9 Evolução de software

OBJETIVOS

Os objetivos deste capítulo são explicar por que a evolução do software é uma parte tão importante da engenharia de software e descrever os desafios de manter uma grande base de sistemas desenvolvidos ao longo de muitos anos. Ao ler este capítulo, você:

- compreenderá que os sistemas de software precisam se adaptar e evoluir para que continuem sendo úteis e que a modificação e a evolução de um software devem ser consideradas partes integrantes da engenharia de software;
- compreenderá o que significa dizer que os sistemas são legados e por que eles são importantes para as empresas;
- verá como os sistemas legados podem ser avaliados para decidir se devem ser descartados, mantidos, reprojetados ou substituídos;
- aprenderá sobre os diferentes tipos de manutenção de software e os fatores que afetam os custos de modificar sistemas de software legados.

CONTEÚDO

9.1 Processos de evolução
9.2 Sistemas legados
9.3 Manutenção de software

Os grandes sistemas de software possuem vida útil longa. Sistemas militares e de infraestrutura, como os de controle de tráfego aéreo, por exemplo, podem ter uma vida útil de 30 anos ou mais, enquanto que os sistemas de negócio costumam ter mais de dez anos de idade. Como o software corporativo costuma ser muito caro, uma empresa precisa usar seu sistema de software por muitos anos para que tenha retorno sobre o investimento. Produtos de software e aplicativos bem-sucedidos podem ter sido introduzidos muitos anos atrás, com novas versões lançadas de tempos em tempos. A primeira versão do Microsoft Word, por exemplo, foi introduzida em 1983; ele vem sendo usado, portanto, há mais de 30 anos.

Durante sua vida útil, os sistemas de software em operação precisam mudar se quiserem permanecer úteis. As mudanças nas empresas e nas expectativas dos usuários geram novos requisitos. As partes do software podem precisar de

alterações para corrigir erros encontrados durante a operação, para adaptá-lo às mudanças em suas plataformas de hardware e software e para melhorar seu desempenho ou outras características não funcionais. Produtos de software e aplicativos precisam evoluir para lidar com as mudanças de plataforma e as novas características introduzidas por seus concorrentes. Portanto, os sistemas de software se adaptam e evoluem ao longo de sua vida útil, desde a implantação inicial até a desativação.

As empresas precisam mudar o software que usam para garantir que ele continue a gerar valor. Seus sistemas são ativos comerciais críticos e elas devem investir na mudança para manter o valor desses ativos. Consequentemente, a maioria das grandes empresas gasta mais na manutenção dos sistemas existentes do que no desenvolvimento de novos sistemas. Dados históricos sugerem que entre 60% e 90% dos custos de software são relativos à evolução (LIENTZ; SWANSON, 1980; ERLIKH, 2000). Jones (2006) constatou que aproximadamente 75% dos profissionais de desenvolvimento nos Estados Unidos, no ano de 2006, estava envolvido na evolução de software, e sugeriu que essa porcentagem provavelmente não cairia tão cedo.

A evolução de um software é particularmente cara nos sistemas corporativos em que cada um dos sistemas faz parte de um 'sistema de sistemas' mais amplo. Nesses casos, não é possível considerar as mudanças em apenas um sistema; também é preciso examinar como elas afetam o sistema de sistemas mais amplo. Modificar um sistema pode significar que outros sistemas em seu ambiente também precisam evoluir para lidar com a mudança.

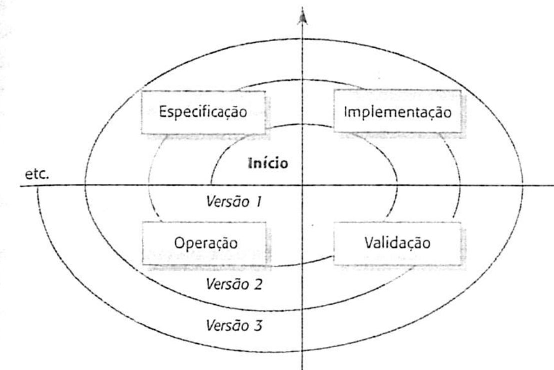
Portanto, além de compreender e analisar o impacto de uma proposta de mudança no próprio sistema, também é necessário avaliar como essa mudança pode afetar outros sistemas no ambiente operacional. Hopkins e Jenkins (2008) cunharam o termo *desenvolvimento de software brownfield* para descrever as situações nas quais os sistemas de software têm de ser desenvolvidos e gerenciados em um ambiente no qual dependem de outros sistemas de software.

Os requisitos dos sistemas de software instalados mudam conforme a empresa e seu ambiente mudam, então novos lançamentos dos sistemas são criados em intervalos regulares, incorporando mudanças e atualizações. Portanto, a engenharia de software é um processo em espiral em que os requisitos, o projeto, a implementação e os testes ocorrem durante toda a vida útil do sistema (Figura 9.1). O processo é iniciado com a criação da versão 1 do sistema. Depois de entregue, mudanças são propostas e o desenvolvimento da versão 2 começa quase imediatamente. Na verdade, a necessidade de evolução pode ficar óbvia antes mesmo de o sistema ser instalado; lançamentos posteriores do software podem começar a ser desenvolvidos antes de a versão atual ter sido lançada.

Nos últimos 10 anos, o tempo entre as iterações da espiral diminuiu radicalmente. Antes do uso generalizado da internet, novas versões de um sistema de software só eram lançadas a cada dois ou três anos. Hoje, devido à pressão da concorrência e à necessidade de reagir rapidamente ao *feedback* dos usuários, o intervalo entre os lançamentos de alguns aplicativos e sistemas web pode ser de semanas, em vez de anos.

Esse modelo de evolução do software é aplicável nos casos em que a mesma empresa é responsável pelo software durante toda a vida útil dele. Existe uma transição suave do desenvolvimento para a evolução, e os mesmos métodos e processos de

FIGURA 9.1 Um modelo em espiral do processo de desenvolvimento e evolução.



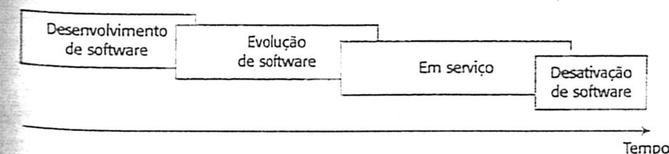
desenvolvimento de software são aplicados durante toda a sua vida útil. Os produtos de software e os aplicativos são desenvolvidos usando essa abordagem.

No entanto, a evolução de software personalizado normalmente segue um modelo diferente. O cliente do sistema pode pagar a uma empresa para desenvolver o software e depois assumir a responsabilidade por seu suporte e evolução usando sua própria equipe. Alternativamente, o cliente do software poderia assinar outro contrato, relativo a suporte e evolução, com uma empresa de software diferente.

Nesse caso, provavelmente há descontinuidades no processo de evolução. Os documentos de requisitos e de projeto (*design*) podem não ser passados de uma empresa para outra. As empresas podem se fundir ou se reorganizar, herdar um software de outras empresas e, então, descobrir que ele precisa ser modificado. Quando a transição do desenvolvimento para a evolução não é suave, o processo de modificação do software após a entrega recebe o nome de manutenção. Conforme discutirei mais adiante neste capítulo, a manutenção envolve atividades de processo extras, como a compreensão do programa, além das atividades normais de desenvolvimento de software.

Rajlich e Bennett (2000) propõem uma visão alternativa do ciclo de vida de evolução do software para sistemas de negócios. Nesse modelo, os autores fazem a distinção entre evolução e estar 'em serviço' (*servicing*, em inglês). Na fase de evolução, são feitas alterações significativas na arquitetura e na funcionalidade do software. Na fase 'em serviço', as únicas mudanças feitas são as relativamente pequenas, mas essenciais. Essas fases se sobrepõem umas às outras, como mostra a Figura 9.2.

FIGURA 9.2 Fases evolução e 'em serviço'.



Segundo Rajlich e Bennett, quando o software é utilizado com êxito pela primeira vez, muitas mudanças nos requisitos são propostas pelos *stakeholders*, e então são implementadas. A essa fase dá-se o nome de evolução. No entanto, à medida que o software é modificado, sua estrutura tende a se degradar, e as mudanças no sistema ficam cada vez mais caras. Frequentemente, isso acontece após alguns anos de uso, quando outras mudanças de ambiente, como as de hardware e as de sistemas operacionais, também se fazem necessárias. Em algum estágio no ciclo de vida, o software alcança um ponto de transição no qual as mudanças significativas e a implementação de novos requisitos têm um custo-benefício cada vez menor. Nesse estágio, o software passa da fase evolução para 'em serviço'.

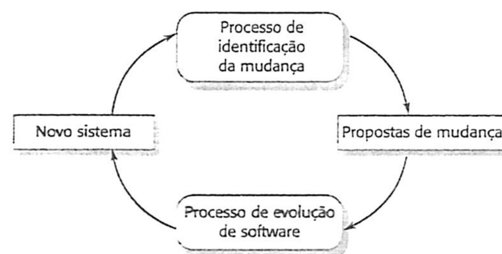
Durante a fase 'em serviço', o software ainda é útil, mas sofre pequenas alterações táticas. No decorrer desse estágio, a empresa geralmente está considerando como o software pode ser substituído. No estágio final, o software ainda pode ser utilizado, mas são feitas apenas as mudanças essenciais. Os usuários têm de contornar os problemas que descobrirem. No final das contas, o software é desativado. Muitas vezes, isso incorre em custos adicionais à medida que os dados são transferidos de um sistema antigo para um novo sistema substituto.

9.1 PROCESSOS DE EVOLUÇÃO

Assim como em todos os processos de software, não existe um processo padrão de mudança ou de evolução de software. O processo de evolução mais adequado para um sistema de software depende do tipo de software que está sendo mantido, dos processos de desenvolvimento de software utilizados em uma organização e das habilidades das pessoas envolvidas. Para alguns tipos de sistema, como aplicativos de celular, a evolução pode ser um processo informal, no qual as solicitações de mudança vêm principalmente de conversas entre os usuários e os desenvolvedores do sistema. Para outros tipos de sistema, como sistemas críticos embarcados, a evolução do software pode ser formalizada, com documentação estruturada produzida em cada etapa do processo.

As propostas de mudança formais e informais são a força motriz da evolução dos sistemas em todas as organizações. Em uma proposta de mudança, um indivíduo ou grupo sugere modificações e atualizações em um sistema de software existente. Essas propostas podem se basear em requisitos existentes que não foram implementados no sistema lançado, em solicitações de novos requisitos, nos relatórios de defeitos emitidos por *stakeholders* do sistema e em novas ideias do time de desenvolvimento para melhoria do software. Os processos de identificação da mudança e da evolução do sistema são cíclicos e continuam por toda a vida útil de um sistema (Figura 9.3).

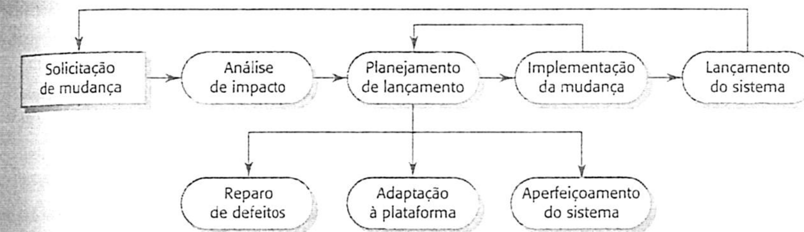
FIGURA 9.3 Processos de identificação da mudança e evolução.



Antes de uma proposta de mudança ser aceita, uma análise do software é necessária, para averiguar quais componentes precisam ser alterados. Essa análise permite a avaliação do custo e do impacto da mudança. Isso faz parte do processo geral de gerenciamento da mudança, que também deve assegurar que as versões corretas dos componentes sejam incluídas em cada lançamento do sistema. Discuto o gerenciamento de mudanças e de configuração no Capítulo 25.

A Figura 9.4 exibe algumas das atividades envolvidas na evolução do software. O processo inclui as atividades fundamentais de análise da mudança, planejamento de lançamento, implementação do sistema e lançamento do sistema para os clientes. O custo e o impacto dessas mudanças são avaliados para ver quanto do sistema é afetado pela mudança e quanto custaria implementá-la.

FIGURA 9.4 Modelo geral de processo de evolução do software.



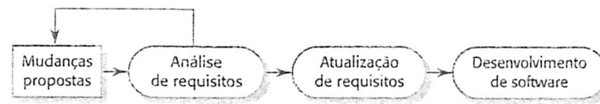
Se as mudanças propostas forem aceitas, um novo lançamento do sistema é planejado, e nesse processo, todas as mudanças propostas (reparo de defeitos, adaptação e nova funcionalidade) são consideradas. Depois, toma-se uma decisão sobre quais mudanças devem ser implementadas na próxima versão do sistema. As mudanças são implementadas e validadas, e uma nova versão do sistema é lançada. Então, esse processo é iterado com um novo conjunto de mudanças propostas para o próximo lançamento.

Nas situações em que o desenvolvimento e a evolução são integrados, a implementação da mudança é simplesmente uma iteração do processo de desenvolvimento. As revisões no sistema são projetadas, implementadas e testadas. A única diferença entre desenvolvimento inicial e evolução é que o *feedback* do cliente após a entrega precisa ser considerado no momento em que são planejados os novos lançamentos de uma aplicação.

Nas situações em que há envolvimento de times diferentes, uma diferença fundamental entre o desenvolvimento e a evolução é que o primeiro estágio da implementação da mudança requer uma compreensão do programa. Durante a fase de compreensão, os novos desenvolvedores precisam entender como ele está estruturado, como ele proporciona a funcionalidade e como a mudança proposta poderia afetá-lo. Eles precisam desse conhecimento para ter certeza de que a mudança a ser implementada não causará novos problemas quando for introduzida no sistema existente.

Se a especificação de requisitos e os documentos de projeto estiverem disponíveis, eles devem ser atualizados durante o processo de evolução para refletir as mudanças necessárias (Figura 9.5). Os novos requisitos de software devem ser escritos, analisados e validados. Se o projeto foi documentado com modelos em UML, eles devem ser atualizados. As mudanças propostas podem ser prototipadas como parte do processo de análise da mudança, no qual as implicações e os custos dessa mudança são avaliados.

FIGURA 9.5 Implementação da mudança.

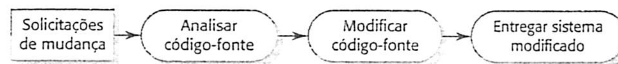


No entanto, as solicitações de mudança às vezes estão relacionadas a problemas nos sistemas em operação que precisam ser enfrentados urgentemente. Essas mudanças urgentes podem surgir por três motivos:

1. Se for detectado um defeito grave no sistema, que precise ser reparado para permitir a operação normal ou para tratar de uma grave vulnerabilidade de segurança da informação.
2. Se as mudanças no ambiente operacional dos sistemas tiverem efeitos inesperados que perturbem a operação normal.
3. Se houver mudanças imprevistas na empresa que executa o sistema, como o surgimento de novos concorrentes ou a introdução de nova legislação que afete o sistema.

Nesses casos, a necessidade de fazer a mudança rapidamente significa que pode não ser possível atualizar toda a documentação do software. Em vez de modificar os requisitos e o projeto, prioriza-se o reparo de emergência no programa para solucionar o problema imediato (Figura 9.6). O perigo aqui é que os requisitos, o projeto do software e o código podem ficar inconsistentes. Embora se possa querer documentar a mudança nos requisitos e no projeto, podem ser necessários outros reparos emergenciais no software. Esses reparos têm prioridade sobre a documentação. Eventualmente, a mudança original é esquecida, e a documentação e o código do sistema nunca mais se realinham. Esse problema de manter várias representações de um sistema é um dos argumentos para a documentação mínima, que é fundamental para os processos de desenvolvimento ágil.

FIGURA 9.6 Processo de reparo emergencial.



Os reparos emergenciais no sistema devem ser feitos o mais rápido possível. É escolhida uma solução rápida e exequível, em vez da melhor, no que diz respeito à estrutura do sistema, o que tende a acelerar o processo de envelhecimento do software de modo que as futuras mudanças ficam cada vez mais difíceis e os custos de manutenção aumentam. Em condições ideais, após a execução dos reparos emergenciais no código, o novo código deve ser refatorado e melhorado para evitar a degradação do programa. Naturalmente, o código de reparo pode ser reutilizado, se for possível. Entretanto, pode ser encontrada uma solução alternativa melhor para o problema quando houver mais tempo disponível para a análise.

Os métodos e processos ágeis, discutidos no Capítulo 3, podem ser utilizados para a evolução do programa e também no seu desenvolvimento. Como esses métodos se baseiam no desenvolvimento incremental, a transição do desenvolvimento ágil para a evolução pós-entrega deve ser suave.

No entanto, podem surgir problemas durante a passagem de um time de desenvolvimento para outro responsável pela evolução do sistema. Existem duas situações potencialmente problemáticas:

1. Se o time de desenvolvimento utilizou uma abordagem ágil, mas o time de evolução prefere uma abordagem dirigida por plano. O time de evolução pode estar esperando uma documentação detalhada para apoiar a evolução, mas isso raramente é produzido nos processos ágeis. Pode não haver uma declaração definitiva dos requisitos do sistema que possa ser modificada enquanto as mudanças são feitas no sistema.
2. Se foi utilizada uma abordagem dirigida por plano, mas o time de evolução prefere usar métodos ágeis. Nesse caso, o time de evolução pode ter de começar do zero, desenvolvendo testes automatizados. O código do sistema pode não ter sido refatorado e simplificado, como é esperado no desenvolvimento ágil. Nesse caso, pode ser necessária alguma reengenharia do programa para melhorar o código antes que ele seja utilizado em um processo de desenvolvimento ágil.

As técnicas ágeis, como o desenvolvimento dirigido por testes e o teste de regressão automatizado, são úteis quando há modificações no sistema. Essas modificações podem ser expressas como histórias de usuário, e o envolvimento do cliente pode ajudar a priorizar as mudanças necessárias em um sistema em operação. A abordagem do Scrum de focar em um *backlog* de trabalho a ser realizado pode ajudar a priorizar as mudanças mais importantes do sistema. Em suma, a evolução envolve simplesmente continuar o processo de desenvolvimento ágil.

No entanto, os métodos ágeis utilizados no desenvolvimento podem precisar de modificações quando forem utilizados na manutenção e na evolução do programa. Pode ser praticamente impossível envolver os usuários no time de desenvolvimento, pois as propostas de mudança vêm de uma ampla gama de *stakeholders*. Pode ser preciso interromper os ciclos curtos de desenvolvimento para lidar com os reparos de emergência, e o intervalo entre os lançamentos pode ter de ser prolongado para evitar a desestruturação dos processos operacionais.

9.2 SISTEMAS LEGADOS

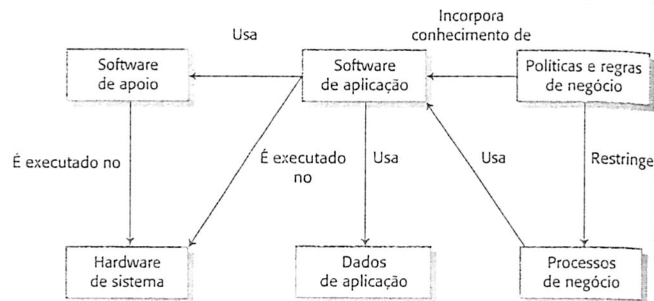
Grandes empresas começaram a informatizar suas operações nos anos 1960. Então, nos últimos 50 anos, aproximadamente, houve a introdução de cada vez mais software. Muitos desses sistemas foram substituídos (várias vezes em alguns casos) à medida que as empresas mudaram e evoluíram. No entanto, muitos sistemas antigos, chamados de sistemas legados, ainda estão em uso e desempenham um papel fundamental na operação dos negócios.

Os sistemas legados são mais antigos e se baseiam em linguagens e tecnologia que não são mais usadas no desenvolvimento dos novos sistemas. Geralmente, eles vêm sofrendo manutenção há muito tempo e sua estrutura pode ter sido degradada pelas modificações feitas. Os softwares legados podem depender de um hardware mais antigo, como os *mainframes*, e ter processos e procedimentos legados associados. Pode ser impossível mudar para processos de negócio mais eficazes porque o software legado não pode ser modificado para apoiar novos processos.

Os sistemas legados não são apenas sistemas de software, mas sistemas sociotécnicos mais amplos que incluem hardware, software, bibliotecas, software de apoio e

processos de negócio. A Figura 9.7 mostra as partes lógicas de um sistema legado e suas relações.

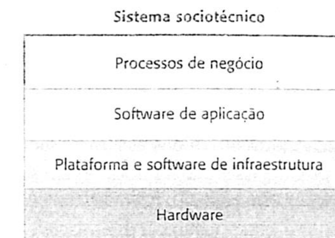
FIGURA 9.7 Elementos de um sistema legado.



1. *Hardware de sistema.* Os sistemas legados podem ter sido escritos para um hardware que não existe mais, que é caro para manter e que pode não ser compatível com as mais recentes políticas organizacionais de aquisição de TI.
2. *Software de apoio.* O sistema legado pode depender de uma gama de software de apoio, do sistema operacional e software utilitário fornecido pelo fabricante do hardware aos compiladores utilizados para o desenvolvimento do sistema. Mais uma vez, eles podem estar obsoletos e não mais receberem suporte de seus fornecedores originais.
3. *Software de aplicação.* O sistema de aplicação que fornece os serviços de negócio é composto normalmente de uma série de programas de aplicação que foram desenvolvidos em momentos diferentes. Alguns desses programas também farão parte de outros sistemas de aplicação.
4. *Dados de aplicação.* Esses dados são processados pelo sistema de aplicação. Em muitos sistemas legados, um imenso volume de dados se acumulou ao longo da vida útil do sistema. Esses dados podem estar inconsistentes, duplicados em vários arquivos e espalhados por uma série de bancos de dados diferentes.
5. *Processos de negócio.* São utilizados para atingir algum objetivo de negócio. Um exemplo de processo de negócio em uma empresa de seguros seria a emissão de uma apólice de seguros; em uma fábrica, um processo de negócio seria aceitar um pedido de produtos e configurar o processo de produção associado. Os processos de negócio podem ser concebidos em torno de um sistema legado e restringidos pela funcionalidade que ele proporciona.
6. *Políticas e regras de negócio.* São definições de como o negócio deve ser tocado e as suas limitações. O uso de um sistema de aplicação legado pode estar incorporado nessas políticas e regras.

Um modo alternativo de olhar para esses componentes de um sistema legado é como uma série de camadas, conforme mostra a Figura 9.8.

FIGURA 9.8 Camadas do sistema legado.



Cada camada do sistema depende da camada imediatamente abaixo dela e faz interface com ela. Se as interfaces forem mantidas, então deve ser possível fazer mudanças dentro de uma camada sem afetar nenhuma das camadas adjacentes. Na prática, porém, esse encapsulamento básico é uma simplificação excessiva e as mudanças em uma camada do sistema podem exigir mudanças consecutivas nas camadas acima e abaixo do nível modificado. As razões para isso são:

1. Modificar uma camada do sistema pode introduzir novos recursos, e as camadas mais altas do sistema podem ser modificadas para tirar proveito disso. Por exemplo, um novo banco de dados introduzido na camada de software de apoio pode incluir recursos para acessar os dados por meio de um navegador web, e os processos de negócio podem ser modificados para aproveitar esse recurso.
2. Alterar o software pode tornar o sistema mais lento, e novo hardware pode ser necessário para melhorar o desempenho desse sistema. O aumento no desempenho devido ao novo hardware pode significar que agora se tornam possíveis novas mudanças no software, que antes eram impraticáveis.
3. Muitas vezes é impossível manter as interfaces de hardware, especialmente se for introduzido um novo hardware. Esse é um problema particularmente no que diz respeito a sistemas embarcados, nos quais há uma forte ligação entre hardware e software. Podem ser necessárias alterações importantes no software de aplicação para que o novo hardware seja utilizado com eficácia.

É difícil saber exatamente quanto código legado ainda está em uso, mas, como um indicativo, a indústria estimou que existem mais de 200 bilhões de linhas de código em COBOL nos sistemas de negócio atuais. COBOL é uma linguagem de programação concebida para codificar sistemas de negócio, tendo sido a principal linguagem de desenvolvimento corporativo dos anos 1960 até os anos 1990, particularmente no setor financeiro (MITCHEL, 2012). Esses programas ainda funcionam com eficácia e eficiência, e as empresas que os utilizam não veem necessidade de mudá-los. No entanto, um grande problema que elas enfrentam é a escassez de programadores COBOL à medida que os desenvolvedores originais do sistema se aposentam. As universidades não ensinam mais COBOL, e os engenheiros de software mais jovens estão mais interessados nas linguagens de programação modernas.

A escassez de competência é só um dos problemas de manter sistemas de negócio legados. Outros problemas incluem vulnerabilidades de segurança da informação, já que esses sistemas foram desenvolvidos antes do uso generalizado da internet e dos problemas de interface com sistemas codificados em linguagens de programação

modernas. O fornecedor original da ferramenta de software pode ter desaparecido ou não manter mais as ferramentas de apoio utilizadas para desenvolver o sistema. O hardware do sistema pode estar obsoleto e, portanto, cada vez mais caro de manter.

Por que, então, as empresas não substituem seus sistemas por equivalentes mais modernos? A resposta simples para essa pergunta é que isso é caro e arriscado demais. Se um sistema legado funciona de forma eficaz, os custos de substituição podem exceder a economia proveniente dos custos de suporte reduzidos de um novo sistema. Descartar os sistemas legados e trocá-los por software moderno abre a possibilidade de as coisas darem errado e o novo sistema não satisfazer as necessidades da empresa. Os gestores tentam minimizar esses riscos e, portanto, não querem enfrentar as incertezas dos novos sistemas de software.

Descobri alguns dos problemas da substituição dos sistemas legados quando estive envolvido na análise de um projeto desse tipo em uma grande organização. Essa empresa usava mais de 150 sistemas legados para operar o seu negócio e decidiu substituir todos eles por um único sistema de ERP centralizado. Por uma série de razões tecnológicas e de negócio, a implantação do novo sistema foi um fracasso e não proporcionou as melhorias prometidas. Após gastar mais de £ 10 milhões, apenas uma parte do novo sistema estava operacional e funcionava com menos eficácia do que os sistemas que substituiu. Os usuários continuaram a usar os sistemas antigos, mas não conseguiram integrá-los com a parte do novo sistema que havia sido implantada, então foi necessário um processo manual adicional.

Existem várias razões para ser caro e arriscado substituir sistemas legados por sistemas novos:

1. Raramente há uma especificação completa do sistema legado. A especificação original pode ter se perdido. Se existir uma especificação, é improvável que tenha sido atualizada com todas as modificações feitas no sistema. Portanto, não existe uma maneira direta de especificar um novo sistema que seja funcionalmente idêntico ao que está em uso.
2. Os processos de negócio e as maneiras como os sistemas legados operam costumam estar indissociavelmente entrelaçados. Esses processos provavelmente evoluíram para tirar proveito dos serviços de software e contornar suas deficiências. Se o sistema for substituído, esses processos precisam mudar, incorrendo em custos e consequências potencialmente imprevisíveis.
3. Regras de negócio importantes podem estar incorporadas no software e não estar documentadas em outro lugar. Uma regra de negócio é uma restrição que se aplica a alguma função de negócio cuja violação pode ter consequências imprevisíveis para o negócio. Por exemplo, uma empresa de seguros pode ter suas regras embutidas para avaliar o risco de uma apólice em seu software. Se essas regras forem violadas, a empresa pode aceitar apólices de alto risco que poderiam resultar em caras reivindicações futuras.
4. O desenvolvimento de um novo software é inerentemente arriscado, então pode haver problemas inesperados com um sistema novo. Ele pode não ser entregue dentro do prazo ou da estimativa de preço prevista.

Manter sistemas legados em uso evita os riscos da substituição, mas inevitavelmente encarece as mudanças no software existente à medida que os sistemas envelhecem. A alteração dos sistemas de software legados com alguns anos de idade é particularmente onerosa:

1. O estilo do programa e as convenções de uso são incoerentes porque diferentes pessoas foram responsáveis pelas mudanças no sistema. Esse problema aumenta a dificuldade de compreender o código do sistema.
2. O sistema pode, em parte ou como um todo, ser implementado com linguagens de programação obsoletas. Pode ser difícil encontrar pessoas que tenham conhecimento dessas linguagens. Portanto, terceirizar a manutenção do software pode ser necessário, e isso é caro.
3. A documentação do sistema frequentemente é inadequada e obsoleta. Em alguns casos, a única documentação é o código-fonte do sistema.
4. Muitos anos de manutenção costumam degradar a estrutura do sistema, tornando-o cada vez mais difícil entender. Novos programas podem ter sido adicionados e feito interface de uma forma especial com outras partes do sistema.
5. O sistema pode ter sido otimizado para utilização do espaço ou para velocidade de execução a fim de ser executado de modo eficaz no hardware mais antigo e lento. Isso normalmente envolve o uso de otimizações específicas de máquina e de linguagens, que costuma levar a um software difícil de entender, causando problemas para os programadores que aprenderam técnicas de engenharia de software modernas e que não vão entender os truques de programação que foram utilizados para otimizar o software.
6. Os dados processados pelo sistema podem ser mantidos em diferentes arquivos com estruturas incompatíveis. Pode haver duplicação de dados, e eles próprios podem ser obsoletos, imprecisos e incompletos. Podem ser usados vários bancos de dados de diferentes fornecedores.

No mesmo estágio, o custo de gerenciar e manter o sistema legado fica tão alto que ele precisa ser substituído por um novo sistema. Na próxima seção, discutirei uma abordagem sistemática para a tomada de decisão quanto a essa substituição do sistema legado.

9.2.1 Gerenciamento de sistemas legados

Para os novos sistemas desenvolvidos usando processos modernos de engenharia de software, tais quais o desenvolvimento ágil e as linhas de produto de software, é possível planejar como integrar o desenvolvimento e a evolução do sistema. Cada vez mais empresas compreendem que o processo de desenvolvimento de sistemas é um ciclo de vida completo. É inútil separar o desenvolvimento de software da evolução, além de levar a custos mais altos. No entanto, conforme discuti, ainda há uma quantidade imensa de sistemas legados que são sistemas de negócio críticos. Eles têm de ser ampliados e adaptados para as práticas de *e-business* em constante mudança.

A maioria das organizações tem um orçamento limitado para manter e atualizar seu portfólio de sistemas legados e, por isso, devem decidir como auferir o melhor retorno sobre o investimento. Isso envolve fazer uma avaliação realista de seus sistemas legados e depois definir a estratégia mais adequada para desenvolver esses sistemas. Existem quatro pontos estratégicos:

1. *Descartar completamente o sistema*. Essa deve ser a opção se o sistema não estiver contribuindo efetivamente para os processos de negócio. Normalmente isso

acontece quando os processos de negócio mudaram desde a instalação do sistema e não dependem mais do sistema legado.

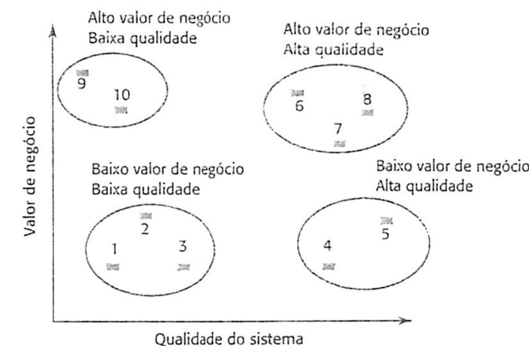
2. *Deixar o sistema intacto e continuar com a manutenção regular.* Essa é a opção se o sistema ainda for necessário, mas for razoavelmente estável e seus usuários fizerem relativamente poucas solicitações de mudança.
3. *Promover a reengenharia do sistema para melhorar sua manutenibilidade.* Essa deve ser a opção nos casos em que a qualidade do sistema foi degradada pela mudança e novas mudanças no sistema ainda estão sendo propostas. Esse processo pode incluir o desenvolvimento de novos componentes de interface, de modo que o sistema original possa trabalhar com outros sistemas mais novos.
4. *Substituir todo ou parte do sistema por um sistema novo.* Essa deve ser a opção escolhida quando fatores — como um novo hardware, por exemplo — impedem que o sistema antigo continue em operação, ou quando sistemas de prateleira permitem que o novo sistema seja desenvolvido a um custo razoável. Em muitos casos, pode ser adotada uma estratégia de substituição evolucionária, na qual os principais componentes do sistema são trocados por sistemas de prateleira com outros componentes reusados sempre que possível.

Ao avaliar um sistema legado, é necessário examiná-lo por uma perspectiva de negócio e uma perspectiva técnica (WARREN, 1998). Pela perspectiva de negócio, é preciso decidir se a empresa realmente precisa do sistema. Pela perspectiva técnica, é preciso avaliar a qualidade do software de aplicação, assim como o software e o hardware de apoio ao sistema. Então, se usa uma combinação do valor de negócio e da qualidade do sistema para fundamentar a decisão sobre o que fazer com o sistema legado.

Por exemplo, suponha que uma organização tenha dez sistemas legados. É preciso avaliar a qualidade e o valor de negócio de cada um desses sistemas e depois criar um gráfico apontando valor de negócio e qualidade do sistema relativos. Um exemplo disso é exibido na Figura 9.9. Partindo do diagrama, é possível perceber que existem quatro grupos de sistemas:

1. *Baixa qualidade, baixo valor de negócio.* Esses sistemas devem ser descartados, pois mantê-los em operação será oneroso e a taxa de retorno para a empresa será relativamente pequena.
2. *Baixa qualidade, alto valor de negócio.* Esses sistemas contribuem de forma importante para a empresa, então não podem ser descartados. No entanto, sua baixa qualidade significa que são caros de manter. Esses sistemas devem passar por uma reengenharia para melhorar sua qualidade ou podem ser substituídos, se houver sistemas de prateleira adequados à disposição.
3. *Alta qualidade, baixo valor de negócio.* Esses sistemas não contribuem para o negócio da empresa, mas podem ter uma manutenção mais barata. Não vale a pena substituí-los, então a manutenção normal pode continuar se não forem necessárias mudanças onerosas e o hardware do sistema continuar em uso. caso contrário, o software deve ser descartado.
4. *Alta qualidade, alto valor de negócio.* Esses sistemas devem ser mantidos em operação. No entanto, sua alta qualidade significa que não é preciso investir na transformação ou substituição do sistema. A manutenção normal do sistema deve continuar.

FIGURA 9.9 Exemplo de avaliação de sistema legado.



O valor de negócio de um sistema é uma medida de quanto tempo e esforço ele poupa em comparação com os processos manuais ou com o uso de outros sistemas. Para avaliar o valor de negócio de um sistema, devemos identificar seus *stakeholders* — os usuários finais e seus gerentes, por exemplo —, e fazer uma série de perguntas. Existem quatro questões básicas que devem ser discutidas:

1. *O uso do sistema.* Se um sistema for utilizado apenas ocasionalmente ou por um pequeno número de pessoas, isso pode significar que ele tem um baixo valor de negócio. Um sistema legado pode ter sido desenvolvido para satisfazer uma necessidade de negócio que mudou ou que agora pode ser atendida com mais eficácia de outras maneiras. No entanto, é preciso ter cuidado com o uso ocasional, embora importante, dos sistemas. Um sistema universitário para matrícula de alunos, por exemplo, é utilizado apenas no início de cada ano acadêmico; embora seu uso seja esporádico, é um sistema essencial e com alto valor de negócio.
2. *Os processos de negócio apoiados.* Quando um sistema é introduzido, os processos de negócio normalmente são introduzidos para aproveitar as capacidades do sistema. Se ele for inflexível, esses processos podem ser impossíveis de mudar. No entanto, à medida que o ambiente muda, os processos de negócio originais podem ficar obsoletos. Um sistema pode ter um baixo valor de negócio, portanto, porque força a utilização de processos de negócio ineficientes.
3. *Dependabilidade do sistema.* A dependabilidade do sistema não é um problema apenas técnico, mas também de negócio. Se um sistema não for confiável e seus problemas afetarem diretamente os clientes da empresa, seu valor de negócio é baixo.
4. *Os resultados do sistema.* A questão-chave aqui é a importância dos resultados do sistema para o bom funcionamento da empresa. Se ela depender desses resultados, então o sistema tem um alto valor de negócio. Por outro lado, se esses resultados puderem ser gerados de forma barata por outros meios, ou se o sistema produzir resultados raramente utilizados, então ele tem um baixo valor de negócio.

Suponha, por exemplo, que uma empresa forneça um sistema de reservas de viagem utilizado pelo pessoal responsável por organizá-las. Eles podem fazer reservas com um agente de viagens licenciado. As passagens são entregues e a empresa

recebe por elas. No entanto, uma avaliação do valor de negócio pode revelar que esse sistema é utilizado em uma porcentagem bem pequena dos pedidos de viagem. As pessoas que organizam as viagens acham mais barato e conveniente lidar diretamente com os fornecedores por meio de seus sites. Esse sistema ainda pode ser utilizado, mas não vale a pena mantê-lo — a mesma funcionalidade está disponível em sistemas externos.

Por outro lado, digamos que uma empresa tenha desenvolvido um sistema que controle todos os pedidos anteriores do cliente e gere lembretes automaticamente para que ele encomende novamente os produtos. Isso resulta em uma grande quantidade de pedidos repetidos e mantém o cliente satisfeito, pois ele sente que o fornecedor está a par de suas necessidades. As saídas de um sistema como esse são importantes para o negócio, então esse sistema tem um alto valor de negócio.

Para avaliar um sistema de software pela perspectiva técnica, é preciso considerar o próprio sistema de aplicação e o ambiente no qual o sistema opera. O ambiente inclui o hardware e todo o software de apoio associado, como compiladores, depuradores e ambientes de desenvolvimento que são necessários para manter o sistema. O ambiente é importante porque muitas mudanças no sistema, como atualizações no hardware ou no sistema operacional, resultam de mudanças no ambiente.

Os fatores que devem ser considerados durante a avaliação do ambiente são exibidos na Figura 9.10. Repare que nem todos esses fatores são características técnicas do ambiente. Também é preciso considerar a confiabilidade dos fornecedores do hardware e do software de apoio. Se os fornecedores não estiverem mais no negócio, os sistemas podem não ser mais suportados e deverão ser substituídos.

FIGURA 9.10 Fatores utilizados na avaliação do ambiente.

Fator	Perguntas
Estabilidade do fornecedor	O fornecedor ainda existe? O fornecedor é financeiramente estável e propenso a continuar existindo? Se o fornecedor tiver falido, alguém mais mantém os sistemas?
Taxa de falha	O hardware tem uma alta taxa de falhas? O software de apoio falha e força o sistema a reiniciar?
Idade	Quais são as idades do hardware e do software? Quanto mais velhos o hardware e o software de apoio, mais obsoletos eles serão. Eles ainda podem funcionar corretamente, mas poderia haver benefícios econômicos e de negócio significativos em passar para um sistema mais moderno.
Desempenho	O desempenho do sistema é adequado? Os problemas de desempenho têm um efeito significativo nos usuários do sistema?
Requisitos de suporte	Qual é o suporte local necessário exigido pelo hardware e pelo software? Se esse suporte implicar em custos elevados, pode valer a pena considerar a substituição do sistema.
Custos de manutenção	Quais são os custos de manutenção do hardware e das licenças do software de apoio? O hardware mais antigo pode ter custos de manutenção mais altos do que os sistemas modernos. O software de apoio pode ter custos de licenciamento anual elevados.
Interoperabilidade	Existem problemas quando o sistema faz interface com os outros sistemas? Os compiladores, por exemplo, podem ser utilizados com versões atuais do sistema operacional?

No processo de avaliação do ambiente, se for possível, é ideal coletar dados sobre o sistema e suas mudanças. Os exemplos de dados que podem ser úteis incluem os custos de manter o hardware do sistema e o software de apoio, o número de defeitos de hardware que ocorrem ao longo de um determinado período de tempo e a frequência das correções e reparos aplicados ao software de apoio do sistema.

Para avaliar a qualidade técnica de um sistema de aplicação, é necessário avaliar fatores (Figura 9.11) relacionados principalmente com a dependabilidade, dificuldades de manutenção do sistema e sua documentação. Também é possível coletar dados que ajudarão a julgar a qualidade do sistema, como:

1. *O número de solicitações de mudança no sistema.* As mudanças no sistema costumam corromper sua estrutura e tornam mudanças posteriores mais difíceis. Quanto maior esse valor acumulado, menor a qualidade do sistema.
2. *O número de interfaces com o usuário.* Esse é um fator importante nos sistemas baseados em formulário: cada formulário pode ser considerado uma interface com o usuário diferente. Quanto mais interfaces, mais provável que existam inconsistências e redundâncias nessas interfaces.
3. *O volume de dados utilizados pelo sistema.* À medida que o volume de dados (número de arquivos, tamanho do banco de dados etc.) processados pelo sistema aumenta, também aumentam as inconsistências e erros nesses dados. Quando os dados foram coletados ao longo de um período de tempo, os erros e as inconsistências são inevitáveis. Limpar dados antigos é um processo muito caro e demorado.

FIGURA 9.11 Fatores utilizados na avaliação da aplicação.

Fator	Perguntas
Compreensibilidade	Qual é o grau de dificuldade para compreender o código-fonte do sistema atual? Qual é a complexidade das estruturas de controle utilizadas? As variáveis têm nomes significativos que refletem sua função?
Documentação	Qual é a documentação de sistema disponível? A documentação está completa, atualizada e coerente?
Dados	Existe um modelo de dados explícito para o sistema? Até que ponto os dados estão duplicados nos arquivos? Os dados utilizados pelo sistema estão atualizados e são consistentes?
Desempenho	O desempenho da aplicação é adequado? Os problemas de desempenho têm um efeito significativo nos usuários do sistema?
Linguagem de programação	Existem compiladores modernos disponíveis para a linguagem de programação utilizada para desenvolver o sistema? A linguagem de programação ainda é utilizada no desenvolvimento de novos sistemas?
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 dos componentes que são utilizados no sistema atual?
Dados de teste	Existem dados de teste do sistema? Existe um registro dos testes de regressão executados quando novas características foram acrescentadas ao sistema?
Habilidades do pessoal	Há pessoas disponíveis que tenham habilidades para manter a aplicação? Existem pessoas disponíveis que tenham experiência no sistema?

Em condições ideais, deve-se utilizar uma avaliação objetiva para fundamentar as decisões sobre o que fazer com o sistema legado. No entanto, em muitos casos, essas decisões não são realmente objetivas, mas baseadas em considerações organizacionais ou políticas. Por exemplo, se duas empresas se fundem, a parceira politicamente mais poderosa normalmente vai manter seus sistemas e descartar os da outra empresa. Se a alta gestão em uma organização decidir passar para uma nova plataforma de hardware, então isso pode exigir que as aplicações sejam substituídas.

Se não houver orçamento disponível para a transformação do sistema em um determinado ano, então sua manutenção pode continuar, embora isso vá resultar em custos mais altos no longo prazo.

Dinâmica da evolução de programas

A dinâmica da evolução de programas é o estudo da evolução dos sistemas de software, cujos pioneiros foram Lehman e Les Belady nos anos 1970. Isso levou às conhecidas Leis de Lehman, que, segundo consta, são aplicadas a todos os sistemas de software de larga escala. As leis mais importantes são:

1. Um programa deve mudar continuamente para continuar sendo útil.
2. À medida que um programa em desenvolvimento muda, sua estrutura se degrada.
3. Ao longo da vida útil de um programa, a taxa de mudança é aproximadamente constante e independente dos recursos disponíveis.
4. A mudança incremental em cada lançamento de um sistema é aproximadamente constante.
5. Novas funcionalidades devem ser acrescentadas aos sistemas para aumentar a satisfação do usuário.

9.3 MANUTENÇÃO DE SOFTWARE

Manutenção de software é o processo geral de modificar um sistema após a sua entrega. O termo é aplicado geralmente ao software personalizado, no qual diferentes grupos de desenvolvimento estão envolvidos antes e depois da entrega. As mudanças feitas no software podem ser simples, para corrigir erros de código; mais extensas, para corrigir erros de projeto; ou significativas, para corrigir erros de especificação ou acomodar novos requisitos. As mudanças são implementadas modificando os componentes do sistema existente e, onde for necessário, adicionando novos componentes ao sistema.

Existem três tipos diferentes de manutenção de software:

1. *Reparo de defeitos para corrigir bugs e vulnerabilidades.* Os erros de codificação costumam ser relativamente baratos de corrigir; os erros de projeto são mais caros, porque podem envolver reescrever vários componentes do programa; os erros de requisitos são os mais caros para consertar, porque pode ser necessário projetar novamente o sistema.
2. *Adaptação ao ambiente para adaptar o software a novas plataformas e ambientes.* Esse tipo de manutenção é necessário quando se muda algum aspecto do ambiente de um sistema, como o hardware, o sistema operacional da plataforma ou outro software de apoio. Os sistemas de aplicação podem precisar de adaptações para lidar com essas mudanças de ambiente.
3. *Acréscimo de funcionalidade para adicionar novas características e apoiar novos requisitos.* Esse tipo de manutenção é necessário quando os requisitos do sistema mudam em resposta à mudança organizacional ou de negócios. A escala das mudanças necessárias no software frequentemente é muito maior do que as dos outros tipos de manutenção.

Na prática, não há uma distinção nítida entre esses tipos de manutenção. Quando se adapta um sistema a um novo ambiente, é possível adicionar funcionalidade para tirar vantagem das novas características do ambiente. Os defeitos de software são expostos frequentemente porque os usuários utilizam o sistema de maneiras imprevistas, e a melhor forma de corrigir esses defeitos é mudar o sistema a fim de acomodar a forma como eles trabalham.

De modo geral, esses tipos de manutenção são reconhecidos, mas pessoas diferentes podem lhes dar nomes diferentes. 'Manutenção corretiva' é utilizado universalmente para se referir à manutenção para reparar defeitos. No entanto, 'manutenção adaptativa' às vezes significa adaptar o software a um novo ambiente; em outros casos, significa adaptá-lo a novos requisitos. 'Manutenção perfectiva' às vezes significa aperfeiçoar o software implementado novos requisitos; em outros casos, significa manter a funcionalidade do sistema, mas melhorar a sua estrutura e o seu desempenho. Devido a essa incerteza de nomenclatura, evitei usar esses termos neste livro.

A Figura 9.12 mostra uma distribuição aproximada dos custos de manutenção, com base em dados da pesquisa mais recente de Davidsen e Krogstie (2010). Esse estudo comparou a distribuição do custo de manutenção com uma série de estudos anteriores, realizados de 1980 a 2005. Os autores constataram que a distribuição dos custos de manutenção mudou pouco ao longo de 30 anos. Embora não tenhamos dados mais recentes, isso sugere que essa distribuição ainda é, em grande parte, correta. Corrigir defeitos do sistema não é a atividade de manutenção mais cara. Evoluir o sistema para lidar com ambientes novos, requisitos novos ou modificados geralmente consome a maior parte do esforço de manutenção.

FIGURA 9.12 Distribuição do esforço de manutenção.

A experiência mostrou que normalmente é mais caro acrescentar novas características a um sistema durante a manutenção do que implementá-las durante o desenvolvimento inicial. As razões para isso são:

1. *Um novo time precisa entender o programa que está sendo mantido.* Depois que um sistema é entregue, é normal que o time de desenvolvimento se desfaça e as pessoas passem a trabalhar com novos projetos. O novo time ou os indivíduos responsáveis pela manutenção do sistema não entendem o sistema ou o contexto das decisões de seu projeto. Eles precisam investir tempo entendendo o sistema existente antes de implementar mudanças nele.

2. *A separação entre a manutenção e o desenvolvimento significa que não há incentivo para o time de desenvolvimento escrever um software de fácil manutenção.* O contrato para manter um sistema normalmente é separado do contrato de desenvolvimento do sistema. Uma empresa diferente, em vez de o desenvolvedor original, pode ser responsável pela manutenção do software. Nessas circunstâncias, um time de desenvolvimento não se beneficia ao investir esforço para tornar o software de fácil manutenção. Se um time de desenvolvimento puder usar atalhos para poupar esforço durante o desenvolvimento, vale a pena fazê-lo, mesmo se isso significar que o software será mais difícil de mudar no futuro.
3. *O trabalho de manutenção de programa não é popular.* A manutenção tem uma imagem negativa entre os engenheiros de software. Ela é vista como um processo menos qualificado que o desenvolvimento de sistemas e frequentemente é alocada para pessoal menos experiente. Além disso, os sistemas antigos podem ter sido escritos em linguagens de programação obsoletas. Os desenvolvedores que trabalham na manutenção podem não ter muita experiência com essas linguagens e devem aprendê-las para manter o sistema.
4. *À medida que os programas envelhecem, sua estrutura se degrada e eles ficam mais difíceis de modificar.* Ao passo que são feitas alterações nos programas, sua estrutura tende a se degradar. Consequentemente, eles ficam mais difíceis de serem entendidos e modificados. Alguns sistemas foram desenvolvidos sem técnicas modernas de engenharia de software. É possível que eles não tenham sido bem estruturados e talvez tenham sido otimizados para eficiência, em vez de inteligibilidade. A documentação do sistema pode ter se perdido ou ser inconsistente. Sistemas antigos podem não ter sido submetidos a um gerenciamento de configuração rigoroso, então os desenvolvedores têm de investir tempo encontrando as versões certas dos componentes do sistema para alterar.

Documentação

A documentação do sistema pode ajudar no processo de manutenção, fornecendo aos responsáveis pela manutenção informações sobre a estrutura e a organização do sistema e sobre as características que ele oferece aos usuários. Embora os proponentes das abordagens ágeis sugiram que o código deve ser a documentação principal, os modelos de projeto de alto nível e as informações sobre as dependências e restrições podem fazer com que fique mais fácil compreender e alterar esse código.

Os três primeiros problemas decorrem do fato de que muitas organizações ainda consideram o desenvolvimento e a manutenção de software atividades separadas. A manutenção é vista como uma atividade de segunda classe, e não há incentivo para gastar dinheiro durante o desenvolvimento para reduzir os custos da modificação do sistema. A única solução de longo prazo para esses problemas é considerar que os sistemas evoluem por toda a sua vida útil por meio de um processo de desenvolvimento contínuo. A manutenção deveria ter um status tão elevado quanto o do desenvolvimento de um novo software.

O quarto problema, a degradação da estrutura do sistema, é, de algumas formas, o mais fácil de resolver. As técnicas de reengenharia de software (descritas mais adiante neste capítulo) podem ser aplicadas para melhorar a estrutura e a inteligibilidade do sistema. Transformações de arquitetura podem adaptar o sistema ao novo hardware. A refatoração pode melhorar a qualidade do código do sistema e facilitar sua alteração.

A princípio, quase sempre é economicamente compensador investir esforços no projeto e na implementação de um sistema para reduzir os custos de alterações futuras. É oneroso acrescentar nova funcionalidade após a entrega, pois é necessário investir tempo aprendendo o sistema e analisando o impacto das alterações propostas. O trabalho feito durante o desenvolvimento, com o objetivo de estruturar o software para torná-lo mais fácil de ser entendido e modificado, diminuirá os custos da evolução. Boas técnicas de engenharia de software, como a especificação precisa, o desenvolvimento com testes *a priori* (*test-first*), o uso de desenvolvimento orientado a objetos e o gerenciamento de configuração ajudam a reduzir o custo de manutenção.

Os argumentos desses bons princípios para a economia no longo prazo, pelo investimento em tornar os sistemas mais fáceis de manter, infelizmente são impossíveis de fundamentar com dados reais. Coletar dados é caro, e o valor desses dados é difícil de julgar; portanto, a ampla maioria das empresas não acha que valha a pena reunir e analisar dados de engenharia de software.

Na realidade, a maioria das empresas reluta em gastar mais no desenvolvimento de software para reduzir os custos de manutenção no longo prazo. Existem duas razões principais para essa relutância:

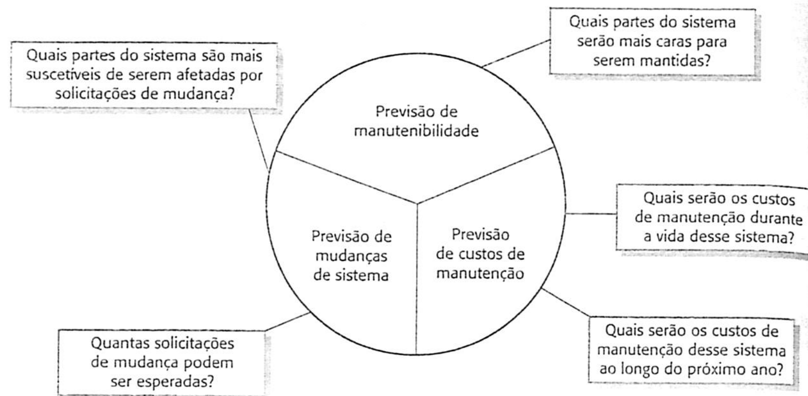
1. As empresas fazem planos de investimentos trimestrais ou anuais, e os gestores são incentivados a reduzir os custos de curto prazo. Investir em manutenibilidade faz com que os custos no curto prazo, que são mensuráveis, cresçam. No entanto, os ganhos de longo prazo não podem ser medidos no mesmo tempo, então as empresas relutam em gastar dinheiro com algo que tenha um retorno futuro desconhecido.
2. Os desenvolvedores normalmente não são responsáveis por manter o sistema que desenvolveram. Consequentemente, eles não veem vantagens em fazer o trabalho adicional que poderia reduzir os custos de manutenção, já que não obterão qualquer benefício dele.

A única maneira de contornar esse problema é integrar o desenvolvimento e a manutenção para que o time de desenvolvimento original continue responsável pelo software durante toda a sua vida útil. Isso é possível para produtos de software e empresas como a Amazon, que desenvolve e mantém o seu próprio software (O'HANLON, 2006). Entretanto, para o software personalizado, desenvolvido e mantido por uma empresa de software para um cliente, é improvável que isso aconteça.

9.3.1 Previsão de manutenção

A previsão de manutenção se preocupa em tentar avaliar as mudanças que podem ser necessárias em um sistema de software e em identificar as partes do sistema que tendem a ser mais caras para alterar. Se entender isso, você pode projetar os componentes de software de forma que sejam mais fáceis de modificar, tornando-os mais adaptáveis. Você também pode investir esforços em melhorar os componentes para reduzir seus custos de manutenção durante a vida do sistema. Ao prever as mudanças, você também pode avaliar os custos de manutenção globais de um sistema em um determinado período de tempo e estabelecer um orçamento para manter o software. A Figura 9.13 mostra as possíveis previsões e as perguntas que elas podem responder.

FIGURA 9.13 Previsão de manutenção.



Prever o número de solicitações de mudança de um sistema requer uma compreensão da relação entre esse sistema e o seu ambiente externo. Alguns sistemas têm uma relação muito complexa com seu ambiente externo, e as mudanças nesse ambiente resultam, inevitavelmente, em mudanças no sistema. Para avaliar as relações entre um sistema e o ambiente, deve-se examinar:

1. *A quantidade e a complexidade das interfaces do sistema.* Quanto maior a quantidade de interfaces e quanto mais complexas elas forem, é mais provável que haja modificações à medida que novos requisitos sejam propostos.
2. *A quantidade de requisitos de sistema inerentemente voláteis.* Conforme discuti no Capítulo 4, os requisitos que refletem as políticas e os procedimentos organizacionais tendem a ser mais voláteis do que os requisitos que se baseiam em características estáveis de uma área.
3. *Os processos de negócio nos quais o sistema é utilizado.* À medida que os processos de negócio evoluem, eles geram solicitações de mudança no sistema. Quando um sistema é integrado a cada vez mais processos de negócio, há mais demandas por mudanças.

Nos primeiros trabalhos de manutenção de software, pesquisadores examinaram as relações entre a complexidade do programa e a sua manutenibilidade (BANKER *et al.*, 1993; COLEMAN *et al.*, 2008). Esses estudos constataram que, quanto mais complexo um sistema ou componente, mais cara é a sua manutenção. As medições da complexidade são particularmente úteis na identificação de componentes do programa que provavelmente terão uma manutenção cara. Portanto, para reduzir os custos de manutenção, deve-se tentar substituir os componentes complexos do sistema por alternativas mais simples.

Depois que um sistema foi colocado em serviço, dados de processo podem ser usados para ajudar a prever a manutenibilidade. Exemplos de métricas que podem ser utilizadas para avaliar a manutenibilidade são:

1. *Quantidade de solicitações de manutenção corretiva.* Um aumento no número de relatos de defeitos e falhas pode indicar que mais erros estão sendo introduzidos no programa do que estão sendo corrigidos durante o processo de manutenção. Isso pode indicar um declínio na manutenibilidade.
2. *Tempo médio necessário para a análise de impacto.* Está relacionado com a quantidade de componentes do programa que são afetados pela solicitação de mudança. Se o tempo necessário para a análise de impacto aumentar, isso implica que cada vez mais componentes estão sendo afetados e que a manutenibilidade está diminuindo.
3. *O tempo médio para implementar uma solicitação de mudança.* Esse tempo não é igual ao da análise de impacto, embora possa ter alguma correlação com ele. Essa é a quantidade de tempo que você precisa para modificar o sistema e a sua documentação após ter avaliado os componentes afetados. Um aumento no tempo necessário para implementar uma mudança pode indicar um declínio da manutenibilidade.
4. *Quantidade de solicitações de mudança pendentes.* Um aumento nesse número ao longo do tempo pode implicar em uma diminuição da manutenibilidade.

É possível usar as informações previstas sobre as solicitações de mudança e as previsões sobre a manutenibilidade do 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 COCOMO II de estimativa do custo, que será discutido no Capítulo 23, sugere que uma estimativa do esforço de manutenção de software pode se basear no esforço para entender o código existente e no esforço para desenvolver um novo código.

9.3.2 Reengenharia de software

A manutenção de software envolve entender o programa que precisa ser alterado e depois implementar quaisquer alterações necessárias. No entanto, muitos sistemas, especialmente os sistemas legados mais antigos, são difíceis de entender e de alterar. Os programas podem tanto ter sido otimizados para melhor desempenho ou melhor utilização de espaço, em detrimento de sua compreensibilidade, quanto tido sua estrutura inicial corrompida com o passar do tempo por uma série de alterações.

Para tornar os sistemas de software legados mais fáceis de manter, é possível fazer a reengenharia desses sistemas, a fim de melhorar sua estrutura e sua compreensibilidade. Esse processo pode envolver nova documentação do sistema, refatoração da arquitetura do sistema, tradução dos programas para uma linguagem de programação mais moderna ou modificação e atualização da estrutura e dos valores dos dados do sistema. A funcionalidade do software não muda e, normalmente, deve-se evitar grandes alterações na arquitetura do sistema.

A reengenharia tem duas vantagens importantes sobre a substituição:

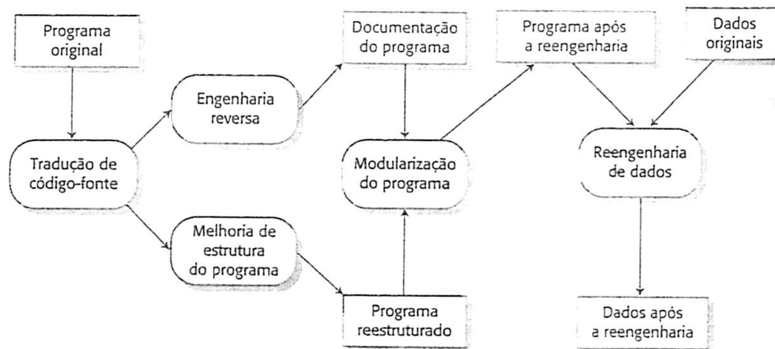
1. *Menor risco.* Existe um risco alto em redesenvolver um software crítico para o negócio. Podem ser cometidos erros na especificação do sistema ou pode haver problemas de desenvolvimento. Os atrasos na introdução do novo software podem significar a perda do negócio e custos extras.
2. *Menor custo.* O custo de reengenharia pode ser significativamente menor que o custo de desenvolver um novo software. Ulrich (1990) cita o exemplo de um sistema comercial para o qual os custos de reimplementação foram estimados

em US$ 50 milhões. O sistema passou por uma reengenharia bem-sucedida no valor de US$ 12 milhões. Suspeito que, com a moderna tecnologia de software, o custo relativo da reimplementação provavelmente é menor do que o número de Ulrich, mas ainda será maior que o custo da reengenharia.

A Figura 9.14 é um modelo geral do processo de reengenharia. A entrada para o processo é um programa legado, e a saída é uma versão melhorada e reestruturada do mesmo programa. As atividades no processo de reengenharia são:

1. *Tradução de código-fonte.* Usando uma ferramenta de tradução, é possível converter o programa de uma linguagem de programação antiga para uma versão mais moderna da mesma linguagem, ou mesmo para uma linguagem diferente.
2. *Engenharia reversa.* O programa é analisado e as informações dele são extraídas. Isso ajuda a documentar sua organização e funcionalidade. Mais uma vez, esse processo costuma ser completamente automatizado.
3. *Melhoria de estrutura do programa.* A estrutura de controle do programa é analisada e modificada para facilitar sua leitura e compreensão. Isso pode ser parcialmente automatizado, mas é necessária alguma intervenção manual.
4. *Modularização do programa.* As partes relacionadas do programa são agrupadas e, nos casos adequados, a redundância é removida. Em alguns casos, esse estágio pode envolver a refatoração da arquitetura (por exemplo, um sistema que usa vários repositórios de dados diferentes pode ser refatorado para usar um único repositório). Esse é um processo manual.
5. *Reengenharia de dados.* Os dados processados pelo programa são modificados para refletir as alterações do programa. Isso pode significar a redefinição dos esquemas de bancos de dados e a conversão dos bancos de dados existentes para uma estrutura nova. Normalmente, os dados também devem ser limpos, o que envolve encontrar e corrigir erros, remover registros duplicados etc. Esse pode ser um processo muito caro e demorado.

FIGURA 9.14 O processo de reengenharia.

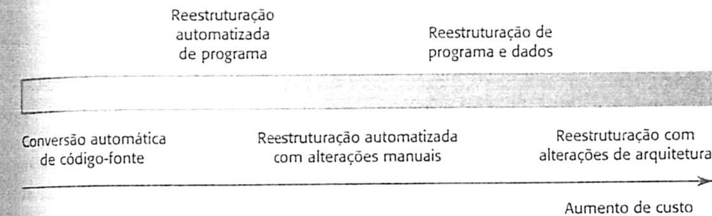


A reengenharia de programas pode não exigir necessariamente todas as etapas da Figura 9.11. Não é preciso traduzir o código-fonte se a linguagem de programação ainda for a mesma da aplicação. Se toda a reengenharia puder ser feita automaticamente, então pode ser desnecessário recuperar a documentação por meio de engenharia reversa. A reengenharia dos dados é necessária apenas se as estruturas de dados no programa mudarem durante a reengenharia do sistema.

Para fazer o sistema submetido à reengenharia interoperar com o novo software, pode ser necessário desenvolver serviços adaptadores, conforme discutido no Capítulo 18. Esses serviços escondem as interfaces originais do sistema de software e apresentam interfaces novas e mais bem estruturadas, que podem ser utilizadas por outros componentes. Esse processo de empacotamento do sistema legado é uma técnica importante para desenvolver serviços reusáveis de larga escala.

Obviamente, o custo da reengenharia depende do volume de trabalho executado. Há um espectro de possíveis abordagens para esse processo, como mostra a Figura 9.15. Os custos aumentam da esquerda para a direita, tal que a tradução do código-fonte é a opção mais barata e a reengenharia, como parte da migração da arquitetura, é a mais cara.

FIGURA 9.15 Abordagens para a reengenharia.



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

9.3.3 Refatoração

A refatoração é o processo de fazer melhorias em um programa para desacelerar a degradação por meio da mudança. Isso significa modificar um programa para melhorar a sua estrutura, reduzir a sua complexidade ou facilitar a sua compreensão. A refatoração às vezes é considerada um método limitado ao desenvolvimento orientado a objetos, mas os princípios podem ser aplicados, de fato, a qualquer abordagem de desenvolvimento. Ao refatorar um programa, não se deve acrescentar funcionalidade a ele, mas se concentrar em sua melhoria. Portanto, a refatoração pode ser encarada como uma 'manutenção preventiva' que reduz os problemas de alteração futura.

A refatoração é uma parte inerente dos métodos ágeis porque eles se baseiam na mudança. A qualidade do programa está sujeita à rápida degradação, então os desenvolvedores ágeis costumam refatorar seus programas para evitar essa deterioração. A ênfase no teste de regressão nos métodos ágeis diminui o risco de introduzir

novos erros por meio da refatoração. Quaisquer erros que sejam introduzidos devem ser detectáveis, já que os testes antes bem-sucedidos podem fracassar. No entanto, a refatoração não depende de outras 'atividades ágeis'.

Embora a reengenharia e a refatoração se destinem a tornar o software mais fácil de entender e de modificar, elas não são a mesma coisa. A reengenharia ocorre após um sistema ter sido mantido por algum tempo, com custos crescentes de manutenção. São utilizadas ferramentas automatizadas para processar e fazer a reengenharia de um sistema legado, a fim de criar um novo sistema mais fácil de manter. A refatoração é um processo contínuo de melhoria durante todo o processo de desenvolvimento e de evolução, buscando evitar a degradação da estrutura e do código — que aumenta o custo e a dificuldade de manter um sistema.

Fowler *et al.* (1999) sugere que existem situações estereotípicas (chamadas por ele de 'maus cheiros'), em que o código de um programa pode ser melhorado. Exemplos de maus cheiros que podem ser melhorados por meio da refatoração incluem:

1. *Código duplicado.* O mesmo código, ou um muito similar, pode ser incluído em diferentes lugares em um programa. Isso pode ser removido e implementado como um único método ou função invocado conforme a necessidade.
2. *Métodos longos.* Se um método for longo demais, ele deve ser reprojetado como uma série de métodos menores.
3. *Comandos* switch *(case).* Frequentemente, esses comandos envolvem duplicação, em que o *switch* depende do tipo de um valor. Os comandos *switch* podem estar dispersos em um programa. Nas linguagens orientadas a objetos, pode-se usar polimorfismo para obter a mesma coisa.
4. *Aglomeração de dados.* Ocorrem quando o mesmo grupo de itens de dados (campos nas classes, parâmetros nos métodos) ocorrem novamente em vários pontos de um programa. Eles podem ser substituídos por um objeto que encapsule todos os dados.
5. *Generalidade especulativa.* Ocorre quando os desenvolvedores incluem generalidade em um programa, caso venha a ser necessária no futuro. Muitas vezes isso pode ser simplesmente removido.

Fowler, não apenas em seu livro como também em seu site, sugere algumas transformações de refatoração primitiva, que podem ser utilizadas isoladamente ou em conjunto para lidar com os maus cheiros. Exemplos dessas transformações incluem 'Extrair método', em que se remove a duplicação e se cria um novo método; 'Consolidar expressão condicional', em que se substitui uma sequência de testes por um único teste; e 'Subir método na hierarquia', em que métodos similares nas subclasses são substituídos por um único método em uma superclasse. Os IDEs, como o Eclipse, geralmente incluem em seus editores o apoio para refatoração. Isso torna mais fácil encontrar partes dependentes de um programa que precisam ser modificadas para implementar a refatoração.

A refatoração, executada durante o desenvolvimento do programa, é uma maneira eficaz de reduzir os custos de manutenção de um programa no longo prazo. No entanto, quando se assume a manutenção de um programa cuja estrutura tenha se degradado bastante, pode ser praticamente impossível refatorar o código sozinho. Pode ser necessário pensar sobre a refatoração do projeto, que tende a ser um problema mais caro e difícil. A refatoração do projeto envolve identificar padrões de projeto relevantes (discutidos no Capítulo 7) e substituir o código existente por um que implemente esses padrões de projeto (KERIEVSKY, 2004).

PONTOS-CHAVE

- O desenvolvimento e a evolução de software podem ser considerados um processo integrado, interativo, que pode ser representado usando um modelo em espiral.
- Nos sistemas personalizados, o custo de manutenção de software normalmente ultrapassa o custo de desenvolvimento de software.
- O processo de evolução de software é impulsionado por solicitações de mudança e inclui a análise do impacto das mudanças, planejamento de lançamento e implementação da mudança.
- Os sistemas legados são mais antigos, desenvolvidos usando tecnologias de software e hardware obsoletas, mas que continuam úteis para uma empresa.
- Muitas vezes é mais barato e menos arriscado manter um sistema legado do que desenvolver um sistema substituto usando tecnologia moderna.
- O valor de negócio de um sistema legado e a qualidade do software de aplicação e seu ambiente devem ser avaliados para determinar se um sistema deve ser substituído, transformado ou mantido.
- Existem três tipos de manutenção de software: reparo de defeitos, modificação do software para trabalhar em um novo ambiente e implementação de requisitos novos ou modificados.
- A reengenharia de software tem a ver com reestruturar e redocumentar o software para torná-lo mais fácil de entender e mudar.
- A refatoração, que é fazer pequenas mudanças no programa que preservem a funcionalidade, pode ser encarada como uma manutenção preventiva.

LEITURAS ADICIONAIS

Working effectively with legacy code. Aconselhamento prático valioso sobre os problemas e as dificuldades de lidar com sistemas legados. FEATHERS, M., John Wiley & Sons, 2004.

"The economics of software maintenance in the 21st century." Esse artigo é uma introdução geral à manutenção e uma discussão abrangente dos custos de manutenção. Jones discute os fatores que afetam os custos de manutenção e sugere que quase 75% da mão de obra de software está envolvida em atividades de manutenção. JONES, C. 2006. Disponível em: <www.compaid.com/caiinternet/ezine/capersjones-maintenance.pdf>. Acesso em: 4 mai. 2018.

"You can't be agile in maintenance?" Apesar do título, essa postagem de blog argumenta que as técnicas ágeis são apropriadas para a manutenção e discute quais técnicas, conforme sugerido pela Programação Extrema, podem ser eficazes. BIRD, J., 2011. Disponível em: <http://swreflections.blogspot.co.uk/2011/10/you-cant-be-agile-in-maintenance.html>. Acesso em: 4 mai. 2018.

"Software reengineering and testing considerations." Esse é um excelente *white-paper* sobre questões de manutenção de uma importante empresa de software indiana. KUMAR, Y.; DIPTI, 2012. Disponível em: <http://www.infosys.com/engineering-services/white-papers/Documents/software-re-engineering-processes.pdf>. Acesso em: 4 mai. 2018.

SITE[1]

Apresentações em PowerPoint para este capítulo disponíveis em: <http://software-engineering-book.com/slides/chap9/>.
Links para vídeos de apoio disponíveis em: <http://software-engineering-book.com/videos/implementation-and-evolution/>.

1 Todo o conteúdo disponibilizado na seção *Site* de todos os capítulos está em inglês.

EXERCÍCIOS

9.1 Explique por que um sistema de software utilizado em um ambiente do mundo real deve mudar ou se torna cada vez menos útil.

9.2 Na Figura 9.4, você pode ver que a análise do impacto é um subprocesso importante no processo de evolução do

software. Usando um diagrama, sugira quais atividades poderiam estar envolvidas na análise do impacto da mudança.

9.3 Explique por que os sistemas legados devem ser considerados sistemas sociotécnicos em vez de simplesmente sistemas de software que foram desenvolvidos com tecnologia antiga.

9.4 Sob quais circunstâncias uma organização poderia decidir pelo descarte de um sistema quando sua avaliação sugere que ele é de alta qualidade e de alto valor de negócio?

9.5 Quais são as opções estratégicas para evolução dos sistemas legados? Quando você substituiria normalmente todo ou parte de um sistema em vez de continuar a manutenção do software?

9.6 Explique por que os problemas com software de apoio poderiam significar que uma organização tem de substituir seus sistemas legados.

9.7 Como um gerente de projetos de software em uma empresa especializada no desenvolvimento de software para o setor de petróleo *offshore*, você recebeu a tarefa de descobrir os fatores que afetam a manutenibilidade dos sistemas desenvolvidos pela sua empresa. Sugira como você poderia configurar um programa para analisar o processo de manutenção e determinar as métricas corretas de manutenibilidade da empresa.

9.8 Descreva resumidamente os três tipos principais de manutenção de software. Por que às vezes é difícil distinguir entre eles?

9.9 Explique as diferenças entre reengenharia e refatoração de software.

9.10 Os engenheiros de software têm uma responsabilidade profissional de desenvolver código que seja fácil de manter, mesmo que o empregador não solicite isso explicitamente?

REFERÊNCIAS

BANKER, R. D.; DATAR, S. M.; KEMERER, C. F.; ZWEIG, D. "Software complexity and maintenance costs." *Comm. ACM*, v. 36, n. 11, 1993. p. 81-94. doi:10.1145/163359.163375.

COLEMAN, D.; ASH, D.; LOWTHER, B.; OMAN, P. "Using metrics to evaluate software system maintainability." *IEEE Computer*, v. 27, n. 8, 1994. p. 44-49. doi:10.1109/2.303623.

DAVIDSEN, M. G.; KROGSTIE, J. "A longitudinal study of development and maintenance." *Information and Software Technology*, v. 52, n. 7, 2010. p. 707-719. doi:10.1016/j.infsof.2010.03.003.

ERLIKH, L. "Leveraging legacy system dollars for e-business." *IT Professional*, v. 2, n. 3, maio-jun. 2000. p. 17-23. doi:10.1109/6294.846201.

FOWLER, M.; BECK, K.; BRANT, J.; OPDYKE, W.; ROBERTS, D. *Refactoring*: improving the design of existing code. Boston: Addison-Wesley, 1999.

HOPKINS, R.; JENKINS, K. *Eating the IT Elephant*: moving from greenfield development to brownfield. Boston: IBM Press, 2008.

JONES, T. C. "The economics of software maintenance in the 21st century." 2006. Disponível em: <www.compaid.com/caiinternet/ezine/capersjones-maintenance.pdf>. Acesso em: 4 mai. 2018.

KERIEVSKY, J. *Refactoring to Patterns*. Boston: Addison-Wesley, 2004.

KOZLOV, D.; KOSKINEN, J.; SAKKINEN, M.; MARKKULA, J. "Assessing maintainability change over multiple software releases." *Journal of Software Maintenance and Evolution*, v. 20, n. 1, 2008. p. 31-58. doi:10.1002/smr.361.

LIENTZ, B. P.; SWANSON, E. B. *Software maintenance management*. Reading, MA: Addison-Wesley, 1980.

MITCHELL, R. M. "COBOL on the mainframe: does it have a future?" *Computerworld US*, 2012. Disponível em: <http://features.techworld.com/applications/3344704/cobol-on-the-mainframe-does-it-have-a-future/>. Acesso em: 4 mai. 2018.

O'HANLON, C. "A conversation with Werner Vogels." *ACM Queue*, v. 4, n. 4, 2006. p. 14-22. doi:10.1145/1142055.1142065.

RAJLICH, V. T.; BENNETT, K. H. "A staged model for the software life cycle." *IEEE Computer*, v. 33, n. 7, 2000. p. 66-71. doi:10.1109/2.869374.

ULRICH, W. M. "The evolutionary growth of software reengineering and the decade ahead." *American Programmer*, v. 3, n. 10, 1990. p. 14-20.

WARREN, I. (ed.). *The renaissance of legacy systems*. London: Springer, 1998.