

Capítulo 1

O que é qualidade?

A qualidade é relativa. O que é qualidade para uma pessoa pode ser falta de qualidade para outra.

– G. Weinberg

A idéia de qualidade é aparentemente intuitiva; contudo, quando examinado mais longamente, o conceito se revela complexo. Definir um conceito de qualidade para estabelecer objetivos é, assim, uma tarefa menos trivial do que aparenta a princípio.

Este capítulo enfatiza como a noção de qualidade pode ser relativa e introduz as dificuldades básicas relacionadas ao tratamento desse assunto.

1.1 História

Embora o controle de qualidade e o uso de padrões como ISO sejam algo que tenha atraído bastante atenção nas últimas décadas, historicamente o assunto é muito antigo. Existem relatos históricos segundo os quais há mais de quatro mil anos os egípcios estabeleceram um padrão de medida de comprimento: o cúbito. Essa medida correspondia ao comprimento do braço do faraó reinante. Curiosamente, a troca de faraó significava que a medida deveria ser atualizada. Todas as construções deviam ser realizadas utilizando o cúbito como unidade de medida. Para isso eram empregados bastões cortados no comprimento certo e periodicamente – a cada lua cheia – o responsável por uma construção devia comparar o padrão que estava sendo utilizado com o padrão real. Se isso não fosse feito e houvesse um erro de medição, o responsável poderia ser punido com a morte [Juran e Gryna, 1988]. O resultado da preocupação com rigor é evidente na construção das pirâmides, em que os egípcios teriam obtido precisões da ordem de 0,05%¹.

1 Site Internet da EOS (Egyptian Organization for Standardization and Quality Control) (2005).

A história da qualidade prosseguiu com inúmeros exemplos de resultados extraordinários: os grandes templos construídos na Grécia e Roma antigas, os feitos de navegação no século XVI, as catedrais medievais. Em todas essas realizações, não se dispunha de instrumentos de precisão ou técnicas sofisticadas. Na França, os construtores de catedrais utilizavam simples compassos e cordas com nós a intervalos regulares para desenhar e construir edifícios [Vincent, 2004].

Em geral, espera-se que obter mais precisão exija mais recursos ou mais tecnologia. Assim, a regulação da carburação de um motor de um veículo moderno não pode ser feita como no passado, quando, com uma lâmpada, um mecânico conseguia “acertar o ponto” do distribuidor. O exemplo dos antigos egípcios nos faz pensar uma questão curiosa: como teriam sido as pirâmides se, na época, os trabalhadores dispusessem de medidores a laser?

Como veremos ao longo do livro, a qualidade de software ainda depende principalmente do correto emprego de boas metodologias pelos desenvolvedores. Embora eles sejam apoiados por várias ferramentas, ainda restam problemas sérios sem tal suporte. As técnicas para verificação automática, dentre as quais a interpretação abstrata [Cousot, 2000] é um excelente exemplo, ainda são incipientes.

Um grande marco na história da qualidade foi, com certeza, a revolução industrial. Esse período também é associado a profundas mudanças econômicas e sociais, como o início da automação e o surgimento do consumo de massa. Durante essa época de efervescência, milhares de novas empresas surgiram. A criação de diversas indústrias levou rapidamente à concorrência entre elas, o que, por sua vez, desencadeou um processo de melhoria contínua que perdura até hoje. O aumento da eficiência tornou-se uma condição imprescindível para garantir a sobrevivência. Uma ilustração clara disso é a extinção de centenas de fábricas de automóveis nos Estados Unidos: no início do século XX, esse país contava com cerca de 1.800 fabricantes diferentes.

Na década de 1920 surgiu o controle estatístico de produção. Nas fábricas que produziam grande quantidade de itens tornou-se impossível garantir a qualidade individual de cada peça, ao contrário do que se fazia (e ainda se faz) no trabalho artesanal. Dessa forma foi preciso desenvolver mecanismos diferentes e a resposta veio da estatística. Um dos primeiros trabalhos associados ao assunto é o livro publicado por Walter Shewhart em 1931, *Economic Control of Quality of Manufactured Product*. Shewhart, dos Bell Laboratories, teria introduzido os diagramas de controle (*control charts* ou *Shewhart chart*). A Figura 1.1 apresenta um exemplo desse tipo de diagrama.

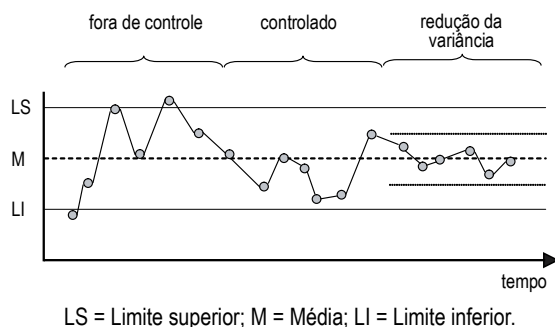


Figura 1.1 – Diagrama de Shewhart.

Para entender o diagrama, vamos supor que o problema tratado consista em controlar o diâmetro de parafusos. O diagrama apresenta três linhas verticais que definem o valor médio e os limites superior e inferior máximos tolerados. Cada ponto no diagrama representa o valor de uma amostra, isto é, um parafuso recolhido aleatoriamente na saída da fábrica. O gráfico permite verificar se os processos estão sendo bem controlados e se há tendências de melhora ou piora da qualidade. Por exemplo, se a variância nas medidas diminui, isso indica uma melhora da fabricação, enquanto um aumento pode indicar um problema como desgaste das máquinas. Outro exemplo de análise possível é detectar desvios: se uma série de medidas está acima do valor médio, isso pode indicar necessidade de calibragem. Para um processo seguindo uma distribuição estatística normal², é pouco provável que várias peças estejam todas com dimensões acima da média.

Na década de 1940 surgiram vários organismos ligados à qualidade; por exemplo, a ASQC (American Society for Quality Control), a ABNT (Associação Brasileira de Normas Técnicas) e, ainda, a ISO (International Standardization Organization). A Segunda Guerra Mundial também contribuiu com o processo, quando as técnicas de manufatura foram aprimoradas para fabricação de material bélico.

Na década de 1940 o Japão destacou-se como um importante pólo no assunto e contribuiu com diversas novas ferramentas: o método de Taguchi para projeto experimental, a metodologia 5S ou, ainda, os diagramas de causa e efeito de Ishikawa, também conhecidos como diagramas espinha de peixe.

A Figura 1.2 apresenta um diagrama de Ishikawa típico. A técnica é utilizada para identificar as causas de um problema e, de preferência, deve ser utilizada durante uma reunião em que todos os envolvidos discutam livremente, sem sanções. Um problema específico que afeta a empresa é representado no eixo central do diagrama. Em seguida, linhas diagonais são incluídas contendo elementos que fazem parte

² No Capítulo 3 são apresentadas algumas fórmulas básicas e úteis em estatística.

do cenário – como trabalhadores e máquinas. Tais elementos são chamados de “categorias”. Depois, para cada categoria procura-se identificar fatores (causas) que possam contribuir para aumentar ou reduzir o problema (efeitos). Dependendo do tipo de indústria, sugere-se usar diferentes categorias:

- **para manufatura:** mão-de-obra, métodos, materiais e máquinas;
- **para serviços e administração:** equipamentos, procedimentos, políticas e pessoas.

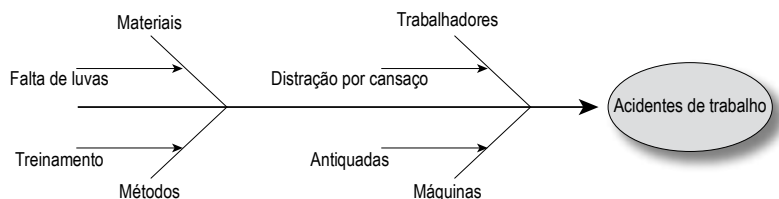


Figura 1.2 – Diagrama de Ishikawa.

No pós-guerra o impulso recebido pelas indústrias se manteve. Os computadores digitais já estavam em uso nessa época, embora estivessem restritos sobretudo a meios militares e acadêmicos. Alguns anos mais tarde, quando as máquinas se tornaram mais acessíveis e um maior número de pessoas as utilizava, a qualidade dos softwares começou a se mostrar um objetivo importante.

Nas décadas de 1960 e 1970 ocorreram mudanças tecnológicas importantes que afetaram a construção dos computadores e, em conseqüência, a de software. O período é conhecido como “crise de software” e as repercussões são sentidas até hoje [Pressman, 2002].

1.2 Uma crise de mais de trinta anos

Um dos fatores que exerce influência negativa sobre a qualidade de um projeto é a complexidade, que está associada a uma característica bastante simples: o tamanho das especificações.

Construir um prédio de 10 andares implica tratar um número de problemas muito maior do que os existentes em uma simples residência: a diferença entre as duas construções, é claro, está longe de ser resolvida com um número de tijolos maior. Em programas de computador, o problema de complexidade e tamanho é ainda mais grave, em razão das interações entre os diversos componentes do sistema.

Por volta da década de 1950 acreditava-se em uma relação chamada lei de Grosch [Pick et al., 1986]: o desempenho de um computador seria proporcional ao quadrado de seu preço. Nessa época – e de acordo com essa lei – uma idéia interessante era reunir um grupo de usuários para adquirir um computador de grande porte (mainframe). Era comum também alugar uma máquina diretamente do fabricante. Problemas maiores significavam apenas a necessidade de máquinas maiores.

E como eram as máquinas “grandes” nessa época?

Até a década de 1970 ainda eram utilizadas as memórias de núcleo (*core memory*): caras, lentas e consumidoras de muita energia. A memória semicondutora só foi criada em 1966 (por Robert H. Dennard, na IBM) e fabricada, pela primeira vez, pela Intel, em 1970. Logo após, em 1971, surgiu o primeiro microprocessador em silício: o Intel 4004, uma CPU de 4 bits utilizando menos de 3 mil transístores, mas com tanto poder de processamento quanto o ENIAC, que possuía 18 mil válvulas. Assim, um computador “grande”, em 1960, era uma máquina ocupando uma sala de dezenas de metros quadrados!

A mudança tecnológica teve um efeito dramático na produção de software. Num breve período de tempo, os recursos de hardware aumentaram muito e permitiram que produtos mais complexos fossem criados. Traçando um paralelo, seria como se os engenheiros civis, depois de anos construindo apenas casas ou pequenos prédios de 2 ou 3 andares, se vissem repentinamente com a tarefa de construir grandes arranha-céus [Dijkstra, 1972]:

A maior causa da crise do software é que as máquinas tornaram-se várias ordens de magnitude mais potentes! Em termos diretos, enquanto não havia máquinas, programar não era um problema; quando tivemos computadores fracos, isso se tornou um problema pequeno e agora que temos computadores gigantesco, programar tornou-se um problema gigantesco.

Mas a situação ainda era agravada por outro motivo: os primeiros programadores não possuíam ferramentas como dispomos hoje. A tarefa que enfrentavam poderia ser comparada, em certos aspectos, a erguer prédios empilhando mais e mais tijolos. Embora a noção de ciclo de vida já houvesse sido esboçada [Naur e Randell, 1968], não havia técnicas consagradas de trabalho. Não havia escolas ou sequer a profissão de programador; as pessoas aprendiam e exerciam essa atividade de maneira empírica [Dijkstra, 1972]: "... em 1957, casei-me e os ritos holandeses requerem declarar a profissão; declarei ser um programador. Mas as autoridades municipais de Amsterdã não aceitaram, com base em que não havia tal profissão".

Provavelmente a primeira vez em que se utilizou o termo “Engenharia de Software” foi em uma conferência com esse nome, realizada em 1968, na Alemanha. A conferência foi realizada por uma entidade que, a rigor, não possuía nenhuma ligação com a área: o Comitê de Ciência da NATO (North Atlantic Treaty Organisation – Organização do Tratado do Atlântico Norte). Curiosamente já havia instituições relacionadas com informática: a primeira delas foi a ACM (Association for Computing Machinery), criada em 1947 e que edita uma revista científica, *Communications of the ACM*, desde 1957.

Hoje, mais de trinta anos depois, quais são os problemas enfrentados na construção e utilização de software? Ao lermos o relatório da conferência da NATO de 1968 e outros documentos produzidos na década de 1970, fazemos uma descoberta assustadora: os problemas são os mesmos que encontramos atualmente.

Façamos uma pequena lista:

- cronogramas não observados;
- projetos com tantas dificuldades que são abandonados;
- módulos que não operam corretamente quando combinados;
- programas que não fazem exatamente o que era esperado;
- programas tão difíceis de usar que são descartados;
- programas que simplesmente param de funcionar.

Os erros parecem estar por toda a parte, como uma epidemia que não conseguimos controlar depois de décadas de trabalho e pesquisas. O último exemplo e certamente mais conhecido é o bug do milênio, quando foi predito o apocalipse da sociedade da informação: aviões cairiam, contas bancárias seriam zeradas, equipamentos militares perderiam o controle e bombas seriam disparadas. Embora seja verdade que poucos problemas de fato aconteceram, também é verdade que o erro realmente existia em muitos sistemas. O assunto será provavelmente lembrado no ano de 2100 como uma boa piada. Hoje em dia, contudo, não há motivos para rir, pois a pergunta permanece: não somos capazes de produzir software de qualidade? Essa questão não é simples de ser respondida, mas podemos aventar algumas razões.

O aspecto não repetitivo do desenvolvimento de software torna essa atividade difícil e, sobretudo, em boa medida imprevisível. Apenas uma pequena parcela da construção de software corresponde a atividades que poderíamos chamar de “montagem”. Quando se constrói uma rodovia ou, por exemplo, uma ponte, os processos de cálculo de estruturas, correção de inclinações, preparação do terreno e pavimentação são conhecidos. Algumas vezes ocorrem surpresas, como descobrir

que o solo de uma seção da estrada não correspondia ao que era imaginado; mas em geral, as diversas etapas são cumpridas sempre da mesma maneira. É durante o projeto dessa ponte que há mais questões em aberto e soluções de engenharia devem ser desenvolvidas. Assim, por exemplo, antes do projeto não se conhece o traçado que será mais confortável, seguro e econômico para os futuros usuários. O problema é comentado na Figura 1.3.

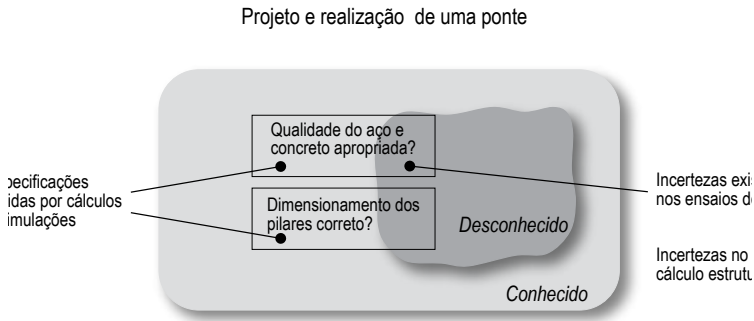


Figura 1.3 – Zonas de sombra em um projeto de engenharia.

Uma vez terminado o projeto, contudo, existem respostas para a maior parte das questões que dizem respeito à realização. Os carros que irão trafegar não mudarão de largura nem de comprimento repentinamente; a quantidade de veículos e, portanto, o peso máximo a ser suportado são calculados com boa precisão. Fatores como vento podem ser superdimensionados para garantir margens de segurança. Dificilmente um pilar deverá ser deslocado cinco centímetros para a direita depois que a construção já foi iniciada!

Comparativamente, quando estamos tratando de software essa zona de sombra que envolve fatores desconhecidos é bem mais abrangente. Quem já trabalhou no desenvolvimento de software sabe como é comum resolver problemas como encurtar ou alongar uma ponte, “apenas alguns centímetros” alguns dias antes de sua inauguração. A Figura 1.4 ilustra alguns aspectos.

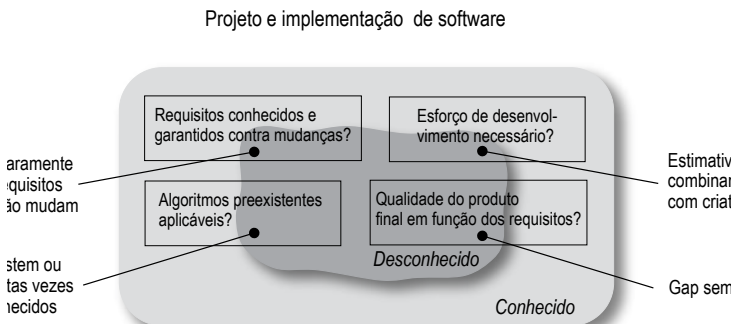


Figura 1.4 – Zonas de sombra em projeto de software.

As dificuldades em informática começam durante as etapas iniciais de um projeto: delimitar o escopo de um sistema está longe de ser uma tarefa trivial. A volatilidade dos requisitos (Capítulo 9) é um das maiores causas de insucesso de projetos de software. Uma mudança nas necessidades declaradas por um usuário pode repercutir em vários elementos da estrutura do programa. Mais tarde, embora a solução tenha sido projetada, ainda não se conhecem de antemão e em detalhes os algoritmos que efetivamente resolvem o problema. Em conseqüência, comumente se considera que é bastante difícil prever como será o programa acabado – apesar de a estrutura toda já ter sido desenhada.

Não bastassem as dificuldades inerentes a esse caráter mutável ou volátil do software, as pessoas que trabalham com ele contribuem bastante para piorar os problemas. O trabalho intelectual que é necessário à construção de programas pode, ao mesmo tempo, influenciar na existência de obstáculos ao sucesso de um projeto. Um exemplo de dificuldade é tratar os aspectos não técnicos que vêm à tona quando se realizam revisões de projetos [Yourdon, 1989]:

Mas você não pode remover o elemento humano de uma revisão. O autor, cujo “trabalho de arte” está sendo revisto, possui sentimentos e emoções humanas. E os revisores também possuem seus próprios sentimentos.

Conciliar a necessidade de disciplina – fundamental para garantir uma certa previsibilidade de resultados – com o caráter relativamente aleatório da criação de soluções talvez seja um dos maiores desafios. As metodologias têm sido desenvolvidas, testadas e adaptadas há décadas, buscando reduzir a um mínimo a necessidade de “inventar soluções”. Em grande parte, as metodologias têm caráter pedagógico: mostram o que é preciso fazer para conduzir um projeto, quais procedimentos adotar e como realizá-los, como trabalhar em equipes de maneira a evitar a veiculação de informações dúbias etc.

Além de metodologias, outro aspecto que contribui para combater o problema é o desenvolvimento de tecnologias e ferramentas. Existem várias tarefas que podem ser automatizadas, diminuindo a carga de trabalho das pessoas e, ao mesmo tempo, garantindo uniformidade: se é uma ferramenta que executa a tarefa, há menos chances de o resultado ser diferente porque diversas pessoas a utilizaram.

Mas a discussão sobre problemas de software não estará completa sem abordar o assunto que é o foco deste livro: a qualidade. Os métodos e ferramentas de engenharia de software servem, entre outras coisas, ao propósito de garantir, ou pelo menos facilitar, a obtenção do objetivo de ter qualidade nos programas. E qualidade é um conceito que, do ponto de vista da engenharia, é bastante complexo.

Sem definir precisamente qual é esse objetivo a atingir, o uso de todo arsenal de engenharia de software de que dispomos pode se revelar menos eficaz. Este é um dos objetivos principais deste livro: auxiliar a definir o alvo a ser atingido e discutir como aplicar a engenharia de software para aproximar-se o máximo possível do resultado desejado.

1.3 Qualidade e requisitos

Uma das primeiras questões a responder quando o assunto é qualidade é como julgá-la. Por exemplo: se estamos diante de produtos alternativos, como escolher o melhor? Esse problema de julgamento acontece com qualquer pessoa cotidianamente, quando se consomem itens como roupas, música, comida ou filmes. Mas curiosamente, apesar da frequência com que avaliamos os objetos à nossa volta, é muito difícil obter consenso a respeito da qualidade de um produto. Isso se traduz, por exemplo, no fato de existir uma profusão de marcas de eletrodomésticos e haver clientes felizes adquirindo aparelhos de marcas diferentes.

Uma escolha torna-se mais clara quando se estabelecem critérios que sirvam para julgar um produto. Em algumas situações, tais critérios são relativamente simples de identificar e estabelecer. Por exemplo: em domínios como engenharia elétrica ou mecânica, as informações necessárias são obtidas em função da finalidade a que se destina um determinado produto. Para dispositivos simples, como um fusível ou uma engrenagem, não é difícil enumerar algumas características que provavelmente são relevantes: ponto de fusão, condutância térmica, resistência a cisalhamento ou dimensões físicas. Passando para objetos mais complexos, como um transistor ou um amortecedor, a complexidade e quantidade de requisitos tendem a aumentar. Finalmente, quando se consideram sistemas compostos de subsistemas, é natural que a especificação de características seja realizada também de maneira hierárquica. Assim, a fonte de alimentação e o disco rígido de um computador possuem cada um sua própria série de especificações técnicas, o mesmo ocorrendo com o motor ou carburador de um carro.

Contudo, essas especificações garantem a qualidade?

O problema de definir e avaliar qualidade ainda não está completamente resolvido pelos requisitos. Todavia, um grande passo da solução consiste em ligar os dois conceitos:

$$\textit{qualidade} = f(\textit{requisitos})$$

Isto nos leva à famosa definição de Crosby [1992]: "A qualidade é conformidade aos requisitos". Essa definição é interessante, pois deixa explícito o fato de que é preciso um ponto de referência para julgar um produto. Traz embutida a idéia de como efetuar esse julgamento e, por fim, mostra como o processo todo pode ser documentado, analisado e os resultados transmitidos a outras pessoas.

Há, contudo, três fatos que perturbam essa definição, os quais é preciso conhecer para poder aplicá-la corretamente.

Em primeiro lugar, a definição nos deixa com a tarefa de definir o que é conformidade. Em alguns casos, isso se traduz em uma decisão booleana: uma lâmpada de 60W não é uma lâmpada de 100W. Mas, em geral, há poucos requisitos que possam ser tratados dessa maneira. O que se faz, então, é especificar margens de precisão: uma lâmpada que consome 59,9W é melhor que outra consumindo 60,1W. Esse exemplo nos leva naturalmente a considerar que possam existir intensidades ou graus de qualidade. Podemos escrever isso assim:

$$\begin{aligned} \text{qualidade} &= f(\text{observado}, \text{especificado}) \\ &= \|\text{observado} - \text{especificado}\| \end{aligned}$$

ou seja, a qualidade de um produto é dada pela distância entre as características observadas e as características que foram especificadas para sua construção.

A fórmula mostra que o problema ainda não está resolvido: precisamos definir uma medida de distância (indicada pelo símbolo $\|$ na expressão). É evidente que quanto mais longe estivermos das especificações, pior será o produto; entretanto, isto não indica o que fazer quando comparamos dois produtos diferentes em relação a uma longa lista de características. A lâmpada que consome 60,1W talvez seja mais brilhante que a de 59,9W; além disso, é possível que aqueça menos, sendo mais eficiente.

O segundo ponto diz respeito à realização da observação do produto. Como sabemos que a lâmpada consome exatamente 60,1W? Não seriam talvez 60,2W? Existem várias fontes de erro que podem corromper os dados utilizados para caracterizar um produto. Um exemplo disso, em informática, são os efeitos de memória cache no desempenho medido de computadores (*benchmarks*). O erro de observação pode ser representado assim:

$$\text{qualidade} = \|\text{observado} - \text{especificado} + \varepsilon\|$$

onde ε representa um erro de medição que não podemos controlar.

Por fim, o terceiro ponto a considerar é o papel de diferentes clientes em um mesmo projeto. Uma ótima exposição do assunto é feita por Weinberg [1994]: os requisitos foram definidos por alguém, logo a qualidade depende das escolhas que alguém efetuou. Segundo [Sommerville, 2003]:

Diferentes *stakeholders* têm em mente diferentes requisitos e podem expressá-los de maneiras distintas. Os engenheiros de requisitos precisam descobrir todas as possíveis fontes de requisitos e encontrar pontos comuns e os conflitos.

Isto é, com frequência, um problema não muito simples de resolver. Projetos grandes, envolvendo muitas funções e pessoas diferentes (diversos tipos de operadores, usuários das informações, gerentes etc. – coletivamente chamados de *stakeholders*), provavelmente têm mais chances de conter requisitos conflitantes. Isso ocorre por falta de consenso em relação a como certas tarefas devem ser feitas, ou mesmo para decidir quais tarefas são mais importantes implementar ou não. Em cenários desse tipo, pode existir uma deficiência de diálogo que antecede o projeto do software. Os desenvolvedores do produto podem se encontrar face a um duplo problema: resolver ou, pelo menos, minimizar o problema organizacional do cliente que contrata o desenvolvimento do software e, depois, obter uma especificação coerente que atenda aos interessados.

1.4 Papel da subjetividade

A qualidade de um produto tem um propósito: satisfazer o cliente. Esse objetivo implica tratar um domínio, em geral, bastante nebuloso. Para compreender o motivo, considere novamente o caso de uma pessoa que deve adquirir um produto comum no mercado. Ninguém compra uma camisa pensando nas propriedades mecânicas do tecido com o qual ela foi fabricada: “Que bela camisa! Pode resistir a 10 kg de tração!”. Em vez disso, são fatores muito difíceis de medir que, em geral, terão maior peso na decisão.

Tomemos outro exemplo: a especificação de um automóvel de qualidade. Essa especificação consistiria em uma lista de itens que se deseja que o veículo tenha. Por exemplo: direção hidráulica, teto solar, freios ABS, espaço para bagagens, conforto e potência. A maioria das pessoas concordaria, vendo essa lista, que o veículo sendo especificado é certamente um produto de bastante qualidade. Entretanto, para o engenheiro preocupado com a construção de um produto, essa visão é superficial: não se sabe qual a potência do carro ou o que caracteriza o conforto desejado. Além disso, em vários aspectos a especificação é incompleta. Dois exemplos podem demonstrar a razão disso.

Primeiro, considere um comprador que não tem recursos financeiros para adquirir um carro com essa lista de itens. Esse comprador irá procurar um veículo, realizar uma avaliação de possibilidades e, para um dado orçamento, voltar para casa com o carro de melhor qualidade que encontrou. A qualidade não pode ser uma entidade abstrata, mas um objetivo concreto com o qual o fabricante, comerciantes e compradores devem tratar. Por causa disso, o custo é um fator integrante de um modelo de qualidade.

Segundo ponto, considere que alguém deseja comprar outro veículo; mas o problema a ser resolvido é deslocar-se vez por outra dentro da cidade e, nos finais de semana, transportar um saco de ração de 200 kg para os cães que cuidam do sítio da família. Nesse caso é evidente que teto solar ou mesmo conforto tenham menor importância. Como foi ressaltado antes, os requisitos foram especificados por uma pessoa: logo, é preciso saber claramente como ela pensa. Mais do que isso, é preciso saber como cada pessoa envolvida no projeto – cliente, projetistas, gerentes – influi sobre os requisitos para conhecer com precisão o objetivo que se pretende alcançar.

1.5 Qualidade e bugs I: insetos inofensivos

Como em qualquer outra área de conhecimento, em computação existem alguns equívocos bastante comuns no uso de terminologia. Muitos deles são inofensivos, como não saber distinguir entre programação usando tipos abstratos de dados e programação orientada a objetos. Outros equívocos são, talvez, um pouco menos inócuos, como afirmar que o uso de saltos incondicionais é incompatível com programação estruturada [Knuth, 1974]. A relação entre bugs e qualidade, às vezes também causa certas confusões.

Geralmente o simples fato de pronunciar a palavra bug equivale a acionar um alarme e fazer uma equipe de programadores estressados entrar em pânico. A discussão sobre o assunto se resume tipicamente à equação inexata

$$\text{qualidade} = \overline{\text{bug}}$$

isto é, qualidade de software e bugs são coisas opostas e incompatíveis. Assumir essa idéia cegamente faz tanto sentido quanto dizer que um programa que tem uma instrução `goto` não é um programa estruturado.

A maioria das pessoas, sobretudo estudantes de computação, fica em geral chocada com a idéia de que um programa de computador possa ter erros e continuar sendo um produto de qualidade. Segundo se pensa geralmente, um bug (inseto em inglês – veja a Figura 1.5) é algo a ser exorcisado a todo custo, pois não é possível dizer que um programa errado é um programa bom.

Como poderia ser diferente?

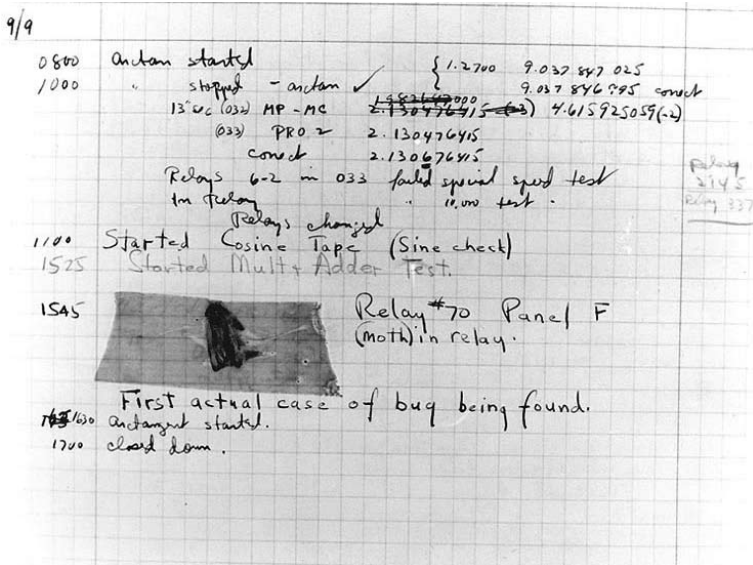


Figura 1.5 – Foto de um inseto encontrado em um computador em 1945 (Copyright: U.S. Naval Historical Center Photograph).

Vamos considerar essa questão à luz de alguns exemplos.

- **O dilema gerencial:** este caso é narrado por Weinberg [1994]: é a história de um programa de edição de texto. O programa tinha falhas que apareciam apenas sob certas condições, como trabalhar com textos muito longos. Procurado por alguém insatisfeito, o fabricante do programa explicou que estava ciente da falha, mas que esta era rara: menos de 1% dos clientes tiveram o problema. Alterar o código para corrigir o defeito implicaria mudanças profundas e poderia introduzir novos problemas – um risco muito grande. Assim, em nome dos 99% de clientes satisfeitos, o gerente tinha razão em não corrigir o programa.
- **A importância relativa I:** muitos programas de computador possuem defeitos conhecidos e considerados pelos usuários como de menor importância. Como exemplo, vários jogos possuem imperfeições no tratamento de colisão entre sólidos e, assim, objetos atravessam paredes, o que não impede que os usuários continuem se divertindo e mesmo recomendando tais produtos. Programas mais “sérios” também apresentam erros; embora seja um defeito raro, o Matlab R12, por vezes, trava. Isso não impede que milhares de cientistas usem o Matlab para realizar cálculos – algo que o programa faz extremamente bem.

- **A importância relativa II:** o sistema de tipografia T_EX é lendário pela qualidade de resultado impresso, pelo preço (é gratuito) e pelo fato de que, depois de anos de uso, poucos erros terem sido encontrados. Entretanto, seu uso requer conhecimento especializado, exigindo certo tempo de adaptação mesmo para um usuário que seja programador. A ferramenta continua sendo a escolha preferida para escrita de livros científicos, artigos e teses, mas não é adequada para enviar rapidamente um orçamento a um cliente, contendo algumas tabelas com diversos formatos e alguns gráficos: nesses casos será preferível usar outro programa.

A qualidade de um software, como se pode ver, depende de se decidir o que significa qualidade! Não é um assunto que possa ser tratado com dogmas: “Não cometerás erros de programação”. Em vez disso, é preciso adotar uma perspectiva técnica e considerar diversos fatores que afetam a construção do produto e que influenciem no julgamento dos usuários:

- tamanho e complexidade do software sendo construído;
- número de pessoas envolvidas no projeto;
- ferramentas utilizadas;
- custos associados à existência de erros;
- custos associados à detecção e à remoção de erros.

Um estudante de computação, cercado de provas e de matéria a estudar no final de um período letivo, normalmente está pronto para aceitar uma nota 7 em troca de alguns bugs. Um programador de uma empresa pressionado por cronogramas e chefes irritados, em raríssimos casos consegue lutar contra o ambiente e dizer: “Vou atrasar a entrega para realizar mais testes”. Finalmente, uma equipe militar responsável pelo programa de controle de um míssil, provavelmente, não iniciaria um projeto sem haver incluído no estudo de viabilidade condições estritas de orçamento, realização de testes e revisões, que, certamente, não existem nos dois casos anteriores.

Há um conceito chamado de “zero-defeitos”; trata-se com certeza, de um conceito interessante e um ideal a ser buscado. Mas, de um ponto de vista de administração e de engenharia, é mais realístico se perguntar até que ponto pode-se evitar os erros em um dado projeto e, o que é decisivo, qual o custo e quais os lucros esperados.

1.6 Um erro é um defeito, uma falha ou bug?

Qual a melhor palavra para explicar que um programa “travou” ou não funciona corretamente?

Há termos relacionados com erros de programação que, algumas vezes, provocam um pouco de confusão. Embora evoquem idéias parecidas, defeito, erro e falha não são sinônimos entre si e são usadas para designar conceitos distintos.

1.6.1 Defeito

Defeito é uma imperfeição de um produto. O defeito faz parte do produto e, em geral, refere-se a algo que está implementado no código de maneira incorreta.

Considere, por exemplo, o código a seguir:

```
a = input ();  
c = b/a;  
d = a ? b/a : 0;
```

A expressão que calcula o valor para *c* pode apresentar um problema: se a variável *a* contiver um valor nulo, a operação de divisão por zero provocará algo anormal na execução do programa. A linha seguinte, onde uma atribuição é feita à variável *d*, não apresenta esse problema.

O comportamento anormal do programa – provavelmente um *crash*, isto é, interrupção ou aborto da execução – é provocado pela divisão *b/a*. A primeira idéia, então, é dizer que essa linha de código é defeituosa. Existe, entretanto, uma outra hipótese: o defeito pode estar na rotina `input()`: imagine que a especificação dessa rotina estabeleça que ela não deve jamais retornar um valor nulo. Nesse caso, o erro foi cometido pelo programador responsável por essa rotina. Esta segunda hipótese é bastante razoável, pois para a maioria dos programas é irrealístico preceder cada operação de divisão com um teste `if`.

Mas a palavra “defeito” não significa apenas um problema que faz um programa não funcionar. Um programa defeituoso, segundo o dicionário Houaiss, é um programa “que não funciona como deve”.

Realizar testes e gastar tempo com revisões apenas procurando possibilidades de *crash* não é suficiente. É claro que um programa que sofre muitas paralisações deve ser corrigido, mas existem outros tipos de erros a serem tratados.

Considere o exemplo que aparece na Figura 1.6:

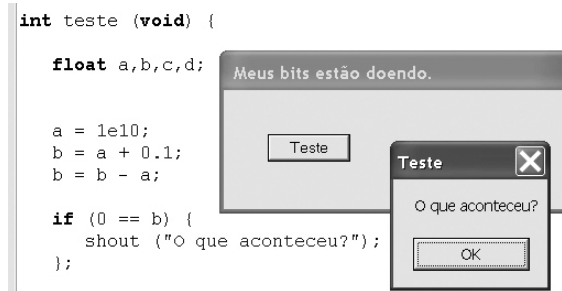


Figura 1.6 – Falha provocada por imprecisão de cálculo.

O código, em C++, faz o seguinte:

- atribui 10.000.000.000 à variável **a**;
- soma 0.1 a esse valor e o armazena em **b**;
- subtrai 10.000.000.000 desse valor.

Qual o resultado desse cálculo? Obviamente, 0.1. Mas, por que, então, o programa apresenta a mensagem na tela?

Este trecho de código é um exemplo bem conhecido dos engenheiros que trabalham com cálculo numérico. Embora o programa esteja aparentemente correto, há aqui um problema semântico: as variáveis possuem uma precisão limitada. Isso significa que certos cálculos terão resultados incorretos, não porque o programador escreveu algo errado, mas porque não há casas decimais suficientes para representar os valores. O defeito, neste caso, pode ser resolvido simplesmente trocando os tipos das variáveis por uma representação mais adequada – por exemplo, um `double`. Em programas extensos, a situação é bem mais complexa, pois existe o fenômeno de propagação de erros de um cálculo a outro.

Podem existir inúmeros outros tipos de defeitos que não resultam no aborto do programa. Tais erros são menos aparentes, mas podem ser igualmente graves.

1.6.2 Falha

Falha é o resultado errado provocado por um defeito ou condição inesperada. No primeiro exemplo de código defeituoso – uma divisão por zero – é possível que o programa funcione durante longos anos, sem que jamais ocorra uma falha: tudo dependerá dos valores retornados pela rotina `input()`. Um exemplo consta na Figura 1.7.

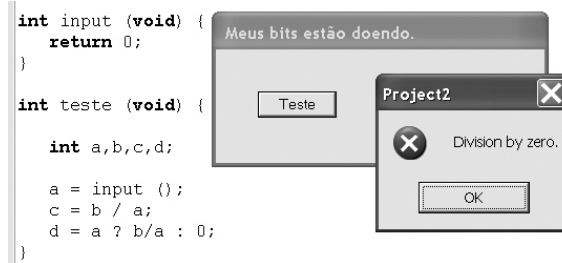


Figura 1.7 – Falha provocada por uma divisão por zero.

Os defeitos podem existir, mas nem sempre ser visíveis. Falhas também podem ocorrer por fatores externos ao programa, como corrupção de bases de dados ou invasões de memória por outros programas.

Como foi dito antes, as falhas que chamam mais a atenção são certamente aquelas em que o programa aborta. Contudo, toda falha potencial pode ser perigosa, mesmo se o programa não for paralisado.

1.6.3 Isolar um defeito

Isolar um defeito consiste em determinar sob quais condições ocorre. O objetivo é encontrar as causas dentro de um programa que estão ocasionando falhas e isso implica descobrir em qual linha de código ocorre uma falha como um crash (ou seja, o programa é abortado).

Isolar um defeito pode ser bastante difícil: apenas olhando janelas como as que aparecem na Figura 1.6, seria, em princípio, impossível determinar onde, em um programa extenso, está localizada a linha defeituosa que provocou a divisão por zero. Além disso, é preciso que se consiga repetir a falha sistematicamente. Se é impossível repeti-la, é improvável que o defeito possa ser encontrado.

Algumas falhas são bastante difíceis de reproduzir. É preciso encontrar precisamente a combinação de fatores como entradas de dados e comandos executados na interface que fazem que ela se manifeste. A tarefa também pode ser cara: em um software no qual um dos autores trabalhou, era preciso que o sistema (um simulador) funcionasse durante algumas dezenas de minutos antes que ocorresse o problema!

Muitos programadores (e, sobretudo, gerentes) desconhecem a existência de técnicas e ferramentas que permitem facilitar a tarefa de depuração de código. Falaremos disso nos Capítulos 15 e 16.

1.6.4 Estabilizar um programa

Estabilizar um programa é o termo, em geral, utilizado para referir-se a correções que resultam na diminuição na frequência de falhas. Um programa estável apresenta poucas falhas – um indicativo de que deve possuir poucos defeitos. De maneira bastante geral, a estabilidade está ligada à idade de um programa. Mais tempo de uso representa mais possibilidades de encontrar e corrigir problemas de execução.

1.7 Qualidade e bugs II: catástrofes

Os defeitos de software que levam ao aborto do programa são certamente bastante inconvenientes. Mas, como foi dito antes, não constituem o único aspecto que determina a qualidade de um produto: há outros fatores, como o preço, que não devem ser desprezados quando se busca determinar a qualidade.

A gravidade de uma falha de software é relativa. Existem falhas com as quais usuários podem conviver, a tal ponto que o sucesso de aplicação de um produto não seja afetado; em outros casos, a falha do programa representa um completo fracasso comercial. Finalmente, há programas de computador responsáveis pelo controle de equipamentos valiosos ou que podem colocar em risco a segurança física de pessoas.

Erros de software já foram responsáveis por prejuízos milionários e mesmo a perda de vidas humanas. A importância de garantir a qualidade é evidente à luz desta citação [Pressman, 2002]: "O software de computadores... está embutido em sistemas de todas as naturezas: de transportes, médicos, de telecomunicações, militares, de processos industriais, de produtos de escritório,... a lista é quase sem-fim".

A análise de falhas que tenham sido identificadas e documentadas abre a possibilidade para que sejam estudadas técnicas para evitar erros no futuro.

Nas próximas seções apresentaremos alguns erros de software cujas conseqüências foram dramáticas.

1.7.1 Ariane 501

Em 4 de junho de 1996, foi lançado o primeiro foguete Ariane 5. Decorridos 40 segundos da seqüência de lançamento e a uma altitude de 3.700 metros, o foguete desviou-se de sua trajetória e se autodestruíu com uma explosão. O custo desse desastre foi avaliado em mais de 300 milhões de dólares, quantia suficiente para pagar um salário de 2,5 mil dólares a cem programadores que trabalhassem durante um século.

A trajetória do foguete era medida por um sistema de referência inercial (SRI), cujos dados alimentavam um computador. Os equipamentos eram redundantes: havia duas unidades SRI exatamente idênticas. Caso a principal falhasse (SRI-2), o computador passava imediatamente a utilizar a de reserva (SRI-1). Havia também um segundo computador redundante. O relatório que analisou o acidente³ descreveu os eventos em ordem cronológica inversa, como segue.

O foguete começou a desintegrar-se a 39 segundos, em razão a uma carga aerodinâmica excessiva: a pressão do ar contra o veículo estava muito elevada. O motivo foi o ângulo de ataque muito pronunciado, ou seja, em vez de “cortar” o ar na vertical, o foguete estava em um ângulo de 20 graus.

O ângulo exagerado de ataque foi causado por um comando de direcionamento dos motores. Esse comando foi enviado pelo computador com base nos dados fornecidos pelo SRI-2. Entre esses dados havia um padrão de bits significando um código de erro, incorretamente interpretado como informação de vôo.

O SRI-2 não forneceu dados corretos, mas um código de erro, em virtude de uma exceção de software. O sistema de reserva (SRI-1) não pôde ser utilizado porque ele próprio já havia reportado a mesma falha, 72 milissegundos antes.

Mas, o que causou, de fato, o problema?

Uma exceção é uma condição inesperada em um programa, que é tratada geralmente por uma sub-rotina-padrão. Nos exemplos de código apresentados na Seção 1.6, a divisão por zero em um processador Intel causa uma exceção que é, em geral, tratada pelo sistema operacional. Tipicamente, o programa é simplesmente abortado.

No caso do Ariane 5, a exceção também foi proveniente de um cálculo: um número em ponto flutuante representado com 64 bits foi convertido para um inteiro com sinal de 16 bits. O número era demasiado grande para ser representado com 16 bits e isso causou uma falha. Existiam outros pontos do mesmo código com conversões semelhantes, mas que eram protegidos por testes. O trecho do software havia sido copiado do Ariane 4, onde funcionava corretamente; no Ariane 5, o cálculo se tornava defeituoso em razão do comportamento diferente do foguete. Pior do que isso, tal cálculo sequer era necessário.

Embora o erro no código fosse muito pequeno (podendo ser tratado por um simples comando `if`), é fácil perceber que do ponto de vista de projeto o defeito era bem mais complexo. Este exemplo ilustra diversos aspectos que cercam a qualidade de software:

3 Ariane 5 – Flight 501 Failure – Report by the Inquiry Board, J. L. Lions. Disponível na Internet.

- o caractere determinante dos requisitos sobre os resultados;
- a dificuldade de garantir que requisitos sejam consistentes em projetos complexos;
- a lei de Dijkstra, segundo a qual testes não garantem a ausência de erros;
- a dificuldade de verificar e validar programas.

1.7.2 Therac-25

O Therac-25 era uma máquina utilizada em terapia radiológica. Diferente de suas versões anteriores, era totalmente controlado por um computador, um PDP-11.

Enquanto as versões anteriores possuíam travas mecânicas para impedir erros de operação, no Therac-25 toda segurança ficou a cargo do software. Infelizmente, havia diversos erros no projeto do equipamento, que levaram à morte de pacientes [Levenson, 1995]. As mensagens de erro não eram claras: algumas se limitavam à palavra *malfunction*, seguida de um número entre 1 e 64. A ocorrência de falhas era bastante comum; não obstante, os operadores teriam recebido a informação de que havia diversos mecanismos de segurança e que, por isso, não era preciso se preocupar.

Em 26 de julho de 1985, na Ontario Cancer Foundation, em Ontário, Canadá, um operador acionou a máquina e decorridos cinco segundos ela parou de funcionar. O display mostrava a mensagem: *htilt error*, erro de posicionamento horizontal. Havia indicações de *no dose*, ou seja, nenhuma radiação teria sido emitida; e *treatment pause*, a máquina estava em simples pausa aguardando para continuar a operação. Como se tratava de mensagens comuns, o operador simplesmente ativou outra vez o Therac-25. A máquina desligou-se novamente com as mesmas mensagens. O operador insistiu um total de cinco vezes, quando, então, o Therac-25 entrou em um modo de suspensão que obrigava uma reinicialização do computador. Um técnico do hospital foi chamado e não verificou nada de anormal com o equipamento; não seria a primeira vez que isso teria acontecido.

A paciente faleceu em 3 de novembro. Uma autópsia revelou que a superexposição à radiação causou tantos danos que, se ela tivesse sobrevivido, seria preciso receber uma prótese para a cabeça do fêmur praticamente destruída pela radiação.

Depois da ocorrência de outros acidentes em diversos hospitais, o médico Frank Borger, da Universidade de Chicago, decidiu investigar por seus próprios meios o que estava acontecendo. Ele havia percebido que quando um grupo de novos estudantes iniciava o uso de um Therac-20 (o modelo anterior), cerca de três vezes por semana havia fusíveis queimados no equipamento. Depois de algum tempo, os

erros desapareciam misteriosamente, até que um novo grupo começasse a usar a máquina pela primeira vez.

O médico orientou um novo grupo de alunos a testar vários tipos de erros diferentes e a serem criativos ao introduzirem parâmetros na máquina. Por meio dessa experimentação, ele descobriu que certas seqüências de comandos e de edição de parâmetros resultavam em fusíveis queimados e outras falhas na operação.

O que Borger estava fazendo era uma excelente demonstração de como isolar um defeito de software. Curiosamente, aparentemente ninguém na empresa responsável, a AECL (Atomic Energy of Canada Limited) pensou em fazer o mesmo.

Os acidentes continuaram acontecendo, revelando possíveis falhas mecânicas, erros de código e no projeto como um todo. O software continha procedimentos concorrentes em que condições de corrida⁴ (*race conditions*) podiam ocorrer. Mensagens de erro que eram empregadas apenas pelos desenvolvedores do software foram vistas no display por operadores do Therac-25. Finalmente, as declarações de confiabilidade feitas pela AECL careciam de embasamento; a cada incidente, a empresa publicava relatórios de melhoria. Em um desses relatórios, afirmava-se que a possibilidade de erros havia sido reduzida de cinco casas decimais – um resultado, a rigor, muito improvável. Seis pacientes foram vítimas dos erros de projeto do Therac-25.

1.8 Qualidade e o SWEBOK

As bases teóricas dos computadores modernos remontam a 1936, com o trabalho de Alan Turing: isso significa somente 64 anos antes do bug do milênio. O computador ABC começou a ser construído em 1937, na Iowa State University, enquanto o ENIAC foi concluído em 1946. Comparativamente, a mecânica newtoniana data de 1664, uma diferença de três séculos. O tempo permite não apenas que novos conhecimentos sejam produzidos, mas também que tais conhecimentos sejam verificados, corrigidos e melhorados. Kuhn [1996] mencionou assim, o papel que o tempo exerce na evolução histórica da ciência:

Se a ciência é o conjunto de fatos, teorias e métodos... o desenvolvimento científico torna-se o processo fragmentário pelo qual esses elementos foram reunidos, separadamente ou em combinação, ao fundo comum em contínuo crescimento que constitui a técnica e o conhecimento científicos.

4 Quando há possibilidade de que dois processos possam acessar uma região crítica em função da seqüência em que as instruções são executadas, diz-se que há uma condição de corrida.

Nas últimas décadas, tornou-se claro o desdobramento da computação em uma extensa lista de subáreas de estudo. Além de um crescimento explosivo da tecnologia, ocorreu também uma evolução importante dos alicerces, isto é, da ciência. A quantidade de informação aumentou de tal modo que a especialização profissional tornou-se comum, senão mesmo necessária⁵ se for desejado um nível de excelência.

Uma das áreas de computação – a Engenharia de Software – passou por um estudo de uma comissão internacional de especialistas, visando a uma definição das fronteiras que a delimitam. Esse estudo foi conduzido no âmbito da IEEE e chama-se SWEBOK (Software Engineering Body Of Knowledge, ou Corpo de Conhecimento de Engenharia de Software) [SWEBOK, 2004].

A Engenharia de Software é dividida no SWEBOK em um total de onze áreas de conhecimento (KA: Knowledge Area): requisitos, gerência de engenharia, projeto, métodos e ferramentas de engenharia, construção, processo de engenharia, testes, qualidade, manutenção, disciplinas relacionadas e gerência de configuração.

Como acontece em todas as ciências, cada uma dessas divisões trabalha em conjunto com as demais e envolve a aplicação (e desenvolvimento) de conhecimentos mesmo em outros domínios. A teoria da computação, por exemplo, não existe como uma entidade separada da matemática. As classificações são úteis para que possamos organizar o conhecimento e direcionar esforços. Pessoas diversas podem adotar divisões um pouco diferentes.

Em relação à qualidade, o SWEBOK fez uma distinção entre técnicas estáticas e dinâmicas. As primeiras aparecem sob a área de conhecimento Qualidade, enquanto as últimas figuram na área de Testes. A norma internacional ISO/IEC 25000 SQuaRE, que trata da qualidade de produtos de software, abrange esses dois tópicos.

Na verdade, todas as subáreas da engenharia de software têm alguma relação com a qualidade de programas de computador. O SWEBOK inclui qualidade como uma área de conhecimento específica, mas não deve ser considerada estanque do restante daquele guia. Outro exemplo é a Ergonomia de Software: no SWEBOK, é tratada dentro de “disciplinas relacionadas”, mas sabe-se que requisitos de ergonomia podem causar impacto até sobre a segurança de operação de um produto. Um exemplo disso são softwares de controle de tráfego aéreo.

Cada área de conhecimento no SWEBOK é subdividida em até dois níveis, formando uma estrutura hierárquica para catalogar os assuntos. No caso da área de qualidade, essa organização hierárquica é apresentada na Figura 1.8.

5 O leitor é convidado a ler *Ponto de Mutação*, de Fritjof Capra, para uma discussão abrangente sobre os malefícios resultantes da especialização e a importância da interdisciplinaridade em todas as profissões.

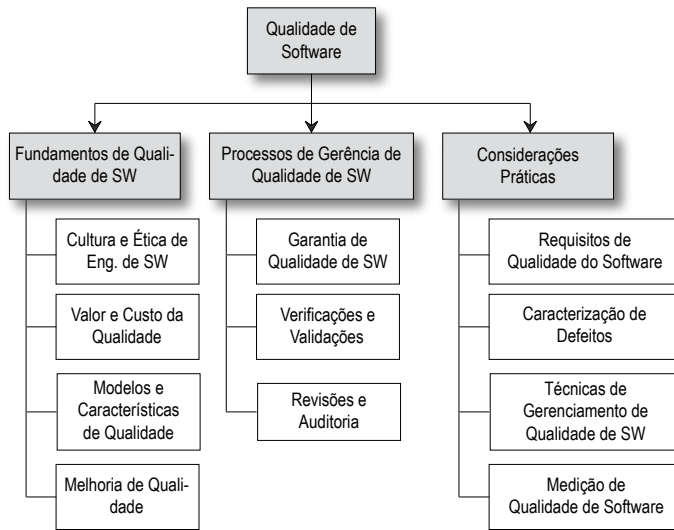


Figura 1.8 – Divisão em tópicos da qualidade – SWEBOOK, 2004.

A divisão em tópicos será rapidamente descrita nas próximas subseções.

1.8.1 Fundamentos de qualidade

Este tópico abrange, sobretudo, a noção de qualidade, ou seja, sua definição. Essa definição, no caso de um produto, materializa-se por meio da definição de requisitos e estes dependem de um modelo. Um dos modelos mais importantes existentes atualmente é a norma SQuARE, ISO/IEC 25000. Este livro contém um capítulo especial dedicado a essa norma.

Os aspectos éticos do trabalho com software têm se tornado mais evidentes com nossa dependência da tecnologia; toda uma nova classe de problemas surgiu com os crimes de computador. Respostas sociais, éticas e de legislação estão sendo desenvolvidas para procurar tratar adequadamente cada caso. Como acontece com outros aspectos atuais da qualidade de software, o problema já havia sido previsto há muito tempo. Norbert Wiener, um professor do MIT preocupado com questões éticas, discutiu questões fundamentais sobre esse assunto dentro da área de computação [Wiener, 1965]: "Muito antes de Nagasaki e da informação pública sobre a bomba atômica, ocorreu-me que estávamos em presença de outra potencialidade social, de importância não conhecida, para o bem e para o mal".

A ética profissional é um tópico estudado em alguns cursos de computação no Brasil, em disciplinas como Informática e Sociedade; até o ano 2000, havia possivelmente uma única publicação em português sobre o assunto [Masiero, 2000].

O próximo tópico sob fundamentos da qualidade é a relação entre valor e custo e abrange basicamente dois aspectos: os prejuízos causados pela falta de qualidade de um produto e os custos com que é preciso arcar para garantir um determinado nível de exigência quanto ao funcionamento do software.

1.8.2 Processos de gerência de qualidade

Os processos de gerência abrangem todos os aspectos de construção do produto. Por conta disso, todos elementos de um projeto estão envolvidos: ferramentas como sistemas para controle de versão e linguagens, metodologias para revisão do produto, técnicas organizacionais e de administração de pessoas etc.

O propósito da subárea de garantia da qualidade é assegurar que os objetivos planejados no início do projeto serão cumpridos. De forma geral, isso significa estabelecer sistemas para controlar tudo o que ocorre durante o ciclo de vida, com o propósito de garantir que o programa que será fabricado fará aquilo que se espera dele.

As verificações e validações (V&V) consistem em atividades com um caráter um tanto diferente do que foi dito até aqui, pois nelas se considera a possibilidade de que algo esteja errado no produto. Idealmente, a garantia da qualidade deve trabalhar de maneira a que essas atividades não sejam necessárias. Ao mesmo tempo, isso não significa em absoluto que as atividades de V&V percam importância ou sejam realizadas com menos intensidade – na realidade, o que acontecerá é o oposto. Se a subárea de Garantia de Qualidade for bem-sucedida, o que se observará com as atividades de V&V será uma aprovação do produto com pouca ou sequer nenhuma restrição.

As auditorias são, em conceito, independentes da construção do software. Uma boa política consiste em empregar auditores que não participaram do projeto, ou ainda, auditores externos contratados de outra empresa. A auditoria é relacionada tanto a normas famosas, como a ISO 9000, como a padrões internos elaborados pela própria organização.

1.8.3 Considerações práticas

Este tópico contém observações de ordem prática, isto é, recomendações gerais sobre como transcorre a execução das atividades relacionadas com qualidade. Não há uma descrição explícita para este tópico no SWEBOK [2004].

Há quatro subtópicos; o primeiro é requisitos de qualidade de software. Nele são mencionados itens como “fatores de influência” sobre requisitos: orçamento para realização; usuários envolvidos; ferramentas e métodos necessários etc. Além disso

são considerados os aspectos relacionados com a segurança de funcionamento e as conseqüências que as falhas podem causar.

A caracterização (e detecção) de erros diz respeito, em última análise, a verificar a não-conformidade aos requisitos. Há diversas técnicas relacionadas, como: vários tipos de teste de software, revisões, inspeções, auditorias e ferramentas automatizadas de verificação.

As técnicas para gerenciamento de qualidade são classificadas no SWEBOOK em quatro tipos: orientadas a pessoas (people-intensive), como é o caso de revisões e auditorias; estáticas, que não envolvem execução do produto; dinâmicas, que são efetuadas durante a execução do software; e, finalmente, as técnicas analíticas, que fazem uso de métodos formais.

O último subtópico é medição da qualidade. Um conjunto de dados obtidos por medidas é um recurso de extrema ajuda para auxiliar a tomada de decisões gerenciais. Embora para muitos gerentes pareça mais natural que as medidas sejam usadas para saber o estado da implementação de um produto, não estão restritas ao estágio final do desenvolvimento do software. Como propõe a norma SQuaRE, o ideal é que os valores desejados para as medidas sejam estabelecidos no início do projeto, durante a fase de definição de requisitos.

1.9 Exercícios

1. Você alguma vez elaborou um cronograma para um software que tivesse que implementar, como a solução de um projeto de faculdade? Experimente: procure um projeto em um bom livro de estruturas de dados (por exemplo, Tenenbaum, Langsam e Augenstein) e elabore um cronograma. Inclua tempo para estudo e projeto, para programação e testes e acrescente uma margem de segurança. Peça a um colega seu para fazer a mesma coisa e depois compare os resultados. Por que há diferenças? Qual cronograma parece mais realístico?
2. A definição de qualidade de Crosby tem ao menos três pontos positivos e três pontos negativos, conforme comentados no texto. Relacione esses pontos comparando-os diretamente.
3. Dois clientes ao comprarem uma mesma camisa terão, possivelmente, opiniões muito diferentes sobre o produto. Isto é um exemplo de ruído de medição de qualidade? Justifique sua resposta.