CAPÍTULO

9

Evolução de software

Objetivos

Os objetivos deste capítulo são explicar por que a evolução é uma parte importante da engenharia de software e descrever os processos de evolução de software. Com a leitura deste capítulo, você:

- compreenderá que a mudança é inevitável caso os sistemas de software devam permanecer úteis, e que o desenvolvimento e a evolução de software podem ser integrados em um modelo em espiral;
- compreenderá os processos de evolução de software e as influências sobre esses processos;
- terá aprendido os diferentes tipos de manutenção de software e os fatores que afetam seus custos;
- entenderá como os sistemas legados podem ser avaliados para decidir se devem ser descartados, mantidos, passar por reengenharia ou substituídos.

Conteúdo

O desenvolvimento de software não é interrompido quando o sistema é entregue, mas continua por toda a vida útil do sistema. Depois que o sistema é implantado, para que ele se mantenha útil é inevitável que ocorram mudanças —, mudanças nos negócios e nas expectativas dos usuários, que geram novos requisitos para o software. Partes do software podem precisar ser modificadas para corrigir erros encontrados na operação, para que o software se adapte às alterações de sua plataforma de hardware e software, bem como para melhorar seu desempenho ou outras características não funcionais.

A evolução do software é importante, pois as organizações investem grandes quantias de dinheiro em seus softwares e são totalmente dependentes desses sistemas. Seus sistemas são ativos críticos de negócios, e as organizações devem investir nas mudanças de sistemas para manter o valor desses ativos. Consequentemente, a maioria das grandes empresas gasta mais na manutenção de sistemas existentes do que no desenvolvimento de novos sistemas. Baseado em uma pesquisa informal da indústria, Erlikh (2000) sugere que 85% a 90% dos custos organizacionais de software são custos de evolução. Outras pesquisas sugerem que cerca de dois terços de custos de software são de evolução. Com certeza, os custos para mudança de software requerem grande parte do orçamento de TI para todas as empresas.

Uma evolução de software pode ser desencadeada por necessidades empresariais em constante mudança, por relatos de defeitos de software ou por alterações de outros sistemas em um ambiente de software. Hopkins e Jenkins (2008) cunharam o termo 'desenvolvimento de software *brownfield*' para descrever situações nas quais os sistemas de

software precisam ser desenvolvidos e gerenciados em um ambiente em que dependem de vários outros sistemas de software.

Portanto, a evolução de um sistema raramente pode ser considerada de forma isolada. Alterações no ambiente levam a mudanças nos sistemas que podem, então, provocar mais mudanças ambientais. Certamente, o fato de os sistemas terem de evoluir em um 'ambiente rico' costuma aumentar as dificuldades e os custos de evolução. Assim como a compreensão e análise do impacto de uma mudança proposta no sistema em si, você também pode avaliar como isso pode afetar outros sistemas operacionais do ambiente.

Sistemas de software úteis muitas vezes têm uma vida útil muito longa. Por exemplo, sistemas militares ou de infraestrutura de grande porte, como sistemas de controle de tráfego aéreo, podem ter uma vida de 30 anos ou mais. Sistemas de negócios têm, usualmente, mais de dez anos de idade. Como os softwares custam muito dinheiro, uma empresa pode usar um sistema por muitos anos para ter retorno de seu investimento. É claro que os requisitos dos sistemas instalados mudam de acordo com os negócios e suas mudanças de ambiente. Portanto, os novos *releases* dos sistemas, que incorporam as alterações e atualizações, são geralmente criados em intervalos regulares.

Portanto, você deve pensar na engenharia de software como um processo em espiral com requisitos, projeto, implementação e testes que dura toda a vida útil do sistema (Figura 9.1). Você começa criando o *release* 1 do sistema. Uma vez entregue, alterações são propostas, e o desenvolvimento do *release* 2 começa imediatamente. De fato, a necessidade de evolução pode se tornar óbvia mesmo antes de o sistema ser implantado, de modo que os *releases* posteriores do software possam ser desenvolvidos antes de o *release* atual ser lançado.

Esse modelo de evolução do software implica que uma única organização é responsável tanto pelo desenvolvimento de software inicial quanto por sua evolução. A maioria dos pacotes de software é desenvolvida com essa abordagem. Para softwares customizados, uma abordagem diferente costuma ser usada. Uma empresa de software desenvolve softwares para um cliente, e ela possui uma equipe própria de desenvolvimento que assume o sistema. Ela é responsável pela evolução do software. Como alternativa, o cliente pode emitir um contrato separado para outra empresa cuidar do suporte e da evolução do sistema.

Nesse caso, as descontinuidades no processo em espiral são bem possíveis. Documentos de requisitos e de projeto não podem ser passados de uma empresa para outra. Empresas podem fundir-se ou reorganizar-se e herdar software de outras empresas e, em seguida, perceber que o software deve ser mudado. Quando a transição do desenvolvimento para a evolução é imperceptível, o processo de mudança do software após a entrega é, muitas vezes, chamado 'manutenção de software'. Como discuto adiante, a manutenção envolve atividades adicionais de processo, como o entendimento de programa, além das atividades normais de desenvolvimento de software.

Como mostra a Figura 9.2, Rajlich e Bennett (2000) propuseram uma visão alternativa do ciclo de vida de evolução do software. Nesse modelo, eles distinguem 'evolução' e 'em serviço'. A evolução é a fase em que mudanças significativas na arquitetura e funcionalidade do software podem ser feitas. Quando em serviço, as únicas mudanças feitas são relativamente pequenas, essenciais.

Figura 9.1 Um modelo em espiral de desenvolvimento e evolução

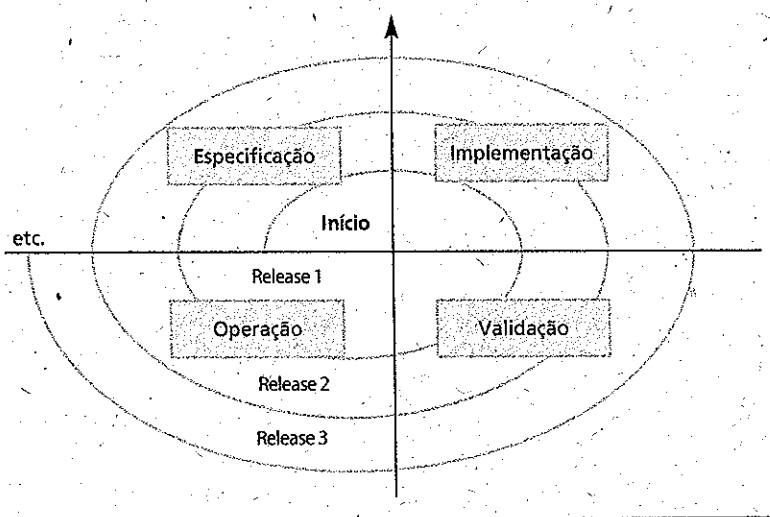
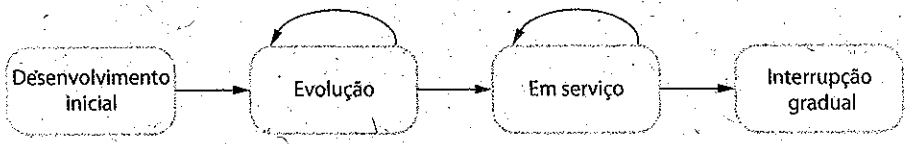


Figura 9.2 Evolução e em serviço



Durante a evolução, o software é usado com sucesso, e existe um fluxo constante de propostas de alterações de requisitos. No entanto, como o software é modificado, sua estrutura tende a degradar e as mudanças ficam mais e mais caras. Isso ocorre com frequência depois de alguns anos de uso, quando outras mudanças ambientais, como o hardware e sistemas operacionais, também são necessárias. Em algum estágio do ciclo de vida, o software chega a um ponto de transição em que mudanças significativas e implementação de novos requisitos se tornam menos rentáveis.

Nessa fase, o software passa da evolução para o serviço. Durante o serviço, o software ainda é útil e usado, mas apenas pequenas mudanças táticas são feitas. Normalmente, durante essa fase, a empresa está considerando como o software pode ser substituído. Na fase final, na interrupção gradual, o software ainda pode ser usado, mas não são implementadas novas mudanças. Os usuários precisam contornar qualquer problema que descubram.

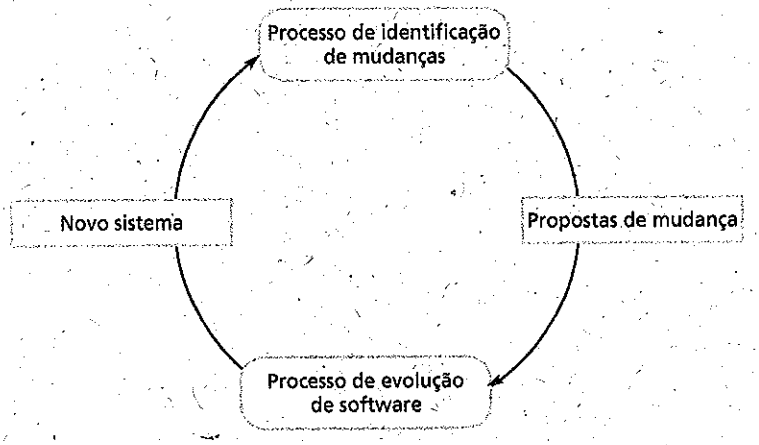
9.1 Processos de evolução

A evolução dos processos de software pode variar dependendo do tipo de software que esteja sendo mantido, dos processos de desenvolvimento usados em uma organização e das habilidades das pessoas envolvidas. Em algumas organizações, a evolução pode ser um processo informal em que as solicitações de mudança resultam, na maior parte, das conversas entre os usuários do sistema e desenvolvedores. Em outras empresas, é um processo formal com documentação estruturada produzida em cada estágio do processo.

Em todas as organizações, as propostas de mudança no sistema são os acionadores para a evolução. As propostas de mudança podem vir de requisitos já existentes que não tenham sido implementados no *release* de sistema, solicitações de novos requisitos, relatórios de *bugs* do sistema apontados pelos *stakeholders* e novas ideias para melhoria do software vindas da equipe de desenvolvimento. Os processos de identificação de mudanças e de evolução de sistema são cíclicos e continuam durante toda a vida de um sistema (Figura 9.3).

As propostas de mudança devem ser relacionadas aos componentes do sistema que necessitam ser modificados para implementar essas propostas, o que permite que o custo e o impacto da alteração sejam avaliados. Isso faz parte do processo geral de gerenciamento de mudanças, que também deve assegurar que as versões corretas dos componentes estejam incluídas em cada *release* do sistema. No Capítulo 25, discuto gerenciamento de mudanças e de configuração.

Figura 9.3 Processos de identificação de mudanças e de evolução



A Figura 9.4, adaptada de Arthur (1988), mostra uma visão geral do processo de evolução. O processo inclui as atividades fundamentais de análise de impacto, planejamento de *release*, implementação de sistema e liberação de um sistema para os clientes. O custo e o impacto dessas mudanças são avaliados para ver quanto do sistema é afetado pelas mudanças e quanto poderia custar para implementá-las. Se as mudanças propostas são aceitas, um novo *release* do sistema é planejado. Durante o planejamento de *release*, todas as mudanças propostas (correção de defeitos, adaptação e nova funcionalidade) são consideradas. Uma decisão é tomada de acordo com as mudanças a serem implementadas na próxima versão do sistema. As mudanças são implementadas e validadas, e uma nova versão do sistema é liberada. O processo itera com um novo conjunto de mudanças propostas para o próximo *release*.

Você pode pensar em implementação de mudanças como uma iteração do desenvolvimento, em que as revisões do sistema são projetadas, implementadas e testadas. No entanto, uma diferença fundamental é que o primeiro estágio da implementação da mudança pode envolver a compreensão do programa, especialmente se os desenvolvedores do sistema original não forem responsáveis pela implementação de mudanças. Durante esse estágio de compreensão do programa, é necessário que se entenda como o programa está estruturado, como implementa a funcionalidade e como a mudança proposta pode afetá-lo. Você precisa desse entendimento para se certificar de que as mudanças implementadas não causarão novos problemas quando forem implementadas em um sistema existente.

Idealmente, o estágio de implementação desse processo deve modificar a especificação, o projeto e a implementação do sistema para refletir as alterações do sistema (Figura 9.5). Novos requisitos que refletem as mudanças do sistema são propostos, analisados e validados. Componentes do sistema são reprojetados e implementados, e o sistema é testado novamente. Se for o caso, a prototipação das alterações propostas pode ser realizada como parte do processo de análise de mudanças.

Durante o processo de evolução, os requisitos são analisados detalhadamente, e as implicações das mudanças surgem onde não eram aparentes no processo anterior de análise. Isso significa que as alterações propostas podem ser modificadas e mais discussões com os clientes podem ser necessárias antes que estas sejam implementadas.

Às vezes, as solicitações de mudança são relacionadas a problemas do sistema que devem ser resolvidos com urgência. Essas mudanças urgentes podem surgir por três motivos:

1. Se ocorrer um defeito grave no sistema, que precisa ser corrigido para permitir a continuidade do funcionamento normal.
2. Se as alterações no ambiente operacional dos sistemas tiverem efeitos inesperados que interrompam o funcionamento normal.
3. Se houver mudanças inesperadas no funcionamento do negócio que executa o sistema, como o surgimento de novos concorrentes ou a introdução de nova legislação que afete o sistema.

Figura 9.4 O processo de evolução de software

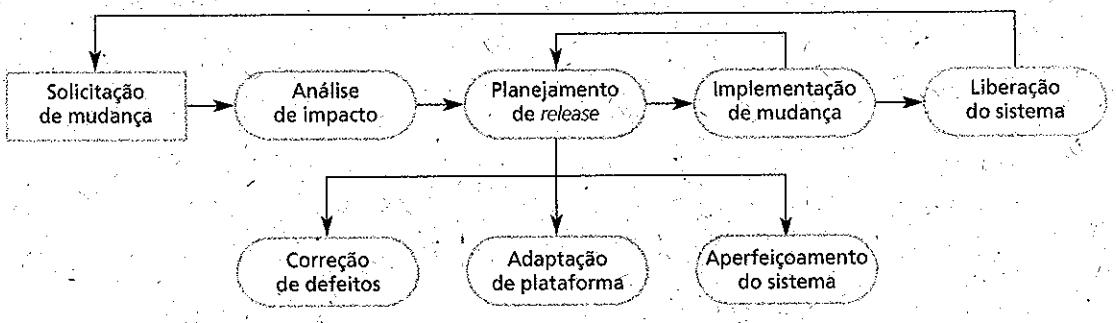
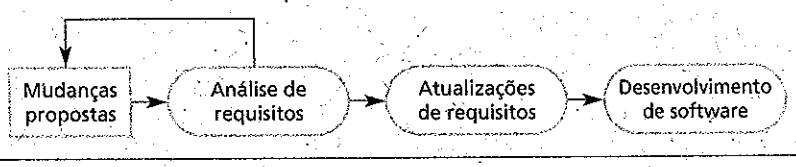


Figura 9.5 Implementação de mudança



Nesses casos, a necessidade de fazer a mudança rapidamente significa que você pode não ser capaz de acompanhar o processo de análise formal da mudança. Em vez de modificar os requisitos e o projeto, para resolver o problema imediatamente você faz uma correção de emergência no programa (Figura 9.6). No entanto, o perigo é que os requisitos, o projeto do software e o código podem tornar-se inconsistentes. Embora você possa ter a intenção de documentar a mudança nos requisitos e no projeto, correções adicionais de emergência para o software podem, então, ser necessárias. Elas têm prioridade sobre a documentação. Eventualmente, a mudança original é esquecida e a documentação e o código do sistema nunca são realinhados.

Em geral, as correções de emergência no sistema devem ser concluídas o mais rapidamente possível. Considerando a estrutura do sistema, você possivelmente optou por uma solução rápida e viável, e não a melhor solução. Isso acelera o processo de envelhecimento de software, faz que futuras alterações se tornem progressivamente mais difíceis, e os custos de manutenção, mais altos.

Idealmente, quando são feitas correções de emergência no código, após os defeitos serem corrigidos, a solicitação de mudança deve permanecer em aberto. O código de emergência pode então ser reimplementado mais cuidadosamente após uma análise mais aprofundada. Certamente, o código usado na correção pode ser reusado. Uma alternativa, uma melhor solução para o problema pode ser descoberta quando houver mais tempo disponível para análise. Na prática, porém, é quase inevitável que essas melhorias tenham baixa prioridade. Frequentemente, elas são esquecidas, sobretudo se as alterações são feitas no sistema; então, torna-se muito complicado refazer as correções de emergência.

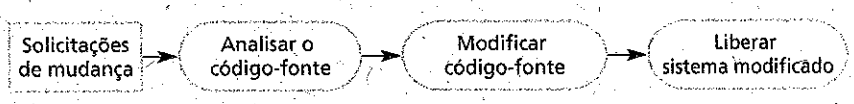
Métodos e processos ágeis discutidos no Capítulo 3 podem ser usados para a evolução de programa, bem como para seu desenvolvimento. Na verdade, por esses métodos serem baseados em desenvolvimento incremental, a transição de desenvolvimento ágil para evolução pós-entrega deveria ser imperceptível. Técnicas como os testes de regressão automatizados são úteis quando são feitas alterações no sistema. As alterações podem ser expressas como histórias de usuário e o envolvimento do cliente pode priorizar as mudanças necessárias em um sistema funcional. Em suma, simplesmente, a evolução envolve a continuação do processo de desenvolvimento ágil.

No entanto, podem surgir problemas em situações em que ocorre a transferência de uma equipe de desenvolvimento para uma equipe independente responsável pela evolução. Existem duas situações possivelmente problemáticas:

1. Quando a equipe de desenvolvimento usou uma abordagem ágil, mas a equipe de evolução não está familiarizada com os métodos ágeis e prefere uma abordagem baseada em planos. A equipe de evolução pode esperar uma documentação detalhada para apoiar a evolução, e, em processos ágeis, esta raramente é produzida. Não pode haver qualquer declaração definitiva de requisitos do sistema que possa ser modificada enquanto as alterações no sistema são feitas.
2. Quando uma abordagem baseada em planos for usada para o desenvolvimento, mas a equipe de evolução preferir usar métodos ágeis. Nesse caso, a equipe de evolução pode ter de começar do zero a partir do desenvolvimento de testes automatizados, e os códigos do sistema podem não ser refatorados e simplificados, como se prevê no desenvolvimento ágil. Nesse caso, alguma reengenharia pode ser necessária para melhorar o código antes que ele possa ser usado em um processo de desenvolvimento ágil.

Poole e Huisman (2001) relatam suas experiências no uso de 'Extreme Programming' para a manutenção de um sistema de grande porte originalmente desenvolvido pela abordagem baseada em planos. Depois da reengenharia do sistema para melhoria da sua estrutura, o XP foi usado com êxito no processo de manutenção.

Figura 9.6 O processo de correção de emergência



9.2 Dinâmica da evolução de programas

A dinâmica da evolução de programas é o estudo da mudança de sistema. Nas décadas de 1970 e 1980, Lehman e Belady (1985) realizaram vários estudos empíricos sobre a mudança de sistema com intenção de compreender mais sobre as características de evolução de software. O trabalho continuou na década de 1990 com Lehman e outros pesquisando o significado de *feedback* nos processos de evolução (LEHMAN, 1996; LEHMAN et al., 1998; LEHMAN et al., 2001). A partir desses estudos, eles propuseram as 'Leis de Lehman', relativas às mudanças de sistema (Tabela 9.1).

Lehman e Belady alegam que essas leis são suscetíveis de serem verdadeiras para todos os tipos de sistemas de software de grandes organizações (que eles chamam sistemas E-type). Nesses sistemas, os requisitos estão mudando para refletir as necessidades dos negócios. Novos *releases* do sistema são essenciais para o sistema fornecer valor ao negócio.

A primeira lei afirma que a manutenção de sistema é um processo inevitável. Como o ambiente do sistema muda, novos requisitos surgem, e o sistema deve ser modificado. Quando o sistema modificado é reintroduzido no ambiente, este promove mais mudanças no ambiente, de modo que o processo de evolução recomeça.

A segunda lei afirma que, quando um sistema é alterado, sua estrutura se degrada. A única maneira de evitar que isso aconteça é investir em manutenção preventiva. Você gasta tempo melhorando a estrutura do software sem aperfeiçoar sua funcionalidade. Obviamente, isso implica custos adicionais, além da implementação das mudanças de sistema exigidas.

A terceira lei é, talvez, a mais interessante e a mais controversa das leis de Lehman. Ela sugere que sistemas de grande porte têm uma dinâmica própria, estabelecida em um estágio inicial do processo de desenvolvimento. Isso determina a tendência geral do processo de manutenção de sistema e limita o número de possíveis alterações no mesmo. Lehman e Belady sugerem que essa lei seja uma consequência de fatores estruturais que influenciam e restringem a mudança do sistema, bem como os fatores organizacionais que afetam o processo de evolução.

Os fatores estruturais que afetam a terceira lei resultam da complexidade dos sistemas de grande porte. Assim que você altera e amplia um programa, sua estrutura tende a degradar. Isso é verdadeiro para todos os tipos de sistemas (não apenas software) e ocorre porque você está adaptando uma estrutura destinada a uma finalidade para uma finalidade diferente. Se essa degradação não for verificada, torna-se mais e mais difícil fazer alterações

Tabela 9.1 Leis de Lehman

Lei	Descrição
Mudança contínua	Um programa usado em um ambiente do mundo real deve necessariamente mudar, ou se torna progressivamente menos útil nesse ambiente.
Aumento da complexidade	Como um programa em evolução muda, sua estrutura tende a tornar-se mais complexa. Recursos extras devem ser dedicados a preservar e simplicar a estrutura.
Evolução de programa de grande porte	A evolução de programa é um processo de autorregulação. Atributos de sistema como tamanho, tempo entre *releases* e número de erros relatados são aproximadamente invariáveis para cada *release* do sistema.
Estabilidade organizacional	Ao longo da vida de um programa, sua taxa de desenvolvimento é aproximadamente constante e independente dos recursos destinados ao desenvolvimento do sistema.
Conservação da familiaridade	Durante a vigência de um sistema, a mudança incremental em cada *release* é aproximadamente constante.
Crescimento contínuo	A funcionalidade oferecida pelos sistemas tem de aumentar continuamente para manter a satisfação do usuário.
Declínio de qualidade	A qualidade dos sistemas cairá, a menos que eles sejam modificados para refletir mudanças em seu ambiente operacional.
Sistema de *feedback*	Os processos de evolução incorporam sistemas de *feedback* multiagentes, *multiloop*, e você deve tratá-los como sistemas de *feedback* para alcançar significativa melhoria do produto.

no programa. Fazer pequenas alterações diminui o grau de degradação estrutural e, assim, diminui os riscos que causam sérios problemas de confiança do sistema. Se você fizer grandes alterações, existe alta probabilidade de essas alterações introduzirem novos defeitos, os quais, por sua vez, inibem outras mudanças no programa.

Os fatores organizacionais que afetam a terceira lei refletem o fato de os sistemas de grande porte geralmente serem produzidos por grandes empresas. Essas empresas têm burocracias internas que definem o orçamento de mudanças para cada sistema e controlam o processo de tomada de decisão. As empresas têm de tomar decisões sobre os riscos e o valor das alterações e sobre os custos envolvidos. Tais decisões levam tempo para serem tomadas e, às vezes, leva-se mais tempo para decidir sobre as alterações a serem feitas do que para implementá-las. A velocidade do processo decisório da organização, portanto, regula a taxa de mudança do sistema.

A quarta lei de Lehman sugere que a maioria dos grandes projetos de programação trabalha em um estado 'saturado'. Ou seja, uma mudança de recursos ou de pessoal tem efeitos imperceptíveis sobre a evolução do sistema a longo prazo. Isso é consistente com a terceira lei, que sugere que a evolução de programa é independente das decisões de gerenciamento. Essa lei confirma que grandes equipes de desenvolvimento de software são, em muitos casos, improdutivas, porque os *overheads* de comunicação dominam o trabalho da equipe.

A quinta lei de Lehman está preocupada com o aumento das mudanças em cada *release* de sistema. Adicionar nova funcionalidade a um sistema inevitavelmente introduz novos defeitos. Quanto mais funcionalidade for adicionada em cada versão, mais falhas haverá. Portanto, uma grande adição de funcionalidade em um *release* de sistema significa que este deverá ser seguido por um *release* posterior, no qual os defeitos do novo sistema serão corrigidos e uma nova funcionalidade relativamente pequena deverá ser incluída. Essa lei sugere que não deve haver orçamento para grandes incrementos de funcionalidade em cada *release* sem levar em conta a necessidade de correção de defeitos.

As cinco primeiras leis estavam nas propostas iniciais de Lehman; as leis remanescentes foram adicionadas posteriormente, em outro trabalho. A sexta e sétima leis são semelhantes e, essencialmente, dizem que os usuários de um software estarão a cada dia mais descontentes com ele, a menos que ele seja mantido e que nova funcionalidade seja adicionada. A última lei reflete o mais recente trabalho com processos de *feedback*, embora ainda não esteja claro como isso pode ser aplicado em desenvolvimento prático de software.

As observações de Lehman parecem, de modo geral, sensatas. Devem ser levadas em consideração ao se planejar o processo de manutenção. É possível que, em alguns momentos, considerações comerciais possam nos forçar a ignorá-las. Por exemplo, por razões de marketing, pode ser necessário fazer várias alterações no sistema principal em um único *release*. As prováveis consequências disso são a necessidade de um ou mais *releases* dedicados à correção de erros. Muitas vezes, você vê isso em softwares de computadores pessoais, quando um importante novo *release* de uma aplicação é seguido rapidamente por uma atualização de correção de *bugs*.

9.3 Manutenção de software

A manutenção de software é o processo geral de mudança em um sistema depois que ele é liberado para uso. O termo geralmente se aplica ao software customizado em que grupos de desenvolvimento separados estão envolvidos antes e depois da liberação. As alterações feitas no software podem ser simples mudanças para correção de erros de codificação, até mudanças mais extensas para correção de erros de projeto, ou melhorias significativas para corrigir erros de especificação ou acomodar novos requisitos. As mudanças são implementadas por meio da modificação de componentes do sistema existente e, quando necessário, por meio da adição de novos componentes.

Existem três diferentes tipos de manutenção de software:

1. *Correção de defeitos*. Erros de codificação são relativamente baratos para serem corrigidos; erros de projeto são mais caros, pois podem implicar reescrever vários componentes de programa. Erros de requisitos são os mais caros para se corrigir devido ao extenso reprojeto de sistema que pode ser necessário.
2. *Adaptação ambiental*. Esse tipo de manutenção é necessário quando algum aspecto do ambiente do sistema, como o hardware, a plataforma do sistema operacional ou outro software de apoio sofre uma mudança. O sistema de aplicação deve ser modificado para se adaptar a essas mudanças de ambiente.
3. *Adição de funcionalidade*. Esse tipo de manutenção é necessário quando os requisitos de sistema mudam em resposta às mudanças organizacionais ou de negócios. A escala de mudanças necessárias para o software é, frequentemente, muito maior do que para os outros tipos de manutenção.

Na prática, não existe uma distinção clara entre esses tipos de manutenção. Ao adaptar o sistema a um novo ambiente, você pode adicionar funcionalidade para tirar proveito de novas características do ambiente. Os defeitos de software são frequentemente expostos porque os usuários usam o sistema de formas inesperadas. Mudar o sistema para acomodar sua maneira de trabalhar é a melhor maneira de corrigir tais defeitos.

Esses tipos de manutenção geralmente são reconhecidos, mas pessoas costumam dar-lhes nomes diferentes. 'Manutenção corretiva' é universalmente usado para se referir à manutenção para corrigir defeitos. No entanto, 'manutenção adaptativa' significa, em alguns casos, adaptação ao novo ambiente e, em outros, adaptar o software aos novos requisitos. 'Manutenção perfectiva' às vezes significa aperfeiçoar o software por meio da implementação de novos requisitos; às vezes, significa manutenção de funcionalidade de sistema para melhorar sua estrutura e seu desempenho. Em razão dessa incerteza quanto à nomenclatura, neste capítulo tenho evitado o uso de todos esses termos.

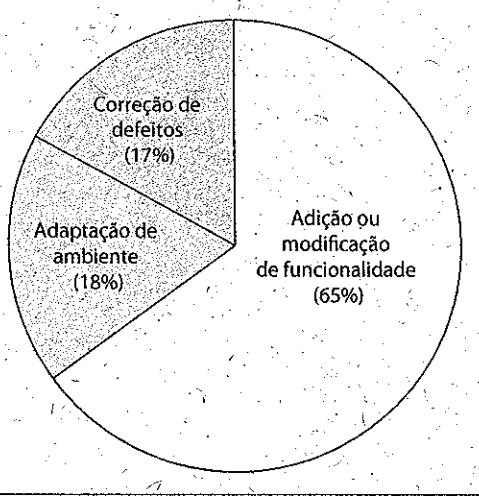
Existem vários estudos de manutenção de software que observaram os relacionamentos entre a manutenção e o desenvolvimento e entre as diferentes atividades de manutenção (KROGSTIE et al., 2005; LIENTZ e SWANSON, 1980; NOSEK e PALVIA, 1990; SOUSA, 1998). Devido às diferenças de terminologia, os detalhes desses estudos não podem ser comparados. Apesar das mudanças na tecnologia e dos diferentes domínios de aplicação, parece ter havido pouquíssimas mudanças na distribuição do esforço de evolução desde a década de 1980.

As pesquisas em geral concordam que a manutenção de software ocupa uma proporção maior dos orçamentos de TI que o desenvolvimento (a manutenção detém, aproximadamente, dois terços do orçamento, contra um terço para desenvolvimento). Elas também concordam que se gasta mais do orçamento de manutenção na implementação de novos requisitos do que na correção de *bugs*. A Figura 9.7 mostra, aproximadamente, a distribuição dos custos de manutenção. As porcentagens específicas variam de uma organização para outra, mas, universalmente, a correção de defeitos de sistema não é a atividade de manutenção mais cara. Evoluir o sistema para lidar com novos ambientes e novos ou alterados requisitos consome mais esforço de manutenção.

Os custos relativos de manutenção e desenvolvimento variam de um domínio de aplicação para outro. Guimaraes (1983) verificou que os custos de manutenção para sistemas de aplicação de negócio são comparáveis aos custos de desenvolvimento de sistema. Para sistemas embutidos de tempo real, os custos de manutenção são até quatro vezes maiores do que os custos de desenvolvimento. A alta confiabilidade e os requisitos de desempenho desses sistemas significam que os módulos devem ser fortemente ligados e, portanto, difíceis de serem alterados. Ainda que essas estimativas tenham mais de 25 anos, é improvável que a distribuição de custos para diferentes tipos de sistema tenha mudado significativamente.

Geralmente, vale a pena investir esforços no projeto e implementação de um sistema para redução de custos de mudanças futuras. Adicionar uma nova funcionalidade após a liberação é caro porque é necessário tempo para aprender o sistema e analisar o impacto das alterações propostas. Portanto, o trabalho feito durante o desenvolvimento para tornar a compreensão e a mudança no software mais fáceis provavelmente reduzirá os custos de evolução. Boas técnicas de engenharia de software, como descrição precisa, uso de desenvolvimento orientado a objetos e gerenciamento de configuração, contribuem para a redução dos custos de manutenção.

Figura 9.7 Distribuição do esforço de manutenção



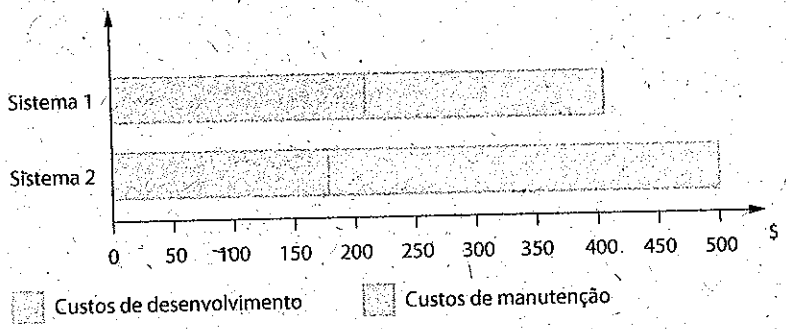
A Figura 9.8 mostra como maiores esforços durante o desenvolvimento do sistema para produção de um sistema manutenível reduzem os custos gerais durante a vida útil do sistema. Devido à redução potencial de custos de compreensão, análise e testes, existe um significativo efeito multiplicador quando o sistema é desenvolvido para manutenção. Para Sistema 1, o custo extra de desenvolvimento de 25 mil dólares é investido para tornar o sistema mais manutenível. Isso resulta em uma economia de cem mil dólares em custos de manutenção durante a vida útil do sistema, o que pressupõe que um aumento percentual no custo de desenvolvimento resulta na redução dos custos gerais do sistema em uma porcentagem comparável.

Essas estimativas são hipotéticas, mas não há dúvida de que o desenvolvimento de software para torná-lo mais manutenível é viável, quando todos os custos da vida do software são levados em conta. Essa é a razão da refatoração em desenvolvimento ágil. Sem refatoração, as mudanças no código tornam-se cada vez mais difíceis e caras. No entanto, no desenvolvimento dirigido a planos, a realidade é que raramente são feitos investimentos adicionais na melhoria do código durante o desenvolvimento. Isso é devido, principalmente, às maneiras como as organizações lidam com seus orçamentos. Investir na manutenção leva a aumentos de custos a curto prazo, que são mensuráveis. Infelizmente, os ganhos de longo prazo não podem ser medidos, assim como as empresas são relutantes em gastar dinheiro em um retorno futuro incerto.

Geralmente, é mais caro adicionar funcionalidade depois que um sistema está em operação do que implementar a mesma funcionalidade durante o desenvolvimento. As razões para isso são:

1. *Estabilidade da equipe.* Depois de um sistema ter sido liberado, é normal que a equipe de desenvolvimento seja desmobilizada e as pessoas sejam remanejadas para novos projetos. A nova equipe ou as pessoas responsáveis pela manutenção do sistema não entendem o sistema ou não entendem a estrutura para tomar as decisões de projeto. Antes de implementar alterações é preciso investir tempo em compreender o sistema existente.
2. *Más práticas de desenvolvimento.* O contrato para a manutenção de um sistema é geralmente separado do contrato de desenvolvimento do sistema. O contrato de manutenção pode ser dado a uma empresa diferente da do desenvolvedor do sistema original. Esse fator, juntamente com a falta de estabilidade da equipe, significa que não há incentivo para a equipe de desenvolvimento escrever um software manutenível. Se uma equipe de desenvolvimento pode cortar custos para poupar esforço durante o desenvolvimento, vale a pena fazê-lo, mesmo que isso signifique que o software será mais difícil de mudar no futuro.
3. *Qualificações de pessoal.* A equipe de manutenção é relativamente inexperiente e não familiarizada com o domínio de aplicação. A manutenção tem uma imagem pobre entre os engenheiros de software. É vista como um processo menos qualificado do que o desenvolvimento de sistema e é muitas vezes atribuída ao pessoal mais jovem. Além disso, os sistemas antigos podem ser escritos em linguagens obsoletas de programação. A equipe de manutenção pode não ter muita experiência de desenvolvimento nessas linguagens e precisa primeiro aprender para depois manter o sistema.
4. *Idade do programa e estrutura.* Com as alterações feitas no programa, sua estrutura tende a degradar. Consequentemente, como os programas envelhecem, tornam-se mais difíceis de serem entendidos e alterados. Alguns sistemas foram desenvolvidos sem técnicas modernas de engenharia de software. Eles podem nunca ter sido bem-estruturados e talvez tenham sido otimizados para serem mais eficientes do que inteligíveis. As documentações de sistema podem ter-se perdido ou ser inconsistentes. Os sistemas mais antigos podem não ter sido submetidos a um gerenciamento rigoroso de configuração, então se desperdiça muito tempo para encontrar as versões certas dos componentes do sistema para a mudança.

Figura 9.8 Custo de desenvolvimento e manutenção



Os primeiros três problemas decorrem do fato de que muitas organizações ainda consideram o desenvolvimento e a manutenção como atividades separadas. Manutenção é vista como uma atividade de segunda classe, e não há incentivo para gastar dinheiro, durante o desenvolvimento, para reduzir os custos na alteração do sistema. A única solução a longo prazo para esse problema é aceitar que os sistemas raramente têm um tempo de vida definido, mas continuam em uso, de alguma forma, por um período indeterminado. Como sugeri na introdução, você deve pensar em sistemas evoluindo ao longo de sua vida por um processo contínuo de desenvolvimento.

O quarto item, o problema da estrutura do sistema degradado, é o problema mais fácil de se resolver. Técnicas de reengenharia de software (descritas adiante neste capítulo) podem ser aplicadas para melhorar a estrutura do sistema e sua inteligibilidade. Transformações de arquitetura podem adaptar o sistema para um novo hardware. A refatoração pode melhorar a qualidade do código do sistema e facilitar a mudança.

9.3.1 Previsão de manutenção

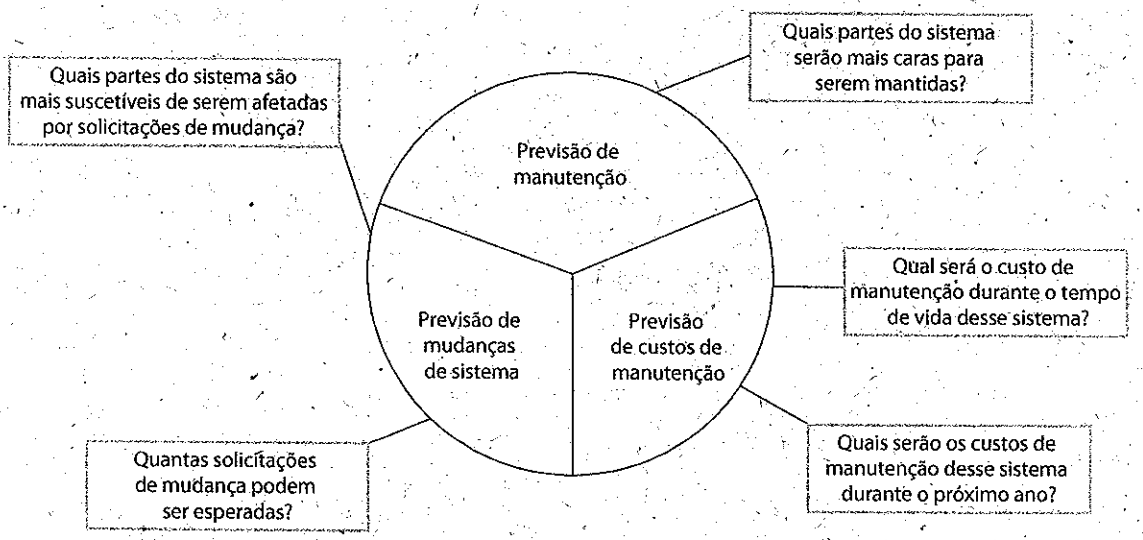
Gerentes odeiam surpresas, especialmente quando resultam em elevados custos inesperados. Você deve, portanto, tentar prever quais mudanças no sistema podem ser propostas e que partes do sistema são, provavelmente, as mais difíceis de serem mantidas. Você também deve tentar estimar os custos globais de manutenção para um sistema em determinado período de tempo. A Figura 9.9 mostra essas previsões e questões associadas.

Prever o número de solicitações de mudança para um sistema requer uma compreensão do relacionamento entre o sistema e seu ambiente externo. Alguns sistemas possuem um relacionamento muito complexo com seu ambiente externo, e as mudanças nesse ambiente inevitavelmente resultam em alterações. Para avaliar os relacionamentos entre um sistema e seu ambiente, você deve avaliar:

1. *O número e a complexidade das interfaces de sistema.* Quanto maior o número de interfaces e mais complexas elas forem, maior a probabilidade de serem exigidas as alterações de interface quando novos requisitos forem propostos.
2. *O número de requisitos inerentemente voláteis de sistema.* Como discutido no Capítulo 4, os requisitos que refletem as políticas e procedimentos organizacionais são provavelmente mais voláteis do que requisitos baseados em características estáveis de domínio.
3. *Os processos de negócio em que o sistema é usado.* Como processos de negócios evoluem, eles geram solicitações de mudança de sistema. Quanto mais processos de negócios usarem um sistema, maior a demanda por mudanças.

Por muitos anos, os pesquisadores analisaram os relacionamentos entre a complexidade do programa, medida por métricas como a complexidade ciclomática (McCABE, 1976), e sua manutenibilidade (BANKER et al., 1993 ; COLEMAN et al., 1994; KAFURA e REDDY, 1987; KOZLOV et al., 2008). Não é de se estranhar que esses estudos tenham

Figura 9.9 Previsão de manutenção



revelado que, quanto mais complexo for um sistema ou componente, mais cara será sua manutenção. Medidas de complexidade são particularmente úteis na identificação de componentes de programa suscetíveis a altos custos de manutenção. Kafura e Reddy (1987) analisaram um conjunto de componentes de sistema e descobriram que o esforço de manutenção tende a se centrar em um pequeno número de componentes complexos. Para reduzir os custos de manutenção, portanto, você deve tentar substituir componentes complexos de sistema com alternativas mais simples.

Depois que um sistema foi colocado em serviço, você pode ser capaz de usar dados de processo para ajudar a prever a manutenibilidade. Exemplos de métricas de processo que podem ser usadas para avaliação de manutenibilidade são apresentados a seguir:

1. *Número de solicitações de manutenção corretiva.* Um aumento no número de relatórios de *bugs* e falhas pode indicar que mais erros estão sendo introduzidos no programa do que corrigidos durante o processo de manutenção. Isso pode indicar um declínio na manutenibilidade.
2. *O tempo médio necessário para a análise de impacto.* Isso reflete o número de componentes de programa que são afetados pela solicitação de mudança. Se esse tempo aumenta, isso implica que cada vez mais componentes estão sendo afetados e a manutenibilidade está diminuindo.
3. *O tempo médio gasto para implementar uma solicitação de mudança.* Esse não é o mesmo que o tempo para análise de impacto, embora possam ser correlacionados. Essa é a quantidade de tempo que você precisa para modificar o sistema e sua documentação, depois de ter avaliado quais componentes são afetados. Aumento no tempo necessário para implementar uma mudança pode indicar declínio na manutenibilidade.
4. *Número de solicitações de mudança pendentes.* Ao longo do tempo, um aumento nesse número pode implicar uma diminuição na manutenibilidade.

Você usa informações previstas sobre mudanças de requisitos e previsões sobre a manutenibidade de sistema para prever os custos de manutenção. A maioria dos gerentes combina essas informações com a intuição e a experiência para estimar os custos. O modelo de estimativa de custos COCOMO 2 (BOEHM et al., 2000), discutido no Capítulo 24, sugere que uma estimativa para o esforço de manutenção de software pode ser baseada em um esforço para entender o código existente e os esforços para desenvolver o novo código.

9.3.2 Reengenharia de software

Como discutido na seção anterior, o processo de evolução de sistema envolve a compreensão do programa que tem de ser mudado e, em seguida, a implementação dessas mudanças. No entanto, muitos sistemas, especialmente sistemas legados mais velhos, são difíceis de serem compreendidos e mudados. Os programas podem ter sido otimizados para o desempenho ou uso de espaço à custa de inteligibilidade, ou, ao longo do tempo, a estrutura inicial do programa pode ter sido danificada por uma série de mudanças.

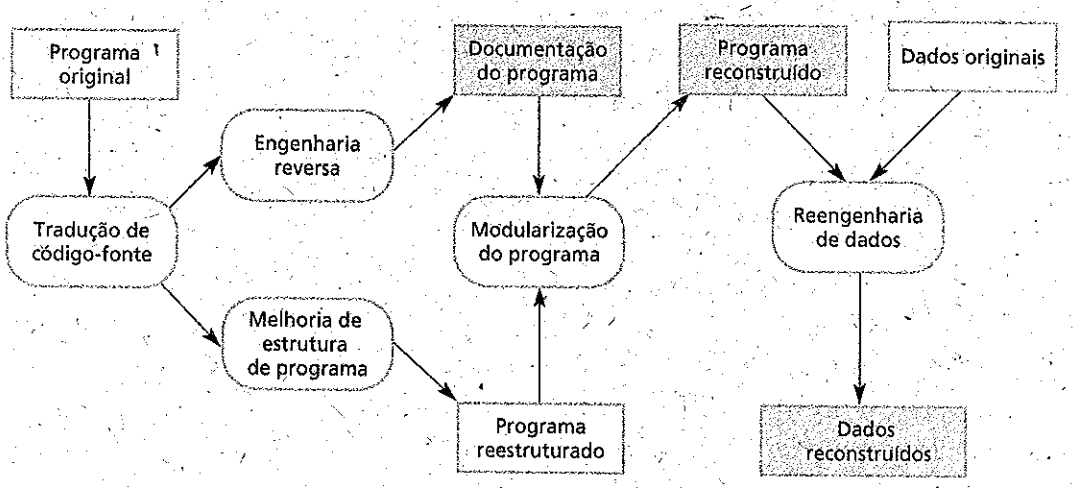
Para fazer com que os sistemas legados de software sejam mais fáceis de serem mantidos, é preciso aplicar reengenharia nesses sistemas visando a melhoria de sua estrutura e inteligibilidade. A reengenharia pode envolver a redocumentação de sistema, a refatoração da arquitetura de sistema, a mudança de linguagem de programação para uma linguagem moderna e modificações e atualizações da estrutura e dos dados de sistema. A funcionalidade de software não é alterada, e você geralmente deve evitar grandes mudanças na arquitetura de sistema.

Existem dois benefícios importantes na reengenharia, em vez de substituição:

1. *Risco reduzido.* Existe um alto risco em desenvolver novamente um software crítico de negócios. Podem ocorrer erros na especificação de sistema ou pode haver problemas de desenvolvimento. Atrasos no início do novo software podem significar a perda do negócio e custos adicionais.
2. *Custo reduzido.* O custo de reengenharia pode ser significativamente menor do que o de desenvolvimento de um novo software. Ulrich (1990) cita um exemplo de um sistema comercial cujos custos de reimplementação foram estimados em 50 milhões de dólares. O sistema foi reconstruído com sucesso por 12 milhões de dólares. Suspeito que, com a tecnologia moderna de software, o custo relativo de reimplementação é provavelmente inferior a esse, mas ainda consideravelmente superior aos custos da reengenharia.

A Figura 9.10 é um modelo geral de processo de reengenharia. A entrada para o processo é um programa legado, e a saída, uma versão melhorada e reestruturada do mesmo programa. As atividades desse processo de reengenharia são:

Figura 9.10 O processo de reengenharia



1. *Tradução de código-fonte.* Usando uma ferramenta de tradução, o programa é convertido a partir de uma linguagem de programação antiga para uma versão mais moderna da mesma linguagem ou em outra diferente.
2. *Engenharia reversa.* O programa é analisado e as informações são extraídas a partir dele. Isso ajuda a documentar sua organização e funcionalidade. Esse processo também é completamente automatizado.
3. *Melhoria de estrutura de programa.* A estrutura de controle do programa é analisada e modificada para que se torne mais fácil de ler e entender. Isso pode ser parcialmente automatizado, mas, normalmente, alguma intervenção manual é exigida.
4. *Modularização de programa.* Partes relacionadas do programa são agrupadas, e onde houver redundância, se apropriado, esta é removida. Em alguns casos, esse estágio pode envolver refatoração de arquitetura (por exemplo, um sistema que usa vários repositórios de dados diferentes pode ser refeito para usar um único repositório). Esse é um processo manual.
5. *Reengenharia de dados.* Os dados processados pelo programa são alterados para refletir as mudanças de programa. Isso pode significar a redefinição dos esquemas de banco de dados e a conversão do banco de dados existente para a nova estrutura. Normalmente devem-se limpar os dados, o que envolve encontrar e corrigir erros, remover registros duplicados etc. Ferramentas são disponíveis para dar suporte à reengenharia de dados.

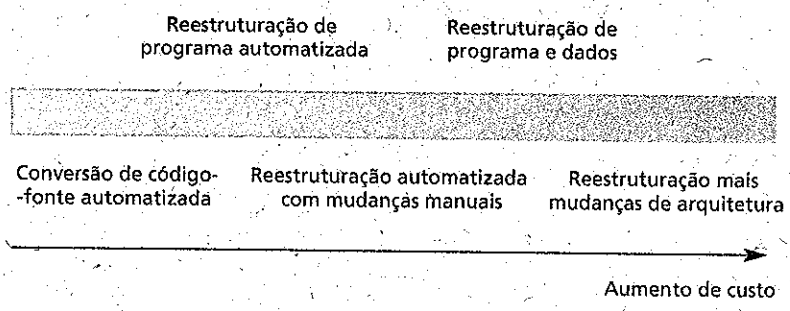
A reengenharia de programa pode não exigir necessariamente todas as etapas da Figura 9.10. Caso você ainda use o ambiente de desenvolvimento da linguagem de programação, você não precisa da tradução do código-fonte. Se você puder fazer automaticamente a reengenharia, a recuperação de documentação por meio da engenharia reversa pode ser desnecessária. A reengenharia de dados só é necessária se as estruturas de dados de programa mudarem durante a reengenharia de sistema.

Como discutido no Capítulo 19, para fazer o sistema que passou pela reengenharia interoperar com o novo software, você pode precisar desenvolver serviços adaptores. Eles escondem as interfaces originais do sistema de software e apresentam as novas interfaces, bem mais estruturadas e passíveis de serem usadas por outros componentes. Esse processo de empacotamento do sistema legado é uma técnica importante para o desenvolvimento em larga escala de serviços reusáveis.

Os custos da reengenharia, obviamente, dependem da extensão do trabalho. Como mostra a Figura 9.11, existe um espectro de possíveis abordagens para a reengenharia. Os custos aumentam da esquerda para a direita, de modo que a tradução de código-fonte é a opção mais barata. A reengenharia como parte da migração da arquitetura é a mais cara.

O problema com a reengenharia de software é que existem limites práticos para o quanto você pode melhorar um sistema por meio da reengenharia. Não é possível, por exemplo, converter um sistema escrito por meio de uma abordagem funcional para um sistema orientado a objetos. As principais mudanças de arquitetura ou a reorganização radical do sistema de gerenciamento de dados não podem ser feitas automaticamente, pois são muito caras. Embora a reengenharia possa melhorar a manutenibilidade, o sistema reconstruído provavelmente não será tão manutenível como um novo sistema, desenvolvido por meio de métodos modernos de engenharia de software.

Figura 9.11 Abordagens de reengenharia



9.3.3 Manutenção preventiva por refatoração

A refatoração é o processo de fazer melhorias em um programa para diminuir a degradação gradual resultante das mudanças (OPDYKE e JOHNSON, 1990). Isso significa modificar um programa para melhorar sua estrutura, para reduzir sua complexidade ou para torná-lo mais compreensível. No desenvolvimento orientado a objeto, a refatoração muitas vezes é considerada limitada, mas seus princípios podem ser aplicados a qualquer abordagem de desenvolvimento. Quando você refatorar um programa, não deve adicionar funcionalidade, mas concentrar-se na melhoria dele. Portanto, você pode pensar em refatoração como uma 'manutenção preventiva', que reduz os problemas de mudança no futuro.

Embora tanto a reengenharia como a refatoração sejam destinadas a tornar o software mais fácil de entender e mudar, elas não são a mesma coisa. A reengenharia ocorre depois que um sistema foi mantido por algum tempo e com o aumento dos custos de manutenção. Você pode usar ferramentas automatizadas para processar e reestruturar um sistema legado para criar um novo sistema mais manutenível. A refatoração é um processo contínuo de melhoria ao longo do processo de desenvolvimento e evolução, com o intuito de evitar a degradação do código, que aumenta os custos e as dificuldades de manutenção de um sistema.

A refatoração é uma parte inerente dos métodos ágeis, como o Extreme Programming, pois esses métodos são baseados em mudanças. Portanto, a qualidade de programa é suscetível de degradar rapidamente, de modo que os desenvolvedores frequentemente refatoram seus programas para evitar essa degradação. Em métodos ágeis, a ênfase em testes de regressão reduz o risco de introdução de novos erros por meio da refatoração. Quaisquer erros introduzidos deveriam ser detectáveis pelos testes que anteriormente foram bem-sucedidos e deveriam falhar. Entretanto, a refatoração não é dependente de outras 'atividades ágeis' e pode ser usada com qualquer abordagem de desenvolvimento.

Fowler et al. (1999) sugerem que existam situações estereotipadas (que ele chama 'maus cheiros') nas quais o código de um programa pode ser melhorado. Exemplos de maus cheiros que podem ser melhorados por meio de refatoração incluem:

1. *Código duplicado.* O mesmo código ou um muito semelhante pode ser incluído em diferentes lugares de um programa. Este pode ser removido e implementado com um único método ou função que usamos quando necessário.
2. *Métodos longos.* Se um método for muito longo, ele deve ser reprojetado como uma série de métodos mais curtos.
3. *Declarações* switch (case). Geralmente, envolvem a duplicação em situações em que o *switch* depende do tipo de algum valor. As declarações de *switch* podem ser espalhadas em torno de um programa. Em linguagens orientadas a objetos, muitas vezes você pode usar o polimorfismo para conseguir os mesmos resultados.
4. *Aglutinação de dados.* A aglutinação de dados ocorre quando um mesmo grupo de itens de dados (campos em classes, parâmetros em métodos) reincide em vários lugares de um programa. Muitas vezes, eles podem ser substituídos por um objeto que encapsule todos os dados.
5. *Generalidade especulativa.* Isso ocorre quando os desenvolvedores incluem generalidades em um programa, pois estas podem ser necessárias no futuro. Muitas vezes, elas podem ser removidas.

Em seu livro e site, Fowler também sugere algumas transformações primitivas de refatoração para lidar com o mau cheiro, as quais podem ser usadas isoladamente ou em conjunto. Exemplos dessas transformações incluem o método Extract, em que você remove dados duplicados e cria um novo método; a expressão condicional Consolidate, em que você substitui uma sequência de testes por um único teste; e o método Pull up, em que você substitui os métodos similares em subclasses por um único método em uma superclasse. Ambientes de desenvolvimento interativo, como o Eclipse, incluem suporte à refatoração em seus editores e isso facilita encontrar as partes dependentes de um programa que precisam ser alteradas para implementar a refatoração.

A refatoração, quando realizada durante o desenvolvimento de programa, é uma forma eficaz de reduzir os custos de manutenção a longo prazo de um programa. Entretanto, se você assumir um programa para manutenção cuja estrutura foi significativamente degradada, então pode ser praticamente impossível refatorar o código sozinho. É possível que você também tenha de pensar sobre a refatoração de projeto, a qual, provavelmente, é um problema mais caro e difícil. A refatoração de projeto envolve a identificação de padrões de projetos relevantes (discutido no Capítulo 7) e substituir o código existente por um código que implementa esses padrões de projeto (KERIEVSKY, 2004). Por falta de espaço, eu não discuto esse tema aqui.

9.4 Gerenciamento de sistemas legados

Para novos sistemas de software desenvolvidos por meio de processos modernos de engenharia de software, como o desenvolvimento incremental e CBSE, é possível planejar como integrar o desenvolvimento do sistema e sua evolução. Mais e mais empresas estão começando a entender que o processo de desenvolvimento de um sistema é um processo integral de ciclo de vida e que uma separação artificial entre desenvolvimento de software e manutenção de software é inútil. No entanto, ainda existem muitos sistemas legados que são sistemas críticos de negócio. Eles precisam ser ampliados e adaptados às mudanças das práticas de *e-business*.

Geralmente, as organizações têm um portfólio dos sistemas legados que usam, com um orçamento limitado para a manutenção e modernização desses sistemas. Elas precisam decidir como obter o melhor retorno de seus investimentos, o que envolve fazer uma avaliação realista de seus sistemas legados e, em seguida, decidir sobre a estratégia mais adequada para a evolução desses sistemas. Existem quatro opções estratégicas:

1. *Descartar completamente o sistema.* Essa opção deve ser escolhida quando o sistema não está mais contribuindo efetivamente para os processos dos negócios. Isso geralmente ocorre quando os processos de negócios se alteram desde que o sistema foi instalado e já não são dependentes do sistema legado.
2. *Deixar o sistema inalterado e continuar com a manutenção regular.* Essa opção deve ser escolhida quando o sistema ainda é necessário, mas é bastante estável e os usuários do sistema fazem poucas solicitações de mudança.
3. *Reestruturar o sistema para melhorar sua manutenibilidade.* Essa opção deve ser escolhida quando a qualidade do sistema foi degradada pelas mudanças, e novas mudanças para o novo sistema ainda estão sendo propostas. Esse processo pode incluir o desenvolvimento de novos componentes de interface, para que o sistema original possa trabalhar com outros sistemas mais novos.
4. *Substituir a totalidade ou parte do sistema por um novo sistema.* Essa opção deve ser escolhida quando fatores como hardwares novos significam que o sistema antigo não pode continuar em operação ou quando sistemas de prateleira podem permitir o desenvolvimento do novo sistema a um custo razoável. Em muitos casos, uma estratégia de substituição evolutiva pode ser adotada, na qual, sempre que possível, os componentes principais do sistema são substituídos por sistemas de prateleira com outros componentes reusados.

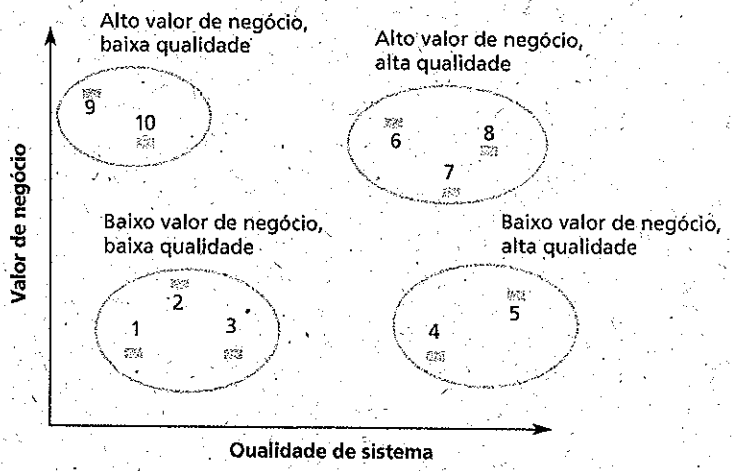
Naturalmente, essas opções não são exclusivas. Quando um sistema é composto de vários programas, várias opções podem ser aplicadas a cada programa.

Quando você está avaliando um sistema legado, precisa olhar para isso de uma perspectiva de negócio e de uma perspectiva técnica (WARREN, 1998). De uma perspectiva de negócio, você precisa decidir se o negócio realmente necessita do sistema. De uma perspectiva técnica, você precisa avaliar a qualidade do software de aplicação e o apoio de software e hardware do sistema. Em seguida, deve usar uma combinação do valor de negócio e da qualidade de sistema para informar sua decisão sobre o que deve ser feito com o sistema legado.

Por exemplo, suponha que uma organização tenha dez sistemas legados. Você deve avaliar a qualidade e o valor de negócio de cada um e, então, pode criar um gráfico mostrando o valor de negócio relativo e qualidade de sistema, conforme a Figura 9.12.

A partir da Figura 9.12, você pode ver que existem quatro grupos de sistemas:

Figura 9.12 Um exemplo de uma avaliação de sistema legado



1. *Baixa qualidade, baixo valor de negócio.* Manter esses sistemas em funcionamento se tornará caro, e a taxa de retorno para o negócio será bastante reduzida. Esses sistemas devem ser descartados.
2. *Baixa qualidade, alto valor de negócio.* Esses sistemas dão uma contribuição importante para o negócio, portanto, eles não podem ser descartados. Contudo, sua baixa qualidade significa que seu custo de manutenção é alto. Esses sistemas devem ser reestruturados para melhorar a qualidade. Eles podem ser substituídos, caso um sistema de prateleira esteja disponível.
3. *Alta qualidade, baixo valor de negócio.* Esses sistemas não contribuem muito para o negócio, mas seus custos de manutenção podem não ser altos. Não vale a pena substituir esses sistemas; assim, a manutenção normal do sistema pode ser mantida se mudanças de alto custo não forem necessárias e o hardware do sistema permanecer em uso. Se as mudanças necessárias ficarem caras, o software deve ser descartado.
4. *Alta qualidade, alto valor de negócio.* Esses sistemas precisam ser mantidos em operação. No entanto, sua alta qualidade significa que não há necessidade de investimentos na transformação ou substituição do sistema. A manutenção normal do sistema deve ser mantida.

Para avaliar o valor de negócio de um sistema, você precisa identificar os *stakeholders* do sistema, como os usuários finais do sistema e seus gerentes, e fazer uma série de perguntas sobre eles. Existem quatro questões básicas que você precisa discutir:

1. *O uso do sistema.* Se os sistemas são usados apenas ocasionalmente ou por um pequeno número de pessoas, eles podem ter um valor de negócio baixo. Um sistema legado pode ter sido desenvolvido para suprir uma necessidade de negócio que tenha se alterado ou que agora pode ser atendida de formas mais eficazes. No entanto, você precisa ter cuidado acerca de usos ocasionais, mas importantes, do sistema. Por exemplo, em uma universidade, um sistema de registro do aluno só pode ser usado no início de cada ano letivo. No entanto, trata-se de um sistema essencial com um alto valor de negócio.
2. *Os processos de negócios que são apoiados.* Quando um sistema é introduzido, processos de negócios são desenvolvidos para explorar as capacidades dele. Se o sistema for inflexível, mudar os processos pode ser impossível. No entanto, como o ambiente muda, os processos originais de negócio podem tornar-se obsoletos. Portanto, um sistema pode ter um valor de negócio baixo, porque obriga o uso de processos de negócios ineficientes.
3. *Confiança do sistema.* A confiança do sistema não é apenas um problema técnico, mas também um problema de negócio. Se um sistema não for confiável e os problemas afetarem diretamente os clientes do negócio ou significar que as pessoas no negócio são desviadas de outras tarefas para resolverem esses problemas, o sistema terá um valor de negócio baixo.
4. *As saídas do sistema.* A questão-chave aqui é a importância das saídas do sistema para o bom funcionamento do negócio. Se o negócio depender dessas saídas, o sistema terá um alto valor de negócio. Inversamente, se essas saídas puderem ser facilmente geradas de outra forma ou se o sistema produzir saídas que são raramente usadas, então seu valor de negócio poderá ser baixo.

Por exemplo, suponha que uma empresa ofereça um sistema de pedidos de viagem, usado pela equipe responsável pela organização de viagens. Eles podem fazer pedidos com um agente de viagem aprovado. Os bilhetes são entregues e a empresa é cobrada por eles. No entanto, uma avaliação de valor do negócio pode revelar que esse sistema é usado apenas para uma porcentagem bastante pequena dos pedidos colocados; as pessoas que fazem pedidos de viagens acham mais barato e mais conveniente lidar diretamente com os fornecedores de viagens por meio de seus sites. Esse sistema ainda pode ser usado, mas não há motivos reais para que seja mantido. A mesma funcionalidade está disponível em sistemas externos.

Por outro lado, digamos que uma empresa tenha desenvolvido um sistema que mantém o controle de todos os pedidos anteriores de clientes e automaticamente gera lembretes, para que os clientes repitam os pedidos. Esse procedimento resulta em um grande número de pedidos repetidos e mantém os clientes satisfeitos porque sentem que seu fornecedor está ciente de suas necessidades. As saídas de um sistema desse tipo são muito importantes para o negócio e, portanto, esse sistema tem um alto valor de negócio.

Para avaliar um sistema de software a partir de uma perspectiva técnica, é preciso considerar tanto o sistema de aplicação em si, quanto o ambiente no qual ele opera. O ambiente inclui o hardware e todos os softwares de apoio associados (compiladores, ambientes de desenvolvimento etc.) necessários para manter o sistema. O ambiente é importante porque muitas mudanças de sistema resultam de mudanças no ambiente, como atualizações no hardware ou sistema operacional.

Se possível, no processo de avaliação ambiental, você deve fazer as medições de sistema e de seus processos de manutenção. Exemplos de dados que podem ser úteis incluem os custos de manutenção do hardware do sistema e do software de apoio, o número de defeitos de hardware que ocorrem durante um período de tempo e a frequência de *patches* e correções aplicadas ao software de suporte do sistema.

Fatores que você deve considerar na avaliação do ambiente são mostrados na Tabela 9.2. Observe que essas não são todas as características técnicas do ambiente. Você também deve considerar a confiabilidade dos fornecedores de hardware e software de apoio. Se esses fornecedores não estiverem mais no negócio, pode não haver mais suporte para seus sistemas.

Para avaliar a qualidade técnica de um sistema de aplicação, você precisa avaliar uma série de fatores (Tabela 9.3), os quais estão essencialmente relacionados com a confiança do sistema, as dificuldades de sua manutenção e sua documentação. Você também pode coletar dados que o ajudarão a julgar a qualidade do sistema. Dados que podem ser úteis na avaliação de qualidade são:

Tabela 9.2 Fatores usados na avaliação de ambientes

Fator	Questões
Estabilidade do fornecedor	O fornecedor ainda existe? O fornecedor é financeiramente estável e deve continuar existindo? Se o fornecedor não está mais no negócio, existe alguém que mantém os sistemas?
Taxa de falhas	O hardware tem uma grande taxa de falhas reportadas? O software de apoio trava e força o reinício do sistema?
Idade	Quantos anos têm o hardware e o software? Quanto mais velho o hardware e o software de apoio, mais obsoletos serão. Ainda podem funcionar corretamente, mas poderia haver significativos benefícios econômicos e empresariais se migrassem para um sistema mais moderno.
Desempenho	O desempenho do sistema é adequado? Os problemas de desempenho têm um efeito significativo sobre os usuários do sistema?
Requisitos de apoio	Qual apoio local é requisitado pelo hardware e pelo software? Se houver altos custos associados a esse apoio, pode valer a pena considerar a substituição do sistema.
Custos de manutenção	Quais são os custos de manutenção de hardware e de licenças de software de apoio? Os hardwares mais antigos podem ter custos de manutenção mais elevados do que os sistemas modernos. Os softwares de apoio podem ter altos custos de licenciamento anual.
Interoperabilidade	Existem problemas de interface do sistema com outros sistemas? Compiladores podem, por exemplo, ser usados com as versões atuais do sistema operacional? É necessária a emulação do hardware?

Tabela 9.3 Fatores usados na avaliação de aplicações

Fatores	Questões
Inteligibilidade	Quão difícil é compreender o código-fonte do sistema atual? Quão complexas são as estruturas de controle usadas? As variáveis têm nomes significativos que refletem sua função?
Documentação	Qual documentação do sistema está disponível? A documentação é completa, consistente e atual?
Dados	Existe um modelo de dados explícito para o sistema? Até que ponto os dados nos arquivos estão duplicados? Os dados usados pelo sistema são atuais e consistentes?
Desempenho	O desempenho da aplicação é adequado? Os problemas de desempenho têm um efeito significativo sobre os usuários do sistema?
Linguagem de programação	Compiladores modernos estão disponíveis para a linguagem de programação usada para desenvolver o sistema? A linguagem de programação ainda é usada para o desenvolvimento do novo sistema?
Gerenciamento de configuração	Todas as versões de todas as partes do sistema são gerenciadas por um sistema de gerenciamento de configuração? Existe uma descrição explícita das versões de componentes usadas no sistema atual?
Dados de teste	Existem dados de teste para o sistema? Existem registros dos testes de regressão feitos quando novos recursos forem adicionados ao sistema?
Habilidades de pessoal	Existem pessoas disponíveis com as habilidades necessárias para manter a aplicação? Existem pessoas disponíveis que tenham experiência no sistema?

1. *O número de solicitações de mudança no sistema.* As alterações no sistema geralmente corrompem a estrutura do sistema e tornam futuras alterações mais difíceis. Quanto maior esse valor acumulado, menor é a qualidade do sistema.
2. *O número de interfaces de usuário.* Esse é um fator importante nos sistemas baseados em formulários, em que cada formulário pode ser considerado uma interface de usuário independente. Quanto mais interfaces, mais prováveis as inconsistências e redundâncias nessas interfaces.
3. *O volume de dados usados pelo sistema.* Quanto maior o volume de dados (número de arquivos, o tamanho do banco de dados etc.), maior a possibilidade de inconsistência nos dados, o que reduz a qualidade do sistema.

Idealmente, a avaliação objetiva deve ser usada para informar as decisões sobre o que fazer com um sistema legado. No entanto, em muitos casos, as decisões não são realmente objetivas, mas baseadas em considerações organizacionais ou políticas. Por exemplo, se duas empresas se fundem, o parceiro politicamente mais poderoso costuma manter seu sistema e se desfazer dos outros. Se a gerência sênior de uma organização decide mudar para uma nova plataforma de hardware, pode ser necessário que as aplicações sejam substituídas. Se não houver orçamento disponível para a transformação do sistema em um determinado ano, então a manutenção do sistema pode continuar, mesmo que isso resulte, a longo prazo, em maiores custos.

PONTOS IMPORTANTES

- O desenvolvimento e a evolução de software podem ser pensados como um processo integrado e interativo, que pode ser representado por um modelo em espiral.
- Para sistemas customizados, os custos de manutenção de software geralmente excedem os custos de desenvolvimento de software.
- O processo de evolução do software é dirigido pelas solicitações de mudança e inclui a análise do impacto da mudança, o planejamento de *release* e implementação da mudança.
- As leis de Lehman, como a noção de que a mudança é contínua, descrevem uma série de considerações provenientes de estudos, de longo prazo, de evolução de sistema.
- Existem três tipos de manutenção de software, ou seja, correção de *bugs*, modificação do software para funcionar em um novo ambiente e implementação de requisitos novos ou alterados.