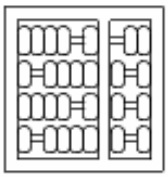


INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

Um processo para testes de sistemas com reuso de componentes

Ivan R. D. C. Perez, Camila Rocha,
Regina Moraes, Josiane A. Cardoso,
Ruth F. Soliani, Eliane Martins

Technical Report - IC-06-021 - Relatório Técnico

October - 2006 - Outubro

Um processo para testes de sistemas com reuso de componentes

Ivan Rodolfo Duran Cruz Perez Camila Rocha
Regina Moraes Josiane Aparecida Cardoso
Ruth Fabiana Soliani Eliane Martins

Resumo. *Devido a necessidades de mercado, onde se busca cada vez mais desenvolver softwares em prazos mais curtos, o desenvolvimento baseado em componentes (DBC) tem sido cada vez mais empregado na indústria de desenvolvimento de software. Contudo, para manter sempre um bom nível da qualidade do software em desenvolvimento, um processo de testes para DBC se faz necessário, tanto para garantir qualidade dos componentes em desenvolvimento quanto para validação dos componentes que serão reusados, neste caso, verificando se o componente satisfaz a especificação desejada na arquitetura do sistema. Com isto propõe-se um processo de testes genérico adaptado para processos DBC.*

1. Introdução

O desenvolvimento baseado em componentes é uma tendência na Engenharia de Software hoje em dia. Seja usando componentes de terceiros (desenvolvimento com reuso), seja desenvolvendo componentes para serem usados em outros projetos (desenvolvimento para reuso), o fato é que o reuso é aplicado em qualquer tipo de sistema.

Para que o reuso atinja os objetivos de melhoria da produtividade e da qualidade dos sistemas desenvolvidos, atividades de verificação e validação devem ser conduzidas ao longo de todo o desenvolvimento e manutenção de sistemas baseados em componentes. Em particular, a testabilidade deve ser uma qualidade a ser assegurada, dado que o reuso implica na realização de muitos testes. A cada vez que um componente é reutilizado em um novo contexto ele deve ser retestado [Weyuker98].

Para se conseguir uma boa testabilidade as atividades de teste devem ser integradas ao processo de desenvolvimento. Em outros termos, os testes devem ser planejados desde cedo no ciclo de desenvolvimento.

Neste texto é descrita a metodologia de testes proposta para integrar um processo de desenvolvimento baseado em componentes. A metodologia é genérica, podendo ser aplicada a qualquer processo, mas foi criada para integrar o processo definido em [Brito05].

Dentre outros objetivos, a metodologia visa garantir que as atividades de teste sejam realizadas ao longo de todo o processo de desenvolvimento. Com isso, diminuem-se os custos com a fase de aplicação dos testes, por um lado, pois atividades que podem ser realizadas antes desta fase o serão, efetivamente. Em segundo lugar, quer se evitar o tradicional problema que ocorre quando tempo ou recursos são escassos, cuja solução em geral consiste em sacrificar a fase de testes. Para isso, a metodologia proposta também se preocupa com a automação dos testes, de forma a tornar sua geração e execução mais eficiente.

O relatório está estruturado da seguinte forma: a Seção 1 apresenta a terminologia usada nesse texto, bem como conceitos básicos sobre testes de software. A Seção 2 apresenta aspectos básicos de desenvolvimento baseado em componentes, bem como uma apresentação das técnicas para a melhoria da testabilidade de componentes. Em particular, a técnica considerada nesse trabalho também é apresentada. A Seção 3 apresenta a metodologia de testes proposta e a Seção 4 conclui o trabalho.

2. Conceitos básicos

Esta seção apresenta as bases teóricas sobre testes de *software* utilizadas neste trabalho. Durante todo o texto será utilizada, para os termos *fault*, *error* e *failure*, os termos falha, erro e defeito, respectivamente, de acordo com a terminologia em português definida por Leite e Loques [LL87]. .

O padrão IEEE (IEEE STD. *Glossary of Software Engineering Terminology*, padrão 610.12/1990) define que os problemas introduzidos no *software* pelo desenvolvedor são chamados de falhas (*fault*). Enganos podem ser cometidos tanto na especificação quanto no código do sistema. Quando uma falha é ativada durante a execução do *software*, um erro é gerado (*error*). Caso o problema se manifeste nas fronteiras do sistema, ocorre um defeito (*failure*), que pode ser percebido pelo usuário.

Seguindo a terminologia apresentada, teste de *software* é o processo de executar um sistema com o objetivo de encontrar defeitos [Myers79]. A ocorrência de defeito indica a presença de falhas no software. Um caso de teste bem sucedido é aquele que revela uma falha no sistema que ainda não tinha sido descoberta. Os testes não são capazes de provar a ausência de falhas em um sistema: caso nenhum defeito ocorra durante os testes, isso não significa que o sistema não tenha falhas.

Uma limitação dos testes é o fato de não sabermos, a priori, onde estão as falhas. Também não é possível realizar testes exaustivos, isto é, aplicar ao sistema todas as combinações de entradas possíveis ou exercitar todos os caminhos de execução possíveis. Tal é impraticável, pois mesmo em sistemas triviais, o número de casos de teste seria tão grande, em alguns casos, até infinito, o que inviabilizaria sua execução. Assim, é importante adotar critérios de seleção que produzam casos de teste com alta probabilidade de produzir defeitos, revelando assim a presença de falhas. Para a criação desses casos de teste, é recomendada a adoção de uma estratégia que possibilite sua criação em paralelo com o desenvolvimento, de forma incremental.

2.1 Fases de Teste

Uma estratégia de testes deve envolver tanto testes de baixo nível, que verificam se uma pequena parte do código fonte foi corretamente implementada, quanto testes de alto nível, que validam funções do sistema relativas aos requisitos do cliente. Pode ser dividida em quatro fases [Beizer97],

1. ***Teste de Unidade***: Uma unidade é a menor porção de um *software* que pode ser executada. O teste de unidade verifica se uma porção do código executa adequadamente a sua funcionalidade, isoladamente do resto do sistema. Por isso, geralmente o caso de teste faz uso de *drivers* e *stubs*. Um *driver* é um elemento (classe, programa principal ou *software* externo) que aplica os casos de teste à unidade sob teste [Binder99, cp.19]. O *driver* é responsável por fornecer os dados de entrada, coletar os dados de saída e apresentá-los ao usuário. Um *stub* substitui módulos necessários para a execução da unidade, simulando seu comportamento. A utilização de *drivers* e *stubs* possibilita que a unidade seja testada isoladamente.
2. ***Teste de Componente***: Um componente é integração de diversas unidades, com interfaces bem definidas. Nesta fase o componente é testado de acordo com a especificação das funcionalidades e de sua estrutura. Também podem ser necessários *drivers* e *stubs*.
3. ***Teste de Integração***: Nesta fase os componentes são integrados, e os casos de teste são direcionados à descoberta de erros arquiteturais, relacionados às interfaces dos componentes. A integração pode se dar com o uso de uma abordagem *big-bang* [Pressman97], na qual todos os componentes são integrados de uma só vez, dispensando, o uso de *drivers* e *stubs,* mas dificultando a localização de falhas. O mais recomendado é,

portanto, o uso de uma abordagem incremental, na qual os componentes são integrados pouco a pouco. O uso de *drivers* e *stubs* pode ser necessário neste caso. Testes de integração reutilizam casos de testes gerados durante as fases de teste anteriores para verificar se o novo componente integrado não afeta os demais.

4. ***Teste de Sistema***: Nesta fase todo o sistema é integrado, incluindo outros sistemas, *hardware*, etc. São testadoss todas as funcionalidades propostas na especificação do sistema, bem como seus atributos requeridos para qualidade, segurança, desempenho e confiabilidade.

Além das quatro fases acima, existem ainda os **testes de regressão**, realizados durante a manutenção do sistema. Nesta fase um conjunto de casos de teste é novamente executado com o objetivo de atestar que partes do sistema não foram afetadas pelas mudanças realizadas.

2.2 Critérios de Teste

Como não é possível testar exaustivamente um sistema, é importante a adoção de critérios que estabeleçam condições a serem satisfeitas durante os testes. Os três principais critérios são: o critério funcional, o estrutural e o baseado em falhas.

No critério funcional de testes, também conhecido como "caixa-preta", os detalhes internos do sistema não são considerados na elaboração e execução dos casos de teste [Beizer97], apenas suas funcionalidades. São fornecidas entradas ao sistema, e suas saídas são verificadas de acordo com o comportamento especificado.

As fontes utilizadas pelos testes funcionais são as especificações de requisitos e de projeto. A principal dificuldade na realização de testes funcionais é a obtenção do menor conjunto de dados de teste de entrada para a obtenção do maior número de defeitos, a chamada eficácia dos testes [Freedman91]. Algumas abordagens para os testes funcionais são [Pressman01]

1. ***Partição de Equivalência***: essa técnica é utilizada para a escolha de dados de entrada para os testes. O domínio de entrada é dividido em classes de equivalência, que representam um conjunto de valores válidos ou inválidos para as condições de entrada. Por exemplo, caso os dados válidos sejam representados por uma faixa de valores, três classes de equivalência são derivadas: os números menores que a faixa, os dentro da faixa, e os maiores que a faixa. Os valores de entrada dos testes são escolhidos dentro de cada classe de equivalência.
2. ***Análise de Valores Limite***: essa técnica também é utilizada para a escolha de dados de entrada, combinada com a de partição de equivalência. Como um grande número de erros tende a ocorrer nos limites dos valores de entrada, a técnica de análise de valores limite determina que os dados de entrada escolhidos sejam valores limite de cada uma das classes de equivalência.
3. ***Baseado em Modelos***: o comportamento do sistema é descrito em forma de modelos e diagramas como os propostos pela UML [UML03]. Normalmente testes baseados em modelo utilizam cobertura do diagrama e comportamento. Cobertura é uma forma de exercitar todos os comportamentos e estados mostrados em um modelo, como percorrer todos estados em um diagrama de estados UML. Esta cobertura (varredura) é feita buscando sempre encontrar defeitos na implementação.

No critério estrutural, também conhecido como caixa branca ou caixa de vidro, considera-se a estrutura interna do sistema e sua implementação. Os casos de teste são derivados do código fonte do sistema, e exploram a cobertura do código para que os possíveis defeitos sejam detectados.

Dentre os critérios estruturais podemos citar [Pressman01]: teste baseado em caminhos

básicos ou linearmente independentes, que busca executar caminhos de execução dentro do código; teste de condições, no qual as condições lógicas do código são testadas; teste baseado em dados, que seleciona os caminhos a serem testados de acordo com a localização das definições e usos de variáveis.

O critério baseado em falhas busca testar o comportamento do sistema na presença de falhas. Falhas podem acontecer tanto no próprio sistema sob teste, quanto nos sistemas com os quais ele interage. As principais técnicas são mutação e injeção de falhas. Nos testes de mutação as falhas são introduzidas diretamente no código do programa; casos de teste são produzidos de forma a revelar a presença dessas falhas [DLS78]. Já a técnica de injeção de falhas consiste em modificar seja o código (fonte ou objeto) ou o estado do sistema durante sua execução. Seu objetivo é determinar o comportamento do sistema em presença de falhas ou erros [HT+97]. Os testes de robustez é um tipo de teste que pode usar injeção de falhas [KS+03].

2.3 Processo de Teste

Um processo de testes tem duas grandes atividades: preparação e realização dos testes [Paula01]. Durante a preparação, são elaborados os planos e especificações de teste, além da implementação dos casos de teste. Durante a realização, os testes são executados e seus resultados analisados.

Como apenas para a fase de realização é necessário que o sistema esteja implementado, a fase de preparação pode ser feita em paralelo com o desenvolvimento, como mostra a Figura 1.

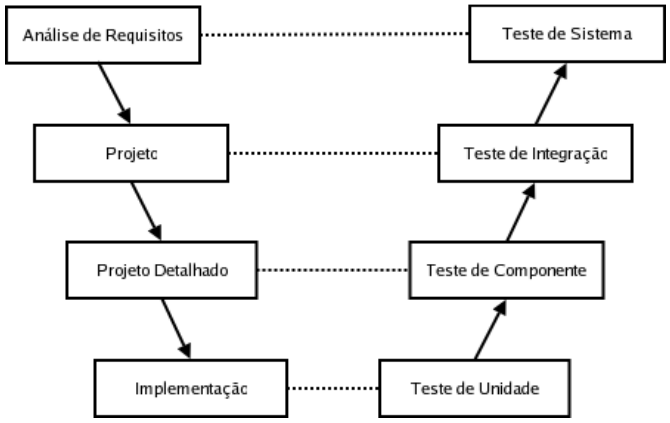


Figura 1 Paralelismo entre as fases do processo de desenvolvimento e da estratégia de teste

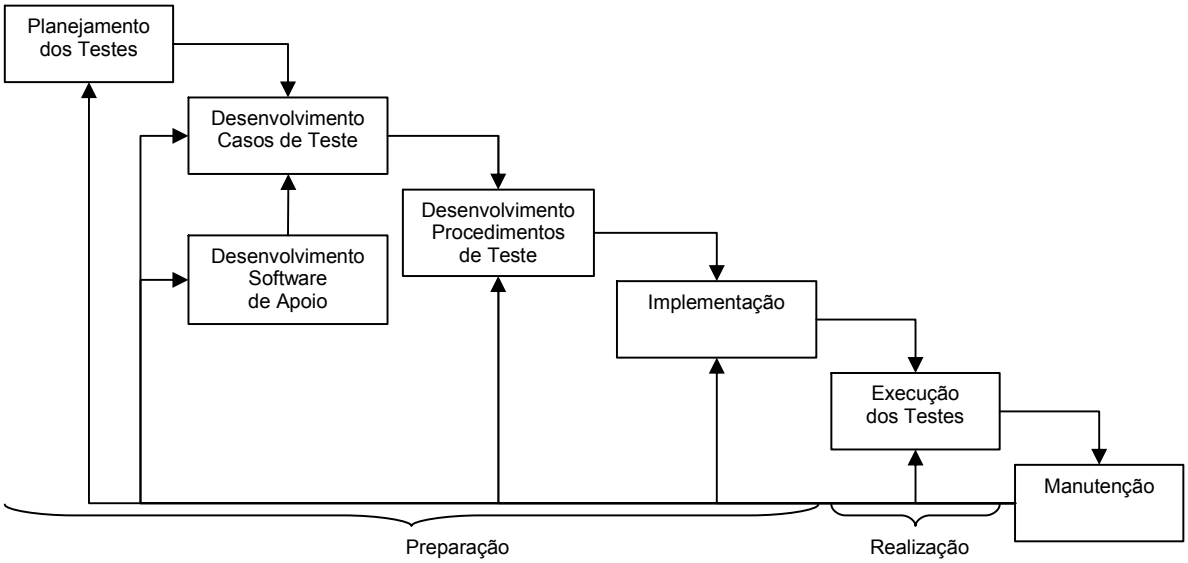


Figura 2 O processo de testes realizado a cada fase

O plano de testes de sistema é baseado na análise de requisitos, o plano de testes de integração, na arquitetura do sistema, e o de componentes, no projeto detalhado. Os testes de unidade geralmente são planejados e executados pelos próprios programadores durante a implementação [Paula01].

Cada uma das fases da estratégia de testes segue um processo de teste, cujas fases são ilustradas na Figura 2 [Drabick04]. Apesar da utilização da notação do desenvolvimento em cascata, resultados de uma fase podem interferir na fase anterior. Cada uma das fases deve ser documentada: o conjunto de documentos de acordo com a norma IEEE 829 [IEEE87, Binder99] é ilustrado na Figura 3, de acordo com as fases nas quais são confeccionados.

A primeira fase é o planejamento dos testes, na qual são definidos quais serão os itens e aspectos a testar, as abordagens adotadas, os critérios de cobertura, os recursos necessários e cronogramas para a realização dos testes. Essas informações compõem o documento "Plano de teste".

O documento "Especificação de Projeto de Testes" é preenchido no início da segunda fase, "Desenvolvimento dos Casos de Teste". Apresenta um refinamento do plano de testes, detalhando as funcionalidades e características a serem testadas. Durante a segunda fase também é preenchido o documento "Especificação dos Casos de Testes", contendo as entradas e saídas esperadas para cada caso de teste, além das condições gerais para sua execução. Paralelamente, se necessário, são especificados componentes de apoio aos testes, como os *drivers* e *stubs* [Binder99, cp.19].

Na terceira fase, "Desenvolvimento dos Procedimentos de Testes", são descritos os passos para a execução de um conjunto de casos de teste, registrados no documento "Especificação dos Procedimentos de Testes". A fase de preparação é encerrada com a implementação dos casos de teste e componentes de apoio, e com a preparação do ambiente.

A fase de realização é iniciada após a implementação do sistema ou partes deste. Nesta fase os casos de teste são executados, e são produzidos os seguintes documentos:

- Diário de teste, com os registros cronológicos da execução.
- Relatório resumo dos testes, com a descrição resumida das atividades de teste e uma avaliação dos resultados.

- Relatório de incidentes de testes, registrando eventos ocorridos durante os testes que mereçam análise posterior.
- Relatório de encaminhamento do item de teste, encaminhando as correções para as equipes responsáveis.

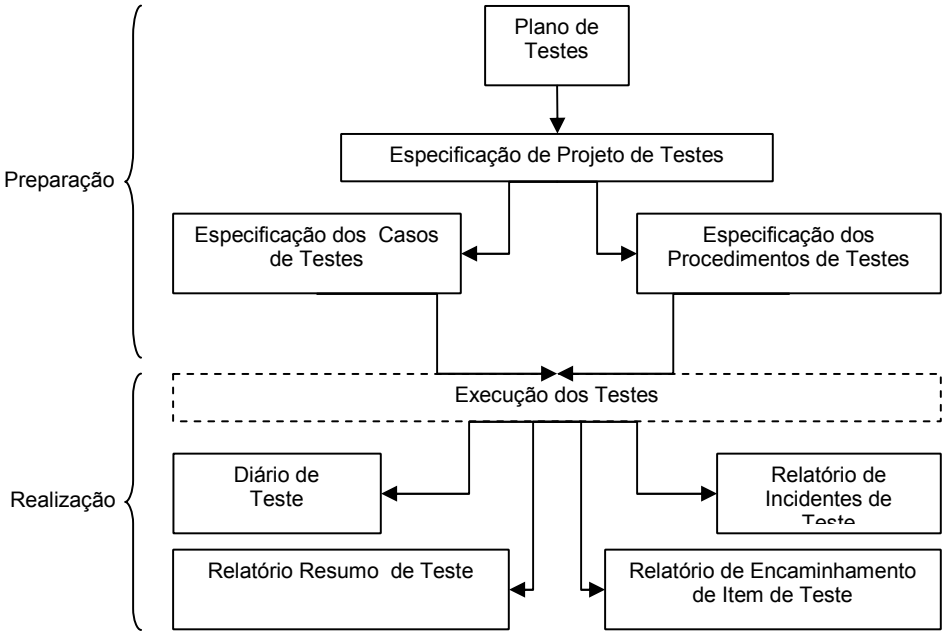


Figura 3 Documentos de teste propostos pela norma IEEE 829

3. Desenvolvimento Baseado em Componentes

3.1 Noção de componente

O conceito de desenvolvimento baseado em componentes surgiu devido às pressões sofridas pela indústria de *software* por prazos mais curtos e produtos de maior qualidade. Ainda não há um consenso na literatura quanto ao conceito de componentes de *software* [BW98]. Neste projeto, foi a adotada a visão de Szyperski: "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 desenvolvido independentemente e ser utilizado para a composição de sistemas de terceiros" [Szyperski98]. Componentes possuem interfaces bem definidas, tanto para suas dependências (interfaces requeridas) quanto com os serviços oferecidos (interfaces providas).

Um componente pode assumir vários formatos, cada um relativo a uma etapa do ciclo de desenvolvimento [CD02]:

- ***Padrão***: formato imposto ao componente para que ele se adapte a um determinado modelo de componentes, facilitando a posterior montagem da aplicação. Exemplos de padrões de componentes são OMG Corba [Corba] e Sun Enterprise JavaBeans [EJB].
- ***Especificação***: componente como a abstração arquitetural, representando uma unidade de composição separada do restante do sistema, com funcionalidades e interfaces bem definidas. É parte da arquitetura e independente de implementação.
- ***Implementação***: componente que pode ser executado, podendo ser integrado a outro

software. É o conceito utilizado para os componentes de prateleira, ou COTS.
Um sistema baseado em componentes é composto por componentes que interagem entre si para fornecer as funcionalidades desejadas. Na Figura 4 pode-se ver a notação UML 2.0 [UML04] para um sistema baseado em componentes: as interfaces requeridas são representadas por círculos preenchidos, e as requeridas, por semicírculos. Uma conexão ilustra uma dependência entre os componentes.

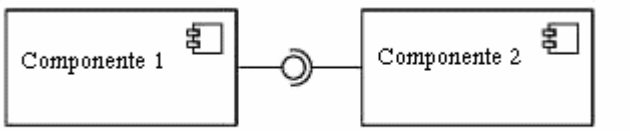


Figura 4 Notação UML 2.0 para um sistema baseado em componentes.

Os principais benefícios proporcionados pelo desenvolvimento baseado em componentes são a redução no tempo de desenvolvimento e a simplificação da manutenção, reduzindo o custo do sistema como um todo [Hopkins00, Vitharana03]. A redução no tempo é proporcionada pela reutilização de código, diminuindo o tempo necessário às fases de projeto detalhado e implementação. A manutenção é simplificada pela facilidade da localização das mudanças, e por essas não afetarem outros componentes.

As subseções seguintes apresentam um processo de desenvolvimento genérico para desenvolvimento baseado em componentes. A abordagem *Design-by-contract* [Meyer97] aplicada no contexto do desenvolvimento baseado em componentes, também é descrita como a melhoria para a testabilidade de componentes.

3.2 Design-by-contract

O termo *Design-by-contract* [Meyer97] foi introduzido no contexto de orientação a objetos, como uma maneira de melhorar a confiabilidade dos sistemas, contribuindo para a sua correção. O conceito *Design-by-contract* "representa os relacionamentos entre uma classe e seus clientes como um acordo formal, expressando os direitos e obrigações de ambas as partes". O mesmo conceito é utilizado no contexto de interfaces de componentes [Szyperski98].

O contrato é especificado através de assertivas, que expressam "uma propriedade que algumas entidades do *software* têm que satisfazer em determinados pontos da execução" [Meyer97]. As propriedades devem cobrir o comportamento normal e excepcional esperado do componente, e podem ser relativas tanto a aspectos funcionais quanto não-funcionais do sistema (evitando-se, porém, referências à plataforma específica utilizada).

Um contrato é formado por três tipos de assertivas: invariante, pré-condição e pós-condição. Invariantes são propriedades que devem ser satisfeitas por todas as operações de uma interface, antes e depois de sua execução; pré-condições são específicas de cada operação e devem ser satisfeitas pelo cliente para que a operação seja corretamente executada; pós-condições também são específicas de operações e são propriedades garantidas pelo componente após a execução da operação.

Contratos são simples e práticos, porém limitados. Não é possível, por exemplo, no contexto de um componente caixa preta isolado, verificar se uma operação sem um valor de retorno foi corretamente executada, apenas se terminou normalmente. Operações como atualizações em um banco de dados, gravação de um arquivo ou ainda, informações mostradas na tela para um usuário, não são possíveis de serem verificadas pelas assertivas.

Apesar das limitações, contratos são de grande valia para os testes. Experimentos

comprovam que, mesmo sendo pouco precisos, a presença de contratos facilita a descoberta e localização de falhas [BLS02]. Os contratos podem ser satisfatoriamente utilizados representando os resultados esperados dos casos de teste, possibilitando economia de tempo na sua criação. Quando verificado em tempo de execução, uma violação do contrato manifesta a presença de falhas no sistema, seja no cliente (pré-condição violada) seja no componente (pós-condição ou invariante violada).

Além disso, os contratos contribuem para a melhoria da qualidade da documentação e para a diminuição da taxa de erros, pois aumentam o entendimento da equipe sobre a especificação do sistema.

A UML possui uma linguagem especial que pode ser utilizada para a especificação de contratos, a *Object Constraint Language* (OCL) [WK03]. É uma linguagem formal, baseada em teoria de conjuntos e lógica de primeira ordem, que permite aos desenvolvedores especificar restrições relativas ao modelo de objetos em UML textualmente, de uma forma precisa e simplificada. Com OCL, é possível especificar os contratos de uma forma precisa ainda na fase de especificação do sistema, possibilitando a geração automática dos contratos a serem verificados em tempo de execução.

3.4 Testabilidade

Testabilidade é a medida da probabilidade de ocorrer um defeito caso o software contenha falhas. Testes e testabilidade são complementares: os testes revelam as falhas, e a testabilidade sugere locais onde estas podem estar ocultas dos testes [VM95].

A testabilidade de um sistema é influenciada por várias características, relativas tanto ao desenvolvimento do componente quanto aos testes, mais especificamente, à especificação e à execução de casos de teste. Os principais fatores a serem considerados na construção de um sistema testável podem ser [Binder94]:

- *Representação*: A qualidade da documentação do sistema é crucial para sua testabilidade, já que os casos de teste são gerados a partir desta. É importante que a documentação seja mapeável à implementação e esteja sempre atualizada, para que os casos de teste tenham objetivos bem definidos e reflitam os requisitos reais do sistema.
- *Implementação*: Sistemas com componentes altamente coesos e pouco acoplados têm melhor testabilidade, pois facilitam a realização de testes de unidade e de componentes, e a localização das falhas. Fatores que contribuem para a baixa testabilidade são otimizações ligadas ao desempenho, concorrência e tratamento de exceções, por serem situações mais difíceis de serem simuladas em um ambiente de testes.
- *Capacidade de Teste Embutido* (*Built-in test*): A testabilidade pode ser melhorada com a adição de métodos *set/reset*, para a definição de um estado para o sistema; métodos relatores, que permitem a observação do estado do sistema; e assertivas, que implementam a abordagem *Design-by-contract.* A inclusão de tais mecanismos contribui fortemente para a observabilidade e controlabilidade do sistema, mas seu impacto sobre o armazenamento e o desempenho do sistema deve ser considerado durante os testes. É importante que sejam habilitados ou desabilitados automaticamente pelo compilador, e acessado através de interfaces específicas, separando-os das funcionalidades do componente.
- *Conjunto de Testes*: Deve ser bem documentado, para se reduzir os custos com a manutenção dos testes.
- *Ferramentas de teste*: A automação dos testes é crucial para a testabilidade do sistema, já que permite a realização de um grande volume de testes a um custo reduzido.
- *Processo de desenvolvimento*: Além de contribuir com a qualidade do sistema e de sua

documentação, um processo estruturado permite a construção dos casos de teste paralelamente ao desenvolvimento do sistema, gerando testes com maior cobertura e qualidade.

A metodologia de testes proposta visa atender aos itens citados acima.

3.5 Construção de componentes testáveis

Técnicas para melhoria da testabilidade e o conceito de *built-in test* (BIT) já vem sendo usado em hardware há décadas. Mais recentemente estas técnicas ganharam a atenção da comunidade de software. Um trabalho pioneiro foi o de D. Hoffman, com sua proposta de "componentes integrados de software" ou *software ICs* (do inglês Integrated Components) [Hoffman89]. Este trabalho também já propunha o uso de assertivas para implementar *Desgin–by-Contract*, bem como o uso da especificação para a derivação automática de testes, mais especificamente, de *drivers* de testes.

Depois disso, diversos autores têm proposto diferentes abordagens para a construção de componentes testáveis, isto é, contendo mecanismos BIT e formas de fazer acesso a esses mecanismos. Uma descrição geral dessas abordagens é dada em [BG03] e [Bhor01].

Apresentamos a seguir nossa abordagem para construção de componentes testáveis [RM04]. Foi proposta inicialmente uma abordagem para a construção de classes testáveis, baseada no trabalho em [Binder94], denominada ConCAT (de Construção de Classes Testáveis) [Toyota00]. A abordagem apresentada aqui foi baseada na proposta da ConCAT.

A construção de um componente testável requer adição de informação extra a ser utilizada nos testes. Para o projeto da estrutura desse componente, seguimos algumas diretrizes, quais sejam:

- *Facilidade de uso*: o componente testável deve ser desenvolvido e utilizado a um baixo custo de programação. A inclusão da instrumentação tem como objetivo facilitar a validação do componente no ambiente do usuário, que não conhece o componente detalhadamente.
- *Separação de interesses* (do inglês *Separation of concerns*): os mecanismos de testes e monitoração devem ser independentes tanto do componente sob teste (CST), quanto um do outro.
- *Extensibilidade* (do inglês *Extensibility*): a adição de novos mecanismos e a manutenção dos já existentes deve ser realizada a um baixo custo.
- *Intrusão reduzida*: os mecanismos incluídos no CST não devem gerar um impacto demasiado em seu desempenho e espaço de armazenamento.
- *Independência do código fonte*: os mecanismos devem ser incluídos ou removidos do COTS sem a necessidade de compilação, permitindo a instrumentação de componentes adquiridos de terceiros, os chamados COTS (do inglês *Commercial off-the-shelf*).

O componente testável é composto pela estrutura apresentada na Figura 5, oferecendo duas interfaces públicas adicionais: uma responsável pelos mecanismos de monitoração e outra pelos mecanismos de teste. Seguindo as diretrizes de separação de interesses e extensibilidade, o componente alvo (CA) foi acrescido de dois subcomponentes. Um deles implementa a interface de monitoração; o outro implementa a interface de testes. Essas interfaces devem ser usadas pelo *driver* para a preparação do CA para os testes. Esses subcomponentes são detalhados a seguir:

- *Componente de monitoração*, cuja interface oferece serviços para inclusão de mecanismos de monitoração no CA. São oferecidos três tipos de monitoração [GG+02]: operacional, que registra as interações entre os métodos públicos do CA (tanto das interfaces providas quanto requeridas); de erros, que registra as exceções lançadas pelo CA; e de estados, que registra o

estado de atributos públicos ou propriedades do componente.

- *Componente de testes*, o qual oferece serviços para a inclusão de assertivas no CA, bem como para a notificação de violações e também para a recuperação das especificações do componente mencionadas na Seção 3.4. Este componente possui duas interfaces; uma delas é pública, podendo ser usada pelo *driver* durante os testes. A outra interface tem visibilidade restrita; é utilizada pelo CA após sua instrumentação para o registro de violações de assertivas em um arquivo de *log*.

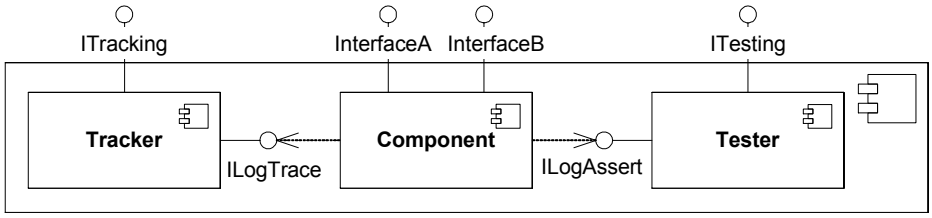


Figura 5 Arquitetura do Componente Testável

Mais detalhes sobre a arquitetura do componente testável pode ser encontra em [Rocha05].

4. Processo para desenvolvimento e testes de sistemas baseados em componentes

Esta seção apresenta o processo de testes proposto. Este processo pode ser integrado a qualquer processo de desenvolvimento. Nesse trabalho mostramos sua integração a um processo de desenvolvimento baseado em componentes proposto em [Brito05].

4.1 O Processo de desenvolvimento

O processo descrito em [Brito05] é um processo genérico, no sentido de ser constituído de atividades gerais, comuns à maioria das metodologias de DBC atuais. Com isso, pretende-se que ele seja facilmente adaptável aos vários processos utilizados, o que facilitaria sua utilização. Essa facilidade de utilização é decorrente da não imposição de um processo específico. Mudanças de processo costumam afetar a estrutura do desenvolvimento e por isso é considerada uma atividade arriscada. As subseções seguintes descrevem as fases desse processo.

Por se tratar de um processo de desenvolvimento de software, seu objetivo é permitir desenvolver um sistema a partir de descrições textuais do que se espera dele. Dessa forma, a partir de refinamentos sucessivos, são gerados artefatos cada vez mais específicos, a ponto de no final se ter os componentes do sistema especificados e implementados em uma linguagem de programação. Além disso, o foco no reuso obriga o processo a adicionar atividades que antecipem e maximizem o número de componentes reutilizados.

O diagrama na Figura 6 mostra o processo de desenvolvimento genérico usado no projeto. Este processo é composto de oito passos básicos, os quais são descritos resumidamente a seguir:

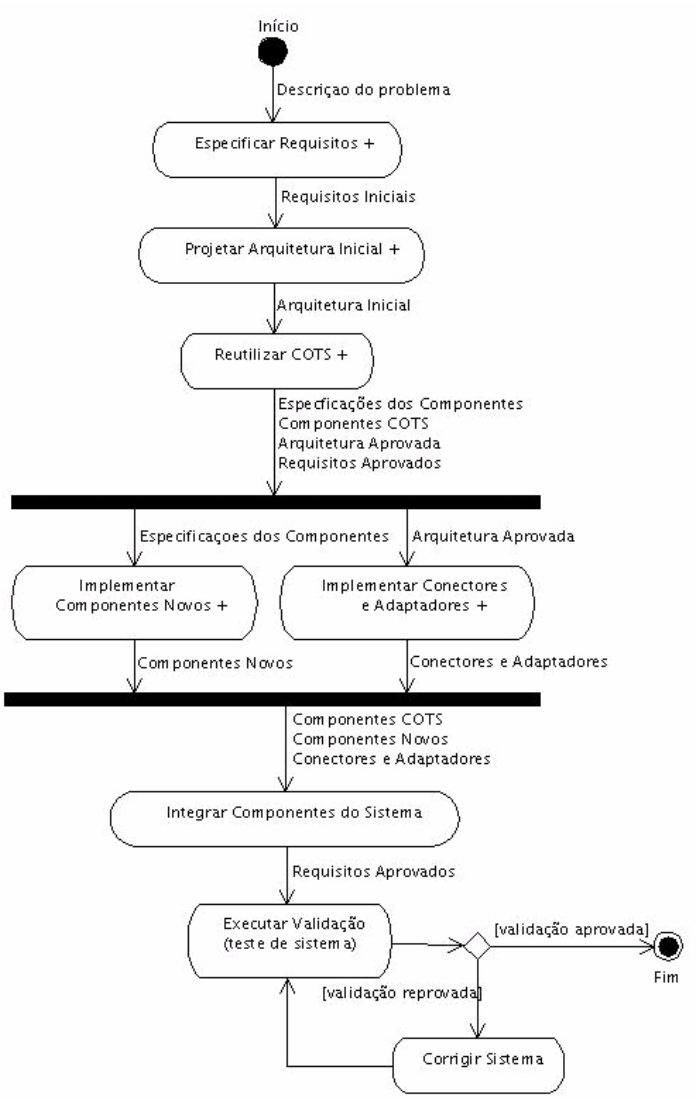


Figura 6 Fases do processo genérico DBC

- *Especificação de Requisitos*: Esta é a primeira fase a ser executada durante o desenvolvimento de um software. Em geral, os requisitos podem ser vistos como os objetivos esperados para o sistema, assim como as condições e capacidades necessárias para tal. Por se tratar de um processo voltado para o reuso de componentes, já nessa fase devem ser construídos protótipos a partir de componentes prontos.
- *Projeto arquitetural:* Nesta fase é definida a arquitetura do sistema. Nesta fase é escolhido o estilo arquitetural e modelada a arquitetura do sistema em desenvolvimento. Para que isso possa ser feito deve-se seguir um *workflow* para escolha do estilo arquitetural [Brito05] que passa por: Avaliar restrições arquiteturais, Listar arquiteturas compatíveis, priorizar atributos de qualidade, selecionar arquiteturas candidatas, selecionar arquiteturas mais adequadas e identificar componentes iniciais.
- *Reutilização de COTS*: Para o desenvolvimento de sistemas utilizando o processo proposto têm-se três formas de materializar o componente: reuso de componentes já utilizados pela

empresa, aquisição de componentes de terceiros, e por último, desenvolvimento de novos componentes. Nesta fase o foco principal é a seleção de componentes de terceiros, para que isto possa ser feito deve-se inicialmente identificar os COTS a serem reusados, em seguida avaliar o impacto na arquitetura e por ultimo validar o componente selecionado.

- *Implementação dos componentes Novos:* Na ausência de componentes que possam ser reusados para o desenvolvimento, então, deve-se desenvolver novos componentes para o sistema. Devido a característica recursiva deste processo, para componentes com alto grau de granularidade pode-se aplicar este processo de desenvolvimento recursivamente iniciando se da fase de especificação de requisitos.
- *Implementação dos Conectores e Adaptadores*: Neste ponto do desenvolvimento já se tem todos os componentes do sistema: falta apenas materializar as conexões entre eles. Essas conexões são realizadas através de conectores, componentes especiais que fazem ligação das interfaces requeridas de um componente às respectivas interfaces providas de outros. Na reutilização de componentes prontos, os componentes reutilizados nem sempre obedecem aos mesmos contratos especificados nos requisitos. Nesses casos pode ser necessário adaptar o componente reutilizado a fim de satisfazer os contratos estabelecidos, construindo adaptadores.
- *Integração dos Componentes do sistema:* Nesta fase devem-se integrar todos componentes e subsistemas para montar o sistema completo.
- *Validações & Correções do Sistema:* Basicamente, a validação do software consiste em uma seqüência de testes, elaborados a partir dos documentos de requisitos. Em sistemas reais, devido ao grande número de serviços que podem ser oferecidos, essa atividade deve ser executada automaticamente, com o auxílio de ferramentas CASE.

Dependendo da granularidade do componente, pode ser necessário executar o processo recursivamente, no intuito de reutilizar as partes que o constituem. O fato de o processo ser recursivo o deixa aplicável tanto ao desenvolvimento de sistemas de grande porte, quanto ao desenvolvimento de sistemas pequenos.

4.2. O processo de testes

O processo de testes proposto tem por objetivo permitir que atividades de teste possam ser realizadas em paralelo com o processo de desenvolvimento descrito em 4.1.

A realização em paralelo das atividades de teste e de desenvolvimento apresenta duas principais vantagens: i) as atividades de testes, mais precisamente a de Preparação (ver Figura 2), podem ser realizadas ao longo de todo o desenvolvimento, ficando para o final deste ciclo somente a execução e análise de resultados (Realização); ii) os testes podem ser desenvolvidos por uma equipe independente daquela de desenvolvimento. Dessa forma procura-se garantir que os testes determinem se o sistema atende aos requisitos dos usuários, e não ao que os desenvolvedores pensam que o sistema deve fazer [MS01].

Assim como o sistema, o desenvolvimento de testes também envolve uma fase de Planejamento, em que são desenvolvidos os Planos de Testes, conforme mostrado na Figura 3. Durante o planejamento de cada fase deve ser definido o escopo dos testes. Dado que não é viável testar todo o sistema de forma sistemática, é necessário escolher as partes a serem enfocadas durante os testes. Diversos autores recomendam uma análise de riscos [MS01, Perry95] para determinar as funções ou componentes mais críticos para o sistema e que, portanto, merecem especial atenção durante os testes.

Além disso, são definidos também nessa fase os recursos e o cronograma da fase de testes, entre outras atividades. Após definido o escopo de testes, a outra atividade de preparação diz respeito ao desenvolvimento dos casos de teste e do software de apoio, que constarão das

especificações de testes. Para cada fase do processo de desenvolvimento DBC são previstas fases de testes, as quais serão cobertos pela preparação e execução dos testes como visto na Seção 2.3. Na Figura 7 são mostradas de forma simplificada, algumas fases do processo DBC e qual fase de testes deve ser executada.

Para cada fase de teste mostrada na Figura 7 são realizadas as atividades mostradas na Figura 2. As atividades compreendidas na Preparação, conforme descrito na Seção 2.3, são realizadas durante o desenvolvimento. Assim, por exemplo, os testes de componentes podem ser planejados, os casos de testes, especificados, e os componentes de testes (tais como os *drivers* e *stubs*) também podem ser especificados durante a Implementação de Componentes, Conectores e Adaptadores.

Os itens a seguir descrevem cada fase de testes.

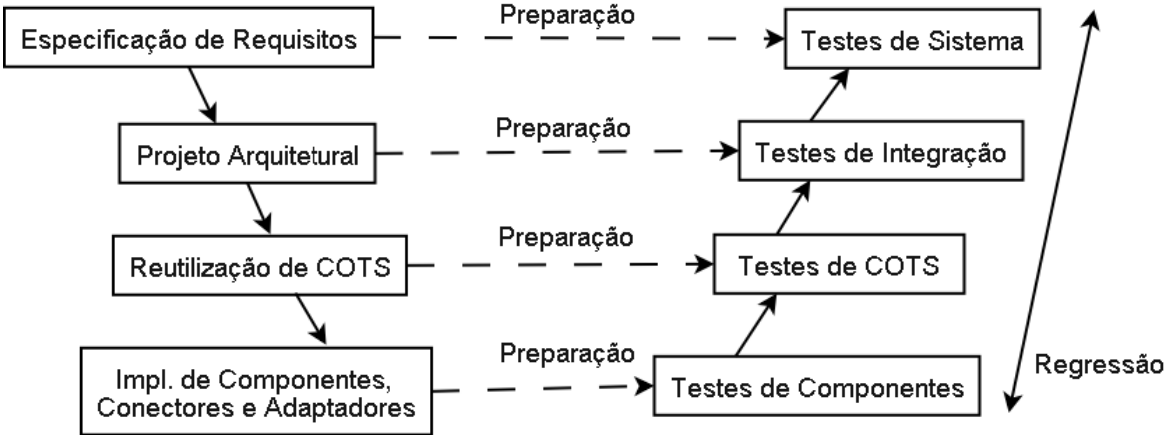


Figura 7 Processo de testes adaptado para DBC

4.3 Testes de sistema

Os Testes de Sistema verificam se o sistema, composto pela integração de componentes desenvolvidos (novos) e de terceiros, atende aos requisitos funcionais e aos atributos de qualidade desejados, tais como desempenho, usabilidade, portabilidade, segurança. No processo proposto ressaltamos dois tipos de teste.

- Testes de Funcionalidade, visando determinar se os requisitos funcionais do sistema foram atendidos.
- Testes de Robustez, visando determinar a robustez do sistema em presença de falhas de seus componentes.
- Testes de configuração, que visam verificar se o sistema é compatível com outras aplicações existentes, com as quais deverá interagir, e também com as diferentes plataformas de hardware e de software onde deverá operar.
- Testes de desempenho, que visam determinar se o sistema executa de acordo com as restrições de tempo e recursos especificados. Estes testes compreendem os testes de carga, que determinam como o sistema se comporta sob a variação de condições operacionais, tais como número de usuários ou número de transações. Compreendem também os testes de volume, que verifica o comportamento do sistema sob variações de volume de dados de entrada ou tamanhos de arquivos processados.

Outros tipos de testes podem ser considerados, dependendo dos atributos de qualidade requeridos pelo sistema, como por exemplo, testes de segurança (*security*), para determinar se o sistema resiste a acessos não permitidos, e assim por diante. Aqui, daremos enfoque a dois tipos de testes: os de Funcionalidade e os de Robustez.

Os Testes de Funcionalidades são definidos com base em modelos do comportamento do sistema. Técnicas de testes baseados em modelos [Binder99, cp.12] podem ser utilizadas para esse fim. Idealmente, os modelos utilizados durante o desenvolvimento para representar os diversos aspectos do sistema serão utilizados nos testes.

Os Testes de Robustez têm como objetivo verificar o comportamento de um sistema em presença de falhas ou quando este sistema precisa lidar com uma grande carga de dados de entrada [KS+03]. Podemos dizer que um software é robusto se, alimentado com entradas anômalas, não propaga erros que levem o sistema a apresentar um defeito. Dessa forma, o sistema demonstra que pode produzir serviços de confiança mesmo quando colocado num ambiente hostil [Voas98b]. Os Testes de Robustez são úteis para exercitar os mecanismos de tratamento de exceções implementados no sistema.

Para os Testes de Robustez será considerado o uso de injeção de falhas, que consiste em produzir ou simular a presença de falhas e observar o sistema sob teste para se verificar qual será sua resposta nessas condições [HT+97]. A técnica é um meio para se demonstrar de maneira experimental a qualidade do sistema e permitir que se observe o comportamento do mesmo durante sua execução em presença das falhas injetadas [Voas98b].

Para se aplicar a técnica de injeção de falhas é preciso que se definam quatro conjuntos: o conjunto F que define as falhas a injetar, o conjunto A que define o domínio de ativação do sistema e especifica a funcionalidade do sistema que deve ser exercitada, o conjunto R que especifica as observações (*readouts*) a efetuar e quais as características do sistema que interessa observar e o conjunto M que especifica o modelo que se deve aplicar às observações do conjunto R de modo a obter um conjunto de medidas com significado prático [Arlat 89, Arlat90]. Mais recentemente o conjunto F é referido como *faultload,* o conjunto A como *workload* e o conjunto M como modelo de defeitos (*failure mode)*.

Um modelo de falhas inclui a informação a respeito das ações orientadas à reprodução de falhas de uma determinada classe (por exemplo, apagar um arquivo ou terminar um processo). A *faultload* é a especificação de um conjunto de falhas pertencentes a um determinado modelo de falhas e pode incluir regras de injeção, descrevendo parâmetros tais como: onde, quando e quantas vezes as falhas são injetadas [Christmansson 96]. A *workload* pode ser classificada como real, realística ou sintética. Uma *workload* real é composta pelo conjunto de tarefas reais submetidas pelos utilizadores do sistema. Uma *workload* realística é composta por programas que não são realmente utilizados mas que tentam abranger as tarefas mais representativas do sistema. *Workloads* sintéticas não representam as tarefas submetidas ao sistema pelos usuários, mas permitem ativar de forma controlada funcionalidades específicas do sistema.

Na Figura 8 pode-se visualizar os passos do processo de testes de sistema que devem ser executados durante o desenvolvimento DBC.

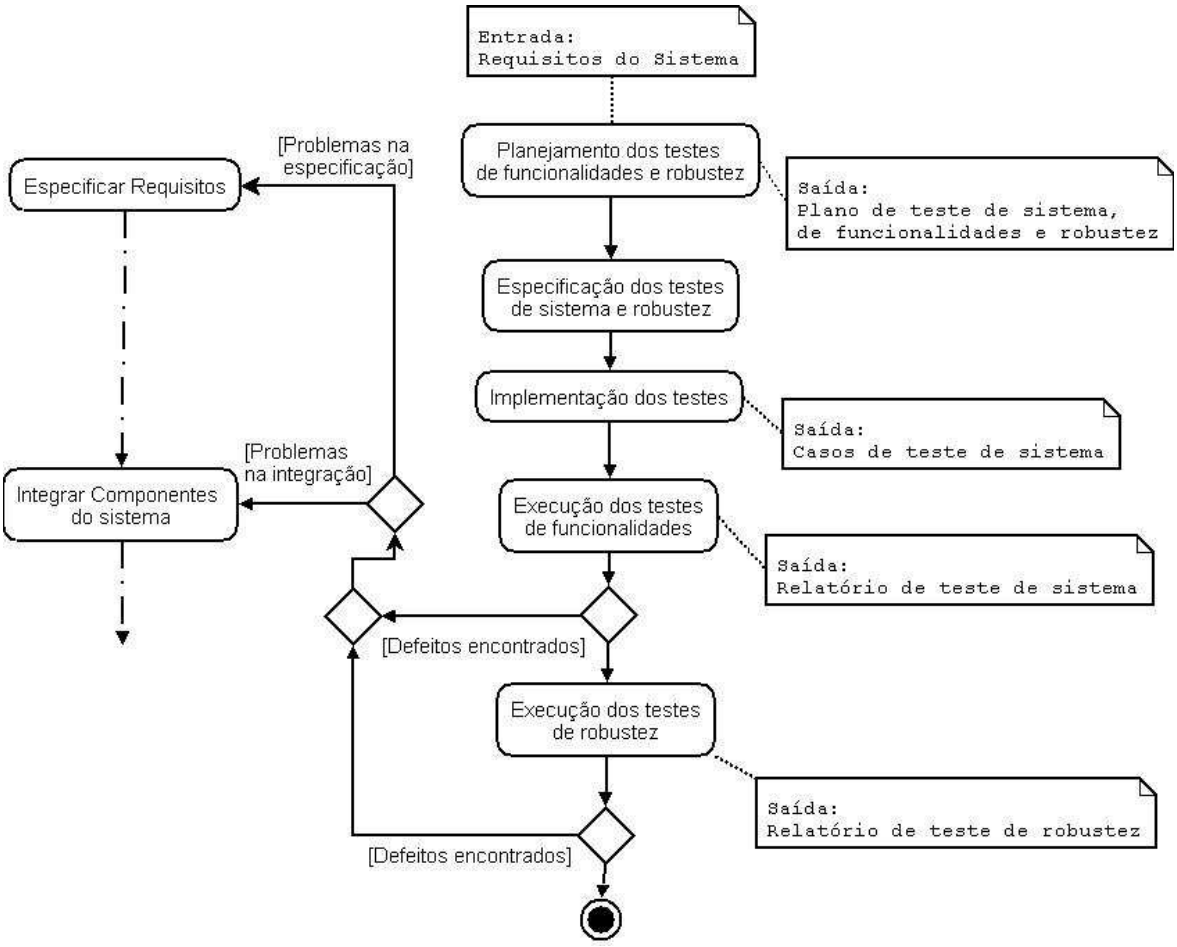


Figura 8 Atividades dos testes de sistema

De forma mais detalhada os passos do processo de teste de sistema são:

1. Planejamento: conforme citado na Seção 2.3, esta atividade consiste na elaboração do Plano de Teste de Sistema contendo, entre outros, a indicação de quais os objetivos dos testes e quais os itens (por exemplo, casos de uso) serão testados. Também nesta fase devem ser planejados os testes de robustez escolhendo o modelo de falhas e a *workload* mais apropriados para se validar o sistema sob teste. Também devem ser definidos o injetor e o monitor para se injetar as falhas e monitorar os resultados respectivamente.
2. Especificação dos testes de sistema: Nesta atividade são especificados os casos de teste para os itens a testar, definidos no Plano de Testes de Sistemas. Para os testes de robustez é definido a *faultload*. As especificidades e limitações do injetor devem ser levadas em consideração ao se definir a *faultload*. Uma vez definida a *faultload* também os casos de testes a serem utilizados nos testes de robustez são especificados. Caso seja necessário definir algum componente de testes (por exemplo, um simulador de algum dispositivo), este também deve ser especificado. Eventualmente, da mesma forma que no desenvolvimento do sistema, componentes de teste podem ser reutilizados.

3. Implementação dos testes: Os casos de teste, bem como os componentes especificados na atividade precedente, são implementados, criando-se assim os testes executáveis. Caso se reutilize componentes de testes, estes devem ser selecionados e adaptados.
4. Execução dos testes de funcionalidades: Deve-se verificar se as falhas foram geradas por problemas de especificação de requisitos ou no momento da integração do sistema. Na Figura 8 pode-se verificar estas duas possíveis saídas, na qual se busca solucionar problemas do sistema.
5. Execução dos testes de robustez: Uma vez exercitados todos os casos de testes especificados para os testes de funcionalidades, devem ser executados os casos de testes de robustez. Durante a aplicação dos testes de robustez, as falhas devem ser injetadas uma de cada vez e os resultados devem ser observados e classificados de acordo com o modelo de defeitos mais apropriado ao domínio do sistema. Normalmente o modelo de defeitos classifica os resultados observados de acordo com a sua severidade (impacto sobre o sistema). Deve-se verificar se a severidade dos defeitos é aceitável e se atende à especificação do sistema no que se refere à taxa de defeitos que se pode tolerar. Caso isso não ocorra, deve-se re-projetar a arquitetura do sistema acrescentando mecanismos de proteção para evitar a propagação de erros advindos do restante do sistema e que possam atingir o componente comprometendo o seu funcionamento correto. Deve-se também, proteger o sistema dos erros que podem ser propagados a partir do componente e que possam comprometer o funcionamento do sistema como um todo (por exemplo, acrescentando *wrappers* de proteção).

As atividades acima servem para quaisquer dos tipos de teste de sistemas mencionados anteriormente. É importante ressaltar que os artefatos produzidos nestas atividades devem ser também revisados, por um lado, para eliminar falhas ao desenvolvê-los, e por outro, para acompanhar as modificações ocorridas no sistema em desenvolvimento. Os componentes de teste que forem implementados ou reutilizados também devem ser testados antes de serem utilizados na Execução dos testes.

4.4 Testes de integração

Sistemas de software são constituídos por componentes que colaboram para a realização dos serviços prestados pelo sistema. Os testes de integração visam determinar a interoperabilidade entre esses componentes. O teste de integração é o processo de verificar se os componentes do sistema, juntos, trabalham conforme descrito nas especificações do sistema [Pfleeger04].

Na Figura 9 pode-se visualizar as fases do processo de testes de integração previstas para o desenvolvimento DBC.

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

Uma vez que um Planos de Testes de Integração é normalmente baseado nas relações de dependências entre componentes, a arquitetura do sistema é primordial. O processo de teste de integração freqüentemente revela erros, omissões e ambigüidades nos requisitos e na arquitetura.

A integração pode ser feita de forma incremental ou big-bang [Pressman97, cp.13]; nesta última, todos os componentes são integrados de uma única vez. A abordagem incremental é recomendada pois a localização e remoção de erros ou bugs fica mais fácil.

A abordagem incremental tem outras vantagens. As interfaces são sistematicamente testadas para que as falhas de interfaces sejam localizadas. Uma das dificuldades da integração incremental é a determinação da ordem de integração dos componentes.

Segundo [Binder99,cp.13], os componentes, tipicamente, dependem uns dos outros de diversas maneiras. As dependências são necessárias para implementar colaborações e conseguir a separação de alguns interesses. Algumas dependências são acidentais ou efeitos colaterais de uma

implementação, linguagem ou ambiente.
As dependências no escopo das classes e clusters resultam de alguns mecanismos como: composição e agregação, herança, variáveis globais, uso de objetos como parâmetros de mensagens entre outros.
De maneira semelhante, as dependências ocorrem entre componentes no escopo de um sistema. As dependências explícitas entre componentes correspondem às interfaces que o componente necessita para sua correta operação, ou seja, as interfaces requeridas.
O objetivo da análise de dependências, no contexto de Testes de Integração, é determinar a ordem de testes. Esta ordem indica que componentes devem ser testados primeiro de forma a reduzir, tanto quanto possível, o uso de stubs. Essa redução é importante para a viabilidade dos testes de integração, pois a construção de stubs pode demandar um grande esforço, e estes são dificilmente reutilizáveis.

Na fase de Preparação dos Testes de Integração é criado o Plano de Integração, o qual deve ocorrer logo após à especificação da arquitetura do sistema. Esse plano deve ser revisado após a fase de Implementação pois a arquitetura do sistema pode ter mudado, já que alguns componentes podem ter sido comprados, e também foram desenvolvidos conectores, adaptadores e outros componentes necessários.

Caso sejam encontrados defeitos no momento da integração do sistema deve-se verificar se os problemas de integração foram causados por violação da arquitetura especificada, caso em que a implementação dever ser corrigida. Erros também podem ocorrer devido à inconsistência na arquitetura do sistema, caso em que esta deve ser revisada e corrigida. Pode ainda, ocorrer erros nos testes devido a algum problema de integração, como ordem de que ocorreu a integração, ou ainda devido à má configuração do ambiente, como visto na Figura 9.

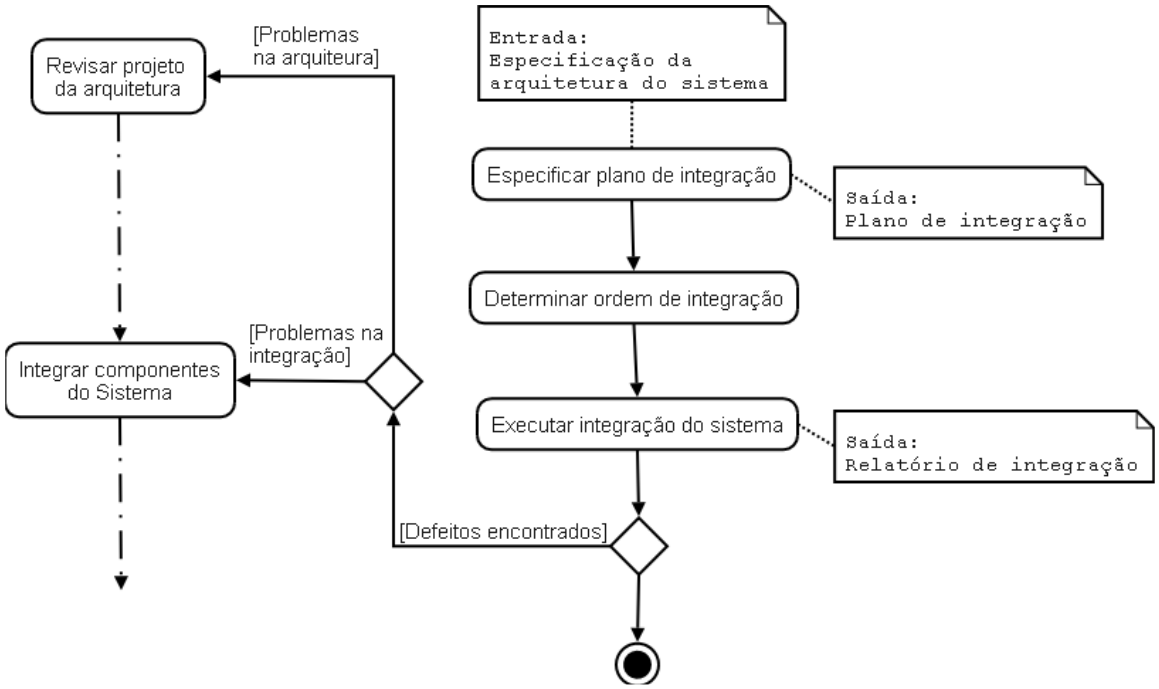


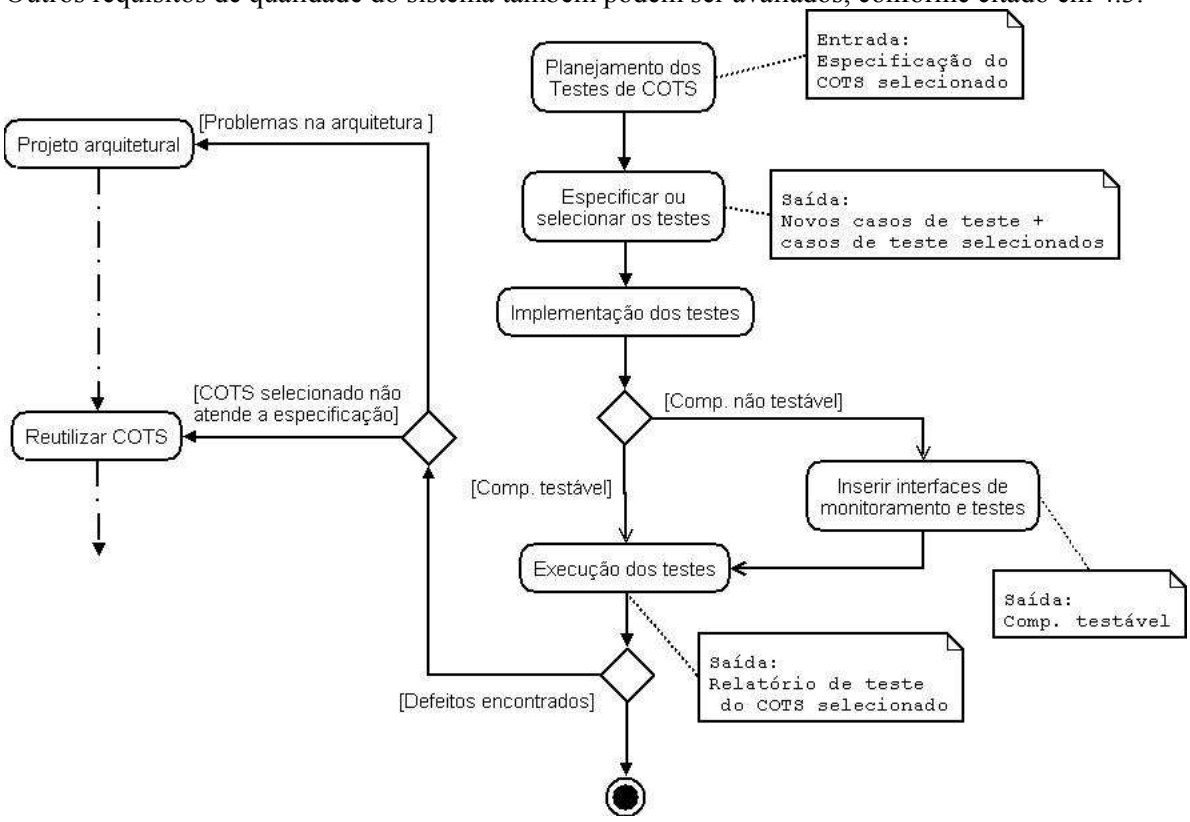
Figura 9 Fases do processo de testes que visam a integração do sistema

4.5 Testes de COTS

Esta fase é realizada ao se reutilizar componentes de terceiros, ou COTS. Os testes aplicados

nesta fase são geralmente de caixa preta, dado que os usuários dos COTS geralmente não têm acesso ao código fonte dos mesmos. O que se dispões, para esse fim, é da especificação da interface, o que pode incluir tanto aspectos funcionais quanto atributos de qualidade. Pode-se, nesta fase, reutilizar os testes criados pelos provedores do componente, caso estes existam. Os testes podem também ser definidos pelo usuário do COTS, a partir dos requisitos que estes componentes devem satisfazer dentro do sistema em desenvolvimento. Modelos podem ser desenvolvidos para representar o comportamento esperado do componente, ou podem ser utilizados os modelos existentes na especificação. A equipe de testes pode também tornar o componente testável, caso ele não o seja.

Além dos Testes Funcionais do componente isoladamente, os Testes de Robustez também podem ser aplicados nesse caso, para determinar o efeito que erros no resto do sistema teriam sobre o componente, e vice-versa. Dessa forma, nesta fase os testes de robustez são úteis para: (i) ajudar a selecionar os componentes mais adequados para o sistema em desenvolvimento; (ii) identificar o comportamento anormal de componentes e ajudar a validar os mecanismos de tratamento de erros; (iii) determinar quais componentes poderiam ser selecionados e, com a ajuda adaptadores (*wrappers*), poderiam prestar os serviços com a qualidade desejada para o sistema. Outros requisitos de qualidade do sistema também podem ser avaliados, conforme citado em 4.3.



Figuras 10 Fases do processo de testes para COTS selecionados

4.6 Testes de Componentes

Essa fase compreende os testes de componentes desenvolvidos pelo provedor, sejam eles específicos para o sistema, sejam eles genéricos e reutilizáveis. O objetivo é determinar se esses componentes atendem às regras de negócio especificadas. Caso seja utilizado algum modelo de componente (por exemplo, EJB), deve-se testar a compatibilidade do componente

desenvolvimento com esse modelo.

Cabe ressaltar que os Testes de Componentes, no contexto desse processo, não são Testes de Unidades. Estes visam testar as unidades que constituem cada componente (classes ou funções), e são considerados como responsabilidade do programador do componente. A equipe de testes considera o componente como uma caixa preta, e vai verificar se ele atende aos requisitos especificados.

Nesta fase deve-se considerar também a melhoria da testabilidade do componente. Cabe à equipe de desenvolvimento, juntamente com a de testes, definir quais componentes serão tornados testáveis. Para os componentes escolhidos, as bibliotecas de testes e de monitoração devem ser criadas, montando a arquitetura do componente testável para cada um dos componentes selecionados, conforme apresentado em 3.5. O modelo de comportamento do componente também deve ser criado.Da mesma forma que nas demais fases de testes, aqui também, antes da implementação e execução dos Testes de Componentes, deve-se verificar se não houve mudança na especificação do componente antes da sua implementação. A especificação e implementação devem considerar não somente os casos de teste, mas também as bibliotecas de testes e monitoração previstas em 3.5.

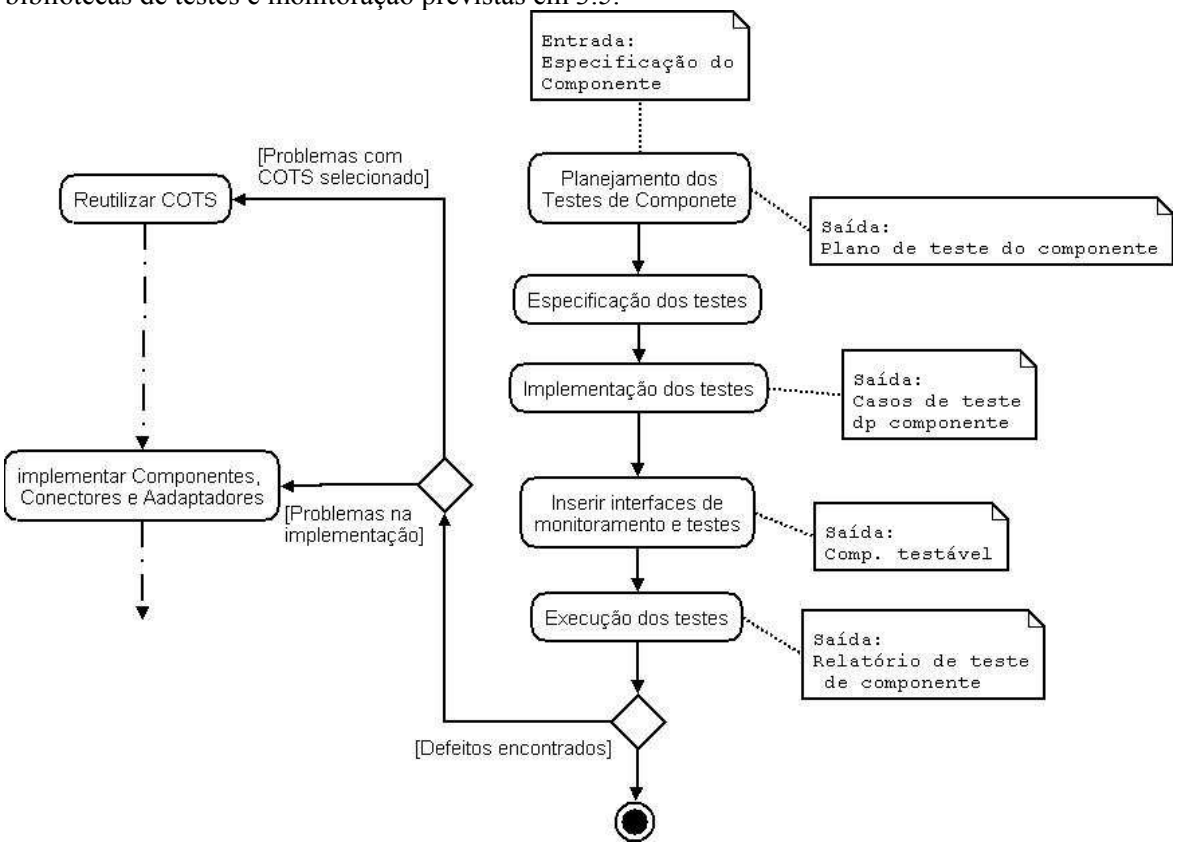


Figura 11 Fases do processo de testes de componentes para o processo DBC

4.7 Testes de Regressão

Quando um software sofre uma modificação, se a modificação causar efeitos colaterais em partes do software que funcionavam corretamente antes daquela modificação e que não deveriam ter sido afetadas pela mesma, diz-se que houve uma regressão. Os testes de regressão são executados no intuito de detectar se houve uma regressão no software [Binder94].

A principal diferença entre testes de regressão e testes de desenvolvimento é que no teste de regressão o conjunto de testes que foi aplicado à versão original já está disponível.

Existem basicamente duas estratégias de execução de testes de regressão: *retestar-tudo* e *seletiva* [RH94]. A estratégia *retestar-tudo* consiste simplesmente em reaplicar todo o conjunto de testes original à nova versão do software. A estratégia *seletiva* consiste em reaplicar apenas um subconjunto dos casos de testes originais. A estratégia *retestar-tudo* é desaconselhável, pois pode requerer um grande esforço e custo, devido ao grande volume de casos de testes.

Sejam P e P' as versões original e modificada do software, respectivamente, e T o conjunto de casos de testes aplicado à P, o processo de testes de regressão engloba os seguintes passos [RH96]:

1. Seleção de T' $\subseteq$ T, o conjunto de casos de testes a ser aplicado à nova versão do software P';
2. Executar conjunto de testes T' em P';
3. Se necessário, criar T'', conjunto com novos casos de testes para cobrir novas funcionalidades implementadas em P';
4. Executar conjunto de testes T'' em P';
5. Criar T''' a partir de T, T' e T''. T''' é uma nova linha de base (*baseline*) do conjunto de testes, compatível com P', a nova versão do software.

Para executar esses passos, técnicas de regressão de testes tratam quatro problemas diferentes. O passo 1 trata do problema de *seleção dos testes de regressão*: o problema de selecionar T' a partir de T. O passo 3 é o problema da *identificação de cobertura*: o problema de identificar quais partes de P' requerem mais testes. Os passos 2 e 4 envolvem o problema de *execução do conjunto de testes*: o problema de se executar os casos de teste eficientemente, e verificar os resultados com correção. O passo 5 é o problema da *manutenção do conjunto de testes*: atualizar o conjunto de casos de testes de acordo com a versão atual do *software* [RH96].

O processo de teste de regressão consiste de uma fase inicial, que ocorre em paralelo à implementação da modificação do *software*, quando o conjunto de testes a ser aplicado já pode ser selecionado; e de uma segunda fase, denominada "fase crítica" [RH96], que ocorre após o término da implementação. A execução dos casos de teste propriamente dita ocorre sempre durante a fase crítica. Já a seleção dos casos de testes pode ocorrer antes ou durante a fase crítica.

A seleção dos casos de testes pode ser realizada através de análise do código fonte (técnicas caixa branca), ou através da análise de mudanças aplicadas à especificação (técnicas caixa preta).

Nesse processo estamos propondo a aplicação da estratégia *seletiva* para execução de testes de regressão. A seleção dos casos de testes de regressão será feita através da análise de mudanças feitas no modelo de comportamento de um componente de software, de forma a minimizar o esforço durante a fase crítica.

No processo de DBC, os Testes de Regressão podem ser realizados a cada novo incremento que é desenvolvido e integrado ao sistema. Testes de Regressão são úteis também durante a manutenção do sistema.

4.8 Resumo

A Tabela 1 mostra resumidamente as fases de testes abordadas, bem como os artefatos usados e gerados em cada fase. Artefatos gerados como Plano de Testes, Especificação dos casos de teste e Relatórios de Teste, podem seguir padrões de documentos propostos pela norma IEEE 829 [IEEE87, Binder99].

O processo é inteiramente baseado nas especificações produzidas ao longo do desenvolvimento, desde a especificação do sistema à especificação do componente. Não se quer

com isso considerar os testes de caixa branca como menos importantes, mas no processo apresentado, estes ficam a cargo do programador, e não da equipe de testes.

Tabela 1 Artefatos de entrada e saída de cada fase de testes

Fase	Artefatos de entrada	Artefatos de saída
Sistema	Especificação dos requisitos do sistema.	Plano, Casos e Relatório de testes de sistema e robustez.
Integração	Especificação da arquitetura do sistema.	Plano de integração e Relatório de integração.
COTS	Especificação do COTS selecionado.	Novos casos de teste e casos de testes selecionados, relatório de testes.
Componentes	Especificação do componente.	Plano, Casos e Relatório de testes de componente.

5. Trabalhos correlatos

Nesta seção serão mostrados alguns trabalhos relacionados, quais sejam, propostas de processos para testes já existentes.

Uma primeira proposta para um processo de testes para desenvolvimento baseado em componentes é apresentada por A. Bertolino e A. Polini em [BP03]. Nesta proposta inicialmente é apresentada uma comparação de alguns pontos importantes ao processo de testes para desenvolvimento tradicional de software e o mesmo para o processo baseado em componentes, para em seguida apresentar um *framework* de testes.

Segundo Bertolino e Polini, no modelo tradicional de desenvolvimento de software os passos de teste previstos essencialmente são: Testes de Unidade, Testes de Integração e Testes de Sistema. Já em desenvolvimento baseado em componentes estas fases são estendidas e podem ser visualizadas e compostas de uma forma diferente: Testes de Componentes, Testes de deployment (entrega) e Testes de Sistema.

A fase de Testes de Componentes possui características comuns aos Testes de Unidade, que são a avaliação da cobertura de código do componente, onde o desenvolvedor do componente fica responsável por esta parte, obedecendo a um critério de cobertura de testes. Além disto, em Testes de Componentes, deve-se validar as funcionalidades do componente a procura de falhas em sua implementação e a validação da documentação do mesmo.

A fase de Testes de Entrega é a que mais se difere em relação aos testes em um processo de desenvolvimento tradicional. Um processo de desenvolvimento baseado em componentes, segundo os autores citados, possui minimamente três principais fases: a de identificar, buscar e validar os componentes. Na primeira fase deve-se identificar e especificar os componentes do sistema. Na segunda, deve-se buscar componentes candidatos a serem usados no desenvolvimento do sistema. Na terceira e ultima fase deve-se validar se a implementação do componente candidato selecionado atende às necessidades do sistema, sendo esta a fase de teste de entrega. Aqui os componentes candidatos são testados no ambiente do sistema, ou seja, juntamente com os demais componentes já selecionados para compor o sistema. Uma maneira de aproximar esta fase a fase de Testes de Integração do processo tradicional é efetuar a integração dos componentes no sistema de forma incremental, integrando o sistema ou alguns subsistemas de forma a minimizar o esforço de testes.

Os Testes de Sistema, tanto no modelo tradicional quanto no DBC possuem características similares, em que se verifica se a implementação de um dado sistema satisfaz a sua especificação.

Em [GTW03] é a apresentado outra abordagem para o processo de testes para componentes. Gao separa o processo de teste em dois tipos, o processo de testes para reuso e com reuso. Onde o teste do componente com reuso (de terceiros) é feito em seu novo ambiente de operação.

As diferenças entre eles é que no desenvolvimento com reuso não se possui conhecimento técnico, artefatos de desenvolvimento e código fonte deste componente. Por exemplo, testar componentes que serão reusados não prevê uma fase de testes caixa branca. Sendo assim em [GTW03, cap. 2] é apresentado o processo de testes para componentes de terceiros contendo 5 fases princiapis:

1. *Entrega do componente (deployment)*: onde o objetivo é verificar o se o componente pode ser inserido no novo contexto e ambiente operacional do projeto.
2. *Customização e adaptação do componente*: Aqui são identificadas as necessidades de adaptação para o uso do componente, tais como criação de conectores e adaptadores para tornar o uso do componente mais apropriado no novo contexto.
3. *Teste de usos do componente*: devem ser projetados casos de teste para verificar o uso do componente através de suas interfaces. Estes testes englobam atividades tais como: determinar a freqüência de chamadas de operações da interface do componente, e usar em sua interface dados padrões ou dados para o uso mais comum do componente.
4. *Validar componente*: aplicam-se testes caixa preta visando validar o comportamento e as funcionalidades do componente. Isto deve ser feito no ambiente de reuso, com o componente já integrado aos demais componentes do sistema. Esta fase é muito similar a fase testes de sistema do processo de testes tradicional.
5. *Avaliação de desempenho do componente*: Verificar se o componente irá satisfazer aos requisitos não funcionais do sistema neste novo ambiente operacional.

Para os componentes que necessitam customização ainda é previsto um outro processo de testes ligeiramente diferente, o qual contempla testes caixa branca para a nova porção de código agregada ao componente.

Considerando esta abordagem recursiva, testar sistema com reuso de componentes pode ser feito reaplicando este processo para cada novo componente agregado ao sistema. Assim, o processo é válido para diferentes níveis de granularidade dos componentes.

6. Conclusões e trabalhos futuros

Este relatório objetivou mostrar uma introdução a testes e sua importância em um projeto de desenvolvimento de sistemas. Neste caso, desenvolvimento baseado em componentes (DBC), onde foi apresentado um processo de desenvolvimento genérico para sistemas em DBC.

Apresentamos uma proposta de processo de testes a serem efetuados em paralelo a cada fase do processo DBC. Na fase de (i) especificação de requisitos deve-se especificar os testes de sistema; (ii) no projeto da arquitetura deve-se especificar os testes de integração; (iii) na seleção de COTS para reuso deve-se especificar e testar os COTS selecionados; (iv) paralelamente à implementação de componentes, conectores e adaptadores, deve-se revisar o plano de integração, especificar implementar e executar os testes de interação entre componentes (sequencialmente); (v) no momento da integração dos componentes deve-se executar os teste de integração; (vi) e finalmente para validar e corrigir o sistema deve-se executar os testes de sistemas.

Uma proposta de trabalho futuro é a instanciação do processo de testes para que se possa fazer uma avaliação do mesmo. Outra ação importante para essa proposta seria a automação do processo de teste, isso pode ser feito desenvolvendo um conjunto de ferramentas que apóiem as atividades de teste a serem realizadas nas diversas fases do processo proposto.

Pode-se também efetuar uma adaptação deste processo de teste DBC genérico para o processo *UMLComponents* [CD02]. Dado que o processo de desenvolvimento ao qual foi acoplado o processo de testes está baseado nessa metodologia, a adaptação a ela não será muito problemático.

7. Referências bibliográficas

[ABM00] Atkinson, Colin & Bayer, Joachim & Muthig , Dirk, "*Component-Based Product Line Development: The Kobr{A} Approach"*, P. Donohoe, pag. 289-309, http://citeseer.ist.psu.edu/atkinson00componentbased.html, 2000.

[Arlat 89] Arlat, J., Crouzet, Y., Laprie, J.C. "Fault Injection for Dependability Validation of Fault- Tolerant Computing Systems". In: Proceedings of the 19th IEEE International Symposium on Falt Tolerant Computing – FCTS'89, Chicago, Illinois, pp. 348-355, 1989.

[Arlat 90] Arlat, J., Aguera, M., Amat, L., Crouzet, Y., Fabre, J.C., Laprie, J.C., Martins, E., Powell, D. "Fault Injection for Dependability Validation: A Methodology and Some Applications". IEEE Transactions on Software Engineering,vol. 16, pp. 166-174, 1990.

[Bach 99] Bach, J.: Heuristic risk-based testing. Software Testing and Quality Engineering Magazine, (1999)

[Ball98] Ball, T., "*On the Limit of Control-Flow Analysis for Regression Test Selection*.", In International Symposium on Software Testing and Analysis (ISSTA), pp143-242, 1998.

[BB+01] Basanieri, F & Bertolino, A & Marchetti, E & Ribolini, A, *"An Automated Test Strategy Based on UML Diagrams"* Proc. Ericsson Rational User Conference, Upplands Vasby, 2001 isti.cnr.it

[BL02] Briand, L. & Labiche, Y., *"A UML-Based Approach to System Testing"*, Software and Systems Modeling, 2002, springerlink.com

[BL+02] Briand, L. & Labiche, Y. & Soccar, G., "*Automating Impact Analysis and Regression Test Selection Based on UML Designs*.", International Conference on Software Maintenance (ICSM'02), p.0252, Montreal, Quebec, Canada, Oct/2002

[BLS02] Briand, L. C. & Labiche, Y. & Sun, H., "*Investigating the use of analysis contracts to improve the testability of object-oriented code*", journal Software - Practice & Experience, vol. 33,
nº 7, pag. 637-672, John Wiley & Sons, Inc., http://dx.doi.org/10.1002/spe.520, 2003.

[Beizer97] Beizer, Boris, "*Sofware Testing Techniques*", International Thomson Computer Press, 1997.

[BP03] Bertolino, A. & Polini, A. "*A Framework for Component Deployment Testing*", International Conference on Software Engineering, 2003 - portal.acm.org

[BG03] Beydeda, Sami & Gruhn, Volker, "*State of the art in testing components*", 3rd International Conference on Quality Software, 2003.

[Bhor01] Bhor, Adrita, "*Software component testing strategies*", University of California, Irvine, June, 2001.

[Binder94] Binder, Robert V., "*Design for Testability in Object-Oriented Systems*.", journal Communications of the ACM, vol. 37, nº 9, pag. 87-101, DBLP, http://dblp.uni-trier.de, 1994.

[Binder99] Binder, Robert V., "*Testing object-oriented systems: models, patterns, and tools*", 0-201-80938-9, Addison-Wesley Longman Publishing Co., Inc., 1999.

[Brito05] Brito, Patrick H. da Silva et al, "*Um processo para o DBC com reuso de Componentes*", Relatório Técnico do Instituto de Computação da Univ. Estadual de Campinas. (em preparação), Junho, 2005.

[BW98] Brown, Alan W. & Wallnau, Kurt C., "*The Current State of CBSE*", journal IEEE Software, vol. 15, nº 5, pag. 37-46, IEEE Computer Society Press, 1998.

[CD02] Chessman, John & Daniels, John, "*UML Components: A Simple Process for Specifying Component-Based Software"*, Paperback, 2002.

[Chen02] Chen, Y., "*Specification-based Regression Testing Measurement with Risk Analysis*.", Mastering Thesis, University of Ottawa, Oct/2002.

[Corba] Object Management Group, "*The OMG's CORBA Website*", http://www.corba.org, 2004.

[De Milo94] De Millo, R. A., Li, T., Mathur, A. P.: Architecture of TAMER: A Tool for dependability analysis of distributed fault-tolerant systems. Purdue University, (1994)

[Christmansson 96] Christmansson, J., Chillarege, R. "Generation of an Error Set that Emulates Software Faults". In: Proceedings of the 26th International Fault-Tolerant Computing Symposium, FTCS-26, Sendai, Japan, pp. 304-313, 1996.

[DLS78] DeMillo, R.A.; Lipton, R.J.; Sayward, F.G. "*Hints on test data selection: Help for the Practicing Programmer*". IEEE Computer, 11(4), pp34-43, abr/1978.

[Drabick04] Drabick, Rodger D., "*Best Practices for the Formal Software Testing Process: A Menu of Testing Tasks*", Dorset House Publishing CO., Inc., 2004.

[EJB] Sun Microsystems Inc., "*Enterprise JavaBeans Technology"*, http://java.sun.com/products/ejb, 2004.

[FRA00] Fabre, J-C, Rodriguez, M., Arlat, J., Sizum, J-M.: Building dependable COTS microkernel-based systems using MAFALDA. In Proc. of 2000 Pacific Rim International Symposium on Dependable Computing - PRDC'00, Los Angeles, USA, (2000)

[Freedman91] Freedman, Roy S., "*Testability of Software Components."*, IEEE Transactions on Software Engineering, vol. 17, nº 6, pag. 553-564, 1991.

[GG+02], Gao, Jerry Z. & Gupta, Kamal K. & Gupta, Shalini & Shim, Simon S. Y., "*On Building Testable Software Components*", Proc. First International Conference on COTS-Based Software Systems, pag. 108-121, Fevereiro, 2002.

[GTW03] Gao, Jerry Z. & Tsao, H. S. J. & Wu, Ye "Testing and quality assurance for component-based software", Atech house, 2003.

[HJ+01] Harrold, M.J. & Jones, J.A. & Li, T. & Liang, D. & Orso, A. & Pennings, M. & Sinha, S. & Spoon, S.A. & Gujarathi, A., "*Regression Test Selection for Java Software*.", In ACM Conf. on OO Programming, Systems, Languages and Applications (OOPSLA), 2001.
[Hoffman89] Hoffman, Daniel, "*Hardware testing and software Ics*", Proc. Pacific Northwest Software Quality Conference, Portland, Oregon, pag. 234-244, 1998.

[Hopkins00] Hopkins, Jon; "*Component primer*", journal Communications of the ACM, vol. 43, nº10, pag, 27-30, ACM Press, 2000.

[HT+97] Hsueh, Mei-Chen & Tsai, Timothy & Iyer, Ravishankar, "*Fault Injection Techniques and Tools*", IEEE Computer, 75-82 , 1997.

[IEEE87] "*ANSI/IEEE Standard 829-1983: IEEE Standard for Software Test Documentation*", The Institue of Electrical and Electronic Engineers, 1987.

[KSD+97] Koopman, P., Sung, J., Dingman, C., Siewiorek, D., Marz, T.: Comparing Operating Systems using Robustness Benchmarks, In Proc of SRDS97, Carnegie Mellow University, (1997).

[KS+03] Koopman, P. & Siewiorek, D & DeVale, K. & DeVale, J. & Fernsler, K. & Guttendorf, D. & Kropp, N. & Pan, J. & Shelton, C. & Shi, Y, "*COTS Software Robustness Testing. Carnegie Mellon University*", Ballista Project, http://www.ece.cmu.edu/~koopman/ballista, 2003.

[LL87] Leite, Julius C.B.; Loques Fº, Orlando G. Introdução à Tolerância a Falhas. Mini-curso apresentado durante o II Simpósio de Software Tolerante a Falhas (SCTF), Campinas, SP, 1987.

[MM05] Moraes, R., Martins, E. "Fault Injection Approach based on Architectural Dependencies", In Architecting Dependable Systems III – ADSIII, UK (Book chapter), Rogerio de Lemos, Cristina Gacek, A. Romanovsky Editors, Springer Verlag, Berlin Heidelberg, UK, 2005.

[MV05] Martins, E. & Vieira, V.G., "*Regression Test Selection for testable classes*.", Fifth European Dependable Computing Conference, Budapest, Hungary, Apr/2005.

[Meyer97] Meyer, Bertrand, "*Object-Oriented Software Construction*", Prentice Hall, 1997.

[MRL00] Martins, E., Rubira, C., Leme, N.: Jaca: A reflective fault injection tool based on patterns. In: Proc of the 2002 International Conference on Dependable Systems & Networks, Washington D.C. USA, pp. 483-487, (2002).

[MS01] McGregor, John & Sykes, David A. "A Practical Guide to Testing Object-Oriented Software". Addison-Wesley, março, 2001.

[Myers79] Myers, Glenford, "*The Art of Software Testing*", John Wiley and Sons, 1979.

[Paula01] Paula Filho, Wilson de P., "Engenharia de Software: Fundamentos, Métodos e

Padrões", Livros Técnicos e Científicos, 2001.

[Perry95] Perry, W., "*Effective methods for software testing*", John Wiley & Sons Inc., cap. 2, 3, 1995.

[Pfleeger04] Pfleeger, S. L., "*Engenharia de Software: Teoria e Prática*",Prentice Hall, 2a. Edição, 2004.

[Pressman97] Pressman, Roger S., "*Software Engineering"*, McGraw-Hill, 1997.

[Pressman01] Pressman, Roger S., "*Software Engineering: a Practitioner's Approach*", McGraw-Hill, 2001.

[Rocha05] Rocha, Camila R. "Um Método de Testes para Componentes Tolerantes a Falhas." Campinas: Instituto de Computação da UNICAMP, 2005. 153 p. (Dissertação, Mestrado em Ciência da Computação)

[RH94] Rothermel, G. & Harrold, M.J., "*Selecting regression tests for object-oriented software*", IEEE Proceedings of International Conference on Software Maintenance, pp14-25, Sep/1994.

[RH96] Rothermel, G. & Harrold, M.J., "*Analyzing Regression Test Selection Techniques*", IEEE Transactions on Software Engineering, 22(8), 1996.

[RM04] Rocha, Camila R. & Martins, Eliane "A Strategy to Improve Component Testability without source code". Proc. Net.ObjectDays Workshops on Testing of Component-Based Systems (TECOS 2004) and Software Quality (SOQUA 2004). Erfurt, Germany, pp47-62, Setembro, 2004.

[Rosenberg 00] Rosenberg, L, Stapko, R, Gallo, A.: Risk-based Object Oriented Testing. In: Proc. of 13th International Software / Internet Quality Week (QW2000), San Francisco, California USA, (2000)

[Stafford 97] Stafford, J.A., Richardson, D.J., Wolf, A.L.: Chainning: A Software Architecture Dependence Analysis Technique. Technical Report CU-CS845-97, Department of Computer Science, University of Colorado, (1997)

[Szyperski98] Szyperski, Clemens, "*Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, 1998.

[Toyota00] Toyota, Cristina M., "*Melhoria da Testabilidade de classes usando o conceito de autoteste*", *IC, Unicamp*, 120 pags., Junho, 2000.

[UML03] Object Management Group, "*OMG Unified Modeling Language Specification Version 1.5*", OMG document formal/03-03-01, 2003.

[UML04] Object Management Group, "*UML 2.0 Superstructure Final Adopted Specification*", OMG document ptc/03-08-02, 2004.

[Vitharana03] Vitharana, Padmal, "*Risks and challenges of component-based software development*", journal Communications of the ACM, vol. 46, nº 8, Pag. 67-72, ACM Press , 2003.

[VM95] Voas, Jeffrey M. & Miller, Keith W., "*Software Testability: The New Verification"*, journal IEEE Software,vol. 12, nº 3, pag. 17-28, http://citeseer.lcs.mit.edu/voas95software.html, 1995.

[Voas98a] Voas, Jeffrey, "*Software Fault Injection: Inoculating Programs Against Errors"*, Wiley Computer Publishing, 1998.

[Voas98b] Voas, Jeffrey M., "*Certifying Off-the-Shelf Software Components*.", journal IEEE Computer, vol. 31, nº 6, pag. 53-59, DBLP, http://dblp.uni-trier.de, 1998.

[WK03] Warmer, Jos & Kleppe, Anneke, "*The Object Constraint Language: Getting your Models Ready for MDA*", Addison-Wesley, 2003.

[Weyuker98] Elaine J. Weyuker. Testing Component-Based Software: A Cautionary Tale. *IEEE Software*, Vol. 15, Nº 5, September/October 1998