1 Introdução

Este capítulo apresenta uma introdução ao assunto de engenharia de software, iniciando pelo clássico "*a crise do software*" (Seção 1.1) e pelo, também clássico "*os mitos do software*" (Seção 1.2). Procura-se dar um entendimento ao termo "*engenharia de software*" (Seção 1.3) e "*engenheiro de software*" (Seção 1.4), já que a literatura não é homogênea em relação a isso. A *evolução* da área é brevemente apresentada (Seção 1.5), bem como uma classificação dos *tipos de sistemas* perante a engenharia de software (Seção 1.6), já que este livro se especializa em engenharia de software para sistemas de informação (outros tipos de sistemas poderão necessitar de técnicas específicas que não são tratadas aqui). Finalmente, os *princípios* da engenharia de software, que permeiam todos os capítulos do livro são brevemente apresentados (Seção 1.7).

1.1 A Crise dos Desenvolvedores de Software menos Preparados

Por algum motivo os livros de engenharia de software quase sempre iniciam com o tema "*crise do software*". O termo vem dos anos 1970. Mas o que é isso afinal? O software está em crise? Parece que não, visto que o software hoje está presente em quase todas as atividades humanas. Mas as *pessoas* que desenvolvem software estão em crise há décadas e em alguns casos parecem impotentes para sair dela.

Em grande parte parece haver desorientação em relação a como planejar e conduzir o processo de desenvolvimento de software. Muitos desenvolvedores concordam que não utilizam um processo adequado e que deveriam investir em algum, mas ao mesmo tempo dizem que não tem tempo ou recursos financeiros para fazê-lo. Essa história se repete há décadas.

O termo "crise do software" foi usado pela primeira vez com impacto por Dijkstra (1971)[2]. Ele avaliava que considerando o rápido progresso do hardware e das demandas por sistemas cada vez mais complexos, os desenvolvedores simplesmente estavam se perdendo, porque a engenharia de software, na época era uma disciplina incipiente.

Os problemas relatados por Dijkstra eram os seguintes:

a) Projetos que estouram o cronograma.
b) Projetos que estouram o orçamento.
c) Produto final de baixa qualidade ou não atendendo aos requisitos.
d) Produtos não gerenciáveis e difíceis de manter e evoluir.

Alguma semelhança com sistemas do início do Século XXI? Muitas! Sucede que embora a engenharia de software tenha evoluído como ciência, sua aplicação na prática ainda é muito limitada, especialmente em empresas de pequeno porte e em empresas novas.

Mesmo depois de 40 anos ainda são comuns as queixas por parte da alta administração das empresas em relação ao setor de informática em relação a prazos não são cumpridos, custos ainda muito elevados, sistemas em uso que demandam muita manutenção, e também de que é difícil recrutar profissionais qualificados.

[2] www.cs.utexas.edu/users/EWD/ewd03xx/EWD340.PDF

Os usuários também estão infelizes: encontram erros e falhas inadmissíveis em sistemas entregues, sentem-se inseguros em usar tais sistemas, e reclamam da constante necessidade de manutenção e do seu alto custo.

Os desenvolvedores, por sua vez, não estão mais satisfeitos: sentem que sua produtividade é baixa em relação ao seu potencial, lamentam a falta de qualidade no produto gerado por seu trabalho, sofrem pressão para cumprir prazos e orçamentos apertados, e ficam inseguros com as mudanças de tecnologia que afetam sua qualificação em relação ao mercado.

Booch (1994) afirma: "*We often call this condition the software crisis, but* frankly*, a* malady *that has carried on this long must be called normal*"[3]. Pode-se concluir que a crise do software continuará enquanto os desenvolvedores de software continuarem a utilizar processos artesanais e a não capitalizarem erros e acertos.

Teixeira (2010)[4] compara o desenvolvimento de software ao artesanato da Idade Média, quando, por exemplo, um artesão fazia cada par de sapatos como um produto único para cada cliente. Buscava-se a matéria prima, cortava-se e costurava-se para produzir um sapato que servisse o cliente. O software, em muitas empresas, ainda é desenvolvido desta forma artesanal.

Porém, apenas a adoção de processos efetivamente industriais para a produção de software poderá fazer esta área desenvolver-se mais rapidamente, com mais qualidade e, finalmente, sair desta crônica dificuldade que já nem pode ser chamada de crise.

1.2 Os Eternos Mitos

São bastante conhecidos também os *mitos* do software identificados por Pressman (2005). Estes mitos são crenças tácitas e explícitas que permeiam a cultura de desenvolvimento de software. Os mais experientes acabam percebendo que essas crenças não têm fundamento, constituindo-se realmente em mitos, mas a cada ano novos desenvolvedores de software entram no mercado, reavivando as velhas crenças, já que seu apelo é grande.

Pressman classifica os mitos em três grupos: *administrativos*, do *cliente* e do *profissional*. Seguem abaixo alguns comentários sobre os mitos administrativos:

a) *A existência de um manual de procedimentos e padrões é suficiente para a equipe produzir com qualidade.* Na verdade deve-se questionar se o manual é realmente usado, se ele é completo e atualizado. Deve-se trabalhar com processos que possam ser gerenciáveis e otimizados, ou seja, sempre que a equipe identificar falhas no processo deve haver um processo para modificar o processo.
b) *A empresa deve produzir com qualidade, pois tem ferramentas e computadores de última geração.* Na verdade ferramentas e computadores de boa qualidade são condições necessárias, mas não suficientes. Como disse Larman (2001), comprar um martelo não transforma você em arquiteto.

[3] Frequentemente chamamos esta condição de crise do software, mas francamente, uma doença que já dura tanto tempo devia ser chamada de normalidade.
[4] pt.scribd.com/doc/37503268/A-Crise-de-SW-Hugo-Vidal-Teixeira

c) *Se o projeto estiver atrasado sempre é possível adicionar mais programadores para cumprir o cronograma*. O desenvolvimento de software é uma tarefa altamente complexa. Adicionar mais pessoas sem que houvesse antes um planejamento para isso pode causar mais atrasos ainda. Se não fosse assim, um programa de 20 mil linhas poderia ser escrito rapidamente por 20 mil programadores.
d) *Um bom gerente pode gerenciar qualquer projeto*. Gerenciar não é fazer. E o desenvolvimento de software é um processo complexo por vários motivos. Assim, mesmo que o gerente seja competente, se não houver boa comunicação com a equipe, competência técnica e um processo previsível de trabalho, pouco o gerente poderá fazer para obter o produto com a qualidade desejada.

Dentre os mitos relacionados ao cliente, pode-se citar:

a) *Uma declaração geral de objetivos é suficiente para iniciar a fase de programação. Os detalhes podem ser adicionados depois.* É verdade que não se pode esperar que a especificação inicial do sistema esteja correta e completa antes de iniciar a programação. Mas ter isso como meta é péssimo. Deve-se procurar obter o máximo de detalhes que for *possível* antes de iniciar a construção do sistema. Técnicas mais sofisticadas de análise de requisitos e uma equipe bem treinada poderão ajudar a construir as melhores especificações possíveis sem perda de tempo.
b) *Os requisitos mudam com frequência, mas sempre é possível acomodá-los, pois o software é flexível*. Na verdade o *código* é fácil de mudar. Basta usar um editor. Mas mudar código sem introduzir erros é uma tarefa bastante improvável, especialmente em empresas com baixa maturidade de processo. O software só será efetivamente flexível se for construído com esse fim. É necessário, entre outras coisas, identificar os requisitos permanentes e transitórios, e, no caso dos transitórios, preparar o sistema para sua mudança, pela utilização de padrões de *design* adequados. Mesmo que o software não seja um elemento físico como um edifício ou uma ponte (estes mais difíceis de serem modificados), a mudança do software também implica em esforço e custo (em tempo), e muitas vezes este esforço e custo não são triviais.
c) *Eu sei o que preciso.* Já os desenvolvedores usualmente dizem o inverso: o cliente não sabe o que precisa. É necessário que os analistas entendam que os clientes (a não ser que sejam técnicos especializados), raramente sabem o que realmente precisam, e tem grande dificuldade em descrever e mesmo de lembrar-se de suas necessidades. Por outro lado, analistas muitas vezes confundem *necessidades de cliente* (alvo da *análise*) com *soluções possíveis* (alvo do *design*). Por exemplo, um analista pode achar que o cliente precisa de um sistema *web* com tabelas relacionais, mas isso não é uma boa descrição de uma necessidade: é a descrição de uma possível solução para uma necessidade que possivelmente não foi claramente estabelecida. Outras soluções possíveis poderiam se aplicar.

Outros mitos de Pressman dizem respeito ao profissional, e são comentados abaixo:

a) *Assim que o programa for colocado em operação nosso trabalho terminou*. Na verdade, ainda haverá *muito* esforço a ser despendido depois da instalação do sistema devido a erros dos mais diversos tipos. Estudos indicam que mais da metade do

esforço despendido com um sistema ocorre depois de sua implantação (von Mayrhauser & Vans, 1995).

b) *Enquanto o programa não estiver funcionando, não será possível avaliar sua qualidade*. Na verdade o programa é apenas um dos artefatos produzidos no processo de construção do software (possivelmente o mais importante, mas não o único). Existem formas de avaliar a qualidade de artefatos intermediários como casos de uso e modelos conceituais para verificar se estão adequados mesmo antes da implementação do sistema.

c) *Se eu esquecer alguma coisa, posso arrumar depois*. Quanto mais o processo de desenvolvimento avança, mais caras ficam as modificações em termos de tempo e dinheiro.

d) *A única entrega importante em um projeto de software é o software funcionando*. Talvez essa seja a entrega mais importante, porém, se os usuários não conseguirem utilizar este sistema, ou se os dados não forem corretamente importados, ele pouco valor terá. O final de um projeto de informática usualmente envolve mais do que simplesmente entregar o software na portaria da empresa do cliente. É necessário realizar testes de operação, importar dados, treinar usuários, definir procedimentos operacionais, e assim, outros artefatos atém do software funcionando poderão ser necessários.

Leveson (1995) também apresenta um conjunto de mitos correntes, a maioria dos quais relacionados à confiabilidade do software:

a) *O teste do software ou sua verificação formal pode remover todos os erros*. Na verdade o software é construído baseado em uma especificação de requisitos que usualmente é feita em linguagem natural e, portanto, aberta a interpretações. Nestas interpretações pode haver erros ocultos. Além disso, a complexidade do software contemporâneo é tão grande que se torna inviável testar todos os caminhos possíveis. Assim, a única possibilidade é testar aqueles caminhos mais prováveis de ocorrer ou com maior probabilidade de provocar erros. As técnicas de teste é que vão indicar quais caminhos é interessante testar.

b) *Aumentar a confiabilidade do software aumenta a segurança*. O problema é que o software pode ser confiável apenas em relação à sua especificação, ou seja, ele pode estar *fazendo certo a coisa errada*. Isso normalmente se deve a requisitos mal compreendidos. É consenso que corrigir erros nos requisitos é muito mais barato do que corrigi-los em um sistema em operação.

c) *Reuso de software aumenta a segurança*. O reuso de software certamente aumenta a confiança, pois os componentes reusados já foram testados. Mas o problema é saber se os requisitos foram corretamente estabelecidos, o que leva de volta ao mito anterior. Além disso, se os componentes reusados forem sensíveis ao contexto (o que acontece muitas vezes), coisas inesperadas poderão acontecer.

Porém, de nada adianta estar consciente dos mitos. Para produzir software com mais qualidade e confiabilidade é necessário utilizar um série de conceitos e práticas. Ao longo deste livro vários conceitos e práticas úteis em engenharia de software serão apresentados ao

leitor de forma que possa compreender o alcance desta área e seu potencial para melhoria dos processos de produção de sistemas

1.3 (In)Definição de Engenharia de Software

Segundo a Desciclopédia[5], engenharia de software pode ser definida assim: "A engenharia de software forma um aglomerado de conceitos que dizem absolutamente nada e que geram no estudante desta área um sentimento de 'Nossa, li 15 kg de livros desta matéria e não aprendi nada'. É tudo bom senso.".

Apesar de considerar que o material publicado na Desciclopédia seja de caráter humorístico, a sensação que muitas vezes se tem da engenharia de software é aproximadamente essa mesmo. Pelo menos essa foi a sensação do autor deste livro e de muitos de seus colegas ao longo de vários anos.

Efetivamente não é algo simples conceituar e praticar a engenharia de software. Mas é *necessário*. Primeiramente deve-se ter em mente que os processos de engenharia de software são diferentes dependendo do tipo de software que se vai desenvolver. O Capítulo 2, por exemplo, vai mostrar que dependendo do nível de conhecimento ou estabilidade dos requisitos deve-se optar por um ou outro ciclo de vida, o qual será mais adequado ao tipo de desenvolvimento que se irá encontrar. O Capítulo 8, por outro lado, vai mostrar que uma área aparentemente tão subjetiva como "riscos" pode ser sistematizada e tratada efetivamente como um processo de engenharia, e não como adivinhação. O Capítulo 7 apresentará formas objetivas e padronizadas para mensurar o esforço do desenvolvimento de software de forma a gerar números que sejam efetivamente realistas, o que já vem sendo comprovado em diversas empresas.

Assim, espera-se que este livro consiga deixar no leitor a sensação de que a engenharia de software é efetivamente possível. Ela é viável desde que se compreenda em que ela realmente consiste e como pode ser utilizada na prática.

Várias definições de engenharia de software podem ser encontradas na literatura, como, por exemplo:

a) Engenharia de software é uma profissão dedicada a projetar, implementar e modificar software, de forma que ele seja de alta qualidade, a um custo razoável, manutenível e rápido de construir (Laplante, 2007).
b) Engenharia de software é a aplicação de abordagens sistemáticas, disciplinadas e quantificáveis ao desenvolvimento, operação e manutenção de software, além do estudo destas abordagens (IEEE Computer Society, 2004)[6].

Neste livro será considerada com maior ênfase, a definição da engenharia de software como o processo de estudar, criar e otimizar os processos de trabalho para os desenvolvedores de software. Assim, considera-se, embora isso não seja consenso geral, que as atividades de

[5] desciclo.pedia.ws/wiki/Engenharia_de_Software (Consultado em: 10/08/2010)
[6] www.computer.org/portal/web/swebok/htmlformat

levantamento de requisitos, modelagem, design[7] e codificação, por exemplo, não são atividades típicas de um engenheiro de software, embora ele, muitas vezes, seja habilitado a realizá-las. Sua tarefa consiste mais em observar, avaliar, orientar e alterar os processos produtivos, quando necessário. A seção seguinte procura deixar essa distinção entre as atividades dos desenvolvedores e do engenheiro de software mais clara.

1.4 O Engenheiro de Software

Uma das primeiras confusões que se faz nesta área é confundir o *desenvolvedor* com o *engenheiro* de software. Isso equivale a confundir o engenheiro civil com o pedreiro ou com o mestre de obra.

O desenvolvedor, seja ele analista, *designer*, programador ou gerente de projeto, é um *executor* do processo de construção de software. Os desenvolvedores, de acordo com seus papeis, têm a responsabilidade de descobrir os requisitos e transformá-los em um produto executável. Mas o engenheiro de software tem um metapapel em relação a isso. Pode-se dizer que o engenheiro de software não coloca a mão na massa, assim como o engenheiro civil não vai à obra assentar tijolos ou concretar uma laje.

Então, o engenheiro de software não é um desenvolvedor que trabalha nas atividades de análise e produção de código. Porém, a comparação com a engenharia civil termina por aqui, já que o engenheiro civil será o responsável pela especificação do *design*. Na área de computação, a especificação do *design* fica a cargo do analista e do *designer*, o primeiro com a responsabilidade de identificar os requisitos e o segundo com a responsabilidade de desenhar uma solução que utilize a tecnologia para transformar estes requisitos em um sistema executável. No âmbito do software estes papeis tem sido por muito tempo confundidos com o papel do engenheiro de software, mas por que dar outro nome ao analista e *designer*? Não há razão nisso, trata-se apenas de uma incompreensão do verdadeiro papel do engenheiro de software.

O engenheiro de software assemelha-se assim mais ao engenheiro de produção. Ele deve fornecer aos desenvolvedores (inclusive gerentes, analistas e *designers*), as ferramentas e processos que estes deverão usar e ele será o responsável por verificar que tais ferramentas e processos sejam efetivamente usados, que sejam usados de forma otimizada e que, caso apresentem qualquer problema, ele será responsável por realizar as modificações necessárias no próprio processo, garantindo assim sua contínua melhoria.

O engenheiro de software, assim, não desenvolve nem especifica software. Ele viabiliza e acompanha o processo de produção fornecendo e avaliando as ferramentas e técnicas que julgar mais adequadas a cada projeto ou empresa.

Ainda é necessário distinguir o engenheiro de software do gerente de projeto. O gerente de projeto deve planejar e garantir que o projeto seja executado de forma adequada dentro dos prazos e orçamento especificados. Mas o gerente de projeto tem uma responsabilidade mais restrita ao projeto em si e não ao processo de produção. Neste sentido, o gerente de projeto também é um executor, ele utiliza as disciplinas definidas no processo de engenharia de

[7] As palavras inglesas *design* e *project* são traduzidas para o portugues usualmente como “projeto”. Para evitar confusão entre os dois significados, neste livro, o termo “*design*” não será traduzido.

software para gerenciar seu projeto específico, mas ele não é necessariamente o responsável pela evolução destes processos nem necessariamente responsável pela sua escolha. Este papel cabe ao engenheiro de software.

Resumindo, os diferentes papeis poderiam ser caracterizados assim:

a) O *engenheiro de software* escolhe e, muitas vezes, especifica os processos de planejamento, gerência e produção a serem utilizados. Ele acompanha e avalia o desenvolvimento de todos os projetos da empresa para verificar se o processo estabelecido é executado de forma eficiente e efetiva. Caso sejam necessárias mudanças no processo estabelecido, ele as identifica e realiza, garantindo que a equipe adote tais mudanças. Ele reavalia o próprio processo continuamente.
b) O *gerente de projeto* cuida de um projeto específico, garantindo que os prazos e orçamento sejam cumpridos. Ele segue as práticas definidas no processo de engenharia. Ele é responsável pela verificação da aplicação do processo por parte dos desenvolvedores e, se necessário, reporta-se ao engenheiro de software para sugerir melhorias no processo.
c) O *analista* é um desenvolvedor responsável pela compreensão do problema relacionado ao sistema que se deve desenvolver, ou seja, pelo levantamento dos requisitos e sua efetiva modelagem. O analista então deve descobrir o que o cliente precisa (por exemplo, controlar suas vendas, comissões, produtos, etc.).
d) O *designer* deve tomar as especificações do analista e propor a melhor tecnologia para produzir um sistema executável para elas. O *designer* então deve apresentar uma solução para as necessidades do cliente (por exemplo, propor uma solução baseada em *web*, com um banco de dados centralizado acessível por dispositivos móveis, etc.).
e) O *programador* vai construir a solução física a partir das especificações do *designer*. É ele que gera o produto final. Ele deve conhecer profundamente a linguagem e ambiente de programação, bem como as bibliotecas que for usar, e também deve ter algum conhecimento sobre teste e depuração de software.

Evidentemente que esta divisão de papéis não é muitas vezes observada estritamente nas empresas. Apenas empresas de médio e grande porte podem se dar ao luxo de terem um ou mais engenheiros de software dedicados. Mas é importante que se tenha em mente que, mesmo que a mesma pessoa execute mais de um papel neste processo, são papeis distintos.

1.5 Evolução da Engenharia de Software

Os primeiros computadores, construídos na década de 1940 não possuíam software: os comandos eram implantados fisicamente na máquina a partir de conexões físicas entre componentes. À medida que se percebeu a necessidade de computadores mais flexíveis, surgiu o software como tal, o qual consiste em um conjunto de instruções que fazem a máquina produzir algum tipo de processamento. Como o software era um construto abstrato, sua produção não se encaixava perfeitamente em nenhuma das engenharias, mesmo a mecânica e a elétrica, que são as mais próximas por terem relação com as máquinas que efetuam as computações. Surgiu então o conceito de engenharia de software, inicialmente referindo-se aos processos para a produção deste tipo de construto abstrato.

Aceita-se que a primeira conferência de engenharia de software tenha sido a Conferência de Engenharia de Software da OTAN, organizada em Garmish, Alemanha, em 1968 (Bauer, 1968)[8]. Mas, apesar disso, o termo já era usado desde os anos 1950.

A década de 1960 até meados da década de 1980 foi marcada pela chamada "crise do software", durante a qual foram identificados os maiores problemas relacionados à produção de software, especialmente em larga escala. Inicialmente a crise referenciava especialmente questões relacionadas com orçamento e cronograma de desenvolvimento, mas posteriormente também passou a abranger aspectos de qualidade de software, uma vez que sistemas, depois de prontos, apresentavam muitos erros, causando prejuízos.

Um exemplo clássico da crise de software dos anos 1960 foi o projeto do sistema operacional OS/360, que utilizou mais de mil programadores. Brooks (1975) afirmou ter cometido um erro que custou milhões à IBM neste projeto por não ter definido uma arquitetura estável antes de iniciar o desenvolvimento propriamente dito. Atualmente, a *Lei de Brooks* afirma que adicionar programadores a um projeto atrasado faz com que ele fique mais atrasado ainda.

A atividade de pesquisa por décadas tentou resolver a crise do software. Cada nova abordagem era apontada como uma *bala de prata*, para solucionar a crise. Porém, pouco a pouco houve um convencimento geral de que não havia tal solução mágica. Ferramentas *CASE* (*Computer Aided Software Engineering*), especificação formal, processos, componentes, etc. eram boas técnicas que ajudaram a engenharia de software a evoluir, mas correntemente não se acredita mais em uma solução única e salvadora para os complexos problemas envolvidos com a produção de software.

Os anos 1990 presenciaram o surgimento da Internet e a consolidação da orientação a objetos como paradigma predominante em produção de software. A mudança de paradigma e o *boom* da Internet mudaram de forma determinante a maneira como o software era produzido. Novas necessidades surgiram e sistemas cada vez mais complexos, acessíveis de qualquer lugar do mundo, substituíram os antigos sistemas *stand-alone*. Com isso, novas preocupações relacionadas à segurança da informação e à proliferação de vírus e *spam* surgiram e passaram a fazer parte da agenda dos desenvolvedores de software.

Nos anos 2000, o crescimento da demanda por software em organizações de pequeno e médio porte levou ao surgimento de soluções mais simples e efetivas para o desenvolvimento de software para essas organizações. Assim surgiram os métodos ágeis, que procuram desburocratizar o processo de desenvolvimento e deixá-lo mais adequado a equipes pequenas, mas competentes, capazes de desenvolver sistemas sem a necessidade de extensas listas de procedimentos ou *receitas de bolo*.

Atualmente, a área vem tentando se estabelecer como um corpo de conhecimentos coeso. O surgimento do *SWEBOK* (IEEE Computer Society, 2004) e sua adoção como padrão internacional em 2006 (ISO/IEC TR 19759) foi um avanço para a sistematização do corpo de conhecimentos da área.

[8] homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF

1.6 Tipos de Software do Ponto de Vista da Engenharia

Não existe um processo único e ideal para desenvolvimento de software, porque cada sistema tem suas particularidades. Porém, usualmente pode-se agrupar os sistemas de acordo com certas características e então definir modelos de processo mais adequados a estas características. Do ponto de vista da engenharia de software, os sistemas podem ser classificados como abaixo:

a) *Software básico*: são os compiladores, *drivers* e componentes do sistema operacional.

b) *Software de tempo real*: são sistemas que monitoram, analisam e controlam eventos do mundo real.

c) *Software comercial*: são os sistemas aplicados nas empresas, como controle de estoque, vendas etc. Tais sistemas usualmente acessam bancos de dados. São também conhecidos como *sistemas de informação*.

d) *Software científico e de engenharia*: são sistemas que utilizam intenso processamento de números.

e) *Software embutido ou embarcado*: são sistemas de software presentes em celulares, eletrodomésticos, automóveis, etc. Normalmente tais sistemas precisam trabalhar sob severas restrições de espaço, tempo de processamento e gasto de energia.

f) *Software Pessoal*: são os sistemas usados por pessoas no seu dia-a-dia, como processadores de texto, planilhas etc.

g) *Jogos*: embora existam alguns jogos cujo processamento não é muito complexo, existem aqueles que exigem o máximo dos computadores em função da qualidade de gráficos e necessidade de reação em tempo real. Apesar disso, todas as categorias de jogos têm características intrínsecas que extrapolam o domínio da engenharia de software.

h) *Inteligência artificial*: são os sistemas especialistas, redes neurais e sistemas capazes de alguma forma de aprendizado. Além de serem sistemas independentes, com um tipo de processo de construção próprio, podem também ser embutidos em outros sistemas.

Esta classificação, porém, não é completa, detalhada, nem exaustiva, mas apenas ilustrativa sobre os diferentes tipos de sistemas que são desenvolvidos com o uso de software e eventualmente hardware.

Neste livro, a ênfase estará nos sistemas de informação, ou seja, será mostrado basicamente como desenvolver um processo de engenharia de software para sistemas do tipo “comercial”. Apesar disso, algumas técnicas poderão ser aplicadas também ao desenvolvimento de outros tipos de software.

1.7 Princípios da Engenharia de Software

A engenharia de software classicamente apresenta um conjunto de princípios que devem ser usados quando um projeto de desenvolvimento de software for realizado. A ideia é que estes

princípios funcionem como boas práticas ou lições aprendidas sobre como desenvolver software. Usualmente não se trata de regras, mas de uma filosofia de desenvolvimento. Entre outros princípios pode-se citar:

a) *Decomposição*. Um dos mais antigos princípios em desenvolvimento de software é a noção de que o software é um produto complexo construído a partir de partes cada vez mais simples. A decomposição funcional é uma maneira de conceber o software como um conjunto de funções de alto nível (requisitos) que são decompostas em partes cada vez mais simples até chegar a comandos individuais de uma linguagem de programação. Modernamente essa noção foi substituída pela decomposição em objetos, onde ao invés de decompor a função original em funções cada vez mais simples, procura-se determinar uma arquitetura de classes de objetos que possa realizar a função.
b) *Abstração*. Outro princípio antigo da engenharia de software é o uso da abstração, que consiste em descrever um elemento em uma linguagem de nível mais alto do que o nível necessário para sua construção. Um sistema de software, por exemplo, pode ser composto por 100 mil linhas de código (sua representação física concreta), mas usando-se abstração, o mesmo sistema talvez possa ser descrito por um diagrama ou por umas 50 linhas de texto. A abstração ajuda os interessados no processo de desenvolvimento a entenderem estruturas grandes e complexas através de descrições mais abstratas.
c) *Generalização*. O princípio da generalização deu origem à orientação a objetos, onde um objeto pode ser classificado simultaneamente em mais de uma classe. Por exemplo, Rin-tin-tin, além de ser um cão também é mamífero, vertebrado e animal. Esse princípio é usado não só na classificação de dados, mas também em várias outras situações em projetos de desenvolvimento de software, como, por exemplo, a arquitetura dos sistemas.
d) *Padronização*. A criação de padrões (*patterns*) de programação, *design* e análise, ajuda a criar produtos com qualidade mais previsível. Padrões também são importantes para a capitalização de experiências. Desenvolvedores que não utilizam padrões podem estar repetindo erros que já tem solução conhecida.
e) *Flexibilização*. Uma das qualidades a ser buscada no desenvolvimento de software é a flexibilização para que se tenha mais facilidade para acomodar as inevitáveis mudanças nos requisitos. Infelizmente, essa qualidade não é obtida sem certo custo inicial. Mas muitas vezes este investimento compensa a economia que se obtém depois.
f) *Formalidade*. Como o software é um construto formal, aceita-se, via de regra, que deva ser especificado também de maneira formal. Existem técnicas que iniciam com descrições informais, mas à medida que caminham para o produto final, elas precisam ser cada vez mais formais. Outras técnicas já iniciam com documentos formais, de alta cerimônia e tendem a ser menos ambíguas, embora algumas vezes mais difíceis de compreender.
g) *Rastreabilidade*. O software é desenvolvido por um processo complexo no qual diferentes documentos e especificações são produzidos e, algumas vezes, derivados uns dos outros. É necessário manter um registro de *traços*, entre os diferentes

artefatos para que se saiba quando alterações feitas em um artefato se refletem em outros. Por exemplo, um requisito que muda tem efeito sobre vários outros artefatos como, modelos de classes, casos de uso, ou mesmo código executável.

h) *Desenvolvimento iterativo*. No início da era do computador, quando o software era mais simples, até se podia conceber o desenvolvimento como um processo de um único ciclo, com início e fim bem definidos. Mas com a crescente complexidade do software, cada vez mais as modernas técnicas apontam para o desenvolvimento iterativo como uma maneira de lidar com a complexidade do software. Assim, vários ciclos de desenvolvimento são realizados, cada um deles com um conjunto de objetivos distinto, e cada um deles contribuindo para a geração do produto final.
i) *Gerenciamento de requisitos*. Antigamente, acreditava-se que era possível descobrir quais eram os requisitos de um sistema e depois bastava desenvolver o melhor sistema possível que atendesse a estes requisitos. Modernamente, sabe-se que requisitos mudam com muita frequência, e é necessário assim gerenciar sua mudança como parte integrante do processo de desenvolvimento e evolução de software, não apenas como um mal necessário.
j) *Arquiteturas baseadas em componentes.* A componentização do software é uma das formas de se obter reusabilidade e também uma maneira de lidar melhor com a complexidade.
k) *Modelagem visual.* O princípio da abstração, por vezes exige que o software seja entendido de forma visual. A UML (*Unified Modeling Language*) é usada para representar visualmente várias características da arquitetura de um sistema. Cada vez mais se espera que a partir de modelos visuais seja possível gerar sistemas completos, sem passar pelas etapas mais clássicas de programação.
l) *Verificação contínua da qualidade.* A qualidade de um produto de software não é um requisito que possa ser obtido no final ou que possa ser facilmente melhorada ao final do desenvolvimento. A preocupação com a qualidade e sua garantia deve estar presentes ao longo de qualquer projeto de desenvolvimento.
m) *Controle de mudanças.* Não é mais possível gerenciar adequadamente um processo de desenvolvimento sem ter controle de versões de componentes do software e de sua evolução. Muitas vezes é necessário voltar atrás em um caminho de desenvolvimento ou ainda bancar o desenvolvimento em paralelo de diferentes versões de um mesmo componente. Sem um efetivo sistema de gerenciamento de configuração e mudança seria muito difícil, senão impossível, fazer esse controle de forma efetiva.
n) *Gerenciamento de riscos.* É sabido que uma das principais causas de fracasso de projetos de software é o fato de que os planejadores e gerentes podem não estar atentos aos riscos do projeto. Quando um risco se torna um problema, um gerente que já tenha um plano de ação certamente vai se sair melhor do que aquele que é pego de surpresa. Além disso, com gerenciamento de riscos é possível tomar ações preventivas de forma a reduzir a probabilidade ou impacto do risco.

Estes são apenas alguns princípios. Muitos outros poderiam ser adicionados. Mas isso será feito ao longo dos próximos capítulos quando estes princípios e muitos outros serão mencionados nos diversos tópicos relacionados à área de engenharia de software.

