CAPÍTULO

4 Princípios Que Orientam a Prática

Conceitos-Chave

Em um livro que pesquisa o cotidiano e reflexões dos engenheiros de software, Ellen Ullman [Ull 97] descreve um pouco da experiência de vida, à medida que relata os pensamentos do profissional atuante sob pressão:

> Não tenho ideia de que horas são. Não há janelas neste escritório, nem relógio, somente o LED vermelho de um micro-ondas piscando 12:00, 12:00, 12:00, 12:00. Joel e eu estamos programando há dias. Temos um defeito, um demônio teimoso... Da mesma forma, o pisca vemelho não se acerta em temporização, como se fosse uma representação de nossos cérebros, que, de alguma forma, acabaram sincronizados com a mesma faísca temporal do LED...
>
> Em que estamos trabalhando?... Os detalhes me escapam agora. Podemos estar auxiliando os pobres ou sintonizando uma série de rotinas de baixo nível, a fim de verificar os bits de um protocolo de distribuição de dados. Nem me importo. Deveria me importar; em uma outra época — talvez, quando sairmos desta sala cheia de computadores —, me importarei muitíssimo com o porquê, para quem e para qual finalidade estou desenvolvendo software. Mas neste exato momento: não. Ultrapassei uma camada na qual o mundo real e seus usuários não mais importam. Sou um engenheiro de software...

Muitos leitores estarão capacitados para lidar com esse lado sombrio da engenharia de software, com ponderação.

Panorama

O que é? A prática da engenharia de software consiste em uma série de princípios, conceitos, métodos e ferramentas que devem ser considerados ao planejar e desenvolver um software. Princípios que direcionem a ação estabelecem a infraestrutura a partir da qual a engenharia de software é conduzida.

Quem realiza? Praticantes (engenheiros de software) e seus coordenadores (gerentes) desenvolvem uma variedade de tarefas de engenharia de software.

Por que é importante? O processo de software propicia a todos os envolvidos na criação de um sistema computacional ou produto para computador um roteiro para conseguir chegar a um destino com sucesso. A prática fornece o detalhamento necessário para seguir ao longo da estrada. Aponta para onde estão as pontes, os conjuntos de rodovias, as bifurcações. Auxilia a compreender os conceitos e princípios que devem ser compreendidos e seguidos para que se dirija com rapidez e segurança. Orienta quanto ao como seguir, onde diminuir e aumentar o ritmo. No contexto da engenharia de software, a prática consiste no que se realiza diariamente, à medida que o software evolui de ideia a realidade.

Quais são as etapas envolvidas? Aplicam-se três elementos práticos, independentemente do modelo de processo escolhido. São eles: princípios, conceitos e métodos. Um quarto elemento de prática, ferramentas, sustenta a aplicação dos métodos.

Qual é o artefato? A prática engloba as atividades técnicas que produzem todos os artefatos definidos pelo modelo de processo de software escolhido.

Como garantir que o trabalho foi realizado corretamente? Primeiro, compreenda completamente os princípios aplicados ao trabalho (por exemplo, projeto) que está sendo realizado no momento. Em seguida, esteja certo de que escolheu um método apropriado, certifique-se de ter compreendido como aplicar o método, use ferramentas automatizadas, quando estas forem apropriadas à tarefa, e seja inflexível quanto à necessidade de técnicas para se assegurar da qualidade dos produtos de trabalho produzidos.

As pessoas que criam software praticam a arte ou ofício ou disciplina[1] em que consiste a engenharia de software. Mas em que consiste a "prática" de engenharia de software? De forma genérica, *prática* é um conjunto de conceitos, princípios, métodos e ferramentas aos quais um engenheiro de software recorre diariamente. A prática permite aos coordenadores (gerentes) administrar os projetos e aos engenheiros de software criar programas computacionais. A prática preenche um modelo de processo de software com recursos de como fazer para ter o trabalho realizado em termos de gerenciamento e de tecnologia. Transforma uma abordagem desfocada e confusa em algo mais organizado, mais efetivo e mais propenso a obter sucesso.

Diversos aspectos da engenharia de software serão examinados ao longo do livro.

Neste capítulo, o foco será em princípios e conceitos que norteiam a prática da engenharia de sofwtare em geral.

4.1 Conhecimento da Engenharia de Software

Em um editorial publicado na *IEEE Software* uma década atrás, Steve McConnell [McC99] fez o seguinte comentário:

> Muitos desenvolvedores (de software) veem o conhecimento da engenharia de software quase que exclusivamente como um conhecimento de tecnologias específicas: Java, Perl, html, C++, Linux, Windows NT etc. O conhecimento de detalhes específicos de tecnologia é necessário para desenvolver a programação de computador. Se alguém o contrata para desenvolver um programa em C++, você deverá saber algo sobre C++ para fazer seu programa funcionar.
>
> Frequentemente, diz-se que conhecimento sobre desenvolvimento de software tem uns três anos de vida: metade do que se precisa saber hoje estará obsoleto daqui a três anos. No que diz respeito ao conhecimento de tecnologia, essa afirmação está próxima da verdade. Mas há um outro tipo de conhecimento sobre desenvolvimento de software — um tipo visto como "princípios da engenharia de software" — que não possui três anos de meia-vida. Tais princípios provavelmente servirão ao programador profissional durante toda a sua carreira.

McConnell continua afirmando que a base do conhecimento em engenharia de software (por volta do ano 2000) evoluiu para uma "essência estável" que ele estimou representar cerca de "75% do conhecimento necessário para desenvolver um sistema complexo". No entanto, no que consiste essa essência estável?

Como indica McConnell, princípios essenciais — ideias elementares que guiam engenheiros de software em seus trabalhos — fornecem, em nossos dias, uma infraestrutura a partir da qual os modelos de engenharia de software, métodos e ferramentas podem ser aplicados e avaliados.

4.2 Princípios Fundamentais

"Teoricamente, não há nenhuma diferença entre a teoria e a prática. Porém, na prática, ela existe."

Jan van de Snepscheut

A engenharia de software é norteada por um conjunto de princípios fundamentais que auxiliam na aplicação de um processo de software significativo e na execução de métodos de engenharia de software efetivos. No nível relativo a processo, os princípios fundamentais estabelecem uma infraestrutura filosófica que guia uma equipe de software à medida que desenvolve atividades de apoio e estruturais, navega no fluxo do processo e cria um conjunto de artefatos de engenharia de software. Quanto à prática, os princípios fundamentais estabelecem um artefatos e regras que servem como guias conforme se analisa um problema, projeta uma solução, implementa e testa uma solução e, por fim, emprega o software na comunidade de usuários.

No Capítulo 1, identificou-se uma série de princípios fundamentais que abrange o processo e a prática da engenharia de software: (1) fornecer valor aos usuários finais, (2) simplificar, (3)

1 Certos autores argumentam que um desses termos exclui outros. Na realidade, a engenharia de software se aplica a todos os três.

manter a visão (do produto e do projeto), (4) reconhecer que outros usuários consomem (e devem entender) o que se produz, (5) manter abertura para o futuro, (6) planejar antecipadamente para o reúso, e (7) raciocinar!

Embora tais princípios gerais sejam importantes, são caracterizados em um nível tão elevado de abstração que, às vezes, torna-se difícil a tradução para a prática diária da engenharia de software. Nas subseções seguintes, há um maior detalhamento dos princípios essenciais que conduzem o processo e a prática.

4.2.1 Princípios que orientam o processo

Na Parte 1 deste livro, discutiu-se a importância do processo de software e descreveram-se os diferentes modelos de processos propostos para o trabalho de engenharia de software. Não importando se um modelo é linear ou iterativo, prescritivo ou ágil, pode ser caracterizado usando-se uma metodologia de processo genérica aplicável a todos os modelos de processo. O conjunto de princípios fundamentais apresentados a seguir pode ser aplicado à metodologia e, por extensão, a todos os processos de software.

Todo projeto e toda equipe são únicos. Isso significa que se deve adaptar o processo para que melhor se ajuste às suas necessidades.

Princípio 1. ***Seja ágil.*** Não importa se o modelo de processo que você escolheu é prescritivo ou ágil, os princípios básicos do desenvolvimento ágil devem comandar sua abordagem. Todo aspecto do trabalho deve enfatizar a economia de ações — mantenha a abordagem técnica tão simples quanto possível, mantenha os produtos tão concisos quanto possível e tome decisões localmente sempre que possível.

Princípio 2. ***Concentre-se na qualidade em todas as etapas.*** A condição final para toda atividade, ação e tarefa do processo deve focar na qualidade do produto produzido.

Princípio 3. ***Esteja pronto para adaptações.*** Processo não é uma experiência religiosa e não há espaço para dogma. Quando necessário, adapte sua abordagem às restrições impostas pelo problema, pelas pessoas e pelo próprio projeto.

Princípio 4. ***Monte uma equipe efetiva.*** O processo e a prática de engenharia de software são importantes, mas o fator mais importante são as pessoas. Forme uma equipe que se auto-organize, que tenha confiança e respeito mútuos.

Princípio 5. ***Estabeleça mecanismos para comunicação e coordenação.*** Os projetos falham devido à omissão de informações importantes nas "fendas" da estrutura do programa e/ou devido a indivíduos que falham na coordenação de seus esforços para criar um produto final bem-sucedido. Esses são itens de gerenciamento e devem ser direcionados.

Princípio 6. ***Gerencie mudanças.*** A abordagem pode ser tanto formal quanto informal, entretanto os mecanismos devem ser estabelecidos para gerenciar a maneira como as mudanças serão solicitadas, avaliadas, aprovadas e implementadas.

"A verdade da questão é que sempre sabemos a coisa certa a ser feita. A parte difícil é fazê-la."

General H. Norman Schwarzkopf

Princípio 7. ***Avalie os riscos.*** Uma série de coisas pode dar errada quando um software é desenvolvido. É essencial estabelecer planos de contingência.

Princípio 8. ***Gere artefatos que forneçam valor para outros.*** Crie apenas artefatos que proporcionarão valor para outro processo, atividades, ações ou tarefas. Todo artefato produzido como parte da prática da engenharia de software será repassado para alguém mais. Uma lista de funções e características requisitadas será repassada para a pessoa (ou pessoas) que irá desenvolver um projeto, o projeto será repassado para aqueles que gerarão o código e assim por diante. Certifique-se de que o artefato contenha a informação necessária sem ambiguidades ou omissões.

A Parte 4 deste livro enfoca o projeto, os fatores de gerenciamento do processo e aborda os aspectos variados de cada um desses princípios com certo detalhe.

4.2.2 Princípios que orientam a prática

A prática da engenharia de software tem um objetivo primordial único — entregar dentro do prazo, com alta qualidade, o software operacional contendo funções e características que satisfaçam as necessidades de todos os envolvidos. Para atingir esse objetivo, deve-se adotar um conjunto de princípios fundamentais que orientem o trabalho técnico. Tais princípios são importantes, independentemente do método de análise ou projeto aplicado, das técnicas de desenvolvimento (por exemplo, linguagem de programação, ferramentas de automação) escolhidas, ou da abordagem de verificação e validação utilizada. O cumprimento de princípios fundamentais a seguir é essencial para a prática de engenharia de software:

PONTO-CHAVE

Os problemas tornam-se mais fáceis para resolver quando subdivididos por interesses, cada um distinto, individualmente solucionável e passível de verificação.

Princípio 1. *Divida e conquiste.* De forma mais técnica, a análise e o projeto sempre devem enfatizar a *separação por interesses*[2] (— *separation of concerns* — SoC). Um problema terá sua resolução mais fácil se for subdividido em conjuntos de interesses. Na forma ideal, cada interesse fornece uma funcionalidade distinta a ser desenvolvida, e, em alguns casos, validada, independentemente de outros negócios.

Princípio 2. *Compreenda o uso da abstração.* Na essência, abstrair é simplificar algum elemento complexo de um sistema utilizado para comunicar significado em uma única frase. Quando se usa a abstração *planilha*, assume-se que se compreende o que vem a ser uma planilha, a estrutura geral de conteúdo que uma planilha apresenta e as funções típicas que podem ser aplicadas a ela. Na prática de engenharia de software, usam-se muitos níveis diferentes de abstração, cada um incorporando ou implicando um significado que deve ser comunicado. No trabalho de análise de projeto, uma equipe de sofware normalmente inicia com modelos que representam altos níveis de abstração (por exemplo, planilha) e, aos poucos, refina tais modelos em níveis de abstração mais baixos (por exemplo, uma coluna ou uma função de soma).

Joel Spolsky [Spo02] sugere que "todas as abstrações não triviais são, até certo ponto, frágeis". O objetivo de uma abstração é eliminar a necessidade de comunicar detalhes. Algumas vezes, efeitos problemáticos se precipitam em virtude da "aresta" dos detalhes. Sem a compreensão dos detalhes, a causa de um problema não poderá ser facilmente diagnosticada.

Princípio 3. Esforce-se por consistência. Seja criando um modelo de requisitos, desenvolvendo um projeto de software, gerando código-fonte, seja criando pacotes de teste, o princípio de consistência sugere que um contexto familiar facilita o uso do software. Consideremos, por exemplo, o projeto de uma interface para o usuário de uma WebApp. A colocação consistente de menu de opções, o uso consistente de um esquema de cores e de ícones identificáveis, tudo colabora para uma interface ergonomicamente forte.

Princípio 4. *Foque na transferência de informação.* Software trata a transferência de informações: do banco de dados para um usuário final, de um sistema judiciário para uma aplicação na Web (WebApp), do usuário final para uma interface gráfica (GUI), de um sistema operacional de um componente de software para outro; a lista é quase infinita. Em todos os casos, a informação flui através de uma interface, e, como consequência, há a possibilidade de erros, omissões e ambiguidade. A implicação desse princípio é que se deve prestar especial atenção para interfaces de análise, projeto, construção e testes.

Princípio 5. Construa *software que apresente modularidade efetiva.* A separação por interesse (Princípio 1) estabelece uma filosofia para software. A *modularidade* fornece um mecanismo para colocar a filosofia em prática. Qualquer sistema complexo pode ser dividido em módulos (componentes), porém a boa prática de engenharia de software demanda mais

2 N. de T.: A palavra *concern* foi traduzida como interesse ou afinidade. Dividir é uma técnica utilizada para lidar com complexidade. Uma das estratégias para dividir é separar usando critérios como interesses ou afinidade tanto, no domínio do problema como no domínio da tecnologia do software.

que isso . A modularidade deve ser *efetiva*. Isto é, cada módulo deve focalizar exclusivamente um aspecto bem restrito do sistema — deve ser coeso em sua função e/ou apresentar conteúdo bem preciso. Além disso, os módulos devem ser interconectados de uma maneira relativamente simples — cada módulo deve apresentar baixo acoplamento com outros módulos, fontes de dados e outros aspectos ambientais.

Use padrões (Capítulo 12) para armazenar conhecimento e experiência para as futuras gerações de engenheiros de software.

Princípio 6. Padronize. Brad Appleton [App00] propõe que:

O objetivo da padronização é criar uma fonte literária para ajudar os desenvolvedores de software a solucionar problemas recorrentes, encontrados ao longo de todo o desenvolvimento de software. Os padrões ajudam a criar uma linguagem compartilhada para transmitir conteúdo e experiência acerca dos problemas e de suas soluções. Codificar formalmente tais soluções e suas relações permite que se armazene com sucesso a base do conhecimento, a qual define nossa compreensão sobre boas arquiteturas, correspondendo às necessidades de seus usuários.

Princípio 7. *Quando possível, represente o problema e sua solução sob uma série de perspectivas diferentes.* Ao analisar um problema e sua solução sob uma série de perspectivas diferentes é mais provável que se obtenha uma melhor visão e que os erros e omissões sejam revelados. Por exemplo, um modelo de requisitos pode ser representado usando um ponto de vista orientado a dados, um ponto de vista orientado a funções ou um ponto de vista comportamental (Capítulos 6 e 7). Cada um deles fornece uma visão diferente do problema e de seus requisitos.

Princípio 8. Lembre-se *de que alguém fará a manutenção do software.* A longo prazo, conforme defeitos são descobertos, o software será corrigido, adaptado de acordo com as mudanças de seu ambiente e ampliado conforme solicitação por mais capacidade dos envolvidos. As atividades de manutenção podem ser facilitadas se for aplicada uma prática de engenharia de software consistente ao longo do processo.

Esses princípios não constituem o todo necessário para a construção de um software de alta qualidade, mas realmente estabelecem uma base para todo método de engenharia de software discutido neste livro.

4.3 Princípios das Atividades Metodológicas

"O engenheiro ideal é uma composição: não é um cientista, não é um matemático, não é um sociólogo ou escritor; mas pode utilizar o conhecimento e as técnicas de qualquer uma ou de todas as disciplinas para resolver os problemas de engenharia."

N. W. Dougherty

Nas seções seguintes, serão apresentados princípios que constituem uma forte base para o sucesso de cada atividade metodológica genérica, definida como parte do processo de software. Em muitos casos, são um aprimoramento dos princípios apresentados na Seção 4.2, fundamentais e simplificados, situando-se em um nível de abstração mais baixo.

4.3.1 Princípios da comunicação

Antes que os requisitos dos clientes sejam analisados, modelados ou especificados, eles devem ser coletados através da atividade de comunicação. Um cliente apresenta um problema que pode ser amenizado por uma solução baseada em computador. Responde-se ao pedido de ajuda e inicia-se a comunicação. Entretanto, o percurso da comunicação ao entendimento é, frequentemente, acidentado.

A comunicação efetiva (entre parceiros técnicos com o cliente, com outros parceiros interessados e com gerenciadores de projetos) constitui uma das atividades mais desafiadoras. Nesse contexto, discutem-se princípios de comunicação aplicados na comunicação com o cliente. Entretanto, muitos se aplicam a todas as formas de comunicação.

Princípio 1. *Ouça.* Concentre-se mais em ouvir do que em se preocupar com respostas. Solicite esclarecimento caso necessário, e evite interrupções constantes. Nunca se mostre contestador tanto em palavras quanto em ações (por exemplo, revirar olhos e menear a cabeça) enquanto uma pessoa estiver falando.

Antes de se comunicar, assegure-se de compreender o ponto de vista alheio, suas necessidades, e saiba ouvir.

"Planeje perguntas e respostas, trace o caminho mais curto para a maioria das perplexidades."

Mark Twain

Princípio 2. ***Prepare-se antes de se comunicar.*** Dedique tempo para compreender o problema antes de se encontrar com outras pessoas. Se necessário, realize algumas pesquisas para entender o jargão da área de negócios em questão. Caso seja sua a responsabilidade pela condução de uma reunião, prepare, com antecedência, uma agenda.

Princípio 3. ***Alguém deve facilitar a atividade.*** Toda reunião de comunicação deve ter um líder (um facilitador) para manter a conversação direcionada e produtiva, (2) para mediar qualquer conflito que ocorra e (3) para garantir que outros princípios sejam seguidos.

Princípio 4. ***Comunique-se pessoalmente.*** No entanto, em geral, comunicar-se pessoalmente é mais produtivo tendo um outro representante presente. Por exemplo, um participante pode criar um desenho ou um esboço de documento que servirá como foco para a discussão.

Princípio 5. ***Anote e documente as decisões.*** As coisas tendem a cair no esquecimento. Algum participante da comunicação deve servir como um "gravador" e anotar todos os pontos e decisões importantes.

Princípio 6. ***Esforce-se por colaboração.*** Colaboração e consenso ocorrem quando o conhecimento coletivo dos membros da equipe for usado para descrever funções e características do produto ou sistema. Cada pequena colaboração servirá para estabelecer confiança entre os membros e chegar a um objetivo comum.

Princípio 7. ***Mantenha o foco e crie módulos para a discussão.*** Quanto mais pessoas envolvidas, maior a probabilidade de a discussão seguir de um tópico a outro. O facilitador deve manter a conversa modular, abandonando um tópico somente depois de este ter sido resolvido (veja, entretanto, o Princípio 9).

Princípio 8. ***Faltando clareza, represente graficamente.*** A comunicação verbal flui até certo ponto. Um esboço ou uma representação gráfica pode permitir maior clareza quando palavras são insuficientes.

Princípio 9. ***(a) Uma vez de acordo, siga em frente. (b) Se não chegar a um acordo, siga em frente. (c) Se uma característica ou função estiver obscura e não puder ser elucidada no momento, siga em frente.*** A comunicação, assim como qualquer outra atividade da engenharia de software, toma tempo. Em vez de ficar interagindo indefinidamente, os participantes precisam reconhecer que muitos tópicos exigem discussão (veja o Princípio 2) e que "seguir em frente" é, algumas vezes, a melhor maneira de alcançar agilidade na comunicação.

O que Acontecerá caso não consiga chegar a um acordo com um cliente sobre alguma questão relativa ao projeto?

Princípio 10. ***Negociação não é uma contestação nem um jogo. Funciona melhor quando ambas as partes saem ganhando.*** Há muitas ocasiões em que se têm de negociar funções e características, prioridades e prazos de entrega. Se a equipe interagiu adequadamente, todas as partes têm um objetivo comum. Mesmo assim, a negociação exigirá compromisso de todos.

INFORMAÇÕES

A diferença entre clientes e usuários finais

Os engenheiros de software se comunicam com muitos interessados diferentes, mas clientes e usuários finais têm o maior impacto sobre o trabalho técnico que virá a seguir. Em alguns casos, cliente e usuário final são a mesma pessoa, mas, para muitos projetos, são pessoas que trabalham para gerentes diferentes, em organizações diferentes.

Cliente é a pessoa ou grupo que: (1) originalmente, requisita o software a ser construído, (2) define os objetivos gerais do negócio para o software, (3) fornece os requisitos básicos do produto e (4) coordena os recursos financeiros para o projeto. Em uma negociação de sistemas ou de produtos, o cliente, com frequência, é o departamento de marketing. Em um ambiente de tecnologia da informação (TI), o cliente pode ser um departamento ou componente de negócio.

Usuário final é uma pessoa ou grupo que (1) irá realmente usar o software que é construído para atingir algum propósito de negócio e (2) irá definir os detalhes operacionais do software de modo que o objetivo seja alcançado.

CasaSegura

Erros de comunicação

Cena: Área de trabalho da equipe de engenharia de software

Atores: Jamie Lazar, membro da equipe de software; Vinod Raman, membro da equipe de software; Ed Robbins, membro da equipe de software.

Conversa:

Ed: "O que você ouviu falar sobre o projeto CasaSegura?"

Vinod: "A reunião inicial está marcada para a próxima semana."

Jamie: "Eu já andei investigando, mas não deu muito certo."

Ed: "O que você quer dizer com isso?"

Jamie: "Bem, dei uma ligada para Lisa Perez. Ela é a manda-chuva de marketing dessa coisa."

Vinod: "E . . . ?"

Jamie: "Eu queria que ela me desse informações sobre as características e funções do CasaSegura... esse tipo de coisa. Mas, em vez disso, ela começou a me fazer perguntas sobre sistemas de segurança, sistemas de vigilância... Eu não sou um especialista na área."

Vinod: "O que isso lhe diz?" (Jamie encolhe os ombros.)

Vinod: "Isso quer dizer que o marketing precisará de nós como consultores e que é melhor nos prepararmos nesta área de produto antes da primeira reunião. Doug disse que quer que 'colaboremos' com nosso cliente; portanto, é melhor aprendermos como fazer isso."

Ed: "Provavelmente teria sido melhor ter dado uma passada na sala dela. Telefonemas simplesmente não funcionam bem para esse tipo de coisa."

Jamie: "Vocês dois estão certos. Temos que agir em conjunto ou nossos primeiros contatos serão difíceis."

Vinod: "Vi o Doug lendo um livro sobre 'engenharia de requisitos'. Aposto que ele enumera alguns princípios de comunicação efetiva. Vou pedi-lo emprestado amanhã mesmo."

Jamie: "Boa ideia... depois você poderá nos ensinar."

Vinod (sorrindo): "É isso aí."

4.3.2 Princípios de planejamento

A atividade de comunicação ajuda a definir metas gerais e objetivos (sujeitos a mudanças, é claro, à medida que o tempo passa). Entretanto, compreender essas metas e objetivos não é o mesmo que definir um plano para alcançá-los. A atividade de planejamento engloba um conjunto de práticas técnicas e gerenciais que permitem à equipe de software definir um roteiro à medida que segue na direção de seus objetivos estratégicos e táticos.

"Na preparação para a batalha sempre achei que os planos fossem inúteis, mas planejamento é indispensável."

General Dwight D. Eisenhower

Por mais que se tente, é impossível prever com exatidão como um projeto de software irá evoluir. Não há nenhuma maneira simples para determinar quais problemas técnicos não previstos serão encontrados, quais enganos ocorrerão ou quais itens de negócio sofrerão mudanças. Ainda assim, uma boa equipe deve planejar sua abordagem.

Há muitas filosofias de planejamento.[3] Algumas pessoas são "minimalistas", afirmando que alterações, frequentemente, evidenciam a necessidade de um plano detalhado. Outras são "tradicionalistas", afirmando que o plano fornece um roteiro efetivo e que, quanto mais detalhes apresentar, menos probabilidade terá a equipe de se perder. Outras ainda são "agilistas", afirmando que um rápido "planejamento do jogo" pode ser necessário, entretanto, o roteiro surgirá como "verdadeiro trabalho" no início do software.

WebRef

Um excelente banco de informações sobre planejamento e gerenciamento de projetos pode ser encontrado em **www.4pm.com/repository.htm.**

O que fazer? Em muitos projetos, planejamento em excesso representa consumo de tempo sem resultado produtivo (mudanças demais), entretanto, pouco planejamento é uma receita para o caos. Como a maioria das coisas na vida, o planejamento deveria ser conduzido de forma moderada, o suficiente para servir de guia para a aquipe – nem mais, nem menos. Não importando o rigor com o qual o planejamento seja feito, os seguintes princípios sempre se aplicam:

Princípio 1. ***Compreenda o escopo do projeto.*** É impossível usar um roteiro caso não se saiba para onde se está indo. O escopo indica ao grupo um destino.

Princípio 2. ***Envolva os interessados na atividade de planejamento.*** Os interessados definem prioridades e estabelecem as restrições de projeto. Para adequar essas característi-

3 Na Parte 4 do livro é apresentada uma discussão detalhada sobre planejamento e gerenciamento de projetos de software.

cas, os engenheiros devem negociar com frequência a programação de entrega, cronograma e outras questões relativas ao projeto.

Princípio 3. *Reconheça que o planejamento é iterativo.* Um plano de projeto jamais é gravado na pedra. Conforme se inicia o trabalho, muito provavelmente ocorrerão alterações. Como consequência, o plano deverá ser ajustado para incluir as alterações. Além do mais, os modelos de processos incremental e iterativo exigem replanejamento após a entrega de cada incremento de software baseado em feedbacks recebidos dos usuários.

"O sucesso deve-se mais ao bom senso do que à genialidade."

An Wang

Princípio 4. *Faça estimativas com base no que conhece.* O objetivo da estimativa é oferecer indicações de esforço, custo e prazo para a realização, com base na compreensão atual do trabalho a realizar. Se a informação for vaga ou não confiável, as estimativas serão igualmente não confiáveis.

Princípio 5. *Considere os riscos ao definir o plano.* Caso tenha identificado riscos de alto impacto e alta probabilidade, o planejamento de contingência será necessário. Além disso, o plano de projeto (inclusive o cronograma) deve ser ajustado para incluir a tendência de um ou mais desses riscos ocorrerem.

Princípio 6. *Seja realista.* As pessoas não trabalham 100% de todos os dias. Sempre há interferência de ruído em qualquer comunicação humana. Omissões e ambiguidades são fatos da vida. Mudanças ocorrem. Até mesmo os melhores engenheiros de software cometem erros. Essas e outras realidades devem ser consideradas quando se estabelece um plano de projeto.

PONTO-CHAVE

O termo particularidade refere-se a detalhes por meio dos quais alguns elementos do planejamento são representados ou conduzidos.

Princípio 7. *Ajuste particularidades ao definir o plano.* *Particularidades* referem-se ao nível de detalhamento introduzido conforme o plano de projeto é desenvolvido. Um plano com alto grau de particularidade fornece considerável detalhamento de tarefas planejadas para incrementos em intervalos relativamente curtos para que o rastreamento e controle ocorram com frequência. Um plano com baixo grau de particularidade resulta em tarefas mais amplas para intervalos maiores. Em geral, particularidades variam de alto para baixo conforme o cronograma de projeto se distancia da data corrente. Nas semanas ou meses seguintes, o projeto pode ser planejado com detalhes significativos. As atividades que não serão realizadas por muitos meses não requerem alto grau de particularidade (muito pode ser alterado).

Princípio 8. *Defina como se pretende garantir a qualidade.* O plano deve determinar como a equipe pretende garantir a qualidade. Se forem necessárias revisões técnicas[4], deve-se agendá-las. Se a programação pareada for utilizada (Capítulo 3), deve-se definir claramente dentro do plano.

Princípio 9. *Descreva como pretende adequar as alterações.* Mesmo o melhor planejamento pode ser prejudicado por alteração sem controle. Deve-se identificar como as alterações serão integradas ao longo do trabalho em engenharia. Por exemplo, o cliente pode solicitar uma alteração a qualquer momento? Se for solicitada uma mudança, a equipe é obrigada a implementá-la imediatamente? Como é avaliado o impacto e o custo de uma alteração?

Princípio 10. *Verifique o plano frequentemente e faça ajustes necessários.* Os projetos de software atrasam uma vez ou outra. Portanto, é de bom senso checar diariamente seu progresso, procurando por áreas ou situações problemáticas nas quais o que foi programado não está em conformidade com o trabalho real a ser conduzido. Ao surgir um descompasso, deve-se ajustar o plano adequadamente. Para máxima eficiência, todos da equipe de software devem participar da atividade de planejamento. Só dessa maneira os membros estarão comprometidos (engajados) com o plano.

4 As revisões técnicas são discutidas no Capítulo 15.

4.3.3 Princípios de modelagem

Criam-se modelos para uma melhor compreensão do que será realmente construído. Quando a entidade for algo físico (por exemplo, um edifício, um avião, uma máquina), pode-se construir um modelo que seja idêntico na forma e no formato, porém menor em escala. Entretanto, quando a entidade a ser construída for software, nosso modelo deve assumir uma forma diferente. Deve ser capaz de representar as informações que o software transforma, a arquitetura e as funções que possibilitam a transformação, as características que os usuários desejam e o comportamento do sistema à medida que a transformação ocorra. Os modelos devem cumprir esses objetivos em diferentes níveis de abstração — primeiro, descrevendo o software do ponto de vista do cliente e, posteriormente, em um nível mais técnico.

PONTO-CHAVE

Os modelos de requisitos representam os requisitos dos clientes. Os modelos de projeto oferecem uma especificação concreta para a construção do software.

No trabalho de engenharia de software, podem ser criadas duas classes de modelos: modelos de requisitos e modelos de projeto. Os *modelos de requisitos* (também denominados *modelos de análise*) representam os requisitos dos clientes, descrevendo o software em três domínios diferentes: o domínio da informação, o domínio funcional e o domínio comportamental. Os *modelos de projeto* representam características do software que ajudam os desenvolvedores a construí-lo efetivamente: a arquitetura, a interface para o usuário e os detalhes quanto a componentes.

Em seu livro sobre modelagem ágil, Scott Ambler e Ron Jeffries [Amb02b] estabelecem uma série de princípios[5] de modelagem destinados àqueles que usam o modelo de processos ágeis (Capítulo 3), mas são apropriados para todos os engenheiros de software que executam ações e tarefas de modelagem:

Princípio 1. ***O objetivo principal da equipe de software é construir software, e não criar modelos.*** Agilidade significa entregar software ao cliente no menor prazo possível. Os modelos que fazem com que isso aconteça são criações valiosas, entretanto, os que retardam o processo oferecem pouca novidade, devem ser evitados.

Princípio 2. ***Viaje leve — não crie mais modelos do que necessita.*** Todo modelo criado deve ser atualizado quando ocorrem alterações. E, mais importante, todo modelo novo demanda tempo que poderia ser dispendido em construção (codificação e testes). Portanto, crie somente aqueles modelos que facilitem mais e diminuam o tempo para a construção do software.

O objetivo de qualquer modelo é transmitir informações. Para tanto, use um formato consistente. Considere o fato de não estar lá para explicá-lo. Ele deve ser autoexplicativo.

Princípio 3. ***Esforce-se ao máximo para produzir o modelo mais simples possível.*** Não exagere no software [Amb02b]. Mantendo modelos simples, o software resultante também será simples. O resultado será um software mais fácil de ser integrado, testado e mantido. Além disso, modelos simples são mais fáceis de compreender e criticar, resultando em uma forma contínua de *feedback* que otimiza o resultado final.

Princípio 4. ***Construa modelos que facilitem alterações.*** Considere que os modelos mudarão, mas, ao considerar tal fato, não seja relapso. Por exemplo, uma vez que os requisitos serão alterados, há uma tendência a dar pouca atenção a seus modelos. Por quê? Porque se sabe que mudarão de qualquer forma. O problema dessa atitude é que sem um modelo de requisitos razoavelmente completo, criar-se-á um projeto (modelo de projeto) que invariavelmente irá deixar de lado funções e características importantes.

Princípio 5. ***Seja capaz de estabelecer um propósito claro para cada modelo.*** Toda vez que criar um modelo, pergunte-se o motivo para tanto. Se você não for capaz de dar justificativas sólidas para a existência do modelo, não desperdice tempo com ele.

Princípio 6. ***Adapte o modelo desenvolvido ao sistema à disposição.*** Talvez seja necessário adaptar a notação ou as regras do modelo ao aplicativo; por exemplo, um aplicativo

5 Os princípios citados nessa seção foram resumidos e reescritos de acordo com os propósitos do livro.

de videogame pode requerer uma técnica de modelagem diferente daquela utilizada em um software embutido e de tempo real que controla o motor de um automóvel.

Princípio 7. ***Crie modelos úteis, mas esqueça a construção de modelos perfeitos.*** Ao construir modelos de requisitos e de projetos, um engenheiro de software atinge um ponto de retornos decrescentes. Isto é, o esforço necessário para fazer o modelo absolutamente completo e internamente consistente não vale os benefícios resultantes. Estaria eu sugerindo que a modelagem deve ser descuidada ou de baixa qualidade? A resposta é "não". Porém, a modelagem deve ser conduzida tendo em vista as etapas de engenharia de software seguintes; iterar indefinidamente para criar um modelo "perfeito" não atende à necessidade de agilidade.

Princípio 8. ***Não seja dogmático quanto à sintaxe do modelo. Se esta transmite o conteúdo com sucesso, a representação é secundária.*** Embora todos de uma equipe devam tentar usar uma notação consistente durante a modelagem, a característica mais importante do modelo reside em transmitir informações que possibilitem a próxima tarefa de engenharia. Se um modelo viabilizar isso com êxito, a sintaxe incorreta pode ser perdoada.

Princípio 9. ***Se os instintos indicam que um modelo não está correto, embora pareça correto no papel, provavelmente há motivos com os quais se preocupar.*** Se você for um engenheiro experiente, confie em seus instintos. O trabalho com software nos ensina muitas lições — muitas das quais em um nível subconsciente. Se algo lhe diz que um modelo de projeto parece falho, embora não haja provas explícitas, há motivos para dedicar tempo extra examinando o modelo ou desenvolvendo outro diferente.

Princípio 10. ***Obtenha feedback o quanto antes.*** Todo modelo deve ser revisado pelos membros da equipe de software. O objetivo das revisões é proporcionar *feedback* que seja usado para corrigir erros de modelagem, alterar interpretações errôneas, e adicionar características ou funções omitidas inadvertidamente.

Princípios de modelagem de requisitos. Ao longo das últimas três décadas, desenvolveu-se um grande número de métodos de modelagem de requisitos. Pesquisadores identificaram problemas de análise de requisitos e suas causas e desenvolveram uma série de notações de modelagem e uma série de "heurísticas"correspondentes para superá-los. Cada um dos métodos de análise possui um ponto de vista particular. Entretanto, todos estão inter-relacionados por uma série de princípios operacionais:

Princípio 1. ***O universo de informações de um problema deve ser representado e compreendido.*** O universo *de informações engloba* os dados constantes no sistema (do usuário final, de outros sistemas ou dispositivos externos), os dados que fluem para fora do sistema (via interface do usuário, interfaces de rede, relatórios, gráficos e outros meios) e a armazenagem de dados que coleta e organiza objetos de dados persistentes (isto é, dados que são mantidos permanentemente).

PONTO-CHAVE

A modelagem de análise focaliza três atributos de software: informações a serem processadas, função a ser entregue e comportamento a ser apresentado.

Princípio 2. ***As funções que o software desempenha devem ser definidas.*** As funções de software oferecem benefício direto aos usuários finais e também suporte interno para fatores visíveis aos usuários. Algumas funções transformam dados que fluem no sistema. Em outros casos, as funções exercem certo nível de controle sobre o processamento interno do software ou sobre elementos de sistema externo. As funções podem ser descritas em diferentes níveis de abstração, desde afirmação geral até uma descrição detalhada dos elementos de processo que devem ser requisitados.

Princípio 3. ***O comportamento do software (como consequência de eventos externos) deve ser representado.*** O comportamento de um software é comandado por sua interação com o ambiente externo. Alimentações fornecidas pelos usuários finais, informações referentes a controles provenientes de um sistema externo ou dados de monitoramento

"O primeiro problema do engenheiro em qualquer situação de projeto é descobrir qual é realmente o problema."

Autor desconhecido

coletados de uma rede, todos esses elementos determinam que o software se opere de maneira específica.

Princípio 4. ***Os modelos que descrevem informações, funções e comportamentos devem ser divididos para que revelem detalhes por camadas (ou hierarquicamente). A modelagem de requisitos*** é a primeira etapa da solução de um problema de engenharia de software. Permite que se entenda melhor o problema e se estabeleçam bases para a solução (projeto). Os problemas complexos são difíceis de resolver em sua totalidade. Por essa razão, deve-se usar a estratégia dividir-e-conquistar. Um problema grande e complexo é dividido em subproblemas até que cada um seja relativamente fácil de ser compreendido. Esse conceito é denominado *fracionamento* ou *separação por interesse* e é uma estratégia-chave na modelagem de requisitos.

Princípio 5. ***A análise deve partir da informação essencial para o detalhamento da implementação. A modelagem de requisitos se inicia*** pela descrição do problema sob a perspectiva do usuário final. A "essência" do problema é descrita sem levar em consideração como será implementada uma solução. Por exemplo, um jogo de videogame requer que o jogador "instrua" seu protagonista sobre qual direção seguir para continuar conforme se movimenta em situações perigosas. Essa é a essência do problema. O detalhamento da implementação (em geral descrito como parte do modelo de projeto) indica como a essência será implementada. No caso do videogame, talvez fosse usada entrada de voz. Outra alternativa seria utilizar o comando por meio do teclado, ou por joystick ou mouse para uma direção específica, ou um dispositivo com sensor de movimento poderia ser empregado externamente. Aplicando-se tais princípios, um engenheiro de software aborda um problema de forma sistemática. Entretanto, como os princípios são aplicados na prática? Essa pergunta será respondida nos Capítulos 5 a 7.

Princípios da modelagem de projetos. A modelagem de projetos de software é análoga ao planejamento de uma casa feito por um arquiteto. Ela começa pela representação do todo a ser construído (por exemplo, uma representação tridimensional da casa) e, gradualmente, foca os detalhes, oferecendo um roteiro para a sua construção (por exemplo, a estrutura do encanamento). De modo similar, a modelagem de projeto fornece uma variedade de diferentes focos do sistema.

"Primeiramente, verifique se o projeto é inteligente e precisa: feito isso, persista resolutamente. Não desista do seu propósito por causa de uma rejeição."

William Shakespeare

Não há poucos métodos para identificar os vários elementos de um projeto. Alguns são voltados a dados, permitindo que a estrutura de dados determine a arquitetura do programa e dos componentes de processamento resultantes. Outros são voltados para padrões, usando informações a respeito do domínio do problema (da modelagem dos requisitos) para desenvolver os estilos arquitetônicos e os padrões de processamento. Outros, ainda, são voltados a objetos, usando os objetos da área do problema como determinantes para a criação dos métodos e das estruturas de dados que os manipularão. Ainda assim, todos englobam uma série de princípios de projeto que podem ser aplicados independentemente do método empregado:

WebRef

Comentários mais específicos sobre o processo dos projetos, juntamente com um debate sobre sua estética, poderão ser encontrados em **cs.wwc.edu/~aabyan/Design/**.

Princípio 1. ***O projeto deve ser roteirizado para a modelagem de requisitos.***

A modelagem de requisitos descreve a área de informação do problema, funções visíveis ao usuário, desempenho do sistema e um conjunto de classes de requisitos que embala objetos de negócios juntamente com os métodos que ele serve. A modelagem de projeto traduz essa informação em forma de uma arquitetura, um conjunto de subsistemas que implementam funções mais amplas e um conjunto de componentes que são a concretização das classes de requisitos. Os elementos da modelagem de projetos devem ser roteirizados para a modelagem de requisitos.

Princípio 2. ***Sempre considere a arquitetura do sistema a ser construído.***

A arquitetura de software (Capítulo 9) é a espinha dorsal do sistema a ser construído. Afetando interfaces, estruturas de dados, desempenho e fluxo de controle de programas, maneira

pela qual os testes podem ser conduzidos, a manutenção do sistema realizada e muito mais. Por todas essas razões, o projeto deve começar com as considerações arquitetônicas. Só depois de a arquitetura ter sido estabelecida devem ser considerados os elementos relativos a componentes.

Princípio 3. ***O projeto de dados é tão importante quanto o projeto das funções de processamento.*** O projeto de dados é um elemento essencial do projeto da arquitetura. A forma como os objetos de dados são percebidos no projeto não pode ser deixada ao acaso. Um projeto de dados bem estruturado ajuda a simplificar o fluxo do programa, torna mais fácil a elaboração do projeto e a implementação dos componentes de software, tornando mais eficiente o processamento como um todo.

"As diferenças não são insignificantes — são bem semelhantes às diferenças entre Salieri e Mozart. Estudos após estudos demonstram que os melhores projetistas produzem estruturas mais rápidas, menores, mais simples, mais claras e com menos esforço."

Frederick P. Brooks

Princípio 4. ***As interfaces (tanto internas quanto externas) devem ser projetadas com cuidado.*** A forma como os dados fluem entre os componentes de um sistema tem muito a ver com a eficiência do processamento, com a propagação de erros e com a simplicidade do projeto. Uma interface bem elaborada facilita a integração e auxilia o responsável pelos testes quanto à validação das funções dos componentes.

Princípio 5. ***O projeto de interface do usuário deve ser voltado às necessidades do usuário final. Entretanto, em todo caso, deve enfatizar a facilidade de uso.*** A interface do usuário é a manifestação visível do software. Não importa quão sofisticadas sejam as funções internas, quão amplas sejam as estruturas de dados, quão bem projetada seja a arquitetura; um projeto de interface pobre leva à percepção de que um software é "ruim".

Princípio 6. ***O projeto no nível de componentes deve ser funcionalmente independente.*** Independência funcional é uma medida para a "mentalidade simplificada" de um componente de software. A funcionalidade entregue por um componente deve ser coesa — isto é, focarem uma, e somente uma, função ou subfunção.[6]

Princípio 7. ***Os componentes devem ser relacionados livremente tanto entre componentes quanto com o ambiente externo.*** O relacionamento é obtido de várias maneiras — via interface de componentes, por meio de mensagens, através de dados em geral. Na medida em que o nível de correlacionamento aumenta, a tendência para a propagação do erro também aumenta e a manutenção geral do software decresce. Portanto, o relacionamento entre componentes deve ser mantido tão baixo quanto possível.

Princípio 8. ***Representações de projetos (modelos) devem ser de fácil compreensão.*** A finalidade de projetos é transmitir informações aos desenvolvedores que farão a codificação, àqueles que irão testar o software e para outros que possam vir a dar manutenção futuramente. Se o projeto for de difícil compreensão, não servirá como meio de comunicação efetivo.

Princípio 9. ***O projeto deve ser desenvolvido iterativamente. A cada iteração, o projetista deve se esforçar para obter maior grau de simplicidade.*** Como todas as atividades criativas, a elaboração de um projeto ocorre de forma iterativa. As primeiras iterações são realizadas para refinar o projeto e corrigir erros, entretanto, as iterações finais devem dirigir esforços para tornar o projeto tão simples quanto possível.

Quando tais princípios são aplicados apropriadamente, elabora-se um projeto com fatores de qualidade tanto externos quanto internos [Mye78]. *Fatores externos de qualidade* são as propriedades que podem ser prontamente notadas pelos usuários (por exemplo, velocidade, confiabilidade, correção, usabilidade). Os *fatores internos de qualidade* são de extrema importância para os engenheiros de software. Conduzem a um projeto de alta qualidade do ponto de vista técnico. Para obter fatores de qualidade internos, o projetista deve entender sobre conceitos básicos de projeto (Capítulo 8).

---

6 Mais informações a respeito de coesão podem ser encontradas no Capítulo 8.

"Por muito tempo em minha vida, fui um observador em relação ao mundo do software, espiando furtivamente os códigos poluídos de outras pessoas. Ocasionalmente, encontro uma joia verdadeira, um programa bem estruturado, escrito em um estilo consistente, livre de gambiarras, desenvolvido de modo que cada componente seja simples, organizado e projetado de forma que o produto possa ser facilmente alterado."

David Parnas

4.3.4 Princípios de construção

A atividade de construção engloba um conjunto de tarefas de codificação e testes que conduzem ao software operacional pronto para ser entregue ao cliente e ao usuário final. Na atividade moderna de engenharia de software, a codificação pode ser: (1) a criação direta do código-fonte da linguagem de programação (por exemplo, Java), (2) a geração automática de código-fonte usando uma representação intermediária semelhante a um projeto do componente a ser construído ou então (3) a geração automática de código executável usando uma "linguagem de programação de quarta geração" (por exemplo, Visual C++).

O foco inicial dos testes é voltado para componentes, com frequência denominado *teste de unidade*. Outros níveis de testes incluem: (1) *teste de integração* (realizado à medida que o sistema é construído), (2) *teste de validação* que avalia se os requisitos foram atendidos para o sistema completo (ou incremento de software) e (3) *teste de aceitação conduzido* pelo cliente voltado para empregar todos os fatores e funções requisitados. A série de princípios e conceitos fundamentais a seguir é aplicável à codificação e aos testes:

Princípios de codificação. Os princípios que regem a codificação são intimamente alinhados com o estilo de programação, com as linguagens e com os métodos de programação. Entretanto, há uma série de princípios fundamentais que podem ser estabelecidos:

Princípios de preparação: ***Antes de escrever uma única linha de código, certifique-se de que***

- Compreendeu bem o problema a ser solucionado.
- Compreendeu bem os princípios e conceitos básicos sobre o projeto.
- Escolheu uma linguagem de programação adequada às necessidades do software a ser desenvolvido e ao ambiente em que ele irá operar.
- Selecionou um ambiente de programação que forneça ferramentas que tornarão seu trabalho mais fácil.
- Elaborou um conjunto de testes de unidades que serão aplicados assim que o componente codificado estiver completo.

Princípios de programação: ***Ao começar a escrever código***

- Restrinja seus algoritmos seguindo a prática de programação estruturada [Boh00].
- Considere o emprego de uso de programação pareada.
- Selecione estruturas de dados que venham ao encontro das do projeto.
- Domine a arquitetura de software e crie interfaces consistentes com ela.
- Mantenha a lógica condicional tão simples quanto possível.
- Crie "loops" agrupados de tal forma que testes sejam facilmente aplicáveis.
- Escolha denominações de variáveis significativas e obedeça a outros padrões de codificação locais.
- Faça uma documentação que seja autodocumentável.
- Crie uma disposição (layout) visual (por exemplo, recorte e linhas em branco) que auxilie a compreensão.

Princípios de validação: ***Após completar a primeira etapa de codificação, certifique-se de***

- Aplicar uma revisão de código quando for apropriado.
- Realizar testes de unidades e corrigir erros ainda não identificados.
- Refazer a codificação.

Mais livros foram escritos sobre programação (codificação) e sobre os princípios e os conceitos que a guiam do que qualquer outro tópico do processo de software. Livros sobre o tema incluem trabalhos antigos em estilo de programação [Ker78], construção prática de software [McC04],

Evite desenvolver um programa elegante que resolva o problema errado. Preste particular atenção ao primeiro princípio referente à preparação.

WebRef

Uma ampla variedade de links para padrões de codificação pode ser encontrada no site **www.literateprogramming.com/fpstyle.html**.

pérolas da programação [Ben99], a arte de programar [Knu98], elementos da programação pragmática [Hun99] e muitos, muitos outros assuntos. Um debate amplo sobre esses princípios e conceitos foge do escopo deste livro. Caso haja maior interesse, examine uma ou mais das referências indicadas.

Princípios de testes. Em um clássico sobre testes de software, Glen Myers [Mye79] estabelece uma série de regras que podem servir, bem como objetivos da atividade de testes:

Quais são os objetivos dos testes de software?

- Teste consiste em um processo de executar um programa com o intuito de encontrar um erro.
- Um bom pacote de testes é aquele em que há uma alta probabilidade de encontrar um erro ainda não descoberto.
- Um teste bem-sucedido é aquele que revela um novo erro.

AVISO

Em um contexto mais amplo de projeto de software, atente para iniciar "no geral", focalizando a arquitetura de software, e terminar "no particular", focalizando os componentes. Para testes, simplesmente reverta o foco e teste seu desenvolvimento.

Esses objetivos implicam uma mudança radical de ponto de vista para alguns desenvolvedores. Vão contra a visão comumente difundida de que um teste bem-sucedido é aquele em que nenhum erro é encontrado. Seu objetivo é o de projetar testes que descubram, sistematicamente, diferentes classes de erros, consumindo o mínimo de esforço e tempo.

Se testes forem conduzidos com êxito (de acordo com os objetivos declarados previamente), irão encontrar erros no software. Como benefício secundário, os testes demonstram que as funções do software estão funcionando de acordo com as especificações e que os requisitos relativos ao desempenho e ao comportamento parecem estar sendo atingidos. Os dados coletados durante os testes fornecem um bom indício da confiabilidade do software, assim como fornecem a indicação da qualidade do software como um todo. Entretanto, os testes não são capazes de mostrar a ausência de erros e defeitos; podendo apenas mostrar que erros e defeitos de software estão presentes. É importante manter isso em mente (do que negligenciá-lo) enquanto os testes estão sendo aplicados.

Davis [Dav95b] sugere um conjunto de princípios de testes[7] adaptados para este livro:

Princípio 1. ***Todos os testes devem estar alinhados com os requisitos do cliente.***[8]
O objetivo dos testes é desvendar erros. Constata-se que os efeitos mais críticos do ponto de vista do cliente são aqueles que conduzem a falhas no programa quanto a seus requisitos.

Princípio 2. ***Os testes devem ser planejados muito antes de ser iniciados.*** O planejamento dos testes (Capítulo 17) pode começar assim que o modelo de requisitos estiver completo. A definição detalhada dos pacotes de teste pode começar tão logo o modelo de projeto tenha sido solidificado. Portanto, todos os testes podem ser planejados e projetados antes que qualquer codificação tenha sido gerada.

Princípio 3. ***O princípio de Pareto se aplica a testes de software.*** Neste contexto o princípio de Pareto implica que 80% de todos os erros revelados durante testes provavelmente estarão alinhados a aproximadamente 20% de todos os componentes de programa. O problema, evidentemente, consiste em isolar os componentes suspeitos e testá-los por completo.

Princípio 4. ***Os testes devem começar "em particular" e progredir rumo ao teste "em geral".*** Os primeiros testes planejados e executados geralmente focam os componentes individuais. À medida que progridem, o foco muda para tentar encontrar erros em grupos de componentes integrados e, posteriormente, no sistema inteiro.

Princípio 5. ***Testes exaustivos são impossíveis.*** A quantidade de trocas de direções, mesmo para um programa de tamanho moderado, é excepcionalmente grande. Por essa

7 Apenas um pequeno subconjunto dos princípios de testes de David é citado aqui. Para maiores informações, veja [Dav95b].

8 Esse princípio refere-se a testes funcionais, por exemplo, testes que se concentram em requisitos. Os testes estruturais (testes que se concentram nos detalhes lógicos ou arquitetônicos) não podem referir-se a requisitos específicos diretamente.

razão, é impossível executar todas as rotas durante os testes. Sendo possível, no entanto, cobrir adequadamente a lógica do programa e garantir que todas as condições referentes ao projeto no nível de componentes sejam exercidas.

4.3.5 Princípios de disponibilização

Como observado anteriormente, na Parte 1 deste livro, a disponibilização envolve três ações: entrega, suporte e feedback. Pelo fato de os modernos modelos de processos de software serem, em sua natureza, evolucionários ou incrementais, a disponibilização não ocorre imediatamente, mas sim, em muitas vezes, quando o software segue para sua finalização. Cada ciclo de entrega propicia ao cliente e ao usuário um incremento de software operacional que fornece fatores e funções utilizáveis. Cada ciclo de suporte fornece assistência humana e documentação para todas as funções e fatores introduzidos durante todos os ciclos de disponibilização até o presente. Cada ciclo de feedback fornece à equipe de software importante roteiro que resulta em alteração de funções, elementos e abordagem adotados para o próximo incremento.

Certifique-se de que seu cliente saiba o que esperar antes da entrega de um incremento de software. Caso contrário, com certeza ele contará mais do que você poderá entregar.

A entrega para um incremento de software representa um marco importante para qualquer projeto de software. Uma série de princípios essenciais deve ser seguida enquanto a equipe se prepara para a entrega de um incremento:

Princípio 1. ***As expectativas dos clientes para o software devem ser gerenciadas.*** Muitas vezes, o cliente espera mais do que a equipe havia prometido entregar e, imediatamente, ocorre o desapontamento. Isso resulta em feedback não produtivo e arruína o moral da equipe. Em seu livro sobre gerenciamento de expectativas, Naomi Karten [Kar94] afirma: "O ponto de partida para administrar expectativas consiste em se tornar mais consciencioso sobre como e o que vai comunicar." Ela sugere que um engenheiro de software deve ser cauteloso em relação ao envio de mensagens conflituosas ao cliente (por exemplo, prometer mais do que pode entregar racionalmente no prazo estabelecido ou entregar mais do que o prometido para determinado incremento e, em seguida, menos do que prometera para o próximo).

Princípio 2. ***Um pacote de entrega completo deve ser auditorado e testado.*** Um CD-ROM ou outra mídia (inclusive downloads de Web) contendo todo o software executável, arquivos de dados de suporte, documentos de suporte e outras informações relevantes devem ser completamente checados e testados por meio de uma versão beta com os reais usuários. Todos os roteiros de instalação e outros itens operacionais devem ser rodados inteiramente em tantas configurações computacionais quanto forem possíveis (isto é, hardware, sistemas operacionais, dispositivos periféricos, disposições de rede).

Princípio 3. Deve-se estabelecer uma estrutura de suporte antes da entrega do software. Um usuário final conta com recebimento de informações acuradas e com reponsabilidade caso surja um problema. Se o suporte for local, ou, pior ainda, inexistente, imediatamente o cliente ficará insatisfeito. O suporte deve ser planejado, e seus materiais devem estar preparados, e mecanismos para manutenção de registros apropriados devem estar determinados para que a equipe de software possa oferecer uma avaliação de qualidade das formas de suporte solicitadas.

Princípio 4. ***Material adequado referente a instruções deve ser fornecido aos usuários finais.*** A equipe de software deve entregar mais do que o software em si. Auxílio em treinamento de forma adequada (se solicitado) deve ser desenvolvido; orientações quanto a problemas inesperados devem ser oferecidas; quando necessário, é importante editar uma descrição acerca de "diferenças existentes no incremento de software".[9]

Princípio 5. ***Software com bugs (erros), primeiro, deve ser corrigido, e, depois, entregue.*** Sob a pressão relativa a prazo, muitas empresas de software entregam incrementos

9 Durante a atividade de comunicação, a equipe de software deve determinar quais materiais de apoio os usuários desejam.

de baixa qualidade, notificando o cliente que os bugs "serão corrigidos na próxima versão". Isso é um erro. Há um ditado no mercado de software: "Os clientes esquecerão a entrega de um produto de alta qualidade em poucos dias, mas jamais esquecerão os problemas causados por um produto de baixa qualidade. O software os faz lembrar disso todos os dias".

O software entregue propicia benefício ao usuário final, mas este também fornece feedback proveitoso para a equipe de software. À medida que um incremento é colocado em uso, os usuários finais devem ser encorajados a tecer comentários acerca das características e funções, facilidade de uso, confiabilidade e quaisquer outras características apropriadas.

4.4 Resumo

A prática de engenharia de software envolve princípios, conceitos, métodos e ferramentas aplicados por engenheiros da área ao longo de todo o processo de desenvolvimento. Cada projeto de engenharia de software é diferente. Ainda assim, uma gama de princípios genéricos se aplica ao processo como um todo e à prática de cada atividade metodológica independentemente do projeto ou do produto.

Um conjunto de princípios essenciais auxilia na aplicação de um processo de software significativo e na execução de métodos efetivos de engenharia de software. Quanto ao processo, os princípios essenciais estabelecem uma base filosófica que orienta a equipe durante essa fase de desenvolvimento. Quanto ao nível relativo à prática, os princípios estabelecem uma série de valores e regras que servem como guia ao se analisar um problema, projetar uma solução, implementar e testar uma solução e, por fim, diponibilizar o software para a sua comunidade de usuários.

Princípios de comunicação enfatizam a necessidade de reduzir ruído e aumentar a dimensão conforme o diálogo entre o desenvolvedor e o cliente progride. Ambas as partes devem colaborar para que ocorra a melhor comunicação.

Princípios de planejamento proporcionam roteiros para a construção do melhor mapa para a jornada rumo a um sistema ou produto completo. O plano pode ser projetado para um único incremento de software ou pode ser definido para o projeto inteiro. Independentemente da situação, deve indicar o que será feito, quem o fará e quando o trabalho estará completo.

Modelagem abrange tanto análise quanto projeto, descrevendo representações do software que se tornam progressivamente mais detalhadas. O objetivo dos modelos é solidificar a compreensão do trabalho a ser feito e providenciar orientação técnica aos implementadores do software. Os princípios de modelagem servem como infraestrutura para os métodos e para a notação utilizada para criar representações do software.

Construção incorpora um ciclo de codificação e testes no qual o código-fonte para um componente é gerado e testado. Os princípios de codificação definem ações genéricas que devem ocorrer antes da codificação ser feita, enquanto está sendo criada e após estar completa. Embora haja muitos princípios de testes, apenas um é dominante: teste consiste em um processo de executação de um programa com o intuito de encontrar um erro.

Disponibilização ocorre na medida em que cada incremento de software é apresentado ao cliente e engloba a entrega, o suporte e o feedback. Os princípios fundamentais para a entrega consideram o gerenciamento das expectativas dos clientes e o fornecimento ao cliente de informações de suporte apropriadas sobre o software. O suporte exige preparação antecipada. Feedback permite ao cliente sugerir mudanças que tenham valor agregado e fornecer ao desenvolvedor informações para o próximo ciclo de engenharia de software.

Problemas e Pontos a Ponderar

4.1. Uma vez que o foco em qualidade demanda recursos e tempo, é possível ser ágil e ainda assim manter o foco em qualidade?

4.2. Dos oito princípios básicos que orientam um processo (discutido na Seção 4.2.1), qual você acredita ser o mais importante?

4.3. Descreva o conceito de *separação por interesses* com suas próprias palavras.

4.4. Um importante princípio de comunicação afirma "prepare-se antes de se comunicar". Como essa preparação se manifesta no trabalho prévio que você realiza? Quais produtos de trabalho podem resultar da preparação antecipada?

4.5. Pesquise sobre "facilitação" para a atividade de comunicação (use as referências fornecidas ou outras) e prepare um conjunto de passos focados somente em facilitação.

4.6. Em que a comunicação ágil difere da tradicional em engenharia de software? Em que ela é similar?

4.7. Por que é necessário "seguir em fente"?

4.8. Pesquise sobre "negociação" para a atividade de comunicação e prepare uma série de etapas concentrando-se apenas na negociação.

4.9. Descreva o que significa "particularidade"no contexto do cronograma de um projeto.

4.10. Por que os modelos são importantes no trabalho de engenharia de software? São eles sempre necessários? Existem qualificadores para sua resposta sobre necessidade?

4.11. Quais são os três "domínios" considerados durante a modelagem de requisitos?

4.12. Tente acrescentar mais um princípio àqueles determinados para a codificação na Seção 4.3.4.

4.13. Em que consiste um teste bem-sucedido?

4.14. Você concorda ou discorda com a seguinte afirmação: "Uma vez que vários incrementos são entregues ao cliente, por que se preocupar com a qualidade nos incrementos iniciais — pode-se corrigir os problemas em iterações posteriores". Justifique sua resposta.

4.15. Por que o feedback é importante para uma equipe de software?

Leituras e Fontes de Informação Complementares

A comunicação com o cliente é uma atividade crítica na engenharia de software, embora poucos profissionais invistam em sua leitura. Withall (*Software Requirements Patterns*, Microsoft Press, 2007) apresenta uma gama de padrões úteis que tratam de problemas de comunicação. Sutliff (*User-Centred Requirements Engineering*, Springer, 2002) se concentra muito nos desafios relacionados com a comunicação. Livros como os de Weigers (*Software Requirements*, 2. ed., Microsoft Press, 2003), Pardee (*To Satisfy and Delight Your Customer*, Dorset House, 1996) e Karten [Kar94] dão uma excelente visão sobre métodos para interação efetiva com os clientes. Embora a obra não se concentre em software, Hooks e Farry (*Customer Centered Products*, American Management Association, 2000) apresentam úteis diretrizes genéricas para a comunicação com os clientes. Young (*Effective Requirements Practices*, Addison-Wesley, 2001) enfatiza uma "equipe conjunta" entre clientes e desenvolvedores que desenvolvem requisitos de forma colaborativa. Somerville e Kotonya (*Requirements Engineering: Processes and Techniques*, Wiley, 1998) discutem técnicas e conceitos de "suscitação", bem como outros princípios de engenharia de requisitos.

Conceitos e princípios de comunicação e planejamento são considerados em vários livros de gerenciamento de projetos. Entre eles podemos citar: Bechtold (*Essentials of Software Project Management*, 2. ed., Management Concepts, 2007), Wysocki (*Effective Project Management: Traditional, Adaptive, Extreme*, 4. ed., Wiley, 2006), Leach (*Lean Project Management: Eight Principless for Success*, BookSurge Publishing, 2006), Hughes (*Software Project Management*, McGraw-Hill, 2005) e Stellman e Greene (*Applied Software Project Management*, O'Reilly Media, Inc., 2005).

Davis [Dav95] compilou um excelente conjunto de princípios de engenharia de software. Além disso, praticamente todo livro sobre engenharia de software contém uma discussão

proveitosa sobre conceitos e princípios para análise, projeto e testes. Entre os livros mais largamente usados (além deste livro, é claro!), temos:

Abran, A. e J. Moore, *SWEBOK: Guide to the Software Engineering Body of Knowledge*, IEEE, 2002.

Christensen, M. e R. Thayer, *A Project Manager's Guide to Software Engineering Best Practices*, IEEE-CS Press (Wiley), 2002.

Jalote, P., *An Integrated Approach to Software Engineering*, Springer, 2006.

Pfleeger, S., *Software Engineering: Theory and Practice*, 3. ed., Prentice-Hall, 2005.

Schach, S., *Engenharia de Software: Os Paradigmas Clássico e Orientado a Objetos*, McGraw-Hill, 7. ed., 2008.

Sommerville, I., *Software Engineering*, 8. ed., Addison-Wesley, 2006.

Esses livros também apresentam uma discussão detalhada sobre princípios de modelagem e construção.

Princípios de modelagem são considerados em muitos textos dedicados à análise de requisitos e/ou projeto de software. Livros como os de Lieberman (*The Art of Software Modeling*, Auerbach, 2007), Rosenberg e Stephens (*Use Case Driven Object Modeling with UML: Theory and Practice*, Apress, 2007), Roques (*UML in Practice*, Wiley, 2004), Penker e Eriksson (*Business Modeling with UML: Business Patterns at Work*, Wiley, 2001) discutem os princípios e métodos de modelagem.

A obra de Norman (*The Design of Everyday Things*, Currency/Doubleday, 1990) é uma leitura obrigatória para todo engenheiro de software que pretenda realizar trabalhos de projeto. Winograd e seus colegas (*Bringing Design to Software*, Addison-Wesley, 1996) editaram um excelente conjunto de artigos que tratam das questões práticas no projeto de software. Constantine e Lockwood (*Software for Use*, Addison-Wesley, 1999) apresentam os conceitos associados ao "projeto centrado no usuário". Tognazzini (*Tog on Software Design*, Addison-Wesley, 1995) traz uma discussão filosófica proveitosa sobre a natureza do projeto e o impacto das decisões na qualidade e na habilidade de a equipe produzir software que forneça grande valor a seu cliente. Stahl e seus colegas (*Model-Driven Software Development: Technology, Engineering*, Wiley, 2006) discutem os princípios do desenvolvimento dirigido por modelos.

Centenas de livros tratam um ou mais elementos da atividade de construção. Kernighan e Plauger [Ker78] escreveram um clássico sobre estilo de programação, McConnell [McC93] apresenta diretrizes pragmáticas para a construção prática de software, Bentley [Ben99] sugere uma ampla gama de pérolas da programação, Knuth [Knu99] escreveu uma série clássica de três volumes sobre a arte de programar e Hunt [Hun99] sugere diretrizes de programação pragmáticas.

Myers e seus colegas (*The Art of Software Testing*, 2. ed., Wiley, 2004) fizeram uma revisão importante de seu texto clássico e discutem vários princípios de testes importantes. Livros como os de Perry (*Effective Methods for Software Testing*, 3. ed., Wiley, 2006), Whittaker (*How to Break Software*, Addison-Wesley, 2002), Kaner e seus colegas (*Lessons Learned in Software Testing*, Wiley, 2001) e Marick (*The Craft of Software Testing*, Prentice-Hall, 1997) introduzem, cada um deles, importantes conceitos e princípios para testes e muita orientação pragmática.

Uma ampla gama de fontes de informação sobre a prática da engenharia de software se encontra à disposição na Internet. Uma lista atualizada de referências na Web, relevantes à prática da engenharia de software, pode ser encontrada no site **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**.