11. Considerações sobre o Sistema
Como o Tamanho do Programa Afeta a Construção

Aumentar a escala no desenvolvimento de software não é uma simples questão de pegar um projeto pequeno e tornar maior cada parte dele. Suponha que você tenha escrito o pacote de software Gigatron de 25.000 linhas, envolvendo 20 homens/mês, e descobriu 500 erros nos testes de campo. Suponha que o Gigatron 1.0 teve êxito, assim como o Gigatron 2.0, e você comece a trabalhar no Gigatron Deluxe, uma versão bastante melhorada do programa, que se espera ter 250.000 linhas de código.

Mesmo sendo 10 vezes maior do que o Gigatron original, o Gigatron Deluxe não exigirá 10 vezes o esforço para ser desenvolvido; ele exigirá 30 vezes mais. Além disso, 30 vezes o esforço total não significa 30 vezes o trabalho de construção. Isso provavelmente significa 25 vezes o trabalho de construção e 40 vezes o trabalho de arquitetura e testes de sistema. Você também não terá 10 vezes mais erros; terá 15 vezes mais - ou ainda mais.

Se você estava acostumado a trabalhar em projetos pequenos, seu primeiro projeto de médio a grande porte pode ficar desenfreadamente fora de controle, tornando-se uma fera indomável, em vez do projeto bem-sucedido que você havia previsto. Este capítulo mostra que tipo de fera você deve esperar e onde encontrar o chicote e a cadeira para domá-la. Em contraste, se você estiver acostumado a trabalhar em projetos grandes, poderia utilizar estratégias formais demais para um projeto pequeno. Este capítulo descreve como você pode economizar, para evitar que um projeto pequeno venha abaixo com o peso de sua própria carga.

11.1 Comunicação e tamanho

Se você for a única pessoa em um projeto, o único caminho de comunicação será entre você e o cliente, a não ser que leve em conta o caminho através de seu corpo caloso, aquele que liga o lado esquerdo ao direito de seu cérebro. À medida que aumenta o número de pessoas envolvidas em um projeto, o número de caminhos de comunicação também aumenta. Esse número não aumenta de forma aditiva com o aumento do número de pessoas. Ele aumenta de forma multiplicativa, proporcionalmente ao quadrado do número de pessoas, conforme mostra a Figura 1 a seguir:

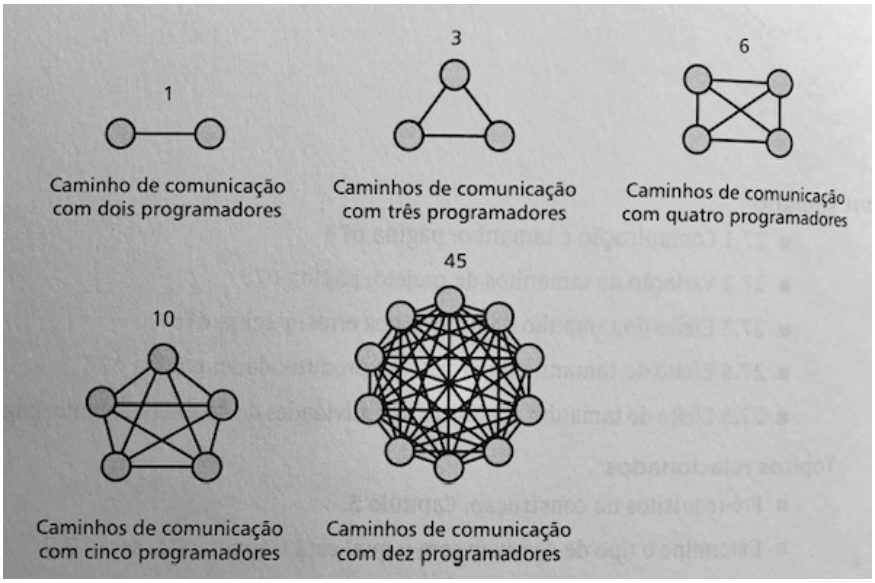

Figura 1: O número de caminhos de comunicação aumenta proporcionalmente ao quadrado do número de pessoas pertencentes à equipe.

Como você pode ver, um projeto com duas pessoas oferece apenas um caminho de comunicação. Um projeto com cinco pessoas oferece 10 caminhos. Um projeto com 10 pessoas oferece 45 caminhos, supondo que cada pessoa fale com todas as outras pessoas. Os 10% de projetos com 50 programadores ou mais oferece pelo menos 1.200 caminhos em potencial. Quanto mais caminhos de comunicação você tem, mais tempo gasta se comunicando e mais oportunidades são criadas para erros de comunicação. Os projetos de maior tamanho exigem técnicas organizacionais que simplifiquem a comunicação ou a limitem de uma maneira sensata.

A estratégia típica para simplificar a comunicação é formalizá-la em documentos. Em vez de ter 50 pessoas falando-se entre si em cada combinação concebível, 50 pessoas leem e redigem documentos. Alguns são documentos em texto; alguns são gráficos. Alguns são impressos no papel; outros são mantidos em forma eletrônica.

11.2 Variação de tamanhos de projeto

O tamanho do projeto no qual você está trabalhando é típico? A ampla variação de tamanhos de projeto não permite que você considere típico nenhum tamanho. Uma maneira de avaliar o tamanho do projeto é levar em consideração o tamanho da equipe que trabalha nele. Eis, a seguir, uma estimativa aproximada dos percentuais de todos os projetos realizados por equipes de vários tamanhos:

Tamanho da equipe	Percentual aproximado de projetos
1-3	25%
4-10	30%
11-25	20%
26-50	15%
50 +	10%

Fonte: adaptado de "A Survey of Software Engineering Practice: Tools, Methods, and Results" (Beck e Perkins 1983), Ag* Software Development Ecosystems (Highsmith 2002) e Balancing Agility and Discipline (Boehm e Tumer 2003).

Um aspecto dos dados do tamanho do projeto, que pode não ficar imediatamente aparente, é a diferença entre os percentuais de projetos de vários tamanhos e o número de programadores que trabalham em cada um desses projetos de diferentes tamanhos. Como os projetos maiores usam mais programadores em cada projeto do que os menores, eles empregam um grande percentual do total de programadores. Eis uma estimativa aproximada do percentual de todos os programadores que trabalham em projetos de vários tamanhos:

Tamanho da equipe	Percentual aproximado de programadores
1-3	5%
4-10	10%
11-25	15%
26-50	20%
50+	50%

11.3 Efeito do tamanho do projeto nos erros

Tanto a quantidade quanto o tipo dos erros são afetados pelo tamanho do projeto. Você poderia não acreditar que o tipo do erro fosse afetado, mas à medida que o tamanho do projeto aumenta, um maior percentual de erros normalmente acontece.

Nos projetos pequenos, os erros de construção constituem cerca de 75% de todos os erros encontrados. A metodologia tem menos influência sobre a qualidade do código; a maior influência sobre a qualidade do programa é frequentemente a aptidão da pessoa que o está escrevendo (Jones 1998).

Nos projetos maiores, os erros de construção podem diminuir gradualmente para cerca de 50% dos erros totais; os erros de requisitos e de arquitetura constituem a diferença. Presumivelmente, isso está relacionado ao fato de que mais desenvolvimento de requisitos e projeto arquitetônico são exigidos nos projetos grandes; assim, a oportunidade de erros provenientes dessas atividades é proporcionalmente maior. Em alguns projetos muito grandes, entretanto, a proporção dos erros de construção permanece alta; às vezes, mesmo com 500.000 linhas de código, até 75% dos erros podem ser atribuídos à construção (Grady 1987).

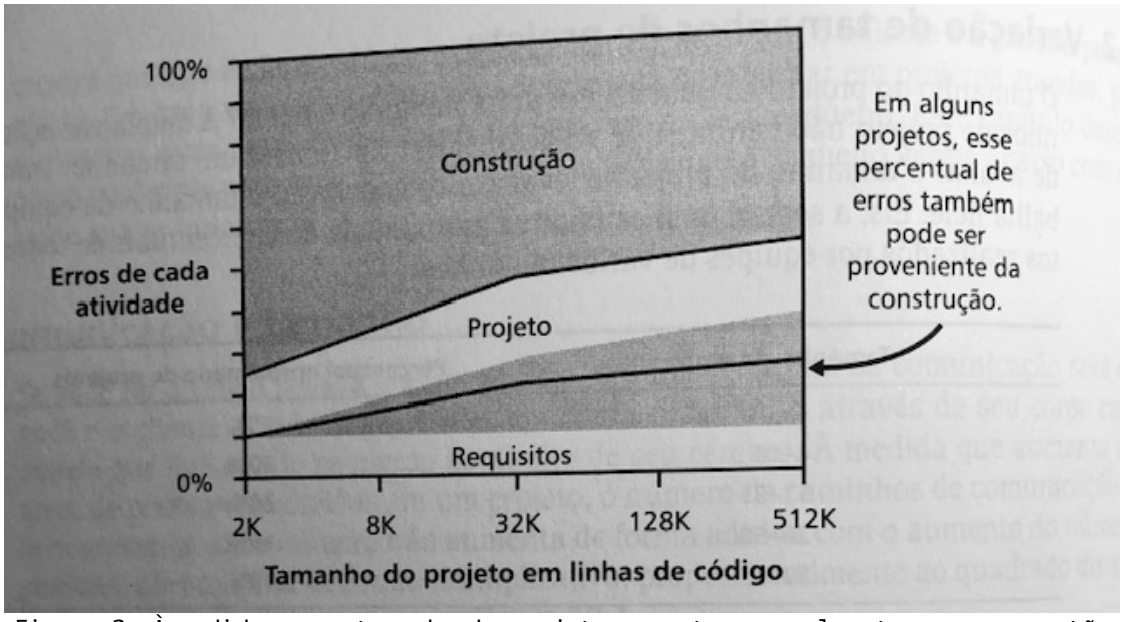

Figura 2: À medida que o tamanho do projeto aumenta, normalmente os erros estão mais localizados nos requisitos e do projeto do software. Às vezes, eles se encontram principalmente na construção.

Assim como os tipos de defeitos mudam conforme o tamanho, o mesmo acontece com o número deles. Você, naturalmente, esperaria que um projeto duas vezes maior que outro tivesse duas vezes mais erros. Mas a densidade de defeitos - o número de defeitos por 1000 linhas de código - aumenta. Um produto duas vezes maior provavelmente terá mais do que duas vezes mais erros. A Tabela a seguir mostra o intervalo de densidades de defeito que você pode esperar em projetos de vários tamanhos.

Os dados dessa tabela foram obtidos a partir de projetos específicos e os números podem ter pouca semelhança com aqueles dos projetos em que você trabalhou. Entretanto, como um instantâneo do setor, os dados são esclarecedores. Eles indicam que o número de erros aumenta substancialmente à medida que o tamanho do projeto aumenta, com os projetos muito grandes tendo até quatro vezes mais erros por 1.000 linhas de código do que os projetos pequenos. Será necessário trabalhar mais arduamente em um projeto grande do que em um projeto pequeno para obter o mesmo índice de erros.

Tamanho do projeto e a densidade típica de erro

Tamanho do Projeto (em linhas)	Densidade típica de erro
Menor do que 2K	0-25 erros por 1.000 linhas de código (KLDC)
2K-16K	0-40 erros por KLDC
16K-64K	0,5-50 erros por KLDC

64K-512K	2-70 erros por KLDC
512K ou mais	4-100 erros por KLDC

11.4 Efeito do tamanho do projeto na produtividade

A produtividade tem muito em comum com a qualidade do software quando se trata do tamanho do projeto. Em tamanhos pequenos (2.000 linhas de código ou menos), a maior influência sobre a produtividade é a capacidade do programador individual (Jones 1995). À medida que o tamanho do projeto aumenta, o tamanho e a organização da equipe se tornam influências maiores sobre a produtividade.

Que tamanho um projeto precisa ter, antes que o tamanho da equipe comece a afetar a produtividade? Alguns autores relataram que equipes menores concluíram seus projetos com uma produtividade 39% maior do que as equipes maiores. Qual era o tamanho das equipes? Duas pessoas para os projetos pequenos e três para o projeto grande.

A Tabela a seguir mostra o relacionamento geral entre tamanho do projeto e produtividade.

Tamanho do projeto e a produtividade

Tamanho do projeto (LOC)	Linhas de código por ano de trabalho em equipe
1K	2.500-25.000
10K	2.000-25.000
100K	1.000-20.000
1.000K	700-10.000
10.000K	300-5.000

A produtividade é principalmente determinada pelo tipo de software em que você está trabalhando, pela qualidade do pessoal, pela linguagem de programação, pela metodologia, pela complexidade do produto, pelo ambiente de programação, pelo suporte de ferramenta, por como as “linhas de código” são contadas, por como o trabalho de apoio de quem não é programador é decomposto no valor das “linhas de código por homens/ano” e de muitos outros fatores; portanto, os valores específicos da Tabela variam consideravelmente.

Perceba, entretanto, que a tendência geral mostrada pelos números é significativa. A produtividade nos projetos pequenos pode ser de duas a três vezes mais alta do que nos projetos grandes; a produtividade pode variar por um fator de 5 a 10, dos projetos menores para os maiores.

11.5 Efeito do tamanho nas atividades de desenvolvimento

Se você estiver trabalhando em um projeto de uma só pessoa, a maior influência sobre o sucesso ou a falha dele é você mesmo. Se estiver trabalhando em um projeto que envolva 25 pessoas, é concebível que você ainda seja a maior influência, mas é mais provável que nenhuma pessoa exiba medalha por essa distinção,- sua empresa terá uma influência mais forte sobre o sucesso ou a falha do projeto.

Proporções e tamanho da atividade

À medida que o tamanho do projeto e a necessidade de comunicações formais aumentam, os tipos de atividades necessárias mudam substancialmente. A Figura 3 a seguir mostra as proporções das atividades de desenvolvimento em relação a projetos de diferentes tamanhos.

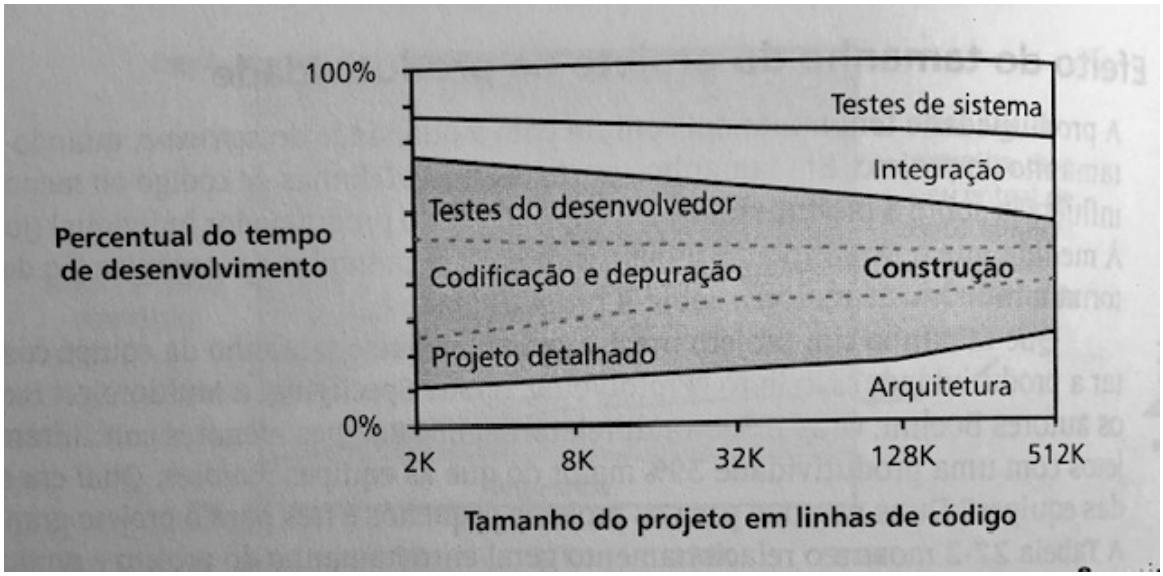

Figura 3: As atividades de construção dominam os projetos pequenos. Os projetos maiores exigem mais arquitetura, trabalho de integração e testes de sistema para ter sucesso. A atividade de requisitos não aparece nesse diagrama porque não é uma função direta do tamanho do programa, como acontece com as outras atividades.

Em um projeto pequeno, a construção é, de longe, a atividade mais proeminente, ocupando até 65% do tempo total do desenvolvimento. Em um projeto de tamanho médio, a construção ainda é a atividade dominante, mas sua parcela no trabalho total cai para cerca de 50%. Em projetos muito grandes, a arquitetura, a integração e os testes de sistema ocupam mais tempo, e a construção se torna menos dominante. Em resumo, à medida que o tamanho do projeto aumenta, a construção se torna uma parte menor do esforço total. No gráfico, parece que você poderia estendê-lo para a direita e fazer a construção desaparecer completamente; assim, para proteger meu trabalho, eu o limitei a 512K.

A construção se torna menos predominante porque, à medida que o tamanho do projeto aumenta, as atividades de construção - projeto detalhado, codificação, depuração e teste de unidade - aumentam proporcionalmente, mas muitas outras atividades aumentam com maior rapidez. Observe com atenção a Figura 4 abaixo.

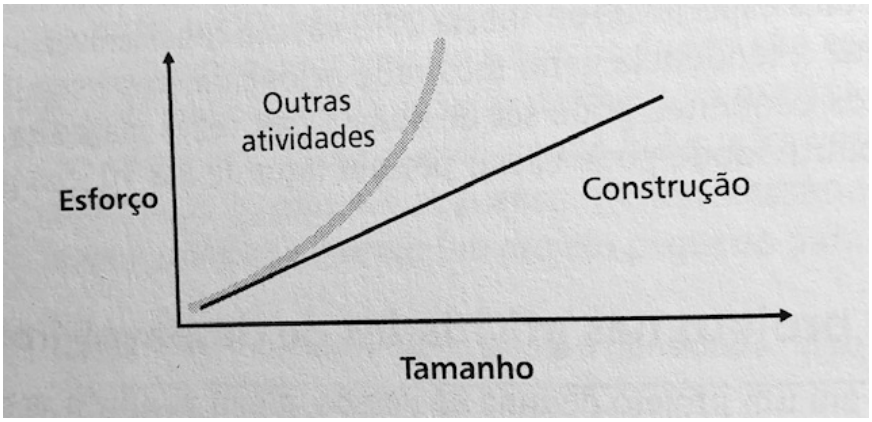

Figura 4: O volume do trabalho de construção de software é uma função quase linear do tamanho do projeto. Outros tipos de trabalho aumentam de forma não-linear, à medida que o tamanho do projeto aumenta.

Projetos de tamanho semelhante executarão atividades semelhantes, mas à medida que os tamanhos variam, os tipos de atividades também irão variar. Conforme foi descrito na introdução deste capítulo, quando o Gigatron Deluxe atingir 10 vezes o

tamanho do Gigatron original, ele precisará de 25 vezes mais trabalho de construção, de 25 a 50 vezes mais trabalho de planejamento, de 30 vezes mais trabalho de integração, e de 40 vezes mais arquitetura e testes de sistema.

As proporções das atividades variam porque diferentes atividades se tornam críticas em diferentes tamanhos de projeto. Barty Boehm e Richard Turner descobriram que aplicar cerca de 5% dos custos totais do projeto na arquitetura produzia o menor custo para projetos na faixa de 10.000 linhas de código. Mas, para projetos na faixa de 100.000 linhas de código, aplicar entre 15 e 20% do esforço na arquitetura produzia os melhores resultados (Boehm e Turner 2004).

Eis uma lista de atividades que crescem a uma taxa acima do linear, à medida que o tamanho do projeto aumenta:

- Comunicação
- Planejamento
- Gerenciamento
- Desenvolvimento de requisitos
- Projeto funcional do sistema
- Projeto e especificação da interface
- Arquitetura
- Integração
- Remoção de defeitos
- Testes de sistema
- Produção de documentos

Independentemente do tamanho de um projeto, algumas técnicas são sempre valiosas: práticas de codificação disciplinadas, inspeções de projeto e de código realizadas por outros desenvolvedores, bom suporte de ferramenta, e uso de linguagens de alto nível. Essas técnicas são valiosas em projetos pequenos e imprescindíveis em projetos grandes.

Programas, produtos, sistemas e produtos de sistema

As linhas de código e o tamanho da equipe não são as únicas influências no tamanho de um projeto. Uma influência mais sutil é a qualidade e a complexidade do software final. O Gigatron original, o Gigatron Jr., pode ter demorado apenas um mês para ser escrito e depurado. Ele era um único programa, escrito, testado e documentado por uma só pessoa. Se o Gigatron Jr. de 2.500 linhas demorou um mês, por que o Gigatron completo de 25.000 linhas demora 20 meses?

O tipo mais simples de software é um único "programa", usado sozinho pela pessoa que o desenvolveu ou, informalmente, por algumas outras.

Um tipo de programa mais sofisticado é um "produto" de software, um programa destinado ao uso por pessoas que não o desenvolvedor original. Um produto de software é usado em ambientes diferentes daquele em que foi criado. Antes de ser lançado, ele é extensivamente testado, documentado e pode ser mantido por outras pessoas. Um produto de software custa cerca de três vezes mais para ser desenvolvido do que um programa de software.

Para desenvolver um grupo de programas que trabalham em conjunto, é exigido um outro nível de sofisticação. Tal grupo é chamado de "sistema" de software. O desenvolvimento de um sistema é mais complicado do que o desenvolvimento de um programa simples, devido à complexidade de desenvolver as interfaces entre as partes e o cuidado necessário para integrá-las. No todo, um sistema também custa cerca de três vezes mais do que um programa simples.

Quando um "produto de sistema" é desenvolvido, ele tem o aperfeiçoamento de um produto e as várias partes de um sistema. Os produtos de sistema custam em torno de

nove vezes mais do que os programas simples (Brooks 1995, Shull et al. 2002).

Uma falha na apreciação das diferenças no aperfeiçoamento e na complexidade entre programas, produtos, sistemas e produtos de sistema é uma causa comum de erros de estimativa. Os programadores que usam sua experiência na construção de um programa, para estimar o cronograma de construção de um produto de sistema, podem subestima-la por um fator de quase 10. Ao examinar o exemplo a seguir, consulte o gráfico da Figura 3. Se você usasse sua experiência na escrita de 2.000 linhas de código para estimar o tempo que levaria para desenvolver um programa de 2.000 linhas, sua estimativa seria de apenas 65% do tempo total que realmente precisaria para executar todas as atividades relacionadas ao desenvolvimento de um programa. Escrever 2.000 linhas de código não demora tanto quanto criar um programa inteiro contendo 2.000 linhas de código. Se você não considerar o tempo que leva para realizar atividades que não estão relacionadas à construção, o tempo de desenvolvimento será 50% maior do que sua estimativa.

Quando você faz uma ampliação, a construção se torna uma parte menor do trabalho total em um projeto. Se você basear suas estimativas unicamente na experiência com a construção, o erro da estimativa aumentará. Se você usasse sua própria experiência na construção de um programa de 2.000 linhas para avaliar o tempo que levaria para desenvolver um programa de 32.000 linhas, sua estimativa seria de apenas 50% do tempo total exigido; o desenvolvimento levaria 100% de tempo a mais do que você estimaria.

O erro de estimativa aqui seria completamente atribuído ao seu não-entendimento do efeito do tamanho no desenvolvimento de programas maiores. Se, além disso, você não considerasse o grau de aperfeiçoamento extra exigido para um produto, em vez de um simples programa, o erro aumentaria facilmente por um fator de três ou mais.

Metodologia e tamanho

As metodologias são usadas em projetos de todos os tamanhos. Em projetos pequenos, as metodologias tendem a ser irregulares e instintivas. Em projetos grandes, elas tendem a ser rigorosas e cuidadosamente planejadas.

Algumas metodologias podem ser tão vagas que os programadores nem mesmo sabem que as estão usando. Alguns programadores argumentam que as metodologias são rígidas demais e impõem resistência ao seu uso. Embora possa ser verdade que um programador não tenha escolhido uma metodologia conscientemente, qualquer estratégia de programação constitui uma metodologia, independentemente do quanto ela seja inconsciente ou primitiva. O simples fato de levantar pela manhã e ir ao trabalho é uma metodologia rudimentar, embora não seja muito criativa. O programador que insiste em evitar metodologias está, na realidade, apenas evitando escolher uma delas explicitamente - mas ninguém pode evitar completamente o uso de metodologias.

As estratégias formais nem sempre são divertidas e, se forem aplicadas de maneira errada, sua sobrecarga devora suas outras economias. No entanto, a maior complexidade dos projetos maiores exige uma atenção mais consciente à metodologia. Construir um arranha-céu exige uma estratégia diferente da construção de um canil. Os diferentes tamanhos de projetos de software funcionam da mesma maneira. Em projetos grandes, escolhas inconscientes são inadequadas para a tarefa. Os planejadores de projetos bem-sucedidos escolhem explicitamente suas estratégias para projetos grandes.

Nos círculos sociais, quanto mais formal é o evento, mais desconfortáveis são os trajes (salto alto, gravata, etc.). No desenvolvimento de software, quanto mais formal é o projeto, mais documentos você tem de gerar para certificar-se de ter feito seu dever de casa. Capers Jones salienta que um projeto de 1.000 linhas de código terá em média cerca de 7% de seu esforço em trabalho administrativo, enquanto que um projeto de 100.000 linhas de código terá em média cerca de 26% de seu esforço em trabalho administrativo (Jones 1998).

Esse trabalho administrativo não é criado pela simples satisfação de escrever documentos. Ele é criado como um resultado direto do fenômeno ilustrado na Figura 1: quanto mais cérebros de pessoas você tiver de coordenar, mais documentação formal precisará para coordená-los.

Você não cria toda essa documentação para seu próprio bem. O objetivo de escrever um plano de controle de configuração, por exemplo, não é exercitar seus músculos de escritor. O objetivo de escrever o plano é forçá-lo a pensar criteriosamente sobre o controle de configuração e explicar seu plano para todo mundo. A documentação é um efeito colateral palpável do trabalho real que você realiza quando planeja e constrói um sistema de software. Se você sentir que está apenas agindo de forma mecânica e escrevendo documentos genéricos, algo está errado.

No que diz respeito às metodologias, "mais" não significa melhor. Na revisão que fizeram de suas metodologias ágeis versus orientadas por planos, Bany Boebm e Richard Turner advertem que normalmente é melhor se você iniciar com métodos pequenos e ampliá-los para um projeto grande, do que começar com um método que inclua tudo e reduzi-lo para um projeto pequeno (Boehm e Turner 2004). Algumas autoridades em software falam sobre metodologias "leves" e "pesadas", mas, na prática, o segredo é considerar o tamanho e o tipo específicos de seu projeto e então encontrar a metodologia que tenha "o peso certo".

Pontos-chave

- À medida que o tamanho do projeto aumenta, a comunicação precisa ser mantida. O objetivo da maioria das metodologias é reduzir os problemas de comunicação,- uma metodologia deve fazer o máximo para facilitar a comunicação.
- Se todos os outros elementos forem iguais, a produtividade será menor em um projeto grande do que em um pequeno.
- Se todos os outros elementos forem iguais, um projeto grande terá mais erros por 1.000 linhas de código do que um pequeno.
- As atividades que são dadas como certas em projetos pequenos devem ser cuidadosamente planejadas nos projetos maiores. A construção se torna menos predominante à medida que o tamanho do projeto aumenta.
- A ampliação de uma metodologia leve tende a funcionar melhor do que a redução de uma metodologia pesada. A estratégia mais eficaz de todas é usar a metodologia "de peso certo".

11. Considerações sobre o Sistema Gerenciando a Construção

Gerenciar o desenvolvimento de software tem sido um enorme desafio há várias décadas. O tópico geral do gerenciamento de projetos de software segue além dos objetivos da disciplina, mas este tópico discute alguns aspectos específicos do gerenciamento, que se aplicam diretamente à construção. Se você é desenvolvedor, este tópico poderá auxiliá-lo a entender os problemas aos quais os gerentes precisam estar bem atentos. Se você é gerente, o ajudará a compreender como os desenvolvedores veem o gerenciamento; você também tomará conhecimento dos procedimentos para gerenciar a construção eficientemente.

11.6 Estimulando a boa codificação

Como o código é o principal produto da construção, uma pergunta importante no gerenciamento da construção é: "Como você estimula as boas práticas de codificação?".

Em geral, não é uma boa ideia a gerência impor um conjunto rigoroso de padrões técnicos. Os programadores tendem a considerar os gerentes em um nível mais baixo de evolução técnica, em algum lugar entre os organismos unicelulares e os mamutes que desapareceram durante a Era Glacial; portanto, se vão existir padrões de programação, os programadores precisam aceitá-los.

Se, em um projeto, alguém for destacado para definir padrões, proponha que eles sejam definidos por um arquiteto e não pelo gerente. Os projetos de software funcionam tanto em uma "hierarquia da experiência" como em uma "hierarquia da autoridade". Se o arquiteto é visto como líder de pensamento do projeto, a equipe geralmente seguirá os padrões definidos por ele.

Se você escolher essa estratégia, certifique-se de que o arquiteto seja realmente respeitado. Às vezes, o arquiteto de um projeto é apenas uma pessoa mais velha, que está na empresa há muito tempo e não está em contato com os problemas da codificação de produção. Os programadores se ressentirão de que o "arquiteto" que está definindo os padrões não está a par do trabalho que estão fazendo.

Considerações sobre o estabelecimento de padrões

Os padrões são mais úteis em algumas empresas do que em outras. Alguns desenvolvedores acolhem os padrões com prazer, pois entendem que estes reduzem a disparidade arbitrária no projeto. Caso seu grupo resista à adoção de padrões rigorosos, considere as seguintes alternativas: diretrizes flexíveis, uma coleção de sugestões no lugar das diretrizes, ou um conjunto de exemplos que incorporem as melhores práticas.

Técnicas para estimular a boa codificação

Esta seção descreve várias técnicas para alcançar boas práticas de codificação que sejam menos opressivas do que o estabelecimento de padrões de codificações rígidos:

Designe duas pessoas para cada parte do projeto Se duas pessoas forem designadas para trabalhar em cada linha de código, você garantirá que pelo menos essas duas acreditam que esse código funciona e é legível. Os mecanismos para juntar duas pessoas podem variar da programação em pares até a formação de pares *mentor-trainee*, passando por revisões informais entre colegas.

Reveja cada linha de código Uma revisão de código normalmente envolve o programador e pelo menos dois revisores. Isso significa que pelo menos três pessoas leem cada linha de código. Outro nome da revisão em pares é "pressão de pares". Além de proporcionar uma rede de segurança, no caso de o programador original deixar o projeto, as revisões melhoram a qualidade do código, pois o programador sabe que este será lido por outras pessoas. Mesmo que sua empresa não tenha criado padrões de codificação explícitos, as revisões oferecem uma maneira sutil de promover um padrão de codificação em grupo - as decisões são tomadas pelo grupo durante as revisões e, com o passar do tempo, o grupo infere seus próprios padrões.

Exija assinaturas no código Em outros setores, os desenhos técnicos são aprovados e assinados pelo engenheiro responsável. A assinatura significa que tal engenheiro está atestando que os desenhos são tecnicamente adequados e estão livres de erros. Algumas empresas tratam o código da mesma maneira. Antes que ele seja considerado concluído, o pessoal técnico graduado deve assinar a respectiva listagem.

Mostre bons exemplos de código para a revisão Uma parte significativa do bom gerenciamento é comunicar seus objetivos claramente. Uma maneira de comunicar seus objetivos é fazer circular um código bem elaborado entre seus programadores ou afixá-lo para exibição pública. Ao fazer isso, você fornece um claro exemplo da qualidade que está buscando. Analogamente, um manual de padrões de codificação pode consistir principalmente em um conjunto das "melhores listagens de código". Identificar certas listagens como as "melhores" fornece um exemplo para os outros seguirem. Esse manual é mais fácil de atualizar do que um manual de padrões redigido em linguagem natural, e apresenta sem dificuldade as sutilezas no estilo de codificação que são difíceis de capturar ponto a ponto nas descrições em linguagem comum.

Acentue o fato de que as listagens de código são bens públicos Muitas vezes, os programadores interpretam que o código que escrevem "pertence a eles", como se fosse uma propriedade privada. Muito embora seja de fato resultado do trabalho deles, o código faz parte do projeto e deve estar livremente disponível para qualquer pessoa envolvida nesse projeto. Ele deve ser visto por outras pessoas durante as revisões e a manutenção, mesmo que não o seja em nenhum outro momento.

Recompense um código bem elaborado Use o método de premiação de sua empresa para reforçar as boas práticas de codificação. Lembre-se dos seguintes aspectos, quando desenvolver seu sistema de reforço:

- A recompensa deve ser algo realmente significativo para o programador. (Muitos programadores consideram desagradáveis recompensas do tipo "Muito bem!", especialmente quando elas partem de gerentes que não são técnicos.)
- Para ser digno de uma recompensa, o código deve ser excepcionalmente bom. Se for oferecido um prêmio para um programador que todos sabem que trabalha mal, você parecerá Homer Simpson tentando fazer um reator nuclear funcionar. Não importa que o programador tenha uma atitude cooperativa ou sempre chegue pontualmente ao trabalho. Você perderá credibilidade se sua recompensa não corresponder aos méritos técnicos da situação. Se você não se sentir habilitado tecnicamente o bastante para julgar se o código é bom ou ruim, não o julgue! Não ofereça recompensa, ou então deixe para sua equipe escolher quem irá recebê-la.

Um único padrão fácil Se você estiver gerenciando um projeto de programação e for experiente no assunto, uma técnica fácil e eficaz para obter um bom trabalho é dizer: "Eu devo ser capaz de ler e entender todo código escrito para o projeto". O fato de o gerente não ser o melhor técnico do mundo pode ser uma vantagem, no sentido de que isso pode desestimular a existência de código "engenhoso" ou cheio de truques.

11.7 Controle de configuração

Um projeto de software é dinâmico. O código muda, o projeto do software muda e os requisitos mudam. Além disso, alterações nos requisitos levam a mais alterações no projeto, c estas levam a ainda outras alterações no código e nos casos de teste.

O que é controle de configuração?

Controle de configuração é a prática de identificar artefatos do projeto e tratar das alterações sistematicamente, de modo que um sistema possa manter sua integridade

com o passar do tempo. Outro nome para isso é “controle de alteração". Ele inclui técnicas para avaliar as alterações propostas, controlar alterações e manter cópias do sistema em vários momentos no tempo.

Se você não controlar as alterações nos requisitos, talvez acabe escrevendo código para partes do sistema que eventualmente serão eliminadas. Você poderá inclusive escrever código Incompatível com novas partes do sistema. Poderá não detectar muitas incompatibilidades até o momento da integração, o que se transformaria num momento de acusações, pois ninguém saberia realmente o que estaria acontecendo.

Se as alterações no código não são controladas, talvez aconteça de você mudar uma rotina que outra pessoa está alterando simultaneamente; combinar com êxito suas alterações com as dessa outra pessoa será problemático. Alterações de código descontroladas podem fazer o código parecer mais testado do que realmente está. A versão que foi testada provavelmente será a versão inalterada antiga; a versão modificada pode não ter sido testada. Sem um bom controle de alteração, você poderia fazer alterações em uma rotina, encontrar novos erros e não conseguir voltar para a antiga rotina que funcionava.

Os problemas, assim, continuariam indefinidamente. Se as alterações não forem tratadas sistematicamente, você acabará dando passos a esmo na névoa densa, em vez de seguir diretamente para um destino claro. Sem um bom controle de alteração, em vez de desenvolver código, você está perdendo seu tempo. O controle de configuração o ajuda a usar seu tempo produtivamente.

O controle de configuração não foi inventado pelos programadores, mas como os projetos de programação são tão voláteis, ele é extremamente útil. Quando aplicado aos projetos de software, o controle de configuração é normalmente denominado “controle de configuração de software” (CCS). O CCS enfoca os requisitos, o código-fonte. a documentação e os dados de teste de um programa.

O problema sistêmico do CCS é o excesso de controle. A maneira mais segura de evitar acidentes automobilísticos é impedir que todo mundo dirija; a maneira mais segura de evitar problemas de desenvolvimento de software é impedir desenvolvimentos de software. Embora essa seja uma maneira de controlar as alterações, ela é impraticável para o desenvolvimento de software. Você precisa planejar o CCS cuidadosamente para que ele seja um trunfo, e não um morcego em volta de seu pescoço.

Em um projeto pequeno, de uma só pessoa, você provavelmente pode ficar muito bem sem nenhum CCS, além do planejamento de backups periódicos informais. Contudo, o controle de configuração ainda é útil. Em um projeto grande, envolvendo 50 pessoas, você provavelmente precisará de um esquema de CCS completo, incluindo procedimentos bastante formais para backups, controle de alteração para requisitos e design, além de controle sobre documentos, código-fonte, conteúdo, casos de teste e outros artefatos do projeto. Se o seu projeto não é muito grande nem muito pequeno, você terá que estabelecer um grau de formalidade em algum ponto entre os dois extremos. As subseções a seguir descrevem algumas das opções na implementação do CCS.

Alterações de requisitos e do projeto do software

Durante o desenvolvimento, você fatalmente terá muitas ideias sobre como melhorar o sistema. Se você implementar cada alteração à medida que elas lhe ocorrerem, logo se encontrará andando em uma espécie de “esteira” de software - embora o sistema mude, ele não se aproximará da conclusão. Aqui estão algumas diretrizes para controle de alterações de projeto:

Siga um procedimento de controle de alteração sistemático Conforme mostrado, um procedimento de controle de alteração sistemático é uma bênção quando

você tem muitos pedidos de mudança. Estabelecendo um procedimento sistemático, você torna claro que as alterações serão consideradas no contexto do que é melhor para o projeto como um todo.

Trate dos pedidos de alteração em grupos É tentador implementar as alterações fáceis à medida que as ideias surgem. O problema de tratar as alterações dessa maneira é que boas modificações podem ser desperdiçadas. Se pensar em uma alteração simples, com 25% do projeto concluído, e estiver dentro do cronograma, você fará a alteração. Se você pensar em outra alteração simples, com 50% do projeto concluído, e já estiver atrasado, não a fará. Quando seu tempo começar a ficar escasso no final do projeto, não importará que a segunda alteração seja 10 vezes melhor do que a primeira - você não estará em posição favorável para fazer quaisquer alterações não-essenciais. Algumas das melhores alterações podem escapar das mãos simplesmente porque você pensou nelas tarde demais.

Uma solução para esse problema é anotar todas as ideias e sugestões, independentemente de quanto elas sejam fáceis de implementar, e então reservá-las para quando você tiver tempo para trabalhar nelas. Portanto, tomando-as como grupo, escolha aquelas que serão as mais benéficas.

Estime o custo de cada alteração Quando seu cliente, seu chefe ou você mesmo estiverem tentados a alterar o sistema, faça uma estimativa do tempo que levaria para fazer a alteração, incluindo a revisão do código necessária para ela e para testar novamente o sistema inteiro. Inclua em sua estimativa o tempo necessário para tratar do efeito de propagação da alteração nos requisitos, no projeto, no código, no teste e nas mudanças na documentação do usuário. Faça com que todas as partes interessadas saibam que o software é intricadamente entrelaçado e que a estimativa de tempo é necessária, mesmo que, à primeira vista, a alteração pareça pequena.

Independentemente do quanto você se sinta otimista quando a alteração for sugerida, não faça uma estimativa improvisada. Tais estimativas são frequentemente erradas por um fator de 2 ou mais.

Seja cauteloso com altos volumes de alteração Embora certo grau de alteração seja inevitável, um alto volume de pedidos de alteração é um sinal de alerta importante de que os requisitos, a arquitetura ou os projetos de alto nível não foram elaborados o suficiente para suportar a construção efetiva. Voltar a trabalhar nos requisitos ou na arquitetura pode parecer dispendioso, mas custaria bem mais reconstruir o software ou descartá-lo por causa de características ou funcionalidades que você não precisava.

Crie um Conselho de Controle de Alteração, ou algo equivalente, que favoreça seu projeto A tarefa de um Conselho de Controle de Alteração é separar o joio do trigo nos pedidos de alteração. Qualquer um que queira propor uma modificação deve enviar o pedi do para esse Conselho. O termo “pedido de alteração" refere-se a qualquer solicitação que venha a modificar o software-, uma ideia de um novo recurso, uma mudança era um recurso já existente, um "relato de erro" que pode ou não estar informando um erro real, etc. O Conselho então se reúne periodicamente para examinar as alterações propostas. Ele aprova, reprova ou adia cada alteração. Os Conselhos de Controle de Alteração são considerados a melhor prática para priorizar e controlar alterações de requisitos; entretanto, eles ainda são bastante raros nas estruturas empresariais (Jones 1998).

Esteja atento à burocracia, mas não deixe que o medo dela impeça um

controle de alteração efetivo A falta de um controle de alteração disciplinado é um dos maiores problemas de gerenciamento que o setor de software enfrenta atualmente. Um percentual significativo de projetos considerados atrasados na verdade estaria dentro do cronograma se levasse em conta o impacto das alterações não-controladas, mas aceitas. Um controle de alteração deficiente permite que as alterações se acumulem desenfreadamente, o que mina a visibilidade do status, a previsibilidade a longo prazo, o planejamento do projeto, o controle de riscos específicos e o controle do projeto em geral.

O controle de alteração tende a descambar para a burocracia: portanto, é importante procurar maneiras que simplifiquem o processo de controle de alteração. Se você não usa pedidos de alteração tradicionais, configure um endereço de e-mail chamado "Conselho de Alteração" e oriente as pessoas para que enviem seus respectivos pedidos de alteração para esse endereço. Ou, então, sugira que as pessoas apresentem propostas de alteração interativamente, em uma reunião do conselho de alteração. Uma estratégia particularmente poderosa é registrar os pedidos de alteração como defeitos, em seu software de controle de defeitos. Os puristas classificarão tais alterações como "defeitos de requisitos"; ou então, você poderia classificá-las como alterações, em vez de defeitos.

Você pode implementar o Conselho de Controle de Alteração, em si, formalmente ou pode definir um Grupo de Planejamento de Produto ou Conselho de Guerra que tenha sobre si as responsabilidades tradicionais de um Conselho de Controle de Alteração. Ou, ainda, você pode identificar uma pessoa para ser o Czar da Alteração. Mas, seja como for que você o denomine, faça isso!

Ocasionalmente, vejo projetos sofrendo de implementações de controle de alteração malfeitas. Entretanto, vejo 10 vezes mais projetos padecendo por nenhum controle de alteração significativo. O importante é a essência do controle de alteração; portanto, não deixe o medo da burocracia impedi-lo de perceber os muitos benefícios daí decorrentes.

Alterações no código do software

Outro problema de controle de configuração é controlar o código-fonte. Se você alterar o código e surgir um novo erro que parece não estar relacionado com a alteração feita, provavelmente desejará comparar a nova versão do código com a antiga, na busca pela origem do erro. Se isso nada informar, talvez você queira examinar uma versão ainda mais antiga. Esse tipo de excursão pela história será fácil se você tiver ferramentas de controle de versão que rastreiem várias versões do código-fonte.

Software de controle de versão Um bom software de controle de versão funciona tão facilmente que você mal percebe que o está usando. Ele é particularmente útil para projetos em equipe. Um estilo de controle de versão bloqueia os arquivos-fonte, para que somente uma pessoa por vez possa modificar um arquivo. Normalmente, quando precisa trabalhar no código-fonte de um determinado arquivo, você o retira do controle de versão. Se alguém já o tiver retirado, você será notificado de que não pode retirá-lo. Quando você pode retirar o arquivo, trabalha nele como faria sem controle de versão, até estar pronto para devolvê-lo. Outro estilo permite que várias pessoas trabalhem nos arquivos simultaneamente e trata do problema da união das alterações quando o código for devolvido ao controle. Em qualquer um dos casos, quando você devolve o arquivo, o controle de versão pergunta por que ele foi alterado, e você digita o motivo.

Para esse modesto investimento em esforço, você obtém diversas vantagens importantes:

- Você não tropeça em ninguém que esteja trabalhando em um arquivo, enquanto

o estiver utilizando (ou, pelo menos, você saberá disso, se tal fato ocorrer).
- Você pode atualizar facilmente suas cópias de todos os arquivos do projeto com as versões correntes executando apenas um único comando.
- Você pode retroceder para qualquer versão de qualquer arquivo que já tenha sido inserido no controle de versão.
- Você pode obter uma lista das alterações feitas em qualquer versão de qualquer arquivo.
- Você não precisa se preocupar com backups pessoais, pois a cópia do controle de versão é uma rede de segurança.

O controle de versão é indispensável nos projetos em equipe. Tudo funciona ainda melhor quando o controle de versão, o rastreamento de defeitos e o controle de alteração estão integrados. A divisão de aplicativos da Microsoft verificou que sua ferramenta de controle de versão patenteada é uma "importante vantagem competitiva".

Versões de ferramenta

Para alguns tipos de projetos, pode ser necessário reconstruir o ambiente exato usado para cada versão específica do software, incluindo compiladores, editores de ligação, bibliotecas de código, etc. Nesse caso, você deve colocar também todas essas ferramentas no controle de versão.

Configurações de máquina

Muitas empresas têm obtido bons resultados com a criação de configurações de máquina de desenvolvimento padronizadas. É criada uma imagem de disco de uma estação de trabalho padrão para desenvolvimento, incluindo todas as ferramentas de desenvolvedor comuns, aplicativos de escritório, etc. Essa imagem é carregada na máquina de cada desenvolvedor. Ter configurações padronizadas ajuda a evitar muitos problemas associados a cenários de configuração ligeiramente diferentes, versões diferentes de ferramentas usadas e coisas do gênero. Uma imagem de disco padronizada também simplifica muito a configuração de novas máquinas, em comparação com a necessidade de instalar cada software individualmente.

Plano de backup

Um plano de backup não é um conceito novo - é a ideia de fazer backup de seu trabalho periodicamente. Se você estivesse escrevendo um livro à mão, não deixaria as folhas empilhadas na varanda. Se fizesse isso, elas poderiam tomar chuva, ser levadas pelo vento ou o cachorro do seu vizinho poderia tomá-las emprestado para ler na hora de dormir. Por certo, você as guardaria em algum lugar seguro. O software é menos palpável; portanto, é mais fácil esquecer que você tem algo de enorme valor na máquina.

Muitas coisas podem acontecer com dados computadorizados: um disco pode falhar, você ou outra pessoa pode excluir acidentalmente arquivos importantes, um funcionário irritado pode sabotar sua máquina, ou você pode ficar sem sua máquina por causa de roubo, enchente ou incêndio. Tome as providências necessárias para proteger seu trabalho. Seu plano de backup deve incluir backups periódicos e transferências periódicas de backups para armazenamento fora do prédio, e deve abranger todos os materiais importantes de seu projeto - documentos, gráficos e anotações –, além do código-fonte.

Um aspecto frequentemente desprezado de um plano de *backup é o* teste de seu procedimento de *backup.* Tente fazer uma restauração em algum ponto, para certificar-se de que o *backup* contém tudo o que você precisa e que a recuperação

funciona.

Ao terminar um projeto, faça um repositório de arquivos para ele. Salve uma cópia de tudo: código-fonte, compiladores, ferramentas, requisitos, projeto, documentação - tudo que você precisa para recriar o produto. Guarde tudo em local seguro.

Lista de verificação: controle de configuração

Geral

- Seu plano de controle de configuração de *software* está projetado de forma a ajudar os programadores e minimizar a sobrecarga?
- Sua estratégia de CCS (Controle de Configuração *de Software)* evita um controle exagerado do projeto?
- Você agrupa os pedidos de alteração, seja por intermédio de meios informais (como uma lista de alterações pendentes), seja por intermédio de uma estratégia mais sistemática (como um Conselho de Controle de Alteração)?
- Você estima sistematicamente o custo, o cronograma e o impacto na qualidade de cada alteração proposta?
- Você encara as alterações importantes como um alerta de que o desenvolvimento dos requisitos ainda não está concluído?

Ferramentas

- Você usa *software* de controle de versão para facilitar o controle de configuração?
- Você usa *software* de controle de versão para reduzir os problemas de coordenação do trabalho em equipe?

Backup

- Você faz *backup* de todos os materiais do projeto periodicamente?
- Os *backups* do projeto são transferidos para armazenamento fora do prédio periodicamente?
- É feito o *backup* de todos os materiais, incluindo código-fonte, documentos, gráficos e anotações importantes?
- Você testou o procedimento de backup-recuperação?

11.8 Estimando um cronograma de construção

Gerenciar um projeto de software é um dos maiores desafios do século XXI, e estimar o tamanho de um projeto e o esforço exigido para concluí-lo é um dos aspectos mais difíceis do gerenciamento de projetos de software. Em média, um grande projeto de software atrasa um ano e ultrapassa o orçamento em 100% (Standish Group 1994, Jones 1997, Johnson 1999). Em nível individual, os comparativos dos cronogramas estimados e reais têm verificado que as estimativas dos desenvolvedores tendem a apresentar um fator otimista de 20 a 30% (van Genuchten 1991). Isso está fortemente relacionado com estimativas de tamanho e esforço insuficientes, assim como com fracos esforços de desenvolvimento. Esta seção destaca os problemas envolvidos na estimativa de projetos de software e indica onde procurar mais informações.

Estratégias de estimativa

Você pode estimar o tamanho de um projeto e o trabalho exigido para concluí-lo de várias maneiras:

- Use software de estimativa.
- Use uma estratégia algorítmica, como o Cocomo II.
- Peça para especialistas em estimativa externos estimarem o projeto.
- Faça uma reunião de ensaio para estimativas.

- Estime partes do projeto e, depois, agrupe-as.
- Peça às pessoas para estimarem suas próprias tarefas e, então, agrupe as estimativas.
- Consulte soluções experimentadas em projetos anteriores.
- Consulte estimativas anteriores e veja o quanto elas foram precisas. Utilize-as para ajustar as novas estimativas.

Eis, a seguir, uma boa estratégia para estimar um projeto:

Estabeleça objetivos Por que você precisa de uma estimativa? O que você está estimando? Você está estimando apenas atividades de construção ou todo o desenvolvimento? Você está estimando apenas o esforço relativo a seu projeto ou relativo a seu projeto mais férias, feriados, treinamento e outras atividades não relacionadas ao projeto em si? O quanto a estimativa precisa ser exata para atingir seus objetivos? Que grau de segurança precisa estar associado à estimativa? Uma estimativa otimista ou pessimista produziria resultados substancialmente diferentes?

Separe um tempo para elaborar a estimativa e planeje-a Estimativas feitas às pressas são imprecisas. Se você estiver estimando um projeto grande, trate a estimativa como um miniprojeto e reserve tempo para fazer um miniplanejamento da estimativa, para que possa fazê-la bem.

Explique detalhadamente os requisitos de software Assim como um arquiteto não pode estimar quanto custará uma casa “muito grande”, você não pode estimar com segurança um projeto de software “muito grande”. Não é razoável alguém esperar que você possa estimar a quantidade de trabalho para construir algo, quando esse "algo” ainda não foi definido. Defina os requisitos ou planeje uma fase exploratória preliminar, antes de fazer uma estimativa.

Estime em um nível baixo de detalhes Dependendo dos objetivos que você tiver identificado, baseie a estimativa em um exame detalhado das atividades do projeto. Em geral, quanto mais detalhado for seu exame, mais precisa será sua estimativa. A Lei dos Números Grandes diz que um erro de 10% em uma peça grande será de 10% para mais ou para menos. Em 50 peças pequenas, parte dos 10% dos erros nas peças serão para mais e parte serão para menos, e os erros tenderão a se anular.

Use várias técnicas de estimativa diferentes e compare os resultados A lista de estratégias de estimativa do início da seção identificou diversas técnicas. Nem todas elas produzirão os mesmos resultados; portanto, experimente várias delas. Estude os diferentes resultados das diferentes estratégias. As crianças aprendem desde cedo que, se pedirem uma terceira taça de sorvete a cada um dos pais, individualmente, terão uma chance maior de ouvir pelo menos um “sim", do que st pedirem para apenas um deles. Às vezes, os pais percebem e dão a mesma resposta; às vezes, eles não percebem, Veja quais diferentes respostas você consegue obter a partir das diferentes técnicas de estimativa.

Nenhuma estratégia específica é a melhor em todas as circunstâncias, e as diferenças entre elas podem ser esclarecedoras.

Faça novas estimativas periodicamente Em um projeto de software, os fatores mudam após a estimativa inicial; portanto, planeje atualizar suas estimativas periodicamente. A precisão de suas estimativas deve melhorar à medida que você se

aproximar da conclusão do projeto. De tempos era tempos, compare seus resultados reais com os resultados estimados e use essa avaliação para aprimorar as estimativas para o restante do projeto.

Estimando a quantidade de construção

A maior influência que a construção terá sobre o cronograma de um projeto depende, em parte, da proporção desse projeto que será dedicada a ela - a construção levando em conta o projeto detalhado do software, a codificação e a depuração, e o teste unitário. Observe novamente a Figura 3. Como se pode verificar, a proporção varia de acordo com o tamanho do projeto. Até que sua empresa tenha seus próprios dados históricos de um projeto, a proporção de tempo dedicada a cada atividade mostrada na figura é um bom começo para as estimativas de seus projetos.

A melhor resposta para a pergunta sobre o quanto um projeto exigirá, em termos de construção, é esta: a proporção irá variar de um projeta para outro e de uma empresa para outra. Mantenha registros da experiência de sua empresa com projetos e utilize-os para estimar o tempo que os futuros projetos exigirão.

Influências sobre o cronograma

A maior influência sobre o cronograma de uns projeto de software é o tamanho do programa a ser produzido. Mas muitos outros fatores também influenciam o cronograma de um desenvolvimento de software. Estudos feitos em programas comerciais quantificaram alguns fatores; eles são mostrados na tabela a seguir:

Fator	Influência útil em potencial	Influência prejudicial em potencial
Desenvolvimento no mesmo local versus em vários lugares	-14*%	22%
Tamanho do banco de Dados	-10%	28%
Correspondência da documentação com as necessidades do projeto	-19%	23%
Flexibilidade permitida na interpretação dos requisitos	-9%	10%
O quanto os riscos são tratados ativamente	-12%	14%
Experiência com a linguagem e com as ferramentas	-16%	20%
Continuidade do pessoal (rotatividade)	-19%	29%
Volatilidade da plataforma	-13%	30%
Maturidade do processo	-13%	15%
Complexidade do produto	-27%	74%
Habilidade do programador	-24%	34%
Confiabilidade exigida	-18%	26%
Habilidade do analista de requisitos	-29%	42%
Requisitos de reutilização	-5%	24%
Aplicativo moderno	-11%	12%
Restrição de armazenamento (armazenamento disponível a ser consumido)	0%	46%
Coesão da equipe	-10%	11%
Experiência da equipe na área de aplicações	-19%	22%
Experiência da equipe na plataforma de tecnologia	-15%	19%
Restrição de tempo (da aplicação em si)	0%	63%
Uso de ferramentas de software	-22%	17%

Eis alguns fatores cuja quantificação é menos fácil, os quais podem influenciar um cronograma de desenvolvimento de software.

- Experiência e habilidade do desenvolvedor de requisitos
- Experiência e habilidade do programador
- Motivação da equipe
- Qualidade do gerenciamento
- Volume de código reutilizado
- Rotatividade do pessoal
- Volatilidade dos requisitos
- Qualidade do relacionamento com o cliente
- Participação do usuário nos requisitos
- Experiência do cliente com o tipo de aplicativo
- Grau em que os programadores participam do desenvolvimento dos requisitos
- Ambiente de segurança classificado para computador, programas e dados
- Volume de documentação
- Objetivos do projeto (cronograma versus qualidade versus usabilidade versus muitos outros objetivos possíveis)

Cada um desses fatores pode ser significativo; portanto, considere-os juntamente com aqueles apresentados na Tabela 1 (os quais incluem alguns dos que foram mencionados aqui),

Estimativa versus controle

a questão im- a estimativa é uma parte importante do planejamento necessário para concluir um pro- cêquwprevw Jet0 de so/tware a tempo. Tão logo você tenha uma data de entrega e uma especificação ou quer contro- de produto determinadas, o principal problema é como controlar a utilização de recursos lar? humanos e técnicos para entregar o produto no prazo. Nesse sentido, a precisão da estimativa inicial e muito menos importante do que seu êxito subsequente no controle dos recursos para cumprir o cronograma.

O que fazer se você se atrasar

Em média, um projeto ultrapassa o cronograma planejado em cerca de 100%, conforme mencionado anteriormente neste capítulo. Quando você estiver atrasado, esticar o tempo normalmente será uma opção inviável. Mas se isso for possível, faça-o. Caso contrário, você pode tentar uma ou mais das seguintes soluções:

Tenha esperança de pôr-se em dia O otimismo esperançoso é uma resposta comum para um atraso no cronograma de um projeto. O raciocínio normalmente é o seguinte: “Os requisitos demoraram um pouco mais do que esperávamos, mas agora eles são sólido; portanto, fatalmente economizaremos tempo mais tarde. Supriremos o déficit durante a codificação e os testes". Dificilmente isso acontece. Um levantamento envolvendo mais de 300 projetos de software revelou que os atrasos e “furos” de cronograma geralmente aumentam no final de um projeto (van Genuchten 1991). Os projetos não compensam posteriormente o tempo perdido; eles ficam ainda mais atrasados.

Aumente a equipe De acordo com a lei de Fred Brooks, acrescentar outras pessoas para trabalhar em um projeto de software atrasado em geral provoca um atraso ainda maior (Brooks 1995). É como jogar gasolina na fogueira. A explicação de Brooks é convincente: novas pessoas precisam de tempo para se familiarizar com o projeto, antes que possam se tornar produtivas. A sua adaptação ocupa o tempo das pessoas que já

foram adaptadas. O simples fato de aumentar o número de pessoas aumenta também a complexidade e o volume da comunicação no projeto. Brooks destaca que o fato de uma mulher poder ter um bebê em nove meses não significa que nove mulheres poderão ter um bebê em um mês.

Sem dúvida, o alerta da lei de Brooks deve ser levado em consideração mais frequentemente do que acontece. É tentador colocar pessoas em um projeto e esperar que elas o terminem no prazo. Os gerentes precisam entender que desenvolver software não é como rebitar metal laminado: mais funcionários trabalhando não significa necessariamente que mais trabalho será feito.

Entretanto, a simples afirmação de que acrescentar programadores em um projeto atrasado fará com ele atrase ainda mais, mascara o fato de que - em determinadas circunstâncias - é possível acrescentar pessoas em um projeto atrasado e acelerá-lo. Conforme Brooks destaca na análise de sua lei, acrescentar pessoas em projetos de software, em que as tarefas não podem ser divididas e realizadas independentemente, não ajuda. Mas, quando as tarefas de um projeto podem ser fracionadas, você pode dividi-las ainda mais e distribuí-las entre diferentes pessoas, mesmo entre aquelas que passaram a atuar no final do projeto. Outros pesquisadores têm identificado, formalmente, circunstâncias sob as quais você pode acrescentar pessoas em um projeto atrasado, sem postergá-lo ainda mais (Abdel-Hamid 1989, McConnell 1999).

Reduza a abrangência do projeto A técnica poderosa da redução da abrangência do projeto é frequentemente desprezada. Se elimina uma característica, você elimina o projeto, a codificação, a depuração, o teste e a documentação dessa característica. Você elimina a interface dessa característica com outras.

Quando inicialmente você planejar o produto, divida suas características em “obrigatórias”, “interessantes” e “opcionais”. Se você se atrasar, estabeleça uma escala de prioridade para as características “opcionais” e “interessantes”, e abandone aquelas que forem as menos importantes.

Sem contar a total exclusão de uma característica, você pode fornecer uma versão mais econômica da mesma funcionalidade. Você poderia fornecer uma versão que estivesse no prazo, mas que não tivesse o desempenho otimizado. Poderia fornecer uma versão na qual a funcionalidade menos importante fosse implementada de maneira menos criteriosa. Você poderia optar por desistir de um requisito de velocidade, pois é muito mais fácil fornecer uma versão lenta. Poderia desistir de um requisito de espaço, pois é mais fácil fornecer uma versão que use muita memória.

Estime novamente o tempo de desenvolvimento para os requisitos menos importantes. Que funcionalidade você pode fornecer em duas horas, dois dias ou duas semanas? O que você ganha construindo a versão de duas semanas em vez da versão de dois dias, ou a versão de dois dias em vez da versão de duas horas?

11.9 Medição

Os projetos de software podem ser medidos de várias maneiras. Eis, a seguir, duas sólidas razões para você medir seu processo:

Para qualquer atributo de projeto, é sempre mais vantajoso medi-lo do que não medi-lo A medição pode não ser perfeitamente precisa, talvez seja difícil de pô-la em prática e é provável que ela precise ser aprimorada com o passar do tempo. No entanto, ela pode prestar uma grande ajuda em seu processo de desenvolvimento de software (Gilb 2004).

Sempre que os dados se destinarem a ser usados em uma experiência científica, eles precisam ser quantificados. Você consegue imaginar um cientista recomendando

a proibição de um novo alimento apenas porque um grupo de ratos brancos "pareceu ficar mais doente" do que outro grupo? Isso é absurdo. Você exigiria um motivo quantificado, como "Os ratos que se alimentaram com o novo produto adoeceram 3,7 mais dias por mês do que os que não o ingeriram". Para avaliar os métodos de desenvolvimento de software, você precisa medi-los. Afirmações como "Esse novo método parece mais produtivo" não são suficientemente satisfatórias.

Esteja informado dos efeitos colaterais da medição A prática de medir tem um efeito de motivação. As pessoas prestam atenção ao que é medido, supondo que isso seja usado para avaliá-las. Escolha cuidadosamente o que você vai medir. As pessoas tendem a se concentrar no trabalho que é medido e a ignorar o que não é.

Argumentar contra a medição é inferir que é melhor não saber o que está realmente acontecendo em seu projeto Ao medir um determinado aspecto de um projeto, você fica sabendo algo sobre ele que não sabia antes. Você pode constatar se esse aspecto fica maior, menor ou permanece o mesmo. A medição fornece uma janela para que se possa ver pelo menos esse aspecto de seu projeto. A janela pode ser pequena e oferecer pouca nitidez, até você aperfeiçoar suas medições, mas ainda assim é melhor do que não ter nenhuma. Opor-se a todas as medições porque algumas não são conclusivas é como opor-se às janelas porque algumas têm os vidros embaçados.

Você pode medir praticamente qualquer aspecto do processo de desenvolvimento de software. A Tabela a seguir lista alguns aspectos a serem medidos, considerados úteis por profissionais da área.

Tabela: Aspectos úteis de desenvolvimento de software a serem medidos

Tamanho	Qualidade global
Total de linhas de código escritas	Número total de defeitos
Total de linhas de comentário	Número de defeitos em cada classe ou rotina
Número total de classes ou rotinas	Média de defeitos por 1.000 linhas de código
Total de declarações de dados	Tempo médio entre falhas
Total de linhas em branco	Erros detectados pelo compilador

Rastreamento de defeitos	Manutenibilidade
Gravidade de cada defeito	Número de rotinas públicas em cada classe
Localização de cada defeito (classe ou rotina)	Número de parâmetros passados para cada rotina
Origem de cada defeito (requisitos, projeto, construção, teste)	Número de rotinas e/ou variáveis privadas em cada classe
Maneira pela qual cada defeito é corrigido	Número de variáveis locais usadas por cada rotina
Pessoa responsável por cada defeito	Número de rotinas chamadas por cada classe ou rotina
Número de linhas afetadas por cada correção de defeito	Número de pontos de decisão em cada rotina
Horas de trabalho gastas na correção de cada defeito	Complexidade do fluxo de controle em cada rotina
Tempo médio exigido para encontrar um defeito	Linhas de código em cada classe ou rotina
Tempo médio exigido para corrigir um defeito	Linhas de comentários em cada classe ou rotina
Número de tentativas para corrigir	Número de declarações de dados em cada classe

cada defeito	ou rotina
Número de novos erros resultantes da correção do defeito	Número de linhas em branco em cada classe ou rotina
	Número de instruções goto em cada classe ou rotina
	Número de instruções de entrada ou salda em cada classe ou rotina

Produtividade
Horas de trabalho gastas no projeto Horas de trabalho gastas em cada classe ou rotina
Número de vezes que cada classe ou rotina é alterada
Valor gasto no projeto Valor gasto por linha de código Valor gasto por defeito

Você pode reunir a maior parte dessas medições com as ferramentas de software correntemente disponíveis. Neste momento, a maioria das medições não serve para fazer distinções claras entre programas, classes e rotinas (Shepperd e Ince 1989). Elas são úteis principalmente para identificar rotinas que estão "fora da média"; medições anormais em uma rotina são um sinal de alerta de que você deve examiná-la novamente, para identificar uma qualidade extraordinariamente baixa.

Não comece reunindo dados com todas as medições possíveis - você ficará mergulhado em dados tão complexos que não conseguirá descobrir o que eles significam. Comece com um conjunto de medições simples, como o número de defeitos, o número de meses de trabalho, o valor total gasto e o total de linhas de código. Primeiramente, padronize as medições em seus projetos, para depois, então, aprimorá-las e ampliá-las, quando estiver mais seguro do que deseja medir (Pietrasanta 1990).

Certifique-se de ter um bom motivo para estar reunindo dados. Estabeleça objetivos, determine as perguntas que você precisa fazer para atingir esses objetivos e depois meça para responder às perguntas (Basili e Weiss 1984). Certifique-se de perguntar apenas sobre o volume de informação que seja possível obter e lembre de que a reunião de dados sempre estará em posição secundária em relação aos prazos finais (Basili et al. 2002).

11.10 Tratando os programadores como pessoas

O caráter abstrato da atividade de programação exige uma compensação natural no ambiente do escritório e contatos ricos entre os colegas. Empresas altamente técnicas oferecem instalações que se assemelham a parques, estruturas organizacionais orgânicas, escritórios confortáveis e outras características ambientais "refinadas" para contrabalançar a característica intelectual intensa e, às vezes, árida do trabalho em si. As empresas técnicas de maior sucesso combinam elementos de alta tecnologia e refinamento. Esta seção descreve as maneiras pelas quais os programadores são mais do que reflexos orgânicos de seus alter egos de silício.

Como os programadores usam seu tempo?

Os programadores usam seu tempo programando, mas eles também passam o tempo em reuniões, em treinamento, lendo e-mails e apenas pensando. Um estudo realizado na

Bell Laboratories, revelou que os programadores gastam seu tempo de várias maneiras: falando, ouvindo, falando com o gerente, telefonando, lendo, gravando dados, andando.

Esses dados são baseados em um estudo de tempo e movimento realizado com 70 programadores. Os dados são antigos e as proporções do tempo gasto nas diferentes atividades varia entre os programadores, mas os resultados dão o que pensar. Cerca de 30% do tempo de um programador é gasto em atividades que não são técnicas, as quais não ajudam o projeto diretamente: caminhar, negócios pessoais, etc. Nesse estudo, os programadores gastavam 6% de seu tempo andando; o que representa cerca de 2,5 horas por semana, ou cerca de 125 horas por ano. Talvez isso não pareça muito, até você perceber que os programadores gastam, a cada ano, o mesmo tempo caminhando e em treinamento, três vezes mais tempo do que gastam lendo manuais técnicos, e seis vezes mais do que falando com seus gerentes. Pessoalmente, não tenho visto muitas mudanças atualmente nesse padrão.

Questões religiosas

Os gerentes de projetos de programação nem sempre estão cientes de que determinadas questões da programação são questões religiosas. Se você for gerente e tentar exigir obediência a certas práticas de programação, estará provocando a ira de seus programadores. Eis uma lista de questões religiosas:

- Linguagem de programação
- Estilo de endentação
- Colocação de chaves
- Escolha do IDE
- Estilo de comentário
- Contrapartidas entre eficiência e legibilidade
- Escolha da metodologia - por exemplo, Scrum versus Extreme Programming
- Utilitários de programação
- Convenções de atribuição de nomes
- Uso de instruções goto
- Uso de variáveis globais
- Medições, especialmente medidas de produtividade, como linhas de código por dia

O denominador comum entre esses tópicos é que a posição de um programador sobre cada um deles é um reflexo de seu estilo pessoal. Se você crê que precisa controlar um programador em qualquer uma dessas áreas religiosas, considere os seguintes aspectos:

Saiba que você está tratando com uma área sensível Sonde o programador em cada tópico emocional, antes de se comprometer.

Use "sugestões" ou "diretrizes" com relação à área Evite estabelecer "regras" ou "padrões" rígidos.

Use diplomacia nos problemas que puder, evitando mandatos explícitos Para usar de diplomacia quanto ao estilo de indentação ou à colocação de chaves, exija que o código-fonte passe por um formatador de uma impressora que utilize impressão elegante, antes que ele seja declarado pronto. Deixe essa impressora fazer a formatação, rara usar de diplomacia quanto ao estilo de comentário, exija que todo código seja revisto e que um código que não esteja claro seja modificado até se tornar compreensível.

Faça seus programadores desenvolverem seus próprios padrões Conforme já foi mencionado, os detalhes de um padrão específico são frequentemente menos importantes do que o fato de existir um padrão. Não estabeleça padrões para seus programadores, mas insista para que eles padronizem nas áreas importantes para você.

Quais entre os tópicos religiosos são importantes o bastante para justificar a ida ao confessionário? A conformidade nas questões de estilo secundárias em qualquer área provavelmente não produziria vantagens suficientes para contrabalançar os efeitos da moral mais baixa. Se você encontrar um uso indiscriminado de instruções goto ou variáveis globais, estilos ilegíveis ou outras práticas que afetem projetos inteiros, esteja preparado para tolerar alguns atritos para melhorar a qualidade do código. Se seus programadores são conscienciosos, isso raramente se constitui num problema. As maiores lutas tendem a ser sobre nuances do estilo de codificação e você pode ficar fora delas sem nenhuma perda para o projeto.

11.11 Gerenciando seu gerente

No desenvolvimento de software, é comum encontrar gerentes que não são técnicos, assim como gerentes que têm experiência técnica, mas que estão 10 anos atrasados. Gerentes tecnicamente competentes e tecnicamente atualizados são raros. Se você trabalha para um deles, faça o máximo para manter seu emprego. Trata-se de uma vantagem incomum.

Caso seu gerente seja daqueles típicos, você está diante da tarefa nada invejável de gerenciar seu gerente. "Gerenciar seu gerente" significa que você precisa dizer a ele o que fazer, e não o contrário. O segredo é fazer isso de um modo que ele continue a acreditar que você é que está sendo gerenciado. Eis algumas estratégias para gerenciar seu gerente:

• Plante ideias sobre o que você quer fazer e depois espere que seu gerente tenha uma tempestade cerebral (sua ideia) para decidir fazer o que você queria.

• Oriente seu gerente sobre a maneira certa de fazer as coisas. Essa é uma tarefa que não termina, pois os gerentes frequentemente são promovidos, transferidos ou despedidos.

• Focalize os interesses de seu gerente, fazendo o que ele quer que você faça, e não o perturbe com detalhes de implementação desnecessários. (Considere isso um "encapsulamento" de seu trabalho.)

• Recuse-se a fazer o que seu gerente lhe disser e insista em fazer seu trabalho da maneira correta.

• Encontre outro emprego.

A melhor solução a longo prazo é tentar educar seu gerente. Essa nem sempre é uma tarefa fácil

Pontos-chave

- As boas práticas de codificação podem ser obtidas por meio de padrões impostos ou por meio de estratégias diplomáticas.
- O controle de configuração, quando corretamente aplicado, torna o trabalho dos programadores mais fácil. Isso inclui especialmente o controle de alteração.
- Uma boa estimativa de software é um desafio significativo. Os segredos para o sucesso são o uso de várias estratégias, o ajuste de suas estimativas à medida que você trabalha no projeto, e o uso de dados para criar as

estimativas.

- O segredo para o êxito do controle da construção está na medição. Você pode encontrar várias maneiras de medir qualquer aspecto de um projeto; seja qual for a sua maneira, ainda será melhor do que não fazer nenhuma medição. Uma medição precisa é a chave para elaborar um cronograma também preciso, para o controle eficaz da qualidade e para melhorar seu processo de desenvolvimento.
- Programadores gerentes são pessoas, e trabalham melhor quando tratados dessa forma.