5 – Modelos de Sistemas

Define-se “análise” como o processo de decompor o todo em suas partes componentes, examinando estas partes, conhecendo sua natureza, funções e relações.

A tarefa de construir sistemas é bastante complexa, configurando-se, na realidade em um processo de solução de problemas. Neste sentido, dizemos que a análise de sistemas consiste nos métodos e técnicas de avaliação e especificação da solução de problemas, para implementação em algum meio que a suporte, utilizando mecanismos apropriados.

Segundo Pressman todos os métodos de análise devem ser capaz suportar 5 atividades:

- Representar e entender o domínio da informação
- Definir as funções que o software deve executar
- Representar o comportamento do software em função dos eventos externos
- Particionar os modelos de informação, função e comportamento de maneira a apresentar os detalhes de forma hierárquica
- Prover a informação essencial em direção a determinação dos detalhes de implementação.

Para que a análise seja bem feita devemos não só entender o que vamos fazer, mas também desenvolver uma representação que permita que outros entendam o que é necessário para que o sistema que está sendo desenvolvido atinja sua finalidade.

A grande maioria dos autores advoga que não devemos levar em conta a tecnologia que empregaremos durante a análise de sistemas. A análise deve se preocupar com o “o que fazer” e nunca com o “como fazer”. Para isso, fazemos a modelagem do sistemas, utilizando abstrações.

5.1 – Modelagem

Como visto, um modelo é uma representação abstrata de algo real. Ele tem o objetivo de imitar a realidade, para possibilitar que esta seja estudada quanto ao seu comportamento. Os modelos permitem focalizar a atenção nas características importantes do sistema, dando menos atenção às coisas menos importantes. Seu uso torna o estudo mais barato e seguro: é muito mais rápido e barato construir um modelo do que construir a coisa “real”.

O objetivo do modelo é mostrar como será o sistema, para permitir sua inspeção e verificação, a fim de poder receber alterações e adaptações antes de ficar pronto. Qualquer modelo

realça certos aspectos do que está sendo modelado, em detrimento dos outros.

A ferramenta utilizada para modelar influi diretamente na forma como pensamos sobre a realidade e determina quais aspectos serão mostrados e quais ficarão escondidos.

5.2 – O papel do analista

O Analista de Sistemas assume o papel de elo entre os usuários e o computador. Ele deverá entender e avaliar as necessidades e expectativas de cada usuário, a fim de que estas sejam organizadas e especificadas seguindo uma formalidade técnica.

O analista deve ser capaz de lidar, ao mesmo tempo, com um grupo de usuários, outros profissionais de informática e um corpo administrativo (gerentes/diretores), cada qual trazendo formações, pontos de vistas, vivências, experiências e maturidade totalmente distintas.

Os usuários, ou estarão preocupados em dinamizar seu serviço, tornando-o automático e extremamente rápido, aumentando a confiabilidade de resultados, ou ainda, estarão com medo da informatização, às vezes, até obstruindo o trabalho do Analista de Sistemas. O pessoal técnico estará se preocupando com aspectos de performance, estruturas de dados, topologia de hardware e diversidade de recursos. Por fim, na administração, muitas vezes estão aqueles que só querem saber do retorno sobre o investimento e a proporção custo/benefício.

Por estes motivos, Yourdon alerta que a capacidade do analista não deve estar restrita a fazer diagramas e modelos. Ele deve ter habilidade com as pessoas, conhecimento do negócio da empresa e domínio da tecnologia.

A maior parte do trabalho de análise é feita da comunicação entre pessoas. Muita dessa comunicação é feita na linguagem natural das pessoas, como o português. Porém, línguas como o Português e o Inglês permitem a construção de sentenças ambíguas. Como no desenvolvimento de sistemas temos que evitar ao máximo as ambiguidades, temos que restringir a linguagem utilizada de forma que uma sentença só tenha uma interpretação possível (ou mais provável).

Muitos programadores argumentam que podem desenvolver sistemas sem “perder” tempo com Análises, Projetos e diagramas. Isto só é válido para sistemas pequenos (barracos não precisam de planta). Em sistemas grandes (prédios) deve-se planejar a construção.

No desenvolvimento tradicional (sem metodologia), não se usam técnicas padronizadas e nem representações gráficas, não se separa requerimentos funcionais da implementação técnica, e ainda, os sistemas não são descritos por textos em linguagem natural e diagramas.

5.3 – Paradigmas de Modelagem

5.3.1 – A Solução Estrutrada

A primeira forma de desenvolvimento de software que empregava uma metodologia baseava-se em soluções Estruturadas, como a Análise Estruturada, o Projeto Estruturado, e a Análise Essencial. Esta solução possui as seguintes características:

- Representação gráfica dos requerimentos dos sistemas;
- Preocupação com os fluxos de dados; e
- Prega a separação entre modelo lógico e físico do sistema.

Limitações desta solução:

- Centrados em processos;
- Pouca preocupação com dados;
- Voltados para grandes sistemas batch/mainframe; e
- Altamente burocráticas e consumidoras de tempo.

5.3.2 – A Orientação a Objetos

A *Modelagem e projetos baseados em objetos* é uma forma de estudar (e representar) problemas com utilização de modelos fundamentais em conceitos do mundo real.

A estrutura básica (e fundamental, claro), onde estão fundamentados todos os conceitos, é o *objeto*, que combina a estrutura e o comportamento dos dados em uma única entidade.

Ela se difere das técnicas estruturadas pois:

- Tende a aumentar a produtividade de desenvolvimento através da reutilização de produtos de projetos anteriores (desde que corretamente aplicada);
    - Reusabilidade
    - "montadora" de software
- Reduz drasticamente a complexidade e o esforço de manutenção;
- Produz sistemas mais resistentes a manutenções;
- Permite a construção de sistemas de complexidade arbitrária;

- Está conceitualmente mais de acordo com outras tecnologias emergentes, como Interfaces Gráficas e sistemas Cliente-Servidor; e
- Os modelos utilizados são os mesmos em todas as fases do desenvolvimento (*Traceability*).

Os conceitos da OO podem ser aplicados em diferentes fases do projeto, como Análise, Projeto e Implementação. O objetivo da Análise Orientada a Objetos (AOO ou OAA) é desenvolver uma série de modelos que descrevem como o software irá se "comportar" para satisfazer seus requisitos. Logo, a preocupação maior desta fase é representar "**o que**" o sistema irá fazer, sem considerar "como".

Como na Análise Estruturada, constroem-se modelos que representam diversas perspectivas, descrevendo o fluxo de informações, as funções e o comportamento do sistema dentro do contexto dos elementos do sistema.

No entanto, diferentemente da Análise Estruturada, aplica-se aqui os conceitos do paradigma da Orientação a Objetos para realizar-se o estudo do sistema, buscando-se um desenvolvimento mais adequado.

Seguindo os modelos do processo de software, a Análise Orientada a Objetos é realizada a partir dos documentos de Especificação de Requisitos do software, os quais descrevem, principalmente, as funcionalidades esperadas do sistema.

Nesta fase produz-se um conjunto de diagramas que descrevem as características do software. Deve-se ressaltar que o emprego dos diagramas difere de acordo com as características do sistema. Pode-se ainda desenvolver um Dicionário de Dados descrevendo os componentes dos diagramas.

Ao final desta fase, se possível, deve-se revisar os documentos juntamente com representantes do cliente.

Os diagramas produzidos nesta fase são modelos gráficos representativos do sistema.

- Estrutura Estática;
- Estrutura Dinâmica;
- Comportamento.

Os diagramas gerados podem (e devem) ser empregados para a prototipação do sistema, podendo-se avaliar se ele cumpre realmente seus requisitos. Na Análise Orientada a Objetos, pode-se empregar diferentes diagramas da UML, de acordo com a necessidade do projeto.

Os diferentes diagramas permitem ao Analista observar o software sob diferentes aspectos – **Visões.** O emprego de cada diagrama é determinado de acordo com as características do sistema.

5.4 – Diferença de Enfoques

Apresenta-se a seguir uma comparação entre as duas formas de desenvolvimento:

Desenvolvimento Estruturado:	Desenvolvimento Orientado a Objetos:
■ Fluxo de dados	■Objetos (Classes)
■ Transformações	■ Atributos
■ Repositórios de dados	■ Operações
■ Entidades	■ Associações e Multiplicidade
■ Especificação de procedimentos	■ Herança
■ Dicionário de dados	■ Outros componentes especializados
Foco nos processamentos do sistema.	**Foco nos componentes do sistema.**

É importante observar o foco de cada desenvolvimento. No Desenvolvimento Estruturado, o foco é o processamento dos sistemas, já no OO, os componentes do sistema são priorizados.

Na Orientação a Objetos o software é organizado como uma coleção de objetos separados que incorporam tanto a **estrutura** quanto o **comportamento** dos componentes do sistema.

Isto é diferente da programação convencional, onde a estrutura e o comportamento dos dados têm poucos vínculos entre si. No próximo tópico, são apresentados os conceitos básicos da Orientação a Objetos.

6 – Teste de Software

A atividade de teste de software é um elemento crítico da garantia de qualidade de software e representa a última revisão de especificação, projeto e codificação.

A realização dos testes é importante devido aos custos de associados à correção de falhas, podendo despender até 40% do esforço de desenvolvimento. Em sistema críticos os testes podem custar até 5 vezes mais que todas as outras fases juntas.

Esta atividade constitui uma anomalia interessante para o engenheiro de software. Durante as fases de definição e desenvolvimento anteriores, o engenheiro tenta construir o software, partindo de um conceito abstrato para uma implementação tangível. Agora, surge a fase de testes, e o engenheiro cria uma série de casos de teste que têm a intenção de "demolir" o software que ele construiu. De fato, a atividade de teste é um passo do processo de engenharia de software que poderia ser visto (pelo menos no ponto de vista psicológico) como destrutivo, em vez de construtivo.

A atividade de teste deveria promover culpa? Ela é realmente destrutiva? A resposta é definitivamente **não**. Porém, os objetivos da atividade de teste são bem diferentes do que poderíamos esperar.

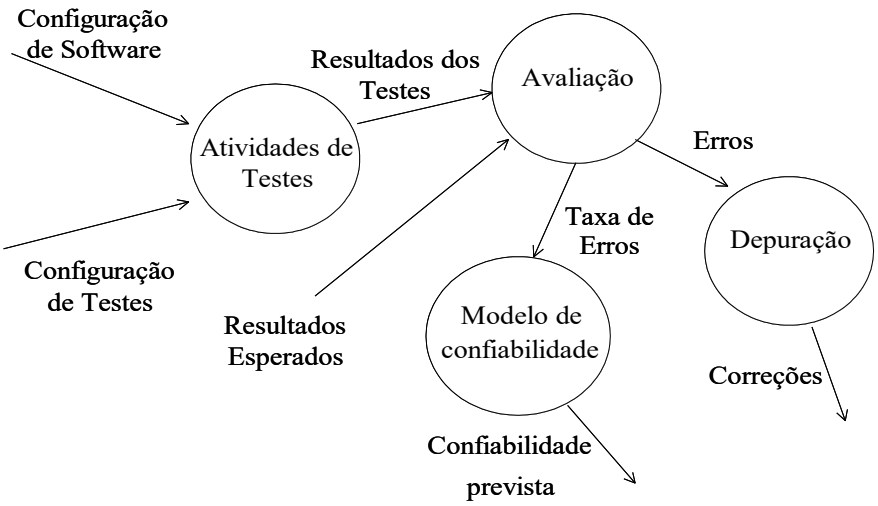
As seguintes regras podem servir como objetivos de teste:

- A atividade de teste é o processo de executar um programa com a intenção de descobrir erros;
- Um bom caso de teste é aquele que tem uma elevada probabilidade de revelar um erro ainda não descoberto; e
- Um teste bem-sucedido é aquele que revela um erro ainda não descoberto ou inesperado.

Os objetivos apontados acima apresentam uma visão diferente daquela que se costuma possuir com relação à atividade de testes, que seria a de que um teste bem-sucedido não apontaria erros. O objetivo na verdade é projetar testes que encontrem sistematicamente diferentes classes de erros e façam-no com uma quantidade de tempo e esforço mínimos.

Se a atividade de testes for conduzida com sucesso, ela descobrirá erros no software. Como um benefício secundário, a atividade de teste demonstra que as funções de software aparentemente estão trabalhando de acordo com as especificações, que os requisitos de desempenho aparentemente foram cumpridos. Além disso, os dados compilados quando a atividade de testes é levada a efeito, proporcionam uma boa indicação da confiabilidade de software e alguma indicação da qualidade do software como um todo.

A realização dos testes de software segue o fluxo de informações apresentado na próxima figura.



Na figura pode-se observar que duas classes de entrada são fornecidas ao processo de teste:

- Configuração de Software (Especificação de requisitos, o Projeto e o Código-fonte); e
- Configuração de teste (Casos de teste)

À medida que os resultados de teste são reunidos e avaliados e avaliados, obtém-se uma indicação da qualidade do produto.

Se nenhum erro for encontrado, pode ser que os testes sejam inadequados. Uma importante característica da atividade de teste, que deve ser lembrada no momento de sua realização: **A atividade de teste não pode garantir a ausência de erros.**

6.1 – Verificação e Validação

O teste de software é um elemento de um aspecto mais amplo, que é frequentemente referido como *verificação e validação* (V&V). *Verificação* se refere ao conjunto de atividades que garante que o software implementa corretamente uma função específica. *Validação* se refere a um conjunto de atividades diferente, que garante que o software construído corresponde aos requisitos do cliente.

Verificação: “estamos construindo o produto corretamente?”
Validação: “estamos construindo o produto certo?”

6.2 – Projeto de Casos de Teste

Os Casos de Teste compõem a Configuração de Testes (veja a figura anterior), devendo retratar situações da realidade/casuais (utilização prevista do sistema), visando identificar falhas existentes no software.

Deve-se projetar testes que tenham alta probabilidade de descobrir a maioria dos erros com uma quantidade mínima de tempo e esforço. Para tanto, emprega-se de técnicas de teste.

Os engenheiros de software geralmente tratam a atividade de testes como uma reflexão tardia, desenvolvendo casos de teste que parecem estar corretos, mas que apresentam pouca garantia de estar completos. Relembrando os objetivos da atividade de teste, os testes devem ser projetados para descobrir a maioria de erros com uma quantidade mínima de tempo e esforço.

6.3 – Técnicas de Testes

Qualquer produto pode ser testado de duas maneiras:

- Conhecendo-se a função específica que um produto deve executar, testes podem ser realizados para demonstrar que cada função é totalmente operacional;
- Conhecendo-se o funcionamento interno de um produto, testes podem ser realizados para garantir que "todas as engrenagens se encaixam", ou seja, que a operação interna do produto tem um desempenho de acordo com as especificações e que os componentes internam foram adequadamente postos à prova.

A primeira abordagem é chamada *teste de caixa preta* e a segunda, *teste de caixa branca*.

Considerando-se a softwares de computador, a expressão *teste de caixa preta* refere-se aos testes que são realizados nas interfaces do software. Não obstante eles sejam projetados com o propósito de descobrir erros, os testes de caixa preta são usados para demonstrar que as funções do software são operacionais; que a entrada é adequadamente aceita e a saída é corretamente produzida; que a integridade das informações externas (como arquivos de dados) é mantida. Um teste de caixa preta examina alguns aspectos de um sistema sem se preocupar muito com a estrutura lógica interna do software.

O *teste de caixa branca* baseia-se num minucioso exame dos detalhes procedimentais. Os caminhos lógicos através do software são testados, fornecendo-se casos de teste que põem à prova conjuntos específicos de condições e/ou laços.

Numa primeira observação, poderia parecer que um teste de caixa branca efetuado muito cuidadosamente levaria a cem por cento de programas corretos. Tudo que seria necessário é definir

todos os caminhos lógicos, desenvolver os casos de teste para pô-los à prova e avaliar os resultados. Porém, testes exaustivos apresentam determinados problemas, mesmo para pequenos programas, o número de caminhos lógicos possíveis pode ser muito grande. Por exemplo, um programa poderia ser constituído de aproximadamente 100 linhas, com um único laço que pode ser executado no máximo 20 vezes, e algumas instruções condicionais, apresenta aproximadamente 10^{14} caminhos que podem ser executados. Para colocar este número em uma perspectiva, suponha um processador de testes que poderia processar testes exaustivos, avaliando cada caso de teste em um milisegundo. Trabalhando 24 horas por dia, 365 dias ao ano, o tempo necessário seria de 3170 anos para testar o programa em sua totalidade.

Entretanto, o teste de caixa branca não deve ser descartado. Um número limitado de caminhos lógicos importantes pode ser selecionado e executado. Estruturas de dados importantes podem ser investigadas quanto á validade. Os atributos, tanto do teste de caixa branca quanto o de caixa preta, podem ser combinados para oferecer uma abordagem que valide a interface com o software e garanta seletivamente que o funcionamento interno do software esteja correto.

Usando métodos de teste de caixa branca, o engenheiro pode derivar casos de teste que:

- Garantam que todos os caminhos independentes dentro de um módulo tenham sido exercitados pelo menos uma vez;
- Exercitem todas as decisões lógicas para valores falsos ou verdadeiros;
- Executem todos os laços em suas fronteiras e dentro de seus limites operacionais;
- Exercitem as estruturas de dados internas para garantir a sua validade.

Os métodos de teste de caixa preta concentram-se nos requisitos funcionais do software. Ou seja, esse teste possibilita que o engenheiro de software derive conjuntos de condições de entrada que exercitem completamente todos os requisitos funcionais para um programa. O teste de caixa preta não é uma alternativa para as técnicas de caixa branca. Ao contrário, trata-se de uma abordagem complementar que tem a probabilidade de descobrir uma classe de erros diferente daquela dos métodos de caixa branca.

O teste de caixa preta procura descobrir erros nas seguintes categorias:

- Funções incorretas ou ausentes;
- Erros de interface;
- Erros nas estruturas de dados ou no acesso a bancos de dados externos;
- Erros de desempenho;
- Erros de inicialização e término.

Ao contrário do teste de caixa branca, que é executado cedo no processo de teste, o teste de caixa preta tende a ser aplicado durante as últimas etapas da atividade de teste. Uma vez que o teste de caixa preta deliberadamente desconsidera a estrutura de controle, a atenção se concentra no domínio de informação. Testes são projetados para responder às seguintes perguntas:

- Como a validade funcional é testada?
- Quais classes de entrada constituirão bons casos de teste?
- O sistema é particularmente sensível a certos valores de entrada?
- Como as fronteiras de uma classe de dados são isoladas?
- Quais índices de dados, e volumes de dados, o sistema pode tolerar?
- Que efeitos terão combinações específicas de dados sobre a operação do sistema?

6.4 – Estratégias de Teste de Software

Uma estratégia de teste de software integra métodos de projeto de casos de teste (abordados anteriormente) numa série bem-planejada de passos, que resultam na construção de software bem-sucedida. A estratégia fornece um roteiro que descreve os passos a serem conduzidos como parte do teste, quando esses passos são planejados e depois executados e quanto de esforço, tempo e recursos serão necessários. Assim, qualquer estratégia de teste deve incorporar planejamento de teste, projeto de casos de teste, execução de teste e a resultante coleta e avaliação de dados.

Uma estratégia de testes de software deve ser suficientemente flexível para promover uma abordagem de teste sob medida. Ao mesmo tempo, deve ser suficientemente rígida para promover planejamento razoável e acompanhamento gerencial, à medida que o projeto progride.

6.4.1 – Uma Abordagem Estratégica ao Teste de Software

O teste trata-se de um conjunto de atividades que podem ser planejadas antecipadamente e conduzidas sistematicamente. Por essa razão, um gabarito para teste de software - um conjunto de passos no qual podemos colocar técnicas de projeto de casos de teste e métodos de teste específicos - pode ser definido para o processo de software.

Algumas estratégias de teste de software têm sido propostas na literatura. Todas fornecem ao desenvolvedor de software um gabarito de teste com características genéricas:

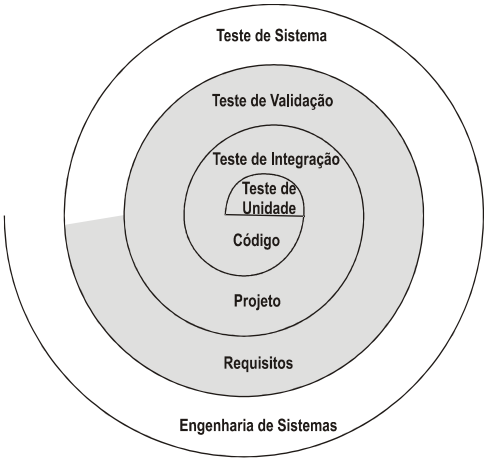
- O teste começa em nível de componente e prossegue “para fora”, em direção à integração de todo o sistema baseado em computador;

- Diferentes técnicas de testes são adequadas em diferentes momentos;
- O teste é conduzido pelo desenvolvedor do software e (para projetos grandes) um grupo de teste independente; e
- O teste e a depuração são atividades diferentes, mas a depuração deve ser prevista em qualquer estratégia de teste.

Uma estratégia de teste de software deve acomodar testes de baixo nível, que são necessários para verificar se um pequeno segmento de código-fonte foi corretamente implementado, bem como testes de alto nível, que validam as principais funções do sistema, com base nos requisitos do cliente. Uma estratégia deve fornecer diretrizes para o profissional e um conjunto de referenciais para o gerente. Como os passos da estratégia de teste ocorrem quando a pressão do prazo de entrega começa a crescer, o progresso deve ser mensurável e os problemas devem aparecer tão cedo quanto possível.

6.4.2 – Uma Estratégia de Teste de Software

O processo de engenharia de software pode ser visto como a espiral ilustrada na figura a seguir.



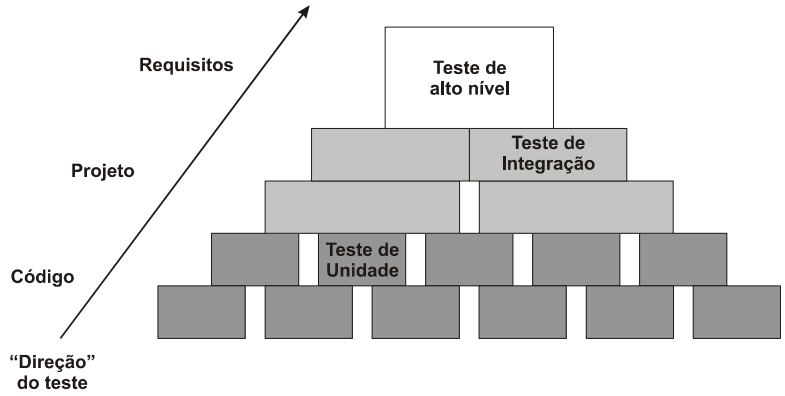
Inicialmente, a *engenharia de sistemas* define o papel do software e leva à análise dos *requisitos* de software, em que são estabelecidos o domínio da informação, a função, o comportamento, o desempenho, as restrições e os critérios de validação para o software. Movendo-se para dentro, ao longo da espiral, chegamos ao *projeto* e finalmente à *codificação*.

Para desenvolver software de computador, nos movemos em espiral para dentro, ao longo de voltas que diminuem o nível de abstração em cada volta.

Uma estratégia para teste de software pode também ser vista no contexto da espiral (figura anterior). O *teste de unidade* começa no centro da espiral e se concentra em cada unidade (componente) do software como implementada no código-fonte. O teste progride movendo-se para fora ao longo da espiral para o *teste de integração*, em que o foco fica no projeto e na construção da arquitetura do software. Dando outra volta para fora, pela espiral, encontramos o *teste de validação*, em que os requisitos estabelecidos como parte da análise dos requisitos do software são validados em contraste com o software que acabou de ser construído. Finalmente, chegamos ao *teste do sistema*, em que o software e os outros elementos do sistema são testados como um todo.

Para testar o software de computador, nos movemos pela espiral para fora ao longo de voltas que ampliam o escopo do teste a cada volta.

Considerando o processo de um ponto de vista procedimental, o teste no contexto da engenharia de software é na realidade uma série de passos, que são implementados seqüencialmente. Os passos são mostrados na figura a seguir.



Inicialmente, o teste focaliza cada componente individualmente, garantindo que ele funciona adequadamente como uma unidade. Daí o nome de *teste de unidade*. O teste de unidade faz uso das técnicas de teste caixa-branca intensamente, exercitando caminhos específicos na estrutura de controle de um módulo, para garantir completa cobertura e máxima detecção de erros. Em seguida, os componentes devem ser montados ou integrados para formar o pacote de software completo.

O *teste de integração* cuida dos aspectos associados com os problemas duais de verificação

e construção de programas. As técnicas de projeto de casos de teste de caixa-preta são mais prevalentes durante a integração, apesar de uma quantidade limitada de testes de caixa-branca poder ser usada para garantir a cobertura dos principais caminhos de controle. Depois que o software foi integrado (construído), um conjunto de *testes de alto nível* é conduzido. Critérios de validação (estabelecidos durante a análise de requisitos) precisam ser testados.

Os testes de validação fornecem garantia final de que o software satisfaz todos os requisitos funcionais, comportamentais e de desempenho. As técnicas de teste de caixa-preta são as únicas usadas durante a validação.

O último passo de teste de alto nível cai fora dos limites da engenharia de software no contexto mais amplo da engenharia de sistemas de computação. O software, uma vez validado, deve ser combinado com os outros elementos do sistema (o hardware, o pessoal, as bases de dados). O *teste de sistema* verifica se todos os elementos combinam adequadamente e se a função/desempenho global do sistema é conseguida.

6.5 – Teste Alfa e Beta

É virtualmente impossível para um desenvolvedor de software prever como o cliente irá realmente usar um programa. As instruções de uso podem ser mal interpretadas; combinações estranhas de dados podem ser usadas regularmente; saída que parecia clara para o testador pode ser ininteligível para um usuário no campo.

Quando um software sob encomenda é construído para um cliente, uma série de *testes de aceitação* é conduzida para permitir ao cliente validar todos os requisitos. Conduzido pelo usuário final ao invés de pelos engenheiros de software, um teste de aceitação pode variar de uma "volta de teste" informal para uma série de testes planejada e sistematicamente executada. De fato, o teste de aceitação pode ser conduzido ao longo de um período de semanas ou meses, descobrindo conseqüentemente erros cumulativos que poderiam degradar o sistema ao longo do tempo.

Se o software é desenvolvido como um produto a ser usado por vários clientes, não é prático realizar testes formais de aceitação com cada um. A maioria dos construtores de produtos de software usa um processo chamado teste alfa e beta para descobrir erros que apenas o usuário final parece ser capaz de descobrir.

O *teste alfa* é conduzido na instalação do desenvolvedor com o cliente. O software é usado num ambiente natural com o desenvolvedor "olhando sobre o ombro" do usuário e registrando erros e problemas de uso. Testes alfa são conduzidos num ambiente controlado.

O *teste beta* é conduzido em uma ou mais instalações do cliente pelo usuário final do software. Diferente do teste alfa, o desenvolvedor geralmente não está presente. Conseqüentemente, o teste beta é uma aplicação "ao vivo" do software num ambiente que não pode ser controlado pelo desenvolvedor. O cliente registra todos os problemas (reais ou imaginários) que são encontrados durante o teste beta e os relata ao desenvolvedor em intervalos regulares. Como resultado dos problemas relatados durante os testes beta, os engenheiros de software fazem modificações e depois se preparam para liberar o produto de software para toda a base de clientes.

6.6 – Organização do teste de software

O desenvolvedor de software é geralmente responsável por testar as unidades individuais (componentes) do programa garantindo que cada uma realiza a função para a qual foi projetada. Em muitos casos, o desenvolvedor também conduz testes de integração. Apenas depois da arquitetura do software ser completada, um *Grupo Independente de Teste* começa a ser envolvido.

O papel do *Grupo Independente de Teste* (*Independent Test Group*, ITG) é remover os problemas inerentes associados em deixar o construtor testar a coisa que ele construiu. O teste independente remove o conflito de interesses que pode de outra forma estar presente. Afinal de contas, o pessoal da equipe do grupo independente é pago para encontrar erros.

No entanto, o engenheiro de software não entrega o programa ao ITG e se retira. O desenvolvedor e o ITG trabalham juntamente durante um projeto de software para garantir que testes rigorosos serão conduzidos. Durante a condução do teste, o desenvolvedor deve estar disponível para corrigir os erros descobertos.

O ITG é parte da equipe do projeto de desenvolvimento de software no sentido de que é envolvido durante a atividade de especificação e continua envolvido (planejando e especificando procedimentos de teste) ao longo de um projeto de grande porte. No entanto, em muitos casos o ITG pertence à organização de garantia da qualidade de software, alcançando assim um grau de independência que poderia não ser possível se ele fizesse parte da organização de engenharia de software.

6.7 – Critérios para a Conclusão de Testes

Como saber se já testamos o suficiente?

O teste nunca termina, simplesmente é transferido da equipe de desenvolvimento para o usuário. Pode se considerar que o programa está sendo testado toda vez que é utilizado.

Geralmente, os testes são terminados quando todas as funcionalidades do software foram verificadas e nenhum erro grave ainda existe. No entanto isto varia de acordo com a aplicação.

Em sistemas mais complexos, pode-se aplicar métodos estatísticos para determinar que o software já foi suficientemente testado.

7 – Manutenção de Software

Define-se Manutenção como um conjunto de modificações realizadas no software após a sua entrega, ou seja, durante a sua utilização. Estas modificações são necessárias para a Correção de erros, a atualização do sistema, o aperfeiçoamento do software ou para sua adaptação a uma nova realidade.

A realização de uma manutenção pode-se incluir novos erros no software, e dependendo do seu tipo, pode até mesmo ser considerada como um novo ciclo de desenvolvimento.

7.1 – Tipos de Manutenção

Pode-se identificar, basicamente, 4 tipos distintos de manutenção:

a) Manutenção Corretiva: trata de erros encontrados durante a utilização do software.
Inclui também o diagnóstico e a correção dos erros.

b) Manutenção Adaptativa: modifica o software para ele adaptar-se a outro ambiente, como novas versões de sistemas operacionais, sistemas diferentes, periféricos e outros elementos do sistema. A vida útil de um software pode ser de até 10 anos, ao passo que outros componentes podem se modificar muito neste período.

c) Manutenção Perfectiva: inclusão de novos requisitos, recomendações de novas capacidades, modificações de funções existentes e ampliações.

d) Manutenção Preventiva: o software é modificado para melhorar a confiabilidade e a manutenibilidade futura, ou para oferecer uma base melhor para futuras ampliações.

7.2 – Atividades do Processo de Manutenção

A realização adequada de um processo de Manutenção requer as seguintes atividades:

- Avaliação da documentação e do código existente;
- Avaliação da arquitetura global, estruturas de dados, interfaces, desempenho, restrições (geralmente difícil de ser interpretado corretamente);
- Identificação das modificações necessárias e estabelecimento de um plano;
- Estudo dos efeitos das modificações;
- Realização das modificações; e
- Realização de Testes de Regressão – aplicação dos testes feitos anteriormente.

Além disto, algumas tarefas podem ser necessárias em um processo de Manutenção, como:

- Organização para manutenção;

- Plano de manutenção;
- Emissão de relatórios;
- Conservação de registros; e
- Avaliação das modificações.

7.3 – Custo de Manutenção

Na década de 80, a manutenção consumia 60% do orçamento do software. Além deste gasto, alguns fatores devem ser considerados:

- Custos intangíveis:
- Insatisfação do cliente;
- Redução da qualidade global;
- Perda da produtividade da equipe;
- Tempo gasto em manutenção (principalmente corretiva) que poderia ser usado para iniciar projetos novos.

7.4 – Problemas e Efeitos Colaterais da Manutenção

Alguns problemas podem ocorrer na realização de um processo de manutenção:

- Dificuldade de rastrear as modificações realizadas no software;
- Dificuldade de entender as modificações realizadas por outras pessoas;
- Controle de versões;
- Falta de documentação;
- A maioria dos softwares não facilita a realização de mudanças;e
- Baixa manutenibilidade.

Além disso, efeitos colaterais podem ocorrer com a realização desse processo de manutenção:

- Inclusão de erros no código fonte;
- Modificações na estrutura do código;
- Efeitos colaterais nos dados;
- Redefinição e modificação nas estruturas de dados do programa; e
- Efeitos colaterais na documentação.

Sendo assim, deve-se identificar e registrar todas as partes que foram afetadas pela modificação.

7.5 – Engenharia Reversa e Reengenharia

O termo Engenharia Reversa origina-se do hardware, onde uma empresa desmonta um equipamento (que pode ser de uma empresa concorrente) para entender a sua forma de funcionamento.

Para o software este conceito é mantido. No entanto, na maioria das vezes, deve-se entender algum produto antigo da própria empresa. O software que sofre um processo de Engenharia reversa não se tem nenhuma documentação (ou muito pouca) e pode não ter nenhum membro da equipe original de desenvolvimento.

Deve-se então analisar o software, num esforço para criar uma representação do programa em um nível de abstração maior (mais próximo do ser humano) que o código-fonte.

- Processo de recuperação do projeto.
- Utilização de ferramentas CASE pode facilitar o trabalho.

Na Reengenharia (também chamada de Renovação ou Recuperação), após recuperar-se informações de projeto de um software existente, usa-se as mesmas para alterar ou reconstituir o sistema existente. A realização da Reengenharia geralmente busca melhorar a qualidade do software.

Normalmente, o software que sofreu Engenharia Reversa passa por uma Reengenharia, melhorando ou substituindo a função do sistema existente (Adicionar novas funções, ou melhorar o desempenho global).

Para aplicar a Reengenharia, geralmente, deve-se:

- Entender o funcionamento;
- Redesenhar, recodificar e testar as partes do software que requerem modificação;
- Aplicação de ferramentas CASE; e
- Integrar novamente o programa.

As atividades necessárias dependem do programa.