

Manutenção de Software

Prof. Raul Sidnei Wazlawick



Manutenção

- Manutenção de software é como se denomina, usualmente, o processo de adaptação e otimização de um software já desenvolvido, bem como a correção de defeitos que ele eventualmente tenha.
- A manutenção é necessária para que um produto de software mantenha sua qualidade ao longo do tempo, já que, se isso não for feito, haverá uma deterioração do valor percebido deste software e, portanto, de sua qualidade.



Evolução

- Modernamente o termo “manutenção de software” vem sendo substituído ou utilizado em conjunto com “evolução de software”.
- *Evolução* talvez seja um termo mais adequado porque usualmente as atividades de modificação do software na fase de Produção não visam mantê-lo como está, mas fazê-lo evoluir de forma a adaptar-se a novos requisitos ou ainda a corrigir defeitos que possivelmente tenha.

Necessidade da Manutenção

- Considera-se que um software uma vez desenvolvido terá um valor necessariamente decrescente com o passar do tempo. Isso ocorre por que:
 - Falhas são descobertas.
 - Requisitos mudam.
 - Produtos menos complexos, mais eficientes ou tecnologicamente mais avançados são disponibilizados.
- Desta forma, torna-se imperativo que, simetricamente, para manter o valor percebido de um sistema:
 - Falhas sejam corrigidas.
 - Novos requisitos sejam acomodados.
 - Seja buscada simplicidade, eficiência e atualização tecnológica.



Leis de Lehman

- As assim chamadas “Leis de Lehman” (Lehman M. M., 1980) (Lehman & J. F. Ramil, 1997) procuram explicar a necessidade e a inevitabilidade da evolução de software.
- São atualmente oito leis baseadas nas observações do autor sobre os processos de evolução de sistemas.



Lehman identifica 2 tipos de sistema:

- *Tipo-S:*
 - são sistemas **especificados formalmente**, entendidos como objetos matemáticos e cuja correção em relação a uma especificação pode ser provada por ferramentas formais.
- *Tipo-E:*
 - são sistemas desenvolvidos pelos **processos usuais** de análise, projeto e codificação, que tem uso corrente em um ambiente real, isto é, são tipicamente sistemas de informação e outros sistemas não gerados por métodos formais.

1. Lei da Mudança Contínua

- Um sistema que é efetivamente usado deve ser **continuamente melhorado**, caso contrário torna-se cada vez menos útil, pois seu contexto de uso evolui.
 - Se o programa não evoluir, ele terá cada vez menos valor até que se chegue à conclusão de que vale a pena substituí-lo por outro programa.
- Programas suficientemente grandes nunca são terminados.
 - Eles simplesmente continuam a evoluir.

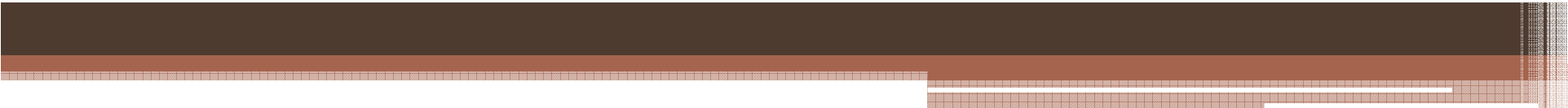
2. Lei da Complexidade Crescente

- À medida que um programa evolui, sua complexidade inerente aumenta, porque as correções feitas podem deteriorar sua organização interna.
 - Isso só não acontece quando medidas específicas de cuidado são tomadas durante as atividades de evolução, como por exemplo, a refatoração do sistema quando necessário.
- À medida que mudanças são introduzidas no software, as interações entre elementos, nem sempre previstas ou planejadas na estrutura do software farão com que a entropia interna aumente, ou seja, vai ocorrer um crescimento cada vez mais desestruturado.
 - A cada nova mudança, a estrutura interna do software se tornará menos organizada, aumentando assim, gradativamente o custo de manutenções posteriores.
 - De fato, chega-se a um ponto em que a refatoração do sistema torna-se obrigatória.



3. Lei Fundamental da Evolução de Programas: Auto regulação

- A evolução de programas é sujeita a uma dinâmica de auto-regulação que faz com que medidas globais de esforço e outros atributos de processo sejam estatisticamente previsíveis (distribuição normal).
- O desenvolvimento e manutenção de um sistema ocorrem dentro de uma organização com objetivos que se estendem muito além do sistema.
 - Então, os processos desta organização acabam regulando a aplicação de esforço em cada um de seus sistemas.
- Quaisquer processos que fujam muito ao padrão da organização são logo refatorados para se adequar, de forma que o esforço gasto nas diferentes atividades permaneça distribuído de forma normal.



4. Lei da Conservação da Estabilidade Organizacional: Taxa de trabalho invariante

- A taxa média efetiva de trabalho global em um sistema em evolução é invariante no tempo: ela não aumenta nem diminui.
- A carga de trabalho aplicada em um projeto não depende apenas de decisões da gerência.
 - Mas na prática, as demandas de usuários também influenciam nestas decisões e estas se mantêm praticamente constantes no tempo.



5. Lei da Conservação da Familiaridade: Complexidade percebida

- Durante a vida ativa de um programa, o conteúdo das sucessivas versões do programa (mudanças, adições e remoções) é estatisticamente invariante.
- Isso ocorre porque para que um sistema evolua de forma saudável, todos os agentes relacionados a ele devem manter a familiaridade com suas características e funções.
- Se o sistema crescer demais essa familiaridade é perdida e leva-se tempo para recuperá-la.
- A taxa de crescimento de um sistema é limitada pela capacidade dos indivíduos envolvidos em absorver as novidades coletivamente e individualmente.



6. Lei do Crescimento Contínuo

- O conteúdo funcional de um sistema deve crescer continuamente para manter a satisfação do usuário.
- A mudança será sempre necessária no software, seja pela correção de erros (manutenção corretiva), aperfeiçoamento de funções existentes (manutenção perfectiva) ou adaptação a novos contextos (manutenção adaptativa).



7. Lei da Qualidade Decrescente

- A qualidade de um sistema vai parecer diminuir com o tempo, a não ser que medidas rigorosas sejam tomadas para manter e adaptar o sistema.
- Mesmo que um software funcione perfeitamente por muitos anos isso não significa que continuará sempre sendo satisfatório.
 - Conforme o tempo passa, os usuários ficam mais exigentes em relação ao software e, conseqüentemente, mais insatisfeitos com ele.



8. Lei do Sistema Realimentado

- A evolução de sistemas é um processo multi-nível, multi-*loop* e multi-agente de realimentação, e deve ser encarado dessa forma para que se obtenha melhorias significativas em uma base razoável.
- A evolução de software é um sistema retroalimentado, lembra que a evolução de software é um sistema complexo que recebe *feedback* constante dos vários interessados.
- Em longo prazo, a taxa de evolução de um sistema acaba sendo determinada então pelos retornos positivos e negativos de seus usuários, bem como pela quantidade de verba disponível, número de usuários solicitando novas funções, interesses administrativos, etc.



Classificação das Atividades de Manutenção

- *Corretiva:*
 - é toda atividade de manutenção que visa corrigir erros ou defeitos que o software tenha.
- *Adaptativa:*
 - é toda atividade que visa adaptar as características do software a requisitos que mudaram, seja novas funções, sejam questões tecnológicas.
- *Perfectiva:*
 - é toda atividade que visa melhorar o desempenho ou outras qualidades do software sem alterar necessariamente sua funcionalidade.
- *Preventiva:*
 - é toda atividade que visa melhorar as qualidades do software de forma que erros potenciais sejam descobertos e mais facilmente resolvidos.



Manutenção Corretiva

- Visa corrigir os defeitos (que provocam erros) que o software possa ter.
- Ela ainda pode ser subdividida em dois subtipos:
 - Manutenção para correção de erros conhecidos.
 - Manutenção para detecção e correção de novos erros.
- Os erros conhecidos de um software são usualmente registrados em um documento de considerações operacionais, ou em notas de versão.

Manutenção Adaptativa

- Conforme visto nas Leis de Lehman, a *manutenção adaptativa* é inevitável quando se trata de sistemas de software. Isso por que:
 - Requisitos de cliente e usuário mudam com o passar do tempo.
 - Novos requisitos surgem.
 - Leis e normas mudam.
 - Tecnologias novas entram em uso.
 - Etc.
- O sistema desenvolvido poderá estar ou não preparado para acomodar tais modificações de contexto.



Requisitos permanentes e transitórios

- Com os requisitos permanentes acontece o seguinte:
 - É mais barato e rápido incorporá-los ao software durante o desenvolvimento.
 - É mais caro e demorado mudá-los depois que o software está em operação.
- Com os requisitos transitórios acontece o inverso:
 - É mais caro e demorado incorporá-los ao software durante o desenvolvimento.
 - É mais barato e rápido mudá-los depois que o software está em operação.



Manutenção Perfectiva

- Consiste em mudanças que afetam mais as características de desempenho do que de funcionalidade do software.
- Usualmente tais melhorias são buscadas em função de pressão de mercado, visto que produtos mais eficientes, ou com melhor usabilidade, com mesma funcionalidade são usualmente preferidos em relação aos menos eficientes, especialmente em áreas onde o processamento é crítico, como jogos e sistemas de controle em tempo real.
- A melhoria de características vai estar quase sempre ligada às qualidades externas do software, mas especialmente àquelas qualidades ligadas a funcionalidade, confiabilidade, usabilidade e eficiência.



Manutenção Preventiva

- Pode ser realizada através de atividades de reengenharia, nas quais o software é modificado para resolver problemas potenciais.
- Por exemplo, um sistema que suporta até 50 acessos simultâneos e que já conta com picos 20 a 30 acessos, pode sofrer um processo de manutenção preventiva, através de reengenharia ou refatoração de sua arquitetura, de forma que passe a suportar 500 acessos, desta forma afastando a possibilidade de colapso por um período de tempo razoável.
- Outro uso da manutenção preventiva consiste em aplicar técnicas de engenharia reversa ou como refatoração ou redocumentação para melhorar a manutenibilidade do software.



Processo de Manutenção

- Análise de esforço para a tarefa de manutenção.
- Análise de risco para a tarefa de manutenção (verificando possíveis riscos, sua probabilidade e impacto, bem como elaborando e executando possíveis planos de mitigação).
- Planejamento da tarefa de manutenção (estabelecendo prazos, responsáveis, recursos e entregas).
- Execução da tarefa de manutenção.

Norma ISO 1219-98

- *Classificação e identificação da requisição de mudança.*
 - Essa atividade vai avaliar, entre outras coisas, se a manutenção necessária será corretiva, adaptativa ou perfectiva e, em função de sua urgência, ela receberá um lugar na fila de prioridades das atividades de manutenção.
 - Opcionalmente também a solicitação de modificação poderá ser rejeitada se for impossível ou indesejável implementá-la.
- *Análise.*
 - Aqui as atividades tradicionais de análise entram em cena, com a identificação ou modificação de requisitos, modelo conceitual, casos de uso e outros artefatos, conforme a necessidade.
- *Design.*
 - Aqui entram as atividades usuais de *design*, com a definição da tecnologia e das camadas de interface, persistência, comunicação, etc.
- *Implementação.*
 - Onde é gerado novo código que atende à modificação solicitada, bem como são feitos os testes de unidade e integração.
- *Teste de sistema.*
 - Onde são feitos os testes finais das novas características do sistema do ponto de vista do usuário.
- *Teste de aceitação.*
 - Onde o cliente é envolvido para aprovar ou não as modificações feitas.
- *Entrega.*
 - Onde o produto é entregue ao cliente para nova instalação.



Ferramenta para Manutenção de Software

- **Bugzilla**, que é um sistema de rastreamento de defeitos, que permite que indivíduos ou equipes mantenham controle efetivo sobre defeitos encontrados e tratados em seus sistemas.
- É utilizada por grandes corporações e projetos, como NASA, NBC e Wikipédia.
- www.bugzilla.org/about/



Tipos de Atividades de Manutenção e suas Métricas

- As atividades de manutenção, longe de serem um mero detalhe, são as atividades onde as empresas colocam mais esforço.
- A análise de centenas de projetos no longo prazo mostrou que mais tempo e esforço foram colocados nas atividades de manutenção do que nas atividades de desenvolvimento de software.



Reparação de Defeitos

- *A reparação de defeitos* é possivelmente a atividade mais importante e urgente em manutenção de software, porque se destina a eliminar problemas que inicialmente não deveriam existir.
- O custo dessas atividades normalmente é absorvido pela empresa desenvolvedora, a não ser que cláusulas contratuais específicas estabeleçam outro tipo de entendimento.

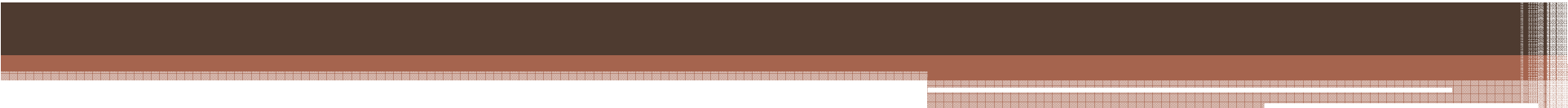
- 
- Uma métrica para este tipo de atividade, uma vez que os custos são arcados pela organização desenvolvedora, é o número de defeitos que a organização consegue reparar em um mês.
 - Um valor aceitável de acordo com normas americanas é de 8 defeitos reparados por mês.
 - Porém, empresas com bons processos e práticas conseguem reparar até 20 defeitos por mês em seus sistemas.

Tabela 14-1: Tempo de Resposta ao Erro em Função de sua Severidade¹⁹⁸.

Severidade	Significado	Tempo nominal da descoberta aos reparos iniciais	Percentual em relação aos defeitos relatados
1	Aplicação não funciona	1 dia	1%
2	Funcionalidade principal não funciona	2 dias	12%
3	Funcionalidade secundária não funciona	30 dias	52%
4	Erro cosmético	120 dias	35%



Fatores que podem influenciar a estimaco de esforo a ser aplicada s atividades de reparaco de defeitos:

- *Defeitos suspensos (abeyant).*
- *Defeitos invlidos.*
- *Consertos ruins (bad fix injection).*
- *Defeitos duplicados.*



Fatores que podem influenciar a estimaco de esforo a ser aplicada s atividades de reparaco de defeitos:

- *Defeitos suspensos (abeyant).*
 - Cerca de 10% das vezes, a falha relatada pelo cliente no  reproduzida no ambiente de manuteno.
 -  a tpica situao “na minha mquina funciona”.
 - Este tipo de defeito deve-se a combinaes de condies (verso do sistema operacional, outros produtos instalados na mesma mquina, etc.) que muitas vezes so difceis de detectar e reproduzir, e portanto  o tipo mais caro de manuteno corretiva.
 - Esses defeitos ficam ento suspensos at que se consiga repeti-los.
- *Defeitos invlidos.*
- *Consertos ruins (bad fix injection).*
- *Defeitos duplicados.*



Fatores que podem influenciar a estimativa de esforço a ser aplicada às atividades de reparação de defeitos:

- *Defeitos suspensos (abeyant).*
- *Defeitos inválidos.*
 - Cerca de 15% dos defeitos relatados por usuários não são propriamente defeitos no software, mas produto de erros produzidos pelos próprios usuários ou por sistemas relacionados.
 - Mesmo assim, esses problemas devem ser catalogados e processados e sua análise demanda tempo e esforço por parte a empresa que faz a manutenção do sistema.
- *Consertos ruins (bad fix injection).*
- *Defeitos duplicados.*



Fatores que podem influenciar a estimaco de esforo a ser aplicada s atividades de reparaco de defeitos:

- *Defeitos suspensos (abeyant).*
- *Defeitos invlidos.*
- *Consertos ruins (bad fix injection).*
 - Cerca de 7% das atividades de correo de erros acabam introduzindo novos erros no software.
 - Essa percentagem pode variar de 1% a 20% dependendo do nvel de qualidade do processo de manuteno e da seriedade com que os testes de regresso so feitos.
- *Defeitos duplicados.*



Fatores que podem influenciar a estimaco de esforo a ser aplicada s atividades de reparaco de defeitos:

- *Defeitos suspensos (abeyant).*
- *Defeitos invlidos.*
- *Consertos ruins (bad fix injection).*
- *Defeitos duplicados.*
 - Em sistemas com muitos usurios  comum que um mesmo defeito seja relatado por mais de um usurio.
 - Assim, embora o defeito s precise ser resolvido uma nica vez, o fato de que ele  relatado por vrios usurios faz com que seja necessrio investir tempo com isso.
 - Grandes empresas de software chegam a ter 10% de seus relatos de defeitos classificados como defeitos duplicados.



Remoção de Módulos Sujeitos a Erros

- Uma pesquisa realizada pela IBM nos anos 1960 demonstrou que os defeitos não se distribuem aleatoriamente ao longo de uma aplicação.
- Muito pelo contrário, eles tendem a se concentrar em determinados módulos da aplicação.
- Foi observado que, em um grande sistema da empresa, com 425 módulos, 300 módulos nunca foram alvo de manutenção corretiva, enquanto que outros 31 módulos concentraram cerca de 2000 relatos de erros ao longo de um ano, correspondendo a mais de 60% do total de erros relatados para o produto inteiro.
- Módulos sujeitos a defeitos podem nunca estabilizar porque a taxa de consertos ruins pode passar de 100%, ou seja, a cada defeito consertado, novos defeitos podem acabar sendo introduzidos.



Suporte a Usuários

- O *suporte a usuários* fará a interface entre o cliente do software e a empresa que presta manutenção ao software. O suporte a usuários usualmente recebe as reclamações, faz uma triagem delas e, ou encaminha uma solução previamente conhecida ao cliente, ou encaminha o problema ao setor de manutenção.
- O tamanho da equipe de suporte dependerá de vários fatores, dentre os quais, os mais importantes são a quantidade esperada de defeitos e a quantidade de clientes.
- Estima-se que para um software típico (que não apresenta grandes problemas de qualidade logo de partida), um atendente consiga tratar as chamadas de cerca de 150 clientes por mês, caso o meio de contato seja o telefone. Por outro lado, se o meio de contato for email ou chat, esse número pode subir para 1000 usuários por atendente por mês.



Migração entre Plataformas

- A migração de um produto para outra plataforma, quando se trata de software personalizado, é feita por demanda do cliente. Quando se trata de software de prateleira, é feita com a intenção de aumentar o mercado.
- Normalmente migrações são projetos por si só, embora possam ser consideradas atividades de evolução de software. Assume-se que sistemas desenvolvidos de acordo com boas práticas e com boa documentação possam ser migrados a uma taxa de 50 pontos de função por desenvolvedor-mês. Porém, se os sistemas forem mal documentados e com organização obscura, essa taxa pode baixar para até 5 pontos de função por desenvolvedor-mês.



Conversão de Arquitetura

- Uma conversão de arquitetura de sistema usualmente é feita por pressão tecnológica. É o caso, por exemplo, de mudar de arquivos simples para bancos de dados relacionais, ou mudar uma interface orientada a linha de comando para uma interface gráfica.
- No caso da migração entre plataformas, se o software for personalizado, a conversão possivelmente será uma demanda do cliente, enquanto que no caso de software de prateleira a conversão será uma estratégia para buscar novos mercados.
- A conversão de arquitetura também pode ser uma estratégia para melhorar a manutenibilidade de um sistema.
- A produtividade de um projeto de conversão de arquitetura dependerá basicamente da qualidade das especificações do sistema. Quanto mais obscuras as especificações, mais difícil será a conversão.
- Usualmente sistemas mal documentados ou obscuros precisarão passar por processos de engenharia reversa antes de serem convertidos para uma nova arquitetura.



Adaptações Obrigatórias

- Talvez o pior tipo de manutenção de software sejam as adaptações obrigatórias, devidas a mudanças em leis, formas de cálculo de impostos, etc.
- O problema é que essas mudanças são completamente imprevisíveis pela equipe de desenvolvimento ou manutenção e mesmo pelo cliente.
- Além disso, elas normalmente têm um prazo curto e estrito para serem aplicadas e as penalidades por não adaptação costumam ser altas.



Otimização de Performance

- Atividades de otimização de performance implicam em analisar e resolver gargalos da aplicação, usualmente relacionados com acesso a dados, processamento e número de usuários simultâneos.
- Tais atividades variam muito em relação ao tipo e a carga de trabalho e, assim, é muito difícil estabelecer um padrão para estimação de custos.
- Uma técnica que pode ser empregada em alguns casos é a otimização estilo *anytime*, usando *timeboxing*, ou seja, faz-se a melhor otimização possível dentro do tempo e recursos previamente destinados a esta atividade.

Melhorias

- São um tipo de manutenção adaptativa e perfectiva que são iniciadas, normalmente por requisição dos clientes, que são os que normalmente acabam arcando com os custos relacionados.
- Melhorias muitas vezes implicam na introdução de novas funcionalidades, de forma que as técnicas usuais de estimação de esforço por CII, pontos de função ou pontos de caso de uso podem ser aplicadas.
- Pode-se considerar que existam dois tipos de melhoria:
 - *Pequenas melhorias*,
 - consistindo de aproximadamente 5 pontos de função, ou seja, a introdução de um novo relatório, consulta ou tela.
 - *Grandes melhorias*,
 - consistindo de um número significativamente maior de pontos de função, tipicamente mais de 20 pontos de função, que devem ser tratadas como pequenos projetos de desenvolvimento.



Modelos de Estimação de Esforço de Manutenção

- ACT
- COCOMO II-Manutenção
- FP e SMPEEM



Modelo ACT

- O *modelo ACT* baseia-se em uma estimativa da percentagem de linhas de código que vão sofrer manutenção.
- São consideradas linhas em manutenção tanto as linhas de código novas criadas quanto as linhas alteradas durante a manutenção.
- O valor da variável ACT é então número de linhas que sofrem manutenção dividido pelo número total de linhas do código em um ano típico.

- Esforço estimado de manutenção durante um ano:
- $E = ACT * SDT$
- ACT = porcentagem de linhas a sofrer manutenção.
- SDT = Software Development Time

Exemplo

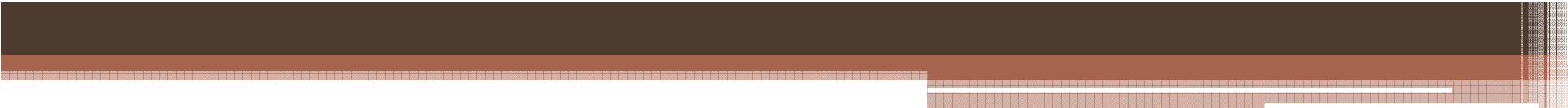
- Um software que foi desenvolvido com um esforço de 80 desenvolvedor-mês terá $SDT = 80$.
- Se a taxa anual esperada de linhas em manutenção (ACT) for de 2%, então o esforço anual esperado de manutenção para este software será dado por:
 - $E = 0,02 * 80 = 1,6$

Variação de Schaefer

- $E = ACT * 2,4 * KSLOC^{1,05}$

Exemplo

- Assim, um software com 20 mil linhas de código e ACT de 2% teria o seguinte esforço anual de manutenção (em desenvolvedor-mês):
- $E = 0,02 * 2,4 * 20^{1,05} = 1,115$

- 
- Este modelo tem as mesmas desvantagens do modelo COCOMO 81, ou seja, não é realisticamente aplicável a sistemas novos, quando não existem dados históricos para *ACT*, a quantidade de linhas de código modificadas não necessariamente indica esforço de manutenção e, acima de tudo, a abordagem não usa nenhum atributo das atividades de manutenção como base para calcular esforço.
 - Porém, é um método simples de aplicar na falta de outras informações.

Modelo de Manutenção de CII

A equação de estimação de esforço para manutenção é semelhante à equação usada no modelo *post-architecture*, com a seguinte forma:

$$E = A * KSLOC_m^S * \prod_{i=1}^n M_i$$

Onde:

- E é o esforço de manutenção em desenvolvedor-mês a ser calculado.
- A é uma constante calibrada pelo método, inicialmente valendo 2,94.
- $KSLOC_m$ é o número de linhas de código que se espera adicionar ou alterar ajustadas pelo fator de manutenção (ver abaixo). Não são contadas as linhas que eventualmente serão excluídas do código.
- S é o coeficiente de esforço, determinado pelos fatores de escala, e calculado como mostrado na Seção 7.3.
- M_i são os multiplicadores de esforço.

Mudanças em relação ao modelo *post-architecture* para o cálculo do esforço de manutenção:

- O multiplicador de esforço **SCED** (Cronograma de Desenvolvimento Requerido) não é usado (ou assumido como nominal), porque se espera que ciclos de manutenção tenham duração fixa predeterminada.
- O multiplicador de esforço **RUSE** (Desenvolvimento para Reuso) não é usado (ou assumido como nominal), porque se considera que o esforço requerido para manter a reusabilidade de um componente de software é balanceada pela redução do esforço de manutenção devido ao projeto, documentação e teste cuidadosos do componente.
- O multiplicador de esforço **RELY** (Software com Confiabilidade Requerida) tem uma tabela de aplicação diferenciada. Assume-se que RELY na fase de manutenção vai depender do valor que RELY tinha na fase de desenvolvimento. Se o produto foi desenvolvido com baixa confiabilidade, haverá maior esforço para consertá-lo. Se o produto foi desenvolvido com alta confiabilidade, haverá esforço menor para consertá-lo, exceto no caso de sistemas com risco à vida humana, nos quais a necessidade de confiabilidade mesmo na fase de manutenção faz crescer o esforço.

Tabela 14-2: Forma de obtenção do equivalente numérico para RELY na fase de manutenção.

Descritor	Pequena inconveniência	Perdas pequenas, facilmente recuperáveis	Perdas moderadas, facilmente recuperáveis	Alta perda financeira	Risco a vida humana	
Avaliação	Muito baixo	Baixo	Nominal	Alto	Muito Alto	Extra-alto
Equivalente numérico	1,23	1,10	1,00	0,99	1,07	n/a

O número de *KSLOC* usado na fase de manutenção deve ser ajustado antes de ser aplicado na equação de cálculo de esforço pelo uso do *fator de ajuste de manutenção (MAF)*:

$$KSLOC_m = (KSLOC_{adicionadas} + KSLOC_{modificadas}) * MAF$$

O fator de ajuste de manutenção *MAF* é calculado a partir da equação a seguir:

$$MAF = 1 + \left(\frac{SU}{100} * UNFM \right)$$

Onde:

- a) *SU* é o fator de ajuste relacionado à *compreensão do software (software understanding)*, calculado de acordo com a Tabela 14-3.
- b) *UNFM* é o fator de *não familiaridade* com relação ao software, calculado de acordo com a Tabela 14-4.

Tabela 14-3: Forma de cálculo do equivalente numérico para *SU*.



	Muito baixo	Baixo	Nominal	Alto	Muito alto
Estrutura	Coesão muito baixa, acoplamento alto, código espaguete	Coesão moderadamente baixa, acoplamento alto	Razoavelmente bem estruturado, algumas áreas fracas	Alta coesão, baixo acoplamento	Modularidade forte, ocultamento de informação em estruturas de dados ou controle (objetos)
Clareza da aplicação	As visões do programa e sua aplicação no mundo real não batem	Alguma correlação entre o programa e a aplicação	Correlação moderada entre o programa e a aplicação	Boa correlação entre o programa e a aplicação	As visões do programa e da aplicação no mundo real claramente batem
Auto-descrição	Código obscuro, documentação faltando, obscura ou obsoleta	Alguns comentários no código e cabeçalhos, alguma documentação útil	Nível moderado de documentação no código e cabeçalhos	Código e cabeçalhos bem documentados, algumas áreas fracas	Código autodescritivo, documentação atualizada e bem organizada baseada em <i>design</i>
Valor de <i>SU</i>	50	40	30	20	10

Tabela 14-4: Forma de cálculo do equivalente numérico para *UNFM*.

Nível de não familiaridade	Valor de <i>UNFM</i>
Completamente familiar	0,0
Basicamente familiar	0,2
Um tanto familiar	0,4
Um tanto não familiar	0,6
Basicamente não familiar	0,8
Completamente não familiar	1,0

Modelos FP e SMPEEM

- O modelo *FP* para cálculo de esforço de manutenção é baseado unicamente em pontos de função e não em linhas de código.
- Segundo este modelo é necessário calcular os pontos de função não ajustados de quatro tipos de funções:
 - *ADD*: UFP de funções que vão ser adicionadas.
 - *CHG*: UFP de funções que vão ser alteradas.
 - *DEL*: UFP de funções que vão ser removidas.
 - *CFP*: UFP de funções que serão adicionadas por conversão.

- Além de classificar as entradas, saídas, consultas, arquivos internos e arquivos externos nestes quatro tipos antes de contabilizar seus pontos de função não ajustados, a técnica propõe que os fatores de ajuste técnico (VAF) sejam calculados para dois momentos:
 - antes da manutenção VAF_A e
 - depois da manutenção VAF_D .
- A equação seguinte é então aplicada:

$$E = (ADD + CHG + CPF) * VAF_D + DEL * VAF_A$$

Evolução SMPEEM

- Inclui mais 10 fatores de ajuste específicos para as atividades de manutenção:
 - Conhecimento do domínio da aplicação.
 - Familiaridade com a linguagem de programação.
 - Experiência com o software básico (sistema operacional, gerenciador de banco de dados).
 - Estruturação dos módulos de software.
 - Independência entre os módulos de software.
 - Legibilidade e modificabilidade da linguagem de programação.
 - Reusabilidade de módulos de software legados.
 - Atualização da documentação.
 - Conformidade com padrões de engenharia de software
 - Testabilidade.



Engenharia Reversa e Reengenharia

- Em algumas situações o processo de manutenção ou evolução de um sistema exige uma atividade mais drástica do que simplesmente consertar partes do código.
- Sistemas antiquados, mal documentados e mal mantidos poderão requerer um processo completo de reengenharia para que possam voltar a evoluir de forma mais saudável.
- A reengenharia de um sistema é, basicamente, o processo de descobrir como um sistema funciona para que se possa refatorá-lo ou mesmo criar um novo sistema tecnologicamente atualizado que cumpra suas tarefas.

Taxionomia de Chikofsky e Cross II

- *Engenharia direta (forward engineering)*.
 - É o processo tradicional de produção de software que vai das abstrações de mais alto nível até o código executável.
- *Engenharia reversa (reverse engineering)*.
 - É o processo de analisar um sistema ou seus modelos de forma a conseguir produzir especificações de nível mais alto. É um processo de exame e explicação.
- *Redocumentação (redocumentation)*.
 - É uma subárea da engenharia reversa. Usualmente trata-se de obter formas alternativas de uma especificação no mesmo nível do artefato examinado.
- *Recuperação de projeto (design recovery)*.
 - É outra subárea da engenharia reversa. Ao contrário da anterior, a recuperação de projeto vai realizar abstrações a partir dos elementos examinados a fim de produzir artefatos em níveis mais altos do que os examinados.
- *Reestruturação (restructuring)*.
 - É uma das formas de refatoração, consistindo em transformar um artefato internamente, mas mantendo sua funcionalidade aparente. Normalmente a reestruturação é realizada para simplificar a arquitetura de sistemas de forma a minimizar futuros problemas de manutenção.
- *Reengenharia (reengineering)*.
 - É o exame e alteração de um sistema para reconstruí-lo de uma forma diferente. Geralmente inclui alguma forma de engenharia reversa seguida de engenharia direta ou reestruturação.

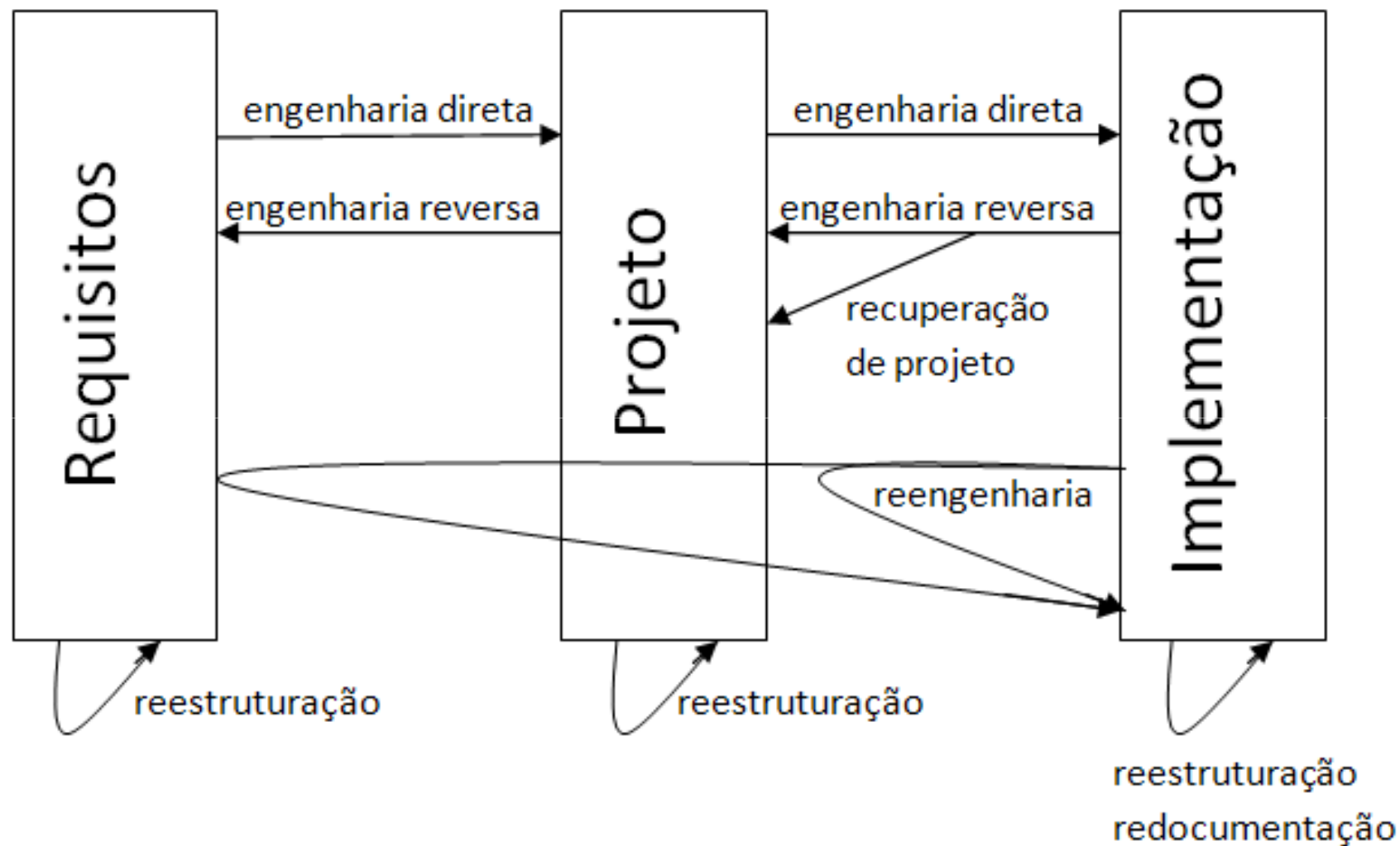


Figura 14-1: Relação esquemática entre os diferentes termos relacionados à engenharia reversa²⁰¹.



Engenharia reversa

- De código
 - Fonte
 - Objeto
- De dados



Técnicas de engenharia reversa de código

- *Análise de fluxo de dados.*
 - Consiste em verificar o comportamento do sistema como uma caixa preta, ou seja, sem ter conhecimento sobre sua estrutura interna.
 - A análise do comportamento do sistema pode permitir então que um novo sistema seja desenvolvido para ter o mesmo comportamento.
- *Dessassemblagem.*
 - Consiste em usar um desassemblador que converte o código executável em mnemônicos de linguagem *Assembly*.
- *Descompilação.*
 - Consiste em usar um descompilador para obter uma aproximação do código original usado para produzir o executável.
 - Os resultados podem variar bastante, pois há questões difíceis de tratar como, por exemplo, a escolha de nomes para variáveis e procedimentos.

Componentes de um descompilador

- **Carregador.** Este componente faz o carregamento do programa e identifica algumas informações básicas como o tipo de processador para o qual o código foi gerado e o ponto de entrada. Pode chegar até a encontrar o equivalente ao módulo principal de um programa a partir do qual as inicializações e chamadas são feitas.
- **Desassemblador.** Este componente procura transformar os códigos de máquina carregados por uma representação mnemônica independente de processador.
- **Identificador de expressões idiomáticas.** Alguns processadores usam instruções muito específicas para realizar operações que seriam bem mais simples em uma linguagem independente de tecnologia. Por exemplo, a instrução “xor eax,eax” é usada para atribuir zero ao registrador eax. Ela poderia ser mais claramente descrita como “eax:=0”. Expressões idiomáticas são catalogadas para cada processador.
- **Análise de programa.** O analisador de programa vai identificar sequências de operações e tentar agrupá-las em comandos. Por exemplo, uma expressão que em linguagem de alto nível seria escrita como “ $x := y + 45 * (z - x) / 2$ ” seria compilada como uma sequência de operações elementares de adição, multiplicação, subtração e divisão. O analisador de programa deve ser capaz de identificar esta sequência e transformá-la na expressão que possivelmente era a original do programa fonte.
- **Análise de fluxo de dados.** Consiste em detectar as variáveis e seu escopo no programa. O problema é complexo porque a mesma posição de memória pode ser ocupada por mais de uma variável em momentos diferentes e uma variável pode ocupar mais de uma posição da memória. A abordagem geral para tratar este problema é baseada em grafos e foi definida por Kildall (1973).
- **Análise de tipos.** Observando as operações (de máquina) efetuadas sobre determinadas variáveis pode-se inferir seu possível tipo. Por exemplo, operações AND nunca são executadas em variáveis de ponto flutuante ou ponteiros.
- **Estruturação.** Consiste em transformar estruturas de máquina em estruturas de alto nível como *if* e *while*.
- **Geração de código.** A fase final consiste em gerar o código na linguagem alvo. Possivelmente vários problemas ainda restarão e terão que ser resolvidos interativamente pelo usuário.



Engenharia Reversa de Dados

- Pode ser considerada como um caso especial da engenharia reversa onde o foco está na localização, organização e reinterpretação do significado dos dados de um sistema.
- Uma das atividades relacionadas à engenharia reversa de dados é a *análise de dados*.
- Esta atividade consiste em recuperar um modelo de dados atualizado (a partir de um sistema em operação), estruturalmente completo e semanticamente anotado.
- Esta atividade é particularmente difícil de ser automatizada.
- A recuperação dos modelos de bancos de dados (quando existem) é relativamente simples, mas estas estruturas frequentemente não contêm informações semânticas e estruturais completas sobre os dados, que acabam sendo diluídas em código executável e documentação.