14 Manutenção e Evolução de Software

Este capítulo apresenta os conceitos de *manutenção e evolução de software*, iniciando pelas *leis de Lehman* (Seção 14.1), que procuram explicar a necessidade e as características do processo de manutenção e evolução. Em seguida são caracterizadas as quatro *formas de manutenção em função dos objetivos* (Seção 14.2). O *processo de manutenção* também é explicado (Seção 14.3), bem como um conjunto de *ferramentas* úteis para manutenção de software (Seção 14.4). Finalmente são apresentados os diferentes tipos de *atividades* de manutenção e suas *métricas* (Seção 14.5) e *modelos para estimação de esforço* em atividades de manutenção e evolução (Seção 14.6). O capítulo termina com a apresentação dos conceitos relacionados a *reengenharia* e *engenharia reversa* (Seção 14.7), que são técnicas frequentemente usadas em processos de manutenção de software.

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.

Classicamente, a atividade de manutenção é considerada como o conjunto de modificações que o software sofre depois de ter sido terminado, ou seja, depois do final da fase de Transição no UP. Porém, o EUP (Seção 5.6) e o OUM (Seção 5.7) adicionam ao Processo Unificado uma fase chamada “Produção”, a qual justamente ocorre no momento em que o software está em operação e necessita de manutenção. Assim, as atividades de manutenção são consideradas tão relevantes e necessárias quanto as atividades de desenvolvimento para o sucesso do produto.

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.

Em uma interpretação do termo, pode-se considerar como “manutenção” o conjunto de tarefas individuais de modificação de um software em uso. Neste caso, a “evolução” seria o processo de mais longo prazo, ou seja, a evolução do software pode ser vista como uma sequência de manutenções ao longo do tempo.

Por outro lado, há autores (Erdil, Finn, Keating, Meattle, Park, & Yoon, 2003) que consideram que apenas as correções de erros (manutenção corretiva) podem ser consideradas atividades tradicionais de manutenção, enquanto que a otimização, adaptação e prevenção de erros é considerada evolução. Esta interpretação será seguida neste livro.

14.1 Necessidade de Manutenção e Evolução de Software (Leis de Lehman)

Considera-se que um software uma vez desenvolvido terá um valor necessariamente decrescente com o passar do tempo. Isso ocorre por que:

a) Falhas são descobertas.
b) Requisitos mudam.
c) 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:

a) Falhas sejam corrigidas.
b) Novos requisitos sejam acomodados.
c) Seja buscada simplicidade, eficiência e atualização tecnológica.

A realização destas atividades é parte do processo de manutenção e evolução de software.

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. Elas foram apresentadas em várias publicações entre 1974 e 1996 até chegar a sua forma atual.

Inicialmente, Lehman identifica dois tipos de sistemas:

a) *Tipo-S*: são sistemas especificados formalmente, entendidos como objetos matemáticos e cuja correção em relação a uma especificação pode ser provada for ferramentas formais.
b) *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.

Entende-se então que os sistemas do tipo-S são diferentes em relação à possibilidade de terem defeitos, pois apenas defeitos relacionados a especificação ausente podem ocorrer. Um exemplo de sistema de tipo-S poderia ser um pacote de funções matemáticas ou de cálculo de equações físicas, ou ainda de engenharia, que tenha sido desenvolvido através de métodos formais.

Já os sistemas de tipo-E são propensos a defeitos em seu processo de produção. Além disso, pelo fato de serem usados em conexão com o mundo real, esses sistemas podem precisar evoluir para atender a requisitos que mudam conforme o contexto de uso, por exemplo, no caso de leis que mudam, ou objetivos da empresa, ou ainda expectativas dos usuários.

As leis de evolução de Lehman, então, se aplicam apenas aos sistemas de tipo-E. Elas estabelecem que a evolução de software desse tipo é inevitável e não apenas resultado de uma equipe de desenvolvimento incapaz. Além disso, as leis mostram que existem limites em relação ao que as equipes de manutenção podem fazer. A auto-regulação impede que trabalho demasiado ou insuficiente seja executado, sob pena de descontinuação do sistema.

14.1.1 Lei da Mudança Contínua

A primeira lei, de 1974, afirma que 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.

A primeira lei expressa o fato conhecido de que programas suficientemente grandes nunca são terminados. Eles simplesmente continuam a evoluir.

Ela também expressa um princípio, segundo o qual o software envelhece como se fosse um organismo vivo. Esse envelhecimento seria resultado das inconsistências entre o software e o ambiente no qual ele está inserido. No caso, o ambiente muda, mas o software não se adapta sozinho. Assim, um processo de evolução é necessário para evitar a obsolescência do software, prolongando sua vida útil. Caso tal evolução não seja feita, o software pode chegar a ser aposentado.

14.1.2 Lei da Complexidade Crescente

A segunda lei, também de 1974, expressa que 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.

A segunda lei estabelece que a 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.

14.1.3 Lei Fundamental da Evolução de Programas: Auto regulação

A terceira lei, também de 1974, estabelece que 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).

A terceira lei, da auto-regulação, reconhece que 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.

Os pontos de controle e realimentação são exemplos de formas de regulação da organização. 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.

14.1.4 Lei da Conservação da Estabilidade Organizacional: Taxa de trabalho invariante

A quarta lei, de 1980, expressa que a taxa média efetiva de trabalho global em um sistema em evolução é invariante no tempo: ela não aumenta nem diminui.

A quarta lei, da conservação da estabilidade organizacional, parece ser a menos intuitiva de todas, pois se acredita que a carga de trabalho aplicada em um projeto depende apenas de decisões da gerência e que vá diminuindo com o tempo. Mas na prática, as demandas de usuários também influenciam nestas decisões e estas se mantêm praticamente constantes no tempo.

14.1.5 Lei da Conservação da Familiaridade: Complexidade percebida

A quinta lei, também de 1980, expressa que 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 quinta lei, da conservação da familiaridade, expressa o fato de que a taxa de crescimento de um sistema é limitada pela capacidade dos indivíduos envolvidos em absorver as novidades coletivamente e individualmente. Os dados observados sugerem que se certo valor limite de novidades for excedido, mudanças comportamentais ocorrem de forma a levar a baixar novamente taxa de crescimento.

14.1.6 Lei do Crescimento Contínuo

A sexta lei, de 1980, estabelece que o conteúdo funcional de um sistema deve crescer continuamente para manter a satisfação do usuário.

A sexta lei, do crescimento contínuo, estabelece que 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).

14.1.7 Lei da Qualidade Decrescente

A sétima lei, de 1996, expressa que 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.

A sétima lei, da qualidade decrescente, estabelece que 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, consequentemente, mais insatisfeitos com ele.

14.1.8 Lei do Sistema Realimentado

A oitava lei, de 1996, estabelece que 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 oitava lei, que estabelece que 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.

14.2 Classificação das Atividades de Manutenção

Segundo a norma ISO-IEC 14764:2006[194] a manutenção ou evolução de software deve ser classificada em quatro tipos:

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

14.2.1 Manutenção Corretiva

A *manutenção corretiva* visa corrigir os defeitos (que provocam erros) que o software possa ter. Ela ainda pode ser subdividida em dois subtipos:

a) Manutenção para correção de erros conhecidos.
b) 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 (Seção 10.3), de forma que os usuários do software possam contornar as falhas evitando maiores transtornos.

Conforme visto no Capítulo 13, nem sempre o fato de se saber que um software tem uma falha implica em saber onde se localiza o defeito que provoca essa falha. Assim, a atividade de teste pode até ter descoberto que numa determinada situação o software não se comporta de acordo com o esperado, mas a atividade de depuração pode não ter ainda encontrado o ponto no código fonte causador deste erro. Neste caso, enquanto a atividade de depuração ou mesmo de refatoração do software prossegue, a versão atual do software pode ter seus erros conhecidos registrados.

Porém, novos erros podem ser detectados pelos usuários ao longo do uso do software. Tais erros, quando relatados, devem ser incluídos no relatório de erros da versão do software e, dependendo de como a disciplina é organizada, imediatamente encaminhados ao setor de manutenção para que o defeito seja identificado e corrigido. O processo deve ser rastreado de forma que se saiba sempre se o erro foi encaminhado, se já foi resolvido e se foi incorporado a uma nova versão do software. Além disso, deve-se saber se o usuário que relatou o erro recebeu a nova versão do software em um prazo razoável.

14.2.2 Manutenção Adaptativa

Conforme visto nas Leis de Lehman (Seção 14.1), a *manutenção adaptativa* é inevitável quando se trata de sistemas de software. Isso por que:

[194] A norma pode ser adquirida por CHF 142,00 (preço consultado em 29/10/2010) em www.iso.org/iso/catalogue_detail.htm?csnumber=39064
Além disso, há uma *preview* oficial disponível em:
webstore.iec.ch/preview/info_isoiec14764%7Bed2.0%7Den.pdf

a) Requisitos de cliente e usuário mudam com o passar do tempo.
b) Novos requisitos surgem.
c) Leis e normas mudam.
d) Tecnologias novas entram em uso.
e) Etc.

O sistema desenvolvido poderá estar ou não preparado para acomodar tais modificações de contexto. Na verdade, qualquer requisito inicialmente identificado é passível de mudança durante a operação do sistema. Cabe ao analista, na fase de levantamento de requisitos, identificar quais destes requisitos serão considerados *permanentes* e quais serão considerados *transitórios* (Wazlawick, 2011, p. 29).

Com os requisitos permanentes acontece o seguinte:

a) É mais barato e rápido incorporá-los ao software durante o desenvolvimento.
b) É mais caro e demorado mudá-los depois que o software está em operação.

Com os requisitos transitórios acontece o inverso:

a) É mais caro e demorado incorporá-los ao software durante o desenvolvimento.
b) É mais barato e rápido mudá-los depois que o software está em operação.

Durante a análise de requisitos, então, cabe ao analista, identificar quais requisitos serão considerados transitórios para que possam ser adaptados por um simples processo de configuração. Os demais requisitos somente serão alterados a partir de um processo de manutenção que pode envolver inclusive a refatoração do software.

Normalmente são os requisitos não funcionais que devem ser classificados como transitórios ou permanentes. Por exemplo, considere que um sistema de venda de livros (Livraria Virtual) será produzido e um dos requisitos não funcionais (suplementar) é de que a moeda usada é o Real. Se o analista considerar este requisito como transitório, ele deverá implementar no sistema a possibilidade de trabalhar com outras moedas, mesmo que em um primeiro momento apenas o Real seja cadastrado. Se novas moedas passarem a ser utilizadas no futuro, bastará cadastrá-las.

Mas este processo tem um custo de desenvolvimento mais alto do que se o analista considerar o requisito como permanente. Ser *permanente* significa que o sistema não será preparado para sua mudança. Assim, apenas uma moeda existirá para o sistema: o Real. E não será possível cadastrar novas moedas. Se no futuro surgir a necessidade de cadastrar uma nova moeda, então será necessário aplicar manutenção adaptativa ao sistema, refatorando algumas partes do sistema para passar a trabalhar com essa possibilidade. Assim, a economia que se obteve durante o projeto será gasta durante a operação do sistema.

Desta forma, pode-se considerar que o analista deve ponderar, antes de decidir se um requisito é transitório ou permanente, qual a real probabilidade de que ele mude durante a operação do sistema, qual o impacto de se prevenir para esta mudança durante o projeto, e qual o impacto de mudar o sistema durante sua operação. Ponderando estes três valores

deverá então tomar uma decisão. Via de regra, na falta de bons motivos para o contrário, a maioria dos requisitos é deixada como permanente.

14.2.3 Manutenção Perfectiva

A *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 (Capítulo 11), mas especialmente àquelas qualidades ligadas a funcionalidade, confiabilidade, usabilidade e eficiência.

14.2.4 Manutenção Preventiva

A *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.

14.3 Processo de Manutenção

A manutenção pode ser vista como um processo a ser executado. A não ser no caso da manutenção preventiva, o processo de manutenção usualmente é iniciado a partir de uma requisição do cliente ou usuário do software.

No caso de manutenção corretiva, pode ser interessante que o usuário apresente uma descrição clara do erro, que pode geralmente ser produzida a partir de captura de telas.

No caso da manutenção perfectiva, o usuário deve descrever da melhor forma possível os novos requisitos que deseja. Para efetuar esta atividade ele pode precisar do apoio de um analista capacitado em levantamento de requisitos.

Sugere-se que as tarefas de manutenção identificadas sejam priorizadas e colocadas em uma fila de prioridades para serem resolvidas de acordo com a urgência. Nesta fila, as solicitações de manutenção perfectiva usualmente ocupam os últimos lugares. No caso da manutenção corretiva e adaptativa, o impacto do erro ou inadequação sobre os clientes deve ser o primeiro critério de desempate, devendo ser tratados prioritariamente os problemas de maior impacto.

Recomenda-se também que, uma vez selecionada uma tarefa de manutenção, as seguintes ações sejam efetuadas:

a) Análise de esforço para a tarefa de manutenção (Seção 14.6).

b) Análise de risco para a tarefa de manutenção (verificando possíveis riscos, sua probabilidade e impacto, bem como elaborando e executando possíveis planos de mitigação – Capítulo 8).
c) Planejamento da tarefa de manutenção (estabelecendo prazos, responsáveis, recursos e entregas – Capítulo 6).
d) Execução da tarefa de manutenção.

Ao final da tarefa de manutenção devem ser executados os testes de regressão (Seção 13.2.6).

Os custos com a manutenção e evolução de software podem ser bastante elevados em comparação com o custo de desenvolvimento. Algumas diretrizes gerenciais podem ser definidas para a redução destes custos, dentre as quais se destacam (Pressman, 2005):

a) Envolver a equipe de manutenção no desenvolvimento do sistema, se possível realizando inclusive rodízio entre as equipes.
b) Uso de padrões de projeto tanto no desenvolvimento quanto na manutenção.
c) Padronização de pedidos de alteração: o mesmo modelo de pedido de alteração usado durante o desenvolvimento pode ser usado na fase de manutenção.
d) Alocar verba para manutenção preventiva.
e) Tornar possível o rastreamento de requisitos e testes de regressão automatizados.

A norma IEEE 1219-98[195] define que o processo de manutenção de software deve ser composto por oito atividades:

a) *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.
b) *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.
c) *Design*. Aqui entram as atividades usuais de *design*, com a definição da tecnologia e das camadas de interface, persistência, comunicação, etc.
d) *Implementação*. Onde é gerado novo código que atende à modificação solicitada, bem como são feitos os testes de unidade e integração.
e) *Teste de sistema*. Onde são feitos os testes finais das novas características do sistema do ponto de vista do usuário.
f) *Teste de aceitação*. Onde o cliente é envolvido para aprovar ou não as modificações feitas.
g) *Entrega*. Onde o produto é entregue ao cliente para nova instalação.

O processo de manutenção é, portanto, bastante semelhante ao processo de desenvolvimento de software. Existem algumas diferenças específicas especialmente no que se refere ao tratamento com o cliente e também ao fato de que o processo de manutenção usualmente

[195] A partir de junho de 2010 esta norma está sendo revisada e substituída por outra, a P14764 (draft) ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=11168

dura muitos anos, ao passo que o processo de desenvolvimento usualmente dura meses ou poucos anos.

14.4 Ferramenta para Manutenção de Software

Um dos pontos chave referentes ao processo de manutenção é que se tenha uma boa disciplina de gerenciamento de configuração e mudança (Capítulo 10), sem a qual se poderá perder o controle sobre atividades de manutenção realizadas e disponibilizadas aos clientes. Essa atividade deverá ser, então suportada por ferramentas adequadas.

Existem várias ferramentas disponíveis que auxiliam o processo de manutenção ou evolução de software. Um exemplo é o Bugzilla[196], 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.

Pelo fato de ser gratuita e ter várias funcionalidades interessantes, essa ferramenta tem se tornado uma das mais populares. A ferramenta é baseada em *Web* e *email*, servindo para o controle e acompanhamento de defeitos, bem como para pedidos de alterações e correções. É utilizada por grandes corporações e projetos, como NASA, NBC e Wikipédia[197].

14.5 Tipos de Atividades de Manutenção e suas Métricas

Vários autores como Boehm (1981) apontam que 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.

Assim, é necessário que se tenha também modelos de estimação de esforço e custo para atividades de manutenção.

Jones (2002) apresenta um estudo onde observa que em média cada funcionário remove 8 defeitos de um sistema em operação por mês. Porém, as atividades de manutenção variam muito e podem ser classificadas em vários tipos (Jones T. C., 1998), os quais são discutidos nas próximas subseções.

14.5.1 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 clausulas contratuais específicas estabeleçam outro tipo de entendimento.

Dificilmente essas atividades podem ser estimadas com pontos de função porque a maioria das atividades de reparação de defeitos implica em escrever algumas poucas linhas de código, o que equivaleria a uma fração de um ponto de função. Mas por outro lado, algumas dessas atividades podem ser altamente consumidoras de tempo.

[196] www.bugzilla.org/about/
[197] pt.wikipedia.org/wiki/Bugzilla

Uma métrica mais comum 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.

Jones (1998) ainda indica que conforme a severidade do erro, diferentes tempos de espera são toleráveis (Tabela 14-1).

Tabela 14-1: Tempo de Resposta ao Erro em Função de sua Severidade[198].

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%

Os seguintes fatores ainda podem influenciar a estimação de esforço a ser aplicada às atividades de reparação de defeitos:

a) *Defeitos suspensos* (*abeyant*). Cerca de 10% das vezes, a falha relatada pelo cliente não é reproduzida no ambiente de manutenção. É a típica situação "na minha máquina funciona". Este tipo de defeito deve-se a combinações de condições (versão do sistema operacional, outros produtos instalados na mesma máquina, etc.) que muitas vezes são difíceis de detectar e reproduzir, e portanto é o tipo mais caro de manutenção corretiva. Esses defeitos ficam então suspensos até que se consiga repeti-los.
b) *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.
c) *Consertos ruins* (*bad fix injection*). Cerca de 7% das atividades de correção de erros acabam introduzindo novos erros no software. Essa percentagem pode variar de 1% a 20% dependendo do nível de qualidade do processo de manutenção e da seriedade com que os testes de regressão são feitos.
d) *Defeitos duplicados.* Em sistemas com muitos usuários é comum que um mesmo defeito seja relatado por mais de um usuário. Assim, embora o defeito só precise ser resolvido uma única vez, o fato de que ele é relatado por vários usuários faz com que seja necessário investir tempo com isso. Grandes empresas de software chegam a ter 10% de seus relatos de defeitos classificados como defeitos duplicados.

[198] Fonte: Jones (1998).

14.5.2 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 (Mukhija, 2003).

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[199].

14.5.3 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.

14.5.4 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 sistema forem mal documentados e com organização obscura, essa taxa pode baixar para até 5 pontos de função por desenvolvedor-mês.

14.5.5 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.

---

[199] A internet apresenta uma brincadeira relacionada a esse fato, que é incorporado na "técnica" *XGH*, ou *eXtreme Go Horse*. Uma das regras dessa técnica estabelece que a cada defeito consertado com *XGH* 7 novos defeitos são criados, o que faz com que a quantidade de defeitos tenda ao infinito (gohorseprocess.wordpress.com)

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, onde, por exemplo, pode-se transformar um sistema monolítico ou feito com blocos *ad-hoc*, em um sistema bem estruturado com componentes e classes coesas.

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.

14.5.6 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.

14.5.7 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* (Seção 4.2.2), ou seja, faz-se a melhor otimização possível dentro do tempo e recursos previamente destinados a esta atividade.

14.5.8 Melhorias

As *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:

a) *Pequenas melhorias*, consistindo de aproximadamente 5 pontos de função, ou seja, a introdução de um novo relatório, consulta ou tela.

b) *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.

Um dos aspectos que possivelmente diferencia a estimação de esforço das melhorias em relação ao desenvolvimento de software novo é que no caso das melhorias deve-se levar em conta o estado atual do sistema. Se o sistema for bem organizado e documentado será mais fácil integrar as novas funcionalidades, caso contrário, haverá um esforço de integração maior que deverá ser levado em conta como fator técnico no momento de aplicar a estimativa.

Estima-se que sistemas em operação, em média, aumentem seus pontos de função em cerca de 7% anualmente em função de melhorias (Jones T. C., 1998).

14.6 Modelos de Estimação de Esforço de Manutenção

Alguns modelos de estimação de esforço para atividades de manutenção foram propostos na literatura, alguns dos quais são apresentados nas subseções seguintes. Assim como os modelos de estimação de esforço de desenvolvimento, esses modelos paramétricos podem usar como base para a estimação tanto o número de pontos de função quanto o número de linhas de código estimadas.

14.6.1 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.

Boehm (1981) estabeleceu a seguinte equação para calcular o esforço estimado de manutenção durante um ano:

$$E = ACT * SDT$$

Onde *E* é o esforço, medido em desenvolvedor-mês a ser aplicado no período de um ano nas atividades de manutenção, *ACT* é a porcentagem esperada de linhas modificadas ou adicionadas durante um ano em relação ao tamanho do software e *SDT* é o tempo de desenvolvimento do software (*Software Development Time*).

Por 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:

$$E = 0{,}02 * 80 = 1{,}6$$

Então, para este sistema hipotético, de acordo com esta fórmula, espera-se um esforço anual de 1,6 desenvolvedor-mês em atividades de manutenção.

Schaefer (1985) apresenta a seguinte variação para esta fórmula:

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

Ou seja, Schaefer substitui o tempo de desenvolvimento do produto (caso este não seja conhecido), por uma fórmula baseada no número total de milhares de linhas de código do produto (*KSLOC*).

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.

14.6.2 Modelo de Manutenção de CII

Como já foi visto na Seção 7.3, CII é um método de estimação de esforço de desenvolvimento que toma como entrada o número de linhas de código a serem desenvolvidas ou pontos de função convertidos em *KSLOC*.

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 menutenção 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.

As seguintes mudanças em relação ao modelo *post-architecture* são implementadas para o cálculo do esforço de manutenção:

a) 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.
b) 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.

c) 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 (Tabela 14-2), 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

Assim, a tabela acima deve ser aplicada da seguinte forma: avalia-se a confiabilidadade com a qual o sistema foi originalmente desenvolvido (primeira linha) e encontra-se o valor numérico a ser aplicado na fase de manutenção na terceira linha. Nota-se que os valores são inversamente proporcionais aos usandos na fase de desenvolvimento (Tabela 7-12).

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*):

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

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*), calulado 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 *design*
Valor de *SU*	50	40	30	20	10

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

Nível de não familiaridade	Valor de *UNFM*
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

Caso não se conheça a priori o número de linhas a serem adicionadas ou modificadas, esse valor pode ser obtido a partir de uma análise histórica dos processos de manutenção do projeto ou da empresa. A partir de uma estimativa da porcentagem de linhas adicionadas ou alteradas, pode-se prever a quantidade de manutenção que será necessária no futuro.

14.6.3 Modelos FP e SMPEEM

O modelo *FP* (Albrecht, 1979) 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:

a) *ADD*: UFP de funções que vão ser adicionadas.
b) *CHG*: UFP de funções que vão ser alteradas.
c) *DEL*: UFP de funções que vão ser removidas.
d) *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*) (Seção 7.4.3) sejam calculados para dois momentos: antes da manutenção VAF_A e depois da manutenção VAF_D. A equação seguinte é então aplicada:

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

Ahn, Suh, Kim e Kim (2003)[200] apresentam uma evolução deste modelo, na qual incluem mais 10 fatores de ajuste específicos para as atividades de manutenção:

a) Conhecimento do domínio da aplicação.
b) Familiaridade com a linguagem de programação.
c) Experiência com o software básico (sistema operacional, gerenciador de banco de dados).
d) Estruturação dos módulos de software.
e) Independência entre os módulos de software.
f) Legibilidade e modificabilidade da linguagem de programação.
g) Reusabilidade de módulos de software legados.
h) Atualização da documentação.
i) Conformidade com padrões de engenharia de software
j) Testabilidade.

Os três primeiros fatores são referentes às habilidades de engenharia da equipe. Os quatro fatores seguintes são referentes a características técnicas e o três últimos fatores estão relacionados ao ambiente de manutenção.

14.7 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.

Chikofsky e Cross II (1990) apresentam uma taxonomia de termos relacionados à reengenharia de software:

a) *Engenharia direta* (*forward engineering*). É o processo tradicional de produção de software que vai das abstrações de mais alto nível até o código executável.
b) *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.
c) *Redocumentação* (*redocumentation*). É uma subárea da engenharia reversa. Usualmente trata-se de obter formas alternativas de uma especificação no mesmo nível do artefato examinado.
d) *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.

[200] https://files.ifi.uzh.ch/rerg/arvo/ftp/kvse/ahn03.pdf

e) *Reestruturação* (*reestructuring*). 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.
f) *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.

A Figura 14-1 apresenta esquematicamente a relação entre os diferentes termos.

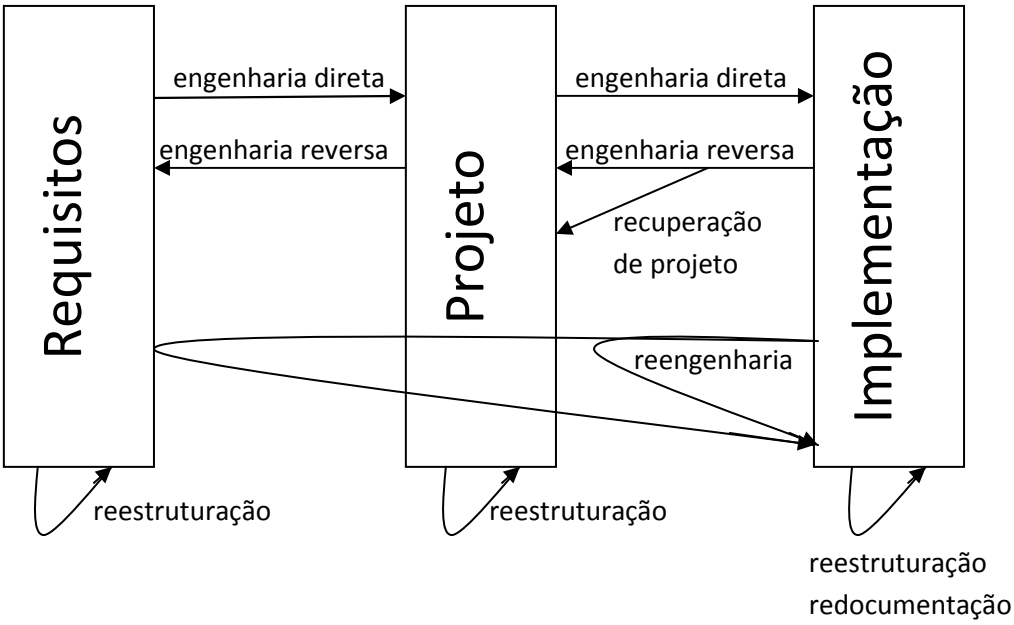

Figura 14-1: Relação esquemática entre os diferentes termos relacionados à engenharia reversa[201].

Segundo os autores, o termo "engenharia reversa" teve sua origem na análise de hardware, onde a prática de decifrar projetos a partir de produtos prontos é usual. Porém, na área de hardware, o objetivo da engenharia reversa normalmente é obter informações para duplicar um produto. Já na área de software os objetivos podem ser outros:

a) Lidar com a complexidade do código.
b) Gerar visões alternativas como, por exemplo, diagramas.
c) Recuperar informações perdidas sobre modificações não documentadas.
d) Detectar efeitos colaterais que não foram previstos no processo de engenharia direta.
e) Sintetizar conceitos e estrutura em abstrações de nível mais alto.
f) Facilitar o reuso de ativos de software existentes.

Uma das diretrizes da engenharia reversa consiste em, em função de seu alto custo, aplicá-la apenas onde for realmente necessária, não necessariamente ao sistema todo (Bennett, 2000).

[201] Fonte: Chikofsy e Cross II (1990).

Em função do seu objeto a engenharia reversa pode ser classificada em dois tipos: engenharia reversa de código e engenharia reversa de dados. As subseções seguintes apresentam estes dois tipos.

14.7.1 Engenharia Reversa de Código

Segundo Muller et al. (2000), a *engenharia reversa de software* tem se concentrado no *código* enquanto a *engenharia direta* atua mais amplamente nos diferentes níveis de abstração de uma especificação de sistema. Isso pode ser explicado pelo fato de que a questão de engenharia reversa usualmente se coloca quando existe um sistema legado e é necessário entre outras coisas descobrir regras de negócio escondidas nele.

O problema é que muitas vezes o código sozinho não contém todas as informações que se necessita. Muitas decisões tomadas ficam registradas apenas na memória dos desenvolvedores. Com o tempo essas memórias podem ser perdidas, ou as pessoas ficarem inacessíveis e, além disso, segundo as Leis de Lehman (Seção 14.1) a complexidade do software tende a crescer com o tempo. Assim, fica cada vez mais difícil lidar com o código.

É possível caracterizar dois tipos de engenharia reversa de código:

a) A partir de *código fonte*.
b) A partir de *código objeto*.

O primeiro tipo é usado para melhorar a compreensão sobre sistemas legados ou mal documentados. Já o segundo tipo pode ser usado entre outras coisas para a clonagem de produtos. Neste caso, pode-se usar a técnica de *clean room design*[202] (Schwartz, 2001) para evitar a quebra de direitos autorais. Essa técnica consiste em usar a engenharia reversa apenas apara entender o produto original e então desenvolver um produto novo que não use quaisquer partes do produto original. Deve-se tomar cuidado, porém porque no caso de patentes essa técnica não pode ser aplicada, já que os projetos patenteados são protegidos por leis.

A engenharia reversa de código pode ser realizada a partir de uma ou mais das seguintes técnicas:

a) *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.
b) *Dessassemblagem*. Consiste em usar um desassemblador que converte o código executável em mnemônicos de linguagem *Assembly*.
c) *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.

[202] Não confundir com o método *cleanroom* da Seção 11.4.3.

A análise de fluxo de dados pode ser feita com ferramentas como *bus analyzers* ou *packet sniffers*. Um *bus analyzer* é uma combinação de software e hardware usada para monitorar e apresentar em formato adequado o fluxo de informações que passa por um barramento de dados escolhido. Um exemplo de ferramenta é *JTAG* (*Joint Test Action Group*) baseada no padrão IEEE 1149.1[203].

Já os *packet sniffers* usualmente são conectados a redes de computadores (com ou sem fio). Uma comparação entre vários *packet sniffers* pode ser encontrada na Wikipédia[204].

Um desassemblador bastante popular é o *IDA*[205], disponível para vários sistemas operacionais e processadores. A empresa fabricante também disponibiliza um descompilador para programas compilados em C ou C++. Embora a versão corrente seja paga, versões anteriores do software são *freeware*. Outros exemplos de desassembladores são: *PVDasm*[206], *OllyDbg*[207] e *Ildasm*[208].

Um tipo especial de desassemblador é o *debugger* que usualmente permite que o código seja executado e que alterações de partes do código sejam feitas interativamente.

Os descompiladores são bem mais complexos do que os desassembladores, pois precisam gerar comandos em nível bem mais alto do que estes últimos que fazem apenas uma tradução direta dos códigos de máquina em mnemônicos. Os descompiladores usualmente trabalham em fases caracterizadas por diferentes componentes:

a) *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.
b) *Desassemblador*. Este componente procura transformar os códigos de máquina carregados por uma representação mnemônica independente de processador.
c) *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.
d) *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.

[203] www.jtag.com/
[204] en.wikipedia.org/wiki/Comparison_of_packet_analyzers
[205] www.hex-rays.com/idapro/
[206] pvdasm.reverse-engineering.net/
[207] www.ollydbg.de/
[208] msdn.microsoft.com/en-us/library/f7dy01k1%28VS.80%29.aspx

e) *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).
f) *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.
g) *Estruturação.* Consiste em transformar estruturas de máquina em estruturas de alto nível como *if* e *while*.
h) *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.

14.7.2 Engenharia Reversa de Dados

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. Davis e Aiken (2000) apresentam um apanhado histórico da evolução desta área.

Uma das atividades relacionadas à engenharia reversa de dados é a *analise de dados*. Segundo Muller et al. (2000), 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.

