Um Método de Teste Funcional para Verificação de Componentes

Carina Machado de Farias

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Informática da Universidade Federal de Campina Grande como parte dos requisitos necessários para obtenção do grau de Mestre em Informática.

Área de Concentração: Ciência da Computação

Linha de Pesquisa: Engenharia de Software

Patrícia Duarte de Lima Machado

(Orientadora)

Campina Grande, Paraíba, Brasil

FARIAS, Carina Machado de

F224M

Um Método de Teste Funcional para Verficação de Componentes

Dissertação de Mestrado, Universidade Federal de Campina Grande, Centro de Ciências e Tecnologia, Coordenação de Pós-Graduação em Informática, Campina Grande, PB, Fevereiro de 2003.

114p. Il.

Orientadora: Patrícia Duarte de Lima Machado

1. Engenharia de Software
2. Teste Funcional
3. Componentes de Software
4. Orientação a Objetos

CDU – 519.683

Resumo

O interesse no desenvolvimento de software baseado em componentes tem crescido substancialmente devido à promessa de redução de custos e tempo de desenvolvimento através do reuso. A maioria das metodologias existentes tem se concentrado nas fases de análise e projeto. Entretanto, o reuso efetivo de componentes está diretamente relacionado à confiabilidade dos mesmos. O principal objetivo deste trabalho é propor um método de teste funcional aplicável a componentes de software. Já que o processo de desenvolvimento utilizado influencia significativamente a testabilidade dos sistemas, o método de teste proposto é apresentado dentro de um processo de desenvolvimento de componentes bem definido - Componentes UML. Artefatos de teste são gerados a partir de especificações em UML (Unified Modelling Language), especialmente a partir dos diagramas de seqüência. Um estudo de caso foi desenvolvido para ilustrar a aplicabilidade do método.

Abstract

Interest in component-based software development has increased significantly due its promise to reduce development costs and time through reuse. The majority of existing methodologies has focus in the analysis and design disciplines. Nevertheless, effective reuse of components is closely related to their reliability. The main goal of this work is to propose a method of functional testing to verify software components. Since testability of systems is greatly influenced by the development process chosen, the proposed method is integrated with a well-defined component development process. Test artifacts are generated from UML (Unified Modelling Language) specifications, specially sequence diagrams. A case study has been developed to illustrate the applicability of the method.

Agradecimentos

À professora Rita Suzana Pitangueiras e ao grande amigo José Amancio Macedo Santos pelo incentivo em ingressar no programa.

Aos professores e amigos Carlo Tolla (UFRJ), Simone Branco e Rita Suzana (Faculdade Ruy Barbosa) pelas cartas de recomendação.

A minha orientadora Patrícia Duarte de Lima Machado pelo grande empenho, dedicação, apoio, incentivo, amizade e confiança.

Aos meus pais Maria Denideth Machado e Carlos Guilherme Farias pelo amor, amizade e admiração que sempre tiveram por mim.

A minhas irmãs Tati, Cris, Renata, Binha, Deni, Giovana e Nildete pelo incentivo constante.

A minha segunda família, Mercês, Simone e Bárbara pelo apoio incondicional.

Aos funcionários do DSC pela boa vontade e disponibilidade no atendimento, em especial a Aninha e Zeneide.

Aos professores do DSC por todos os conhecimentos transmitidos, especialmente a Jacques, Jorge, Patrícia e Dalton.

Aos colegas do Labpetri (Edna, Érica, Amancio, Sandro, Cássio, Emerson, Rodrigo, Paulo, Taciano, Loreno e Marcelino) pelas discussões e momentos de descontração.

A todos os colegas de mestrado, em especial, Amancio, Rico, Petrônio, Alexandre, Alisson, Cidinha, Castro e Edna pela companhia, amizade e disponibilidade em ajudar.

Aos amigos do Rubi (Rico, Tiago, Daniel, Rodrigo, Renato e Ênio) e do Jade (Ricardo e Ramon) pela amizade e pelas inúmeras horas de alegria.

A todos os outros amigos que conheci na Paraíba e que levarei na minha lembrança para o resto da vida, em especial, Geraldo, Pergentino, Felipe, Tayana, Flavius, Luis e Milton.

Agradecimentos mais do que especiais aos meus grandes amigos de Salvador, Mercês, Luiz e Joe, por estarem sempre ao meu lado nos momentos em que a saudade de casa apertava.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A crescente preocupação com o aumento dos custos envolvidos no processo de desenvolvimento de software tem motivado pesquisadores e setores da indústria a desenvolver novas tecnologias capazes de produzir código eficiente e fácil de manter e de compreender, com recursos humanos e de tempo limitados.

Nesse sentido, tecnologias centradas no reuso de software têm sido alvo de pesquisa e de investimentos, já que o reuso favorece efetivamente a amortização dos custos do desenvolvimento do software entre seus usuários, além de possibilitar a redução no tempo de desenvolvimento.

Dentre os paradigmas de desenvolvimento que apresentam o reuso como uma de suas principais características, a engenharia de software baseada em componentes é uma das abordagens que vem mais crescentemente sendo adotada no desenvolvimento de software, tendo em vista que trata-se de um paradigma capaz de combinar unidades pré-desenvolvidas, com o objetivo de reduzir o tempo de desenvolvimento, facilitar o acoplamento de novas funcionalidades, ou a mudança de funcionalidades já existentes, e ainda promover o reuso de partes do software.

Embora a designação do termo componente ainda seja ambígua, algumas de suas características já estão bem estabelecidas e são de acordo geral. Neste trabalho, estamos considerando a definição de C. Szyperski [Szy98]: "Um componente de software é uma unidade de composição com interfaces especificadas através de contratos e dependências de contexto explícitas. Um componente de software pode ser distribuído independentemente e está sujeito a composição com outras partes".

A engenharia de software baseada em componentes pode apresentar diversos benefícios para o processo de desenvolvimento de sistemas, dentre os quais pode-se destacar [BBB+00; Cle95]:

- Maior flexibilidade aos sistemas. A idéia é que novas funcionalidades possam ser adicionadas aos sistemas de acordo com a necessidade e conveniência do cliente. De forma semelhante, funcionalidades já existentes e que tenham sofrido algum tipo de mudança podem também ser substituídas sem causar nenhum impacto às outras partes do sistema.

- Redução no tempo para o mercado. Supondo-se a disponibilidade de uma ampla variedade de tipos de componentes, comprar um componente pronto que atenda às necessidades do cliente pode levar muito menos tempo do que projetar, codificar, testar e implantar um novo componente. Sendo assim, é possível acompanhar de forma mais veloz as mudanças que ocorrem no mercado, favorecendo dessa forma a competitividade.

- Aumento na confiabilidade dos sistemas. Já que um componente pode ser utilizado em vários sistemas, seus erros podem ser descobertos mais rapidamente e a correção desses erros é também providenciada o quanto antes. Assim, os componentes tendem a se estabilizar mais rapidamente, tornando os sistemas mais confiáveis.

Entretanto, apesar dos benefícios, a engenharia de software baseada em componentes ainda apresenta muitas limitações e desafios que devem ser vencidos.

O maior desafio da engenharia de software baseada em componentes atualmente é a rápida montagem de sistemas a partir de componentes, de forma que determinadas propriedades do sistema final possam ser previstas a partir das propriedades individuais dos componentes que compõem o sistema. Esta não é uma tarefa fácil, especialmente pela dificuldade em descrever as propriedades individuais dos componentes. As especificações de interfaces atuais, quase sempre informais, são ainda muito limitadas, especialmente em determinar propriedades extra-funcionais tais como portabilidade, reusabilidade, confiabilidade, manutenibilidade, dentre outras.

A tecnologia também não dispõe de padrões efetivos, especialmente no que tange a interoperabilidade entre os componentes, já que a engenharia de software tradicional está

completamente concentrada na produção de software monolítico. A falta de padronização também representa um empecilho para o desenvolvimento do mercado consumidor de componentes. Atualmente, cada fornecedor segue seu próprio padrão e é difícil integrar componentes distribuídos por diferentes fornecedores.

Por fim, é notório que para se ter sucesso no reuso de componentes é necessário que se possa fazer uma verificação adequada de sua funcionalidade, tanto por parte do fornecedor do componente quanto por parte do cliente. O fornecedor precisa verificar o componente independente de contextos específicos de utilização. Por outro lado, o cliente precisa verificar o componente, possivelmente de propósitos mais gerais, dentro de um contexto específico. Teste é uma das técnicas de verificação de software mais utilizadas na prática. Se usado de forma efetiva, pode fornecer indicadores importantes sobre a qualidade e confiabilidade de um produto. A necessidade de se desenvolver técnicas que testem efetivamente os componentes individualmente e os sistemas montados a partir destes componentes é cada vez mais evidente.

O teste funcional apresenta-se como uma ferramenta de suma importância tanto para o fornecedor quanto para o cliente do componente. Baseando-se numa descrição black-box da funcionalidade do software, o teste funcional é capaz de checar se o software está de acordo com sua especificação independentemente da sua implementação. O fornecedor de um componente precisa garantir que as propriedades especificadas na interface do componente serão válidas para todos os possíveis contextos deste componente. Em termos teóricos, essa garantia apenas pode ser obtida por meio de provas formais, dado que exista uma especificação precisa do componente e das propriedades do ambiente onde ele deverá executar. Não é possível alcançar essa garantia completamente por meio de testes, já que seria necessário executar o componente em todos os possíveis contextos, o que não é viável, ou mesmo possível. Entretanto, o teste funcional pode ser usado para verificar propriedades do componente independentemente da sua implementação em contextos específicos. Se o conjunto de contextos for criteriosamente selecionado e representativo, é possível ampliar os resultados obtidos com os testes para um conjunto maior de contextos equivalentes [MS02]. Para os usuários do componente o teste funcional pode ser a única forma de testar o componente de forma satisfatória, já que normalmente o código fonte do componente não está disponível.

A partir das observações e constatações expostas até aqui, concluímos que um dos requi-

sitos para se obter sucesso no reuso de componentes de software é a verificação adequada da sua funcionalidade, tanto por parte do fornecedor, quanto por parte do cliente. Teste é uma das técnicas de verificação de software mais utilizada na prática. Se usado de forma efetiva, pode fornecer indicadores importantes sobre a qualidade e confiabilidade de um produto.

1.1 Objetivos do Trabalho

Neste trabalho, estamos propondo um método de teste funcional aplicável a componentes de software. A principal contribuição deste trabalho está em possibilitar a verificação de propriedades individuais dos componentes e empacotar artefatos e resultados de teste de forma adequada, facilitando o reuso e a composição de componentes pelos clientes. Vale ressaltar que não estamos produzindo mais uma técnica de teste isolada como tantas outras que existem. Ao contrário, o método de teste aqui proposto combina técnicas existentes e se integra a um processo de desenvolvimento de componentes, uma necessidade já notada anteriormente por outros autores [JO95], o que contribui para aumentar a testabilidade dos componentes desenvolvidos. Testabilidade é uma propriedade fundamental que engloba todos os aspectos que facilitam a elaboração e execução dos testes de software [MS01].

É desejável que este método apresente as seguintes características principais:

- O método deverá testar componentes individualizados, baseando-se numa especificação UML do mesmo.

  Escolhemos UML para orientar nosso estudo por se tratar da linguagem de modelagem mais utilizada atualmente, o que contribui para despertar o interesse de uma quantidade maior de pessoas da comunidade de engenharia de software, e também por essa linguagem ser de domínio dos colaboradores deste trabalho, o que nos permitiu dedicar maior atenção aos aspectos mais específicos do trabalho, já que não foi necessário reservar um tempo para estudar alguma outra linguagem existente, ou desenvolver uma nova linguagem, caso fosse necessário. É importante ressaltar que estaremos restritos apenas aos aspectos funcionais dos componentes. Os requisitos não-funcionais não serão considerados.

- O método deverá estar integrado a um processo de desenvolvimento de componentes.

A utilização efetiva do método é intimamente dependente do processo de desenvolvimento adotado e dos artefatos gerados. Por esse motivo, é nossa preocupação encaixar sistematicamente o método ao processo de desenvolvimento, fornecendo uma visão clara dos artefatos que devem ser produzidos com a finalidade de testar o componente e definindo precisamente quais atividades de teste devem ser realizadas em cada etapa do desenvolvimento.

- O método deverá apresentar potencial para automação.

  Embora não seja objetivo deste trabalho desenvolver ferramentas que apoiem a aplicação do método aqui proposto, estamos conscientes que a existência de ferramentas é de grande relevância para a aplicação prática do método.

  Por este motivo, nos preocupamos em desenvolver um método possível de ser automatizado. Para tanto, procuramos especificar os componentes de forma precisa, especialmente os contratos. Um contrato deve declarar, de forma não ambígua, as responsabilidades e obrigações de cada parte envolvida, além de descrever as conseqüências resultantes do descumprimento do acordo. Escolhemos OCL (Object Constraint Language) como linguagem de descrição dos contratos por se tratar da linguagem mais utilizada em conjunto com UML.

- O método deverá permitir empacotar os artefatos de teste junto com o componente.

  Pretendemos com isso possibilitar que os clientes do componente conheçam os testes que foram realizados. Essa informação contribui claramente para uma possível redução nos custos com re-testes, além de ajudar na definição dos testes de integração e de permitir a realização de testes com novos dados por parte dos usuários do componente.

- O principal beneficiado com o método deverá ser o fornecedor do componente.

  Este método deverá testar os componentes individualmente e não a integração dos componentes em uma aplicação. Por esse motivo, o método é mais direcionado para o fornecedor do componente. Entretanto o cliente do componente é também favorecido com o método, já que é pretendido divulgar juntamente com o componente os artefatos e resultados dos testes executados no componente.

É importante notar que uma metodologia de teste completa deve incorporar também técnicas de teste estruturais, além de permitir que sejam executados testes de integração e sistema, entretanto, isto está fora do escopo deste trabalho.

1.2 Contribuições e Relevância do Trabalho

Teste é uma atividade importante no processo de desenvolvimento de software que é aplicada para fornecer garantias de qualidade do produto. A atividade de teste consiste de projetar os casos de teste, executar o software com os casos de teste projetados e analisar os resultados produzidos. Estudos mostram que cerca de 50% do custo do desenvolvimento de um software é destinado a esta atividade [Har00].

A principal contribuição deste trabalho será possibilitar a verificação efetiva das funcionalidades de um componente de software, aumentando sua confiabilidade e contribuindo assim para aumentar as chances de seu uso por parte dos desenvolvedores de aplicações. Ao permitir que propriedades individuais dos componentes possam ser checadas, este trabalho contribuirá no sentido de motivar a composição de componentes, o que representa atualmente um dos grandes desafios da engenharia de software baseada em componentes.

Uma outra contribuição do trabalho é o fato do método estar voltado para o teste de componentes. Existem atualmente poucas propostas de trabalho para teste funcional de componentes [MTY01] e as existentes se concentram geralmente em aspectos isolados da atividade de teste, além de não estarem integradas a uma metodologia de desenvolvimento de componentes. Além disso, não encontramos técnicas que se preocupassem em disponibilizar os resultados e artefatos de teste juntamente com o componente, o que aumenta a confiabilidade dos usuários, contribui para uma possível redução dos custos envolvidos no re-teste e favorece a montagem dos sistemas a partir de componentes.

Por fim, mas não menos importante, o trabalho contribuirá no sentido de produzir um estudo de caso, a fim de que possamos ter uma noção inicial da viabilidade do método do ponto de vista prático.

1.3 Metodologia

A fim de definirmos inicialmente o escopo deste trabalho, realizamos uma pesquisa bibliográfica sobre engenharia de software baseada em componentes, o que resultou na identificação dos principais problemas ainda não solucionados na área e nos ajudou a estabelecer os objetivos deste trabalho.

Ao fim desta pesquisa decidimos que:

- O método deveria estar integrado a um processo de desenvolvimento.
- O método deveria se basear em especificações UML.
- O método deveria apresentar potencial para automação.
- Especificações mais precisas deveriam ser construídas usando OCL.

Com o objetivo principal do trabalho em mente, passamos a pesquisar metodologias de desenvolvimento de componentes e técnicas de teste funcional para verificação de objetos e componentes.

Da pesquisa realizada, elegemos como candidatas à incorporação do método as metodologias de desenvolvimento Componentes UML [CD01], Catalysis [DW98] e KobrA [ABB+01]. As técnicas de teste selecionadas inicialmente foram TOTEM - Testing Object-orienTed systEms, proposta em [BL01], Cleanroom [PTLP99] e as técnicas propostas em [HIM00; CTF01].

Em seguida, investigamos de forma mais detalhada as metodologias de desenvolvimento candidatas. Elencamos as características mais importantes que gostaríamos que a metodologia apresentasse e escolhemos a metodologia que melhor se enquadrava às nossas exigências, no caso, a metodologia Componentes UML. Os critérios utilizados nessa seleção foram os seguintes:

- A metodologia deveria ser fácil de ser aprendida e aplicada por qualquer usuário que tivesse um conhecimento básico de UML.
- A metodologia deveria produzir apenas artefatos necessários para especificação e validação do software. Esses artefatos deveriam ser os mais comumente produzidos

na modelagem de software orientado a objetos, como Diagrama de Casos de Uso, Diagrama de Classes e Diagrama de Seqüência ou Colaboração.

- A metodologia deveria incorporar OCL na modelagem das invariantes de classes e das pré e pós-condições das operações das classes.

As três metodologias candidatas atendiam às duas últimas condições, mas Catalysis e KobrA eram metodologias mais abrangentes e por isso mais complexas. A Componentes UML é uma metodologia voltada exclusivamente para produção de componentes de software, e por esse motivo apresenta-se de uma forma mais simplificada, o que pode facilitar a sua aplicação prática.

As técnicas de teste selecionadas inicialmente foram também estudadas de forma mais atenta. Fizemos também uma seleção das principais características desejadas na técnica de teste e escolhemos a técnica que seria utilizada no método baseando-nos nesses critérios. Os critérios foram os seguintes:

- A técnica deveria apresentar potencial para automação, considerando-se especialmente a possibilidade de uso de OCL.

- A técnica deveria usar os principais artefatos UML, Diagrama de Casos de Uso, Diagrama de Classes, Diagrama de seqüência ou Colaboração, como fonte para geração dos testes.

- A estratégia de cobertura apresentada pela técnica deveria permitir que as funcionalidades gerais dos componentes pudessem ser verificadas através de sua aplicação.

Das técnicas que tínhamos em mãos, a que mais se adequava aos critérios definidos era a TOTEM, que foi a escolhida.

Durante a escolha da metodologia fizemos também a escolha do estudo de caso. Precisávamos de uma aplicação que gerasse componentes que tivessem funcionalidades não triviais a fim de que pudéssemos produzir um exemplo interessante. Além disso, não gostaríamos de produzir toda a especificação dos componentes. Estávamos procurando portanto, uma aplicação de complexidade razoável, componentizada, e que já tivesse uma especificação disponível. Não encontramos essa aplicação entre os projetos desenvolvidos pelos nossos

colegas do departamento, então selecionamos o exemplo encontrado no livro da metodologia Componentes UML e o adotamos como estudo de caso. Essa decisão foi especialmente encorajada pelo fato da aplicação já estar componentizada e o livro já apresentar boa parte da especificação.

Com a escolha do estudo de caso, passamos a aplicar na prática a metodologia de desenvolvimento Componentes UML. Como tínhamos a preocupação com a testabilidade dos componentes, alguns aspectos da metodologia acabaram sendo adaptados. Essas adaptações são relatadas no Capítulo 4.

Nesse ínterim, foi necessário realizar também um estudo mais aprofundado da linguagem OCL, já que esta não era do domínio dos participantes do projeto.

À medida que caminhamos no processo de desenvolvimento fomos inserindo as atividades de teste, concluindo que é possível, e bastante proveitoso, em termos de qualidade final do produto, ter um processo de teste sendo executado em paralelo com o processo de desenvolvimento.

Algumas adaptações à técnica de teste escolhida foram também sendo identificadas nesse momento. Inclusive percebemos a possibilidade de combinar aspectos da técnica Cleanroom com a técnica TOTEM e o fizemos (ver Capítulo 5).

Por fim, os componentes especificados foram implementados, seus casos de teste foram também implementados e executados, e os resultados foram analisados, produzindo o Capítulo 6.

1.4 Estrutura da Dissertação

Capítulo 2: Neste capítulo, o estado da arte da engenharia de software baseada em componentes é delineado. São apresentadas as principais características desse paradigma, seu impacto na disciplina de Engenharia de Software, os problemas em aberto e ainda alguns trabalhos de pesquisa relacionados à área.

Capítulo 3: O capítulo apresenta os principais conceitos e terminologias relativos a teste funcional. Algumas propostas de técnicas de teste funcional para software orientado a objetos, componentes e arquiteturas são também discutidos.

Capítulo 4: A metodologia de desenvolvimento de componentes **Componentes UML**

, escolhida para exemplificar a aplicação do método, é apresentada. Um estudo de caso - Sistema Gerenciador de Hotéis - é definido, e a metodologia é exemplificada usando este estudo de caso.

Capítulo 5: O capítulo descreve o método de teste proposto, especificando de forma detalhada como gerar casos de teste, selecionar os dados e gerar os oráculos de teste. O processo é exemplificado com a aplicação do estudo de caso definido no capítulo anterior.

Capítulo 6: Neste capítulo, são relatados os resultados obtidos com a realização do experimento, os problemas encontrados e as sugestões de melhoria do método.

Capítulo 2

Engenharia de Software Baseada em Componentes

Atualmente, é notório o crescente interesse da comunidade de Engenharia de Software no desenvolvimento de software baseado em componentes, o que às vezes sugere que esta forma de estruturação de sistemas tenha surgido recentemente. Entretanto, ao se fazer uma análise mais crítica do assunto, percebe-se que qualquer sistema de software se constitui de partes que são estruturadas de alguma forma para compor o sistema mais amplo.

A engenharia de software baseada em componentes, portanto, não representa de fato uma mudança significativa na forma como os sistemas são construídos. Ao invés disso, este paradigma é uma tentativa de conceber sistemas a partir da combinação de unidades pré-desenvolvidas com o objetivo de reduzir o tempo de desenvolvimento, facilitar o acoplamento de novas funcionalidades, ou a mudança de funcionalidades já existentes e ainda promover o reuso de partes do software.

Infelizmente, a falta de acordo entre pesquisadores, produtores de tecnologia e consumidores sobre o que são componentes e como eles podem ser usados para projetar, desenvolver e implantar novos sistemas ainda dificulta os avanços na área.

Este capítulo tem por objetivo apresentar o conceito de componentes, na visão de diferentes autores, e suas características principais. O capítulo trata ainda de algumas das principais metodologias existentes para desenvolvimento de componentes.

2.1 O que é um componente?

Na visão de F. Bachman et. al. [BBB+00], um componente é:

- uma implementação opaca de uma funcionalidade;
- que está sujeito a composição com outras partes;
- e que obedece a um modelo de componente.

Essa visão sugere que um componente deverá permanecer como uma "caixa preta" para os consumidores, e deverá obedecer certas convenções estabelecidas em um modelo de componente a fim de que possa interagir com outros componentes de forma transparente.

Com algumas diferenças sutis, C. Szyperski [Szy98] define um componente como sendo:

- uma unidade de desenvolvimento independente;
- uma unidade de composição com outras partes;
- e que não possui estado persistente.

Dessa definição podemos concluir que o desenvolvimento de um componente deve se dar de forma completamente isolada, independente do contexto em que ele será usado, já que um componente pode ser reutilizado em diferentes contextos.

A definição também indica que um componente encapsula suas características, estando portanto bem separado de seu ambiente e de outros componentes. Entretanto, um componente deve interagir com o mundo exterior e essa interação deve se dar através de especificações claras do que o componente requer e fornece.

Por fim, é sugerido na definição acima que um componente não deve apresentar um estado persistente a fim de que duas instalações de um mesmo componente possam garantir as mesmas propriedades.

As duas definições apresentadas são bastante parecidas e ressaltam as principais características, de acordo geral, que devem estar presentes em um componente.

Como toda implementação de software, um componente é desenvolvido para exercer um determinado papel dentro do sistema onde ele será inserido. Dentro do sistema, o componente não permanece isolado e deve, portanto, fornecer meios para que as outras partes

do sistema possam se comunicar com ele. Isso é feito através de interfaces. As interfaces implementadas por um componente descrevem as propriedades que o componente oferece para seus clientes. Entretanto, para que a interação entre componentes e clientes se dê de forma harmoniosa, é necessário especificar também o que os clientes precisam fazer para usar o componente. A especificação das obrigações recíprocas entre as partes envolvidas na interação é feita através de contratos. Os contratos portanto, na engenharia de software baseada em componentes, declaram o que o cliente precisa fazer para usar uma interface e o que os fornecedores devem implementar para atender os serviços prometidos na interface.

Um último ponto que deve ser ressaltado sobre a definição fornecida em [BBB+00] é que todo componente deve estar em conformidade com um modelo. Um modelo de componente estabelece determinadas convenções que devem ser respeitadas pelos componentes. Essas convenções definem os padrões de interação permitidos entre componentes e auxiliam portanto no processo de composição de componentes.

Uma outra peça chave na tecnologia de software baseada em componentes é o framework de componente. Segundo C. Szyperski [Szy98], no mundo orientado a objetos, um framework é um conjunto de classes, algumas delas abstratas, que cooperam entre si para constituir um projeto reutilizável para uma classe de software específica. Freqüentemente, um framework fornece algumas implementações default e os clientes do framework precisam apenas substituir ou complementar aquelas implementações que não estão adequadas ao seu problema específico. Ainda na visão de C. Szyperski, um framework de componente é um software que suporta o acoplamento de componentes que se adequam a certos padrões. Este framework estabelece as condições ambientais de funcionamento para as instâncias dos componentes e regula as interações entre essas instâncias. O framework de componente pode, portanto, ser visto como uma espécie de sistema operacional de propósito específico, que gerencia os recursos compartilhados pelos componentes e fornece os mecanismos básicos de comunicação entre eles. Um framework de componente é específico para determinados tipos de componentes. Isso faz com que o framework seja uma peça um tanto quanto inflexível, já que os componentes que podem ser acoplados a ele devem seguir determinados padrões, porém aumenta as chances de sucesso na composição dos componentes.

2.2 Como desenvolver componentes

Embora o interesse pela engenharia de software baseada em componentes tenha tomado grandes proporções, tanto na comunidade de software quanto em numerosos setores industriais, processos e metodologias sistemáticas para construção de componentes ainda não são completamente satisfatórios, especialmente por não tratarem de aspectos relacionados a teste de componentes.

A seguir discutimos algumas dessas metodologias.

2.2.1 A Metodologia COMO

Uma proposta de processo de desenvolvimento de componentes interessante pode ser encontrada em [LYC+99]. Esta proposta, batizada de COMO - Object-Oriented Component Development Methodology, extende UML e o processo unificado, definindo notações relacionadas ao desenvolvimento de componentes não contempladas pela UML padrão, além de fornecer um processo sistemático de desenvolvimento de componentes, detalhando cada etapa do processo.

O processo de desenvolvimento proposto consiste de 4 etapas, embora apenas as duas primeiras etapas tenham sido detalhadas pelos autores da proposta:

- Análise de Domínio
- Projeto do Componente
- Construção do Componente
- Teste do Componente

A etapa de Análise do Domínio

Os objetivos desta fase são compreender o domínio do problema e identificar os requisitos do componente.

Esta etapa compreende as seguintes tarefas:

- Identificar requisitos do domínio: esta tarefa consiste em analisar especificações de diversas aplicações relacionadas ao domínio do problema e construir a partir desta

análise um dicionário de termos e um dicionário de funções. Esses dicionários contêm descrições de termos e funções padrões do domínio do problema.

- Extrair atributos comuns: o objetivo desta tarefa é identificar funções comuns às aplicações analisadas. Para cada função identificada, um diagrama de caso de uso deve ser construído.

- Identificar variações: a fim de modelar componentes flexíveis, deve-se identificar variações nos atributos, lógica e fluxo de trabalho das aplicações analisadas. As informações identificadas devem ser sintetizadas em uma lista de variações.

- Construir modelo conceitual de objetos: nesta tarefa são identificados conceitos do negócio, extraídos do conjunto de requisitos do domínio, do dicionário de termos e do conhecimento do domínio. Neste momento, deve ser construído um diagrama de classes, onde são representadas as classes, atributos, relacionamentos e operações das classes.

- Identificar componentes: o objetivo desta tarefa é agrupar classes e casos de uso que se relacionam.

- Associar objetos do negócio a componentes: um componente é composto de uma ou mais classes. A tarefa a ser executada neste momento é identificar quais classes devem ser associadas a um componente específico.

- Definir especificação de requisitos do componente: a especificação do componente consiste do nome do componente, uma breve descrição do componente, as classes contidas no componente, o fluxo de trabalho do componente, etc. A idéia desta tarefa é produzir um documento, onde se possa obter informações gerais da funcionalidade do componente.

- Refinar o modelo de análise do domínio: o objetivo desta tarefa é revisar e refinar os artefatos produzidos na etapa de análise do domínio. Deve-se checar a consistência entre os artefatos e a completude de cada artefato.

A etapa de Projeto do Componente

Esta etapa consiste de seis tarefas que são processadas iterativamente:

- Identificar fluxos de mensagens: um diagrama de seqüência deve ser construído para cada caso de uso do componente para representar os fluxos de mensagens entre objetos e o fluxo de mensagens entre objetos e o componente. Os fluxos de mensagens são obtidos a partir da descrição dos casos de uso do componente.

- Definir interfaces das classes: esta tarefa consiste em identificar as operações das classes que constituem o componente. A identificação das operações deve estar baseada na análise dos diagramas de seqüência produzidos na tarefa anterior. As operações identificadas devem ser adicionadas ao diagrama de classes.

- Definir política de customização: o objetivo desta tarefa é preparar o componente para que ele possa ser customizado futuramente sem maiores impactos na sua estrutura.

- Definir interface do componente: nesta tarefa, deve-se identificar e definir a interface do componente e os métodos presentes na interface. A execução desta tarefa está baseada na análise dos diagramas de seqüência e nos pontos de customização definidos nas tarefas anteriores.

- Definir especificação do componente: inicialmente, deve-se definir um contrato para o componente, contendo pré e pós-condições e exceções para cada método da interface do componente. Tendo definido o contrato, a especificação do componente pode ser construída. A especificação do componente consiste do nome do componente, uma breve descrição do componente, as classes contidas no componente, um diagrama de componente, representando o modelo estático do componente, os diagramas de seqüência, representando o modelo dinâmico, e o contrato do componente.

- Refinar o modelo do projeto: o objetivo desta tarefa é revisar os artefatos da etapa de projeto do componente.

O trabalho de S. Lee et. al. [LYC+99] apresenta ainda um estudo de caso para o domínio de comércio eletrônico e compara de forma superficial a metodologia COMO com outras duas metodologias propostas anteriormente: Catalysis e SCIPIO.

A metodologia COMO é uma metodologia prática, que apresenta grande facilidade de aprendizado e uso. Entretanto, a descrição de suas etapas e atividades precisa ser melhor detalhada. A etapa de teste, que é apenas mencionada no trabalho, precisa ainda ser definida.

2.2.2 Extensão do meta-modelo UML para especificação de contratos de componentes

Para que um componente possa ser utilizado, uma certa quantidade de informações sobre como o componente pode ser utilizado deve ser fornecida. Isso é feito normalmente através de interfaces que expõem aos usuários as funcionalidades do componente. Entretanto, o uso de interfaces apenas não garante o uso adequado do componente. Uma especificação mais precisa do comportamento do componente é necessária e deve abranger seu comportamento funcional e não-funcional. Atualmente, os contratos têm sido usados para suprir essa especificação. Entretanto, o padrão UML ainda não fornece ferramentas para modelar contratos de componentes.

Em [WBGP01] uma proposta de extensão do meta-modelo UML que suporta a modelagem melhorada dos contratos de componentes é apresentada. O trabalho considera o Processo Unificado como processo de desenvolvimento dos componentes e preocupa-se com a integração dos contratos nas diferentes etapas do processo de desenvolvimento.

A Figura 2.1 mostra parte do meta-modelo UML 1.4 beta1 e as novas meta-classes propostas em [WBGP01] que suportam a especificação de contratos.

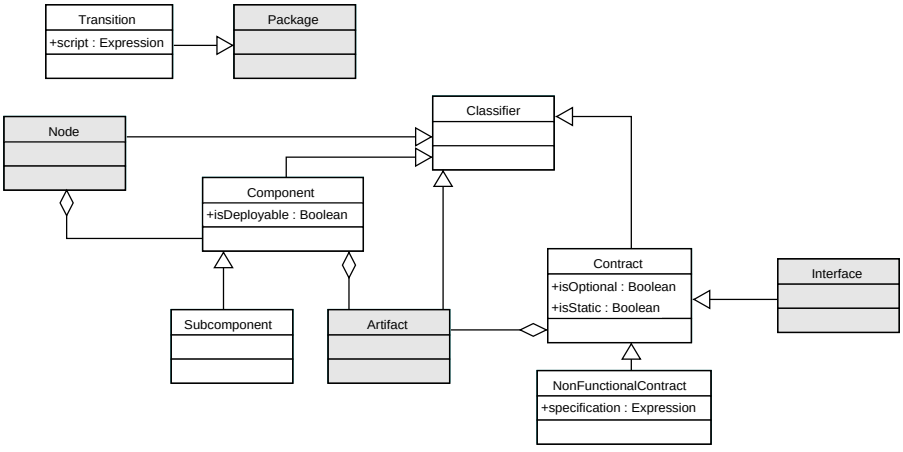

Figura 2.1: Extensões do meta-modelo UML

As seguintes mudanças são propostas (ver Figura 2.1):

- Adição da meta-classe Contract e suas subclasses. A meta-classe Contract apresenta o atributo booleano isOptional para indicar se o contrato é aceito ou não pelo usuário, e o atributo isStatic, também booleano, que indica se a seleção do contrato pode ou não ser feita em tempo de execução.

- O elemento Interface já existia no meta-modelo UML 1.4 beta1 e foi redefinido em [WBGP01]. Nesta extensão, o elemento Interface passa a ser subclasse de Contract e representa contratos funcionais do componente.

- A subclasse NonFunctionalContract representa os contratos não funcionais do componente e apresenta o atributo specification que deve armazenar a natureza do contrato expressa em alguma linguagem própria para especificação.

- Instâncias de NonFunctionalContract podem ser associadas a instâncias de Interface para formarem contratos compostos que definem o comportamento funcional e não-funcional do componente.

- A meta-classe Artefact foi introduzida no meta-modelo UML 1.4 beta1 e pode ser associada a um contrato. Um artefato representa dados físicos que serão instalados se o contrato for selecionado.

- A meta-classe Transition é subclasse de Package e permite descrever as transformações do modelo UML sem poluir o espaço de nomes global. O atributo script pode conter uma descrição textual das transformações. Uma transição pode ser associada a um contrato através de uma relação de dependência.

- Subcomponent é subclasse de Component. Um subcomponente se diferencia de um componente porque um subcomponente não pode ser entregue independentemente, ele depende de outros subcomponentes que formam o componente. O relacionamento entre um subcomponente e seu componente pai se dá através de uma relação de dependência.

Este trabalho propõe ainda uma distinção entre os contratos oferecidos e os contratos requeridos pelo componente. Um componente se relaciona com seus contratos oferecidos

através da relação de realização, que é representada por uma linha pontilhada com seta fechada. Já o relacionamento de dependência entre um componente e um contrato, representado através de uma linha pontilhada com seta aberta, indica que o contrato é requerido pelo componente, ou seja, significa que o componente apresenta certas exigências para que possa executar sua funcionalidade satisfatoriamente. O mesmo é aplicado aos contratos compostos.

Para as mudanças propostas neste trabalho, foram especificadas notações gráficas que podem ser consultadas em [WBGP01]. Por fim, os autores da proposta preocuparam-se em conceber uma extensão que mantivesse a compatibilidade com os modelos construídos anteriormente usando UML 1.4 beta1.

Como proposta de extensão da linguagem UML para especificação dos contratos de componentes, o trabalho está bastante detalhado e fundamentado. Entretanto, embora a proposta seja utilizar o Processo Unificado como metodologia de desenvolvimento de componentes, os autores não tiveram a preocupação de explicitar em que ponto do processo os contatos seriam definidos e como isso seria feito. Portanto, em termos de uma metodologia de desenvolvimento de componentes, esta proposta precisa ser ainda complementada para ser utilizada na prática.

2.2.3 Catalysis

A metodologia Catalysis proposta por D. F. D'Souza e A. C. Wills [DW98] suporta o desenvolvimento baseado em componentes através do uso de objetos e frameworks baseando-se no padrão UML de modelagem.

Os Objetivos

Os principais objetivos desta metodologia são:

- Auxiliar a captar os requisitos do software e implementá-los apropriadamente.
- Promover o desenvolvimento do software por equipes distribuídas, permitindo que as unidades produzidas separadamente sejam reunidas de forma sistemática.
- Produzir software flexível que acompanhe as mudanças de requisitos do negócio.

- Facilitar a construção de famílias de produtos a partir de kits de componentes.

Conceitos Básicos

Catalysis está fundamentada no desenvolvimento de software orientado a objetos. Como já é sabido, o desenvolvimento de software orientado a objetos procura aproximar o mundo do software do mundo do usuário, usando para isso conceitos que fazem parte do vocabulário do usuário.

Os conceitos básicos por trás da Catalysis são:

- Objetos: Blocos que reúnem informação e funcionalidade.
- Ações: Tudo que acontece no sistema, como eventos, mudanças de estado, trocas de mensagens, tarefas, etc.

Catalysis ressalta a importância das ações por acreditar que a modelagem das ações e seus efeitos para o sistema é capaz de produzir um projeto mais desacoplado e portanto mais flexível do que um projeto construído considerando-se apenas os objetos.

Os Princípios

A base da Catalysis está em três princípios:

- Abstração: separar os aspectos mais importantes de um problema dos seus detalhes a fim de lidar melhor com a complexidade do problema.
- Precisão: garantir que apenas uma interpretação do problema a ser resolvido é compartilhada por todas as pessoas envolvidas na solução do problema.
- Partes "plugáveis": facilitar a construção de software flexível, confiável e altamente reutilizável.

O Processo

O processo de desenvolvimento sugerido na Catalysis é um processo não-linear, iterativo e paralelo. O modelo em espiral é aplicado no processo. Neste modelo cada ciclo inclui uma revisão dos resultados e riscos que refina os objetivos do próximo ciclo (ver Figura 2.2). Os

ciclos podem acontecer concorrentemente, já que muitas atividades podem ser desenvolvidas em paralelo.

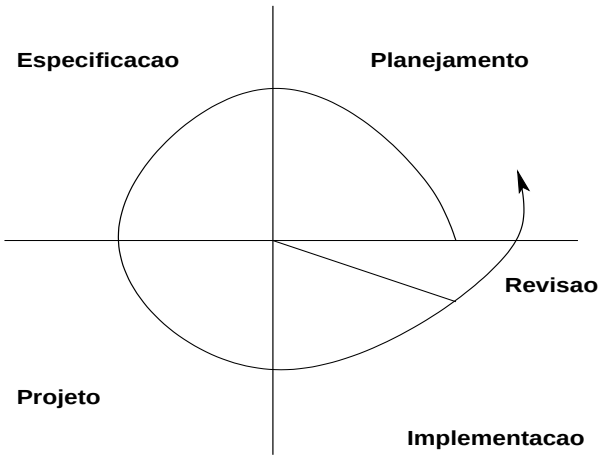

Figura 2.2: Modelo em Espiral

Uma outra característica importante no processo é que o controle de qualidade não é uma atividade realizada apenas quando o produto já está pronto, como acontece na maioria das metodologias de desenvolvimento existentes. Catalysis se preocupa com qualidade desde a produção dos artefatos intermediários, documentação, até o produto final.

O uso de notações precisas para especificar pré e pós-condições e invariantes demonstra essa preocupação com a qualidade. Notações precisas evitam ambigüidades e interpretações equivocadas, além de facilitar o projeto de testes, tanto a nível de unidade quanto a nível de integração.

Existem muitos caminhos que podem ser seguidos para se desenvolver um sistema de software. Cada caminho propõe uma seqüência de tarefas e artefatos que devem ser produzidos que melhor se adequam às características do projeto. As etapas que constituem um processo típico para desenvolvimento de sistemas de negócios são as seguintes:

- Levantamento de Requisitos.
- Especificação do Sistema.
- Projeto Arquitetural.
- Projeto de Componentes.

- Implementação.
- Testes.

Levantamento de Requisitos Nesta etapa procura-se compreender o problema e imaginar uma solução adequada a ele. Para atingir este objetivo, tipicamente existe um ciclo de leitura de material existente sobre o problema e entrevistas de pessoas experientes no domínio do problema.

Os principais artefatos produzidos nesta etapa são:

- Modelo de Negócio: Inclui um modelo de tipos, que representa as informações que devem ser gerenciadas pelo sistema, modelos de colaboração, que identificam os atores do sistema e suas ações sobre o sistema e um glossário, que concentra os principais termos relacionados ao domínio do problema.
- Requisitos Funcionais: Um diagrama de contexto do sistema, definindo claramente os limites do sistema, casos de uso e cenários, representando seqüências de ações envolvidas no fornecimento de uma funcionalidade.
- Requisitos Não-Funcionais: Informações sobre performance, confiabilidade, escalabilidade e objetivos de reuso.
- Restrições Arquiteturais e Plataforma: Informações sobre configurações de hardware, sistema operacional, distribuição, middleware, dentre outros.
- Restrições do Projeto: Informações sobre orçamento, equipe de trabalho, cronograma e envolvimento do usuário.

Ao final desta etapa deve-se ter uma visão clara do produto a ser desenvolvido, que funcionalidades ele deverá fornecer, em que ambiente ele será executado e quais os principais riscos envolvidos em seu desenvolvimento.

Especificação do Sistema Esta etapa dá continuidade à etapa anterior adicionando elementos de interface com o usuário. Além de refinar o modelo de tipos do sistema e especificar suas operações, nesta etapa são produzidos protótipos e especificações de interfaces com

usuário que descrevem as telas, fluxos de diálogos entre as janelas informações apresentadas e solicitadas e relatórios. Esses elementos de interface são extraídos do modelo inicial de tipos e dos cenários produzidos na etapa anterior.

Projeto Arquitetural O projeto arquitetural envolve duas partes principais:

- Arquitetura da Aplicação: Empacota a lógica da aplicação na forma de uma coleção de componentes que colaboram entre si a fim de fornecer um determinado serviço ou funcionalidade. A arquitetura da aplicação é uma camada acima da arquitetura técnica e faz uso dos serviços fornecidos por esta outra camada.

- Arquitetura Técnica: Cobre todas as partes independentes do domínio do sistema, tais como, plataformas de hardware e software, arquitetura de componentes, middleware, sistemas gerenciadores de banco de dados etc. A arquitetura técnica inclui ainda as regras e padrões que serão usados na implementação. O projeto e implementação da arquitetura técnica usa as informações colhidas na etapa de levantamento de requisitos sobre os requisitos não-funcionais.

Projeto de Componentes O objetivo desta etapa é definir uma estrutura interna de componentes que satisfaça os requisitos comportamentais, tecnológicos e não-funcionais previstos para o sistema.

A regra geral é que cada tipo identificado no modelo de tipos representa um componente e cada um desses componentes deverá ser implementado através de uma única classe. Componentes mais complexos podem ser particionados e implementados através de mais de uma classe. Nesses casos, deve-se identificar as responsabilidades de cada classe e construir diagramas de interação para representar as interações existentes entre as classes a fim de executar uma determinada funcionalidade.

Neste ponto do processo, tem-se informações suficientes para se construir o modelo de classes do sistema. O modelo deve apresentar as classes que compõem o sistema, as interfaces que elas implementam e usam, seus atributos e operações e os relacionamentos entre as classes.

Comentários Finais sobre a Catalysis

É válido ressaltar dois aspectos relevantes da Catalysis: o suporte ao desenvolvimento baseado em componentes e o uso de UML na produção dos artefatos, o que permite a utilização de ferramentas de modelagem já consolidadas no mercado. Por outro lado, a Catalysis apresenta a desvantagem de ser muito complexa, o que dificulta seu aprendizado e aplicação prática. Além disso, mais uma vez, a etapa de teste é desprezada, existindo pouca ou nenhuma definição das atividades que devem ser realizadas nesta etapa, e mesmo durante as outras etapas, percebemos que não há preocupação com a testabilidade do produto que está sendo desenvolvido.

2.2.4 KobrA: Engenharia de Linha de Produto baseada em Componentes com UML

A metodologia KobrA proposta em [ABB+01] reúne as principais características de outras metodologias de desenvolvimento como Catalysis [DW98], Componentes UML [CD01], Rational Unified Process - RUP, Cleanroom, Object Modeling Technique - OMT, dentre outras, tendo como objetivo principal facilitar o desenvolvimento de software através do reuso.

Características

As principais características da metodologia KobrA resultam da tentativa de torná-la interessante, do ponto de vista dos usuários potenciais. Dentre as características apresentadas pelo KobrA, destacam-se:

- **Princípio de Uniformidade** - Este princípio garante que toda entidade rica em comportamento é tratada como um componente KobrA, e todo componente é tratado de maneira uniforme, independentemente da sua granularidade. Assim, o sistema como um todo é visto e modelado como um componente.

- **Desenvolvimento Recursivo** - Diferentemente do desenvolvimento tradicional, onde o sistema é desenvolvido através da aplicação de etapas monolíticas desarticuladas, o

desenvolvimento recursivo permite que o software seja desenvolvido a partir da aplicação recursiva de etapas previamente estabelecidas, permitindo que o software seja desenvolvido de forma incremental.

- **Regras de Consistência -** Todo e qualquer artefato produzido durante a aplicação do KobrA deve satisfazer a um conjunto de regras concretas que controlam sua forma e seus relacionamentos. Ao final de cada atividade realizada no KobrA, é verificado se os artefatos produzidos estão consistentes com as regras estabelecidas para sua construção.

- **Princípio de Economia -** Qualquer modelo ou diagrama produzido durante a aplicação do KobrA deve conter apenas a quantidade suficiente de informações necessárias para descrever as propriedades desejadas, nem mais, nem menos. O princípio de economia se aplica também na definição dos artefatos que devem ser produzidos. Devem ser gerados apenas artefatos que contribuam para a compreensão do domínio do problema e para a descrição e validação do software. Artefatos que não têm utilidade prática devem ser evitados.

- **Separação de Assuntos -** Separação de assuntos é uma estratégia de tratamento de complexidade, onde um problema complexo é subdividido em partes que podem ser tratadas separadamente. Esta característica permite que apenas alguns dos aspectos do método sejam usados durante o desenvolvimento de um software.

Conceitos

A essência da metodologia KobrA está na separação estrita e sistemática de assuntos. A manifestação mais visível deste princípio está na separação explícita da definição do produto da definição do processo, ou seja, a descrição do que deve ser feito é separada da descrição de como fazê-lo.

Reforçando o princípio da separação de assuntos, o método está organizado em termos de três dimensões, que se relacionam entre si: generalidade, abstração e composição. A Figura 2.3 ilustra o relacionamento entre estas dimensões.

O desenvolvimento de um projeto KobrA tem início com uma descrição abstrata, genérica e black-box do sistema que será desenvolvido. Esta descrição é representada pelo

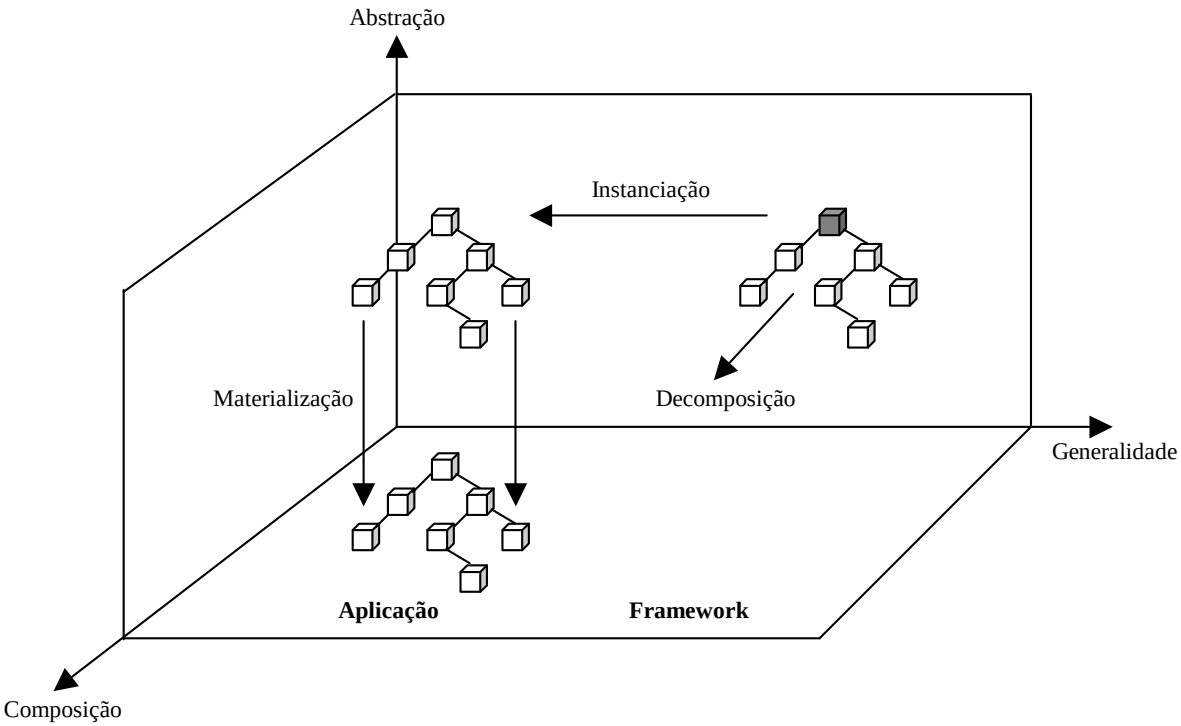

Figura 2.3: Principais Dimensões de Desenvolvimento

cubo preto no canto superior direito da Figura 2.3.

Para criar uma versão executável final deste sistema são necessárias 3 etapas:

- **Instanciação -** Remover a generalidade a fim de se criar uma instância do sistema que satisfaz as necessidades específicas de uma aplicação.
- **Decomposição -** Decompor o sistema em partes melhor estruturadas a fim de se produzir uma árvore de componentes aninhados.
- **Materialização -** Reduzir o nível de abstração a fim de se obter uma representação executável do sistema e de suas partes.

O produto final, obtido após a realização das atividades citadas acima, é representado pela aplicação no canto inferior esquerdo da Figura 2.3. A aplicação é representada em termos de componentes bem estruturados, com grau mínimo de generalidade e uma representação concreta.

Como já foi dito anteriormente, no KobrA as três dimensões ilustradas na Figura 2.3 são estritamente separadas. Os assuntos relacionados à dimensão generalidade são tratados

pela engenharia de linha de produto, a dimensão composição é abordada pela modelagem de componentes e a materialização de componentes ocupa-se da dimensão abstração.

Engenharia de Linha de Produto A engenharia de linha de produto visa desenvolver famílias de produtos similares, sutilmente diferentes. A idéia é reaproveitar partes dos produtos previamente desenvolvidos para desenvolver novos produtos.

Para atingir seus objetivos, a engenharia de linha de produto divide o ciclo de desenvolvimento de software em duas partes: a engenharia de framework e a engenharia de aplicação.

A engenharia de framework abrange atividades e artefatos relacionados ao desenvolvimento e manutenção de um framework. Um framework é um conjunto de artefatos de software que concentra os conceitos gerais compartilhados por várias aplicações.

A engenharia de aplicação está concentrada no desenvolvimento e manutenção de uma aplicação, ou seja, uma instância concreta do framework, adaptada e extendida para satisfazer as necessidades específicas de um cliente.

Modelagem de Componentes Os frameworks e aplicações desenvolvidos utilizando-se a metodologia KobrA são todos estruturados em termos de hierarquias de componentes.

O objetivo principal da modelagem de componentes é captar como transformar componentes rudimentares em um conjunto de componentes mais estruturados. Isso é feito de forma recursiva, até se obter os componentes mais primitivos.

Os componentes são organizados em uma estrutura de árvore, onde a raiz da árvore representa um componente que é constituído dos componentes representados nas ramificações da árvore.

Materialização de Componentes A materialização de componentes preocupa-se em reduzir o nível de abstração dos artefatos desenvolvidos. Existem duas formas de se obter uma versão executável e concreta de um componente: desenvolver completamente um novo componente, aplicando todas as etapas do processo de desenvolvimento, ou reusar um componente existente que apresente as propriedades desejadas e que se adeque ao framework ao qual ele será acoplado.

O reuso representa a forma mais econômica de se obter uma versão concreta do componente lógico. Para verificar se um componente pode ser reusado em determinada situação, o

método KobrA sugere o desenvolvimento de uma especificação baseada em UML do componente candidato. Esta especificação deve então ser comparada com a especificação do componente desejado.

Se não é possível encontrar um componente pré-existente adequado ao problema em questão, a solução é implementar um novo componente. A implementação de um novo componente envolve um processo de transformação de modelos em código-fonte adequado a alguma linguagem de programação.

No método KobrA, este processo de transformação é dividido em duas etapas: refinamento e tradução. Durante o refinamento, os elementos do modelo abstrato são descritos de forma mais detalhada, usando-se a mesma notação usada na descrição abstrata. A tradução mapeia os elementos do modelo em elementos do programa.

Controle e Monitoração de Projeto A criação ou reuso de artefatos de software é suportada por uma quarta dimensão, não mencionada na Figura 2.3, conhecida por controle e monitoração de projeto. Esta dimensão corta todo o processo de desenvolvimento, afetando todo e qualquer artefato produzido. Os principais pontos abordados nesta dimensão são a manutenção e a qualidade do software.

Na engenharia de software baseada em componentes, a qualidade dos componentes é essencial, já que um componente pode ser reutilizado em uma ampla variedade de sistemas diferentes. Qualidade é, portanto, umas das principais preocupações do método KobrA. Por esse motivo, atividades de medida e aperfeiçoamento de qualidade estão embutidas em todas as etapas do método, em todas as dimensões do desenvolvimento.

A principal atividade de garantia de qualidade no KobrA são as inspeções. Elas são aplicadas em todas as etapas e dimensões do desenvolvimento, a fim de checar se os artefatos produzidos satisfazem o conjunto de regras que controlam a construção destes artefatos.

A fim de melhorar a qualidade dos componentes, as inspeções são complementadas por testes. KobrA sugere que sejam desenvolvidos casos de teste funcionais tão logo a especificação do componente esteja completa. Casos de teste estruturais devem ser desenvolvidos assim que a realização do componente esteja pronta.

O Processo

A metodologia KobrA propõe a execução das seguintes etapas:

- **Realização do Contexto**

  O processo de desenvolvimento se inicia com a realização do contexto. Esta atividade tem o propósito de identificar as principais propriedades do ambiente do componente, representando o sistema sendo construído. A realização do contexto baseia-se na análise detalhada do contexto do negócio e de seu uso.

- **Especificação de Componentes**

  A especificação de componentes tem como objetivo principal criar um conjunto de modelos que, juntos, descrevem as propriedades de um componente que são visíveis externamente.

  A especificação pode ser vista como um contrato entre o componente e seus clientes e servidores. Os serviços fornecidos pelo componente definem o contrato com os clientes e os serviços usados pelo componente que são oferecidos por outros componentes definem o contrato com os servidores.

  A especificação de um componente define, portanto, todos os requisitos que uma realização do componente deve satisfazer.

  Ao final da etapa de especificação de componentes, uma descrição completa das operações do componente estará disponível, tornando possível desenvolver também os casos de teste funcional. Desenvolver os casos de teste funcional durante esta etapa não melhora apenas a qualidade dos casos de teste, melhora também a qualidade dos artefatos desenvolvidos.

- **Realização**

  A etapa de realização tem por objetivo principal descrever como o componente satisfaz os requisitos identificados na sua especificação.

- **Componentização**

  A etapa de componentização é composta de duas atividades principais:

  - Definição da árvore de componentes: identifica uma hierarquia de componentes, onde o componente no topo da hierarquia é composto por sub-componentes que se encontram nos níveis mais inferiores.
  - Definição das dependências entre componentes na árvore de componentes: identifica que componentes usam serviços de outros componentes dentro da árvore de componentes.

  Nesta etapa é produzido um diagrama que representa a árvore de componentes. Na árvore, os componentes são representados como pacotes UML, com o estereótipo ≪Komponent≫. No topo da árvore deve existir um pacote sem o estereótipo que representa o contexto do sistema em desenvolvimento. Na árvore são representados ainda os relacionamentos de dependências entre os componentes. A indicação ≪acquires≫ no relacionamento entre dois componentes define que um componente usa serviços do outro componente.

  A árvore de componentes é usada para refinar os artefatos produzidos nas etapas de especificação e realização. Os casos de teste definidos nas etapas anteriores devem também ser revisados considerando-se a árvore de componentes.

- **Implementação**

  A implementação tem por objetivo transformar uma representação abstrata do software em uma representação mais concreta que possa ser executada por ferramentas automáticas.

  No KobrA, esta transformação não se dá através de um único grande passo como na maioria dos métodos orientados a objetos. Estão envolvidos nesta transformação dois passos: refinamento e tradução. No refinamento, os modelos produzidos na etapa de realização são descritos em um nível maior de detalhes, produzindo uma representação mais próxima do código-fonte. A tradução transforma a representação obtida no passo anterior em representação textual escrita em alguma linguagem de programação.

Comentários Finais sobre o KobrA

A metodologia KobrA se mostra bastante completa e abrangente, com grande poder de modelagem. Ela apresenta um diferencial importante quando comparada às outras metodologias apresentadas neste capítulo, que é a preocupação com a qualidade do software produzido, demonstrada com a sugestão de inspeções e revisões contínuas dos artefatos de software.

Na opinião dos autores, a metodologia é simples e prática por sugerir a construção apenas dos artefatos realmente necessários e por especificar claramente como estes artefatos são construídos. Na nossa opinião, a metodologia não é tão simples assim, pois acreditamos que ela exige um esforço não trivial para sua aprendizagem.

Ainda segundo a nossa visão, embora a preocupação com a qualidade do produto seja evidente, acreditamos que a metodologia precisa ser complementada com aspectos relacionados ao planejamento e execução dos testes do produto, já que apenas as inspeções e revisões não são suficientes para fornecer um alto grau de confiabilidade ao mesmo.

2.3 Considerações Finais

Neste capítulo apresentamos o conceito e características de componentes de software na visão de diferentes autores. Apresentamos ainda algumas metodologias de desenvolvimento de componentes que foram consideradas como possíveis candidatas para apoiar o método de teste proposto neste trabalho.

Cada metodologia apresentada foi discutida em termos de suas características principais, suas vantagens e deficiências. A metodologia realmente escolhida para apoiar o método não foi explorada neste capítulo, sendo apresentada no Capítulo 4.

No próximo capítulo fornecemos as bases teóricas de teste funcional, apresentando a terminologia comumente usada e as principais técnicas de teste existentes para verificação de objetos e componentes.

Capítulo 3

Teste Funcional: Uma Visão Geral

À medida que cresce a utilização de sistemas de software na execução de tarefas críticas, cresce também a exigência de produtos cada vez mais qualificados. Teste é uma das técnicas de verificação de software mais utilizadas na prática. Se usado de forma efetiva, pode fornecer indicadores importantes sobre a qualidade e confiabilidade de um produto.

Este capítulo apresenta os principais conceitos relativos a teste bem como a terminologia comumente usada. O capítulo delineia ainda o estado da arte das técnicas de teste funcional existentes, especialmente as técnicas para verificação de objetos, componentes e arquiteturas.

3.1 Teste de Software

Embora o principal objetivo do teste de software seja fornecer garantias de qualidade, segundo B. Beizer [Bei95], teste de software pode ser realizado com diferentes propósitos. O teste sujo ou negativo é um tipo de teste cujo principal propósito é a falsificação, ou seja, utilizamos este teste para demonstrar a presença de bugs. Já o teste limpo ou positivo é um tipo de teste cujo principal propósito é a certificação, ou seja, desejamos demonstrar que o software testado está livre de bugs.

Ainda segundo B. Beizer [Bei95], três estratégias podem ser utilizadas para se realizar teste de software, sujo ou limpo:

- Teste Funcional: Checa se um programa está de acordo com a sua especificação, independentemente do código fonte. Teste funcional é também conhecido como teste caixa-preta ou teste comportamental.

- Teste Estrutural: Baseia-se na estrutura do objeto testado. Requer, portanto, acesso completo ao código fonte do programa para ser realizado. É também conhecido por teste caixa-branca.

- Teste Híbrido: Combina as duas estratégias de teste anteriores.

3.2 Terminologia de Teste

Teste de software apresenta um vocabulário bem estabelecido. A seguir, são listados alguns dos termos mais comumente usados [Bei95; Som96; MS01]:

- Defeito, "fault", "bug": Os termos são equivalentes e são usados para indicar que o objeto em execução apresenta algum tipo de comportamento não esperado, ou seja, a saída produzida pelo objeto não coincide com o resultado previsto. Isso pode acontecer devido a uma entrada errada, uma previsão errada do resultado, ou mesmo um erro no objeto executado, dentre outras situações.

- Erro: Diferença entre o obtido e o esperado, ou seja, um estado intermediário incorreto ou um resultado inesperado na execução do objeto, em função da presença de um "bug".

- Oráculo: Procedimento responsável por decidir se um teste obteve ou não sucesso.

- Caso de Teste: Aspecto ou funcionalidade a ser testado em um sistema, expresso por critérios de entrada e aceitação.

- Dados de Teste: Valores que servem de entrada para a execução de um caso de teste.

- Validação: Processo que avalia se a especificação de um objeto está de acordo com o domínio do problema. Este processo assegura que o software produzido é o software correto.

- Verificação: Processo que avalia se a implementação de um objeto satisfaz os requisitos declarados em sua especificação. Assume-se neste processo que a especificação está correta. Este processo assegura que o software está sendo produzido corretamente.

- Testabilidade: Propriedade fundamental que engloba todos os aspectos que facilitam a elaboração e execução dos testes do software.

- Teste de Unidade: Teste realizado para garantir que uma unidade de software implementa corretamente o seu projeto e está pronta para ser integrada ao sistema. Sub-rotinas e funções chamadas pela unidade são assumidas funcionar corretamente e são substituídas por simuladores.

- Teste de Integração: Teste realizado para garantir que as unidades testadas individualmente interagem corretamente.

- Teste de Componentes: Teste realizado para garantir que um componente de software está de acordo com a sua especificação. Teste de componentes não é o mesmo que o teste de unidade porque no teste de componentes, os componentes associados, as funções e sub-rotinas chamadas, são todos testados como um único agregado.

- Teste de Regressão: Teste realizado para verificar se mudanças promovidas no software introduziram novos erros.

3.3 Princípios Genéricos de Teste

Um processo de teste completo pode envolver as seguintes etapas:

- Planejamento: Nesta etapa são definidos que tipos de teste serão realizados e quais as expectativas com relação aos testes realizados.

- Especificação: Nesta etapa são gerados os modelos de teste dos quais são derivados os casos de teste, dados e oráculos.

- Construção: Nesta etapa os artefatos necessários para execução do teste são criados. Os casos de teste e os oráculos identificados na etapa anterior são implementados utilizando-se alguma linguagem de programação.

- Execução: Nesta etapa são executados os casos de teste desenvolvidos para o sistema na etapa anterior, sendo fornecidos os dados selecionados na etapa de especificação.

- Análise dos Resultados: Nesta etapa os oráculos gerados na etapa de construção são utilizados para analisar se o programa em teste falhou ou não no teste executado.

Além das etapas acima, em se tratando de teste de componentes, uma quinta etapa pode ser realizada. Nesta etapa, os artefatos e resultados dos testes obtidos nas etapas anteriores devem ser empacotados juntamente com o componente desenvolvido a fim de fornecer ao usuário do componente facilidades para desenvolver novos testes, caso seja necessário [Mac00].

Para B. Beizer [Bei95], grafos são os modelos de teste mais genéricos que podem ser usados no processo de derivação dos casos de teste, dados e oráculos. Ao construir um grafo deve-se identificar:

- Os objetos de interesse, representados no grafo por nós.
- As relações existentes entre os nós.
- Quais objetos estão relacionados a que outros objetos, representados no grafo por arcos.
- Propriedades que podem ser associadas aos arcos, ou seja os pesos dos arcos.

Os grafos são usados para ajudar a projetar os casos de teste do software. Diferentes aspectos do software podem ser abordados pelo grafo a depender da definição dos nós, relações, links e pesos. Alguns modelos úteis que abordam aspectos específicos do software podem ser encontrados em [Bei95]:

- Modelo de Fluxo de Transação: Neste modelo os objetos de interesse são os passos envolvidos no processamento de uma transação. Cada passo é representado por um nó no grafo. A relação existente entre os nós é "É seguido por"que indica a seqüência de execução dos passos. Os arcos conectam passos que seguem um ao outro.
- Modelo de Estado Finito de Menus: Os objetos neste modelo são os menus que aparecem na tela. Para cada menu, existe um nó no grafo. A relação existente entre os nós é "Pode alcançar diretamente", e indica que existe uma opção no menu que ao ser selecionada fará com que o programa exiba um novo menu apropriado à opção

selecionada. Existe um arco ligando cada nó com as possíveis opções do menu representado pelo nó.

- Modelo de Fluxo de Dados: Os nós deste grafo representam objetos de dados. A relação entre os nós é "É usado para calcular o valor de". Por exemplo, em X = 2Y + Z, Y e Z são usados para calcular o valor de X. Os links indicam que o valor de um nó é usado para calcular o valor de outro nó.

Tendo definido o grafo mais apropriado para o software em teste, os casos de teste podem ser projetados. O projeto dos casos de teste envolve a definição de uma estratégia de cobertura, ou seja, o que deve ser coberto no grafo, se apenas os nós, que é uma opção não muito útil para revelar bugs, ou se os links do grafo devem ser cobertos. A cobertura de links é uma estratégia mais forte para revelar bugs do que a cobertura de nós, já que a cobertura de links revela o caminho que o software percorreu até a ocorrência do bug.

Em geral, as técnicas de teste existentes traduzem uma especificação do software em um grafo que serve de modelo de teste e usam este grafo para fazer a seleção dos casos de teste.

3.4 Teste Funcional

Teste funcional representa um avanço significativo na área de teste de software. Por se fundamentar em uma especificação, o teste funcional está diretamente relacionado ao que o software em teste deveria fazer, ao invés do que ele realmente faz.

Segundo J. Chang et. al. [CR99], teste funcional é freqüentemente mais fácil de compreender e realizar do que as técnicas de teste tradicionais baseadas em código, porque o teste funcional baseia-se em uma descrição "black-box" da funcionalidade do software, enquanto que as técnicas de teste estrutural exigem um detalhado conhecimento da implementação.

Além disso, o teste funcional pode revelar falhas que freqüentemente não são observadas ao se utilizar as técnicas baseadas em código, como por exemplo, inconsistência entre os modelos do software.

O estudo de técnicas de teste funcional não é recente, e embora muitos trabalhos tenham sido produzidos, poucos apresentam utilidade prática, especialmente pela falta de ferramentas para auxiliar no processo de teste. Muito esforço ainda precisa ser empreendido a fim

de tornar o teste funcional de fato uma ferramenta útil na prática. Para isto, é fundamental o desenvolvimento de técnicas para geração automática de artefatos de teste, como casos de teste, dados e oráculos de teste, bem como métodos práticos e ferramentas. Para alcançar tal automatização é imprescindível que as especificações do software, a partir das quais os artefatos de teste são gerados, sejam feitas usando notações com semântica clara e precisa.

Teste funcional, é portanto, um campo ainda em aberto, que merece atenção pelos benefícios que podem ser obtidos com sua utilização.

3.5 Teste de Software Orientado a Objetos

Dentro do mundo de software orientado a objetos muito se pesquisou sobre técnicas de análise, projeto e programação, mas muito pouca atenção foi dada às técnicas de teste. Entretanto, é claro que o paradigma orientado a objetos não eliminou a necessidade de realização de testes e talvez esta etapa do ciclo de vida do software seja até mais importante para software orientado a objetos do que para software tradicional, tendo em vista que o reuso é um dos pontos fortes do paradigma orientado a objetos.

Apesar disso, a maior parte da pesquisa realizada na área de testes tem sido voltada para software orientado a funções [1]. Embora os métodos tradicionais de teste sejam eficientes, eles não podem ser usados em software orientado a objetos sem que algum tipo de adaptação seja promovida. A natureza do software orientado a objetos introduz novos aspectos que não existem no software tradicional e que devem ser considerados no processo de teste.

O primeiro aspecto relevante, segundo S. Barbey et. al. [BAS94], é que no paradigma orientado a objetos, a unidade básica de teste é a classe e não mais subprogramas ou rotinas. Além disso, as operações de uma classe não podem ser testadas isoladamente, já que as operações de uma classe podem interagir entre si para modificar o estado do objeto que chama estas operações.

S. Barbey et. al. [BAS94] ressaltam que aspectos do paradigma orientado a objetos, tais como encapsulamento, herança e polimorfismo, introduzem novos problemas que precisam ser considerados no processo de teste do software.

[1] Um programa orientado a funções é um programa decomposto funcionalmente em subprogramas que implementam independentemente os serviços do programa [BAS94].

Embora a orientação a objetos traga novos desafios para os métodos de teste, ela pode apresentar também alguns benefícios. Considerando-se, por exemplo, que uma super-classe tenha sido previamente testada, o esforço empregado para testar classes derivadas pode ser consideravelmente reduzido, conseqüentemente o custo associado à atividade de teste pode ser também menor [MS01].

A seguir, tratamos alguns aspectos do paradigma orientado a objetos e suas implicações no processo de teste do software.

3.5.1 Encapsulamento

Encapsulamento é ao mesmo tempo um benefício e um obstáculo para as atividades de teste. É um benefício porque a noção de encapsulamento isola uma classe do resto do sistema tornando a definição das unidades de teste muito mais fácil. Além disso, a presença do encapsulamento esconde os aspectos relacionados à implementação das classes, levando o testador a não se preocupar com a estrutura interna das mesmas.

Por outro lado, a opacidade decorrente da presença do encapsulamento, faz com que o estado interno de um objeto não possa ser verificado diretamente. Apenas através de chamadas a operações da classe é que se pode observar o estado interno do objeto. Dessa forma o teste tem que confiar que as operações de verificação do estado interno do objeto estão corretas. Para contornar este problema, em [BAS94] são propostas duas alternativas:

- Quebrar o encapsulamento do código através do uso de características da linguagem, tais como classes friends em C++, cujas operações têm completa visibilidade das características de outras classes, ou modificando-se a classe testada ou adicionando-se uma subclasse com operações que forneçam ao testador a visibilidade das propriedades escondidas.

- Usar equivalência de cenários: diferentes seqüências de operações que devem colocar o objeto no mesmo estado. Os estados resultantes podem então ser comparados usando-se operações que não influenciam no estado do objeto, ou ainda aplicando-se outra equivalência de cenários.

3.5.2 Herança

O mecanismo de herança permite que uma classe herde propriedades de uma ou mais classes. A classe derivada é então refinada, adicionando-se novas propriedades ou removendo-se propriedades herdadas.

Embora o uso da herança permita reduzir em algumas situações o esforço empregado para testar as classes derivadas, considerando-se que a classe base tenha sido previamente testada, algumas das propriedades herdadas precisam ser testadas novamente no contexto da classe derivada. É preciso então empregar um método que aplique uma abordagem incremental; não testar tudo novamente, mas apenas aquele conjunto de propriedades cujo comportamento se diferencia da super-classe.

3.5.3 Polimorfismo

Este aspecto do paradigma orientado a objetos complica ainda mais as atividades de teste do software orientado a objetos. Já que chamadas polimórficas podem ser feitas a objetos de diferentes classes, é impossível determinar estaticamente qual código será executado como resultado da chamada polimórfica.

3.5.4 Algumas técnicas conhecidas

À medida que cresce a pesquisa em torno das técnicas de teste para software orientado a objetos, surgem algumas propostas bastante interessantes. Para software orientado a objetos, especificado usando UML, L. Briand e Y. Labiche [BL01] propõem a técnica TOTEM (Testing Object-orienTed systEms). Esta técnica baseia-se em artefatos UML, tais como Casos de Uso, Diagramas de seqüência ou Colaboração, Diagramas de Classes e Dicionários de Dados, para construir os modelos de teste, a partir dos quais são selecionados os casos de teste e os oráculos de teste são gerados. A proposta preocupa-se com a testabilidade do sistema ao exigir que toda e qualquer especificação seja feita utilizando OCL, ao invés do uso da linguagem natural. A proposta apresenta ainda um grande potencial para automação, o que lhe confere uma grande vantagem, já que a viabilidade dos testes cresce à medida que o processo é automatizado. Uma outra vantagem é o uso de artefatos comumente desenvolvidos em projetos orientados a objetos, como diagramas de casos de uso e de sequência, para

derivar os modelos de teste.

Uma outra técnica interessante para gerar casos de teste a partir dos diagramas UML é proposta por P. Chevalley e P. Thevenod-Fosse [CTF01]. Este trabalho descreve uma técnica que adapta um método probabilístico, chamado de teste funcional estatístico, para gerar os casos de teste a partir dos diagramas de estado UML. A ênfase do trabalho está em produzir automaticamente os valores de entrada dos casos de teste e os resultados esperados, com ajuda da ferramenta Rose RealTime da Rational Software Corporation.

Para automatizar a geração dos casos de teste no ambiente Rose RealTime, o trabalho propõe que o modelo UML que representa o sistema em teste seja acrescido de duas cápsulas: a cápsula geradora, responsável pela geração dos valores de entrada e a cápsula coletora, responsável por coletar os resultados das transições cobertas no caso de teste.

Um algoritmo de distribuição funcional que gera os valores de entrada para os casos de teste de acordo com uma determinada distribuição de domínio deve ser implementado na cápsula geradora.

Neste processo dois arquivos são gerados: um arquivo contendo os resultados que se espera que sejam produzidos durante a simulação do modelo com os valores de entrada produzidos pela cápsula geradora, e um outro arquivo contendo especificações de uma classe Java que implementa um test driver produzido para gerenciar o processo de teste do programa Java. As saídas produzidas pelo programa são então, automaticamente comparadas com os resultados previstos fornecidos pelo modelo.

A existência de ferramentas de apoio à técnica proposta por P. Chevalley e P. Thevenod-Fosse é o grande diferencial desta técnica.

Entretanto, as técnicas aqui comentadas apresentam a desvantagem de serem técnicas de teste isoladas, ou seja, elas não estão integradas a um processo de desenvolvimento de software, o que pode dificultar a sua aplicação prática.

Além de UML, outras formas de especificação de software orientado a objetos foram também consideradas em outros trabalhos. Por exemplo, a técnica ASTOOT proposta em [DF94] baseia-se em uma especificação algébrica, onde as operações das classes são especificadas através de axiomas. Em [WTWS98] é proposta uma técnica de análise e teste de software orientado a objeto através do uso de Redes de Petri Coloridas. A técnica traduz a especificação do software em uma rede de petri colorida que é então analisada, testada e

simulada como um protótipo da especificação. Uma outra técnica de teste de software orientado a objetos é proposta em [BBP96]. Neste trabalho, considera-se que o software tenha sido especificados utilizando-se a linguagem CO-OPN/2 (Concurrent Object-Oriented Petri Nets).

3.6 Teste de Software Baseado em Componentes

O crescimento da incidência de sistemas baseados em componentes tem tornado notável a necessidade de se desenvolver técnicas que testem efetivamente tais sistemas.

De acordo com M. J. Harrold [Har00], o teste de sistemas baseados em componentes pode ser visto de duas perspectivas: a perspectiva do fornecedor do componente e a perspectiva do usuário do componente. O fornecedor vê o componente independentemente do contexto em que ele será utilizado e deve portanto, testar todas as configurações do componente de uma forma livre de contexto. Já para o usuário, o componente está inserido no contexto da aplicação. Interessa ao usuário portanto testar aqueles aspectos do componente que são relevantes no contexto da aplicação.

Algumas pesquisas têm sido conduzidas no sentido de adaptar técnicas de teste existentes para o contexto dos sistemas baseados em componentes. Alguns pesquisadores dedicam-se à pesquisa de técnicas voltadas para a perspectiva dos fornecedores, enquanto outros voltam-se para a perspectiva dos clientes.

Uma técnica interessante de teste de sistemas baseado em componentes é proposta em [HIM00]. Esta técnica propõe a geração e execução dos casos de teste de forma automática para sistemas baseados em componentes COM/DCOM e CORBA e especificados utilizando UML.

A técnica impõe que diagramas de estado UML sejam utilizados para descrever o comportamento dinâmico dos componentes e a comunicação entre componentes. O trabalho propõe uma notação para rotular as transições dos diagramas de estados, já que a notação UML atual não contempla o aspecto de comunicação entre os componentes.

A técnica propõe que um modelo de comportamento global do sistema seja construído a partir dos diagramas de estados individuais de cada componente. Os diagramas de estados são vistos como máquinas de estados finitas Mealy, ou seja, as saídas produzidas em função

da ocorrência de um evento são associadas à transição e não ao estado [Bei95]. A máquina de estado composta representando o comportamento global do sistema é obtida através da composição das máquinas individuais. As máquinas de estado individuais são equivalentes aos diagramas de estados.

Para gerar os casos de teste, a técnica usa o TDE - Test Development Environment, um produto desenvolvido pela Siemens Corporate Research. Esta ferramenta processa projetos de teste escritos em TSL - Test Specification Language, uma linguagem apropriada para especificação de testes.

O projeto de teste TSL é criado a partir do modelo comportamental global. Cada estado do modelo é mapeado em uma categoria ou partição do projeto de teste e cada transição é representada como uma escolha para a categoria ou partição.

A ferramenta TDE constrói um grafo direcionado que tem uma categoria ou partição na raiz e contém todos os diferentes caminhos de escolha. Um caso de teste é um possível caminho da raiz até a folha da árvore de alcançabilidade do grafo.

A linguagem TSL permite que o projetista dos testes indique os requisitos de cobertura. A ferramenta TDE então, cria casos de teste que satisfazem aos requisitos de cobertura especificados. Para teste de unidade, o critério de cobertura padrão é que todas as transições dentro do diagrama de estados sejam percorridas ao menos uma vez. Já para o teste de integração, apenas as transições que envolvem interação entre componentes precisam ser exercitadas.

A ferramenta TDE está integrada às ferramentas Rose2000 e Rose Real-Time 6.0, da Rational Software, podendo ser executada diretamente a partir destas ferramentas.

A execução automática dos casos de teste é realizada pela ferramenta TECS - Test Environment for Distributed Component-Based Software. Este ambiente foi especificamente projetado para testar componentes COM ou CORBA durante teste de integração e unidade. A ferramenta cria automaticamente os test drivers e fornece ao usuário meios de executar os testes interativamente através de uma interface gráfica ou em modo batch. As informações geradas durante a execução dos testes são armazenadas em um arquivo de controle XML e a ferramenta fornece diferentes visões dessas informações. A ferramenta permite ainda que outras visões sejam facilmente adicionadas.

Esta técnica tem a vantagem de ser apoiada por ferramentas, mas apresenta duas desvan-

tagens: primeiro, exige que sejam construídos máquinas de estados para representar o comportamento global do sistema. Essa exigência é ruim porque aumenta o esforço de modelagem do sistema. O ideal seria que a técnica utilizasse modelos que já tivessem sidos desenvolvidos durante a especificação do sistema. A segunda desvantagem é comum a todas as técnicas de teste que estudamos durante a pesquisa bibliográfica. A técnica está isolada, não havendo integração com qualquer processo de desenvolvimento.

3.7 Considerações Finais

Neste capítulo introduzimos os principais conceitos e termos relacionados a teste de software. Destacamos ainda uma série de técnicas de teste existentes para verificação de software orientado a objetos e baseado em componentes.

Durante a nossa pesquisa, encontramos ainda algumas técnicas de teste para verificação de componentes: em [BG01], componentes são especificados através de máquinas de estado finitas especiais, das quais são extraídos os casos de teste. O problema de testar componentes genéricos, independentemente do contexto de uso e instanciações específicas, com relação a especificações algébricas é investigado em [MS02]. Martins et. al. enfatizam a importância da testabilidade dos sistemas baseados em componentes [MTY01]. Os componentes são especificados através de modelos de fluxo de transação, dos quais são extraídos modelos e especificações de teste. Os casos de teste podem ser gerados, executados e analisados diretamente pelos usuários do componente.

Embora estes trabalhos tenham produzido resultados significativos, eles se concentram, na maioria das vezes, em técnicas isoladas. A integração, aplicação e contextualização de técnicas adequadas em um processo de desenvolvimento de componentes baseado no uso de UML, cobrindo as diferentes atividades de um processo de teste não parece ser o objetivo de nenhum dos trabalhos que encontramos. Essa idéia de integrar a atividade de teste ao processo de desenvolvimento foi explorada em [JO95], mas não se aplica a componentes, especialmente pelo fato do paradigma usado como apoio ser o modelo cascata, reconhecidamente ineficiente como prática de Engenharia de Software [Pre87].

Nesta primeira parte do trabalho fornecemos a fundamentação teórica necessária para a compreensão dos objetivos e importância prática do método aqui proposto. Os capítulos

seguintes apresentam de forma detalhada a construção do método e os resultados obtidos.

Capítulo 4

Metodologia de Desenvolvimento de Componentes

A utilização efetiva do método de teste proposto neste trabalho é intimamente dependente do processo de desenvolvimento e dos artefatos produzidos durante este processo. Na seleção da metodologia adotada no trabalho consideramos as metodologias Componentes UML [CD01], KobrA [ABB+01] e Catalysis [DW98] por serem as mais conhecidas e utilizadas metodologias de desenvolvimento de componentes. Baseando-se no aspecto simplicidade e facilidade de aplicação da metodologia, decidimos usar a metodologia Componentes UML, que foi complementada com algumas atividades ou artefatos que se mostraram necessários durante a aplicação prática da metodologia.

Durante a descrição da metodologia, utilizaremos um estudo de caso para exemplificar cada uma das etapas e os artefatos produzidos. O estudo de caso trata de um sistema de reservas para quartos de hotel. Tal sistema é uma aplicação computadorizada cujo objetivo principal é registrar reservas de quartos em qualquer hotel de uma cadeia de hotéis. O sistema deve ser capaz de fornecer quartos em hotéis alternativos quando o hotel desejado estiver lotado. Para agilizar o processo de reserva, o sistema deve fornecer meios de armazenar e disponibilizar dados de clientes que tenham realizado reservas anteriormente. Por fim, o sistema está integrado com um Sistema de Contas, desenvolvido previamente, responsável por computar a estadia e o consumo do cliente. Uma especificação incompleta do sistema usado como exemplo pode ser encontrada em [CD01]. Complementamos essa especificação durante o desenvolvimento deste trabalho. Os modelos produzidos estão distribuídos ao

longo dos capítulos.

A metodologia adotada consiste das seguintes etapas:

- Definição de Requisitos
- Modelagem de Componentes
- Materialização de Componentes
- Montagem da Aplicação
- Testes
- Distribuição da Aplicação

A Figura 4.1 fornece uma visão geral das etapas do processo.

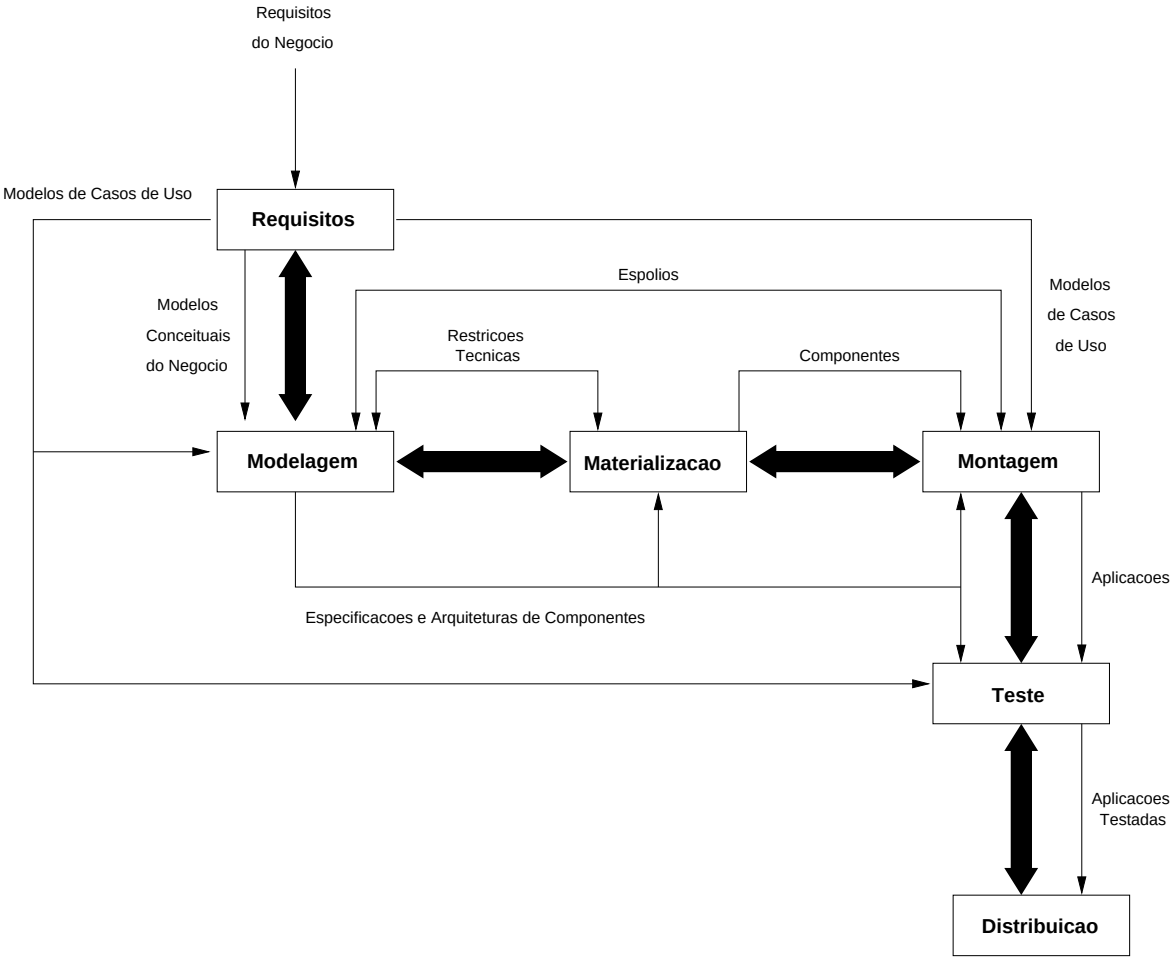

Figura 4.1: Visão Geral das Etapas do Processo de Desenvolvimento

4.1 Definição de Requisitos

O objetivo principal desta etapa é tornar o mais claro possível os objetivos do software antes de começar a construí-lo. Para tanto, propõe-se que sejam construídos os seguintes artefatos:

- Modelo de Processo do Negócio
- Modelo Conceitual do Negócio
- Modelo de Casos de Uso

4.1.1 Modelo de Processo do Negócio

O Modelo de Processo do Negócio é um diagrama de alto nível que fornece uma visão geral do funcionamento do negócio modelado. Este diagrama pode ser construído utilizando-se a notação de um diagrama de atividades UML ou qualquer outra notação semelhante. Um diagrama de atividades UML é uma espécie de fluxograma que fornece uma rica notação para demonstrar uma seqüência de atividades. Cada atividade é representada por um bloco no diagrama, e cada transição é disparada automaticamente após a execução da atividade [Lar98].

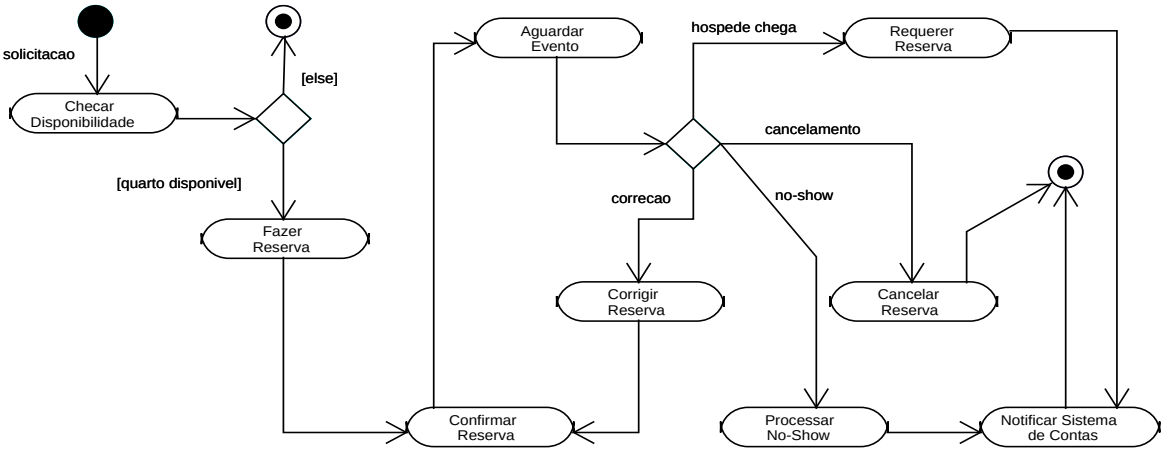

Figura 4.2: Modelo de Processo do Negócio para o Sistema de Reserva de Hotel

O diagrama da Figura 4.2 descreve as atividades envolvidas no processo de reservar um quarto de hotel. Este diagrama introduz uma série de termos importantes que precisam se

tornar claros para melhorar a compreensão do negócio. A descrição destes termos é feita através do Modelo Conceitual do Negócio (ver seção 4.1.2).

Para complementar o Modelo de Processo do Negócio, pode-se produzir um texto descrevendo, em alto nível, quais serão as principais funções do sistema, e como o sistema deverá funcionar, do ponto de vista de seus usuários. Para o Sistema de Reserva de Hotel, tal texto poderia ser o seguinte:

O sistema requerido deve permitir realizar reserva em qualquer hotel da cadeia. Reservas podem ser realizadas por telefone, em uma central de reservas ou diretamente no hotel, ou via internet. O sistema deverá ser capaz de oferecer quartos em hotéis alternativos quando o hotel desejado estiver lotado. Cada hotel tem um administrador de reservas, responsável por controlar as reservas no hotel, mas usuários autorizados podem também fazer reservas no sistema. O tempo desejável gasto no processo de realizar uma reserva é de até 3 (três) minutos. A fim de acelerar o processo, dados de clientes devem ser armazenados e disponibilizados pelo sistema.

4.1.2 Modelo Conceitual do Negócio

O Modelo Conceitual do Negócio contém os principais conceitos envolvidos no negócio modelado e seus relacionamentos. Este modelo pode ser representado utilizando-se um diagrama de classes UML. Um diagrama de classes UML descreve os tipos de objetos em um sistema e os vários tipos de relacionamentos existentes entres eles [FS98].

A Figura 4.3 mostra um possível modelo conceitual para o Sistema de Reserva. É importante ressaltar que durante a construção deste modelo deve-se estar concentrado no problema a ser solucionado e não na solução do mesmo. Neste modelo, portanto, podem existir elementos que não necessariamente farão parte do sistema de software a ser construído. Gradativamente, tais elementos serão eliminados do modelo, à medida que o modelo é refinado em outras etapas.

Deve-se considerar ainda a possibilidade de construir um Dicionário de Dados, onde os termos mais complexos encontrados no Modelo Conceitual podem ser melhor explicados.

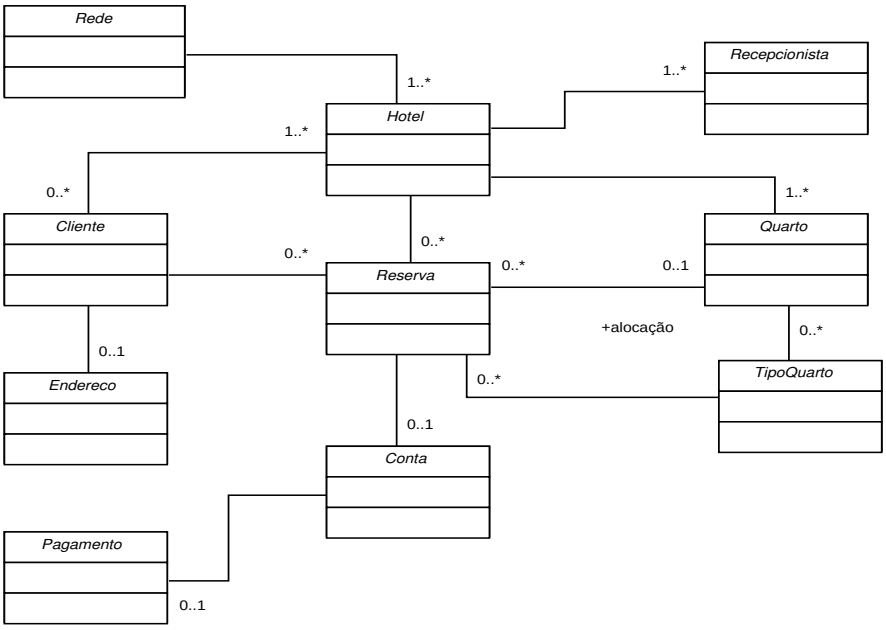

Figura 4.3: Modelo Conceitual do Negócio para o Sistema de Reserva de Hotel

4.1.3 Diagrama de Casos de Uso

A partir da visão adquirida do funcionamento geral do software deve-se definir claramente que funções são de responsabilidade do software e que funções são de responsabilidade de pessoas ou sistemas externos ao software. Neste momento, deve-se então identificar atores e papéis e definir os casos de uso do sistema.

A Figura 4.4 apresenta os atores e papéis definidos para o Sistema de Reserva de Hotel.

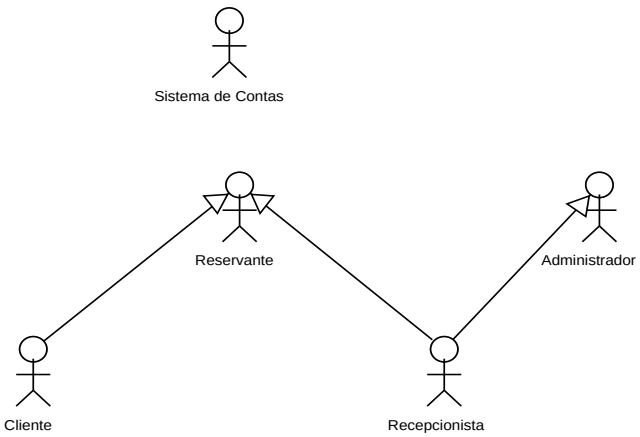

Figura 4.4: Atores e Papéis definidos para o Sistema de Reserva de Hotel

Um caso de uso é uma especificação funcional do software, onde ainda não há preocupação com a estrutura e organização interna do mesmo. Um caso de uso representa uma interação entre um usuário e o sistema [FS98].

A Figura 4.5 apresenta o diagrama de casos de uso para o Sistema de Reserva.

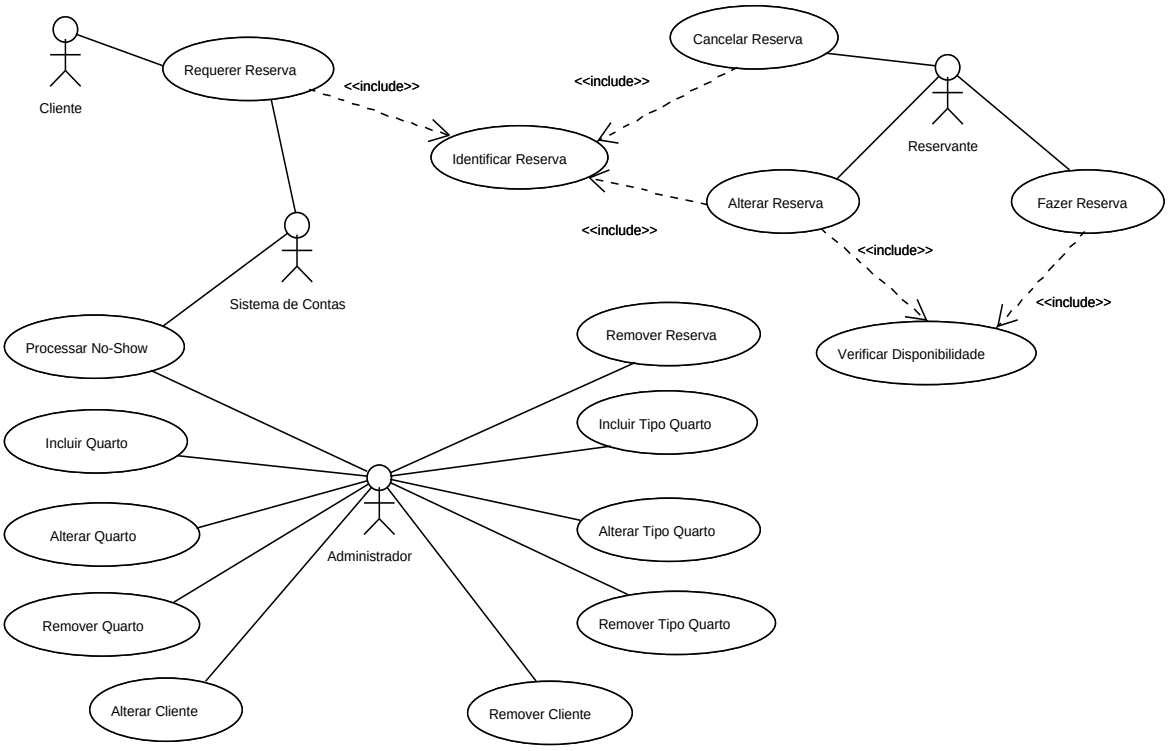

Figura 4.5: Diagrama de Casos de Uso para o Sistema de Reserva de Hotel

Cada caso de uso identificado no Diagrama de Casos de Uso deve apresentar uma descrição textual de como o ator do caso de uso interage com o sistema. Uma descrição de caso de uso deve conter as seguintes informações:

- O nome do caso de uso.
- O nome do ator responsável por iniciar o caso de uso.
- Uma breve descrição do objetivo do caso de uso.
- A seqüência numerada dos passos que descrevem o cenário principal.

Um cenário é uma seqüência específica de ações e interações entre atores e o sistema [Lar98]. O cenário principal descreve a situação mais comum do caso de uso, onde tudo

ocorre como esperado. Entretanto, podem existir cenários alternativos que também devem ser incluídos na descrição do caso de uso. Cada cenário alternativo é especificado através de extensões que são descritas separadamente do cenário principal. Cada extensão deve conter as seguintes informações:

- O número do passo, no cenário principal, ao qual a extensão se aplica.
- A condição que deve ser testada antes do passo ser executado. Se a condição for avaliada verdadeira, a extensão é executada, caso contrário, a extensão é ignorada e os passos no cenário principal seguem normalmente.
- Uma seqüência numerada dos passos que descrevem a extensão.

A descrição textual do caso de uso Fazer Reserva do Sistema de Reserva é fornecida como exemplo:

Nome: **Fazer Reserva**

Ator: **Reservante**

Objetivo: **Reservar quarto(s) em um hotel.**

Cenário Principal:

1. Reservante solicita reserva.
2. Reservante seleciona, em qualquer ordem, datas e tipo do quarto.
3. Sistema informa preço ao Reservante.
4. Reservante confirma a reserva.
5. Reservante fornece nome e CEP.
6. Reservante fornece endereço de e-mail.
7. Sistema faz a reserva e gera um código para a reserva.
8. Sistema revela o código da reserva para o Reservante.
9. Sistema cria e envia uma confirmação por e-mail.

Extensões:

3. Quarto não disponível.
   (a) Termina.
4. Reservante rejeita oferta de preço.
   (a) Termina.
6. Cliente já está registrado (baseando-se no nome e CEP).
   (a) Continua no passo 7.

Nesse exemplo, os passos 1 e 2 são sempre executados. Ao chegar no passo 3, a condição "Quarto não disponível" é avaliada. Se avaliada para verdadeiro é executado o passo (a) da extensão 3, indicando que o uso do sistema é terminado porque não existe quarto disponível para a reserva. Caso contrário, o passo 3 do cenário principal é executado. O mesmo acontece com o passo 4. O passo 6 tem uma diferença. Se a expressão da extensão 6 for avaliada como verdadeira o uso não é terminado, e continua no próximo passo, como indica o passo (a) da extensão 6. Essa extensão apenas indica que o passo 6 do cenário principal não é executado caso a condição da extensão seja avaliada verdadeira.

Uma descrição textual como essa deve ser produzida para cada caso de uso definido no diagrama de casos de uso. Tendo produzido essas informações, podemos partir para a modelagem dos componentes.

4.2 Modelagem dos Componentes

Nesta etapa, pretende-se gerar um conjunto de especificações de componentes e interfaces que serão reunidos posteriormente para dar origem a uma aplicação adequada ao domínio do problema. Uma interface é um conjunto de operações que caracterizam o comportamento visível externamente do componente [RJB98].

A fim de alcançar este objetivo, a etapa de especificação de componentes está subdividida em três atividades:

- Identificação dos Componentes
- Identificação das Interações entre os Componentes
- Especificação dos Componentes

4.2.1 Identificação dos Componentes

A identificação dos componentes é o primeiro estágio da etapa de Modelagem e tem como objetivo principal gerar um conjunto inicial de especificações de interfaces e componentes, organizados em uma proposta inicial de arquitetura de componentes. A ênfase deste estágio está, portanto, em descobrir que informações precisam ser gerenciadas, quais interfaces são necessárias para gerenciar tais informações, quais componentes são necessários

para prover as funcionalidades do sistema e como estas interfaces e componentes poderão interagir.

A metodologia considera que os sistemas desenvolvidos apresentam uma arquitetura distribuída em camadas. Tipicamente, são consideradas as seguintes camadas:

- Interface com o Usuário: apresenta informações para o usuário e captura suas requisições para o sistema.
- Diálogo com o Usuário: gerencia as seções de diálogo com o usuário.
- Serviços do Sistema: representa o sistema externamente, ou seja fornece uma visão dos serviços disponíveis no sistema e controla o acesso a esses serviços.
- Serviços do Negócio: implementa as informações, regras e transformações essenciais do negócio.

As duas últimas camadas formam a idéia do sistema e são independentes de uma interface com o usuário específica. Quando uma interface com o usuário é acoplada ao sistema, obtém-se uma aplicação, como mostrado na Figura 4.6. A metodologia considera ainda que os componentes desenvolvidos se encaixam nas duas camadas inferiores.

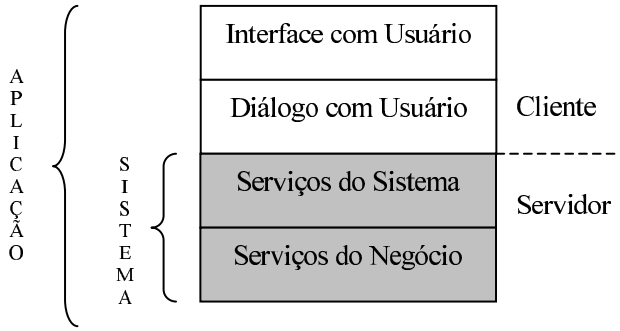

Figura 4.6: Camadas Arquiteturais

A partir desta idéia de arquitetura em camadas, procuramos identificar inicialmente as interfaces do sistema, que representam as funcionalidades principais da aplicação e estão encaixadas na camada de Serviços do Sistema. Estas interfaces são derivadas dos casos de uso. A regra geral é que para cada caso de uso identificado seja definida uma interface. Entretanto, casos de uso relacionados podem ser reunidos em uma única interface. Seguindo essa regra,

identificamos duas interfaces de sistema para o Sistema de Reserva de Hotel: IFReserva e IFCadastros, como mostra a Tabela 4.1. Essa tabela mostra também o mapeamento entre os casos de uso e as interfaces identificadas.

Interfaces de Sistema	*Casos de Uso*
IFReserva	Fazer Reserva
	Alterar Reserva
	Cancelar Reserva
	Requerer Reserva
	Processar No-Show
	Remover Reserva
IFCadastros	Incluir/Alterar/Remover Quarto
	Incluir/Alterar/Remover Tipo Quarto
	Alterar/Remover Cliente

Tabela 4.1: Mapeamento entre os casos de uso e as interfaces de sistema

Tendo identificado as interfaces do sistema, passamos a investigar as operações que devem ser fornecidas por essas interfaces. A orientação da metodologia é analisar os passos definidos na descrição textual dos casos de uso. Em geral, cada passo definido na descrição textual do caso de uso poderá dar origem a uma ou mais operações da interface. Alguns passos podem também ser reunidos para dar origem a uma única operação. A Figura 4.7 exemplifica o mapeamento do caso de uso Fazer Reserva para a interface de sistema IFReserva.

Nesse exemplo, os passos 1, 2 e 3 da descrição textual do caso de uso Fazer Reserva originaram a operação getPrecoReserva, que obtém o valor referente à estada no hotel, no tipo de quarto e período informados. Os demais passos do caso de uso deram origem à operação fazerReserva, que efetivamente realiza a reserva com os dados fornecidos pelo cliente.

Em seguida, interfaces relacionadas ao negócio do sistema devem ser identificadas. Essas interfaces representam as informações que devem ser tratadas pelo sistema e estão localizadas na camada de Serviços do Negócio. O processo proposto para identificar as interfaces do negócio é o seguinte:

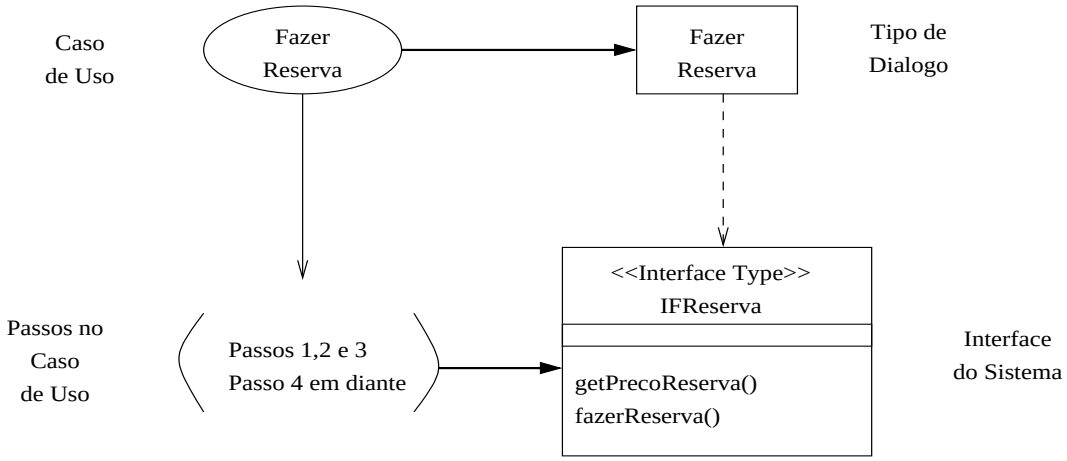

Figura 4.7: Mapeamento do Caso de Uso Fazer Reserva do Sistema de Reserva de Hotel para a Interface IFHotel

1. Criar um modelo de tipos do negócio. O modelo de tipos do negócio é representado por um diagrama de classes UML e deve conter as informações específicas que o sistema deverá gerenciar. O modelo de tipos do negócio é criado, inicialmente, copiando-se o modelo conceitual do negócio, produzido na etapa de Definição de Requisitos (ver Seção 4.1.2).

2. Refinar o modelo de tipos do negócio, adicionando-se ou removendo-se elementos até que o modelo apresente o escopo adequado. A Figura 4.8 mostra o Modelo de Tipos refinado para o Sistema de Reserva de Hotel.

3. Especificar regras de negócio e restrições adicionais. Esses elementos devem preferencialmente ser definidos utilizando-se OCL (Object Constraint Language) [WK99], por se tratar de uma linguagem declarativa que nos permite construir expressões lógicas, o que tornará as especificações mais claras e precisas.

4. Identificar os tipos centrais do negócio. Devem ser considerados tipos centrais, os tipos que possuem existência independente dos outros tipos dentro do negócio. Os tipos não centrais têm a finalidade de fornecer mais detalhes sobre os tipos centrais.

5. Criar interfaces de negócio para cada tipo central do negócio e adicioná-las ao modelo de tipos do negócio, criando um diagrama de responsabilidades de interface. A regra geral é criar uma interface para cada tipo central. Cada interface criada gerencia a

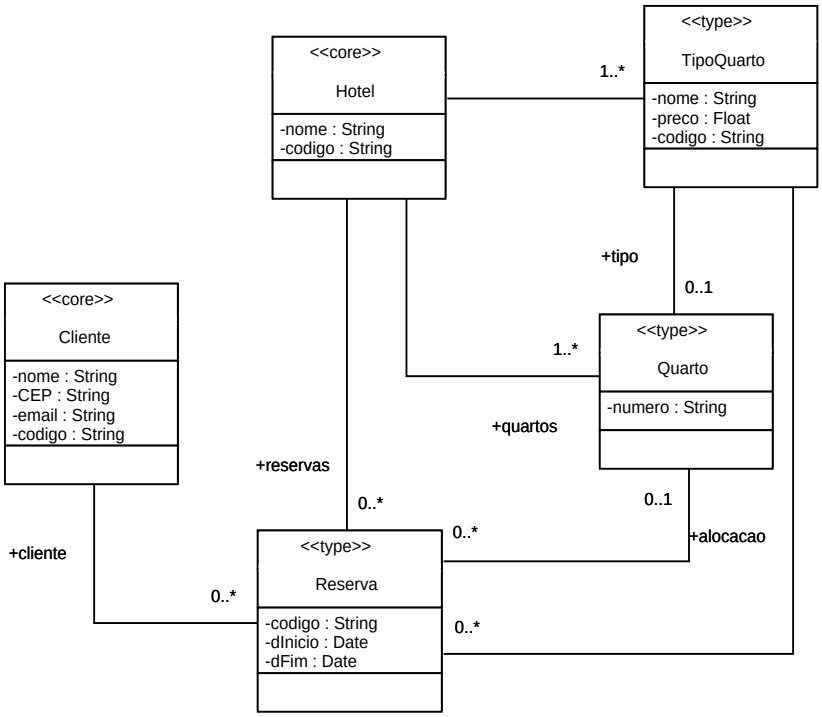

Figura 4.8: Modelo de Tipos de Negócio Refinado para o Sistema de Reserva de Hotel

informação representada pelo tipo central e pelos tipos detalhadores do tipo central. A Figura 4.9 apresenta o Diagrama de Responsabilidades de Interface para o Sistema de Reserva de Hotel.

A Tabela 4.2 mostra as interfaces de negócio identificadas para o Sistema de Reserva e os tipos de informação gerenciados por cada interface.

Interfaces de Negócio	*Tipos de Informação*
IFHotel	Hotel
	Reserva
	Quarto
	Tipo Quarto
IFCliente	Cliente

Tabela 4.2: Interfaces de negócio e o tipos de informação gerenciados por elas

6. Organizar em pacotes de interfaces as interfaces de sistema e de negócio identificadas

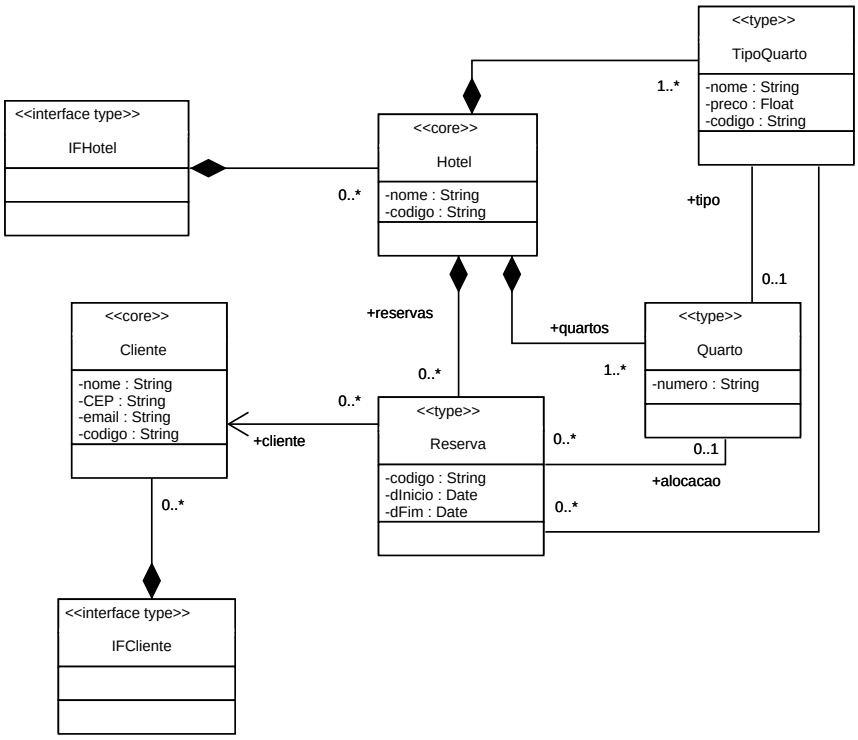

Figura 4.9: Diagrama de Responsabilidades de Interface para o Sistema de Reserva de Hotel

nos passos anteriores. Os tipos de dados relacionados a cada interface devem também ser armazenados nos pacotes adequados. Essa organização em pacotes fornece uma visão clara das interfaces e ajuda a construir mais tarde o modelo de informação de cada interface, além de servir de base para a especificação dos componentes.

Tendo identificado as interfaces de sistema e de negócio, é hora de criar um conjunto inicial de especificações de componentes e pensar numa forma de combinar estes componentes a fim de formar o sistema. Neste estágio, deve-se lembrar que um componente deve ser uma unidade de fácil substituição e gerenciamento, portanto, deve-se ter muito cuidado na escolha dos componentes que farão parte do sistema.

A regra geral é que para cada interface identificada anteriormente seja criada uma especificação de componente, entretanto, em algumas situações, uma especificação de componente pode suportar múltiplas interfaces.

A Figura 4.10 mostra um conjunto inicial dos componentes de sistema para o Sistema de Reserva de Hotel. Resolvemos criar um único componente - Sistema de Reservas - para suportar as interfaces de sistema IFReserva e IFCadastros. Considerando as questões de

facilidade de entrega e substituição de componentes, um segundo componente - Sistema de Contas - foi criado para suportar a interface IFContas, que gerencia a interação com o Sistema de Contas. Supomos nesta etapa do desenvolvimento, a partir da análise das descrições dos casos de uso, que o componente Sistema de Reservas terá interação com a interface externa IFContas e as interfaces de negócio IFHotel e IFClientes. Por esse motivo, acrescentamos uma dependência entre o componente e estas interfaces.

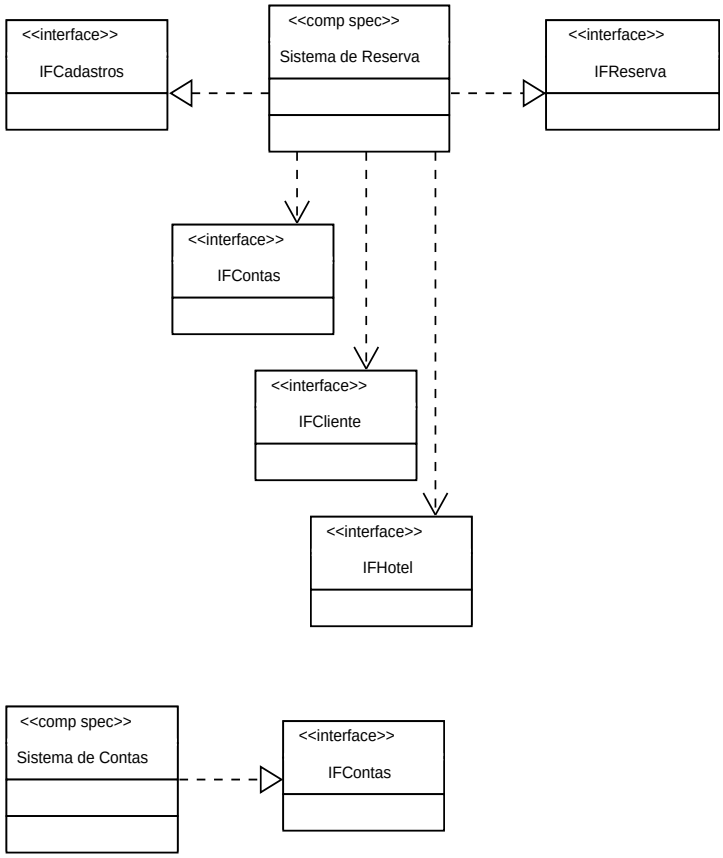

Figura 4.10: Especificação dos Componentes de Sistema para o Sistema de Reserva de Hotel

A Figura 4.11 mostra um conjunto inicial de componentes de negócio para o Sistema de Reserva de Hotel. Foi criado um componente para cada interface de negócio identificada, já que as interfaces gerenciam informações completamente independentes.

A última atividade da etapa de Identificação de Componentes é reunir as especificações de componentes produzidas anteriormente em uma arquitetura inicial. Através da arquitetura é possível identificar que interfaces são suportadas por cada especificação de componente e as dependências entre os componentes e as interfaces. A Figura 4.12 apresenta uma arquite-

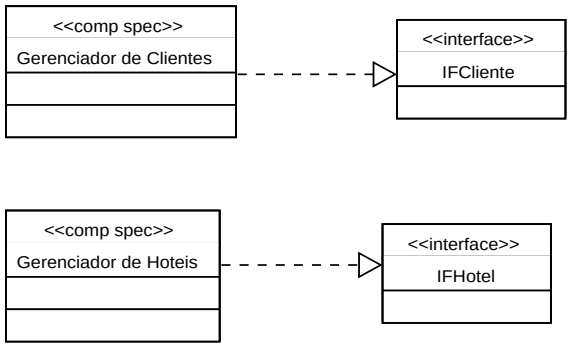

Figura 4.11: Especificação dos Componentes de Negócio para o Sistema de Reserva de Hotel

tura inicial de componentes para o Sistema de Reserva de Hotel.

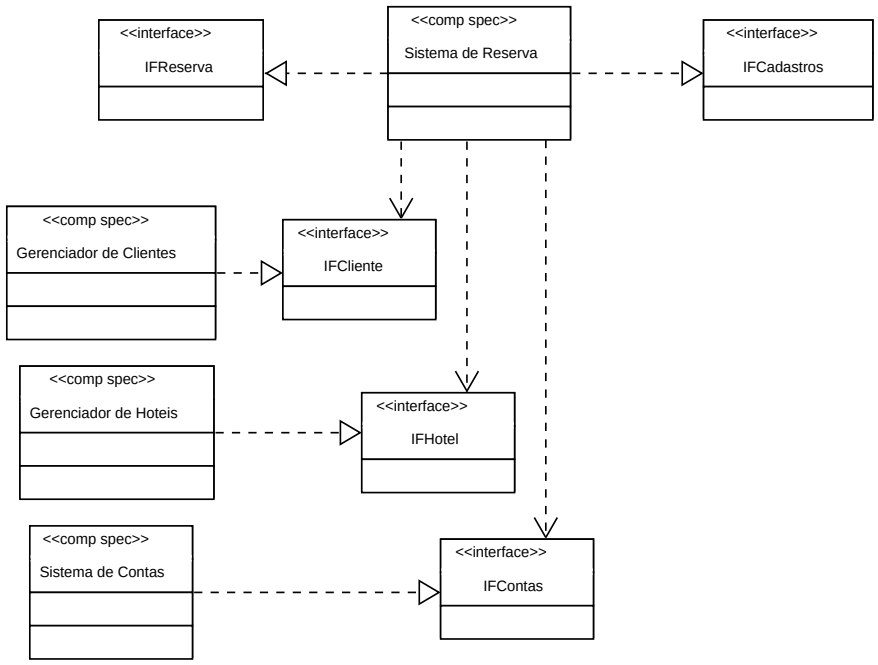

Figura 4.12: Arquitetura Inicial de Componentes para o Sistema de Reserva de Hotel

4.2.2 Identificação das Interações entre os Componentes

Produzimos até então, um conjunto inicial de interfaces e componentes. Agora precisamos decidir como estes componentes funcionarão juntos para executar uma determinada funcionalidade. Identificar as interações entre os componentes nos permitirá refinar as definições das interfaces até então identificadas, descobrir novas interfaces e operações,

identificar como as interfaces serão usadas, e ainda, refinar a arquitetura de componentes definida na seção anterior.

As interações entre os componentes são especificadas através de diagramas de colaboração UML. Um diagrama de colaboração descreve como grupos de objetos colaboram entre si para produzir um determinado comportamento [FS98]. Nos diagramas de colaboração produzidos nesta etapa do processo, cada retângulo representa uma instância de um componente que suporta uma determinada interface e os links representam chamadas a operações fornecidas pela interface que o componente suporta. A numeração nos links indica a seqüência em que as chamadas ocorrem.

Na etapa de Identificação dos Componentes (ver Seção 4.2.1), descobrimos, a partir dos casos de uso, as operações das interfaces de sistema. Usaremos os diagramas de colaboração nesta etapa, para descobrir as operações das interfaces de negócio. Deve ser construído um diagrama de colaboração para cada operação identificada para as interfaces de sistema. À medida que os diagramas de colaboração são construídos, são descobertas as operações das interfaces de negócio e suas assinaturas. As assinaturas das operações das interfaces de sistema são também descobertas durante a construção dos diagramas. Nesta etapa da modelagem, não existe preocupação sobre como as operações serão implementadas. O mais importante é identificar um conjunto inicial de operações que devem ser oferecidas pelas interfaces e suas assinaturas.

A Figura 4.13 mostra o diagrama de colaboração para a operação fazerReserva() da interface IFReserva. De acordo com o caso de uso Fazer Reserva, esta operação deve receber as informações da reserva (hotel, tipo do quarto e período), e do cliente (nome, CEP e e-mail) e realizar a reserva no hotel desejado, se for possível. O sistema deverá retornar para o usuário o código da reserva realizada.

Após todas essas definições, a assinatura adequada para a operação fazerReserva é descoberta:

IFReserva::fazerReserva(reserva:TReserva, cliente:TCliente) : Collection

Os tipos TReserva e TCliente são tipos de dados estruturados criados para conter respectivamente informações de reserva e cliente (Figuras 4.14 e 4.15).

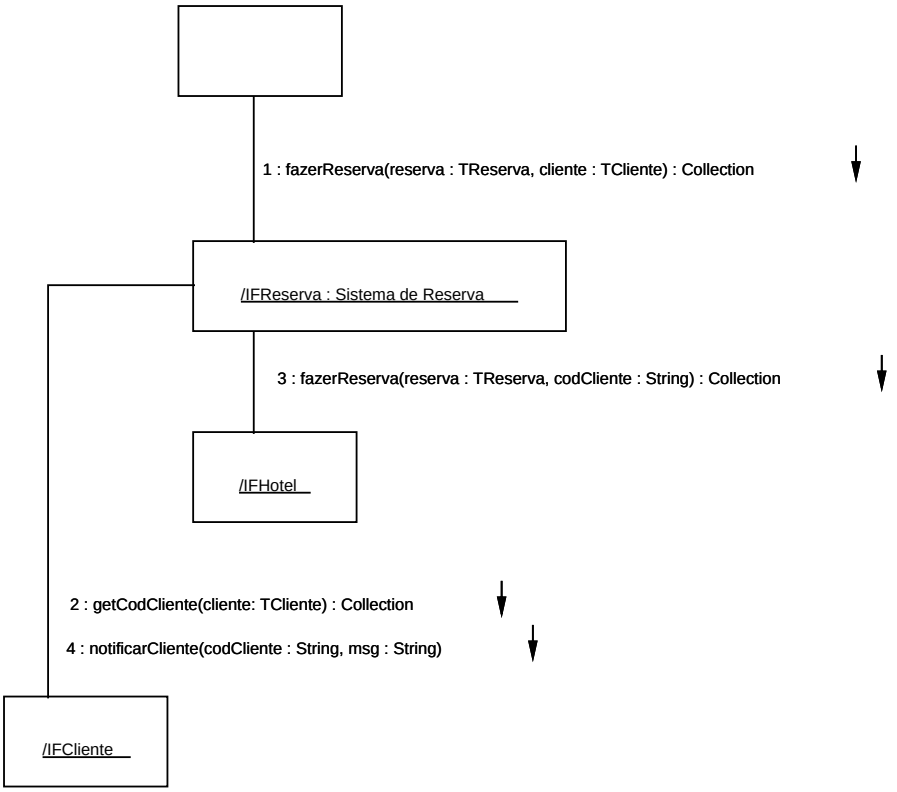

Figura 4.13: Diagrama de Colaboração da Operação Fazer Reserva da Interface IFReserva

O tipo Collection retornado por esta operação é uma interface do pacote java.util e indica que a operação retorna uma coleção de objetos, neste caso específico, retornamos um vetor de duas posições, a primeira contendo um objeto do tipo Integer representando o status de retorno da operação, e a segunda contendo um objeto do tipo String representando o código da reserva realizada.

Como mostrado no diagrama da Figura 4.13, a interface IFHotel deverá ter também a operação fazerReserva(). Isso acontece porque o componente Sistema de Reserva não conhece os dados do negócio e, portanto, não é capaz de realizar a reserva com as informações fornecidas pelo usuário. Este componente, então, repassa a solicitação para algum componente que suporta a interface IFHotel e, portanto, tem acesso aos dados do negócio.

4.2.3 Especificação dos Componentes

A principal atividade desta etapa é especificar os contratos dos componentes. Um contrato é um acordo formal entre duas ou mais partes. Ele descreve de forma não ambígua as

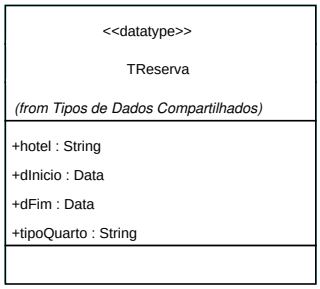

Figura 4.14: Tipo de Dados para Informações de Reserva

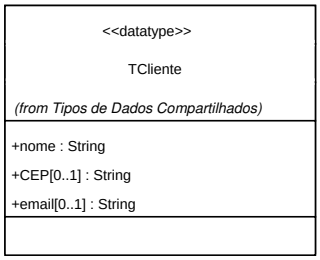

Figura 4.15: Tipo de Dado para Informações de Cliente

responsabilidades e obrigações de cada parte, além de declarar o que acontece se uma das partes descumpre o acordo.

No mundo de componentes podem ser identificados dois tipos de contratos:

- Contrato de Uso: descreve o relacionamento entre uma interface do componente e seus clientes.
- Contrato de Realização: descreve o relacionamento entre uma especificação do componente e sua implementação.

No método de teste proposto neste trabalho, estamos preocupados em fazer a verificação apenas dos contratos de uso. Os contratos de realização não são verificados.

Contrato de Uso

Um contrato de uso é definido por uma especificação de interface. A especificação de uma interface inclui os seguintes itens:

- Operações - Uma interface é um conjunto de operações, onde cada operação define um serviço ou função que o componente fornecerá ao cliente. Cada operação da interface deve ser especificada através da descrição de cada um dos seguintes aspectos da operação:
  - Nome - Nome da operação que está sendo especificada.
  - Descrição - Identificação do objetivo da operação, seguido por uma descrição informal de seus efeitos normais e excepcionais.

- Entradas - Parâmetros de entrada fornecidos por qualquer objeto que chama a operação.
- Saídas - Informações retornadas para qualquer objeto que chama a operação.
- Pré-Condições - Descrição formal ou informal das condições que devem ser satisfeitas para que o componente possa garantir o resultado esperado.
- Pós-Condições - Descrição formal ou informal do estado esperado do componente, alcançado após a execução da operação quando as suposições de execução são satisfeitas.

A metodologia Componentes UML sugere que a especificação das operações contenha apenas parâmetros de entrada e saída, pré e pós-condições. Julgamos que o nome da operação e a descrição do seu objetivo são informações importantes sobre a operação e por isso essas informações foram acrescentadas à especificação da operação.

Neste trabalho preferimos descrever pré e pós-condições de maneira precisa, utilizando OCL, a fim de aumentar as chances de automatização do método de teste aqui proposto.

É importante ressaltar que nem todas as operações precisam ser descritas por meio de todos os aspectos citados acima.

Na Tabela 4.3, é mostrada a especificação da operação fazerReserva() da interface IFHotel. As pré e pós-condições da operação são descritas em OCL.

Operação	**fazerReserva()**
Descrição	Realiza uma reserva em um dos hotéis da rede.
Entradas	reserva: TReserva - Dados da reserva (hotel, período e tipo de quarto).
	cliente: String - Código do cliente da reserva.
Saídas	Collection.
	O primeiro elemento da Collection é um Integer, representando o status de execução da operação: 0: A reserva foi feita.

continua na próxima página

Tabela 4.3: Especificação da operação fazerReserva da interface IFHotel (continua)

Operação	**fazerReserva()**
	1: A reserva não foi feita porque o tipo de quarto não é válido. 2: A reserva não foi feita porque o período não é válido. 3: A reserva não foi feita porque não há quartos disponíveis. 4: A reserva não foi feita porque o hotel não é válido.
	O segundo elemento da Collection é um String, representando o código da reserva gerado pelo sistema.
Pré-Condições	reserva.hotel é um nome de hotel válido, ... reserva.tipoQuarto é um nome de tipo de quarto válido no hotel, ... reserva.dInicio é posterior à data de hoje e anterior a reserva.dFim e ... existe ao menos um quarto disponível. Hotel->exists (h | h.nome = reserva.hotel and h.tiposQuarto.nome->includes(reserva.tipoQuarto) and (h.reservas->select(r | r.cancelada=false and r.tipoQuarto=reserva.tipoQuarto and ((r.dInicio ≤ reserva.dInicio and r.dFim > reserva.dInicio) or (r.dInicio < reserva.dFim and r.dFim ≥ reserva.dFim) or (r.dInicio ≥ reserva.dInicio and r.dFim ≤ reserva.dFim)))-> size < h.quartos->select(q | q.tipo.nome=reserva.tipoQuarto))->size and reserva.dInicio < reserva.dFim and reserva.dInicio > Today())
Pós-Condições	A reserva é criada. Hotel->exists(h | h.nome = reserva.hotel) and h.reservas->exists(r | r.codigo=codReserva and r.dFim=reserva.dFim and r.dInicio=reserva.dInicio and

continua na próxima página

Tabela 4.3: Especificação da operação fazerReserva da interface IFHotel (continua)

Operação	**fazerReserva()**
	r.tipoQuarto.nome=reserva.tipoQuarto and r.cliente=reserva.cliente and r.cancelada = false)

Tabela 4.3: Especificação da operação fazerReserva da interface IFHotel

- Modelos de Informação - Incluem os tipos de informação que são de responsabilidade exclusiva da interface. São representados através de Diagramas de Especificação de Interface. A Figura 4.16 mostra um Diagrama de Especificação de Interface para a interface IFCliente. Vale notar que neste diagrama podem ser encontradas duas classes diferentes manipulando as mesmas informações: Cliente e TCliente. Cliente é uma classe que faz parte da estrutura interna do componente, e modela portanto a informação que é interna ao componente. Os clientes do componente não tomam conhecimento da sua existência. Já TCliente é um tipo estruturado de dado que serve de modelo para a informação que é interna ao componente, sendo visível aos clientes do componente.

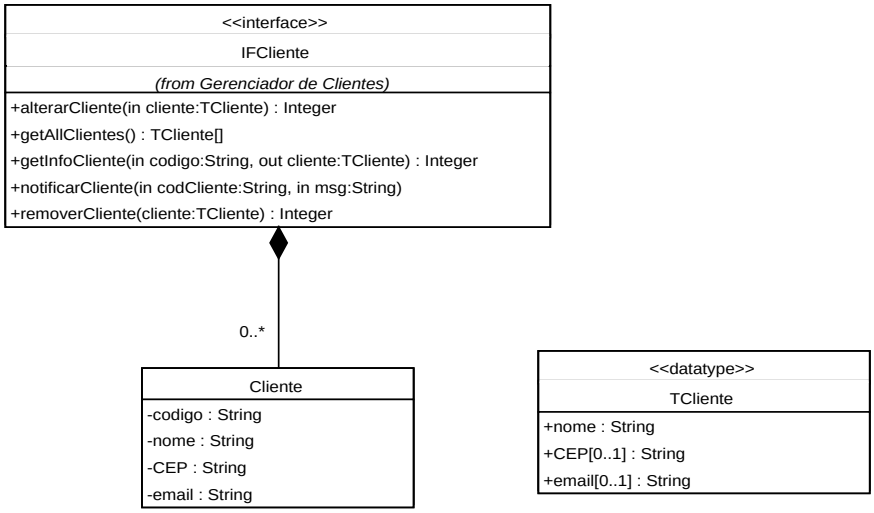

Figura 4.16: Diagrama de Especificação da Interface IFCliente

- Invariantes - São restrições associadas a um tipo de informação que se aplicam a todas as instâncias do tipo. Usamos OCL para especificar as invariantes.

Contrato de Realização

Um contrato de realização é definido por uma especificação de componente. Uma especificação de componente é constituída por um conjunto de interfaces oferecidas e usadas pelo componente e pelas interações entre o componente e outros componentes que oferecem as interfaces usadas pelo componente sendo especificado.

O conjunto de interfaces oferecidas e usadas pelo componente é definido durante a identificação dos componentes. Nesta etapa são construídos os diagramas de especificação dos componentes de sistema e de negócio e a arquitetura inicial do sistema. O diagrama de especificação dos componentes de sistema mostrado anteriormente na Figura 4.10, indica que o componente Sistema de Reserva deve oferecer duas interfaces de sistema (IFReserva e IFCadastros) e deve usar três outras interfaces (IFHotel, IFCliente e IFContas). Já o componente Gerenciador de Clientes, cujo diagrama de especificação é mostrado na Figura 4.11, deve oferecer a interface IFCliente e não usa nenhuma outra interface. Essa informação é importante para que o desenvolvedor saiba quais operações devem ser implementadas no componente e quais operações de outros componentes podem ser usadas pelo componente sendo especificado.

As interações entre os componentes são também definidas durante a identificação dos componentes e de suas operações e são representadas por meio de diagramas de colaboração. Estes diagramas foram construídos a fim de se descobrir as operações fornecidas por cada componente e para demonstrar como os componentes interagem entre si para entregar alguma funcionalidade do sistema. As informações contidas nestes diagramas, portanto, permitem que o desenvolvedor conheça as operações de outros componentes que devem ser utilizadas pelo componente sendo especificado para realizar alguma funcionalidade do sistema. A metodologia sugere que sejam construídos novos diagramas de colaboração, que são fragmentos dos diagramas construídos anteriormente, contendo apenas as interações que envolvem diretamente o componente sendo especificado, deixando de lado as interações entre outros componentes que são também necessárias para a entrega da funcionalidade, mas que não fazem parte do contexto do componente sendo especificado. Nós decidimos não construir tais diagramas por eles representarem um ponto de redundância no projeto, o que poderia causar inconsistências na documentação. Acreditamos que eles sejam dispensáveis, já que toda informação necessária para a implementação do componente pode ser encontrada

nos diagramas previamente construídos.

A fim de facilitar a substituição do componente dentro da aplicação, qualquer implementação do componente deve respeitar as interações definidas.

A metodologia não propõe, mas sentimos a necessidade de construir diagramas de interação entre as classes que fazem parte do componente. Para cada caso de uso, construímos diagramas de seqüência que representam o funcionamento interno do componente. O objetivo da construção destes diagramas é descobrir as operações que devem ser oferecidas por cada classe. Os diagramas servirão ainda como insumo para a atividade de teste do componente. Seguindo a proposta do método de teste apresentado no Capítulo 5, construímos um diagrama de seqüência para cada cenário de uso das funcionalidades do componente. Dessa forma, as Figuras 4.17, 4.18, 4.19, 4.20 e 4.21 mostram diagramas de seqüência para o caso de uso Fazer Reserva, em diferentes cenários de uso. Esses diagramas representam a seqüência de troca de mensagens entre os objetos do componente Gerenciador de Hoteis para entregar a funcionalidade Fazer Reserva.

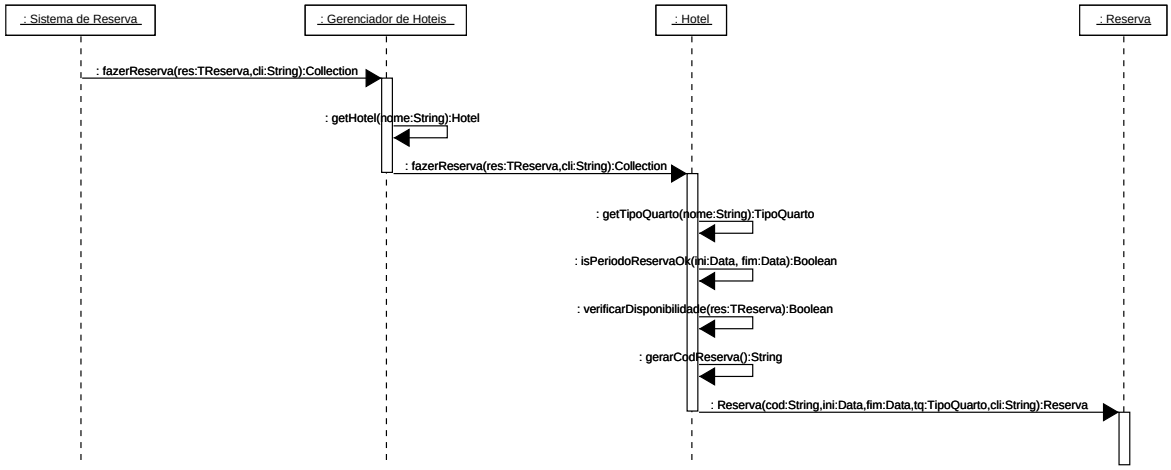

Figura 4.17: Diagrama de seqüência para o Caso de Uso Fazer Reserva (Cenário de Sucesso)

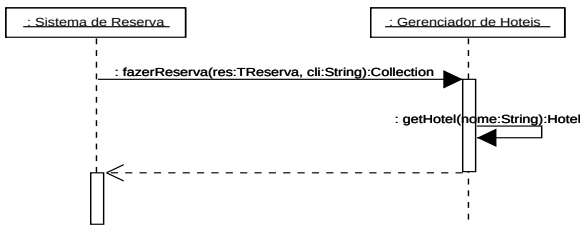

Figura 4.18: Diagrama de seqüência para o Caso de Uso Fazer Reserva (Hotel Inválido)

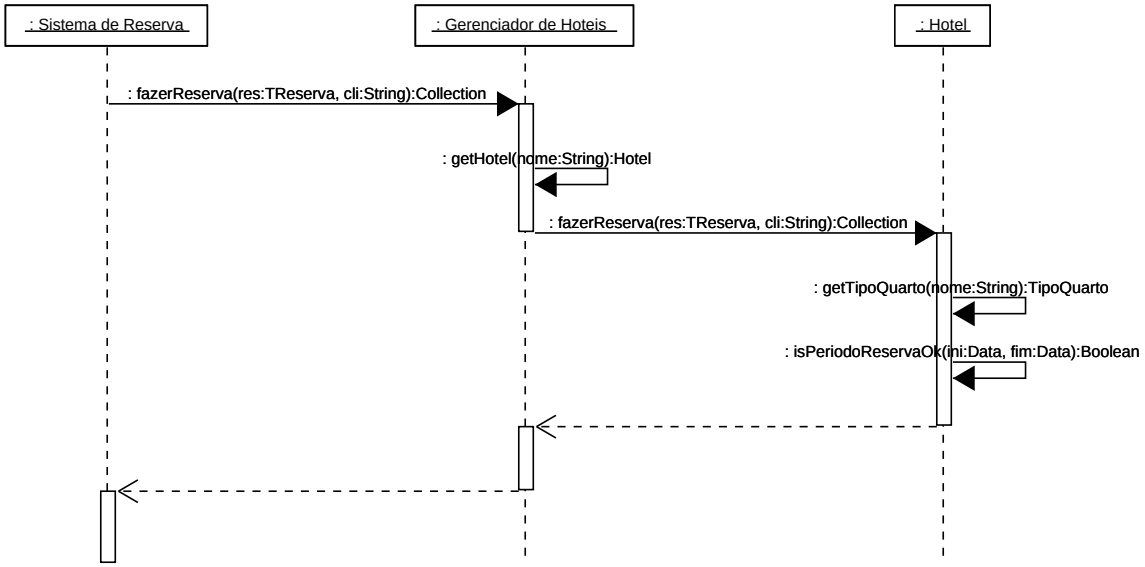

Figura 4.19: Diagrama de seqüência para o Caso de Uso Fazer Reserva (Período Inválido)

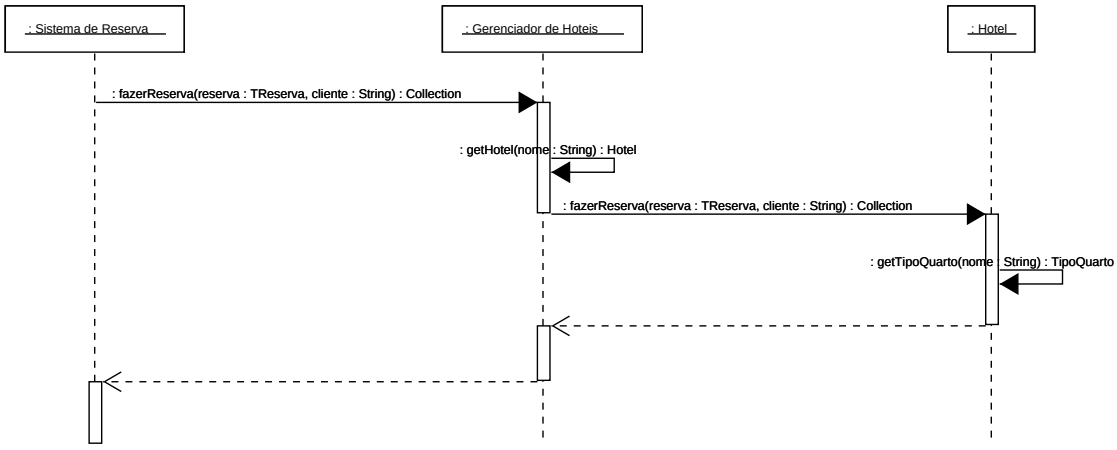

Figura 4.20: Diagrama de seqüência para o Caso de Uso Fazer Reserva (Tipo de Quarto Inválido)

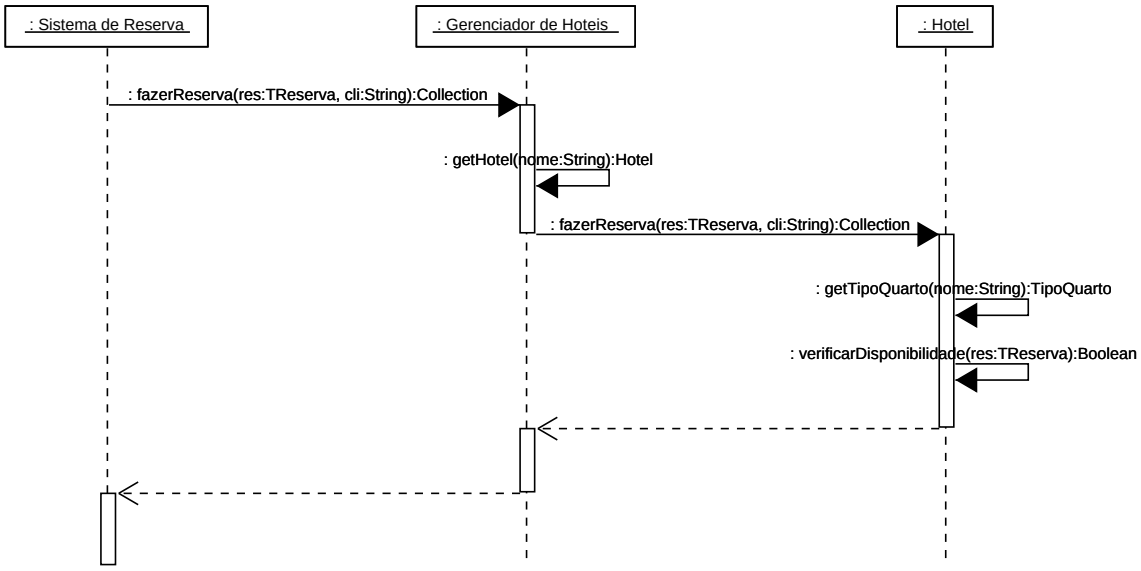

Figura 4.21: Diagrama de seqüência para o Caso de Uso Fazer Reserva (Quarto não Disponível)

Ao final da etapa de especificação dos componentes, uma descrição completa das operações dos componentes estará disponível, tornando possível desenvolver os casos de teste funcional. Desenvolver os casos de teste funcional neste momento não melhora apenas a qualidade dos testes gerados, melhora também a qualidade dos artefatos desenvolvidos.

4.3 Materialização de Componentes

O propósito da etapa de materialização de componentes é fornecer implementações dos componentes especificados até agora, seja implementando diretamente a especificação, seja adquirindo um componente existente que satisfaça a especificação.

Nesta etapa é preciso se preocupar com o ambiente em que o componente será executado, pois o ambiente estabelece um conjunto de regras que devem ser obedecidas pelo componente e um conjunto de serviços de infra-estrutura do qual o componente pode depender, por exemplo, suporte a transação, segurança, concorrência, dentre outros. Os dois principais ambientes de componentes disponíveis no mercado atualmente são Microsoft COM+ e Enterprise JavaBeans (EJB).

A especificação que fizemos dos componentes do Sistema de Reserva de Hotel é independente da tecnologia usada para implementar e executar estes componentes. Decidimos usar a

tecnologia Enterprise JavaBeans por se tratar de uma tecnologia independente de plataforma e de fornecedor, diferentemente do Microsoft COM+, cuja implementação é fornecida pela Microsoft e está limitada a uma única plataforma, Windows 2000 [HC01].

Ao final desta etapa, os componentes implementados devem ser testados para garantir sua funcionalidade, independente do contexto da aplicação. O cuidado de testar tais componentes neste momento favorece seu reuso em outras aplicações. Já os componentes adquiridos podem ser testados posteriormente, após a montagem da aplicação, já que a preocupação principal com relação a estes componentes é que eles se comportem de forma adequada no contexto da aplicação.

4.4 Montagem da Aplicação

A etapa final do processo de desenvolvimento de software baseado em componentes é a etapa de montagem. Nesta etapa, os componentes previamente implementados ou adquiridos e testados individualmente, são reunidos em um sistema e uma interface de usuário para este sistema é projetada de forma a se obter uma aplicação. A arquitetura de componente é essencial na definição da estrutura geral da aplicação.

Ao final da etapa de montagem, aplicação passa para a etapa de testes onde são realizados os testes de aceitação do usuário. Após a realização dos testes a aplicação estará pronta para ser entregue para seus usuários.

Capítulo 5

O Método de Teste

Em geral, a atividade de teste é vista como a última atividade que deve ser feita no processo de desenvolvimento de software. Desse ponto de vista, é necessário que a implementação do software esteja completa para que a atividade se inicie.

Neste trabalho, consideramos que a atividade de teste deve ser aplicada em vários pontos do desenvolvimento e não apenas no final do processo. Mais que isso, consideramos que os testes não devem se aplicar apenas ao código do programa, mas a todos os artefatos produzidos durante o desenvolvimento do software. Aplicar o teste dessa forma aumenta as chances de se desenvolver um sistema que atenda às expectativas do cliente.

Este capítulo descreve o método de teste proposto neste trabalho, especificando em detalhes as etapas sugeridas pelo método e as atividades executadas em cada etapa a fim de gerar os casos de teste, selecionar os dados e gerar os oráculos de teste. Todo o método é ainda exemplificado com a aplicação do estudo de caso.

5.1 Descrição do Método

O objetivo principal do método aqui proposto é permitir que propriedades individuais de componentes de software possam ser verificadas.

Um processo de teste completo pode envolver as seguintes etapas:

- Planejamento: Nesta etapa, são definidos que tipos de teste serão realizados e quais as expectativas com relação aos testes realizados.

- Especificação: Nesta etapa, são gerados os modelos de teste dos quais são derivados os casos de teste, dados e oráculos.

- Construção: Nesta etapa, os artefatos necessários para execução do teste são criados. Os casos de teste e os oráculos identificados na etapa anterior são implementados utilizando-se alguma linguagem de programação.

- Execução: Nesta etapa, são executados os casos de teste desenvolvidos para o sistema na etapa anterior, sendo fornecidos os dados selecionados na etapa de especificação.

- Análise dos Resultados: Nesta etapa, os oráculos gerados na etapa de construção são utilizados para analisar se o programa em teste falhou ou não no teste executado.

Além das etapas acima, em se tratando de teste de componentes, uma sexta etapa pode ser realizada. Nesta etapa os artefatos e resultados dos testes obtidos nas etapas anteriores são empacotados juntamente com o componente desenvolvido a fim de fornecer ao usuário do componente facilidades para conhecer os testes executados e desenvolver novos testes, caso seja necessário [Mac00].

O método aqui proposto fornece um conjunto de diretrizes e técnicas que auxiliam a execução e o acompanhamento de cada uma dessas etapas. Já que a utilização efetiva de um método de teste depende intimamente de uma metodologia de desenvolvimento, adotamos a metodologia descrita no capítulo anterior e fizemos a integração das atividades de teste a esta metodologia. A integração entre as etapas de teste e o processo de desenvolvimento apresentado no capítulo anterior é mostrada na Figura 5.1.

O processo de desenvolvimento se dá em quatro etapas não seqüenciais:

- Definição dos requisitos.

- Modelagem dos componentes, englobando identificação dos componentes, identificação das interações entre os componentes e especificação dos componentes

- Materialização dos componentes, que pode se dar tanto através da implementação dos componentes como através da aquisição de um componente já desenvolvido previamente.

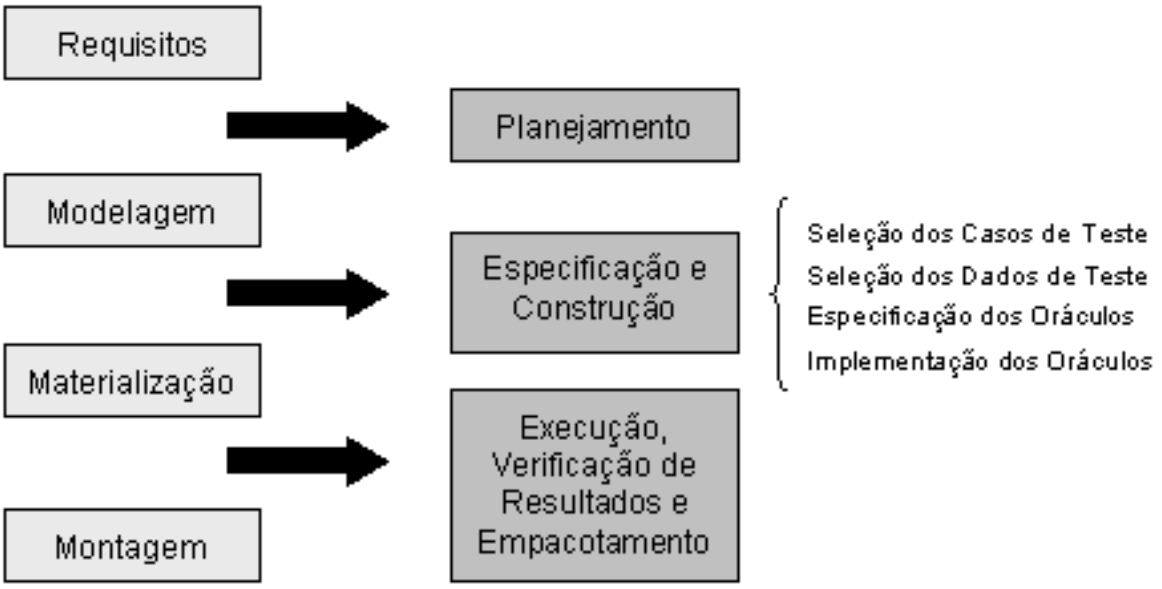

Figura 5.1: Integração das etapas de teste no processo de desenvolvimento

- Montagem da aplicação, que é a integração dos componentes de acordo com uma arquitetura específica.

As atividades de teste foram inseridas no processo da seguinte forma:

- O planejamento dos testes pode ser feito durante a definição dos requisitos. Esta atividade tem por objetivo definir que tipos de teste serão realizados e o que se espera da realização destes testes.

- A especificação dos testes deve ser feita à medida que a modelagem é realizada. Em linhas gerais, durante a etapa de modelagem estaremos selecionando os casos de teste, usando a técnica TOTEM (Testing Object-orienTed systEms with the unified Modeling language) [BL01], combinada com o aspecto estatístico do Cleanroom [PTLP99]. Os dados são também selecionados durante esta etapa. Para esta atividade usamos as orientações de teste de domínio propostas em [Bei95] e de teste por partições proposta em [Som96; Bei90]. Por fim, os oráculos são gerados baseando-se na técnica TOTEM, a partir das especificações de pré e pós-condições feitas em OCL.

- A construção dos testes pode ser realizada em paralelo à sua especificação ou ao final da especificação.

- A execução dos testes e análise dos seus resultados é feita após a materialização dos componentes. Nesta etapa, devemos também empacotar os artefatos de teste juntamente com o componente produzido.

Além dos aspectos citados acima, consideramos que os artefatos produzidos durante a modelagem tenham passado por sessões de inspeções que avaliaram a corretude, completude e consistência desses artefatos. Para se ter testes efetivos, é importante que os artefatos sejam de boa qualidade, visto que retratam a funcionalidade desejada.

Nas subseções seguintes detalhamos cada uma das etapas do processo de teste e suas atividades.

5.1.1 Planejamento dos Testes

O planejamento dos testes pode ser iniciado logo que os requisitos tenham sido definidos. Nesse momento, os artefatos de análise (modelo conceitual e diagrama de casos de uso) estão sendo elaborados e são fontes de informações importantes para o plano de teste. Desenvolver o plano de teste neste momento permite que se tenha idéia da dimensão da tarefa de teste logo no início do projeto, permitindo que a distribuição de recursos durante o processo de desenvolvimento do software seja feita de forma mais consciente e racional.

No planejamento são tomadas as seguintes decisões [MS01]:

- Quem executará os testes? Os próprios desenvolvedores do componente podem executar todos os testes ou equipes especializadas em testes podem ser utilizadas. Existem ainda soluções de meio-termo, como por exemplo, usar um testador para especificar os testes e a equipe de desenvolvimento para implementar e executar os testes. Outra solução é utilizar a técnica "buddy", onde o teste é feito por pares de desenvolvedores que trocam seus produtos e um testa o produto do outro. No estudo de caso desenvolvido neste trabalho, a limitação de recursos nos levou a adotar a opção em que o próprio desenvolvedor do componente testou seu próprio componente, entretanto, da nossa prática de engenharia de software, acreditamos que ter uma equipe especializada em testes pode resultar em um produto de maior qualidade, visto que, testadores são especificamente treinados para realizar a atividade de teste e não estão famil-

iarizados com o código produzido, o que pode aumentar as chances de se desenvolver testes mais prováveis de revelarem erros.

- Que partes do componente serão testadas? Temos aqui 3 opções: não testar, testar todas as funcionalidades ou testar uma amostra das funcionalidades. No estudo de caso, a primeira opção foi descartada por não atender os objetivos do método. A segunda e última opções representam possibilidades viáveis. Em nosso estudo de caso foi possível optar por testar todas as funcionalidades, já que tínhamos um conjunto pequeno de funcionalidades, sendo possível testá-lo com os recursos que tínhamos disponíveis. Entretanto, essa nem sempre é a realidade, então testar uma amostra pode ser uma alternativa. Nesse caso é necessário aplicar uma técnica sistemática que nos ajude a selecionar um subconjunto do conjunto total das funcionalidades. Propomos usar a técnica de análise de riscos que ajuda a definir as funcionalidades que devem receber maior atenção. Embora no nosso exemplo não tenhamos descartado nenhuma funcionalidade, se isso tivesse sido necessário, a técnica nos teria ajudado a decidir quais funcionalidades teriam ficado de fora da bateria de testes. A aplicação da técnica de análise de riscos é descrita na Subseção **Conduzindo o Planejamento**.

- Quando os testes serão executados? Mais uma vez temos 3 opções: testar durante todo o processo de desenvolvimento, testar à medida que os componentes são desenvolvidos, ou seja, estabelecer milestones específicos para teste, ou testar tudo no final do processo. Decidimos testar à medida que os artefatos eram desenvolvidos a fim de aumentar as chances de se produzir uma especificação mais completa, correta e consistente dos componentes e com isso produzir componentes mais prováveis de atender as necessidades do cliente. Testar durante o processo também nos daria este resultado, mas acreditamos que estabelecer milestones específicos para a atividade de teste permite que a atividade de teste seja melhor gerenciada. O importante é que os testes sejam desenvolvidos o mais cedo possível no processo de desenvolvimento.

- Como os testes serão realizados? Serão realizados testes funcionais, estruturais ou ambos? Serão desenvolvidos apenas testes individuais dos componentes? Será feito também teste de integração? Serão realizados testes das classes que fazem parte do componentes? Para este método, estamos considerando apenas a realização de teste

funcional dos componentes individuais, mas concordamos que os testes estruturais e o teste de integração são também de grande importância para a qualidade final do software e sugerimos que trabalhos futuros considerem a possibilidade de incorporar tais aspectos a este método aqui proposto. Fizemos também o teste de algumas classes do componente e acreditamos que seja de grande importância realizar o teste de todas as classes individualmente antes de realizar o teste do componente propriamente dito.

- Testamos o suficiente? Essa decisão está relacionada à cobertura dos testes, ou seja, o quanto os testes exercitarão cada funcionalidade do software. Mais uma vez, podemos não testar a funcionalidade, testá-la de forma exaustiva ou testá-la parcialmente. Teste exaustivo é quase sempre inviabilizado pelas limitações de recursos, então fizemos a opção de fazer teste parcial das funcionalidades. A seleção dos cenários que serão usados no desenvolvimento dos casos de teste é feita utilizando-se a técnica TOTEM em conjunto com a técnica de teste usada no Cleanroom. A análise dos riscos também auxilia nesta seleção. A forma como os casos de teste são selecionados neste método é descrita na Seção 5.1.2.

Ao tomar todas essas decisões, estamos determinando os recursos necessários, os métodos utilizados e a qualidade dos resultados do esforço de teste.

É importante ressaltar que não existe uma decisão melhor nem pior do que outra, entretanto certas decisões são mais apropriadas em certas situações. É preciso, portanto, conhecer as opções e escolher aquela que parecer ser mais adequada ao problema em questão.

Conduzindo o Planejamento

Além dos artefatos produzidos na análise, o planejamento dos testes deve contar ainda com a análise de riscos para determinar o que será testado e o quanto. O princípio geral do teste baseado em risco é testar mais as partes do projeto que apresentam os maiores graus de risco à conclusão do projeto. Decidimos adotar neste método o planejamento baseado em risco, discutido em [MS01].

O objetivo principal da análise de risco é identificar o risco que cada caso de uso oferece à conclusão do projeto. A análise de risco envolve três tarefas: identificar os riscos que cada caso de uso oferece ao desenvolvimento do software, quantificar o risco e produzir uma lista

dos casos de uso, ordenada pelo grau de risco.

A quantificação dos riscos pode variar de um projeto para outro. Ela deve ter níveis suficientes para separar os casos de uso em grupos de tamanho razoável. No Sistema de Reservas, consideramos 3 graus de risco: baixo, médio e alto. A idéia é que os casos de uso que se encaixam no grau mais alto de risco recebam uma atenção especial.

A Tabela 5.1 apresenta a análise de riscos para os casos de uso relacionados ao componente Gerenciador de Hóteis do Sistema de Reservas. O grau de risco associado a cada caso de uso é resultante da análise da freqüência de ocorrência do caso de uso e do quanto o funcionamento do caso de uso é crítico para a satisfação do usuário. A estratégia que usamos para encontrar o grau de risco do caso de uso a partir da freqüência e da criticalidade foi selecionar o mais alto valor dos dois atributos. Dessa forma, o caso de uso Alterar Reserva que tem baixa freqüência e alta criticalidade apresenta grau de risco alto, já o caso de uso Remover Quarto que ocorre com freqüência baixa e tem média criticalidade apresenta grau de risco médio.

Caso de Uso	**Grau de Risco**	**Freqüência**	**Criticalidade**
Fazer Reserva	Alto	Alta	Alta
Requerer Reserva	Alto	Alta	Alta
Alterar Reserva	Alto	Baixa	Alta
Remover Quarto	Médio	Baixa	Média
Remover Tipo Quarto	Médio	Baixa	Média
Cancelar Reserva	Baixo	Baixa	Baixa
Processar No-Show	Baixo	Baixa	Baixa
Incluir Quarto	Baixo	Baixa	Baixa
Alterar Quarto	Baixo	Baixa	Baixa
Incluir Tipo Quarto	Baixo	Baixa	Baixa
Alterar Tipo Quarto	Baixo	Baixa	Baixa
Remover Reserva	Baixo	Baixa	Baixa

Tabela 5.1: Análise dos Riscos dos Casos de Uso relacionados ao Componente Gerenciador de Hóteis do Sistema de Reserva

De acordo com a Tabela 5.1, os casos de uso Fazer Reserva, Requerer Reserva e Alterar Reserva devem receber atenção mais especial do que os casos de uso Remover Quarto e Remover Tipo Quarto, que por sua vez, devem receber mais atenção do que os demais casos de uso.

Ao final dessa etapa o plano de teste do componente deve estar concluído, indicando quantos casos de teste deverão ser desenvolvidos para cada caso de uso, quem desenvolverá e executará cada caso de teste e quando isso será feito.

Em nosso estudo de caso, considerando nossas restrições de tempo e pessoal, assumimos que tínhamos recursos disponíveis para desenvolver apenas 20 [1] casos de teste para o componente Gerenciador de Hotéis. Esses casos de teste foram distribuídos entre os casos de uso considerando a análise de riscos feita anteriormente. Para os casos de uso que apresentam maior grau de risco, foram desenvolvidos mais casos de teste. Essa distribuição é mostrada na Tabela 5.2.

Caso de Uso	**Grau de Risco**	**Número de Casos de Teste**
Fazer Reserva	Alto	3
Requerer Reserva	Alto	3
Alterar Reserva	Alto	3
Remover Quarto	Médio	2
Remover Tipo Quarto	Médio	2
Cancelar Reserva	Baixo	1
Processar No-Show	Baixo	1
Incluir Quarto	Baixo	1
Alterar Quarto	Baixo	1
Incluir Tipo Quarto	Baixo	1
Alterar Tipo Quarto	Baixo	1
Remover Reserva	Baixo	1

Tabela 5.2: Quantidade de Casos de Teste para cada Caso de Uso do Componente Gerenciador de Hóteis do Sistema de Reservas

[1] É possível estimar este número a partir de dados históricos.

5.1.2 Especificação dos Testes

A especificação dos testes pode ser iniciada logo que a especificação do componente tenha sido iniciada. A especificação do componente contém informações importantes sobre a solução adotada para o problema e é usada para derivar os casos de teste, dados e oráculos.

Selecionando os Casos de Teste

Durante a modelagem dos componentes, foram desenvolvidos diagramas de seqüência para cada caso de uso do componente. A construção desses diagramas tinha como objetivo principal descobrir os métodos que cada classe que compõe o componente deveria fornecer para que a funcionalidade proposta no caso de uso pudesse ser entregue.

Para a atividade de teste, os diagramas de seqüência têm ainda uma outra finalidade: orientar a seleção dos casos de teste.

Como foi dito na Seção 5.1.1, o teste exaustivo do componente é na maioria das vezes inviável. Já que os diagramas de seqüência fornecem uma visão dos diversos cenários de uso do componente, resolvemos utilizar neste método, os diagramas de seqüência para decidir quais cenários serão testados. A análise de riscos feita durante o planejamento nos diz quantos cenários de cada funcionalidade do componente serão testados, mas não nos diz quais são esses cenários. Usamos então a combinação de duas técnicas de teste existentes, TOTEM e Cleanroom, para selecionar os cenários de uso do componente que serão testados.

A técnica TOTEM expressa os diagramas de seqüência na forma de expressões regulares que são uma forma mais compacta e analisável dos diagramas. O alfabeto dessas expressões são os métodos públicos dos objetos presentes nos diagramas. As expressões são então constituídas de termos que apresentam o formato $Operacao_{Classe}$, denotando qual é a operação que está sendo executada e a que classe esta operação pertence.

A idéia da técnica TOTEM é gerar uma única expressão regular, a partir de um diagrama de seqüência que representa os cenários principal e alternativos do caso de uso, da qual será possível extrair automaticamente todos os possíveis cenários. A seleção dos cenários neste caso pode ser feita de forma aleatória ou o próprio usuário poderá selecionar um cenário específico.

Embora a técnica TOTEM tenha se mostrado muito interessante na seleção dos casos de

teste, percebemos que a construção de diagramas de seqüência que englobam tanto o cenário principal quanto os cenários alternativos não é uma prática comum, nem tão pouco é uma atividade trivial. Propomos então adaptar a técnica para gerar a expressão regular a partir de vários diagramas de seqüência. Neste caso, cada diagrama deverá representar um cenário de uso diferente, e a expressão regular gerada no final representará então todos os possíveis cenários de uso extraídos dos diagramas de seqüência.

As Figuras 5.2, 5.3, 5.4, 5.5 e 5.6 mostram os diagramas de seqüência para o caso de uso Fazer Reserva. A Figura 5.2 representa o cenário principal, onde a reserva é realizada com sucesso. A Figura 5.3 representa o cenário onde a reserva não pode ser realizada porque o hotel desejado não pertence à rede de hotéis. A Figura 5.4 representa o cenário onde a reserva não pode ser realizada porque o tipo de quarto desejado não existe no hotel desejado. A Figura 5.5 mostra o cenário onde a reserva não pode ser realizada porque o período não é válido. Por fim, a Figura 5.6 representa a situação em que a reserva não pode ser realizada porque não existe quarto disponível do tipo desejado no hotel e período desejados.

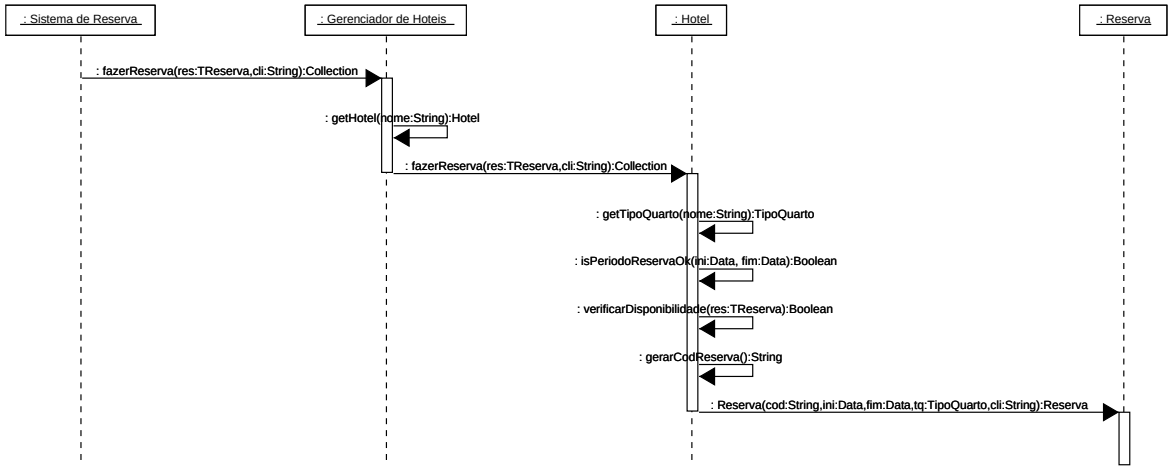

Figura 5.2: Diagrama de seqüência para o Caso de Uso Fazer Reserva - Cenário Principal: A reserva é criada com sucesso

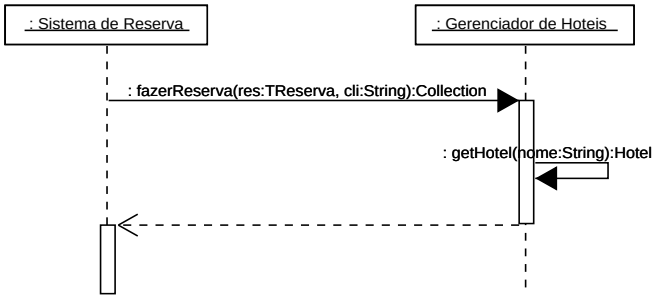

Figura 5.3: Diagrama de seqüência para o Caso de Uso Fazer Reserva - Cenário Alternativo: A reserva não é criada porque o hotel desejado não faz parte da rede de hotéis

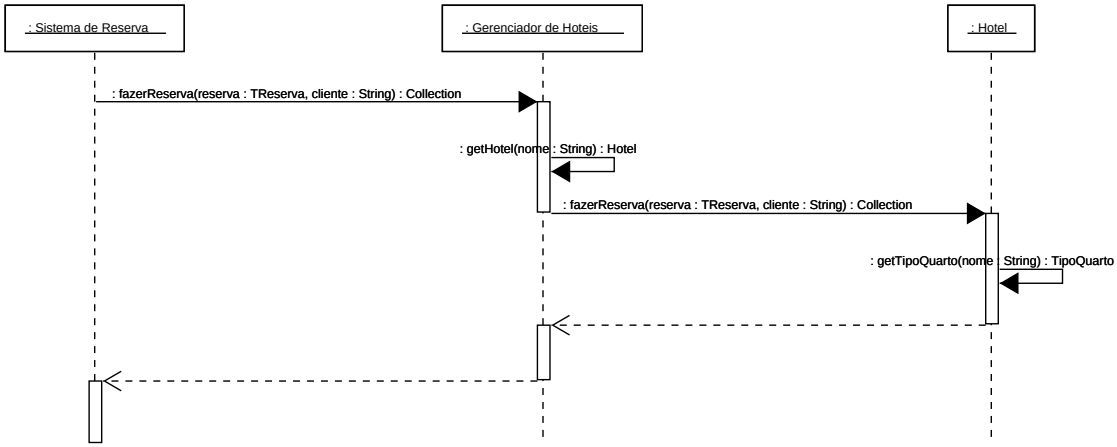

Figura 5.4: Diagrama de seqüência para o Caso de Uso Fazer Reserva - Cenário Alternativo: A reserva não é criada porque o tipo de quarto desejado não existe no hotel desejado

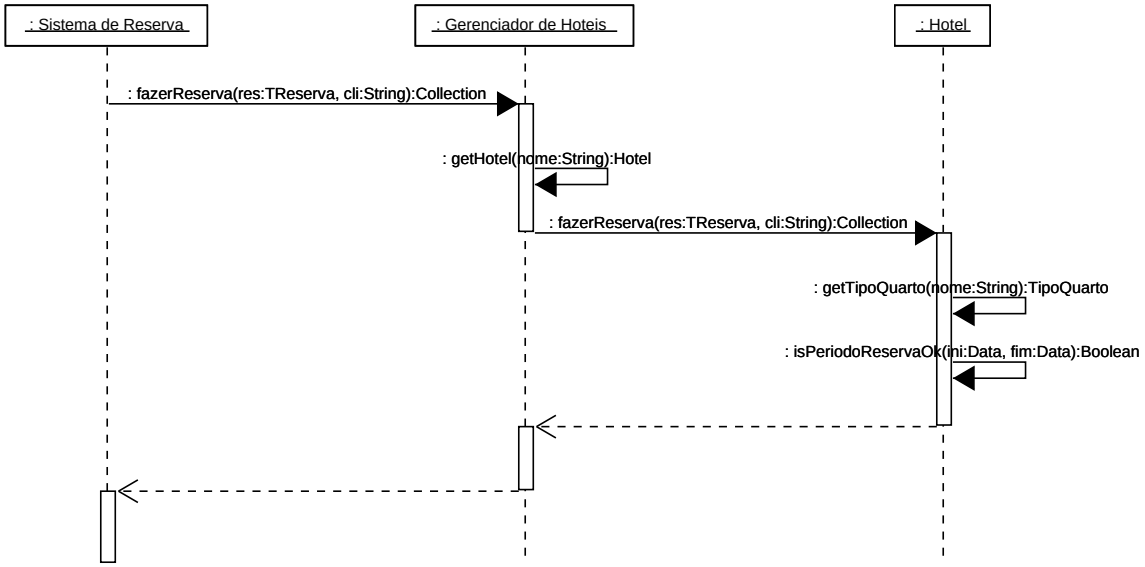

Figura 5.5: Diagrama de seqüência para o Caso de Uso Fazer Reserva - Cenário Alternativo: A reserva não é criada porque o período não é válido

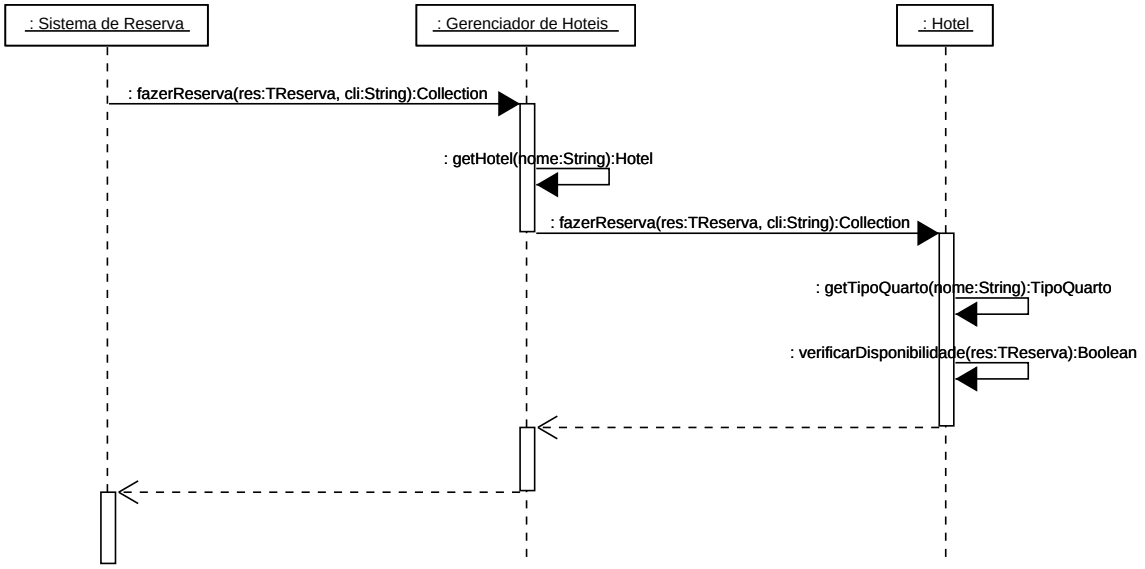

Figura 5.6: Diagrama de seqüência para o Caso de Uso Fazer Reserva - Cenário Alternativo: A reserva não é criada porque não existe quarto disponível do tipo desejado no hotel e período desejados

A expressão regular obtida a partir dos diagramas é a seguinte:

$$
\begin{aligned}
&fazerReserva_{GerenciadordeHoteis}.getHotel_{GerenciadordeHoteis} \\
&+ \\
&fazerReserva_{GerenciadordeHoteis}.getHotel_{GerenciadordeHoteis}.fazerReserva_{Hotel}. \\
&getTipoQuarto_{Hotel} \\
&+ \\
&fazerReserva_{GerenciadordeHoteis}.getHotel_{GerenciadordeHoteis}.fazerReserva_{Hotel}. \\
&getTipoQuarto_{Hotel}.isPeriodoReservaOk_{Hotel} \\
&+ \\
&fazerReserva_{GerenciadordeHoteis}.getHotel_{GerenciadordeHoteis}.fazerReserva_{Hotel}. \\
&getTipoQuarto_{Hotel}.isPeriodoReservaOk_{Hotel}. \\
&verificarDisponibilidade_{Hotel} \\
&+ \\
&fazerReserva_{GerenciadordeHoteis}.getHotel_{GerenciadordeHoteis}.fazerReserva_{Hotel}. \\
&getTipoQuarto_{Hotel}.isPeriodoReservaOk_{Hotel}.\ verificarDisponibilidade_{Hotel}. \\
&gerarCodReserva_{Hotel}.Reserva_{Reserva}
\end{aligned}
$$

Esta expressão é uma soma de produtos onde cada termo representa um cenário de uso do componente. O termo 1 representa o caso onde a reserva não é criada porque o hotel não pertence à rede de hotéis, o termo 2 representa o caso onde a reserva não é criada porque o tipo de quarto desejado não existe no hotel desejado, o termo 3 representa o caso onde a reserva não pode ser criada por o período não é válido, o termo 4 representa o caso onde a reserva não é criada porque não existe quarto disponível do tipo desejado no período e hotel desejados e o termo 5 representa o caso onde a reserva é criada com sucesso. É importante notar que apenas as mensagens representando chamadas a operações no diagrama de seqüência são contempladas na expressão regular. Chamadas contendo retornos ou condições de guarda não foram consideradas na geração da expressão regular e podem ser melhor analisadas em trabalhos futuros.

A fim de melhorar os critérios de seleção dos cenários, e possibilitar uma seleção mais automática e direcionada dos cenários de uso, incorporamos à técnica TOTEM alguns as-

pectos interessantes utilizados na técnica de teste usada no Cleanroom, especialmente os aspectos estatísticos desta técnica.

A técnica de teste usada no Cleanroom propõe a construção de um modelo de uso do sistema que representa todos os possíveis usos do sistema e suas probabilidades de ocorrência. Este modelo é expresso normalmente por meio de um grafo direcionado, onde um conjunto de estados são conectados através de arcos de transição. Cada arco representa um estímulo para o sistema, que o faz mudar de estado, e possui um valor de probabilidade associado. Os casos de teste são gerados percorrendo-se o modelo, partindo-se do seu estado inicial até o estado final. A seqüência de estímulos que leva o sistema do seu estado inicial ao estado final, através de um determinado caminho no modelo, é definida baseando-se nas probabilidades das transições. A Figura 5.7 mostra um simples exemplo de modelo de uso produzido utilizando-se a técnica de teste usada no Cleanroom.

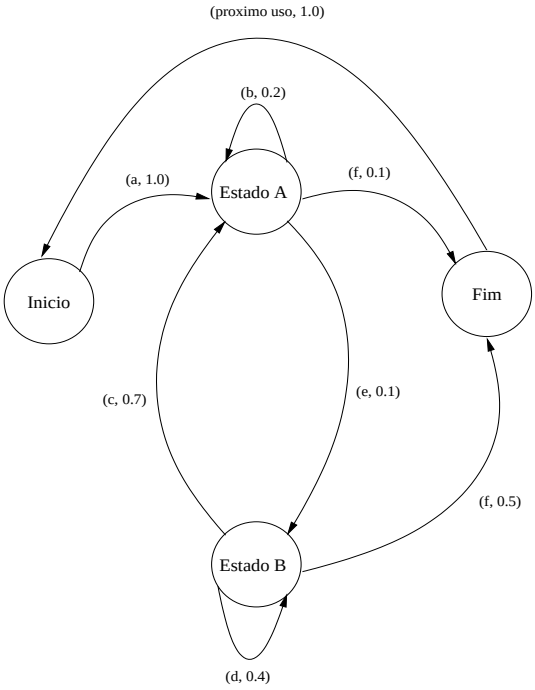

Figura 5.7: Um Exemplo de Modelo de Uso do Cleanroom

Ao combinar as duas técnicas, derivamos um modelo de uso da expressão regular obtida a partir dos diagramas de seqüência. Neste modelo de uso são inseridos dois vértices representando o início e o fim da seqüência de troca de mensagens entre os objetos. Cada troca de mensagem dá origem também a um novo vértice. As transições são rotuladas com uma mensagem, que é uma chamada a uma operação da classe, no formato Classe.Operação. As

transições devem conter ainda uma probabilidade de ocorrência variando de 0 a 1. O fim da seqüência de troca de mensagens dá origem a uma transição ligando o vértice da última chamada de operação ao vértice que representa o fim da seqüência.

É importante ressaltar que todo este processo de geração de expressões regulares e conversão para modelo de uso deve ser automatizado, a fim de aumentar as chances de aplicação prática deste método.

Para exemplificar a conversão da expressão regular em um modelo de uso, usamos a expressão regular gerada para o caso de uso Fazer Reserva, mostrada anteriormente. Por essa expressão percebemos que existem 5 cenários possíveis:

- Fazer reserva em um hotel que não pertence à cadeia.
- Fazer reserva em um hotel que pertence à cadeia, mas que não possui quarto do tipo desejado pelo cliente.
- Fazer reserva em um hotel que pertence à cadeia, possui quarto do tipo desejado pelo cliente, mas o período informado não é válido.
- Fazer reserva em um hotel que pertence à cadeia, que possui quarto do tipo desejado pelo cliente, cujo período é válido, mas nenhum dos quartos do tipo desejado está disponível no período.
- Fazer reserva em um hotel que pertence à cadeia, que possui quarto do tipo desejado pelo cliente e ao menos um quarto do tipo desejado está disponível no período.

O modelo de uso produzido para esta expressão é mostrado na Figura 5.8.

Neste modelo de uso, o vértice 0 representa o início da seqüência de troca de mensagens e o vértice 1 representa o final dessa seqüência. Cada cenário é representado por um caminho que pode ser percorrido no grafo, iniciando-se no vértice 0 e terminando no vértice 1. O primeiro cenário, representando a situação onde se tenta fazer reserva em um hotel que não pertence à cadeia é representado pelo caminho 0231. O segundo cenário que representa a tentativa de se fazer reserva em um hotel pertencente à cadeia, mas que não possui quarto do tipo desejado pelo cliente, é representado no grafo pelo caminho 023451. O terceiro cenário, representando a situação em que se tenta fazer reserva em um hotel que pertence

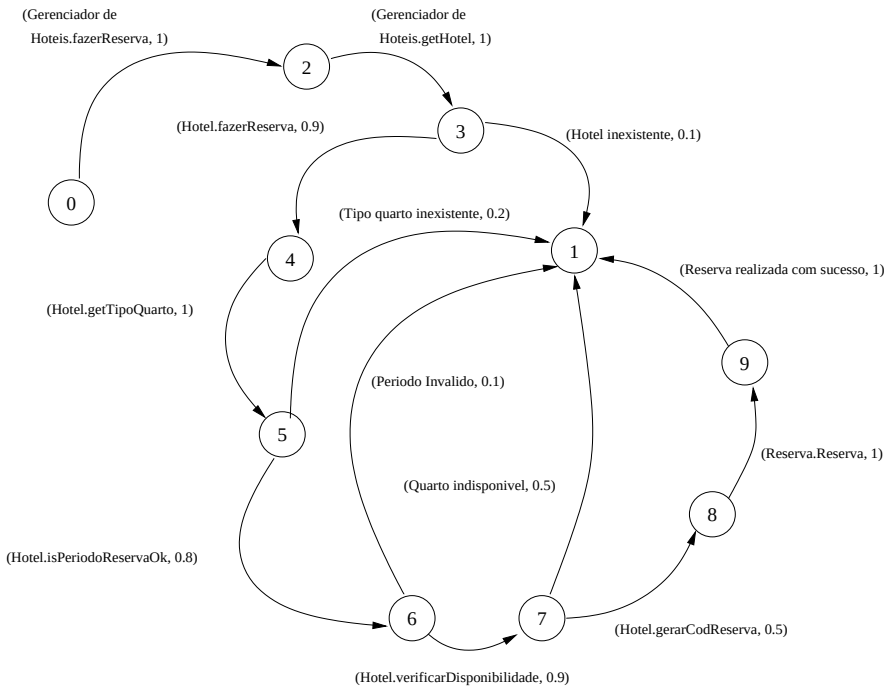

Figura 5.8: Modelo de Uso para o Caso de Uso Fazer Reserva

à cadeia, possuindo quarto do tipo desejado, mas num período inválido, é representado no grafo pelo caminho 0234561. O cenário onde se tenta fazer uma reserva em um hotel pertencente à rede, para um tipo de quarto existente no hotel, em um período válido, mas onde nenhum dos quartos do tipo desejado está disponível no período, é representado no grafo pelo caminho 02345671. Por fim, o último cenário que representa a situação onde a reserva é efetuada com sucesso, é representado no grafo pelo caminho 0234567891. É importante notar que as transições foram ponderadas com um valor de probabilidade. As probabilidades são atribuídas a fim de tentar garantir que os testes reflitam ou cubram os usos mais esperados para o sistema. Neste exemplo, ao atingir o vértice 3, o grafo pode tomar dois caminhos diferentes: ir para o vértice 1, indicando que o uso do componente é finalizado porque o hotel solicitado não pertence à cadeia, fato que tem 10% de chance de ocorrer, ou ir para o vértice 4, dando continuidade ao uso, o que tem 90% de chance de acontecer. Da mesma forma, ao atingir o vértice 5, o grafo novamente se bifurca, podendo seguir para o vértice 1, terminando o uso porque o tipo de quarto desejado não existe no hotel solicitado, o que apresenta 20% de chance de ocorrer, ou ir para o vértice 6, dando continuidade ao uso, o que tem 80% de chance de ocorrer. Mais uma vez, no vértice 6 o grafo se divide, podendo seguir para o vértice 1, indicando que o uso é finalizado porque o período informado para

a reserva não é válido, o que apresenta 10% de chance de ocorrer ou seguir para o vértice 7, dando continuidade ao uso, com 90% de acontecer. Por fim, no vértice 7, o grafo mais uma vez se bifurca, podendo seguir para o vértice 1, terminando o uso porque não existe um quarto disponível para se realizar a reserva, o que apresenta 50% de chance de acontecer, ou seguir para o vértice 8, dando continuidade ao uso, também com 50% de chance de ocorrer. Essas probabilidades podem ser definidas baseando-se no conhecimento dos usuários sobre o domínio do problema ou em dados históricos obtidos do uso de outras aplicações pertencentes ao mesmo domínio.

É importante notar que no nosso grafo, as transições que chegam ao vértice final são rotuladas com uma espécie de comentário que dá um significado textual para o caminho percorrido no grafo. Essas transições representam sempre o fim da troca de mensagens representada nos diagramas de seqüências. Em termos de automação, o rótulo inserido nessas transições não está adequado. É importante definir uma forma que esteja condizente com os demais rótulos do grafo. Esse é mais um ponto que pode ser melhorado em trabalhos futuros.

Aplicando o método ao estudo de caso temos a seguinte situação: definimos na etapa de planejamento que o caso de uso Fazer Reserva tem alto nível de prioridade e portanto podem ser realizados até 3 casos de teste para ele. Devemos usar o grafo produzido para este caso de uso para selecionar 3 caminhos que serão usados como casos de teste. A seleção que fizemos foi baseada nas probabilidades definidas no grafo. Dessa forma, em cada vértice que se ramifica, escolhemos a ramificação com maior probabilidade de ocorrência. Aplicando este princípio, chegamos à seguinte seleção de caminhos para o caso de uso Fazer Reserva:

- 0234567891 - Reserva realizada com sucesso.
- 02345671 - Reserva não realizada: quarto não disponível.
- 0234561 - Reserva não realizada: período inválido.

Vale ressaltar que é de grande valia construir um algoritmo que permita realizar a seleção automática dos caminhos do grafo, baseando-se no princípio da maior probabilidade. A concepção deste algoritmo também pode vir ajudar na utilização prática do método.

É importante ressaltar ainda que esta técnica usada no Cleanroom pode garantir que os usos esperados com maior frequencia pelo sistema sejam testados. Porém, pode-se também

optar por uma combinação de escolha desta forma como uma escolha mais determinística, constituída de um ou mais cenários indicados pelos clientes.

Tendo selecionado os caminhos ou casos de teste que serão executados, o próximo passo é selecionar dados que nos levem por esses caminhos. Entretanto, a técnica proposta para gerar os oráculos de teste na TOTEM pode facilitar a escolha dos dados. Por esse motivo, estamos gerando os oráculos primeiro e em seguida selecionando os dados de teste.

Gerando os Oráculos

A definição precisa dos oráculos de teste é imprescindível para automatizar a atividade de teste. A técnica TOTEM propõe que seja construída uma tabela de decisão para cada expressão regular que representa os cenários de uso do componente. Esta tabela deve conter as condições de realização do uso e as ações que serão tomadas pelo componente diante da ocorrência do uso. A principal fonte de informação para construção desta tabela de decisão são as pré e pós-condições definidas para as operações das classes que fazem parte do componente.

Mais uma vez vamos usar o caso de uso Fazer Reserva para exemplificar a geração dos oráculos.

Como já vimos, o caso de uso Fazer Reserva apresenta 5 cenários de uso possíveis, representados pela seguinte expressão regular:

(1) $fazerReserva_{GerenciadordeHoteis}.getHotel_{GerenciadordeHoteis}$

+

(2) $fazerReserva_{GerenciadordeHoteis}.getHotel_{GerenciadordeHoteis}.fazerReserva_{Hotel}.$
$getTipoQuarto_{Hotel}$

+

(3) $fazerReserva_{GerenciadordeHoteis}.getHotel_{GerenciadordeHoteis}.fazerReserva_{Hotel}.$
$getTipoQuarto_{Hotel}.isPeriodoReservaOk_{Hotel}$

+

(4) $fazerReserva_{GerenciadordeHoteis}.getHotel_{GerenciadordeHoteis}.fazerReserva_{Hotel}.$
$getTipoQuarto_{Hotel}.isPeriodoReservaOk_{Hotel}.$
$verificarDisponibilidade_{Hotel}$

+

(5) $fazerReserva_{GerenciadordeHoteis}.getHotel_{GerenciadordeHoteis}.fazerReserva_{Hotel}.$
$getTipoQuarto_{Hotel}.isPeriodoReservaOk_{Hotel}.\ verificarDisponibilidade_{Hotel}.$
$gerarCodReserva_{Hotel}.Reserva_{Reserva}$

Para cada termo dessa expressão é necessário identificar suas condições de execução e expressá-las em OCL. É importante ressaltar que a especificação OCL é construída com base no modelo de informação - parte "visível ao usuário do componente".

A seguir mostramos as condições de execução, expressas em OCL, para cada termo definido acima para o caso de uso Fazer Reserva [2].

O primeiro termo (1), para ser executado, exige que o hotel onde se deseja fazer a reserva não faça parte da rede de hotéis [3]. Então, para este termo, a condição de execução é a seguinte:

A: not Gerenciador de Hoteis.hoteis->exists(h:Hotel | h.nome=reserva.hotel)

A expressão acima indica que dentre os hotéis controlados pela classe Gerenciador de Hoteis não deve existir nenhum cujo nome seja igual ao nome do hotel da reserva passado como parâmetro para a operação fazerReserva da classe Gerenciador de Hoteis.

O termo (2) indica que o hotel desejado pertence à rede de hotéis, mas o tipo de quarto desejado na reserva não existe no hotel informado. Para este caso, temos a seguinte condição de execução:

B: Gerenciador de Hoteis.hoteis->exists
(h:Hotel | h.nome=reserva.hotel and
not h.tiposQuarto->exists(tq:TipoQuarto | tq.nome=reserva.tipoQuarto))

O termo (3) indica que o hotel desejado pertence à rede de hotéis, o tipo de quarto desejado existe no hotel informado, mas o período informado para reserva não é válido.

[2] O contexto usado nas condições é Gerenciador de Hoteis :: fazerReserva

[3] Os termos 1, 2, 3 e 4 representam caminhos alternativos. O termo 5 representa o cenário de sucesso

Para este caso, temos a seguinte condição de execução:

C: Gerenciador de Hoteis.hoteis->exists
(h:Hotel | h.nome=reserva.hotel and
(h.tiposQuarto->exists
(tq:TipoQuarto | tq.nome=reserva.tipoQuarto) and
($reserva.dInicio \leq Today()$ or
$reserva.dInicio \geq reserva.dFim$)))

O quarto termo (4) da expressão regular representa o cenário onde o hotel desejado na reserva pertence à rede de hotéis, o tipo de quarto desejado existe no hotel informado, o período da reserva é válido, mas não existe nenhum quarto disponível do tipo desejado no período informado. Para este termo, é válida a seguinte condição de execução:

D: Gerenciador de Hoteis.hoteis->exists
(h:Hotel | h.nome=reserva.hotel and
(h.tiposQuarto->exists
(tq:TipoQuarto | tq.nome=reserva.tipoQuarto) and
(reserva.dInicio > Today() and
reserva.dInicio < reserva.dFim) and
(h.reservas->select(cancelada=false and
tipoQuarto=reserva.tipoQuarto and
(($dInicio \leq reserva.dInicio$ and
reserva.dInicio < dFim) or
(dInicio < reserva.dFim and
$reserva.dFim \leq dFim$) or
($reserva.dInicio \leq dInicio$ and
$reserva.dFim \geq dFim$)))-> size =
(h.quartos->select(tipo=reserva.tipoQuarto)->size))))

O último termo (5) da expressão regular representa o cenário onde a reserva é criada

com sucesso. A condição de execução deste termo é a seguinte:

E: Gerenciador de Hoteis.hoteis->exists
(h:Hotel | h.nome=reserva.hotel and
(h.tiposQuarto->exists
(tq:TipoQuarto | tq.nome=reserva.tipoQuarto) and
(reserva.dInicio > Today() and
reserva.dInicio < reserva.dFim) and
(h.reservas->select(cancelada=false and
tipoQuarto=reserva.tipoQuarto and
(($dInicio \leq reserva.dInicio$ and
reserva.dInicio < dFim) or
(dInicio < reserva.dFim and
$reserva.dFim \leq dFim$) or
($reserva.dInicio \leq dInicio$ and
$reserva.dFim \geq dFim$)))-> size <
(h.quartos->select(tipo=reserva.tipoQuarto)->size))))

Tendo identificado as condições de execução de cada cenário do caso de uso, o próximo passo é definir que mudanças de estado ocorrem no componente com a execução do caso de uso. Além disso, deve-se identificar também quais mensagens são retornadas para o ator do caso de uso.

Para o exemplo do caso de uso Fazer Reserva, temos as seguintes mensagens possíveis:

I : Hotel não cadastrado.
II : Tipo de quarto não existe no hotel desejado.
III : Período inválido.
IV : Nenhum quarto disponível do tipo desejado no período e hotel informados.
V : Reserva criada com sucesso.

A mudança de estado prevista para a execução deste caso de uso é a seguinte [4]:

```
Gerenciador de Hoteis->exists
    (h:Hotel | h.nome = reserva.hotel and
            (h.reservas->select(r | r.codigo=codReserva and
                                    r.dFim=reserva.dFim and
                                    r.dInicio=reserva.dInicio and
                                    r.tipoQuarto.nome=reserva.tipoQuarto and
                                    r.cliente=reserva.cliente and
                                    r.cancelada = false)->size = 1))
```

A tabela de decisão que resume todas as informações necessárias para a construção dos casos de teste e oráculos do caso de uso Fazer Reserva é mostrada na Tabela 5.3. Cada versão na tabela representa um termo na expressão regular, que representa um cenário de uso. Para cada termo são então associados as condições de execução, as mensagens retornadas para o usuário após sua execução e a indicação se a sua execução causa ou não uma mudança de estado no componente.

	Condições					Ações					
						Mensagem					Mudança de Estado
Versões	A	B	C	D	E	I	II	III	IV	V	
1	Sim	Não	Não	Não	Não	Sim	Não	Não	Não	Não	Não
2	Não	Sim	Não	Não	Não	Não	Sim	Não	Não	Não	Não
3	Não	Não	Sim	Não	Não	Não	Não	Sim	Não	Não	Não
4	Não	Não	Não	Sim	Não	Não	Não	Não	Sim	Não	Não
5	Não	Não	Não	Não	Sim	Não	Não	Não	Não	Sim	Sim

Tabela 5.3: Tabela de Decisão para o Caso de Uso Fazer Reserva

As informações coletadas durante esta etapa do processo poderão auxiliar na seleção dos

[4]O contexto dessa expressão é também Gerenciador de Hoteis :: fazerReserva

dados de teste e servirão de base para a implementação dos casos de teste e dos oráculos.

Selecionando os Dados

Selecionar os dados de entrada para um caso de teste é ainda um dos maiores problemas enfrentados pelos testadores de software, principalmente dados que sejam capazes de revelar comportamentos anômalos.

A natureza variável dos dados envolvidos no teste de um componente dificulta muito a geração automática destes dados, especialmente por estarmos tratando de objetos.

Existem atualmente algumas ferramentas que já nos fornecem alguns dados selecionados de forma aleatória [MPO01].

Neste método nos limitamos a fornecer apenas orientações sobre como selecionar os dados necessários para a execução dos casos de teste. A geração automática destes dados pode ser auxiliada pelas ferramentas já existentes.

Dentre as técnicas sistemáticas existentes para seleção dos dados de teste, pode-se destacar a técnica de particionamento por equivalência [Som96; Bei90]. B. Beizer [Bei95] propõe uma técnica semelhante, denominada teste de domínio.

Essa técnica parte do princípio que os dados de entrada de um programa podem ser agrupados em classes que apresentam características comuns e que o programa se comporta da mesma forma para todos os membros de uma mesma classe.

Usando essa técnica, o trabalho de selecionar os dados de teste consiste em identificar as partições e escolher dados particulares dentro de cada partição. A identificação das partições deve ser baseada na especificação e documentação do software. Neste método podemos usar as condições de execução definidas durante a geração dos oráculos para identificar partições adequadas ao teste. A escolha dos dados tanto pode ser feita de forma aleatória, como de forma mais direcionada, a fim de obter dados mais prováveis de revelar erros. Na escolha direcionada consideramos os dados encontrados nos limites da partição, por representarem normalmente valores atípicos, e dados considerados típicos, encontrados no meio da partição.

Usando as orientações propostas nesta técnica, fizemos a seleção dos dados usados como entrada para a execução dos casos de teste. Para exemplificar a seleção dos dados, usamos mais uma vez o caso de uso Fazer Reserva. O cenário exemplificado é a situação onde a

reserva não é realizada porque o período não é válido. Um período inválido é caracterizado da seguinte forma:

- Data de início da reserva é menor ou igual à data corrente;
- Data de início da reserva é maior ou igual à data final da reserva.

Para a primeira situação, identificamos três partições:

- Data de início da reserva é anterior à data corrente;
- Data de início da reserva é igual à data corrente;
- Data de início da reserva é posterior à data corrente;

A técnica propõe que sejam selecionados dados nos limites e no meio da partição. Seguindo esta orientação, para a primeira partição, selecionamos a data de início para um dia antes da data corrente e para vários dias antes da data corrente, por exemplo, dez dias antes.

A segunda partição tem apenas uma data disponível, que é a data de início igual à data corrente. Então selecionamos também esta data para executar o caso de teste.

Por fim, a terceira partição não apresenta dados que levem o programa a percorrer o caminho que estamos interessados em testar, porque todos os dados desta partição nos leva a um período válido. Então, nenhum valor foi selecionado desta partição.

Em resumo, para testar a situação em que a reserva não é realizada porque o período é inválido, selecionamos os seguintes dados:

- Início da reserva igual à data de ontem.
- Início da reserva igual a 10 dias antes da data atual.
- Início da reserva igual à data atual.

O mesmo processo foi aplicado para selecionar os dados que serviram de entrada para os demais casos de teste.

5.1.3 Construção e Execução dos Testes e Verificação dos Resultados

Nesta etapa, usamos as informações geradas até o momento para implementar os casos de teste e oráculos. As tabelas de decisão construídas durante a geração dos oráculos são especialmente úteis.

Em nosso estudo de caso, fizemos uso da ferramenta JUnit para implementar os casos de teste e oráculos. A ferramenta foi também utilizada para executar os testes e analisar os resultados obtidos.

Os casos de teste e oráculos para o caso de uso Fazer Reserva foram implementados através da classe TestaFazerReserva. Usamos principalmente as informações existentes na tabela de decisões, elaborada durante a especificação dos oráculos, para implementar esta classe.

A classe contém um método para testar cada versão definida na tabela. Para cada versão, implementamos no método correspondente, as condições de execução definidas na tabela. Por fim, verificamos a mensagem que é retornada e a ocorrência ou não de mudança de estado e comparamos com as definições da tabela para indicarmos se o teste obteve sucesso ou não.

Para exemplificar a construção dos testes, vamos analisar alguns métodos da classe TestaFazerReserva. As referências a objetos usados nos métodos são mostradas no Código 1.

```
1  // Referencia para o Gerenciador de Hoteis
2  GerenciadorDeHoteis g;
3
4  // Referencia para Hotel
5  Hotel h;
6
7  // Referencia para Reserva
8  TReserva tRes;
9
10 // Referencias para informacoes do hotel.
11 String codHotel, nomeHotel;
12
13 // Referencias para informacoes do tipo quarto
14 String nomeTQ;
15 Float preco;
16
17 // Referencias para informacoes da reserva.
18 String codRes, hRes, tqRes, c;
19 Data inicio , fim;
```

Código 1: Referências para objetos usados na classe TestaFazerReserva

Os objetos necessários aos testes são instanciados no método setUp(), mostrado no Código 2.

```
1  // Instancia todos os objetos necessários aos testes.
2  protected void setUp() {
3     // Instancia um gerenciador de hoteis
4     g = new GerenciadorDeHoteis();
5
6     //  Define as informações de um hotel
7     codHotel = new String("01");
8     nomeHotel = new String("Mar Azul Hotel");
9
10    // Adiciona o hotel ao gerenciador
11    g.addHotel(new Hotel(codHotel, nomeHotel));
12
13    // Obtém uma referência para o hotel adicionado
14    h = g.getHotel(nomeHotel);
15
16    // Define os dados de um tipo de quarto
17    nomeTQ = new String("Duplo Luxo");
18    preco = new Float(110);
19
20    // Inclui o tipo de quarto no hotel
21    h.incluirTipoQuarto(nomeTQ, preco);
22
23    // Define as informacoes da reserva
24    hRes = nomeHotel;
25    inicio = new Data();
26    fim = new Data();
27    tqRes = nomeTQ;
28    c = new String("0001");
29    tRes = new TReserva(hRes, inicio, fim, tqRes, c);
30 }
```

Código 2: Método setUp() da classe TestaFazerReserva

No Código 3, mostramos o método testFazerReservaHotelInvalido(). Neste método, é

testada a situação onde uma reserva não é criada porque o hotel informado não é válido.

```
1  // Realizar reserva em um hotel invalido
2  public void testFazerReservaHotelInvalido() {
3          // Define o hotel da reserva
4          tRes.setHotel("Hotel Ouro Branco");
5
6          // Tenta realizar a reserva no hotel definido
7          Vector r = (Vector)g.fazerReserva(tRes);
8
9          // Verifica se a reserva foi realizada
10         assertEquals(new Integer(4), r.elementAt(0));
11         assertEquals(null, g.getHotel("Hotel Ouro Branco"));
12 }
```

Código 3: Método testFazerReservaHotelInvalido() da classe TestaFazerReserva

Na linha 4, definimos o hotel onde desejamos realizar a reserva. Neste caso, de acordo com as condições estabelecidas na tabela de decisão, o hotel definido não faz parte da rede de hotéis, portanto, não está registrado no gerenciador de hotéis, representado pelo objeto g, referenciado na linha 7. Nesta linha 7, o caso de teste é executado através da chamada ao método fazerReserva(), definido na interface GerenciadorDeHoteis. Na linha 10 verificamos se o retorno do método fazerReserva() está correto. Nesta situação, em que o hotel onde se pretende fazer a reserva não pertence à rede, espera-se que o código retornado pelo método seja 4. Essa informação é obtida da especificação da operação fazerReserva() (ver Tabela 4.3). A linha 11 analisa se ocorreu mudança no estado do componente. Neste caso, nenhuma mudança é esperada (de acordo com a tabela de decisão), então verificamos se o hotel onde tentamos realizar a reserva continua não fazendo parte da rede, ou seja, o hotel não está registrado no Gerenciador de Hotéis, portanto, esperamos que o retorno do método getHotel() seja null.

Vejamos agora a situação onde a reserva não é realizada porque o tipo de quarto informado não é válido no hotel onde se deseja fazer a reserva. O método que testa esta situação é mostrado no Código 4.

```
1     // Fazer reserva para um tipo de quarto inexistente no hotel
2     public void testFazerReservaTipoQuartoInvalido() {
3
4     // Define o hotel e otipo de quarto da reserva
5     tRes.setHotel(nomeHotel);
6     tRes.setTipoQuarto("Simples");
7
8     // Tenta realizar reserva com os dados definidos
9     Vector r = (Vector)g.fazerReserva(tRes);
10
11    // Verifica se a reserva foi realizada
12    assertEquals(new Integer(1), r.elementAt(0));
13    assertEquals(null, h.getTipoQuarto("Simples"));
14 }
```

Código 4: Método testFazerReservaTipoQuartoInvalido() da classe TestaFazerReserva

As linhas 5 e 6 definem o hotel e o tipo de quarto da reserva. O hotel é um hotel pertencente à rede de hotéis, portanto está registrado no Gerenciador de Hotéis. Entretanto, o tipo de quarto não deve existir no hotel. Essas informações foram tiradas da seção Condições, da tabela de decisão. Na linha 9 o caso de teste é executado através da chamada ao método fazerReserva(), da interface GerenciadorDeHoteis. As linhas 12 e 13 analisam o resultado da execução do caso de teste. Na linha 12, é verificado se o retorno do método está correto. Esperamos neste caso que seja retornado o código 1, de acordo com a especificação da operação fazerReserva() (ver Tabela 4.3). A linha 13 verifica se ocorreu alguma mudança de estado no componente. Mais uma vez, nenhuma mudança é esperada, então esperamos que o tipo de quarto continue não existindo no hotel informado, portanto é esperado que o retorno do método getTipoQuarto(), da classe Hotel, seja null.

Por estes exemplos chamamos a atenção para o fato de que a construção e execução dos testes e a verificação dos resultados é inteiramente baseada na tabela de decisão e nas especificações das operações.

É importante notar que para aumentar as chances de automatização do método, pode ser interessante incorporar à tabela de decisão algumas informações encontradas nas especifi-

cações das operações, especialmente as informações sobre as saídas geradas pelas operações.

5.2 Considerações Finais

Neste capítulo apresentamos de forma detalhada as etapas envolvidas no método de teste proposto neste trabalho, bem como cada atividade desenvolvida em cada etapa.

Alguns aspectos que não ficaram bem definidos no método foram apontados e podem ser considerados em trabalhos posteriores.

De forma resumida, a aplicação do método aqui proposto consiste das seguintes etapas:

- Planejamento dos Testes: apoiado na análise de riscos, define-se a expectativa de cobertura dos testes, considerando-se as funcionalidades mais críticas e freqüentes do componente.

- Especificação e Construção dos Testes: grafos representando os diferentes cenários de uso do componente são construídos, a partir dos quais são selecionados os casos de teste que deverão ser implementados e executados, utilizando-se dados selecionados durante esta etapa. Tabelas de decisão representando os oráculos de teste, contendo informações sobre cada cenário de uso, suas entradas e saídas esperadas são também construídas durante a etapa e utilizadas na implementação dos casos de teste.

- Execução dos Testes e Análise dos Resultados: os casos de teste implementados na etapa anterior são executados, utilizando-se os dados também selecionados na etapa anterior. As informações contidas nas tabelas de decisão ajudam a decidir se a execução do caso de teste obteve ou não sucesso, ou seja, é possível analisar se a entrada fornecida para o componente produziu ou não o resultado esperado.

A aplicação de todas as etapas descritas acima pode trazer muitos benefícios relacionados à qualidade final do produto desenvolvido, entretanto, acreditamos que a aplicação de um subconjunto mínimo destas etapas já contribui em alcançar tal qualidade. Obviamente algumas etapas são intimamente dependentes das etapas anteriores, mas atividades como a realização de inspeções nos modelos desenvolvidos, análise de riscos e planejamento dos testes, que podem ser realizadas sem a necessidade de aplicar o método completamente, podem trazer melhorias significativas na qualidade final do produto.

O próximo capítulo relata os resultados obtidos com a realização do experimento, os problemas encontrados e as sugestões de melhoria do método.

Capítulo 6

Conclusões

Apresentamos neste trabalho a proposta de um método de teste funcional para verificação de componentes de software. A idéia de seu desenvolvimento surgiu do estudo feito a respeito dos principais desafios enfrentados pela comunidade de engenharia de software baseado em componentes. Dentre esses desafios destacava-se a dificuldade em se montar sistemas rapidamente a partir de componentes individuais. Da análise desse desafio, percebemos que a construção de um método de teste que verificasse as propriedades individuais dos componentes poderia ser de grande valia para ajudar a compor os sistemas.

Com essa proposta inicial, realizamos um estudo detalhado sobre testes e componentes, do qual derivamos os principais requisitos desejados neste método. A partir desses requisitos delineamos o objetivo principal deste trabalho que foi desenvolver um método de teste funcional aplicável a componentes de software, que fosse prático e apresentasse potencial para automação, já que, do ponto de vista prático, a aplicação de um método de teste que não apresente ferramental computacional de apoio é, na maioria das vezes, inviável. Era nosso desejo ainda que o método estivesse integrado a uma metodologia de desenvolvimento de componentes, a fim de suprir a necessidade, já notada anteriormente por outros autores, de inserir a atividade de teste de forma sistemática no processo de desenvolvimento, a fim de melhor aproveitar os recursos disponíveis e, principalmente, minimizar a propagação de problemas existentes na especificação do software para etapas posteriores do desenvolvimento. Tínhamos ainda a idéia de disponibilizar os artefatos e resultados das execuções dos testes para o cliente do componente, pois acreditamos que esta informação pode ter algum valor para o cliente desenvolver novos testes para o componente. Por fim, pretendíamos

ainda neste trabalho, desenvolver um experimento aplicando o método desenvolvido a fim de adquirirmos uma idéia preliminar da aplicação prática do mesmo.

As próximas seções discutem os resultados obtidos e as propostas de trabalhos futuros.

6.1 Discussão dos Resultados

O método de teste desenvolvido neste trabalho é o resultado da combinação sistemática da metodologia de desenvolvimento de componentes Componentes UML com um processo de teste que utiliza as técnicas de teste TOTEM e Cleanroom para gerar os principais artefatos de teste. O método faz uso dos diagramas de seqüência UML para produzir um modelo de teste que permita selecionar de forma adequada os casos de teste mais críticos para o componente. O método faz uso ainda da especificação OCL dos contratos dos componentes para derivar os oráculos de teste.

A metodologia Componentes UML define quatro etapas não seqüenciais para o desenvolvimento de software baseado em componentes: definição dos requisitos, modelagem dos componentes, materialização dos componentes e montagem da aplicação. Existe ainda uma etapa de teste que é executada após a materialização dos componentes.

No método que desenvolvemos adaptamos a etapa final de teste da Componentes UML e a transformamos em um processo de teste, que é executado em paralelo ao processo de desenvolvimento. O processo de teste considerado no método abrange as etapas de planejamento, especificação, construção, execução, verificação dos resultados e empacotamento. No planejamento, decidimos que tipos de teste serão realizados e quais são as nossas expectativas com relação aos testes. A etapa de especificação gera os modelos de teste que são usados para derivar os casos de teste, dados e oráculos. Durante a construção, os casos de teste e oráculos especificados na etapa anterior são implementados em alguma linguagem de programação. A etapa de execução tem por objetivo executar os casos de teste com os dados selecionados durante a etapa de especificação. Na análise dos resultados usamos os oráculos especificados e implementados nas etapas anteriores para decidir se os casos de teste foram executados com sucesso. Na prática, os oráculos já fazem esta análise, comparando os resultados esperados com os obtidos e nos fornecendo a informação sobre o sucesso ou fracasso da execução dos testes. Por fim na etapa de empacotamento, os artefatos

e resultados dos testes realizados são reunidos em um pacote e disponibilizados junto com o componente. Consideramos que os artefatos de teste poderão ser disponibilizados como um componente à parte, tendo sua interface composta pelos métodos especificados nos oráculos e pela especificação dos dados de teste utilizados.

Cada uma dessas etapas foi inserida apropriadamente no processo de desenvolvimento, a fim de permitir que a atividade de teste aconteça em paralelo ao desenvolvimento do componente. Dessa forma, o planejamento dos testes pode ser feito à medida que acontece o levantamento de requisitos, a especificação dos testes pode ser feita durante a modelagem dos componentes, a construção e execução dos testes e a análise dos resultados podem acontecer em paralelo à materialização dos componentes. O empacotamento dos testes somente ocorre no final do processo quando o componente está pronto e seus testes realizados.

Além da concepção do método, este trabalho produziu ainda um estudo de caso realizado que nos forneceu as primeiras impressões sobre a aplicação prática do método.

A aplicação escolhida foi um Sistema de Reserva de Hotel. Esta aplicação tem por objetivo permitir a realização de reservas em diferentes hotéis pertencentes a uma rede de hotéis. Modelamos a aplicação utilizando Componentes UML, com as adaptações propostas no Capítulo 4. Foram modelados três componentes: Gerenciador de Hotéis, responsável por manter as informações dos hotéis, quartos e reservas, Gerenciador de Clientes, responsável por manter as informações dos clientes e o componente Sistema de Reservas, responsável por interligar a camada de interface com o usuário e a camada de negócio onde estão os outros dois componentes.

A preocupação com a testabilidade dos componentes nos levou a realizar sessões de inspeção e revisão dos modelos a fim de que produzíssemos modelos corretos, completos e consistentes. Entretanto, por limitações de tempo, tivemos um número muito reduzido de sessões, apenas duas, com duração média de 2 horas cada uma. Essa limitação não chegou a comprometer a qualidade dos artefatos gerados, mas acreditamos que a depender do grau de complexidade da aplicação em desenvolvimento, um número maior de sessões pode ser necessário.

As atividades de teste foram aplicadas apenas ao componente Gerenciador de Hoteis, por ser o componente mais complexo da aplicação. Este componente exporta uma interface, IFHotel, que comunica aos seus clientes os serviços que ele é capaz de fornecer, tais como,

fazer reserva, cancelar reserva, incluir quarto etc. Cinco classes fazem parte da estrutura interna do componente e colaboram entre si para entregar os serviços declarados na interface do componente.

O componente foi implementado utilizando a linguagem de programação Java, bem como seus casos de teste e oráculos. Usamos a ferramenta JUnit para implementar os casos de teste e oráculos. Implementamos 6 classes de teste e um total de 23 casos de testes distribuídos entre as classes como mostra a Tabela 6.1.

Classe de Teste	**Casos de Teste**
Fazer Reserva	5
Remover Tipo Quarto	4
Cancelar Reserva	4
Incluir Quarto	4
Incluir Tipo Quarto	3
Alterar Tipo Quarto	3

Tabela 6.1: Quantidade de Casos de Teste Implementados por Classe de Teste

Além das 6 classes de teste construídas para testar as funcionalidades do componente, foram implementadas ainda mais duas classes de teste para fazer teste de unidade das classes Quarto e TipoQuarto.

A execução dos casos de teste detectou apenas alguns poucos "bugs" na implementação das classes. Nenhum erro nas funcionalidades do componente chegou a ser detectado. Acreditamos que isso tenha acontecido por três motivos principais:

- A complexidade das funcionalidades do componente é baixa.
- A qualidade da especificação do componente está muito boa.
- A distância existente entre a especificação OCL e a linguagem de programação Java é pequena, o que facilita a produção de um código mais correto.

Enfim, o fato de não ter detectado bugs não implica necessariamente que os testes planejados foram ineficazes. Na prática, processos onde existe um comprometimento em desenvolver o software correto[1] desde o início são caracterizados por um número mínimo de bugs detectados ao final [PTLP99]. Isto ocorre devido, principalmente, às sucessivas revisões empregadas. As atividades preliminares de teste auxiliam bastante na melhoria da qualidade de artefatos e implementabilidade do sistema. No final, os testes automáticos gerados podem constituir uma poderosa ferramenta de documentação (especificação executável do sistema) que pode ser usada não só na certificação da qualidade do produto, mas também em testes de regressão decorrentes da acomodação de novas funcionalidades.

6.1.1 Considerações Finais

Alcançamos o nosso objetivo como proposto no Capítulo 1, concebendo um método que cobre cada uma das principais etapas de um processo de teste e está integrado a um processo de desenvolvimento de componentes.

Outros trabalhos foram encontrados que propõem técnicas para teste de software orientado a objetos [BL01; CTF01; DF94; WTWS98; BBP96], teste de sistemas baseados em componentes [HIM00] e teste de componentes [MTY01; MS02; BG01]. Entretanto todas essas técnicas estão isoladas, fora do contexto de um processo de desenvolvimento e, em geral, dificilmente cobrem todas as etapas do processo de teste aqui adotado.

O método ainda tem a vantagem de sugerir a disponibilização para os clientes dos componentes os casos de teste, oráculos e dados que foram usados na execução dos testes, na forma de um componente à parte.

Por fim, produzimos um exemplo, que mesmo simples, nos deu as primeiras impressões sobre a utilização do método. Basicamente, tivemos as seguintes impressões:

- Quanto à quantidade e utilidade dos artefatos a produzir. A metodologia de desenvolvimento que escolhemos sugere a produção de poucos artefatos, ou seja, Componentes UML sugere que sejam criados apenas artefatos que nos ajudem a compreender melhor o domínio do negócio e a descobrir uma solução adequada para o problema. Procuramos desenvolver um método que usasse apenas os artefatos já sugeri-

[1] Correto por construção.

dos na metodologia de desenvolvimento para a produção dos testes. Dessa forma, adicionamos apenas a produção das expressões regulares a partir dos diagramas de sequência e derivamos modelos de uso a partir das expressões regulares. Na prática, as expressões regulares e os modelos de uso são modelos equivalentes, apenas apresentados de formas diferentes, sendo assim, efetivamente apenas um novo artefato foi gerado com a finalidade de se desenvolver os testes dos componentes.

- Quanto ao grau de dificuldade de aplicar o método. Na nossa opinião, o fato do método ter sido sistematicamente inserido no processo de desenvolvimento facilitou significativamente a sua utilização prática, visto que temos uma definição das etapas que devemos seguir, das atividades que temos que realizar e da sequência em que esse processo acontece. Visto de forma isolada, fora do contexto do processo de desenvolvimento, acreditamos que o o método apresentaria um grau de dificuldade maior em termos de aprendizagem e aplicação prática.

- Quanto à manutenção do sistema face aos diferentes artefatos desenvolvidos. O problema de manter os artefatos atualizados face às mudanças ocorridas no software não foi resolvido com este método. Obviamente aumentamos o número de artefatos produzidos, portanto o trabalho de manter os artefatos também cresceu. Entretanto, acreditamos que o desenvolvimento de ferramentas de suporte ao método possa ao menos minimizar o impacto da mudança sobre os artefatos de teste, ou mesmo eliminá-lo por completo.

- Quanto à dependência do método em relação ao processo de desenvolvimento escolhido. Embora tenhamos utilizado Componentes UML para ilustrar a aplicação prática do método, acreditamos que o método possa ser facilmente utilizado em conjunto com um outro processo de desenvolvimento de componentes, desde que este outro processo de desenvolvimento gere os diagramas de seqüência necessários para a construção dos modelos de teste. Considerando ainda a equivalência entre diagramas de colaboração e de seqüência, acreditamos também que o método possa ser facilmente adaptado processos que sugerem a construção de diagramas de colaboração ao invés de diagramas de seqüência. Neste caso, há a clara necessidade de se definir como transformar o diagrama de colaboração no grafo a partir do qual serão selecionados os

casos de teste.

- Quanto à possibilidade de aplicar o método a sistemas orientados a objetos. Embora nosso objetivo fosse desenvolver um método para testar componentes de software, durante o desenvolvimento do estudo de caso suspeitamos que o método produzido poderia também ser utilizado para testar a integração entre as classes que compõem um sistema orientado a objetos. Obviamente, não comprovamos tal suspeita, mas acreditamos que o método compreenda uma classe maior de software do que aquela a que nos propomos no início deste trabalho.

É necessário ainda fazer ajustes e complementar alguns aspectos deste método aqui apresentado. Esse aspectos são discutidos na próxima seção.

6.2 Trabalhos Futuros

Com a finalização deste trabalho, foi possível fazer uma análise do mesmo, da qual resultou o seguinte conjunto de propostas para continuidade do mesmo:

- **Melhorar o método em termos de sua especificação**

  O método aqui proposto define apropriadamente cada etapa existente no processo e cada atividade desenvolvida nas etapas. Entretanto, alguns aspectos de sua definição podem ser melhorados, especialmente para aumentar seu potencial para automação. Dentre esses aspectos, ressaltamos a necessidade de incorporar às expressões regulares, geradas durante a atividade de seleção dos casos de teste, outros elementos dos diagramas de seqüência, tais como mensagens de retorno e expressões de guarda. Atualmente, são incorporadas às expressões regulares apenas as chamadas síncronas.

- **Complementar o método em termos de sua abrangência**

  O método limita-se a verificar os componentes individualmente, entretanto um método completo deveria considerar também a possibilidade de realização de testes de integração e sistema. Pode-se também analisar a viabilidade de incorporar técnicas estruturais ao método. Uma proposta de realização de teste de integração de sistemas baseados em componentes pode ser encontrada em [Gou02].

- **Validar o método**

  Desenvolvemos um estudo de caso que nos forneceu uma visão prática da aplicação do método, entretanto, a fim de se adquirir maior confiança na viabilidade prática do mesmo, é necessário desenvolver ainda outros estudos de caso.

- **Desenvolver ferramentas de suporte ao método**

  Na nossa opinião, esse é um aspecto importante para viabilizar a aplicação prática do método. Percebemos, da nossa própria prática e experiência de engenharia de software, que o desenvolvimento de testes de maneira manual nem sempre é satisfatório. A existência de ferramentas de apoio pode facilitar e incentivar a utilização do método.

Bibliografia

[ABB+01] Colin Atkinson, Joachim Bayer, Christian Bunse, Erik Kamsties, Oliver Laitenberger, Roland Laqua, Dirk Muthig, Barbara Paech, Juergen Wuest, and Joerg Zettel. *Component-based Product Line Engineering with UML*. Component Software Series. Addison-Wesley, 2001.

[BAS94] Stéphane Barbey, Manuel Ammann, and Alfred Strohmeier. Open issues in testing object-oriented software. In Karol Frühauf, editor, *ECSQ'94 (European Conference on Software Quality), Basel, Switzerland, October 17-20 1994*, pages 257–267. vdf Hochschulverlag AG an der ETH Zürich, 1994. Also available as Technical Report EPFL-DI No 94/45.

[BBB+00] Felix Bachman, Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord, and Kurt Wallnau. Volume II: Technical concepts of component-based software engineering. Technical report, Carnegie Mellon Software Engineering Institute, 2000.

[BBP96] S. Barbey, D. Buchs, and C. Peraire. A theory of specification-based testing for object-oriented software. *Lecture Notes in Computer Science*, 1150:303–320, 1996.

[Bei90] Boris Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, 2nd edition, 1990.

[Bei95] Boris Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, 1995.

[BG01] S. Beydeda and V. Gruhn. An integrated testing technique for component-based

software. In *International Conference on Computer Systems and Applications*, pages 328–334. IEEE Computer Society Press, 26–29 Junho 2001.

[BL01] Lionel Briand and Yvan Labiche. A UML-based approach to system testing. *Lecture Notes in Computer Science*, 2185:60–70, 2001.

[CD01] John Cheesman and John Daniels. *UML Components: A Simple Process for Specifying Component-Based Software*. Addison-Wesley, 2001.

[Cle95] Paul C. Clements. From subroutines to subsystems: Component-based software development. *American Programmer*, 8(11), 1995.

[CR99] Juei Chang and Debra J. Richardson. Structural specification-based testing: Automated support and experimental evaluation. In Oscar Nierstrasz and Michel Lemoine, editors, *ESEC/FSE '99*, volume 1687 of *Lecture Notes in Computer Science*, pages 285–302. Springer-Verlag / ACM Press, 1999.

[CTF01] Philippe Chevalley and Pascale Thevenod-Fosse. Automated generation of statistical test cases from UML state diagrams. In *Proceedings of the 25th Annual International Computer Software and Applications Conference (COMPSAC 2001)*, pages 61–72, Chicago, Outubro 2001. ACM Press.

[DF94] Roong-Ko Doong and Phyllis G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 3(2):101–130, Abril 1994.

[DW98] Desmond F. D'Souza and Alan Cameron Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley Object Technology Series. Addison-Wesley Publishing Company, Reading, Mass., 1 edition, 1998.

[FS98] Martin Fowler and Kendall Scott. *UML Distilled: Applying the Standard Object Modeling Language*. Addison-Wesley Object Technology Series. Addison-Wesley, 1998.

[Gou02] Cidinha Costa Gouveia. Um método de teste de integração para sistemas baseados em componentes. Proposta de dissertação de Mestrado submetida à COPIN/UFCG, Dezembro 2002.

[Har00] Mary Jean Harrold. Testing: A roadmap. In *Proceedings of the 22th International Conference on Software Engineering (ICSE-00)*, pages 61–72, NY, 4–11 Junho 2000. ACM Press.

[HC01] George T. Heineman and William T. Councill. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, 1 edition, Junho 2001.

[HIM00] Jean Hartmann, Claudio Imoberdorf, and Michael Meisinger. UML-based integration testing. *ACM Transactions on Software Engineering and Methodology*, 8(11):60–70, 2000.

[JO95] Zhenyi Jin and Jeff Offutt. Integrating testing with the software development process. Technical Report ISSE-TR-95-112, George Mason University, Agosto 1995.

[Lar98] Craig Larman. *Applying UML and Patterns: an Introduction to Object-Oriented Analysis and Design*. Prentice Hall PTR, Upper Saddle River/NJ, 1998.

[LYC+99] S. Lee, Y. Yang, E. Cho, S. Kim, and S. Rhew. Como: A UML-based component development methodology. In *Proceedings of the 6th Asia Pacific Software Engineering Conference (APSEC 1999)*, pages 54–63, Japan, Dezembro 1999. IEEE Computer Society Press.

[Mac00] Patrícia D. L. Machado. Testing from structured algebraic specifications. In *AMAST: 8th International Conference on Algebraic Methodology and Software Technology*, volume 1816 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[MPO01] Patrícia D. L. Machado, Adriano L. S. Pinto, and Killy A. Oliveira. Automating formal testing from casl specifications. In *Proceedings of IV WMF - Workshop on Formal Methods*, 2001.

[MS01] John D. McGregor and David A. Sykes. *A Practical Guide to Testing Object-Oriented Software*. Object Technology Series. Addison-Wesley, 2001.

[MS02] Patrícia D. L. Machado and Don T. Sannella. Unit testing for casl architectural specifications. In *27th International Symposium on Mathematical Foundations of Compouter Science*, Lecture Notes in Computer Science. Springer-Verlag, Agosto 2002.

[MTY01] Eliane Martins, Cristina Toyota, and Rosileny Yanagawa. Constructing self-testable software components. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks (DSN '01)*, pages 151–160, Washington - Brussels - Tokyo, Julho 2001. IEEE.

[Pre87] Roger S. Pressman. *Software Engineering*. McGraw-Hill, 1987.

[PTLP99] Stacy J. Prowell, Carmen J. Trammell, Richard C. Linger, and Jesse H. Poore. *Cleanroom Software Engineering: Technology and Process*. The SEI Series in Software Engineering. Addison-Wesley, 1999.

[RJB98] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley Object Technology Series. ACM Press and Addison-Wesley, 1998.

[Som96] I. Sommerville. *Software Engineering*. Addison-Wesley Publishing Company, 1996.

[Szy98] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 1998.

[WBGP01] Torben Weis, Christian Becker, Kurt Geihs, and Noël Plouzeau. A UML meta-model for contract aware components. *Lecture Notes in Computer Science*, 2185:442–456, 2001.

[WK99] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modelling with UML*. Object Technology Series. Addison-Wesley, Reading/MA, 1999.

[WTWS98] H. Watanabe, H. Tokuoka, W. Wu, and M. Saeki. A technique for analyzing and testing object-oriented software using coloured Petri nets. In *Asia Pacific Software Engineering Conference*, pages 182–195. IEEE Computer Society Press, 1998.