INF 0573: Análise e Previsão de Desempenho de Sistemas

Objetivos: Apresentar os conceitos fundamentais para análise e previsão de desempenho de sistemas computacionais com ênfase nos testes e na qualidade de sistemas.

Professor: Paulo Cesar Azevedo Teixeira.

Conteúdo Programático:

Unidade 1. Previsão de Performance de Sistemas Computacionais
Unidade 2. Performance em Bancos de Dados
Unidade 3. Performance em Redes.
Unidade 4. Métricas de Processo e Projeto de Software
Unidade 5. Medidas da Qualidade de Software
Unidade 6. Técnicas de Teste de Software

Bibliografia:

- Pressman, Roger S. - Engenharia de Software - 5.ed. - Rio de Janeiro: McGraw-Hill, 2002.
- Weinberg, G. M. - Software com Qualidade - 1.ed. - São Paulo: Brasport, 1999.
- Rezende, Denis A. - Engenharia de Software e Sistemas de Informação - 2.ed. - Rio de Janeiro: Brasport, 2002.
- Molinari, Leonardo – BTO: Otimização da Tecnologia de Negócio – 1.ed. – São Paulo: Érica, 2003.
- Molinari, Leonardo – Testes de Software: Produzindo Sistemas Melhores e Mais Confiáveis – São Paulo: Érica, 2003.
- Feit, Sidnie. "SNMP - A Guide to Network Management". McGraw-Hill. 1994;
- Leinwald, Allan; Fang,Karen. "Network Management - A Practical Perspective".
- KOCK, George; KEVIN, L. Oracle 8: The Complete Reference. Oracle Press, 1997.
- Kevin, L.; THERIAULT, M. Oracle 8i DBA handbook. Osborne : McGraw-Hill, 1999.

Unidade 1 - Previsão de Performance de Sistemas Computacionais

I.1 - Introdução

O coração de um processo de planejamento de capacidade de sistemas é a habilidade de prever adequadamente a performance de uma configuração particular de um sistema computacional executando uma determinada carga de trabalho. Usando essa habilidade o planejador de capacidade pode ter que analisar várias configurações pois podem existir muitas opções disponíveis para construir a solução do sistema. Estações de trabalho, servidores, mainframes, sistemas com objetivos específicos e redes, podem ser combinados em diferentes configurações. Uma vez que a previsão sempre traz um certo grau de incerteza, vários cenários de cargas de trabalho futuras devem ser consideradas também.

Neste ponto do processo de planejamento de capacidade o que é necessário é a adoção de uma técnica de predição de performance. Tendo alternativas de configuração e previsão de carga de trabalho como entrada, a técnica de predição deve ser hábil no cálculo das medidas de performance para o sistema considerado. Não é necessário que se tenha uma predição exata para uso em planejamento de capacidade. O que realmente é levado em conta, neste processo, é a habilidade de se obter tendências corretas. A figura 1, ilustra um esquema do processo de previsão de performance.

Figura 1 – Processo de previsão de performance

Independente da técnica de predição a ser adotada, uma saída típica do processo de previsão de performance é o gráfico relacionando parâmetros de performance (tempo de resposta, throughput, taxa de utilização), em função da intensidade de carga, como apresentado na figura 2, a seguir.

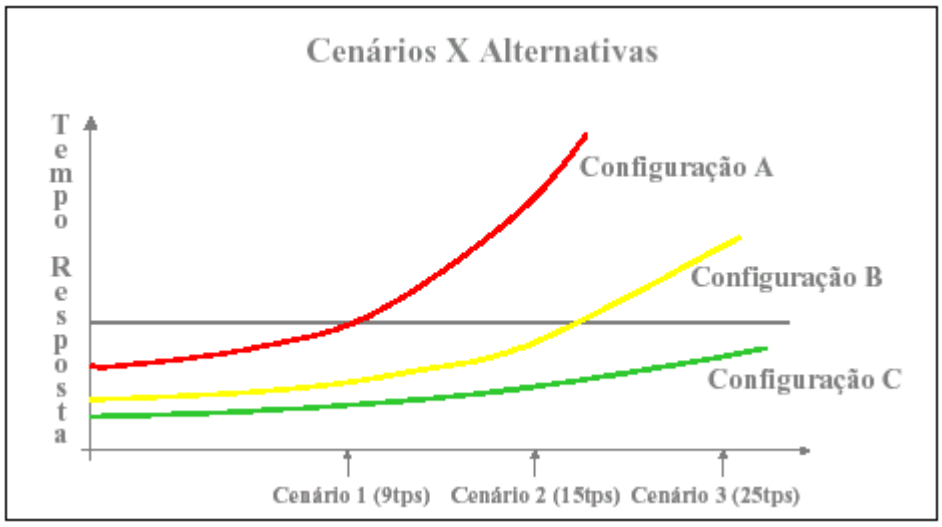


Figura 2 – Cenários de processamento de transações on-line

Nesta figura, temos tempo de resposta do processamento de um sistema como função para várias configurações de sistema. Cada cenário corresponde a uma previsão específica. Cada cenário corresponde a uma determinada previsão de carga. Baseado nos limites aceitáveis de tempo de resposta, é possível selecionar a configuração mais adequada para cada cenário. Nesta seleção além do nível de serviço aceitável, deve-se levar em conta a relação custo x benefício. No exemplo ilustrado pela figura 2, a configuração A, não atende o nível de serviço mínimo para a menor carga de trabalho prevista. A configuração C é a que tem melhor performance. Entretanto, se o cenário 2 corresponder a uma previsão mais provável de carga de trabalho (considerando-se valores médios), e se levarmos em conta a relação custo x benefício, a configuração B pode ser adotada na definição do modelo de previsão de performance.

I.2 - Técnicas de Previsão de Performance

As técnicas de previsão de performance normalmente diferem-se em três aspectos: complexidade, precisão e custo. Existem quatro tipos básicos de técnicas de previsão de performance, ordenadas em grau de complexidade e custo. São elas:

- Experiência do Decisor - Regra do dedo polegar (Rules of Thumb);
- Análise de tendências (Trend Analysis);
- Modelos de Performance (Modelos de Fila e Modelos de Simulação);
- Benchmarks.

A figura 3 ilustra o posicionamento dessas técnicas em termos de complexidade e custo.

Figura 3: Complexidade e custo das ténicas

I.2.1 - Experiência do Decisor - Rules of Thumb (Regra do Dedo Polegar)

Esta técnica é usada para determinar a capacidade total de um sistema como uma função da utilização de componentes chave feita individualmente.. Em geral, a aludida regra define limites que devem ser utilizados como linhas mestras. Usualmente, na regra do polegar, estas linhas mestras são extraídas das informações dos fabricantes, experiências no ambiente de produção, resultados de benchmarks e experiência pessoal. Exemplo de algumas regras populares são mostradas abaixo:

- 80 % de utilização da CPU é um bom limite a ser considerado;
- O pico de CPU não deve exceder 90 %;
- A relação entre utilização de pico e média deve ser de 2.25:1, ou seja, a utilização mínima de CPU não deve ser inferior a 40 %.

Entretanto, estes e outros exemplos de limites, devem ser estabelecidos com cautela, tendo em vista que cada sistema tem as suas especificidades e os parâmetros podem variar de sistema para sistema. A regra do polegar oferece a vantagem da simplicidade e baixo custo, além da fácil utilização.

A grande limitação da regra do polegar, ocorre quando o gargalo não está nos recursos chave. Nesta técnica não são estabelecidas as ligações entre os componentes do sistema, tendo sua aplicabilidade limitada nos modernos sistemas distribuídos onde é difícil identificar um simples recurso.

Desta forma, está é uma técnica não recomendada para ser utilizada em previsão de performance.

I.2.2 - Trend Analisys (Análise de Tendências)

A análise de tendência pode ser visualizada como uma técnica que pode predizer o que pode acontecer com o sistema, baseado no resgate de dados históricos de seu comportamento passado. No caso de comportamento de planejamento de capacidade, dados históricos são armazenados pela aquisição de logs que são usados para analisar o relacionamento entre performance e carga de trabalho. A análise desses dados é utilizada para identificar a tendência do comportamento de performance. O processo de predição consiste em se extrapolar as tendências para prever o nível de carga de trabalho futura. Uma

das forma de se fazer esta previsão é utilizar extrapolação linear de tendências recentes, como mostrado na figura 4.

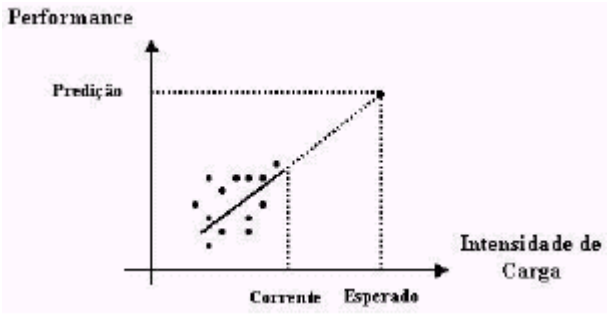


Figura 4 – Predição de performance usando análise de tendência

Genericamente, o problema com a extrapolação linear está no fato de que com o crescimento da carga de trabalho e o aparecimento dos gargalos, o congestionamento cresce, a performance cai de forma não linear, comprometendo a precisão da técnica.

I.2.3 - Benchmarks

O uso de benchmarks tem sido utilizado como uma ferramenta útil para predição de capacidade de novos sistemas computacionais. Uma estudo aponta que a predição de performance de sistema é vista como correta se ambos, carga de trabalho e recursos de sistema, são adequadamente representados num benchmark.

Na previsão de performance, a questão chave é como uma dada configuração de sistema pode interferir no processamento de uma determinada carga de trabalho. Para responder esta questão utilizando aproximação de benchmarks, simplesmente execute a carga de trabalho no sistema e meça o resultado de performance. Esta aproximação de medida direta é somente aplicável para sistemas operacionais. Assim uma análise de várias alternativas de configuração não é possível na prática, pois envolve um alto custo na configuração e execução desses experimentos de benchmarks. Ambientes distribuídos compostos de equipamentos de múltiplos fabricantes tornam o problema mais complexo.

Outra limitação no processo de medida direta é a impossibilidade de avaliar o impacto de novas aplicações que não foram completamente desenvolvidas. Quando conduzidos apropriadamente, os benchmarks podem obter resultados bastante precisos, em sistemas de carga real. Seus resultados são bastante confiáveis e têm grande aceitação.

Benchmarks são freqüentemente tidos como técnicas de comparação de sistemas. Normalmente, a aquisição e análise de comparação de sistemas computacionais, contam com o uso de benchmarks. Entretanto, o seu uso em estudos de planejamento de capacidade práticos, tem sido limitado.

I.2.4 - Modelos de Performance

Um modelo é a representação de um sistema. No caso de sistemas computacionais, modelos podem ser usados para, pelo menos, dois propósitos:

a) para representar a operação do sistema (modelos funcionais); ou
b) para representar o comportamento do sistema em termo de sua performance.

Modelos funcionais podem ser empregados quando um dos interesses é o estudo de certas propriedades de comportamento de um sistema. Modelos de performance são úteis para prever os valores de medida de performance de um sistema a partir de um conjunto de valores de carga de trabalho, sistemas operacionais e parâmetros de hardware. Exemplo de medidas de performance são: tempo de resposta, throughput e utilização. A figura 5, ilustra a representação de um modelo de performance. As entradas do modelo se dividem em três categorias: carga de trabalho, software básico e parâmetros de hardware.

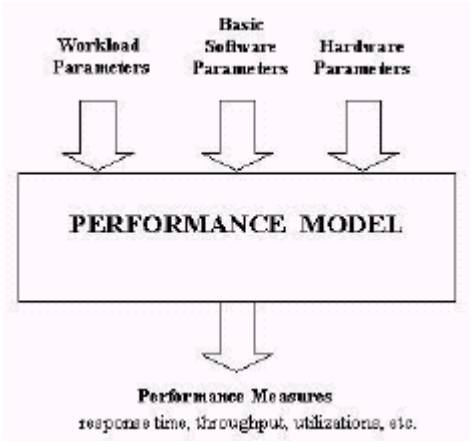


Figura 5 – Representação de um modelo de performance

Os parâmetros de carga de trabalho, descrevem a carga imposta ao sistema computacional nos trabalhos ou transações submetidas ao mesmo. Exemplo desses parâmetros são: tempo médio de transação, número de terminal de sistema interativos, demanda de serviço colocado em cada recurso do sistema por cada tipo de trabalho ou transação. Os parâmetros básicos de softwares descrevem as características básicas do software como o sistema operacional, que afeta a performance. Exemplos desses parâmetros são: máximo grau de multiprogramação e prioridade de despacho de cpu.

Exemplos de parâmetros de performance de hardware incluem a velocidade de processador, latência e taxa de transferência de disco e velocidade de LAN. A saída do modelo de performance é um conjunto de medidas de performance. Exemplos incluem tempo de resposta, throughput e utilização de recursos. Dentro desse enfoque, existem basicamente dois diferentes tipos de modelos de performance: simulação e analítico.

I.2.4.1 - Modelos de Simulação

Modelos de simulação são baseados em programas de computadores que emulam diferentes aspectos dinâmicos de um sistema tão bem como sua estrutura estática.

Consumidores de recurso (transações, jobs, comandos, etc) são gerados através de processos probabilísticos utilizando geradores de número aleatórios, sendo gerados, pelos fluxos do sistema, eventos como a chegada de um cliente à fila de espera de um servidor, iniciando o serviço em um servidor qualquer, indo até o final, e assim por diante. Os eventos são processados de acordo com sua ordem de ocorrência no tempo.

Contadores acumulam estatísticas que são usadas no fim da simulação para executar a estimativa de valores de várias medidas de performance. Os níveis de detalhe necessários nos modelos de simulação, normalmente os tornam caros para executar, desenvolver e validar. Por outro lado, eles habilitam o nível de investigação de um fenômeno proporcional ao que se está pronto para pagar.

Por essa razão, programas de simulação são normalmente executados em mainframes ou estações de trabalho potentes.

I.2.4.2 - Modelos de Filas (Analíticos)

Os modelos analíticos são compostos de um conjunto de formulas e/ou algoritmos computacionais que provêem os valores de medidas de performance desejadas como uma função de um conjunto de valores de parâmetros de performance. Para modelos analíticos serem matematicamente tratáveis, eles são geralmente menos detalhados que os modelos de simulação. Assim eles são geralmente menos precisos, mas muito mais eficiente para executar. Desta forma, modelos analíticos podem ser executados em computadores pessoais.

Podemos resumir as vantagens e desvantagens dos modelos analíticos e de simulação, como:

- Modelos analíticos são mais eficientes para serem executados do que os modelos de simulação;
- Devido ao alto nível de abstração para se obter os valores dos parâmetros de entrada do modelos analítico os tornam mais simples do que os modelos de simulação;
- Modelos de simulação podem ser usados quando se necessita maior detalhamento, sendo mais precisos se comparados aos modelos analíticos.

Unidade 2 – Performance em Bancos de Dados

II.1 - Visão geral

A tradução literal de 'tuning' seria sintonia ou ajuste de alguma coisa para que funcione melhor. Um SGBD é um software que permite vários ajustes que, por sua vez, podem afetar a performance de um banco de dados.

Mas, com conceituar performance de banco de dados? Para responder a esta pergunta vamos fazer uma analogia em termos de oferta e demanda. Os usuários demandam informações do banco de dados. O SGBD fornece informação para aqueles que o pedem. A taxa entre os pedidos que o SGBD atende e a demanda para informação poderia ser denominada performance de banco de dados.

Cinco fatores influenciam a performance do banco de dados: 'workload', 'throughput', recursos, otimização e contenção.

- 'Workload' são os pedidos ao SGBD que definem a demanda. Ele é o conjunto de transações online, jobs batch, pesquisas ad hoc, etc;
- 'Throughput' define a capacidade do computador de processar os dados. Ele é uma composição de velocidade de I/O, velocidade da CPU, capacidades de paralelismo da máquina e a eficiência do sistema operacional e o software básico envolvido.
- O hardware e ferramentas de software disponíveis no sistema são conhecidos como recursos do sistema.
- Todos os sistemas podem ser otimizados, mas os bancos de dados relacionais são os únicos em que a otimização de pesquisas é primariamente realizada internamente ao SGBD.
- Quando a demanda (workload) para um recurso particular é alta, pode acontecer a contenção. Contenção é a condição em que dois ou mais componentes do 'workload' estão tentando usar o mesmo recurso em modos conflitantes (por exemplo, duas atualizações no mesmo dado). Se a contenção cresce o 'throughput' diminui.

Performance de banco de dados, então, pode ser conceituada, como otimização dos recursos usados para aumentar o 'throughput' e minimizar contenção, permitindo que o maior 'workload' possível possa ser processado.

Não importa o quanto um SGBD é complexo e cheio de características, a coisa mais problemática para os que o utilizam é a sua performance. Se houver problema de performance, o uso da aplicação declinará e as supostas vantagens competitivas disponibilizadas pela aplicação não ocorrerão. O planejamento para o gerenciamento da performance do banco de dados é um componente crucial de qualquer implementação de aplicação. Sem um plano para monitorar performance e ajustar o banco de dados, a degradação da performance fatalmente ocorrerá. Um plano completo de gerenciamento de performance incluirá ferramentas para ajudar a monitorar a performance da aplicação e o ajuste do SGBD.

Por ‘tuning’ da base de dados, poderíamos entender como uma customização do sistema sob medida para que a performance atenda melhor as suas necessidades.

II.2 - Metodologia para Ajuste de Performance

Uma metodologia bem planejada é a chave do sucesso para realizar ‘tuning’ de performance. Para obter melhores resultados, o SGBD deve ser ajustado durante a fase de projeto em vez de esperar para ajustar depois da implementação do sistema. O tempo mais efetivo para este ajuste é durante a fase de projeto: você consegue o máximo de benefício por um baixo custo.

A abordagem mais efetiva para realizar ‘tuning’ é a abordagem pró-ativa na fase de projeto. O processo de ‘tuning’ não começa quando os usuários reclamam sobre tempo de respostas ruins. Quando o tempo de resposta está ruim, geralmente é muito tarde para usar algumas das estratégias mais eficientes de ‘tuning’. Neste ponto, se você não quiser redesenhar completamente a aplicação, você pode somente melhorar a performance pela realocação de memória e ajuste de I/O. Você pode chegar à conclusão que tanto o SGBD quanto o sistema operacional estão funcionando bem. Neste caso, para conseguir obter performance adicional você precisaria ajustar a aplicação ou adicionar recursos. Uma metodologia recomendada para ‘tuning’ deveria seguir os seguintes passos:

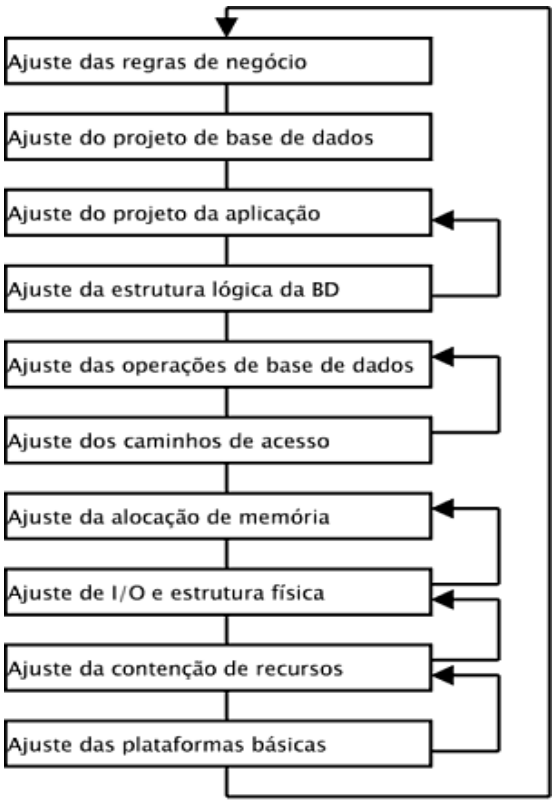


Figura 1 – Estrutura da Metodologia

II.2.1 - Ajuste das Regras de Negócio

Para obter a performance ideal, você pode ter que adaptar regras de negócio. Isto se refere a uma análise do sistema inteiro em mais alto nível. Deste modo, os planejadores garantem que os requisitos de performance do sistema correspondam diretamente às necessidades concretas do negócio.

Problemas de performance encontrados pelo DBA podem ser causados por problemas no projeto e implementação ou por regras de negócio inapropriadas. Por exemplo, na função de negócio "impressão de cheques", o requisito real é pagar dinheiro a pessoas; o requisito não é necessariamente imprimir pedaços de papel. Poderia ser bastante complicado imprimir milhares de cheques por dia. Seria relativamente mais fácil gravar os depósitos de pagamento em uma fita que seria enviada ao banco para processamento.

II.2.2 - Ajuste do projeto de Base de Dados

Na fase de projeto de base de dados, você deve determinar **quais** os dados são necessários à suas aplicações. Você precisa considerar que relações são importantes e quais são seus atributos. Finalmente, você precisa estruturar a informação para melhor atingir suas metas de performance.

O processo de projeto da base de dados geralmente desce ao estágio de normalização quando os dados são analisados para eliminar a redundância de dados. Com a exceção das chaves primárias, qualquer outro dado deveria ser armazenado somente uma vez na sua base. Depois de normalizar os dados, entretanto, você pode precisar denormalizar por razões de performance. Alguns autores afirmam que nenhuma aplicação principal rodará na terceira forma normal.

A aderência muito rígida para projetos de tabelas relacionais trarão performance ruim. O problema é que estes projetos refletem os modos em que os dados de uma aplicação estão relacionados com outros dados. Eles não refletem os caminhos de acessos normais que os usuários empregarão para acessar estes dados. Uma vez que as necessidades de acesso dos usuários evoluem, o projeto de tabelas relacionais se tornará difícil de trabalhar para pesquisas muito grandes. Um problema ocorrerá com pesquisas que retornam um número muito grande de colunas. Estas colunas são normalmente espalhadas entre várias tabelas, forçando a junção de tabelas durante a pesquisa. Se uma das tabelas da junção for grande, então a performance de toda a pesquisa poderá sofrer.

Há várias formas de denormalização de dados, por exemplo, criando pequenas tabelas sumarizadas para tabelas grandes e estáticas. Se os usuários freqüentemente utilizam um dado derivado que não sofre muitas modificações, então faz sentido armazenar periodicamente o dado no formato em que os usuários utilizarão.

Opções de projeto incluem a separação de uma tabela em múltiplas tabelas e, o contrário, combinando múltiplas tabelas em uma. A ênfase deveria ser em fornecer aos usuários o caminho mais direto possível para os dados que eles querem e no formato que eles querem.

II.2.3 - Ajuste do Projeto da Aplicação

Analistas de negócio e projetistas deveriam transformar metas de negócio em um projeto de sistema efetivo. Processos de negócio referem-se a uma aplicação particular dentro do sistema ou uma parte da aplicação. Um exemplo de projeto de processo é deixar alguns dados em cache. Deste modo você evita a recuperação da mesma informação várias vezes durante o dia.

II.2.4 - Ajuste da Estrutura Lógica da Base de Dados

Depois da aplicação ter sido projetada, você pode planejar a estrutura lógica da base de dados. Um ajuste fino no projeto de índices, garantirá que não ocorra falta ou exagero de índices. Nesta fase você pode criar índices adicionais além da chave primária e estrangeira.

II.2.5 - Ajuste das Operações de Base de Dados

Antes de ajustar o SGBD, esteja certo que sua aplicação utiliza todas as vantagens do SQL e das características do gerenciador para o processamento da aplicação. Utilize o otimizador do SGBD e o controle de 'locks'(bloqueios). Entender o mecanismo de processamento da pesquisa no SGBD é importante para escrever comandos SQL efetivos.

II.2.6 - Ajuste dos Caminhos de Acesso

Em bases de dados relacionais, a localização física dos dados não é tão importante como seu lugar lógico dentro do projeto da aplicação. Entretanto, a base de dados tem que encontrar os dados em ordem para retorná-lo para o usuário realizar a pesquisa. A chave para afinar o SQL é minimizar o caminho de pesquisa que a base de dados utiliza para achar o dado.

Garanta que exista acesso aos dados de modo eficiente. Considere o uso de 'clusters', hash 'clusters', índices B-tree, índices bitmap. Isto pode significar que você precise analisar novamente seu projeto depois de já ter sido construído.

II.2.7 - Ajuste da Alocação de Memória

Alocação apropriada de recursos de memória para as estruturas do SGBD pode trazer efeitos positivos na performance. Além disso, a alocação apropriada desses recursos melhoram a performance do cache, o que reduzirá o 'parsing' de comandos SQL e a paginação.

II.2.8 - Ajuste de I/O e Estrutura Física

I/O de disco tende a reduzir a performance de várias aplicações de software. O ajuste de I/O e estrutura física envolve:

- Distribuir dados em discos diferentes para distribuir I/O e evitar contenção de disco;
- Criar 'extents' grandes o suficiente para seus dados e evitar extensões dinâmicas das tabelas. Isto afeta a performance de alto volume de aplicações OLTP.

II.2.9 - Ajuste da Contenção de Recursos

O processamento concorrente de vários usuários pode criar a contenção de recursos. A contenção faz com que os processos esperem até que os recursos sejam disponibilizados.

II.2.10 - Ajuste das Plataformas Básicas

Conforme a versão do gerenciador para um determinado sistema operacional pode haver parâmetros diferentes de ajuste.

II.3 - Como Aplicar a Metodologia

Nunca comece a tarefa de 'tuning' sem ter estabelecido objetivos claros. Você não pode ter sucesso sem uma definição do que é o sucesso. Mantenha seus objetivos em mente para considerar cada medida de ajuste, considere os benefícios de performance à luz de seus objetivos. Lembre que seus objetivos podem entrar em conflito. Por exemplo, para encontrar a melhor performance para um comando SQL, você pode sacrificar a performance de outro comando SQL concorrente em sua base de dados.

Os objetivos para ajuste de performance variam dependendo das necessidades da aplicação: aplicações de processamento online (OLTP), de suporte à decisão, processamento 'batch', aplicações distribuídas, etc. Se estivermos projetando ou mantendo uma aplicação, é necessário estabelecer objetivos de performance específicos para saber quando se deve realizar o ajuste. Podemos desperdiçar tempo ajustando o sistema se alterarmos parâmetros de inicialização sem um objetivo específico.

Desenvolvedores de aplicações e DBA's devem ser cuidadosos em estabelecer expectativas apropriadas de performance para os usuários. Crie uma série mínima de testes que possam ser repetidos. Com um mínimo de testes estabelecido, com um 'script' para conduzir os testes e sumarizar e analisar os resultados; pode-se, então, testar várias hipóteses para ver o seu efeito.

Guarde os registros dos efeitos de cada mudança no 'script' de teste e automatize os testes. Evite mudar alguma coisa no sistema por adivinhação. É possível não se ter pensado em todos os detalhes e afetar todo o ambiente. A performance do ambiente pode degradar a ponto de ter de termos de reconstruí-lo a partir de 'backups'.

Tente evitar preconceitos quando for tratar um problema de performance. Peça para os usuários para descreverem o "*problema de performance*". Não espere que os usuários saibam porque o problema existe.

Pare de realizar o ajuste quando os objetivos forem alcançados. Comunique aos usuários afetados pelo problema e os responsáveis pela aplicação.

II.4 - Ajuste das Operações na Base de Dados

Todo acesso a dados relacionais pelos programas de aplicação é feito usando SQL. Revisões de SQL deveriam ser um componente necessário para análise de performance de aplicações em banco de dados antes e depois da implementação.

O principal culpado por problemas de performance é o SQL no código da aplicação. O consenso da indústria indica que 75% a 80% de todos os problemas de performance em banco de dados pode ser encontrado em códigos SQL ruins. Isto não significa que o SQL nas aplicações seja ruim desde o início. De fato, uma aplicação pode ser 100% ajustada para acesso rápido quando ele é transferido para produção, mas durante o tempo ocorrem degradações de performance. Isto pode ocorrer por muitas razões tais como crescimento da base de dados, novos caminhos de acesso, mudanças no negócio, etc.

Foque seus esforços de ajuste em comandos onde os benefícios de ajuste excedem o custo do ajuste. Em complementação a políticas e procedimentos para análise e revisão de SQL podem ser usadas ferramentas automatizadas para minimizar o volume de códigos SQL ruins. Encontre os comandos que consomem mais recursos e/ou são executados mais freqüentemente. Não adianta ajustar comandos SQL para projetos ineficientes da aplicação.

O SQL é projetado para que o programador especifique que dados são necessários e não como recuperá-los. O otimizador do SGBD se encarrega dessa parte. Otimizadores do SGBD geralmente fazem uma boa escolha do caminho de acesso mais eficiente, mas não sempre. É necessário um método para monitorar e ajustar as operações na base de dados. O SQL é o coração das aplicações modernas com bancos de dados. Se forem mal codificados e concebidos, a performance da aplicação sofrerá e os processos do negócio sofrerão impactos negativamente.

II.4.1 - Abordagens para ajuste de comandos SQL

Reestruture os índices

Reestruturação de índices é um bom ponto de começo, porque ela tem mais impacto na aplicação que a reestruturação do comando ou dos dados.

Analise a eficiência dos índices usando Explain Plan. Veja que índices são usados para pesquisas comuns e exclua qualquer um que não esteja sendo usado. Muitos índices em

uma tabela podem causar uma sobrecarga, pois todos os índices precisam ser atualizados na atualização da tabela.

Remova índices não seletivos, crie índices para caminhos de acesso com performance crítica e considere outros tipos de índices.

Reestruture o comando SQL

Depois de reestruturar os índices, podemos tentar reestruturar os comandos SQL. Escrever novamente um comando SQL ineficiente é mais fácil do que repará-lo. Se você entender a finalidade do comando, você pode rapidamente escrever um novo comando que atenda aos requisitos.

O SQL é uma linguagem flexível. Comandos SQL podem ser formulados de várias maneiras diferentes com a mesma funcionalidade. A performance destas opções podem flutuar grandemente, mas os dados retornados para a aplicação podem ser equivalentes. Por esta razão, é imperativo que a melhor opção seja usada para garantir uma boa performance.

Uma revisão de SQL conduzida por analistas com experiência em performance pode capturar estes tipos de potenciais problemas de performance. Use o resultado do comando Explain Plan para comparar os planos de execução e custo dos comandos e determinar qual é mais eficiente.

O principal objetivo do ajuste de SQL é evitar realizar trabalho desnecessário para acessar linhas que não afetarão o resultado. Evite ler uma tabela inteira se é mais eficiente obter as linhas desejadas através de um índice. Evite usar um índice que traga mais linhas que um outro.

Quando múltiplas tabelas são acessadas em um comando SQL as tabelas podem ser combinadas em qualquer ordem e ainda retornarem o resultado correto. A performance, entretanto, pode variar grandemente dependendo do volume dos dados nas tabelas, da natureza do pedido e dos índices disponíveis para a tabela. A ordem de um Join pode ter um efeito significativo de performance. Escolha uma ordem no Join de modo a juntar menos linhas nas tabelas posteriores à ordem do Join.

Minimize o tempo de recursos bloqueados com o comando ‘commit’ logo após as alterações do banco.

Use valores de coluna sem transformação. Não use funções SQL em cláusulas de predicado ou cláusulas ‘where’ pois eles deixarão de utilizar índice.

Evite misturar tipos de dados em expressões e tome cuidado com conversões implícitas de tipo. Qualquer expressão usando uma função com a coluna como seu argumento, o otimizador ignorará a possibilidade de usar o índice daquela coluna. Podem ocorrer erros na conversão de dados.

Escreva comandos SQL separados para valores específicos. SQL não é uma linguagem procedural. Não é uma boa idéia usar um pedaço de SQL para fazer diferentes coisas. Geralmente o resultado é pior que o resultado para cada tarefa.

Cuidado ao usar IN com uma lista de valores. Isto pode indicar a falta de uma entidade.
Não recicle 'views'. As vezes você estará acessando a 'view' desnecessariamente e seria mais rápido acessar a tabela original.

Modifique ou desabilite 'triggers'

A utilização de 'triggers' consome recursos do sistema. Se você usa muitos 'triggers', você pode afetar negativamente a performance e você pode precisar modificá-lo ou desabilitá-lo.

Reestruture o Dado

Depois de reestruturar os índices e o comando, você pode considerar a reestruturação do dado. Introduza valores derivados. Evite cláusula 'Group By' em códigos com respostas críticas. Implemente entidades que faltam e tabelas de interseção. Reduza a carga da rede. Migre, replique ou particione os dados.

II.5 - Conclusão

O administrador de banco de dados (DBA) pode ser um parceiro estratégico no desenvolvimento e manutenção de aplicações:

- gerenciando pró-ativamente os sistemas.
- identificando problemas potenciais antes de eles ocorrerem, prevenindo quedas do sistema e perda de dados;
- um conhecimento profundo do funcionamento físico do sistema, torna o DBA um membro chave para o time de desenvolvimento;

Como o volume de dados e a demanda por eles cresce exponencialmente, a organização precisa de um bom administrador que possa constantemente sugerir modos melhores, mais rápidos e mais baratos para armazenar e movimentar informações.

Uma metodologia bem planejada é fundamental para ajustes de performance.

Aplicações que acessam banco de dados relacionais somente são tão boas quanto a performance que eles alcançam.

Unidade 3 – Performance em Redes

Podemos constatar que o cerne da questão é o desempenho da rede, sempre uma preocupação para usuários e administradores. Ambos querem definir e medir a qualidade do serviço (QoS) disponibilizada por suas redes. Os usuários precisam contar com uma qualidade de serviço mínima previsível, especialmente a medida que os serviços se dinamizam, tais como o WWW, videoconferência e a colaboração em tempo real. Os administradores, por sua vez, querem poder mostrar o que estão entregando para as organizações.

O que é o desempenho então? O desempenho da rede relaciona-se diretamente à velocidade da rede. Já o desempenho das aplicações relaciona-se à velocidade das aplicações, como é visto pelo usuário final, e depende, também, da rede, do servidor, do cliente, e da aplicação.

O desempenho da rede é uma preocupação central, especialmente porque a rede é geralmente responsabilizada pela maioria dos problemas de desempenho.

Essencial a uma gerência de desempenho eficaz é a determinação de quando os atrasos ocorrem e onde estes se encontram realmente, de modo que as ações corretivas possam ser feitas. Por exemplo, uma rede que escoe o tráfego rapidamente pode ser vista como "lenta" se o servidor for sub-dimensionado ou estiver suportando demasiados usuários. Por outro lado, um servidor finamente ajustado pode não apresentar nenhum impacto positivo na performance caso hajam atrasos excessivos na rede.

A medição de desempenho torna-se mais difícil ao passo que as aplicações se tornam cada vez mais complexas. Por exemplo, um pedido do cliente pode não ser completado com uma resposta de um servidor único - o que seria razoavelmente fácil de medir. Pode mover-se sobre rotas variantes com latências diferentes. Especificamente, um pedido do cliente pode requerer: um look-up do diretório para encontrar o servidor adequado, a criação de uma conexão de rede, passando o pedido, o servidor acessando um outro servidor, retornando então a resposta. Ou uma única transação do cliente pode requerer respostas múltiplas do servidor para ser completada.

III.1 - Parâmetros de Performance

Em virtude do exposto anteriormente, torna-se importante que se defina os principais parâmetros utilizados para a avaliação do desempenho de redes, conceituando-os de maneira criteriosa a fim de que se possa diagnosticar eventuais deficiências de produtividade mais precisamente.

III.1.1 - Disponibilidade

- Definição: Uma medida da disponibilidade de utilização da rede para um serviço. A disponibilidade é medida geralmente como uma porcentagem do dia, da semana, ou do mês onde o recurso poderia ser usado, como 99,99%.

- Valor: Uma maneira de avaliar a saúde básica da rede.
- Conceito errado: A disponibilidade está sempre relacionada com a performance.
- Realidade: A disponibilidade e o desempenho não são interdependentes. Por exemplo, uma rede congestionada pode estar inutilizável por causa da lentidão, mesmo com todos os recursos disponíveis.

III.1.2 – Largura de Faixa

- Definição: Uma medida da capacidade de um link de comunicações. Por exemplo, um link T1 tem uma largura de faixa de 1,544 Mbps. É usada também para medir a capacidade atribuída a um serviço através de um link; por exemplo, uma entrada de vídeo através de um link T1 pode ter uma largura de faixa atribuída de 384 Kbps.
- Valor: Útil para determinar a capacidade necessária para os serviços. Gerentes de IT consultam freqüentemente a utilização da largura de faixa - a porcentagem da largura de faixa total que está sendo usada.
- Limitações: Não se relaciona necessariamente ao desempenho. Um "pipe" de velocidade mais elevada melhora a capacidade (mais largura de faixa), mas uma transferência de um arquivo grande pode ainda retardar todos os demais.

III.1.3 – Linha de Base (BASE LINE)

- Definição: Uma medida do comportamento "normal". Muitas redes experimentam " picos de tráfego " em vários momentos relacionados a operações de negócios fundamentais - acesso de correio eletrônico e outros recursos. Uma linha de base é útil para distinguir um dia " ruim ", ou uma anomalia aleatória, dos dias "normais".
- Valor: As linhas de base ajudam a um administrador identificar uma mudança repentina, que possa indicar um problema. Com o tempo, as linhas de base indicam tendências nas atividades para finalidades de planejamento.

III.1.4 - Congestionamento

- Definição: O congestionamento ocorre em função de cargas mais elevadas, indicando à rede ou a um dispositivo que este está alcançando, ou excedeu, sua capacidade. O congestionamento conduz primeiramente a um rápido crescimento da latência (definição no item E) e, então, à perda dos dados se a situação não for corrigida. Os atrasos enfileirando-se com os pacotes que esperam para ir são uma indicação de problemas possíveis de latência.
- Valor: A detecção adiantada permite ao administrador tomar providências mais rapidamente, evitando lentidões do sistema desastrosas. Os relatórios de desempenho ajudam a identificar pontos potenciais de congestionamento antes que estes afetem o desempenho.

III.1.5 - Latência

- Definição: Uma medida do atraso de uma extremidade de uma rede, de um link, ou de um dispositivo a outra. Uma latência mais elevada indica atrasos mais longos. A latência nunca pode ser eliminada inteiramente, e é usada como uma medida do desempenho da rede. Como a utilização, a latência pode variar em função da carga imposta à rede. Um portador pode mudar sua latência da rede alterando seus circuitos virtuais de modo que estes usem links mais lentos ou incorporem mais "hops". Cada vez que uma célula ou pacote é retransmitido, ele é submetido a um "hop" (salto). O rastreamento da latência de uma rede é essencial.
- Valor: Uma medida chave para identificar rapidamente pontos geradores de problema em potencial. Por exemplo, se o tempo de resposta cair mas a latência da rede se mantiver inalterada, o problema pode muito provavelmente ser atribuído ao servidor ou ao cliente.
- Limitações: A medição de latência requer instrumentação e coleta de dados. A correlação da latência da rede com interações e conexões das múltiplas aplicações é bastante difícil.
- Conceito errado: A latência da rede é o tempo de resposta.
- Realidade: A latência é uma parte do tempo de resposta. Os servidores, os clientes, e as aplicações adicionam sempre alguma latência. Adicionar largura de faixa nem sempre repara problemas de latência. Por exemplo, se um portador tivesse demasiados "hops" em um link de alta velocidade, poderia ainda assim ter uma latência mais elevada (atraso) do que a de um link de baixa velocidade com poucos "hops".

III.1.6 – Disparo (THRESHOLD)

- Definição: Um valor que é ajustado para advertir o sistema de gerência quando a utilização, a latência, ou o congestionamento excederem limites críticos. O disparo é ajustado nos agentes de gerência que medem o comportamento real das redes e dos links. Por exemplo, um administrador pode ajustar um disparo de 50% da utilização em um link da rede, de modo que haja tempo de responder aos volumes de tráfego crescentes.
- Valor: Permite ao administrador ajustar os "trip-wires" e receber alertas a tempo de responder antes que os usuários se queixem.
- Limitações: Os agentes devem ser capazes de rastrear valores e disparar alarmes. Em virtude de os níveis de disparo poderem ser excedidos muitas vezes, alguma espécie de mecanismo de filtragem e de prioridade é necessário para indicar quando tais eventos são significativos.

III.1.7 - Utilização

- Definição: Uma medida de quanto da capacidade está sendo usado realmente em algum ponto. Se um link T1 comporta 924 Kbps, este tem uma utilização de 60% nesse intervalo particular do tempo. A utilização varia de acordo com as cargas reais do

tráfego e com o intervalo de tempo do qual é calculada a média. É também uma medida da carga do processador central nos servidores e nos clientes.

- Valor: Uma medida dos níveis de uso que pode ser usado prevê problemas potenciais com tendências. Pode dar avisos em tempo real de problemas de desempenho potenciais.

III.2 - Nível de Serviço

Outro ponto importante que exerce influência na performance de uma rede é a questão do atendimento aos serviços dela requisitados. Ou seja, devemos avaliar os parâmetros que determinam a qualidade com que estes serviços são recebidos e tratados. Seguem, então, alguns termos fundamentais acerca desta questão.

III.2.1 - Acordos de nível de serviço (Service Level Agreements - SLA)

- Definição: SLAs são contratos entre o fornecedor e o usuário que detalham o que o usuário espera do fornecedor. Um SLA bom tem: as descrições específicas dos serviços que estão sendo entregues, incluindo os critérios usados para avaliação do serviço; requisitos de relatório; acordos de escalação (o que fazer quando houver interrupções sérias); e penalidades para quando não se cumprirem com os termos do contrato.

III.2.2 - Disponibilidade

- Definição: Uma medida do nível de prontidão da rede para a atividade do usuário. É medida geralmente como o tempo médio entre falhas (Medium Time Between Failures - MTBF) e o tempo médio para reparo (Medium Time To Repair - MTTR). Por exemplo, um MTBF de 99,5% todo dia significa que o "downtime" (tempo em que a rede fica inoperante) não pode exceder 7,2 minutos cada 24 horas.

III.2.3 – Tempo de Resposta

- Definição: Mede o tempo para completar um pedido para um cliente, um grupo de clientes, ou a rede.
- Valor: Boa medida da eficiência da rede e do servidor. Algumas aplicações, tais como a Web, requerem tempos de resposta muito curtos enquanto outras, tais como transferência de arquivos, não são tão restritivas. O tempo de resposta é a melhor medida de serviço para o usuário final.
- Limitações: Difícil de medir. Aplicações mais novas podem fazer diversas interações cliente/servidor para terminar um pedido. Além disso, cada tipo de transação pode ter seus próprios perfis e comportamentos. É também difícil de localizar um problema quando os tempos de resposta são insatisfatórios - o problema poderia ser a rede, servidor, cliente, aplicação, localização da informação, ou uma combinação daqueles fatores.

III.2.4 - Vazão

- Definição: Uma medida da quantidade de dados (ou de volume) emitidos em uma quantidade de tempo dada. Por exemplo, a videoconferência pode requerer 384 Kbps a fim de fornecer a qualidade satisfatória. Descarregar uma página de texto de um servidor web em dois segundos requer uma vazão de 20 Kbps.
- Valor: Facilita o planejamento e a análise em tempo real do comportamento da rede.

Unidade 4 – Métricas de Processo e Projeto de Software

IV.1 – Medidas, Métricas e Indicadores

Medidas nos fornecem uma indicação quantitativa da extensão, quantidade, dimensão, capacidade ou tamanho de algum atributo de um produto ou processo. Já *medição* é o ato de determinação de uma medida. O IEEE Standard Glossary of Software Engineering Terms define *métrica* como "medida quantitativa do grau em que um sistema, componente ou processo possui determinado atributo".

Quando dados de um único ponto são coletados (por exemplo, a quantidade de erros descobertos na revisão de um único módulo), uma medida é estabelecida. Medição ocorre como resultado da coleta de um ou mais pontos (por exemplo, um certo número de revisões de módulos são efetuadas para coletar medidas da quantidade de erros para cada uma). Uma métrica de software relaciona as medidas individuais de alguma forma (por exemplo, um número médio de erros encontrado por revisão).

Um *indicador* é uma métrica, ou combinação de métricas, que fornece a compreensão de um processo de software, de um projeto de software, ou do produto propriamente dito. Um indicador fornece compreensão que possibilita ao gerente de projeto ou aos engenheiros de software ajustar o processo, projeto ou produto para tornar as coisas melhores.

A métrica fornece compreensão ao gerente, e compreensão, leva a uma tomada de decisão bem informada.

IV.2 – Métricas nos Domínios do Processo e do Projeto.

A medição é comum no mundo da engenharia. Medimos consumo de energia, peso, dimensões físicas, temperatura, voltagem, relação entre sinal e ruído, etc. Na engenharia de software a medição é pouco comum pois tem-se dificuldade em concordar quanto ao que medir e dificuldade em avaliar as medidas que são coletadas.

Indicadores de processo permitem a organização ter uma idéia da eficácia de um processo existente. Eles permitem aos profissionais avaliar o que funciona e o que não funciona. Com a coleta de métricas de processo ao longo de vários períodos, é possível fornecer indicadores que levem ao aperfeiçoamento do processo de software a longo prazo.
Indicadores de projeto permitem ao gerente do projeto:

- Avaliar o status de um projeto em andamento;
- Acompanhar riscos potenciais;
- Descobrir áreas de problema antes que se tornem críticas;
- Ajustar o fluxo de trabalho ou tarefas; e
- Avaliar a capacidade da equipe de projeto de controlar a qualidade dos produtos do trabalho de software.

IV.2.1 – Métricas de processo e aperfeiçoamento do processo de software

A única maneira racional de se aperfeiçoar qualquer processo é medir os atributos específicos, desenvolver um conjunto de métricas significativas, baseadas nesses atributos, e depois usar as métricas para fornecer indicadores, que levarão a uma estratégia de aperfeiçoamento.

Para os diversos tipos de dados de um processo, haverá sempre usos "públicos e privados". O uso de métricas coletadas com base individual desses dados, devem ser privados e servir como indicador, apenas para o indivíduo. Como exemplo de *métricas privadas* podemos citar a proporção de defeitos (por indivíduo e por módulo) e erros encontrados durante o desenvolvimento.

Algumas métricas de processo são privadas para a equipe de projeto de software, mas são públicas para todos os membros da equipe. Exemplos incluem defeitos encontrados para funções importantes do software (que foram desenvolvidas por um certo número de profissionais), erros encontrados durante revisões técnicas formais, etc. Esses dados são revisados pela equipe para descobrir indicadores que permitam melhorar seu desempenho.

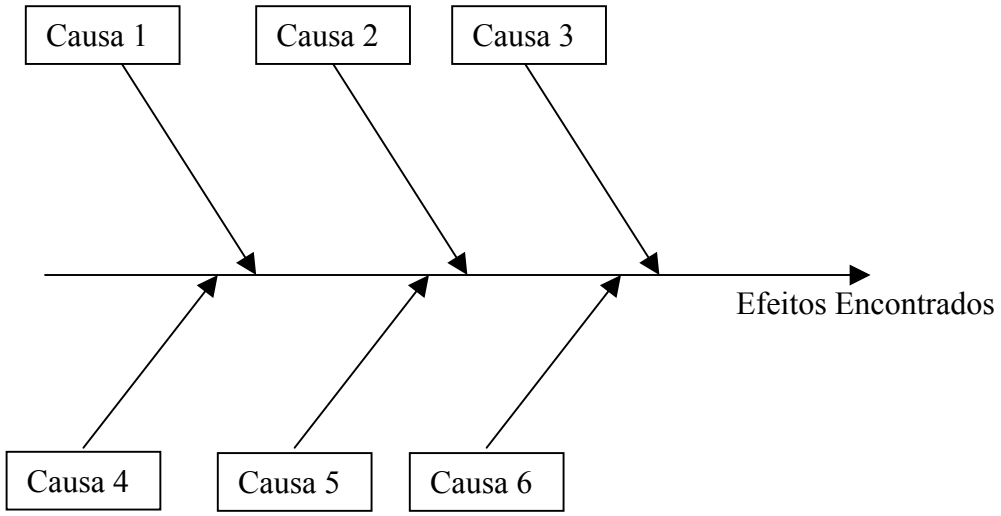
Métricas públicas geralmente assimilam informações, que eram originalmente privadas de indivíduos e de equipes. Proporções de defeitos de projeto, esforço, tempo transcorrido e dados relacionados são coletados e avaliados numa tentativa de descobrir indicadores, que possam aperfeiçoar o desempenho do processo organizacional.

Com a continuidade do uso e coleta de métricas de processo em uma organização, a derivação de indicadores simples dá lugar a uma abordagem mais rigorosa, chamada *melhoria estatística do processo de software (statistical software process improvement,* SSPI). Essencialmente, a SSPI usa uma análise das falhas de software para coletar informação sobre todos os erros e defeitos encontrados à medida que uma aplicação, sistema ou produto é desenvolvido e usado. A análise de falhas funciona da seguinte maneira:

1. Todos os erros e defeitos são categorizados por origem (falha de especificação, falha de lógica, não atendimento a padrões, etc.).
2. O custo para corrigir cada erro e defeito é registrado.
3. A quantidade de erros e defeitos de cada categoria é contada e ordenada de forma decrescente.
4. O custo total de erros e defeitos de cada categoria é calculado.
5. Os dados resultantes são analisados, para descobrir as categorias que produzem maior custo para a organização.
6. São desenvolvidos planos para modificar o processo, com o objetivo de eliminar (ou apenas reduzir a freqüência das) classes de erros e defeitos que são mais dispendiosas.

A criação de um diagrama espinha de peixe (diagrama de causas e efeito) possibilita coletar métricas de processo. Um diagrama de causas e efeito (mostrado abaixo) completo pode ser

analisado para fornecer indicadores que permitirão à organização modificar seu processo para reduzir a freqüência dos erros e defeitos.



A primeira aplicação das métricas de projeto, ocorre durante a estimativa. Métricas coletadas de projetos anteriores são usadas como base, a partir das quais estimativas de esforço e tempo são feitas para o trabalho atual. À medida que o projeto prossegue, medidas de esforço e de tempo despendidos são comparadas com as estimativas originais e com o cronograma do projeto. O gerente usa esses dados para monitorar e controlar o progresso.

A medida que o trabalho técnico se inicia, outras métricas de projeto começam a ter importância. A taxa de produção, representada em termos de páginas de documentação, horas de revisão, pontos por função e linhas de código-fonte entregue, é medida.

O objetivo das métricas de projeto é duplo. Primeiro, essas métricas são usadas para minimizar o cronograma de desenvolvimento, fazendo os ajustes necessários para evitar atrasos e problemas, e riscos em potencial. Segundo, métricas de projeto são usadas para avaliar a qualidade do produto durante sua evolução e, quando necessário, modificar a abordagem técnica para aperfeiçoar a qualidade.

À medida que a qualidade é aperfeiçoada, os defeitos são minimizados, e, à medida que a contagem de defeitos decresce, a quantidade de retrabalho durante o projeto é também reduzida. Isso levará a diminuição do custo total do projeto.

IV.3 – Medição de Software

As medições do mundo físico podem ser categorizadas de dois modos: medidas diretas (o comprimento de um parafuso, por ex.) e medidas indiretas (a qualidade dos parafusos produzidos, medida pela contagem dos refugos, por ex.). As métricas de software podem ser categorizadas de maneira análoga.

Medidas diretas incluem custo e esforço aplicados. Medidas diretas do produto incluem linhas de código (linus of code, LOC) produzidas, velocidade de execução, tamanho de memória e defeitos relatados durante um certo período.

Medidas Indiretas do produto incluem funcionalidade, qualidade, complexidade, eficiência, confiabilidade, manutenibilidade entre outras habilidades.

O custo e esforço necessários para construir software, o número de linhas de código produzidas são relativamente fáceis de coletar, desde que convenções específicas para a medição sejam estabelecidas antecipadamente. Todavia, a qualidade e a funcionalidade do software, ou sua eficiência, ou manutenibilidade são mais difíceis de avaliar e podem ser medidas apenas indiretamente.

IV.3.1 – Métricas orientadas a tamanho

Estas métricas são originadas pela normalização das medidas de qualidade e/ou produtividade, considerando o tamanho do software que foi produzido. Se uma organização mantém registros simples, podemos criar uma tabela de medidas orientadas a tamanho, tal como mostrado na figura abaixo.

Projeto	LOC	Esforço (Pess/Mes)	Custo em R$ 1.000	Páginas de Doc.	Erros	Defeitos	Pessoas
Alfa	12.100	24	168	365	134	29	3
Beta	27.200	62	440	1.224	321	86	5
Gama	20.200	63	314	1.050	256	64	6
♦	♦	♦	♦	♦	♦	♦	♦
♦	♦	♦	♦	♦	♦	♦	♦

Deve-se notar que o esforço e o custo registrados na tabela referem-se a todas as atividades de engenharia de software (análise, projeto, código e teste) e não apenas à codificação.

Pela sua facilidade de integração com outras métricas de projeto, LOC são normalmente escolhidas como valor de normalização. A partir dos dados contidos na tabela, um conjunto de métricas simples orientadas a tamanho pode ser desenvolvido para cada projeto:

- Erros por KLOC (milhares de linhas de código);
- Defeitos por KLOC;
- R$ por KLOC;
- Páginas de documentação por KLOC.

Adicionalmente, outras métricas interessantes poderiam ser calculadas:
- Erros por pessoa-mês;
- LOC por pessoa-mês;
- R$ por página de documentação.

Métricas orientadas a tamanho são muito bem aceitas como modo de medir o processo de desenvolvimento de software. A maior parte das controvérsias, porém, giram em torno do uso de linhas de código como medidas de base.

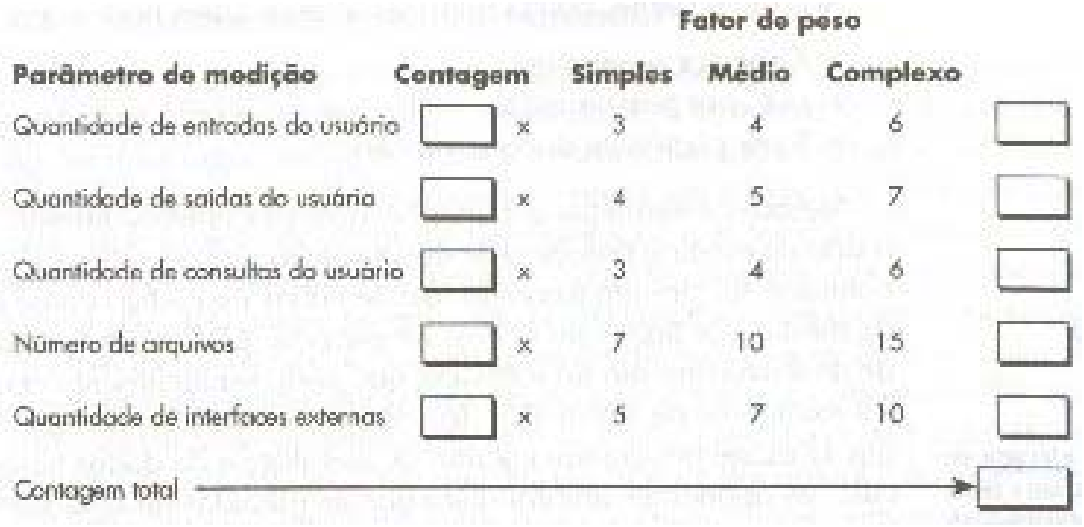
Os que são favoráveis a adoção de tal medida alegam que LOC é um "artefato" de todos os projetos de desenvolvimento de software, que pode ser facilmente contado, que muitos modelos de estimativas existentes usam LOC ou KLOC como entrada chave e que já existe um grande volume de literatura e de dados baseados em LOC.

Os que são contrários argumentam que as medidas de LOC são dependentes da linguagem de programação, que elas penalizam os programas curtos mas bem projetados, que não podem acomodar facilmente linguagens não procedimentais e que seu uso na estimativa requer um nível de detalhe que pode ser difícil de alcançar (ou seja, o planejador deve estimar o LOC a ser produzido, muito antes que a análise e o projeto tenham sido completados).

IV.3.1 – Métricas orientadas a função

Tais métricas usam uma medida da funcionalidade entregue pela aplicação como valor de normalização. Como a "funcionalidade" não pode ser medida diretamente, deve ser originada indiretamente usando outras medidas diretas. Métricas orientadas a função foram inicialmente propostas por A. J. Albretch na década de 70, que sugeriu uma medida chamada ponto por função (ou como alguns denominam, pontos de função). Estes são originados usando uma relação empírica baseadas em medidas de contagem (direta) do domínio de informação do software e da avaliação da complexidade do software.

Pontos por Função são calculados completando-se a tabela mostrada a seguir.



			Fator de peso			
Parâmetro de medição	**Contagem**		**Simples**	**Médio**	**Complexo**	
Quantidade de entradas do usuário		x	3	4	6	
Quantidade de saídas do usuário		x	4	5	7	
Quantidade de consultas do usuário		x	3	4	6	
Número de arquivos		x	7	10	15	
Quantidade de interfaces externas		x	5	7	10	
Contagem total						

Cinco características do domínio da informação são determinadas e as contagens são registradas nos lugares próprios da tabela. Os valores do domínio da informação são definidos da seguinte maneira:

- **Quantidade de entradas do usuário.** Cada entrada do usuário, que fornece dados distintos orientados da aplicação do software, é contada. Entradas devem ser distinguidas de consultas, que são contadas separadamente.
- **Quantidade de saídas do usuário.** Cada saída do usuário, que fornece informação orientada a aplicação para o usuário é contada. Nesse contexto, saída refere-se a relatórios, telas, mensagens de erro, etc. Itens de dados individuais dentro de um relatório não são contados separadamente.
- **Número de consultas do usuário.** Uma consulta é definida como uma entrada on-line, que resulta na geração de uma resposta imediata do software sob a forma de uma saída on-line. Cada saída distinta é contada.
- **Quantidade de arquivos.** Cada arquivo mestre lógico, isto é, grupo de dados lógico, que pode ser parte de uma base de dados maior ou um arquivo separado, é contado.
- **Quantidade de interfaces externas.** Todas as interfaces em linguagem de máquina (por ex. arquivos de dados em meio de armazenamento), que são usadas para transmitir informações a outro sistema, são contadas.

Uma vez coletados esses dados, um valor de complexidade é associado com cada contagem. Para contar os pontos por função (*Function Points*, FP), é usada a seguinte relação:

$$\mathbf{FP = total\ de\ contagem\ X\ [0{,}65 + 0{,}01\ X\ \Sigma(F_i)]}$$

em que total da contagem é a soma de todas as entradas de FP obtidas da figura.

Os F_i (i =1 a 14) são "valores de ajuste de complexidade", baseados nas respostas as seguintes perguntas:

1. O sistema requer salvamento (*backup*) e recuperação (*restore*)?
2. Comunicações de dados são necessárias?
3. Há funções de processamento distribuído?
4. O desempenho é crítico?
5. O sistema vai ser executado em um ambiente operacional existente, intensamente utilizado?
6. O sistema requer entrada de dados on-line?
7. A entrada de dados on-line exige que a transação de entrada seja construída através de várias telas ou operações?
8. Os arquivos mestre são atualizados on-line?
9. As entradas, saídas, arquivos ou consultas são complexas?
10. O processamento interno é complexo?
11. O código é projetado para ser reutilizado?
12. A conversão e a instalação estão incluídas no projeto?
13. O sistema está projetado para instalações múltiplas em diferentes organizações?
14. A aplicação está projetada para facilitar modificações e para facilidade de uso pelo usuário?

Cada uma dessas questões é respondida usando uma escala que varia entre 0 (não-importante ou não-aplicável) a 5 (absolutamente essencial). Os valores constantes na equação acima e os fatores de peso, que são aplicados as contagens do domínio de informação, são determinados de forma empírica.

Uma vez calculados, os pontos por função são usados de modo analógico à LOC, como forma de normalizar medidas de produtividade, qualidade e outros atributos de software:

- Erros por FP;
- Defeitos por FP;
- R$ por FP;
- Páginas de documentação por FP;
- FP por pessoa-mês.

IV.4 – Reconciliação de Diferentes Abordagens de Métricas.

A relação entre linhas de código e pontos de função depende da linguagem de programação que é usada para implementar o software de qualidade de projeto. Vários estudos tetam relacionar as medidas de FP e LOC.

A tabela abaixo fornece estimativas aproximadas do número médio de linhas de código necessárias para construir um ponto por função em várias linguagens de programação:

Linguagem de Programação	**LOC/FP (média)**
Linguagem de Máquina	320
C	128
COBOL/FORTRAN	106
Pascal/Clipper	90
C++/Java	64
Visual Basic/Delphi	32
Powerbuilder	16
SQL	12

As medidas de LOC e FP são freqüentemente usadas para originar métricas de produtividade. Isso leva, invariavelmente, ao debate sobre o uso de tais dados. Métricas baseadas em FP e LOC tem sido consideradas relativamente precisas para prever esforço e custo de desenvolvimento de software.

Unidade 5 – Medidas da Qualidade de Software

A principal meta da engenharia de software é produzir um sistema, aplicação ou produto de alta qualidade. A qualidade de um sistema, aplicação ou produto não é melhor que os requisitos que descrevem o problema, o projeto que modela a solução, o código que leva a um programa executável e os testes que exercitam o software para descobrir erros.

V.1 – Medição de Qualidade

Apesar das várias medidas de qualidade de software, a correção, a manutenibilidade, a integridade e a utilização fornecem indicadores úteis a equipe de software. A seguir, sugerimos definições e medidas para cada uma.

- **Correção.** Um programa precisa operar corretamente ou é de pouco valor para seus usuários. Correção é o grau em que o software desempenha sua necessária função. A medida mais comum de correção é *defeitos por KLOC*, onde defeito é definido como uma falha verificada de obediência aos requisitos. Quando se considera a qualidade geral de um produto de software, defeitos são problemas relatados por um usuário do programa, após o programa ter sido entregue para uso geral. Para fins de avaliação de qualidade, defeitos são contados durante o período de um ano.
- **Manutenibilidade.** Manutenção de software consome mais esforço do que qualquer outra atividade de engenharia de software. Manutenibilidade é a facilidade com que um programa pode ser corrigido, se um erro é encontrado, adaptado, se seu ambiente se modifica, ou aperfeiçoado, se o cliente deseja uma modificação nos requisitos. Não há modo de medir manutenibilidade diretamente; assim precisamos usar medidas indiretas. Uma métrica simples orientada a tempo é tempo médio para modificação (*mean-time-to-change*, MTTC), que é o tempo despendido para analisar o pedido de modificação, projetar uma modificação adequada, implementar a modificação, testá-la e distribuí-la para todos os usuários. Em média, programas manuteníveis vão ter um MTTC mais baixo (para tipos equivalentes de modificação) do que programas não-manuteníveis.
- **Integridade.** A integridade de software tornou-se cada vez mais importante na era dos hackers e firewalls. Esse atributo mede a capacidade do sistema de resistir a ataques (tanto acidentais quanto intencionais) à sua segurança. Os ataques podem ser feitos aos três componentes do software: programas, dados e documentos. Para medir a integridade, dois atributos adicionais precisam ser definidos: ameaça e segurança. *Ameaça* é a probabilidade (que pode ser estimada ou originada de experiência empírica) de um ataque, específico, ocorrer dentro de um certo período. *Segurança* é a probabilidade (que, também, pode ser estimada ou originada de experiência empírica) de um ataque ser repelido. A integridade do sistema pode então ser assim definida (ameaça e segurança devem ser somadas por tipo de ataque):

integridade = somatório [(1 – ameaça) X (1 – segurança)]

- Utilização. A expressão "uso amigável" tornou-se onipresente na discussão de produtos de software. Se um programa não é de uso amigável, freqüentemente, está destinado ao fracasso, mesmo se as funções que ele desempenha forem valiosas. Utilização é uma tentativa de quantificar a característica de ser amigável ao uso e pode ser medida em função de quatro tópicos:
  - a aptidão física e/ou intelectual necessária para aprender a lidar com o sistema;
  - o tempo necessário para se tornar moderadamente eficiente no uso do sistema;
  - o aumento líquido de produtividade (relativo a abordagem que o sistema substitui), medido quando o sistema é usado por alguém moderadamente eficiente; e
  - uma avaliação subjetiva (algumas vezes obtida por intermédio de um questionário) das atitudes dos usuários com relação ao sistema.

V.2 – Eficiência na remoção de defeitos

Uma métrica de qualidade, que fornece benefícios tanto para projetos quanto para processos, é a eficiência na remoção defeitos (*defect removal efficiency*, DRE). A DRE é uma medida da capacidade de filtragem das atividades de controle e garantia de qualidade de software, à medida que são aplicadas as atividades de arcabouço de processo. Quando considerada para o projeto todo, a DRE é definida da seguinte maneira:

$$\mathbf{DRE = E / (E + D)}$$

em que, E é a quantidade de erros encontrados antes da entrega do software ao usuário final e D é a quantidade de defeitos encontrados após a entrega.

O valor ideal da DRE é 1. Isto é, nenhum defeito é encontrado no software. À medida que E aumenta (para um dado valor de D), o valor da DRE se aproxima de 1. De fato, à medida que E aumenta, é provável que o valor final de D diminua (os erros são filtrados antes que se transformem em defeitos). Se usada como métrica, que fornece indicador da capacidade de filtragem das atividades de garantia e controle de qualidade, a DRE motiva uma equipe de projeto de software a instituir técnicas para encontrar tantos erros quanto possível antes da entrega.

V.3 – Normas

Muitas instituições se preocupam em criar Normas para permitir a correta avaliação de Qualidade tanto de produtos de Software quanto de processos de desenvolvimento de Software. Apenas para ter uma visão geral, observe o quadro abaixo com as principais Normais nacionais e internacionais nesta área:

Norma	Comentário
ISO 9126	Características da Qualidade de produtos de Software.
NBR 13596	Versão brasileira da ISO 9126
ISO 14598	Guias para a avaliação de produtos de Software, baseados na utilização prática da Norma ISSO 9126
ISO 12119	Características de Qualidade de pacotes de Software (Software de prateleira, vendido com um produto embalado)
IEEE P1061	Standard for Software *Quality Metrics Methodology* (produto de Software)
ISO 12207	Software *Life Cycle Process*. Norma para a Qualidade do processo de desenvolvimento de Software.
NBR ISO 9001	Sistemas de Qualidade – Modelo para garantia de Qualidade em Projeto, Desenvolvimento, Instalação e Assistência Técnica (processo)
NBR ISO 9000-3	Gestão de Qualidade e garantia de Qualidade. Aplicação da Norma ISO 9000 para o processo de desenvolvimento de Software.
NBR ISO 10011	Auditoria de Sistemas de Qualidade (processo)
CMM	*Capability Maturity Model*. Modelo da SEI (Instituto de Engenharia de Software do Departamento de Defesa dos EEUU) para avaliação da Qualidade do processo de desenvolvimento de Software. Não é uma Norma ISO, mas é muito bem aceita no mercado.
SPICE ISO 15504	Projeto da ISO/IEC para avaliação de processo de desenvolvimento de Software. Ainda não é uma Norma oficial ISO, mas o processo está em andamento.

V.4 - Qualidade de Produtos de Software - ISO 9126

A Norma ISO/IEC 9126 representa a atual padronização mundial para a Qualidade de produtos de Software. Ela é uma das mais antigas da área de Qualidade de Software e já possui sua tradução para o Brasil, publicada como NBR-13596. Esta norma lista o conjunto de características que devem ser verificadas para se considerar um Software de Qualidade. São 6 grandes grupos de características, cada um dividido em algumas sub-características, conforme a tabela abaixo (Norma ISO/IEC 9126):

Característica	Sub-característica	Pergunta chave para a sub-característica
Funcionalidade (satisfaz as necessidades?)	Adequação	Propõe-se a fazer o que é apropriado?
	Acurácia	Faz o que foi proposto de forma correta?
	Interoperbilidade	Interage com os sistemas especificados?
	Conformidade	Está de acordo com as Normas, leis, etc.?
	Segurança de acesso	Evita acesso não autorizado aos dados?
Confiabilidade (é imune a falhas?)	Maturidade	Com que freqüência apresenta falhas?
	Tolerância a falhas	Ocorrendo falhas, como ele reage?
	Recuperabilidade	É capaz de recuperar dados em caso de falha?
Usabilidade (é fácil de usar?)	Intelegibilidade	É fácil entender o conceito e a aplicação?
	Apreensibilidade	É fácil aprender a usar?
	Operacionalidade	É fácil de operar e controlar?
Eficiência (é rápido e "enxuto"?)	Tempo	Qual é o tempo de resposta, a velocidade de execução?
	Recursos	Quanto recurso usa? Durante quanto tempo?
Manutenibilidade (é fácil de modificar?)	Analisabilidade	É fácil de encontrar uma falha, quando ocorre?
	Modificabilidade	É fácil modificar e adaptar?
	Estabilidade	Há grande risco quando se faz alterações?
	Testabilidade	É fácil testar quando se faz alterações?
Portabilidade (é fácil de usar em outro ambiente?)	Adaptabilidade	É fácil adaptar a outros ambientes?
	Capacidade para ser instalado	É fácil instalar em outros ambientes?
	Conformidade	Está de acordo com padrões de portabilidade?
	Capac. Para substituir	É fácil usar para substituir outro?

Embora a atual Norma ISO/IEC 9126 (ou NBR 13596) enumere as características e sub-características um Software, ela ainda não define como dar uma nota a um Software em cada um destes itens.

Algumas características podem ser realmente medidas, como o tempo de execução de um programa, número de linhas de código, número de erros encontrados em uma sessão de teste ou o tempo médio entre falhas. Nestes casos, é possível utilizar uma técnica, uma ferramenta ou um Software para realizar medições. Em outros casos, a característica é tão subjetiva que não existe nenhuma forma óbvia de medí-la.

Ficam, portanto, as questões: como dar uma nota, em valor numérico, a uma característica inteiramente subjetiva? O que representa, por exemplo, uma nota 10 em termos de Segurança de Acesso? Quando se pode dizer que a inteligibilidade de um Software pode ser considerada satisfatória? Para isso, criou-se, então, uma área de estudo à parte dentro da Qualidade de Software: as Métricas de Software. O que se pretende fazer é definir, de forma precisa, como medir numericamente uma determinada característica.

Para avaliar uma determinada sub-característica subjetiva de forma simplificada, por exemplo, pode-se criar uma série de perguntas do tipo “sim ou não”. Cria-se as perguntas de forma tal que as respostas “sim” sejam aquelas que indicam uma melhor nota para a característica. Depois de prontas as perguntas, basta avaliar o Software, respondendo a cada pergunta. Se você conseguir listar 10 perguntas e o Software obtiver uma resposta "sim" em 8 delas, terá obtido um valor de 80% nesta característica.

Obviamente, a técnica acima não é muito eficiente. Para melhorá-la, entretanto, pode-se garantir um número mínimo de perguntas para cada característica. Além disso, algumas perguntas mais importantes podem ter pesos maiores. É possível, ainda, criar perguntas do tipo ABCDE, onde cada resposta indicaria um escore diferenciado. Alguns estudiosos sugerem formas diferentes de medir uma característica, baseada em conceitos do tipo "não satisfaz", "satisfaz parcialmente", "satisfaz totalmente" e "excede os padrões". Estes conceitos, embora pareçam muito subjetivos, não deixam de ser uma forma eficiente de medir uma característica.

Em todos os casos, um fato fica claro: nada ajuda mais a avaliar características de um Software do que um avaliador experiente, que já realizou esta tarefa diversas vezes e em diversas empresas diferentes. Afinal, medir é comparar com padrões e um avaliador experiente terá maior sensibilidade do que um profissional que acaba de ler uma Norma pela primeira vez.

Unidade 6 – Técnicas de Teste de Software

Teste de software é um elemento crítico da garantia de qualidade de software e representa a revisão final da especificação, projeto e geração de código.

Não é raro um empresa de desenvolvimento de software gastar entre 30 e 40% do esforço total do projeto no teste. O teste de software que envolve vidas (controle de vôo, monitoramento de reatores nucleares) pode custar de 3 a 5 vezes mais do que todos os outros passos de engenharia de software combinados.

VI.1 – Fundamentos do Teste de Software

O Engenheiro de Software é exposto a uma anomalia interessante: durante as primeiras atividades desta engenharia, o profissional tenta construir, a partir de um conceito abstrato um produto tangível. Depois, com o teste, o engenheiro cria uma série de casos destinados a demolir o que foi construído. O teste pode ser visto como um passo destrutivo no processo de software.

OBJETIVOS DO TESTE

1 – Teste é um processo de execução de um programa com a finalidade de encontrar erros.
2 – Um bom caso de teste é aquele que tem alta probabilidade de encontrar um erro ainda não descoberto.
3 – Um teste bem sucedido é aquele que descobre um erro ainda não descoberto.

Esses objetivos implicam na mudança do ponto de vista de que um teste bem sucedido é aquele no qual não são encontrados erros. A meta é projetar testes que descobrem sistematicamente diferentes classes de erros e fazê-lo com uma quantidade mínima de tempo e esforço.

Se o teste for conduzido de acordo com os objetivos declarados anteriormente, ele descobrirá erros no software. Os dados coletados a medida que o teste é conduzido fornecem uma boa indicação da confiabilidade do software e alguma indicação da qualidade de todo o software. Mas o teste não pode mostrar ausência de erros e defeitos, mas apenas mostrar que eles estão presentes.

PRINCÍPIOS DO TESTE

Antes de aplicar métodos para projetar casos de teste efetivos, um engenheiro de software deve entender os princípios básicos que guiam o teste de software.

- **Todos os testes devem ser relacionados aos requisitos do cliente.** Os defeitos mais indesejáveis (do ponto de vista do cliente) são aqueles que levam o programa a deixar de satisfazer seus requisitos.

- **Os testes devem ser planejados muito antes do início do teste.** O planejamento de teste pode começar tão logo o modelo de requisitos seja completado. Assim sendo, eles podem ser planejados e projetados antes que qualquer código tenha sido gerado.
- **O princípio de Pareto se aplica ao teste de software.** Colocado simplesmente, tal princípio implica que 80% de todos os erros descobertos durante o teste vão, provavelmente, ser relacionados a 20% de todos os componentes do programa. O problema é isolar os componentes suspeitos e testá-los rigorosamente.
- **O teste deve começar "no varejo" e progredir até o teste "no atacado".** Os primeiros testes planejados e executados concentram-se nos componentes individuais. À medida que o teste progride, o foco se desloca numa tentativa de encontrar erros em conjuntos integrados de componentes e, finalmente, em todo o sistema.
- **Teste completo não é possível.** A quantidade de permutações de caminhos, mesmo para um programa de tamanho moderado, é excepcionalmente grande. Por essa razão é impossível executar todas as combinações de caminhos durante o teste. No entanto, é possível cobrir adequadamente a lógica do programa e garantir que todas as condições no projeto tenham sido executadas (pelo menos no nível de componente).
- **Para ser mais efetivo, o teste deveria ser conduzido por terceiros.** Ser mais efetivo significa ter maior probabilidade de encontrar erros (o principal objetivo do teste). Desta forma, o engenheiro que criou o software não é a pessoa adequada para conduzir testes que encontrarão erros em sua criação.

TESTABILIDADE

Trata-se da facilidade com que o software pode ser testado. Há certas métricas que podem ser usadas para medir a testabilidade na maior parte de seus aspectos.

A lista a seguir fornece um conjunto de características que levam a um software testável:

Operabilidade: "Quanto melhor funciona, mais eficientemente pode ser testado".
- O sistema tem poucos defeitos (bugs);
- Nenhum defeito bloqueia a execução dos testes; e
- O produto evolui em estágios funcionais (permite desenvolvimento e testes simultâneos).

Observabilidade: "O que você vê é o que você testa"
- Uma saída distinta é gerada para cada entrada;
- Variáveis do sistema são consultáveis durante a execução;
- Os estados e variáveis anteriores do sistema são visíveis ou consultáveis (por exemplo registro de transações);
- Todos os fatores que afetam as saídas são visíveis;
- Saída incorreta é facilmente identificada;
- Erros internos são identificados através de mecanismos de auto teste;
- Erros internos são automaticamente relatados;
- O código-fonte é acessível.

Controlabilidade: "Quanto mais você pode controlar o software, mais o teste pode ser automatizado e otimizado"

- Todas as saídas possíveis podem ser geradas por alguma combinação de entradas;
- Todo o código é executável através de uma combinação de entradas;
- Estados e variáveis do software podem ser controladas diretamente pelo condutor do teste;
- Formatos de entradas e saídas são consistentes e estruturados;
- Testes podem ser especificados, automatizados e reproduzidos convenientemente.

Decomponibilidade: "Controlando o escopo do teste, podemos isolar problemas mais rapidamente e realizar retestagem mais racionalmente".

- O sistema de software é construído a partir de módulos independentes;
- Módulos de software podem ser testados independentemente.

Simplicidade: "Quanto menos há a testar, mais rapidamente podemos testá-lo".

- Simplicidade funcional (o conjunto de características é o mínimo necessário para satisfazer os requisitos);
- Simplicidade Estrutural (a arquitetura é modularizada para limitar a propagação de defeitos);
- Simplicidade do código (um padrão de código é adotado para facilitar a inspeção e a manutenção);

Estabilidade: "Quanto menos modificações, menos interrupções no teste"

- Modificações no software não são freqüentes;
- Modificações no software são controladas;
- Modificações no software não invalidam os testes existentes;
- O software se recupera bem das falhas.

Compressibilidade: "Quanto mais informação temos, mais racionalmente vamos testar".

- O projeto é bem compreendido;
- As dependências entre componentes internos, externos e compartilhados são bem compreendidas;
- Modificações no projeto são informadas;
- A documentação técnica é acessível instantaneamente;
- A documentação técnica é bem organizada;
- A documentação técnica é específica e detalhada;
- A documentação técnica é precisa.

Quais são os atributos de um "bom" teste?

1. Um bom teste tem alta probabilidade de encontrar um erro.
2. Um bom teste não é redundante. Cada teste deve ter uma finalidade diferente.
3. Devemos utilizar o teste que tenha a maior probabilidade de revelar toda uma classe de erros.
4. Um bom teste não deve ser muito simples nem muito complexo.

VI.2 – Projeto de Casos de Teste

Qualquer produto, fruto da engenharia, pode ser testado por uma das duas maneiras:

(1) Sabendo a função especificada que o produto foi projetado para realizar, podem ser realizados testes que demonstram que cada função está plenamente operacional, enquanto ao mesmo tempo procuram erros em cada função.
(2) Sabendo como é o trabalho interno de um produto, podem ser realizados testes para garantir que "todas as engrenagens combinam", isto é, que as operações internas são realizadas de acordo com as especificações e que todos os componentes internos foram adequadamente exercitados.

A abordagem (1) é chamada de Teste da Caixa-Preta e a (2), Teste da Caixa-Branca.

Quando o realizamos testes caixa-preta em softwares, nos referimos aos testes que são conduzidos na interface. Apesar de serem projetados para descobrir erros, eles são usados para demonstrar que as funções do software estão operacionais, que a entrada é adequadamente aceita e a saída é corretamente produzida, e que a integridade da informação externa (uma base de dados por ex.) é mantida. Um teste caixa-preta examina algum aspecto fundamental do sistema, se preocupando pouco com a estrutura lógica interna do software.

Um teste caixa-branca é baseado num exame rigoroso dos detalhes procedimentais. Caminhos lógicos internos são testados, definindo casos de testes que exercitam conjuntos específicos de condições ou ciclos. O "estado do programa" pode ser examinado em vários pontos para determinar se o estado esperado ou enunciado corresponde ao estado real.

Observações:

1) Testes Caixa-Branca só podem ser projetados depois que o projeto a nível de componente (código-fonte) existe. Os detalhes lógicos do programa devem estar disponíveis.
2) Não é possível testar completamente todos os caminhos do programa, porque o número de caminhos é simplesmente grande demais.

TESTE CAIXA-BRANCA

É um método de projeto de casos de teste que usa a estrutura de controle do projeto procedimental para derivar casos de teste. Usando métodos de teste caixa-branca, o engenheiro de software pode derivar casos de teste que (1) garantam que todos os caminhos independentes de um módulo tenham sido exercitados pelo menos uma vez, (2) exercitam todas as decisões lógicas em seus lados verdadeiro e falso, (3) executam todos os ciclos nos seus limites e dentro de seus intervalos operacionais e (4) exercitam as estruturas de dados internas para garantir sua validade.

ARGUMENTOS PARA A CONDUÇÃO DE TESTES CAIXA-BRANCA.

- *Erros lógicos e pressupostos incorretos são inversamente proporcionais à probabilidade de que um caminho de programa vai ser executado* → um processamento cotidiano tende a ser bem entendido (e bem examinado), enquanto um processamento de casos especiais tende a ser negligenciado.
- *Freqüentemente acreditamos que um caminho lógico não é provável de ser executado quando, na realidade, ele pode ser executado em base regular* → o fluxo lógico de um programa é algumas vezes contra-intuitivo, ou seja, nossos pressupostos sobre o fluxo de controle e dados podem nos levar a cometer erros de projeto que são descobertos apenas quando o teste de caminhos começa.
- *Erros tipográficos são aleatórios* → quando um programa é traduzido em código-fonte, numa linguagem de programação, é provável que ocorram alguns erros de digitação. Muitos serão descobertos por mecanismos de correção de sintaxe e ortografia, mas outros podem continuar não detectados até que o teste comece. O erro tipográfico pode existir tanto num caminho lógico obscuro quanto num caminho principal.

TESTES DE CAMINHO BÁSICO E ESTRUTURAS DE CONTROLE

São técnicas de teste caixa-branca que permitem ao projetista de casos de teste criar uma medida da complexidade lógica de um projeto procedimental e usar essa medida como guia para definir um conjunto básico de caminhos de execução. Tais casos de teste executam, garantidamente, cada comando do programa pelo menos uma vez durante o teste. Normalmente lançamos mão de um *grafo de fluxo* quando a estrutura lógica de controle de um módulo é complexa. O grafo permite traçar mais facilmente os caminhos do programa.

TESTE CAIXA-PRETA

Também chamado de *Teste Comportamental*, focaliza os requisitos funcionais do software. O teste caixa-preta permite ao engenheiro de software derivar conjuntos de condições de entrada que vão exercitar plenamente todos os requisitos funcionais de um programa. O teste-caixa preta não é uma alternativa às técnicas caixa-branca. Ao contrário, é uma abordagem complementar , que mais provavelmente descobrirá uma classe diferente de erros do que os métodos caixa-branca.

O teste caixa-preta tenta encontrar erros das seguintes categorias:
(1) Funções incorretas ou omitidas;
(2) Erros de interface;
(3) Erros de estrutura de dados ou de acesso a base de dados externa;
(4) Erros de comportamento ou desempenho; e
(5) Erros de iniciação e término.

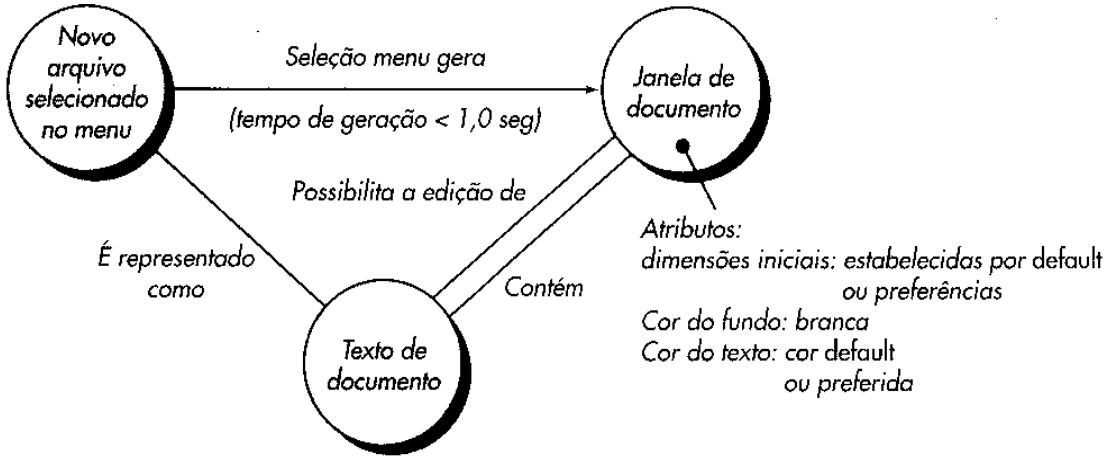
Diferente do teste caixa-branca, que é realizado no início do processo de teste, o teste caixa-preta tende a ser aplicado durante os últimos estágios do teste. Como o teste caixa-preta despreza, de propósito, a estrutura de controle, a atenção é focalizada no domínio da informação. Os testes são projetados para responder às seguintes questões:

- Como a validade funcional é testada?
- Como o comportamento e o desempenho do sistema são testados?
- Que classes de entrada vão constituir bons casos de teste?
- O sistema é particularmente sensível a certos valores de entrada?
- Como são isolados os limites de uma classe de dados?
- Que taxas de volumes de dados o sistema pode tolerar?
- Que efeito as combinações específicas de dados vão ter na operação do sistema?

MÉTODOS DE TESTE BASEADOS EM GRAFOS

O primeiro passo no teste caixa-preta é entender os objetos que estão modelados no software e as relações que conectam esses objetos. Uma vez que isso tenha sido conseguido, o passo seguinte é definir uma série de testes que verifica se todos os objetos têm uma relação esperada uns com os outros. Um grafo representa a relação entre os objetos de dados e objetos de programa, nos permitindo derivar casos de teste que procuram erros associados a essas relações.

Para executar esses passos, o engenheiro de software começa criando um grafo (vide fugura abaixo) – uma coleção de nós que representam objetos; ligações que representam as relações entre objetos; pesos de nós que descrevem as propriedades de um nó; e pesos de ligação que descrevem algumas características de uma ligação.



TESTE DE AMBIENTES, ARQUITETURAS E APLICAÇÕES ESPECIALIZADAS

- TESTE DE GUI

Interfaces gráficas com o usuário (*Graphical User Interfaces*, GUI) apresentam desafios interessantes para os engenheiros de software. Devido a componentes reusáveis fornecidos como parte de ambientes de desenvolvimento, a criação de interface com o usuário tornou-se menos demorada e mais precisa. Ao mesmo tempo, a complexidade das GUI cresceu, levando mais dificuldade no projeto e execução dos casos de teste.

- TESTE DE ARQUITETURAS CLIENTE/SERVIDOR

A natureza distribuída de ambientes cliente/servidor, os aspectos de desempenho, associados com o processamento de transações, a presença potencial de várias plataformas de hardware diferentes, as complexidades das redes de comunicação, a necessidade de atender muitos clientes por uma base de dados centralizada (ou em alguns casos distribuída) e os requisitos de coordenação impostos ao servidor, tudo se combina para tornar o teste de arquiteturas Cliente/Servidor (C/S), e do software nelas residente, consideravelmente mais difícil do que de aplicações isoladas. De fato, estudos recentes na indústria indicam um aumento significativo de tempo e custo de testes no desenvolvimento de sistemas para estas arquiteturas.

- TESTE DA DOCUMENTAÇÃO E DISPOSITIVOS DE AJUDA

É importante notar que os testes devem ser estendidos, também, para a documentação. Erros nesse elemento da configuração, podem ser tão devastadores para a aceitação do programa quanto erros no seu código-fonte. Nada é mais frustrante do que seguir exatamente um guia do usuário ou documento de ajuda on-line e obter resultados ou comportamentos que não coincidam com aqueles previstos na leitura. É por isso que o teste de documentação deve ser uma parte significativa de todo o plano de teste de software.

O teste de documentação pode ser abordado em duas fases:
a) *revisão e inspeção*, que examina o documento quanto à clareza editorial; e
b) *teste ao vivo*, que usa a documentação em conjunto com o uso do programa real.

Um teste de documentação ao vivo pode ser abordado usando técnicas que são análogas aos métodos de teste de caixa-preta discutidos anteriormente. As seguintes questões devem ser respondidas durante ambas as fases:

- A documentação descreve precisamente como escolher cada modo de uso?
- A descrição de cada seqüência de interação é precisa?
- Os exemplos são corretos?
- A terminologia, as descrições dos menus, as respostas do sistema são consistentes com o programa real?
- É relativamente fácil localizar as diretrizes na documentação?

- A correção de defeitos pode ser realizada facilmente com a documentação?
- O sumário e o índice do documento são precisos e completos?
- O projeto do documento (layout, escolha de tipos, tabulação, gráficos) conduz ao entendimento e rápida assimilação da informação?
- As mensagens de erro do software mostradas para o usuário são todas descritas em mais detalhes no documento? As ações a serem tomadas como conseqüência de uma mensagem de erro são claramente delineadas?
- Se é usado hipertexto, o projeto de navegação é apropriado a informação exigida?

O único modo de responder a essas questões é ter um parceiro independente testando a documentação no contexto de uso do programa. Todas as discrepâncias devem ser anotadas e as áreas de ambigüidade ou fraqueza do documento são definidas para potencial reescrita.

- TESTE DE SISTEMAS DE TEMPO REAL

A natureza dependente de tempo e assíncrona, de muitas aplicações em tempo real, adiciona um elemento novo e potencialmente muito difícil ao teste – tempo. O projetista de casos de teste não tem apenas que considerar casos de teste caixa-branca e caixa-preta, mas também a manipulação de eventos (processamento de interrupções, por ex.), a tempestividade dos dados e o paralelismo das tarefas (processos) que manipulam os dados. Em muitas situações, dados de teste fornecidos quando o sistema está num estado diferente podem levar a erro.

Por exemplo, o software de tempo real que controla uma copiadora aceita interrupções do operador (o operador da máquina aperta teclas de controle como RESET ou ESCURECER) sem erro, enquanto a máquina está fazendo cópias.

Além disso, a relação íntima que existe entre software de tempo real e seu ambiente de hardware pode também causar problemas de teste. Testes de software devem considerar o impacto das falhas do hardware no processamento do software. Tais falhas podem ser extremamente difíceis de simular realisticamente.

Podemos propor uma estratégia de quatro passos para os projetos de casos de testes de sistemas de tempo real:

Teste de Tarefa. O primeiro passo é testar cada tarefa independentemente. Isto é, testes caixa-branca e caixa-preta são projetados e executados para cada tarefa. Cada tarefa é executada independentemente durante os testes. O teste de tarefa descobre erros de lógica e de função, mas não de tempestividade ou de comportamento.

Teste Comportamental. Usando modelos de sistemas criados com ferramentas CASE, é possível simular o comportamento de um sistema de tempo real como conseqüência de eventos externos. Essas atividades de análise podem servir de base para o projeto de casos de testes que são conduzidos depois que o software de tempo real tiver sido construído. Cada um dos eventos é testado individualmente e o comportamento do sistema executável é

examinado para detectar erros que ocorrem como conseqüência do processamento associado a esses eventos. O comportamento do sistema (desenvolvido durante a atividade de análise) e o software executável podem ser comparados para verificar sua conformidade. Uma vez testada cada classe de eventos, os eventos serão apresentados ao sistema em ordem e freqüência aleatórias. O comportamento do software é examinado para detectar erros de comportamento.

Testes Intertarefas. Uma vez que erros em tarefas individuais e no comportamento do sistema tenham sido isolados, o teste passa para erros relativos a tempo. Tarefas assíncronas que se comunicam umas com as outras são testadas com diferentes taxas de dados e cargas de processamento para detectar se erros de sincronização intertarefas vão ocorrer. Além disso, tarefas que se comunicam por filas de mensagens ou depósito de dados são testadas para descobrir erros no tamanho dessas áreas de armazenamento.

Teste de sistema. O software e o hardware são integrados e todo um conjunto de testes de sistema é conduzido numa tentativa de descobrir erros na interface software/hardware. A maioria dos sistemas de tempo real processa interrupções. Assim o teste da manipulação desses eventos booleanos é essencial.

Usando o diagrama de transição de estados e a especificação de controle, o testador desenvolve uma lista de todas as possíveis interrupções e do processamento que ocorre como conseqüência das interrupções. Então, são projetados testes para avaliar as seguintes características do sistema:

- As prioridades de interrupção estão atribuídas adequadamente e propriamente manipuladas?
- O processamento para cada interrupção foi conduzido corretamente?
- O desempenho (por ex. tempo de processamento) de cada procedimento de manipulação de interrupção está de acordo com os requisitos?
- Um alto volume de interrupções chegando em tempos críticos cria problemas de função ou desempenho?

Além disso, áreas globais de dados que são usadas para transferir informação como parte do processamento de interrupções, devem ser testadas para avaliar o potencial de geração de efeitos colaterais.