso que, no processo *DBC* ajuda para a redução no tempo de desenvolvimento.

3. *Reutilização de COTS:* nesse processo, visando reduzir o custo e o tempo de desenvolvimento, materializam-se os componentes reutilizando componentes já utilizados pela organização, adquirem-se componentes de terceiros ou desenvolvem-se novos componentes. Nessa fase o objetivo principal é a redução de custo e tempo de desenvolvimento do sistema.

4. *Implementação de componentes novos:* desenvolvem-se novos componentes para o sistema quando da ausência de componentes que possam ser reusados. Esse processo é recursivo a partir da especificação de requisitos, uma vez que os componentes, em geral, possuem alta granularidade e possibilitam o reuso de partes internas dos componentes.

5. *Implementação dos conectores e adaptadores:* nesse passo todos os componentes do sistema estão desconectados. Essas conexões são realizadas através da conexão entre as interfaces requeridas de um componente e a interface provida de outro.

6. *Integração dos componentes:* nessa fase devem-se integrar todos os componentes e subsistemas para montar o sistema.

7. *Validações e correções do sistema:* em geral, a validação do software consiste em uma seqüência de testes elaborados a partir dos documentos de requisitos. Em sistemas reais devido ao grande número de serviços que podem ser oferecidos, essa atividade deve ser executada automaticamente com o auxílio de ferramentas *CASE*.

8. *Corrigir o sistema:* caso alguma falha tenha sido encontrada o sistema será corrigido e o passo referente à validação será realizado novamente.

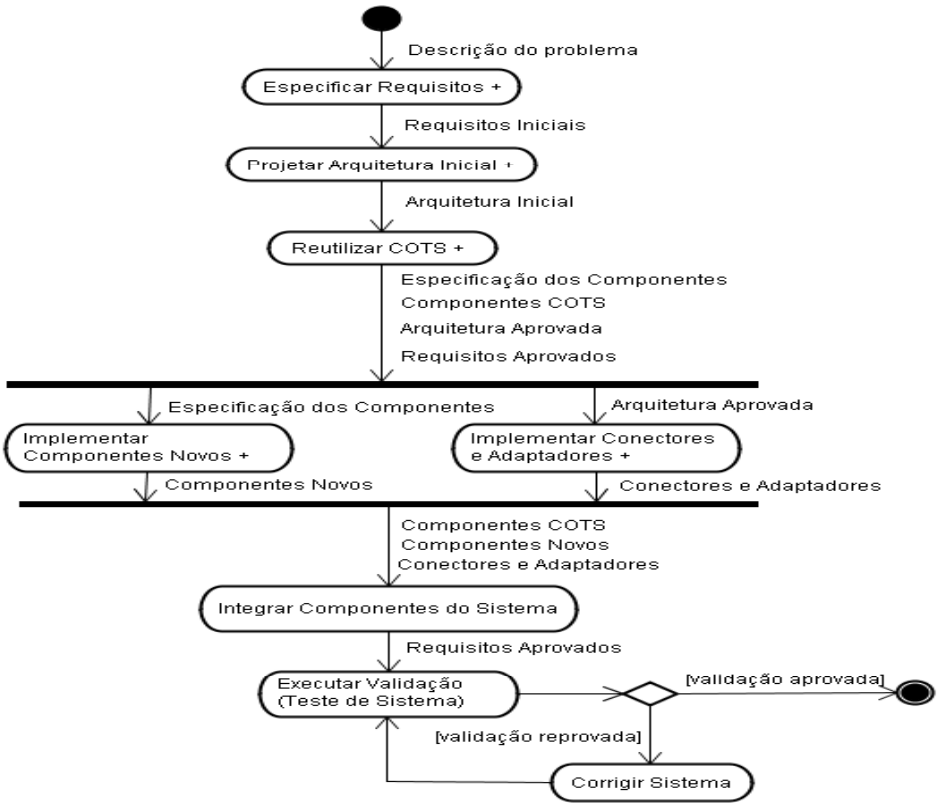


Figura 6.1 - Visão Geral do Processo para o Desenvolvimento Baseado em Componentes com Reuso de Componentes [46].

6.2 Processo de Teste de Sistemas Baseados em Componentes

No contexto do projeto *CompGOV* [24, 46] e do desenvolvimento incremental, o processo de testes genérico que está sendo proposto em [24] torna-se recursivo pois os casos de testes vão sendo realimentados à medida que se desenvolve o sistema. Trata-se de um processo genérico que visa sua agregação a um processo de desenvolvimento como o descrito na *Seção 6.1*.

Para cada fase do processo de desenvolvimento *DBC* são previstas as fases de testes. A *Figura 6.2* apresenta, de forma simplificada, algumas fases do processo *DBC* [46] agregado à fase do processo de testes [24]. Esse processo de testes [24] compõe-se a princípio de cinco fases descritas a seguir.

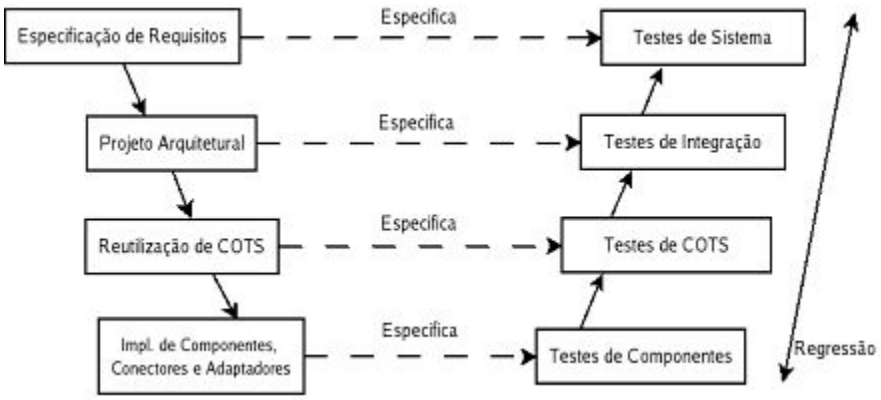


Figura 6.2 - Processo de Teste Genérico Adaptado para o *DBC* [24].

6.2.1 Teste de Sistema

As atividades relacionadas aos testes de sistemas ocorrem no final da especificação de requisitos.

Uma vez que os requisitos funcionais foram devidamente especificados, planejam-se e definem-se os testes de sistema que são constituídos pelos *testes funcionais* visando determinar se os requisitos funcionais foram implementados no sistema sob teste e pelos *testes de robustez* visando determinar a robustez do sistema na presença de alguma falha em seus componentes.

Um software robusto é aquele que demonstra a capacidade de produzir serviços de confiança mesmo em um ambiente hostil [44], ou seja, independente da entrada as possíveis falhas, caso existam, serão detectadas.

6.2.2 Testes de Integração

Uma vez que o foco desse trabalho são os testes de integração e cuja proposta visa à agregação do método ao processo de testes proposto em [24], embora seja possível agregá-lo a qualquer processo de testes, a descrição correspondente a este item faz parte da *Seção 6.3*.

Convém citar que o método de testes de integração para sistemas baseados em componentes propostos nesse trabalho não diferencia um componente já utilizado pela organização de um componente adquirido através de catálogos de terceiros *(COTS)* ou de um componente novo implementado para o sistema em questão.

6.2.3 Testes de *COTS*

A primeira atividade a ser executada nessa fase é a de especificação ou de seleção dos testes para os componentes *COTS*.

Para *COTS* adquiridos de terceiros sem a especificação de casos de teste, especificam-se os casos de teste. Para *COTS* já utilizados dentro da organização, ou seja, que estejam sendo reaproveitados provavelmente os casos de testes já foram especificados e implementados.

Finaliza-se essa primeira atividade selecionando-se um subconjunto dos casos de testes ou até todos os casos de teste para testar o componente COTS em questão.

A segunda atividade nessa fase é a implementação dos casos de teste, necessária no caso de não existirem os casos de teste implementados.

Finalmente em uma terceira atividade nessa fase, executam-se os casos de teste. É importante verificar que no caso em que os testes já estejam especificados e implementados, o componente provavelmente estará sendo re-testado, o que é necessário para avaliar se o componente faz, nesse novo contexto, o que deveria fazer segundo a especificação da arquitetura do sistema.

6.2.4 Teste de Componentes

A especificação dos testes de componentes novos, adaptadores e conectores ocorrem nessa fase, em paralelo à fase de implementação de componentes, conectores e adaptadores.

Exercitam-se os testes à medida que vão sendo finalizadas as implementações desses componentes, conectores e adaptadores.

Nessa fase testa-se cada componente de forma independente dos demais. Os testes são funcionais, isto é, exercitam-se os serviços prestados pelo componente, uma vez que se espera que a estrutura interna deste componente já tenha sido testada pelos desenvolvedores na fase de testes de unidade.

6.2.5 Teste de Regressão

Os testes de regressão são executados com o intuito de verificar se as funcionalidades do sistema continuam executando conforme o esperado ou se alguma funcionalidade já não se comporta como antes de uma determinada correção e/ou manutenção no sistema, ou seja, se os testes anteriormente efetuados ainda são válidos [53].

A principal diferença entre o teste de regressão e o teste de desenvolvimento, é que no teste de regressão o conjunto de testes que foi exercitado na versão anterior ao exercício do teste de regressão já está disponível para execução.

Existem basicamente duas estratégias para a execução do teste de regressão. Uma consiste em *retestar tudo* onde todo o conjunto de teste disponível é exercitado novamente e a outra é uma estratégia *seletiva*, proposta nesse processo de teste genérico, que consiste em *retestar* um subconjunto dos testes originais.

6.3 Proposta do Método de Teste de Integração para Sistemas Baseados em Componentes

O objetivo do teste de integração é a partir dos componentes testados no nível de unidade, construir a arquitetura do sistema que foi determinada pelo projeto de forma sistemática, testando também a interface dos componentes. Sendo assim, sugere-se que o método proposto seja utilizado quando a arquitetura do sistema esteja fechada, ou seja, no nível mais baixo de desenvolvimento. Aplicar o método proposto antes disso pode acarretar re-trabalho, pois qualquer alteração em um componente pode alterar sua ordem em relação aos outros componentes durante os testes de integração.

O objetivo da análise de dependência, no contexto de testes de integração, é determinar a ordem dos testes. Essa ordem indica que componentes devem ser testados primeiro de forma a reduzir, tanto quanto possível, o número de *stubs*. Essa redução se faz necessária para a viabilidade dos testes de integração, pois, conforme descrito na *Seção 4* os *stubs* representam *overhead,* ou seja, software que deve ser escrito, mas que não é entregue com o produto final. Além do que, em geral, necessitam de um grande esforço e tempo para a sua implementação.

Com base nos estudos das técnicas de teste de integração descritas na *Seção 4* e nos estudos dos trabalhos relacionados descritos na *Seção 5,* definem-se seis passos para o método proposto nesse trabalho, em destaque na *cor cinza* no diagrama da *Figura* 6.3 e detalhado nas próximas *Seções.*

Como não faz parte dos objetivos desse trabalho realizar os testes de integração e sim, conforme citado anteriormente, propor uma ordem para a realização dos testes de integração visando diminuir o número *stubs* e o esforço e custos gastos durante os testes de integração, os passos não destacados no diagrama da *Figura 6.3 (cor branca)* não serão descritos e nem tampouco detalhados nesse trabalho.

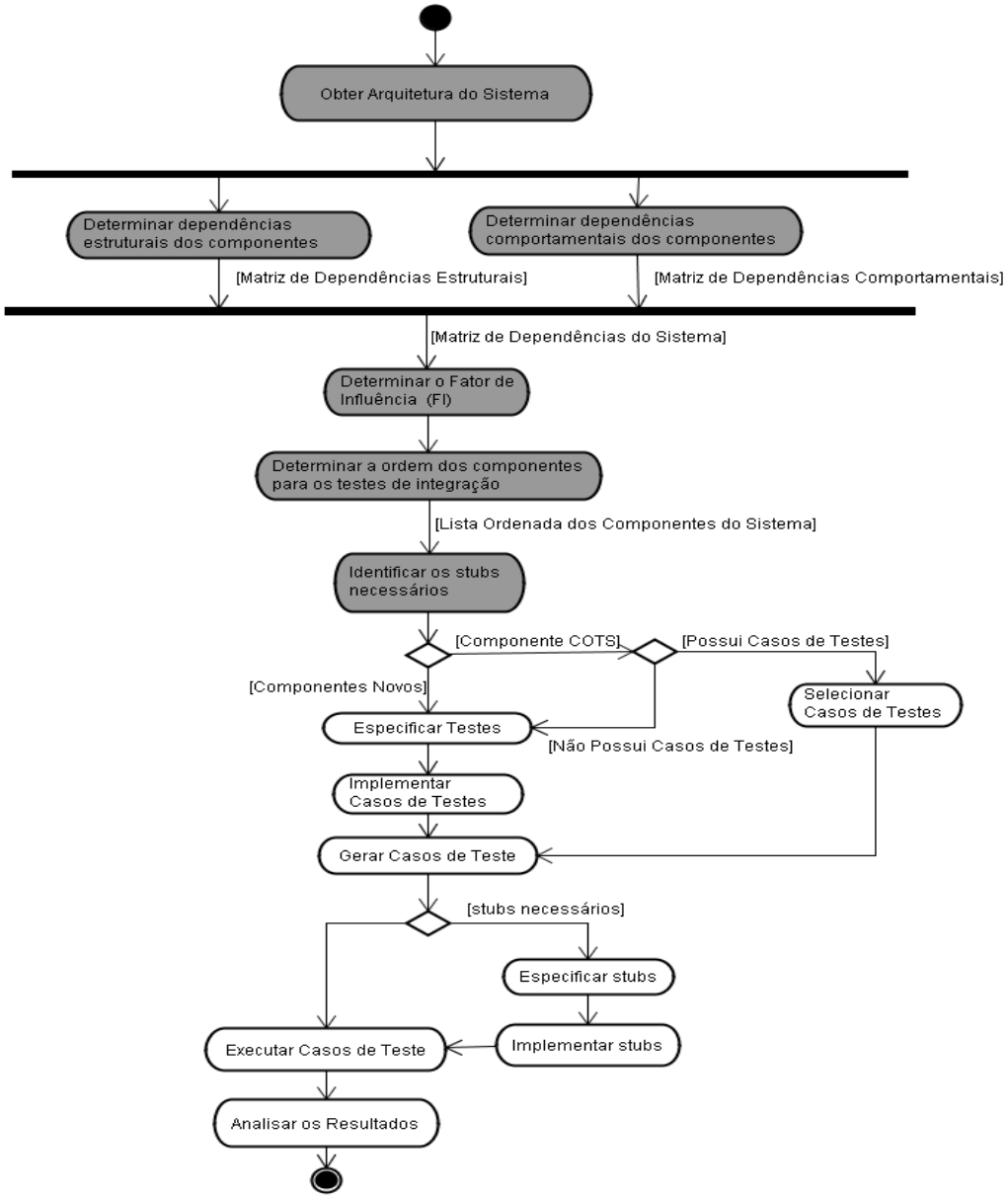


Figura 6.3: Visão Geral do Método de Testes de Integração para Sistemas Baseados em Componentes.

As *Seções* a seguir apresentam os passos desse método. Para ilustrar, será utilizado um exemplo, inspirado de outros estudos [13, 33], cujo *ORD* foi apresentado na *Figura 5.1*.

6.3.1 Obter a Arquitetura do Sistema

Uma vez que a arquitetura do sistema é essencial para o processo de testes de integração e um processo de teste de integração freqüentemente revela omissões e ambigüidades nos requisitos e na arquitetura, no método proposto faz-se necessário para a aplicação dos testes de integração, que o sistema possua e disponibilize um projeto de arquitetura definido e atualizado.

O exemplo em questão baseia-se no *ORD* da *Figura 5.1.* Para tanto as terminologias de cada aresta do *ORD* referente ao paradigma da linguagem *OO* foram substituídas pela correspondente representação *UML*. Assim, representa-se a arquitetura do exemplo através do diagrama de classes e por se tratar de uma proposta que visa às relações dinâmicas método a método entre os componentes, necessitou-se que os métodos públicos fossem representados no diagrama exemplo, em questão *(Figura 6.4).*

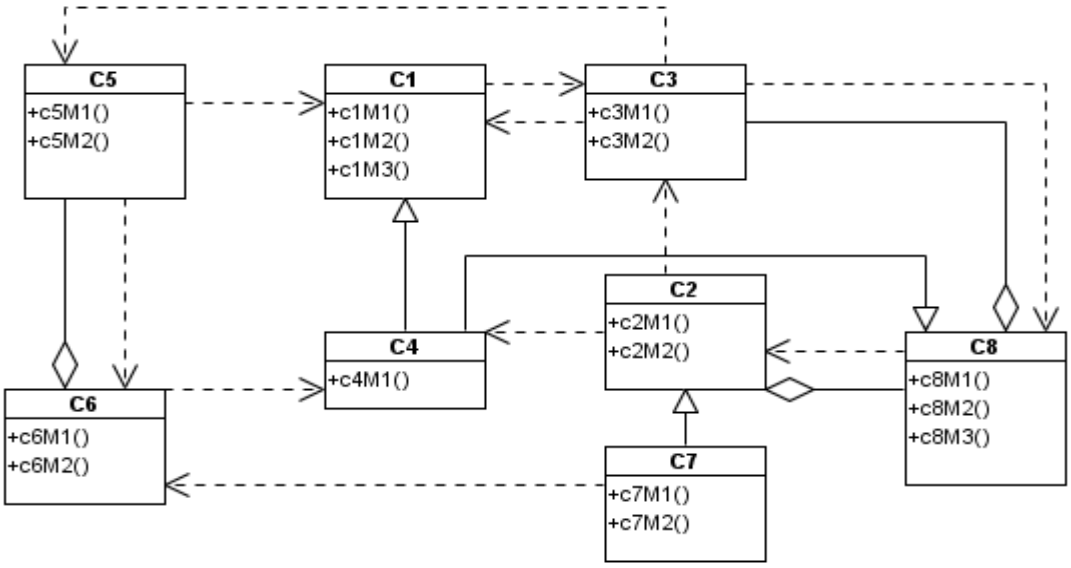


Figura 6.4: Diagrama de Classes Exemplo. Baseado no *ORD* da *Figura 5.1*.

6.3.2 Determinar Dependências Estruturais dos Componentes

Relacionamentos de dependência arquitetural surgem das conexões entre os componentes e das restrições sobre suas interações, conforme visto na *Seção 5*. Esses relacionamentos podem envolver algumas formas de controle ou de fluxo de dados, mas geralmente envolvem a estrutura e o comportamento do sistema.

A dependência estrutural atua sobre as especificações de dependências *estáticas* entre os componentes [27, 28] e permite-nos localizar especificações fontes que contribuem para a descrição de alguns estados ou interações do componente [25, 27].

Somente as dependências estruturais, obtidas através do código fonte, da arquitetura do sistema, do diagrama de componentes ou do diagramas de classes, não são suficientes para um método de teste de integração eficiente pois algumas interações dinâmicas podem não ser levadas em conta, uma vez que não são mostradas em um diagrama que represente apenas interações estáticas. Alguns exemplos de relacionamentos arquiteturais estruturais são [27, 28]:

1. *Inclusão textual* - A especificação de um componente pode ser criada a partir de vários outros pequenos trechos de código fonte *(módulos)* combinados textualmente.

2. *Importação/Exportação* - A especificação de um componente permite descrever a informação exportada ou importada entre os módulos fonte.

3. *Herança* - A especificação de um componente pode ser criada através de herança de outros módulos.

No contexto de uma *inclusão textual* inclui-se o polimorfismo onde mais de um componente oferece a mesma interface provida. No caso do componente ser uma classe, o polimorfismo ocorre entre uma classe e suas derivadas.

Caso a arquitetura do sistema não represente de qual componente a interface está sendo requerida para se determinar sua dependência estrutural, recorre-se às especificações, descrições de *API's* ou até mesmo ao código fonte.

Embora a obtenção das dependências estruturais faça parte do método proposto nesse trabalho e algumas formas de obtê-las tenham sido demonstradas através dos trabalhos estudados

(Seção 5), o foco desse trabalho é determinar as dependências comportamentais entre componentes e as operações executadas por ele (interface requerida).

6.3.3 Determinar Dependências Comportamentais dos Componentes

A dependência comportamental atua sobre as dependências de interações *dinâmicas* [27, 28] e permite-nos relatar estado ou interações para outro estado ou interações do componente [25, 41]. Alguns exemplos de relacionamentos arquiteturais comportamentais são [27, 28]:

1. *Temporal* - O comportamento de um componente precede ou resulta do comportamento de outro componente.

2. *Baseado em Estado* - O comportamento não ocorre a menos que o sistema, ou parte do sistema esteja em um estado específico.

3. *Causal* - O comportamento de um componente implica no comportamento de outro componente.

4. *Entrada/Saída* - O componente requer/provê informações de/para outro componente.

Para a obtenção dos relacionamentos comportamentais: *Temporal, Baseado em Estado e Causal* [27], fazem-se necessários os diagramas de seqüência, ou colaboração, pois estes diagramas representam a interação entre os componentes em tempo de execução, ou ainda os diagramas de detalhamento que detalham ações chamadas de comportamento e podem ser usados para modelar o comportamento de um método para que todas as suas saídas possíveis possam ser representadas.

Já os relacionamentos comportamentais de *Entrada/Saída* podem ser obtidos através das operações da interface do componente, ou seja, interfaces requeridas *(Entrada)* são as operações

que um componente executa de outro componente e interfaces providas são as operações que um componente implementa *(Saída)*.

Faz parte dos objetivos do método proposto determinar uma ordem para os testes de integração de componentes com base nas dependências comportamentais dos mesmos, mais especificamente, baseando-se nas operações executadas e/ou implementadas pelos componentes. Desta forma, para a demonstração do método e baseando-se no exemplo em questão apenas os relacionamentos de *Entrada/ Saída* serão tratados. Para tanto, uma vez que a arquitetura do exemplo representa o componente na granularidade de uma classe, necessitam-se das interfaces públicas dos métodos.

Qualquer documentação, especificação ou relatório que possibilite a visualização das operações providas e/ou requeridas entre os componentes, pode ser usada para a obtenção das dependências comportamentais de *Entrada/Saída*. Para o exemplo em questão usou-se a mesma notação utilizada pela ferramenta *dependency finder* [14] da qual se pode obter uma separação das dependências do sistema, por métodos.

A *Figura 6.5* descreve as relações comportamentais para o exemplo em questão contendo as separações de dependências por métodos.

sistemaExemplo

C1 // componente C1

c1M1() // método do componente C1

← sistemaExemplo.C3.c3M2() // o método c3M2 **usa** o método c1M1 de C1

← sistemaExemplo.C5.c5M1()

c1M2()

← sistemaExemplo.C5.c5M2()

→ sistemaExemplo.C3.c3M2() // o método c3M2 **é usado** pelo método c1M2 de C1

c1M3()

← sistemaExemplo.C4.c4M1()

→ sistemaExemplo.C3.c3M1()

C2

c2M1()

← sistemaExemplo.C8.c8M3()

← sistemaExemplo.C8.c8M1()

→ sistemaExemplo.C8.c8M2()

c2M2()

← sistemaExemplo.C7.c7M2()

← sistemaExemplo.C7.c7M1()

→ sistemaExemplo.C3.c3M1()

→ sistemaExemplo.C4.c4M1()

C3

c3M1()

← sistemaExemplo.C1.c1M1()

← sistemaExemplo.C1.c1M2()

c3M2()

← sistemaExemplo.C2.c2M2()

→ sistemaExemplo.C8.c8M3()

→ sistemaExemplo.C5.c5M2()

→ sistemaExemplo.C1.c1M3()

C4

c4M1()

← sistemaExemplo.C2.c2M1()

← sistemaExemplo.C6.c6M1()

→ sistemaExemplo.C1.c1M2()

→ sistemaExemplo.C8.c8M2()

→ sistemaExemplo.C8.c8M3()

C5

c5M1()

← sistemaExemplo.C3.c3M1()

c5M2()

← sistemaExemplo.C6.c6M1()

C6

c6M1()

← sistemaExemplo.C5.c5M2()

← sistemaExemplo.C7.c7M1()

```
    c6M2()
        ← sistemaExemplo.C7.c7M2()

        → sistemaExemplo.C4.c4M1()
        → sistemaExemplo.C5.c5M2()
C7
    c7M1()
        → sistemaExemplo.C2.c2M1()
C8
    c8M1()
        ← sistemaExemplo.C4.c4M1()

    c8M2()
        ← sistemaExemplo.C3.c3M1()
    c8M3()
```

***Figura 6.5*:** Relatório Exemplo de Dependências entre Métodos. Métodos obtidos da *Figura 6.4.*

Interpreta-se, por exemplo, para o componente C1, a notação utilizada para o relatório exemplo *(Figura 6.5)* da seguinte forma:

1. As *entradas* do relatório "←" identificam quais métodos da interface pública de um determinado componente usam os métodos da interface pública do componente C1. Assim, a linha correspondente a "← sistemaExemplo.C3.c3M2()" no topo do relatório, significa que o método c3M2() da interface pública do componente C3 usa o método c1M1() da interface pública do componente C1, ou seja, o componente C3 requer *(Entrada)* a operação executada pelo método c1M1() do componente C1, e o componente C1 provê *(Saída)* as operações requeridas pelo C3 através da implementação do método c1M1();

2. As saídas do relatório "→" identificam quais componentes são usados pelo componente C1. Assim, a linha para o componente C1 correspondente a "→ sistemaExemplo.C3.c3M2()" significa que o componente C1 requer (*Entrada)* a operação executada pelo método c3M2() do componente C3, e o componente C3 provê *(Saída)* as operações do requeridas por C1 através da implementação do método c3M2().

Representa-se a matriz de dependências baseando-se na proposta do trabalho de Stafford et. al [27], descrito na *Seção 5.7,* onde as células com o *"X"* indicam que o componente pertencente à coluna da matriz que contém esse *"X"* mantém um relacionamento de dependência *(afetados por)* com o componente pertencente à linha da matriz que contém esse *"X",* possibilitando a visualização do encadeamento *(chain)* dos componentes na matriz de dependência.

	c1M1	c1M2	c1M3	c2M1	c2M2	c3M1	c3M2	c1M1	c1M2	c1M3	c4M1	c8M1	c8M2	c8M3	c5M1	c5M2	c6M1	c6M2	c2M1	c2M2	c7M1	c7M2	c8M1	c8M2	c8M3
C1																									
c1M1							x								x										
c1M2											x					x									
c1M3							x				x														
C2																									
c2M1												x		x							x		x		x
c2M2																					x	x			
C3																									
c3M1	x	x	x		x			x	x	x										x					
c3M2		x			x				x											x					
C4																									
c1M1							x								x										
c1M2											x					x									
c1M3							x				x														
c4M1				x	x												x	x	x	x					
c8M1											x														
c8M2						x					x														
c8M3							x				x														
C5																									
c5M1						x																			
c5M2							x										x	x							
C6																									
c6M1																x					x				
c6M2																						x			
C7																									
c2M1												x		x							x		x		x
c2M2																					x	x			
c7M1																									
c7M2																									
C8																									
c8M1											x														
c8M2				x		x					x														
c8M3							x				x														

Figura 6.6: Matriz de Dependência Comportamental do Exemplo *(Figura 6.4).*

Desta forma, baseando-se no relatório de dependências exemplo *(Figura 6.5)* monta-se a matriz de dependências entre os métodos da interface pública do exemplo *(Figura 6.6)*.

Sendo assim, a *Figura 6.6* representa as dependências comportamentais do exemplo, onde cada coluna representa as operações da interface pública dos componentes que dependem de cada operação da interface pública do componente de cada linha da matriz.

Além da relação *"oferece-requer"* na interação entre os componentes, leva-se também em conta a relação de herança mostrada na arquitetura do sistema exemplo. Essa relação é levada em conta acrescentando-se ao componente derivado (subclasse) os métodos da interface do componente mais genérico (superclasse). Com isso pode-se levar em conta o polimorfismo.

6.3.4 Determinar o Fator de Influência (*Afetados por)*

Com base na matriz de dependência do exemplo *(Figura 6.6)* podem-se identificar os *componentes afetados por,* obtidos conforme descrição do trabalho proposto por Stafford et. al [27] (*Seção 5.7).* Esse passo abrange basicamente 2 atividades:

1. Para cada célula da matriz que contém um *"X",* determinam-se quantas operações das interfaces públicas dos componentes da coluna da matriz são *afetadas pela* execução dos métodos da interface pública *(interfaces requeridas)* de cada componente em cada linha da matriz. Por exemplo, seja a matriz M *(Figura 6.7)*, o *"X"* em M[c1M1, c3M2] e M[c1M1, c5M1] indica que o componente C1, que possui a operação c1M1(), pertence ao conjunto de componentes que *afetam* os componentes C3 e C5 e os componentes C3 e C5 que possuem as interfaces públicas c3M2() e c5M1(), respectivamente, pertencem ao conjunto de componentes *afetados pelo* componente C1, ou seja, C3 e C5 dependem de C1. A célula na matriz referente à M[c1M1, *Afetados por*] será preenchida com o valor do somatório dos métodos da operação dos componentes *afetados pelos* métodos da interface pública componente C1. Nesse exemplo, M[c1M1, *Afetados por*] = 2 diz respeito ao *"X"* de M[c1M1, c3M2] e M[c1M1, c5M1] e significa que a execução das operações públicas c3M2

do componente C3 e c5M1 do componente C5 são afetadas pela implementação da operação pública c1M1 do componente C1.

	C1			C2		C3		C4							C5		C6		C7				C8			FI
	c1M1	c1M2	c1M3	c2M1	c2M2	c3M1	c3M2	c1M1	c1M2	c1M3	c4M1	c8M1	c8M2	c8M3	c5M1	c5M2	c6M1	c6M2	c2M1	c2M2	c7M1	c7M2	c8M1	c8M2	c8M3	
C1																										
c1M1							x								x											2
c1M2											x					x										2
c1M3							x				x															2
C2																										
c2M1												x		x							x		x		x	5
c2M2																					x	x				2
C3																										
c3M1	x	x	x		x			x	x	x										x						8
c3M2		x			x				x											x						4
C4																										
c1M1							x								x											2
c1M2											x					x										2
c1M3							x				x															2
c4M1				x	x												x	x	x	x						6
c8M1											x															1
c8M2						x					x															2
c8M3							x				x															2
c5M1						x																				1
c5M2							x										x	x								3
C6																										
c6M1																x					x					2
c6M2																						x				1
C7																										
c2M1												x		x							x		x		x	5
c2M2																					x	x				2
c7M1																										0
c7M2																										0
C8																										
c8M1											x															1
c8M2				x		x					x															3
c8M3							x				x															2

Figura 6.7: Matriz de Dependência do Exemplo (Figura 6.4) - Componentes *Afetados***.**

2. Repete-se o processo para cada linha da matriz, ou seja, para cada componente do sistema.

Assim como a proposta de Travassos [18] e Oliveira [20] (*Seção 5.5)* que quantifica a relação de precedência estática entre as classes e identifica essa quantidade através do termo *FI (fator de influência),* o método proposto também faz uso desse termo *FI (fator de influência).* Porém*,* diferentemente de Travassos e Oliveira esse trabalho representa as relações dinâmicas entre as operações que um componente implementa e as operações executadas de um outro componente, ou seja, entre uma interface provida e sua correspondente interface requerida.

Assim, define-se que o *FI,* proposto nesse trabalho é um valor que quantifica a relação de precedência entre as operações implementadas por um componente *(Figura 6.7 - linha)* e as

operações que ele executa de outro componente *(Figura 6.7 - coluna*), sendo, quantas interfaces públicas (coluna) *são afetadas* pela interface pública (linha).

Portanto, o cálculo de *FI* para o método proposto nesse trabalho é diretamente proporcional ao número de interfaces públicas que precisam ser integradas posteriormente à integração da interface pública de um determinado componente. Quanto maior o número de componentes que executam operações de um determinado componente, isto é, possuam uma relação de precedência *(afetados pelo)* com o componente sob análise, maior será seu fator de influência. Assim, *FI=0* significa que não há precedência desse componente sobre nenhum outro componente do sistema.

Desta forma identifica-se a coluna na matriz *(Figura 6.7)* cujas células irão conter o somatório *(afetados por)* como sendo um *fator de influência (FI).*

6.3.5 Determinar a Ordem dos Componentes para os Testes de Integração

Nesse passo necessita-se determinar a ordem em que os componentes serão testados de forma a minimizar o número de *stubs.*

A matriz obtida na *Seção 6.3.4* é composta por uma coluna *(FI)* cujo conteúdo das células representa o somatório das operações das interfaces públicas dos componentes que dependem diretamente das operações da interface pública de um outro componente.

Na *Figura 6.7* para o componente C7 obtém-se um valor de *FI=0* correspondente à relação de precedência de suas *interfaces públicas* em relação às interfaces públicas dos outros componentes. Isso significa, pela definição do cálculo de *FI (Seção 6.3.4),* que esse componente depende totalmente dos testes dos outros componentes e, portanto, deve ser testado por último. Sendo assim a ordem para o componente C7 já está definida.

Após o cálculo de *FI* e definida a ordem para os componentes cujas interfaces públicas possuem *FI=0*, necessita-se da definição de uma ordem para os componentes restantes na matriz cujas interfaces públicas possuem um *FI* diferente de zero. Para tanto, esse passo baseia-se no cálculo do *fator de integração tardia (FIT),* conforme proposto por Travassos e Oliveira [18] e descrito na *Seção 5.5* desse trabalho.

	C1			C2		C3		C4							C5		C6		C7				C8			FI	FIT 1a.
	c1M1	c1M2	c1M3	c2M1	c2M2	c3M1	c3M2	c1M1	c1M2	c1M3	c4M1	c8M1	c8M2	c8M3	c5M1	c5M2	c6M1	c6M2	c2M1	c2M2	c7M1	c7M2	c8M1	c8M2	c8M3		Interação
C1																											12
c1M1							x								x											2	8
c1M2											x					x										2	12
c1M3							x				x															2	8
C2																											18
c2M1												x		x							x		x		x	5	9
c2M2																					x	x				2	18
C3																											15
c3M1	x	x	x		x			x	x	x										x						8	6
c3M2		x			x				x											x						4	15
C4																											19
c1M1							x								x											2	8
c1M2											x					x										2	12
c1M3							x				x															2	8
c4M1				x	x												x	x	x	x						6	19
c8M1											x															1	10
c8M2						x					x															2	0
c8M3							x				x															2	10
C5																											6
c5M1						x																				1	4
c5M2							x										x	x								3	6
C6																											9
c6M1																x					x					2	9
c6M2																						x				1	9
C7																											18
c2M1												x		x							x		x		x	5	6
c2M2																					x	x				2	18
c7M1																										0	16
c7M2																										0	5
C8																											10
c8M1											x															1	10
c8M2				x		x					x															3	0
c8M3							x				x															2	10

Figura 6.8: Matriz de Dependência do Exemplo – Cálculo de FIT *(1ª. Interação).*

Para o método proposto usa-se o resultado obtido após o cálculo de *FIT* da mesma maneira que Travassos e Oliveira, ou seja, o componente de menor *FIT* será integrado primeiro. A diferença nesse trabalho é que o cálculo considera as dependências dinâmicas entre a implementação e a execução de suas operações sobre a interface pública de outros componentes. Assim, para o exemplo em questão e destacado em cinza na matriz da *Figura 6.8,* obtém-se o *FIT* (última coluna da matriz) para o componente para a operação c1M2 somando-se os valores dos *FI's* obtidos para os métodos da interface pública do componente C3. Por exemplo, o *FIT* M[c1M2, FIT] = 12 correspondendo ao somatório dos valores de M [c3M1, FI] com M [c3M2, FI] destacados na matriz. O *FIT* do componente C1 corresponde ao maior *FIT* obtido para suas operações.

Também se fazem necessários nesse passo, alguns critérios de precedência propostos por Lima e Travassos [18] e descritos na *Seção 5.6* no contexto de classes; porém, conforme citado

Seção 2.1, classes podem ser tratadas na granularidade de componentes, então se adaptam os critérios de precedência no contexto de componentes para o método proposto, desta forma:

- ***Inexistência de Fatores de Integração Tardia Nulos:*** Algumas arquiteturas são compostas apenas por componentes em um cluster ou fortemente conectados *(SCC)* e, portanto não há inicialmente componentes com *fator de integração tardia* nulos, uma vez que não é possível separar ou definir uma ordem dos componentes após o cálculo do *FIT*. Nesses casos, o componente de menor *FIT,* exercitará seu teste de integração anteriormente a execução dos testes dos demais componentes com fatores de integração tardia maiores na matriz de dependência do sistema.

- ***Tratamento de Deadlock:*** Um mesmo valor de *FIT* para mais de um componente pode significar que esses componentes possuem uma dependência entre si, existindo um ciclo. Nesse caso há a necessidade de se estabelecer uma prioridade entre as componentes com mesmo valor de *FIT*, sendo assim:

1. A interface pública do componente selecionado deve executar o menor número de operações e, conseqüentemente necessitar de um menor número de stubs, em comparação ao o número de *stubs* necessário aos outros componentes com o mesmo valor de *FIT*.

2. No caso dos componentes necessitarem da implementação do mesmo número de *stubs* e o acesso ao código do componente e/ou documentação seja viável existe a possibilidade de uma análise do código. Assim sendo, após análise do código, deverá ser testado o componente cujos *stubs* apresentarem a menor complexidade. Como no caso de um componente adquirido de terceiros o código fonte do componente bem como sua documentação, em geral, não está disponível, propõe-se que seja escolhido o componente que depois de testado sirva para diminuir os *stubs* necessários de um número maior de componentes do sistema.

3. Caso persista a coincidência pelo número de *stubs* propõe-se a escolha aleatória do componente a ser testado.

A cada interação na matriz para o cálculo de *FIT* determina-se a ordem de um ou mais componentes do sistema. Dado que o *FIT* é calculado como soma dos *FI's* de todas as operações implementadas pelos componentes que têm precedência direta sobre o componente alvo, essa ordem não determina apenas a ordem dos testes de integração mas também uma ordem para a implementação dos componentes bem como suas operações.

Desta forma os *FI's* correspondentes aos componentes cuja ordem já está definida devem ser removidos dos componentes restantes que possuam uma relação de dependência com este componente.

Portanto uma vez que se obtém a ordem dos testes pela ordem crescente dos valores de *FIT* e respeitando os critérios de precedência citados, determina-se a ordem para o componente C5. *(Figura 6.9)* e executa-se uma nova interação na matriz.

	C1			C2		C3		C4							C5		C6		C7				C8			FI	FIT 1a.	FIT 2a.
	c1M1	c1M2	c1M3	c2M1	c2M2	c3M1	c3M2	c1M1	c1M2	c1M3	c4M1	c8M1	c8M2	c8M3	c5M1	c5M2	c6M1	c6M2	c2M1	c2M2	c7M1	c7M2	c8M1	c8M2	c8M3		Interação	Interação
C1																											12	12
c1M1							x								x											2	8	8
c1M2											x					x										2	12	12
c1M3							x				x															2	8	8
C2																											18	18
c2M1												x		x							x		x		x	5	9	9
c2M2																					x	x				2	18	18
C3																											15	12
c3M1	x	x	x		x			x	x	x										x						8	6	6
c3M2		x			x				x											x						4	15	12
C4																											19	19
c1M1							x								x											2	8	8
c1M2											x					x										2	12	12
c1M3							x				x															2	8	8
c4M1				x	x												x	x	x	x						6	19	19
c8M1											x															1	10	10
c8M2						x					x															2	0	0
c8M3							x				x															2	10	10
C5																											6	
c5M1						x																				1	4	
c5M2							x										x	x								3	6	
C6																											9	6
c6M1																x					x					2	9	6
c6M2																						x				1	9	6
C7																											18	18
c2M1												x		x							x		x		x	5	6	6
c2M2																					x	x				2	18	18
c7M1																										0	16	16
c7M2																										0	5	5
C8																											10	10
c8M1											x															1	10	10
c8M2				x		x					x															3	0	0
c8M3							x				x															2	10	10

Figura 6.9: Matriz de Dependência Exemplo – Cálculo de FIT *(2^a Interação).*

Nesse ponto do método, para o exemplo em questão, tem-se que o componente C5 será o primeiro componente a ser testado. Segue-se recursivamente, calculando-se os *FIT's* e aplicando-

se, caso sejam necessários, os critérios de precedência/prioridade pré-definidos. Determina-se uma ordem, através do menor *FIT*, quando todas as interações possíveis na matriz foram executadas *(Figura 6.10)*.

O mesmo valor de *FIT* obtido para mais de um componente significa que estes componentes estão fortemente conectados ou em um ciclo de dependência, havendo um empate. Sendo assim, caso seja necessário, aplicam-se os critérios de precedência e então se define a ordem para os componentes com mesmo valor de *FIT*. No exemplo em questão obteve-se a seguinte ordem para os componentes: {C5, C6, C8, C3, C1, C2, C7, C4}.

	FI	FIT 1a. Interação	FIT 2a. Interação	FIT 3a. Interação	FIT 4a. Interação	FIT 5a. Interação	FIT 6a. Interação	FIT 7a. Interação	FIT 8a. Interação
C1		12	12	12	12	0			
c1M1	2	8	8	8	8	0			
c1M2	2	12	12	12	12	0			
c1M3	2	8	8	8	8	0			
C2		18	18	18	18	6	6		
c2M1	5	9	9	9	6	6	6		
c2M2	2	18	18	18	18	6	6		
C3		15	12	12	10				
c3M1	8	6	6	5	2				
c3M2	4	15	12	12	10				
C4		19	19	19	13	13	10	10	9
c1M1	2	8	8	8	8	0	0	0	0
c1M2	2	12	12	12	12	0	0	0	0
c1M3	2	8	8	8	8	0	0	0	0
c4M1	6	19	19	19	13	13	9	9	9
c8M1	1	10	10	10	10	0	10	10	5
c8M2	2	0	0	0	0	0	0	0	0
c8M3	2	10	10	10	10	10	10	10	5
C5		6							
c5M1	1	4							
c5M2	3	6							
C6		9	6						
c6M1	2	9	6						
c6M2	1	9	6						
C7		18	18	18	18	14	14	7	
c2M1	5	6	6	6	6	6	6	6	
c2M2	2	18	18	18	18	6	6	6	
c7M1	0	16	16	14	14	14	14	7	
c7M2	0	5	5	4	4	4	4	2	
C8		10	10	10					
c8M1	1	10	10	10					
c8M2	3	0	0	0					
c8M3	2	10	10	10					

Figura 6.10: Matriz de Dependência Exemplo – Interações para o cálculo de *FIT* .

6.3.6 Determinar os *Stubs* Necessários

A partir dessa ordem {C5, C6, C8, C3, C1, C2, C7, C4} definida na *Seção 6.3.5* será possível determinar quantos e quais *stubs específicos* e/ou *realísticos* das operações executadas pelos componentes *(interfaces requeridas)* serão necessários para os testes de integração respeitando-se a ordem obtida para os componentes. Para determinar os *stubs,* procede-se da seguinte forma:

Tomando como exemplo o componente C5 observa-se na matriz de dependência que as interfaces públicas c5M1() e c5M2() de C5 requerem a execução das operações c1M1(), c1M2(), do componente C1 e c6M1() do componente C6; portanto, necessita: (i) do *stub específico* para o componente **C1** e dentro de C1 um *stub* para a operação ***stub* (c1M1, c5M1),** ou seja, um *stub* da operação c1M1() de C1 para testar a operação c5M1() de C5, do *stub específico* ***stub* (c1M2, c5M2),** para testar a operação c5M2() de C5; (ii) do *stub específico* para o componente **C6** e dentro de C6 um do *stub específico* ***stub* (c6M1, c5M2),** ou seja, um *stub* da operação c6M1() de C6 para testar a operação c5M2() de C5.

Procede-se de forma análoga para os demais componentes, obtendo-se os resultados mostrados na *Figura 6.12.*

6.3.7 Considerações Finais

A *Figura 6.11* resume os resultados obtidos de alguns trabalhos estudados e descritos na *Seção 4* que utilizam o *ORD* proposto e usado por Kung et. al [13] e Tai e Daniels [33]; também utilizado para a exemplificação do método proposto nesse trabalho através do correspondente diagrama de classes.

Proposta	**Seqüência de Teste**	***Stubs* Específicos**
Tai e Daniels	{C1,	*Stub* (C3, C1)
	C5,	*Stub* (C6, C5)
	C3,	*Stub* (C8, C3)
	C6,	*Stub* (C4, C6)
	C8,	*Stub* (C2, C8)

	C4, C2, C7}	
Le Traon et. al (começando pelo nó G)	{C1, C8, C4, C5, C6,C3, C2, C7}	*Stub* (C3, C1) *Stub* (C2, C8) e Stub (C3, C8) *Stub* (C6, C5)
Briand et. al	{C1, C5, C3, C8, C4, C6, C2, C7}	*Stub* (C3, C1) *Stub* (C6, C5) *Stub* (C8, C3) *Stub* (C2, C8)
Lima e Travassos	{C1, C5, C8, C4, C3, C6, C2, C7}	*Stub* (C3, C1) *Stub* (C3, C8)

Figura 6.11: Comparação de Resultados Obtidos – Trabalhos estudados.

Uma vez que o método proposto nesse trabalho leva em consideração as dependências dinâmicas entre os componentes e as operações sobre suas interfaces, torna-se inviável a comparação do método proposto nesse trabalho com os trabalhos estudados que levam em consideração apenas dependências estáticas. Para o método proposto nesse trabalho obtiveram-se os resultados da *Figura 6.12*.

Proposta	**Seqüência de Testes**	***Stubs* Específicos**	**# de Stubs**
Método Proposto	{C5,	*Stub* de **C1** e dentro desse: *Stub* (c1M1, c5M1) *Stub* (c1M2, c5M2) *Stub* de **C6** e dentro desse: *Stub* (c6M1, c5M2).	2
	C6,	*Stub* de **C4** e dentro desse: *Stub* (c4M1, c6M1) *Stub* (c4M1, c6M2) Usa o componente C5 já testado.	1
	C8,	*Stub* de **C2** e dentro desse: *Stub* (c2M1, c8M1) *Stub* (c2M1, c8M3) *Stub* de **C7** e dentro desse: *Stub* (c2M1, c8M1) *Stub* (c2M1, c8M3)	2
	C3,	*Stub* de **C1** e dentro desse: *Stub* (c1M1, c3M2) *Stub* (c1M3, c3M2) *Stub* de **C4** e dentro desse: *Stub* (c1M1, c3M2) *Stub* (c1M3, c3M2) *Stub* (c8M2, c3M1) *Stub* (c8M3, c3M2)	2

	C1, C2, C7, C4}	Usa os componentes **C5** e **C8** já testados. Usa o componente **C3** já testado. Usa os componentes **C3, C4** e **C8** já testados. Usa o componente **C2, C3** e **C6** já testados. Usa o componente **C1, C2, C3, C7** e **C8** já testados.	
Quantidade total de *stubs*			**7**

***Figura 6.12*:** Resultados Obtidos – Método proposto.

Portanto serão necessários sete *stubs específicos* para testar o exemplo de acordo com a ordem obtida pelo método proposto nesse trabalho.

Observa-se que os resultados obtidos pela proposta de Le Traon et. al [69] são melhores que os resultados obtidos pela proposta de Tai e Daniels [33].

Uma vez que o resultado da proposta de Briand et. al é determinístico pois, aplicado o método proposto por eles obtém-se sempre a mesma ordem para as classes envolvidas no ORD, e resultado da proposta de Le Traon et. al não é, pois depende de qual classe é escolhida, para iniciar a aplicação do método e ainda, a proposta de Le Traon et. al só apresenta o resultado igual ao de Briand et. al quando a execução da proposta inicia-se pelo componente C7, o esforço de teste da proposta de Briand et. al pode ser considerado melhor que o apresentado por Le Traon et. al.

Lima e Travassos [18] refinam a proposta de Briand, e obtém o melhor resultado dentre os trabalhos estudados.

Os trabalhos apresentados anteriormente *(Seção 5)* usam dependências estáticas, o método proposto apresenta as relações dinâmicas entre métodos e consegue abranger quase todas as associações nesse tipo de análise, porém, não estão cobertas as associações em que um componente usa um atributo de um outro componente. Por exemplo, se um método c6M2() do componente C6 do exemplo instanciasse o componente C4 como parâmetro, essa relação não apareceria na matriz, pois se trata de uma relação de dependência entre um atributo e um método, o que não aparece na matriz de dependência, que trata somente das relações entre métodos.

Embora não sendo possível comparar o método proposto com os trabalhos estudados salienta-se que a proposta do método apresentado nesse trabalho também é determinística, diminui o número de *stubs,* considerando-se os *stubs* das operações dos componentes, e considera as dependências comportamentais dos componentes.

Uma característica marcante desse método é que ele indica não somente quais componentes precisam de *stubs* nos testes do componente alvo, mas também quais operações dentro desses componentes necessitam de *stubs*. Com isso o testador tem uma informação mais precisa para a realização dos testes.

7 Capítulo

Estudo de Caso

O estudo de caso escolhido para a demonstração do método proposto nesse trabalho em um sistema real foi o do controlador de caldeira de vapor [47].

A implementação, desenvolvida por Paulo Guerra et. al [47], é baseada no estilo C2 [50] onde o sistema é estruturado em camadas integradas com a utilização de conectores *(Seção 2.1*) onde as requisições fluem de baixo para cima e as resposta de cima para baixo na arquitetura.

O sistema é composto pelas camadas 1, 2, 3, as quais são integradas com os conectores conn1, conn2, conn3 e possui uma quarta camada composta por sensores e atuadores, que simulam peças de hardware de um ambiente real. Assim, conn3 é considerado um componente de fronteira com os componentes da camada 4 *(hardware)* bem como *BoilerController* (camada 1) é considerado um componente de fronteira responsável pela interação com o usuário. Demonstra-se a arquitetura do sistema através da *Figura 7.1.*

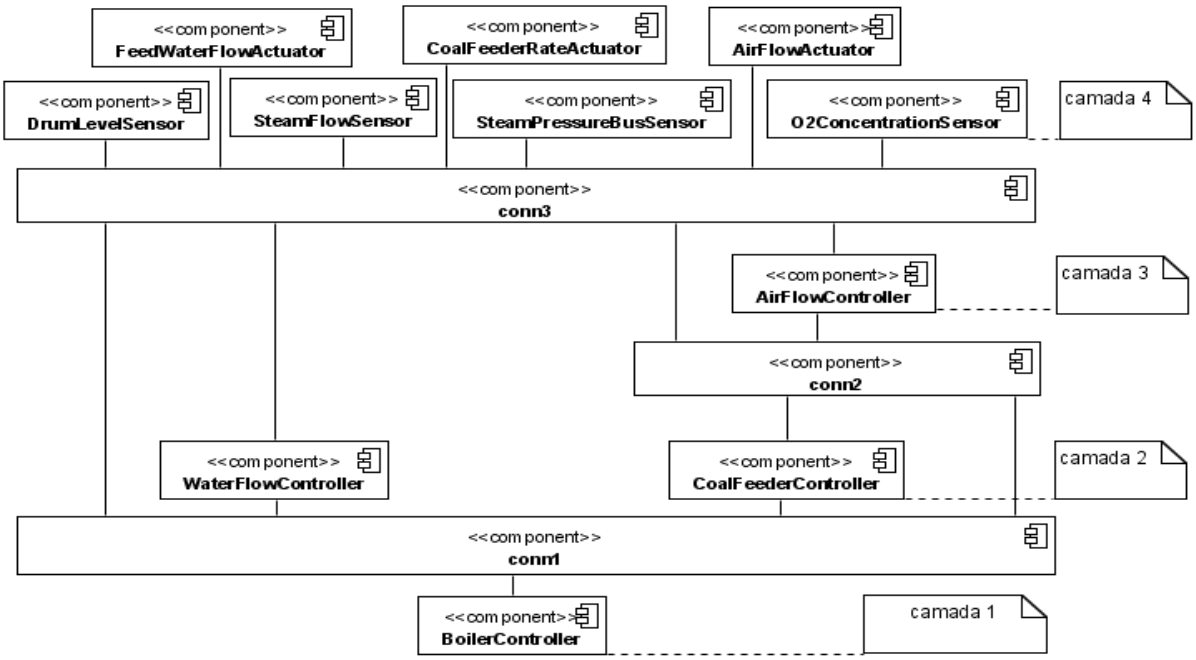


Figura 7.1: Arquitetura do controlador de caldeira a vapor.

7.1 Aplicação do Método

Com base nos passos referentes ao método proposto nesse trabalho *(Seção 6)* tem-se que o passo *um*, correspondente à obtenção da arquitetura do sistema, diz respeito à *Figura 7.1*.

O método proposto foca nas dependências comportamentais entre os componentes e as operações que ele executa *(interfaces requeridas*). Para a obtenção das relações dinâmicas entre os métodos públicos do estudo de caso em questão utilizou-se um relatório de dependências gerado pela ferramenta *dependency finder* [14].

Dependency Finder é um conjunto de ferramentas que analisa o *byte code* Java e permite a geração de relatórios que listam as dependências por pacotes, classes, métodos entre outros, e também relatórios que demonstram alterações de código ou elementos removidos do código (pacotes, classes, métodos, etc.). A *Figura 7.2* mostra um trecho do relatório[1] gerado para o sistema controlador de caldeira de vapor [47].

```
boilerSystem


  AirFlowActuator


    <-- boilerSystem.AirFlowController
    <-- boilerSystem.AirFlowController.actuator
    <-- boilerSystem.AirFlowController.connectTop(c2fw.IComponent)
    <-- boilerSystem.AirFlowController.setCoalFeedRate(double)
    <-- boilerSystem.AirFlowController.timeStep()
    <-- boilerSystem.Conn3
    <-- ftBoilerSystem
```

[1] Relatório gerado por Regina Moraes como parte dos estudos para confecção de um artigo.

```
    <-- ftBoilerSystem.AfcUpperDetector
    <-- ftBoilerSystem.AfcUpperDetector.actuator
    <-- ftBoilerSystem.AfcUpperDetector.connectTop(c2fw.IComponent)
    <-- ftBoilerSystem.AfcUpperDetector.setAirFlow(double)
    --> java.lang
    --> java.lang.Object

    setAirFlow(double)

       <-- boilerSystem.AirFlowController
       <-- boilerSystem.AirFlowController.setCoalFeedRate(double)
       <-- boilerSystem.AirFlowController.timeStep()
       <-- ftBoilerSystem
       <-- ftBoilerSystem.AfcUpperDetector
       <-- ftBoilerSystem.AfcUpperDetector.setAirFlow(double)

 AirFlowController
    --> boilerSystem.IAirFlowController
    --> c2fw
    --> c2fw.ComponentAbst
    AirFlowController()
```

Figura 7.2 **:** Trecho do relatório de dependências gerado para o sistema controlador de caldeira de vapor (*Dependency Finder).*

Com base nesse relatório obtêm-se as matrizes de dependências bem como o cálculo de *FI* para os componentes do sistema *(Apendice1)*.

7.2 Resultados Obtidos

De acordo com os passos mostrados nas *Seções 6.3.4 e 6.3.5* e aplicados os critérios de precedência prioridade entre os componentes cujos valores de *FIT* resultaram em empate, foi então obtida a ordem de integração mostrada na *Figura 7.3*. De acordo com o *Apêndice 2* observa-se que:

- No cálculo de *FI*, obteve-se para o componente StartUp um *FI=0* o que significa que este será o ultimo componente a ser testado.
- Na primeira interação obteve-se para os componentes *AirFlowActuator, CoalFeederRateActuator, Conn3, DrumLevelSensor, FeedWaterFlowActuator, IAirFlowController, IController, O2ConcentrationSensor, PIDController, SteamFlowSensor e SteamPressureBusSensor*, *FIT=0*. Para definir-se a ordem dentre esses componentes foi aplicado o critério de precedência identificado por ***Tratamento de Deadlock*** *(Seção 6.3.5)*.
- Na segunda interação obteve-se para os componentes *WaterFlowActuator, Conn2, CoalFeederController, BoilerController, AirFlowController*, *FIT=0*. Para definir-se a ordem dentre esses componentes foi aplicado o critério de precedência identificado por ***Tratamento de Deadlock*** *(Seção 6.3.5)*.

Sistema	**Seqüência de Teste**	***Stubs* Específicos**	**# de *Stubs***
BoilerSystem	{PIDConroller,	Não há.	0
	IController,	Não há.	0
	IAirFlowController,	Não há.	0
	AirFlowActuator,	Não há.	0
	CoalFeederRateActuator,	Não há.	0

	Conn3,	Não há.	0
	DrumLevelSensor,	Não há.	0
	FeedWaterFlowActuator,	Não há.	0
	SteamFlowSensor,	Não há.	0
	SteamPressureBussSensor,	Não há.	0
	O2ConcentrationSensor,	Não há.	0
	CoalFeederController,	usa CoalFeederRateActuator, *PIDController e* SteamPressureBussSensor já testados.	0
	WaterFlowController,	usa *DrumLevelSensor*, *FeedWaterFlowActuator*, *PIDConroller* e *SteamFlowSensor* já testados.	0
	Conn2,	usa *Conn3*, *IAirFlowConroller* e *IController* já testados.	0
	BoilerController, AirFlowConroller,	usa *IAirFlowConroller* e *IController* já testados.	0
	Conn1,	usa *AirFlowActuator, O2ConcentrationSensor* e *PIController* já testados.	0
	StartUp}	usa *CoalFeederController, CoalFeederRateActuator, Conn2, IController,* e *WaterFlowController* já testados.	0
		usa *AirFlowController, BoilerController, CoalFeederController, CoalFeederRateActuator, Conn2, Conn3, IController, SteamPressureBusSensor*	

		eWaterFlowController já testados.	
Total de *Stubs*			0

Figura 7.3*:* Resultados obtidos – Sistema controlador de Caldeira de Vapor.

Capítulo 8

Conclusões

Nesse trabalho foram apresentados os fundamentos básicos para o desenvolvimento baseado em componentes que vem sendo cada vez mais utilizado hoje em dia devido à reutilização de software.

No contexto de testes de integração foram apresentados alguns problemas associados à reutilização de componentes, pois um componente implementado para um determinado contexto e reaproveitado em um outro com especificações diferentes, pode comprometer a construção de um novo sistema confiável baseado em componentes uma vez que as falhas devem ser encontradas não só nos componentes, mas na interação entre eles.

Alguns trabalhos foram estudados com o intuito de se propor um método visando diminuir o número de *stubs* necessários e minimizar o esforço empenhado durantes os testes de integração.

Com base nesses trabalhos e após alguns refinamentos dos mesmos esse trabalho apresentou um método que trata das relações dinâmicas entre métodos para definir uma relação de dependências entre os componentes do sistema. Aplicados os passos com base na arquitetura do sistema e com a ausência do código fonte obtém-se uma ordem para os componentes bem como, quais e quantos *stubs* serão necessários durante os testes de integração.

7.3 Contribuições

O método proposto nesse trabalho trata-se de um refinamento das propostas de Stafford et. al [27] e Lima e Travassos [18].

Stafford et. al trata da relação *causal* entre métodos e sugerem uma matriz de dependências que identifica os elementos que *afetam* e os que são *afetados* por outros.

Lima e Travassos sugerem o cálculo dos *FI's* e *FIT's* para definir uma ordem para os testes de integração baseando-se nas relações estruturais entre as classes em um diagrama de classes.

O método proposto monta uma matriz de dependência conforme proposta de Stafford et. al e usa o cálculo de *FI* e *FIT* em relações comportamentais entre as operações do componente e uma vez que o cálculo de *FIT* é diretamente proporcional ao número de operações, das quais o componente alvo depende esse método sugere o uso do maior *FIT* encontrado entre as operações do componente para determinar o *FIT* do componente. Assim o método proposto nesse trabalho determina que o *FIT* de um componente seja o maior *FIT* dos métodos desse componente. Além disso, esse método aprofunda a proposta de Stafford et. al e identifica os *stubs* de uma operação dinâmica para outra operação dinâmica entre os componentes do sistema.

7.4 Trabalhos Futuros

- Desenvolver uma ferramenta para automatizar as interações na matriz de dependências para que o ganho com aplicação do método em relação ao esforço empenhado na fase de testes de integração seja aumentado e para que seja possível uma análise mais profunda do método em um sistema maior.
- Analisar melhor a existência de ciclos e dependências, pois o método proposto desceu em um nível de abstração mais baixo que as propostas estudadas uma vez que lida com relações dinâmicas entre métodos e as propostas estudadas lidam com relações estáticas entre componentes.
- Verificar o comportamento do método proposto se for incluída na matriz a relação entre atributos e operações.

Referências Bibliográficas

[1] A. Silberschatz; P. Galvin and G. Cagne. *Sistemas Operacionais. Conceitos e aplicações.* Editora Campus-2001.

[2] A. P. Mathur and W. E. Wong. *An Empirical Comparison of data Flow and Mutation Based test*
adequacy criteria. The Journal of Software Testing, Verification, and Reliability, v.4, número 1, p. 9-31, 1994.

[3] A. M. R. Vincenzi. *Orientação a Objetos: Definição, Implementação e Análise de Recursos de Teste e Validação*. Dissertação de mestrado - Instituto de Ciências Matemáticas e de Computação - ICMC/USP, 2000.

[4] B. Beizer. *Software System Testing and Quality Assurance*. Van Nostrand Reinhold Company Inc, 1984.

[5] B. Beizer. *Software Testing Techniques*. International Thompson Computer Press, 2ª. Edição, 1997.

[6] C. R. Rocha. *Um Método de Testes para Componentes Tolerantes a Falhas.* Dissertação de Mestrado, IC, Unicamp, Novembro 2005.

[7] C. Szyperski . *Component Software: Beyond Object-Oriented Programming*. Addison Wesley, 1997.

[8] C. M. L Werner e R. M. M. Braga. *Desenvolvimento Baseado em Componentes.* XIV SBES - Mini-Cursos/ Tutoriais Anais. João Pessoa, 2000.

[9] C. Atkinson, J. Bayer and D. Muthing. *Component-Based Product Line Development: The Kobra Approach.* In: P. Donohoe, editor, Proceedings of The First Software Product Line Conference, pp.289-309, 2000.

[10] C. Pfister. *A Case Study using Black Box Components*. http://www.oberon.ch/docu/case_study/index.htlm. acesso em: 24/10/2005

[11] D. Garlan and R. Allen. *A Formal Basis for Architectural Connection. ACM Transations On Software Engeneering and Methodology*, 1997, pp. 213-249.

[12] D. Garlan, R. Monroe and D. Wile. ACME: *An Architecture Description Interchange Language*. In Proceedings of CASCON'97. IBM Center for Advanced Studies, 1997.

[13] D. Kung, J. Gao, P. Hsia, J. Lin and Y. Toyoshima, *Class Firewall, Test Order, and Regression Testing Object-Oriented Programs,* J. Object-Oriented Programming, 1995/Vol.8, número 2, p. 51-65

[14] Ferramenta Dependency Finder, **http://depfind.sourceforge.net/,** 2006.

[15] D. d'Souza and A. C. Will, *Objects, Components e Frameworks with UML: The Catalysis Approach.* Addison Wesley, 1998.

[16] D. S. Rosenblum. *Adequate Testing of Component-based Software*. Relatório Técnico UCI-ICS-97-34, University of. California, Irvine, CA, 1997.

[17] E. Martins e C. M. Toyota. *Construção de Classes Autotestáveis*. In: VIII SCTF Simpósio de Computação Tolerante a falhas, Campinas, SP, pp.196-209, 1999.

[18] G. M. P. S. Lima e G. H. Travassos. *Testes de Integração Aplicados a Software Orientados a Objetos: Heurística para Ordenação de Classes,* Relatório Técnico, 2004, ES-632/04, COPPE/UFRJ.

[19] G. Myers. *The Art of Software Testing*. John Wiley and Sons, 1979.

[20] H. Oliveira. *Construção de um Componente Genérico Baseado em Heurísticas para Ordenação das Classes em Ordem de Prioridade de Teste de Integração*, Estudo do Laboratório de Engenharia de Software, COPPE/UFRJ, 2003.

[21] H. Ural and B. Yang. *A Structural Test Selection Criterion.* Information Processing Letters, Vol. 28, p. 157-163, 1988.

[22] I. Sommerville. *Software Engineering*. Addison Wesley . 6th Edition, 2001

[23] I. Jacobson, J. Rumbaugh and G. Booch. *Unified Software Development Process.* Addison Wesley, 1999.

[24] I. Perez, E. Martins, R. Moraes, J. A.Cardoso e R. F. Soliani. *Um Processo para Teste de Sistemas Baseado em Componentes com Reuso de Componentes.* Relatório Técnico. IC – Unicamp, 2005.

[25] J.A. Stafford, A.L. Wolf and M. Caporuscio. *The Application of Dependence Analysis to Software Architecture Descriptions.* IEEE Transactions on Software Engineering, pp. 52-62, 2003.

[26] J. A. Stafford, D. J. Richardson and A. L. Wolf; Alladin: *A Tool for Achitecture-Level Dependence Analysis of Software Systems.* Technical Report CU-CS-858-98, Department of Computer Science, University of Colorado, Boulder, Colorado, 1998.

[27] J. A. Stafford, D. J. Richardson and A.L. Wolf. *Architecture-Level Dependence Analysis for Software Systems*. Technical Report CCR-97-10078, Department of Computer Science, University of Colorado, Boulder, Colorado, 1998.

[28] J. A. Stafford and A.L. Wolf. *Architecture-Level Dependence Analysis in Support of Software Maintenance.* In Proceedings of the Third International Software Architecture Workshop, p. 129-132, 1998.

[29] J. Bishop and R. Faria. *Connectors in Configuration Programming Languages: are They Necessary?*. IEEE Third International Conference on Configurable Distributed System, 1996.

[30] J. Chessman and J. Daniel. *UML Components - A Simple Process for Specifying Component-Based Software,* 1992.

[31] J. C. Maldonado, A. M. Vincenzi, E. F. Barbosa, S. R. S. Sousa e M. E. Delamaro. *Aspectos teóricos e empíricos de teste de cobertura de software.* Relatório Técnico 31, Instituto de Ciências Matemáticas e de Computação -ICMC-USP, 1998.

[32] J. C. Maldonado. *Critérios Potenciais Usos: Uma contribuição ao teste estrutural de software.* Tese de Doutorado, DCA/FEE/UNICAMP, Campinas, SP, 1991.

[33] K. C. Tai and F. J. Daniels. *Interclass Test Order for Object-Oriented Software.* Journal of Object-Oriented Programming, 1999/Vol.12, número 4, p. 18-25.

[34] L. Bass, P. Clements and R. Kasman. *Software Architecture in Pratice.* SEI-Series in Software Engineering, 2002.

[35] L. Feijs. *Architecture visualization and analysis: Motivation and example*. In International Workshop on Development and Evolution of Software Architectures for Product Families, 1996.

[36] L. C. Briand, Y. Labiche and Yihong Wang. *An Investigation of Graph-Based Class Integration Test Order Strategies ,*IEEE Transactions on Software Engineering, 2003/29(7): 594-607, 2003.

[37] M. Bertrand. *On to Components.* IEEE Computer, 32(1):139-140, 1999.

[38] M. Bertrand. *Object-Oriented Software Construction.* Prentice-Hall, 1997.

[39] M. J. Harrold. *Testing: A Roadmap.* In: 22th International Conference on Software Engineering - Future of SE Track, 2000, pp. 61-72.

[40] M. J. Harrold, D. Liang and S. Sinha. *An Approach to Analyzing and Testing Component-based Systems.* In: First Workshop um Testing Distributed Component-Based System at. ICSE'1999, CA: IEEE Computer Society Press, 1999.

[41] M. Lorenz and J. Kidd. *Object-Oriented Metrics: A Pratical Guide.* Prentice Hall, 1994.

[42] M. Silva. *COSMOS - Um Modelo de Estruturação de Componentes para Sistemas Orientados a Objetos.* Dissertação de Mestrado, IC, Unicamp, 2003.

[43] M. Shaw and D. Garlan. *Software Architecture - Perspective in a Emerging Discipline.* Upper Saddle River: Prentice Hall, 1996.

[44] Object Management Group. The OMG's corba website. http://www.corba.org/, 2004.

[45] O. Nierstrasz and D. Tsichritzis. *Object-Oriented Software Composition.* The Object-Oriented Serie; Prentice-Hall, 1995.

[46] P. H. S. Brito, M.A. M. Barbosa, P. A. C. Guerra e C. M. F. Rubira. *Um Processo para o Desenvolvimento Baseado em Componentes com Reuso de Componentes.* Relatório Técnico. IC 05-22, 2005.

[47] P. A. de C. Guerra, C. M. F. Rubira, A. B. Romanosvsky e R. Lemos. *A dependable architecture for cots-based software system using protective wrappers.* In Rogério Lemos, C. Gacek e A. B. Romanosvsky, editores , *WADS*, volume 3069 de *Lecture Notes in computer Science*, pp. 144-166. Springer, 2003.

[48] R. Allen and D. Garlan. *A Formal Basis for Architecture Connection.* ACM transactions on Software Engineering and Methodology. 6(3):213:249, 1997.

[49] RAPIDE Design Team. *Draft: Rapide 1.0 Architecture Language Reference Manual*, 1997.

[50] R. N. Taylor, N. Medvidovic, K. M. Anderson, Jr. E. James White-head, J. E. Robbins, K. A. Nies, P. Oreizy and D. L. Dubrow. A component and message-based architectural style for gui software. *IEEE transactions on Software Engineering*, 22(6):390-406, 1996.

[51] R. Mili, A. Mili and R. T. Mittermeir. *Storing and Retrieving Software Components: A Refinement Based System.* IEEE Trans. on Software Engineering. 23(7):445-460, 1997.

[52] R. T. Monroe, A. Kompanek; R. Melton; D. Garlan. *Architetural Styles, Design Patterns and Objects*. IEEE Software, 1997.

[53] R. V. Binder. *"Design for Testability in Object-Oriented System"*. Communications of the ACM, 37(9):87-101, DLP, http://dblp.uni-trier.de/, 1994.

[54] R. V. Binder. *Testing Object-Oriented System: Models, Patterns, and Tools.* Addison Wesley, 2000.

[55] R. S. Pressman. *Software Engineering - a Practitioner's Approach*. McGraw-Hill, 5ª. edição, 2001.

[56] R. Tarjan. *Depth-First Search and Linear Graph Algorithms.* SIAM J. Computing. 1(2):146-160, 1972.

[57] Sun Microsystems Inc. *Enterprise Javabeans Technology*. http://www.java.sun/products/ejb, 2004.

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

[59] S. Siegel. *Object-Oriented Software Testing: a Hierarchical Approach*. John Wiley & Sons, 1ª. Edição, New York, 1996.

[60] T. Anderson, M. Feng, S. Riddle and A. B. Romanosvsky. Protective wrapper development: a case study. In: M. Hakan Erdogmus e Tao Weng, editors, ICCFSS, volume 2580 de Lecture Notes in Computer Science, pp.1-14, Springer, 2003.

[61] T. E. Colanzi. *Uma abordagem Integrada de desenvolvimento e Teste de Software Baseada em UML*. Dissertação de Mestrado, ICMC-USP, São Carlos-SP, 1999.

[62] U. Linnenkugel, M. Müllerburg. *Test Data Selection Criteria for (Software) Integration Testing.* In: First International Conference on System Integration, Morristown, pp. 709-717, NJ, 1990.

[63] V. A. Pagano. *Uma Abordagem Arquitetural com Tratamento de Exceções para Sistemas de Software Baseados em Componentes.* Dissertação de Mestrado Profissional, IC, Unicamp, 2004.

[64] W. Wolfgang, J. Bosch and C. Szypersky. Summary of the Second International Workshop on Component-Oriented Program (WCOP'97). http://www.abo.fi/~Wolfgang.Weck/WCOP/97/Summary.html. - acesso em: 24/10/2005

[65] W. E. Howden. *Theorical and Empirical Studies of Program Testing.* IEEE Transactions on Software Engineering, v. 4, número 4, p. 293-298, 1978.

[66] W. de P. P. Filho. *Engenharia de Software: Fundamentos, Métodos e Padrões*. Livros

Técnicos e Científicos. 1ª. Edição, 2001.

[67] Wirfs-Brock, R. J. and R.E. Johnson. *Surveying current research in object-oriented design.* Communications of the ACM, 33(9): 04-124, 1990.

[68] W. R. Adrion, M. A. Branstad and J. C. Cherniavisky. *Validation, Verification, and Testing of Computer Software.* ACM Computing Surveys, 14(2):159-192, 1982.

[69] Y. L. Traon, T. Jéron, J. Jézéquel and P. Morel. *Efficient Object-Oriented Integration and Regression Testing.* IEEE Transations Reliability, 49(1):12-25, 0018-9529/00,2000.

Apêndice 1 – Matrizes de Dependências e Cálculo de FI.

	AirFlow Actuator	AirFlowController				BoilerController		
	setAirFlow	AirFlowController	setCoalFeedRate	setConfiguration	timeStep	BoilerController	setConfiguration	timeStep
AirFlowActuator								
setAirFlow			X		X			
AirFlowController								
AirFlowController()								
setCoalFeedRate								
setConfiguration								
timeStep								
BoilerController								
BoilerController()								
setConfiguration								
timeStep								
CoalFeederController								
CoalFeederController()								
setConfiguration								
timeStep								
CoalFeederRateActuator								
setCoalFeedRate								
Conn1								
Conn1()								
setConfiguration								
timeStep								
Conn2								
Conn2()								
readBussPressure								
setCoalFeedRate								
setConfiguration								
timeStep								
Conn3								
Conn3()								
readBussPressure								
readDrumLevel								
readO2Concentration								
readSteamFlow								
setAirFlow								
setCoalFeedRate								
setWaterFlow								
DrumLevelSensor								
readDrumLevel								
FeedWaterFlowActuator								
setFeedWaterFlow								
IAirFlowController								
setCoalFeedRate							X	
IController								
setConfiguration							X	
timeStep								X
O2ConcentrationSensor								
readO2Concentration		X			X			
PIDController								
PIDController()		X						
controlInputA			X					
controlInputB					X			
StartUp								
StartUp()								
main								
SteamFlowSensor								
readSteamFlow								
SteamPressureBusSensor								
readBusPressure								
WaterFlowController								
WaterFlowController()								
timeStep								

	CoalFeederController			CoalFeederRateActuator	Conn1			
	CoalFeederController	setConfiguration	timeStep	SetCoalFeedRate	Conn1	connectTop	setConfiguration	timeStep
AirFlowActuator								
setAirFlow								
AirFlowController								
AirFlowController()								
setCoalFeedRate								
setConfiguration								
timeStep								
BoilerController								
BoilerController()								
setConfiguration								
timeStep								
CoalFeederController								
CoalFeederController()								
setConfiguration							X	
timeStep								X
CoalFeederRateActuator								
setCoalFeedRate			X					
Conn1								
Conn1()								
setConfiguration								
timeStep								
Conn2								
Conn2()								
readBussPressure								
setCoalFeedRate								
setConfiguration							X	
timeStep								X
Conn3								
Conn3()								
readBussPressure								
readDrumLevel								
readO2Concentration								
readSteamFlow								
setAirFlow								
setCoalFeedRate								
setWaterFlow								
DrumLevelSensor								
readDrumLevel								
FeedWaterFlowActuator								
setFeedWaterFlow								
IAirFlowController								
setCoalFeedRate								
IController								
setConfiguration							X	
timeStep								X
O2ConcentrationSensor								
readO2Concentration								
PIDController								
PIDController()	X							
controlInputA								
controlInputB			X					
StartUp								
StartUp()								
main								
SteamFlowSensor								
readSteamFlow								
SteamPressureBusSenso								
readBusPressure			X					
WaterFlowController								
WaterFlowController()								
timeStep								X

	Conn2				
	Conn2	readBusPressure	setCoalFeedRate	setConfiguration	timeStep
AirFlowActuator					
setAirFlow					
AirFlowController					
AirFlowController()					
setCoalFeedRate					
setConfiguration					
timeStep					
BoilerController					
BoilerController()					
setConfiguration					
timeStep					
CoalFeederController					
CoalFeederController()					
setConfiguration					
timeStep					
CoalFeederRateActuator					
setCoalFeedRate					
Conn1					
Conn1()					
setConfiguration					
timeStep					
Conn2					
Conn2()					
readBussPressure					
setCoalFeedRateActuator					
setConfiguration					
timeStep					
Conn3					
Conn3()					
readBussPressure		x			
readDrumLevel					
readO2Concentration					
readSteamFlow					
setAirFlow					
setCoalFeedRate			x		
setWaterFlow					
DrumLevelSensor					
readDrumLevel					
FeedWaterFlowActuator					
setFeedWaterFlow					
IAirFlowController					
setCoalFeedRate			x		
IController					
setConfiguration				x	
timeStep					x
O2ConcentrationSensor					
readO2Concentration					
PIDController					
PIDController()					
controlInputA					
controlInputB					
StartUp					
StartUp()					
main					
SteamFlowSensor					
readSteamFlow					
SteamPressureBusSensor					
readBusPressure					
WaterFlowController					
WaterFlowController()					
timeStep					

	Conn3							
	Conn3	readBusPressure	readDrumLevel	readO2Concentration	readSteamFlow	setAirFlow	setCoalFeedRate	setFeedWaterFlow
AirFlowActuator								
setAirFlow								
AirFlowController								
AirFlowController()								
setCoalFeedRate								
setConfiguration								
timeStep								
BoilerController								
BoilerController()								
setConfiguration								
timeStep								
CoalFeederController								
CoalFeederController()								
setConfiguration								
timeStep								
CoalFeederRateActuator								
setCoalFeedRate								
Conn1								
Conn1()								
setConfiguration								
timeStep								
Conn2								
Conn2()								
readBussPressure								
setCoalFeedRate								
setConfiguration								
timeStep								
Conn3								
Conn3()								
readBussPressure								
readDrumLevel								
readO2Concentration								
readSteamFlow								
setAirFlow								
setCoalFeedRate								
setWaterFlow								
DrumLevelSensor								
readDrumLevel								
FeedWaterFlowActuator								
setFeedWaterFlow								
IAirFlowController								
setCoalFeedRate								
IController								
setConfiguration								
timeStep								
O2ConcentrationSensor								
readO2Concentration								
PIDController								
PIDController()								
controlInputA								
controlInputB								
StartUp								
StartUp()								
main								
SteamFlowSensor								
readSteamFlow								
SteamPressureBusSensor								
readBusPressure								
WaterFlowController								
WaterFlowController()								
timeStep								

	DrumLevel Sensor	FeedWater FlowActuator	IAirFlow Controller	IController		O2Concentration Sensor	PIDController			StratUp	
	readDrumLevel	setFeedWaterFlow	setCoalFeedRate	setConfiguration	timeStep	readO2Concentration	PIDController	controlInputA	controlInputB	StartUp	main
AirFlowActuator											
setAirFlow											
AirFlowController											
AirFlowController()											X
setCoalFeedRate											
setConfiguration											
timeStep											
BoilerController											
BoilerController()											X
setConfiguration											
timeStep											
CoalFeederController											
CoalFeederController()											X
setConfiguration											
timeStep											
CoalFeederRateActuat											
setCoalFeedRate											
Conn1											
Conn1()											X
setConfiguration											
timeStep											
Conn2											
Conn2()											X
readBussPressure											
setCoalFeedRate											
setConfiguration											
timeStep											
Conn3											
Conn3()											X
readBussPressure											
readDrumLevel											
readO2Concentration											
readSteamFlow											
setAirFlow											
setCoalFeedRate											
setWaterFlow											
DrumLevelSensor											
readDrumLevel											
FeedWaterFlowActuat											
setFeedWaterFlow											
IAirFlowController											
setCoalFeedRate											
IController											
setConfiguration											X
timeStep											X
O2ConcentrationSenso											
readO2Concentration											
PIDController											
PIDController()											
controlInputA											
controlInputB											
StartUp											
StartUp()											
main											
SteamFlowSensor											
readSteamFlow											
SteamPressureBusSen											
readBusPressure											X
WaterFlowController											
WaterFlowController()											X
timeStep											

	SteamFlow Sensor	SteamPressure BusSensor	WaterFlowController		FI
	readStreamFlow	readBusPressure	WaterFlowController	timeStep	
AirFlowActuator					
setAirFlow					2
AirFlowController					
AirFlowController()					1
setCoalFeedRate					2
setConfiguration					0
timeStep					0
BoilerController					
BoilerController()					1
setConfiguration					0
timeStep					0
CoalFeederController					
CoalFeederController()					1
setConfiguration					1
timeStep					1
CoalFeederRateActuator					
setCoalFeedRate					1
Conn1					
Conn1()					1
setConfiguration					0
timeStep					0
Conn2					
Conn2()					1
readBussPressure					0
setCoalFeedRate					0
setConfiguration					1
timeStep					1
Conn3					
Conn3()					1
readBussPressure					1
readDrumLevel					0
readO2Concentration					0
readSteamFlow					0
setAirFlow					0
setCoalFeedRate					1
setWaterFlow					0
DrumLevelSensor					
readDrumLevel				x	1
FeedWaterFlowActuator					
setFeedWaterFlow				x	1
IAirFlowController					
setCoalFeedRate					1
IController					
setConfiguration					4
timeStep					4
O2ConcentrationSensor					
readO2Concentration					2
PIDController					
PIDController()			x		3
controlInputA				x	2
controlInputB				x	3
StartUp					
StartUp()					0
main					0
SteamFlowSensor					
readSteamFlow				x	1
SteamPressureBusSensor					
readBusPressure					2
WaterFlowController					
WaterFlowController()					1
timeStep					1

Apêndice 2 – Cálculo de FI e FIT.

	FI	FIT - 1a. Interação	FIT - 2a. Interação	FIT - 3a. Interação
AirFlowActuator		**0**		
setAirFlow	2	0		
AirFlowController		**7**	**0**	
AirFlowController()	1	5	0	
setCoalFeedRate	2	4	0	
setConfiguration	0	0	0	
timeStep	0	7	0	
BoilerController		**5**	**0**	
BoilerController()	1	0	0	
setConfiguration	0	5	0	
timeStep	0	4	0	
CoalFeederController		**6**	**0**	
CoalFeederController()	1	3	0	
setConfiguration	1	0	0	
timeStep	1	6	0	
CoalFeederRateActuator		**0**		
setCoalFeedRate	1	0		
Conn1		**7**	**3**	**1**
Conn1()	1	0	0	0
setConfiguration	0	6	2	0
timeStep	0	7	3	1
Conn2		**4**	**0**	
Conn2()	1	0	0	
readBussPressure	0	1	0	
setCoalFeedRate	0	2	0	
setConfiguration	1	4	0	
timeStep	1	4	0	
Conn3		**0**		
Conn3()	1	0		
readBussPressure	1	0		
readDrumLevel	0	0		
readO2Concentration	0	0		
readSteamFlow	0	0		
setAirFlow	0	0		
setCoalFeedRate	1	0		
setWaterFlow	0	0		
DrumLevelSensor		**0**		
readDrumLevel	1	0		
FeedWaterFlowActuator		**0**		
setFeedWaterFlow	1	0		
IAirFlowController		**0**		
setCoalFeedRate	1	0		
IController		**0**		
setConfiguration	4	0		
timeStep	4	0		
O2ConcentrationSensor		**0**		
readO2Concentration	2	0		
PIDController		**0**		
PIDController()	3	0		
controlInputA	2	0		
controlInputB	3	0		
StartUp				
StartUp()	0			
main	0			
SteamFlowSensor		**0**		
readSteamFlow	1	0		
SteamPressureBusSensor		**0**		
readBusPressure	2	0		
WaterFlowController		**8**	**0**	
WaterFlowController()	1	3	0	
timeStep	1	8	0	