CAPÍTULO

2

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26

Processos de software

Objetivos

O objetivo deste capítulo é apresentar a ideia de um processo de software — um conjunto coerente de atividades para a produção de software. Ao terminar de ler este capítulo, você:

- compreenderá os conceitos e modelos de processos de software;
- terá sido apresentado a três modelos genéricos de processos de software e quando eles podem ser usados;
- conhecerá as atividades fundamentais do processo de engenharia de requisitos de software, desenvolvimento de software, testes e evolução;
- entenderá por que os processos devem ser organizados de maneira a lidar com as mudanças nos requisitos e projeto de software;
- compreenderá como o Rational Unified Process integra boas práticas de engenharia de software para criar processos de software adaptáveis.

Conteúdo

Um processo de software é um conjunto de atividades relacionadas que levam à produção de um produto de software. Essas atividades podem envolver o desenvolvimento de software a partir do zero em uma linguagem padrão de programação como Java ou C. No entanto, aplicações de negócios não são necessariamente desenvolvidas dessa forma. Atualmente, novos softwares de negócios são desenvolvidos por meio da extensão e modificação de sistemas existentes ou por meio da configuração e integração de prateleira ou componentes do sistema.

Existem muitos processos de software diferentes, mas todos devem incluir quatro atividades fundamentais para a engenharia de software:

1. *Especificação de software.* A funcionalidade do software e as restrições a seu funcionamento devem ser definidas.
2. *Projeto e implementação de software.* O software deve ser produzido para atender às especificações.
3. *Validação de software.* O software deve ser validado para garantir que atenda às demandas do cliente.
4. *Evolução de software.* O software deve evoluir para atender às necessidades de mudança dos clientes.

De alguma forma, essas atividades fazem parte de todos os processos de software. Na prática, são atividades complexas em si mesmas, que incluem subatividades como validação de requisitos, projeto de arquitetura, testes unitários etc. Existem também as atividades que dão apoio ao processo, como documentação e gerenciamento de configuração de software.

Ao descrever e discutir os processos, costumamos falar sobre suas atividades, como a especificação de um modelo de dados, o projeto de interface de usuário etc., bem como a organização dessas atividades. No entanto, assim como as atividades, as descrições do processo também podem incluir:

1. Produtos, que são os resultados de uma das atividades do processo. Por exemplo, o resultado da atividade de projeto de arquitetura pode ser um modelo da arquitetura de software.
2. Papéis, que refletem as responsabilidades das pessoas envolvidas no processo. Exemplos de papéis são: gerente de projeto, gerente de configuração, programador etc.
3. Pré e pós-condições, que são declarações verdadeiras antes e depois de uma atividade do processo ou da produção de um produto. Por exemplo, antes do projeto de arquitetura ser iniciado, pode haver uma pré-condição de que todos os requisitos tenham sido aprovados pelo cliente e, após a conclusão dessa atividade, uma pós-condição poderia ser a de que os modelos UML que descrevem a arquitetura tenham sido revisados.

Os processos de software são complexos e, como todos os processos intelectuais e criativos, dependem de pessoas para tomar decisões e fazer julgamentos. Não existe um processo ideal, a maioria das organizações desenvolve os próprios processos de desenvolvimento de software. Os processos têm evoluído de maneira a tirarem melhor proveito das capacidades das pessoas em uma organização, bem como das características específicas do sistema em desenvolvimento. Para alguns sistemas, como sistemas críticos, é necessário um processo de desenvolvimento muito bem estruturado; para sistemas de negócios, com requisitos que se alteram rapidamente, provavelmente será mais eficaz um processo menos formal e mais flexível.

Os processos de software, às vezes, são categorizados como dirigidos a planos ou processos ágeis. Processos dirigidos a planos são aqueles em que todas as atividades são planejadas com antecedência, e o progresso é avaliado por comparação com o planejamento inicial. Em processos ágeis, que discuto no Capítulo 3, o planejamento é gradativo, e é mais fácil alterar o processo de maneira a refletir as necessidades de mudança dos clientes. Conforme Boehm e Turner (2003), cada abordagem é apropriada para diferentes tipos de software. Geralmente, é necessário encontrar um equilíbrio entre os processos dirigidos a planos e os processos ágeis.

Embora não exista um processo 'ideal' de software, há espaço, em muitas organizações, para melhorias no processo de software. Os processos podem incluir técnicas ultrapassadas ou não aproveitar as melhores práticas de engenharia de software da indústria. De fato, muitas empresas ainda não se aproveitam dos métodos da engenharia de software em seu desenvolvimento de software.

Em organizações nas quais a diversidade de processos de software é reduzida, os processos de software podem ser melhorados pela padronização. Isso possibilita uma melhor comunicação, além de redução no período de treinamento, e torna mais econômico o apoio ao processo automatizado. A padronização também é um importante primeiro passo na introdução de novos métodos e técnicas de engenharia de software, assim como as boas práticas de engenharia de software. No Capítulo 26, discuto mais detalhadamente a melhoria no processo de software.

2.1 Modelos de processo de software

Como expliquei no Capítulo 1, um modelo de processo de software é uma representação simplificada de um processo de software. Cada modelo representa uma perspectiva particular de um processo e, portanto, fornece informações parciais sobre ele. Por exemplo, um modelo de atividade do processo pode mostrar as atividades e sua sequência, mas não mostrar os papéis das pessoas envolvidas. Nesta seção, apresento uma série de modelos gerais de processos (algumas vezes, chamados 'paradigmas de processo') a partir de uma perspectiva de sua arquitetura. Ou seja, nós vemos um *framework* do processo, mas não vemos os detalhes de suas atividades específicas.

Esses modelos genéricos não são descrições definitivas dos processos de software. Pelo contrário, são abstrações que podem ser usadas para explicar diferentes abordagens de desenvolvimento de software. Você pode vê-los como *frameworks* de processos que podem ser ampliados e adaptados para criar processos de engenharia de software mais específicos.

Os modelos de processo que abordo aqui são:

1. *O modelo em cascata*. Esse modelo considera as atividades fundamentais do processo de especificação, desenvolvimento, validação e evolução, e representa cada uma delas como fases distintas, como: especificação de requisitos, projeto de software, implementação, teste e assim por diante.

2. *Desenvolvimento incremental.* Essa abordagem intercala as atividades de especificação, desenvolvimento e validação. O sistema é desenvolvido como uma série de versões (incrementos), de maneira que cada versão adiciona funcionalidade à anterior.
3. *Engenharia de software orientada a reúso.* Essa abordagem é baseada na existência de um número significativo de componentes reusáveis. O processo de desenvolvimento do sistema concentra-se na integração desses componentes em um sistema já existente em vez de desenvolver um sistema a partir do zero.

Esses modelos não são mutuamente exclusivos e muitas vezes são usados em conjunto, especialmente para o desenvolvimento de sistemas de grande porte. Para sistemas de grande porte, faz sentido combinar algumas das melhores características do modelo em cascata e dos modelos de desenvolvimento incremental. É preciso ter informações sobre os requisitos essenciais do sistema para projetar uma arquitetura de software que dê suporte a esses requisitos. Você não pode desenvolver isso incrementalmente. Os subsistemas dentro de um sistema maior podem ser desenvolvidos com diferentes abordagens. As partes do sistema que são bem compreendidas podem ser especificadas e desenvolvidas por meio de um processo baseado no modelo em cascata. As partes que são difíceis de especificar antecipadamente, como a interface com o usuário, devem sempre ser desenvolvidas por meio de uma abordagem incremental.

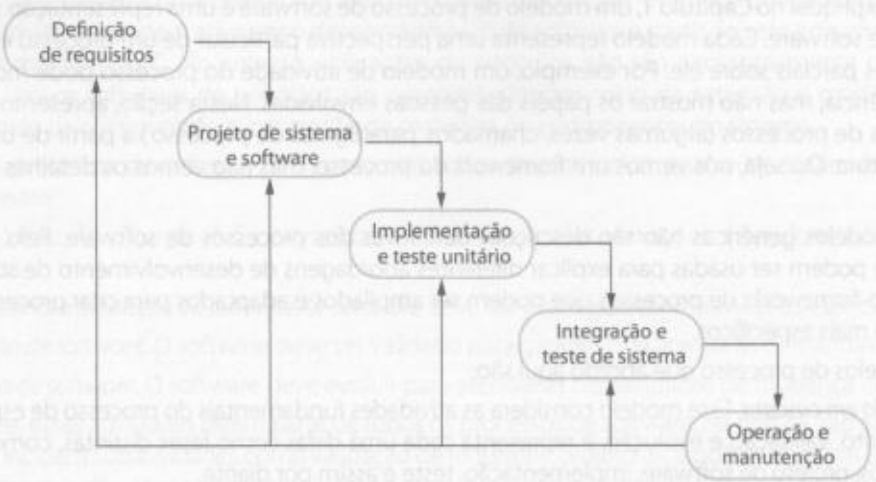
2.1.1 O modelo em cascata

O primeiro modelo do processo de desenvolvimento de software a ser publicado foi derivado de processos mais gerais da engenharia de sistemas (ROYCE, 1970). Esse modelo é ilustrado na Figura 2.1. Por causa do encadeamento entre uma fase e outra, esse modelo é conhecido como 'modelo em cascata', ou ciclo de vida de software. O modelo em cascata é um exemplo de um processo dirigido a planos — em princípio, você deve planejar e programar todas as atividades do processo antes de começar a trabalhar nelas.

Os principais estágios do modelo em cascata refletem diretamente as atividades fundamentais do desenvolvimento:

1. *Análise e definição de requisitos.* Os serviços, restrições e metas do sistema são estabelecidos por meio de consulta aos usuários. Em seguida, são definidos em detalhes e funcionam como uma especificação do sistema.
2. *Projeto de sistema e software.* O processo de projeto de sistemas aloca os requisitos tanto para sistemas de hardware como para sistemas de software, por meio da definição de uma arquitetura geral do sistema. O projeto de software envolve identificação e descrição das abstrações fundamentais do sistema de software e seus relacionamentos.
3. *Implementação e teste unitário.* Durante esse estágio, o projeto do software é desenvolvido como um conjunto de programas ou unidades de programa. O teste unitário envolve a verificação de que cada unidade atenda a sua especificação.

Figura 2.1 O modelo em cascata



4. *Integração e teste de sistema.* As unidades individuais do programa ou programas são integradas e testadas como um sistema completo para assegurar que os requisitos do software tenham sido atendidos. Após o teste, o sistema de software é entregue ao cliente.
5. *Operação e manutenção.* Normalmente (embora não necessariamente), essa é a fase mais longa do ciclo de vida. O sistema é instalado e colocado em uso. A manutenção envolve a correção de erros que não foram descobertos em estágios iniciais do ciclo de vida, com melhora da implementação das unidades do sistema e ampliação de seus serviços em resposta às descobertas de novos requisitos.

Em princípio, o resultado de cada estágio é a aprovação de um ou mais documentos ('assinados'). O estágio seguinte não deve ser iniciado até que a fase anterior seja concluída. Na prática, esses estágios se sobrepõem e alimentam uns aos outros de informações. Durante o projeto, os problemas com os requisitos são identificados; durante a codificação, problemas de projeto são encontrados e assim por diante. O processo de software não é um modelo linear simples, mas envolve o *feedback* de uma fase para outra. Assim, os documentos produzidos em cada fase podem ser modificados para refletirem as alterações feitas em cada um deles.

Por causa dos custos de produção e aprovação de documentos, as iterações podem ser dispendiosas e envolver significativo retrabalho. Assim, após um pequeno número de iterações, é normal se congelarem partes do desenvolvimento, como a especificação, e dar-se continuidade aos estágios posteriores de desenvolvimento. A solução dos problemas fica para mais tarde, ignorada ou programada, quando possível. Esse congelamento prematuro dos requisitos pode significar que o sistema não fará o que o usuário quer. Também pode levar a sistemas mal estruturados, quando os problemas de projeto são contornados por artifícios de implementação.

Durante o estágio final do ciclo de vida (operação e manutenção), o software é colocado em uso. Erros e omissões nos requisitos originais do software são descobertos. Os erros de programa e projeto aparecem e são identificadas novas necessidades funcionais. O sistema deve evoluir para permanecer útil. Fazer essas alterações (manutenção do software) pode implicar repetição de estágios anteriores do processo.

O modelo em cascata é consistente com outros modelos de processos de engenharia, e a documentação é produzida em cada fase do ciclo. Dessa forma, o processo torna-se visível, e os gerentes podem monitorar o progresso de acordo com o plano de desenvolvimento. Seu maior problema é a divisão inflexível do projeto em estágios distintos. Os compromissos devem ser assumidos em um estágio inicial do processo, o que dificulta que atendam às mudanças de requisitos dos clientes.

Em princípio, o modelo em cascata deve ser usado apenas quando os requisitos são bem compreendidos e pouco provavelmente venham a ser radicalmente alterados durante o desenvolvimento do sistema. No entanto, o modelo em cascata reflete o tipo de processo usado em outros projetos de engenharia. Como é mais fácil usar um modelo de gerenciamento comum para todo o projeto, processos de software baseados no modelo em cascata ainda são comumente utilizados.

Uma variação importante do modelo em cascata é o desenvolvimento formal de um sistema, em que se cria um modelo matemático de uma especificação do sistema. Esse modelo é então refinado, usando transformações matemáticas que preservam sua consistência, em código executável. Partindo do pressuposto de que suas transformações matemáticas estão corretas, você pode, portanto, usar um forte argumento de que um programa gerado dessa forma é consistente com suas especificações.

Processos formais de desenvolvimento, como os baseados no método B (SCHNEIDER, 2001; WORDSWORTH, 1996), são particularmente adequados para o desenvolvimento de sistemas com requisitos rigorosos de segurança, confiabilidade e proteção. A abordagem formal simplifica a produção de casos de segurança ou proteção. Isso demonstra aos clientes ou reguladores que o sistema realmente cumpre com seus requisitos de proteção ou segurança.

Processos baseados em transformações formais são geralmente usados apenas no desenvolvimento de sistemas críticos de segurança ou de proteção. Eles exigem conhecimentos especializados. Para a maioria dos sistemas, esse processo não oferece custo-benefício significativo sobre outras abordagens para o desenvolvimento de sistemas.

2.1.2 Desenvolvimento incremental

O desenvolvimento incremental é baseado na ideia de desenvolver uma implementação inicial, expô-la aos comentários dos usuários e continuar por meio da criação de várias versões até que um sistema adequado seja desenvolvido (Figura 2.2). Atividades de especificação, desenvolvimento e validação são intercaladas, e não separadas, com rápido *feedback* entre todas as atividades.

Figura 2.2 Desenvolvimento incremental

Desenvolvimento incremental de software, que é uma parte fundamental das abordagens ágeis, é melhor do que uma abordagem em cascata para a maioria dos sistemas de negócios, *e-commerce* e sistemas pessoais. Desenvolvimento incremental reflete a maneira como resolvemos os problemas. Raramente elaboramos uma completa solução do problema com antecedência; geralmente movemo-nos passo a passo em direção a uma solução, recuando quando percebemos que cometemos um erro. Ao desenvolver um software de forma incremental, é mais barato e mais fácil fazer mudanças no software durante seu desenvolvimento.

Cada incremento ou versão do sistema incorpora alguma funcionalidade necessária para o cliente. Frequentemente, os incrementos iniciais incluem a funcionalidade mais importante ou mais urgente. Isso significa que o cliente pode avaliar o sistema em um estágio relativamente inicial do desenvolvimento para ver se ele oferece o que foi requisitado. Em caso negativo, só o incremento que estiver em denvolvimento no momento precisará ser alterado e, possivelmente, nova funcionalidade deverá ser definida para incrementos posteriores.

O desenvolvimento incremental tem três vantagens importantes quando comparado ao modelo em cascata:

1. O custo de acomodar as mudanças nos requisitos do cliente é reduzido. A quantidade de análise e documentação a ser refeita é muito menor do que o necessário no modelo em cascata.
2. É mais fácil obter *feedback* dos clientes sobre o desenvolvimento que foi feito. Os clientes podem fazer comentários sobre as demonstrações do software e ver o quanto foi implementado. Os clientes têm dificuldade em avaliar a evolução por meio de documentos de projeto de software.
3. É possível obter entrega e implementação rápida de um software útil ao cliente, mesmo se toda a funcionalidade não for incluída. Os clientes podem usar e obter ganhos a partir do software inicial antes do que é possível com um processo em cascata.

O desenvolvimento incremental, atualmente, é a abordagem mais comum para o desenvolvimento de sistemas aplicativos. Essa abordagem pode ser tanto dirigida a planos, ágil, ou, o mais comum, uma mescla dessas abordagens. Em uma abordagem dirigida a planos, os incrementos do sistema são identificados previamente; se uma abordagem ágil for adotada, os incrementos iniciais são identificados, mas o desenvolvimento de incrementos posteriores depende do progresso e das prioridades dos clientes.

Do ponto de vista do gerenciamento, a abordagem incremental tem dois problemas:

1. O processo não é visível. Os gerentes precisam de entregas regulares para mensurar o progresso. Se os sistemas são desenvolvidos com rapidez, não é economicamente viável produzir documentos que reflitam cada uma das versões do sistema.
2. A estrutura do sistema tende a se degradar com a adição dos novos incrementos. A menos que tempo e dinheiro sejam dispendidos em refatoração para melhoria do software, as constantes mudanças tendem a corromper sua estrutura. Incorporar futuras mudanças do software torna-se cada vez mais difícil e oneroso.

Os problemas do desenvolvimento incremental são particularmente críticos para os sistemas de vida-longa, grandes e complexos, nos quais várias equipes desenvolvem diferentes partes do sistema. Sistemas de grande porte necessitam de um *framework* ou arquitetura estável, e as responsabilidades das diferentes equipes de trabalho do sistema precisam ser claramente definidas, respeitando essa arquitetura. Isso deve ser planejado com antecedência, e não desenvolvido de forma incremental.

Você pode desenvolver um sistema de forma incremental e expô-lo aos comentários dos clientes, sem realmente entregá-lo e implantá-lo no ambiente do cliente. Entrega e implantação incremental significa que o software é usado em processos operacionais reais. Isso nem sempre é possível, pois experimentações com o novo software podem interromper os processos normais de negócios. As vantagens e desvantagens da entrega incremental são discutidas na Seção 2.3.2.

2.1.3 Engenharia de software orientada a reúso

Na maioria dos projetos de software, há algum reúso de software. Isso acontece muitas vezes informalmente, quando as pessoas envolvidas no projeto sabem de projetos ou códigos semelhantes ao que é exigido. Elas os buscam, fazem as modificações necessárias e incorporam-nos a seus sistemas.

Esse reúso informal ocorre independentemente do processo de desenvolvimento que se use. No entanto, no século XXI, processos de desenvolvimento de software com foco no reúso de software existente tornaram-se amplamente usados. Abordagens orientadas a reúso dependem de uma ampla base de componentes reusáveis de software e de um *framework* de integração para a composição desses componentes. Em alguns casos, esses componentes são sistemas completos (COTS ou de prateleira), capazes de fornecer uma funcionalidade específica, como processamento de texto ou planilha.

Um modelo de processo geral de desenvolvimento baseado no reúso está na Figura 2.3. Embora o estágio de especificação de requisitos iniciais e o estágio de validação sejam comparáveis a outros processos de software, os estágios intermediários em um processo orientado a reúso são diferentes. Esses estágios são:

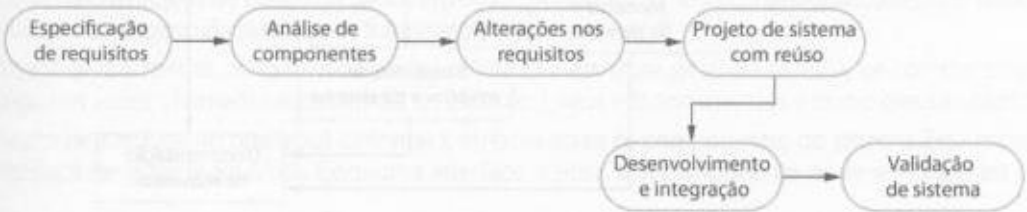
1. *Análise de componentes*. Dada a especificação de requisitos, é feita uma busca por componentes para implementar essa especificação. Em geral, não há correspondência exata, e os componentes que podem ser usados apenas fornecem alguma funcionalidade necessária.
2. *Modificação de requisitos*. Durante esse estágio, os requisitos são analisados usando-se informações sobre os componentes que foram descobertos. Em seguida, estes serão modificados para refletir os componentes disponíveis. No caso de modificações impossíveis, a atividade de análise dos componentes pode ser reinserida na busca por soluções alternativas.
3. *Projeto do sistema com reúso*. Durante esse estágio, o *framework* do sistema é projetado ou algo existente é reusado. Os projetistas têm em mente os componentes que serão reusados e organizam o *framework* para reúso. Alguns softwares novos podem ser necessários, se componentes reusáveis não estiverem disponíveis.
4. *Desenvolvimento e integração*. Softwares que não podem ser adquiridos externamente são desenvolvidos, e os componentes e sistemas COTS são integrados para criar o novo sistema. A integração de sistemas, nesse modelo, pode ser parte do processo de desenvolvimento, em vez de uma atividade separada.

Existem três tipos de componentes de software que podem ser usados em um processo orientado a reúso:

1. *Web services* desenvolvidos de acordo com os padrões de serviço e que estão disponíveis para invocação remota.
2. Coleções de objetos que são desenvolvidas como um pacote a ser integrado com um *framework* de componentes, como .NET ou J2EE.
3. Sistemas de software *stand-alone* configurados para uso em um ambiente particular.

Engenharia de software orientada a reúso tem a vantagem óbvia de reduzir a quantidade de software a ser desenvolvido e, assim, reduzir os custos e riscos. Geralmente, também proporciona a entrega mais rápida do software. No entanto, compromisos com os requisitos são inevitáveis, e isso pode levar a um sistema que não

Figura 2.3 Engenharia de software orientada a reúso



atende às reais necessidades dos usuários. Além disso, algum controle sobre a evolução do sistema é perdido, pois as novas versões dos componentes reusáveis não estão sob o controle da organização que os está utilizando.

Reúso de software é um tema muito importante, ao qual dediquei vários capítulos na terceira parte deste livro. Questões gerais de reúso de software e reúso de COTS serão abordadas no Capítulo 16; a engenharia de software baseada em componentes, nos capítulos 17 e 18; e sistemas orientados a serviços, no Capítulo 19.

2.2 Atividades do processo

Processos reais de software são intercalados com sequências de atividades técnicas, de colaboração e de gerência, com o intuito de especificar, projetar, implementar e testar um sistema de software. Os desenvolvedores de software usam uma variedade de diferentes ferramentas de software em seu trabalho. As ferramentas são especialmente úteis para apoiar a edição de diferentes tipos de documentos e para gerenciar o imenso volume de informações detalhadas que é gerado em um projeto de grande porte.

As quatro atividades básicas do processo — especificação, desenvolvimento, validação e evolução — são organizadas de forma diferente conforme o processo de desenvolvimento. No modelo em cascata são organizadas em sequência, enquanto que no desenvolvimento incremental são intercaladas. A maneira como essas atividades serão feitas depende do tipo de software, das pessoas e das estruturas organizacionais envolvidas. Em *extreme programming*, por exemplo, as especificações estão escritas em cartões. Testes são executáveis e desenvolvidos antes do próprio programa. A evolução pode demandar reestruturação substancial do sistema ou refatoração.

2.2.1 Especificação de software

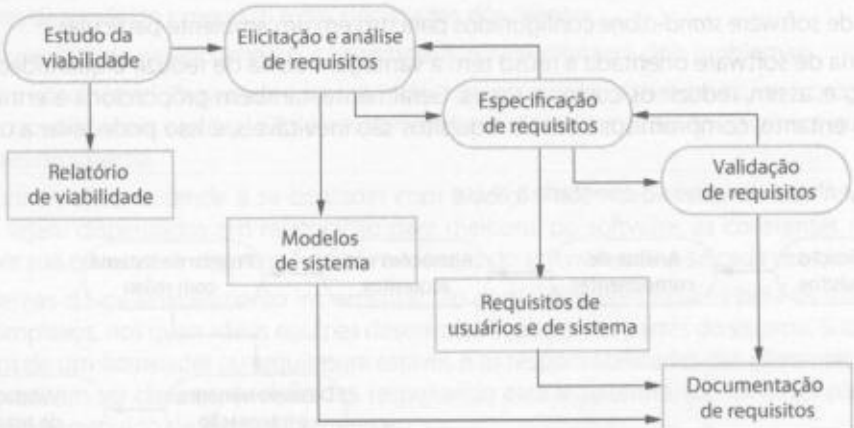
Especificação de software ou engenharia de requisitos é o processo de compreensão e definição dos serviços requisitados do sistema e identificação de restrições relativas à operação e ao desenvolvimento do sistema. A engenharia de requisitos é um estágio particularmente crítico do processo de software, pois erros nessa fase inevitavelmente geram problemas no projeto e na implementação do sistema.

O processo de engenharia de requisitos (Figura 2.4) tem como objetivo produzir um documento de requisitos acordados que especifica um sistema que satisfaz os requisitos dos *stakeholders*. Requisitos são geralmente apresentados em dois níveis de detalhe. Os usuários finais e os clientes precisam de uma declaração de requisitos em alto nível; desenvolvedores de sistemas precisam de uma especificação mais detalhada do sistema.

Existem quatro atividades principais do processo de engenharia de requisitos:

1. *Estudo de viabilidade.* É feita uma estimativa acerca da possibilidade de se satisfazerem as necessidades do usuário identificado usando-se tecnologias atuais de software e hardware. O estudo considera se o sistema

Figura 2.4 Os requisitos da engenharia de processos



proposto será rentável a partir de um ponto de vista de negócio e se ele pode ser desenvolvido no âmbito das atuais restrições orçamentais. Um estudo de viabilidade deve ser relativamente barato e rápido. O resultado deve informar a decisão de avançar ou não, com uma análise mais detalhada.

2. *Elicitação e análise de requisitos.* Esse é o processo de derivação dos requisitos do sistema por meio da observação dos sistemas existentes, além de discussões com os potenciais usuários e compradores, análise de tarefas, entre outras etapas. Essa parte do processo pode envolver o desenvolvimento de um ou mais modelos de sistemas e protótipos, os quais nos ajudam a entender o sistema a ser especificado.
3. *Especificação de requisitos.* É a atividade de traduzir as informações obtidas durante a atividade de análise em um documento que defina um conjunto de requisitos. Dois tipos de requisitos podem ser incluídos nesse documento. Requisitos do usuário são declarações abstratas dos requisitos do sistema para o cliente e usuário final do sistema; requisitos de sistema são uma descrição mais detalhada da funcionalidade a ser provida.
4. *A validação de requisitos.* Essa atividade verifica os requisitos quanto a realismo, consistência e completude. Durante esse processo, os erros no documento de requisitos são inevitavelmente descobertos. Em seguida, o documento deve ser modificado para correção desses problemas.

Naturalmente, as atividades no processo de requisitos não são feitas em apenas uma sequência. A análise de requisitos continua durante a definição e especificação, e novos requisitos emergem durante o processo. Portanto, as atividades de análise, definição e especificação são intercaladas. Nos métodos ágeis, como *extreme programming*, os requisitos são desenvolvidos de forma incremental, de acordo com as prioridades do usuário, e a elicitação de requisitos é feita pelos usuários que integram equipe de desenvolvimento.

2.2.2 Projeto e implementação de software

O estágio de implementação do desenvolvimento de software é o processo de conversão de uma especificação do sistema em um sistema executável. Sempre envolve processos de projeto e programação de software, mas, se for usada uma abordagem incremental para o desenvolvimento, também pode envolver o refinamento da especificação do software.

Um projeto de software é uma descrição da estrutura do software a ser implementado, dos modelos e estruturas de dados usados pelo sistema, das interfaces entre os componentes do sistema e, às vezes, dos algoritmos usados. Os projetistas não chegam a um projeto final imediatamente, mas desenvolvem-no de forma iterativa. Eles acrescentam formalidade e detalhes, enquanto desenvolvem seu projeto por meio de revisões constantes para correção de projetos anteriores.

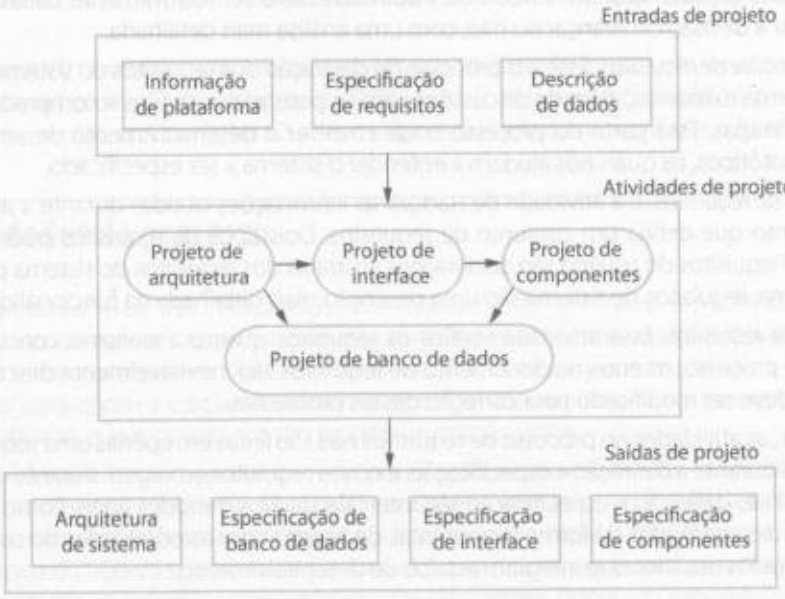
A Figura 2.5 é um modelo abstrato de processo que mostra as entradas para o processo de projeto, suas atividades e os documentos produzidos como saídas dele. O diagrama sugere que os estágios do processo de projeto são sequenciais. Na realidade, as atividades do processo são intercaladas. *Feedback* de um estágio para outro e consequente retrabalho são inevitáveis em todos os processos.

A maioria dos softwares interage com outros sistemas de software, incluindo o sistema operacional, o banco de dados, o *middleware* e outros aplicativos. Estes formam a 'plataforma de software', o ambiente em que o software será executado. Informações sobre essa plataforma são entradas essenciais para o processo de projeto, pois os projetistas devem decidir a melhor forma de integrá-la ao ambiente do software. A especificação de requisitos é uma descrição da funcionalidade que o software deve oferecer, e seus requisitos de desempenho e confiança. Se o sistema for para processamento de dados existentes, a descrição desses dados poderia ser incluída na especificação da plataforma; caso contrário, a descrição dos dados deve ser uma entrada para o processo de projeto, para que a organização dos dados do sistema seja definida.

As atividades no processo de projeto podem variar, dependendo do tipo de sistema a ser desenvolvido. Por exemplo, sistemas de tempo real demandam projeto de *timing*, mas podem não incluir uma base de dados; nesse caso, não há um projeto de banco de dados envolvido. A Figura 2.5 mostra quatro atividades que podem ser parte do processo de projeto de sistemas de informação:

1. *Projeto de arquitetura*, no qual você pode identificar a estrutura geral do sistema, os componentes principais (algumas vezes, chamados subsistemas ou módulos), seus relacionamentos e como eles são distribuídos.
2. *Projeto de interface*, no qual você define as interfaces entre os componentes do sistema. Essa especificação de interface deve ser inequívoca. Com uma interface precisa, um componente pode ser usado de maneira que

Figura 2.5 Um modelo geral do processo de projeto



outros componentes não precisam saber como ele é implementado. Uma vez que as especificações de interface são acordadas, os componentes podem ser projetados e desenvolvidos simultaneamente.

3. *Projeto de componente*, no qual você toma cada componente do sistema e projeta seu funcionamento. Pode-se tratar de uma simples declaração da funcionalidade que se espera implementar, com o projeto específico para cada programador. Pode, também, ser uma lista de alterações a serem feitas em um componente reusável ou um modelo de projeto detalhado. O modelo de projeto pode ser usado para gerar automaticamente uma implementação.
4. *Projeto de banco de dados*, no qual você projeta as estruturas de dados do sistema e como eles devem ser representados em um banco de dados. Novamente, o trabalho aqui depende da existência de um banco de dados a ser reusado ou da criação de um novo banco de dados.

Essas atividades conduzem a um conjunto de saídas do projeto, que também é mostrado na Figura 2.5. O detalhe e a apresentação de cada uma varia consideravelmente. Para sistemas críticos, devem ser produzidos documentos detalhados de projeto, indicando as descrições precisas e exatas do sistema. Se uma abordagem dirigida a modelos é usada, essas saídas podem ser majoritariamente diagramas. Quando os métodos ágeis de desenvolvimento são usados, as saídas do processo de projeto podem não ser documentos de especificação separado, mas ser representadas no código do programa.

Métodos estruturados para projeto foram desenvolvidos nas décadas de 1970 e 1980, e foram os precursores da UML e do projeto orientado a objetos (BUDGEN, 2003). Eles estão relacionados com a produção de modelos gráficos do sistema e, em muitos casos, geram códigos automaticamente a partir desses modelos. Desenvolvimento dirigido a modelos (MDD, do inglês *model-driven development*) ou engenharia dirigida a modelos (SCHMIDT, 2006), em que os modelos de software são criados em diferentes níveis de abstração, é uma evolução dos métodos estruturados. Em MDD, há uma ênfase maior nos modelos de arquitetura com uma separação entre os modelos abstratos independentes de implementação e específicos de implementação. Os modelos são desenvolvidos em detalhes suficientes para que o sistema executável possa ser gerado a partir deles. Discuto essa abordagem para o desenvolvimento no Capítulo 5.

O desenvolvimento de um programa para implementar o sistema decorre naturalmente dos processos de projeto de sistema. Apesar de algumas classes de programa, como sistemas críticos de segurança, serem normalmente projetadas em detalhe antes de se iniciar qualquer implementação, é mais comum os estágios posteriores de projeto e desenvolvimento de programa serem intercalados. Ferramentas de desenvolvimento de software podem ser usadas para gerar um esqueleto de um programa a partir do projeto. Isso inclui o código para definir e implementar interfaces e, em muitos casos, o desenvolvedor precisa apenas acrescentar detalhes da operação de cada componente do programa.

Programação é uma atividade pessoal, não existe um processo geral a ser seguido. Alguns programadores começam com componentes que eles compreendem, desenvolvem-nos e depois passam para os componentes menos compreendidos. Outros preferem a abordagem oposta, deixando para o fim os componentes familiares, pois sabem como desenvolvê-los. Alguns desenvolvedores preferem definir os dados no início do processo e, em seguida, usam essa definição para dirigir o desenvolvimento do programa; outros deixam os dados não especificados durante o maior período de tempo possível.

Geralmente, os programadores fazem alguns testes do código que estão desenvolvendo, o que, muitas vezes, revela defeitos que devem ser retirados do programa. Isso é chamado *debugging*. Testes de defeitos e *debugging* são processos diferentes. Testes estabelecem a existência de defeitos; *debugging* diz respeito à localização e correção desses defeitos.

Quando você está realizando *debugging*, precisa gerar hipóteses sobre o comportamento observável do programa e, em seguida, testar essas hipóteses, na esperança de encontrar um defeito que tenha causado uma saída anormal. O teste das hipóteses pode envolver o rastreio manual do código do programa, bem como exigir novos casos de teste para localização do problema. Ferramentas interativas de depuração, que mostram os valores intermediários das variáveis do programa e uma lista das instruções executadas, podem ser usadas para apoiar o processo de depuração.

2.2.3 Validação de software

Validação de software ou, mais genericamente, verificação e validação (V&V), tem a intenção de mostrar que um software se adequa a suas especificações ao mesmo tempo que satisfaz as especificações do cliente do sistema. Teste de programa, em que o sistema é executado com dados de testes simulados, é a principal técnica de validação. A validação também pode envolver processos de verificação, como inspeções e revisões, em cada estágio do processo de software, desde a definição dos requisitos de usuários até o desenvolvimento do programa. Devido à predominância dos testes, a maior parte dos custos de validação incorre durante e após a implementação.

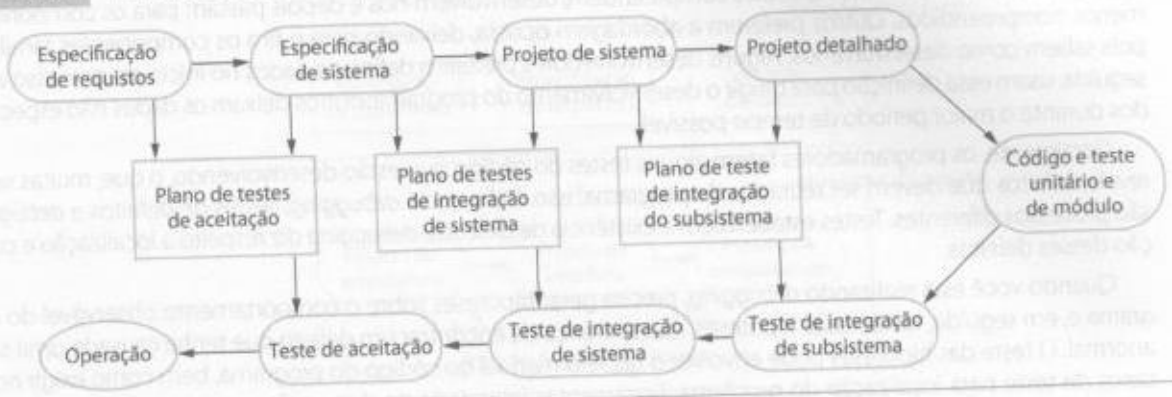
Com exceção de pequenos programas, sistemas não devem ser testados como uma unidade única e monolítica. A Figura 2.6 mostra um processo de teste, de três estágios, nos quais os componentes do sistema são testados, em seguida, o sistema integrado é testado e, finalmente, o sistema é testado com os dados do cliente. Idealmente, os defeitos de componentes são descobertos no início do processo, e os problemas de interface são encontrados quando o sistema é integrado. No entanto, quando os defeitos são descobertos, o programa deve ser depurado, e isso pode requerer que outros estágios do processo de testes sejam repetidos. Erros em componentes de programa podem vir à luz durante os testes de sistema. O proceso é, portanto, iterativo, com informações realimentadas de estágios posteriores para partes anteriores do processo.

Os estágios do processo de teste são:

1. *Testes de desenvolvimento*. Os componentes do sistema são testados pelas pessoas que o desenvolveram. Cada componente é testado de forma independente, separado dos outros. Os componentes podem ser entidades simples, como funções ou classes de objetos, ou podem ser agrupamentos coerentes dessas entidades. Ferramentas de automação de teste, como JUnit (MASSOL e HUSTED, 2003), que podem reexecutar testes de componentes quando as novas versões dos componentes são criadas, são comumente usadas.
2. *Testes de sistema*. Componentes do sistema são integrados para criar um sistema completo. Esse processo se preocupa em encontrar os erros resultantes das interações inesperadas entre componentes e problemas de interface do componente. Também visa mostrar que o sistema satisfaz seus requisitos funcionais e não funcionais, bem como testar as propriedades emergentes do sistema. Para sistemas de grande porte, esse pode ser

Figura 2.6 Estágios de testes

Figura 2.7 Fases de testes de um processo de software dirigido a planos



um processo multiestágios, no qual os componentes são integrados para formar subsistemas individualmente testados antes de serem integrados para formar o sistema final.

3. *Testes de aceitação*. Esse é o estágio final do processo de testes, antes que o sistema seja aceito para uso operacional. O sistema é testado com dados fornecidos pelo cliente, e não com dados advindos de testes simulados. O teste de aceitação pode revelar erros e omissões na definição dos requisitos do sistema, pois dados reais exercitam o sistema de formas diferentes dos dados de teste. Os testes de aceitação também podem revelar problemas de requisitos em que os recursos do sistema não atendam às necessidades do usuário ou o desempenho do sistema seja inaceitável.

Os processos de desenvolvimento de componentes e testes geralmente são intercalados. Os programadores criam seus próprios dados para testes e, incrementalmente, testam o código enquanto ele é desenvolvido. Essa é uma abordagem economicamente sensível, pois o programador conhece o componente e, portanto, é a melhor pessoa para gerar casos de teste.

Se uma abordagem incremental é usada para o desenvolvimento, cada incremento deve ser testado enquanto é desenvolvido — sendo que esses testes devem ser baseados nos requisitos para esse incremento. Em *extreme programming*, os testes são desenvolvidos paralelamente aos requisitos, antes de se iniciar o desenvolvimento, o que ajuda testadores e desenvolvedores a compreender os requisitos e garante o cumprimento dos prazos enquanto são criados os casos de teste.

Quando um processo de software dirigido a planos é usado (por exemplo, para o desenvolvimento de sistemas críticos), o teste é impulsionado por um conjunto de planos de testes. Uma equipe independente de testadores trabalha a partir desses planos de teste pré-formulados, que foram desenvolvidos a partir das especificações e do projeto do sistema. A Figura 2.7 ilustra como os planos de teste são o elo entre as atividades de teste e de desenvolvimento. Esse modelo é, às vezes, chamado modelo V de desenvolvimento (gire a figura de lado para ver o V).

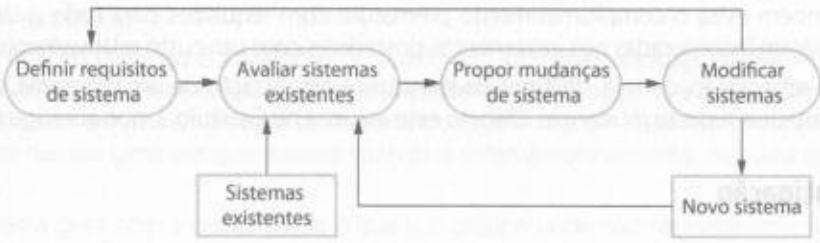
O teste de aceitação também pode ser chamado 'teste alfa'. Sistemas sob encomenda são desenvolvidos para um único cliente. O processo de testes-alfa continua até que o desenvolvedor do sistema e o cliente concordem que o sistema entregue é uma implementação aceitável dos requisitos.

Quando um sistema está pronto para ser comercializado como um produto de software, costuma-se usar um processo de testes denominado 'teste beta'. Este teste envolve a entrega de um sistema a um número de potenciais clientes que concordaram em usá-lo. Eles relatam problemas para os desenvolvedores dos sistemas. O produto é exposto para uso real, e erros que podem não ter sido antecipados pelos construtores do sistema são detectados. Após esse *feedback*, o sistema é modificado e liberado para outros testes-beta ou para venda em geral.

2.2.4 Evolução do software

A flexibilidade dos sistemas de software é uma das principais razões pelas quais os softwares vêm sendo, cada vez mais, incorporados em sistemas grandes e complexos. Uma vez que a decisão pela fabricação do hardware foi tomada, é muito caro fazer alterações em seu projeto. Entretanto, as mudanças no software podem ser feitas a qualquer momento durante ou após o desenvolvimento do sistema. Mesmo grandes mudanças são muito mais baratas do que as correspondentes alterações no hardware do sistema.

Figura 2.8 Evolução do sistema



Historicamente, sempre houve uma separação entre o processo de desenvolvimento e o de evolução do software (manutenção de software). As pessoas pensam no desenvolvimento de software como uma atividade criativa em que um sistema é desenvolvido a partir de um conceito inicial até um sistema funcional. Por outro lado, pensam na manutenção do software como maçante e desinteressante. Embora os custos de manutenção sejam frequentemente mais altos que os custos iniciais de desenvolvimento, os processos de manutenção são, em alguns casos, considerados menos desafiadores do que o desenvolvimento do software original.

Essa distinção entre o desenvolvimento e a manutenção é cada vez mais irrelevante. Poucos sistemas de software são completamente novos, e faz muito mais sentido ver o desenvolvimento e a manutenção como processos contínuos. Em vez de dois processos separados, é mais realista pensar na engenharia de software como um processo evolutivo (Figura 2.8), no qual o software é constantemente alterado durante seu período de vida em resposta às mudanças de requisitos e às necessidades do cliente.

2.3 Lidando com mudanças

A mudança é inevitável em todos os grandes projetos de software. Os requisitos do sistema mudam, ao mesmo tempo que o negócio que adquiriu o sistema responde a pressões externas e mudam as prioridades de gerenciamento. Com a disponibilidade de novas tecnologias, emergem novos projetos e possibilidades de implementação. Portanto, qualquer que seja o modelo do software de processo, é essencial que possa acomodar mudanças no software em desenvolvimento.

A mudança aumenta os custos de desenvolvimento de software, porque geralmente significa que o trabalho deve ser refeito. Isso é chamado retrabalho. Por exemplo, se os relacionamentos entre os requisitos do sistema foram analisados e novos requisitos foram identificados, alguma ou toda análise de requisitos deve ser repetida. Pode, então, ser necessário reprojetar o sistema de acordo com os novos requisitos, mudar qualquer programa que tenha sido desenvolvido e testar novamente o sistema.

Existem duas abordagens que podem ser adotadas para a redução de custos de retrabalho:

1. Prevenção de mudanças, em que o processo de software inclui atividades capazes de antecipar as mudanças possíveis antes que seja necessário qualquer retrabalho. Por exemplo, um protótipo de sistema pode ser desenvolvido para mostrar algumas características-chave do sistema para os clientes. Eles podem experimentar o protótipo e refinar seus requisitos antes de se comprometer com elevados custos de produção de software.
2. Tolerância a mudanças, em que o processo foi projetado para que as mudanças possam ser acomodadas a um custo relativamente baixo. Isso normalmente envolve alguma forma de desenvolvimento incremental. As alterações propostas podem ser aplicadas em incrementos que ainda não foram desenvolvidos. Se isso for impossível, então apenas um incremento (uma pequena parte do sistema) deve ser alterado para incorporar as mudanças.

Nesta seção, discuto duas maneiras de lidar com mudanças e mudanças nos requisitos do sistema. São elas:

1. Prototipação de sistema, em que uma versão do sistema ou de parte dele é desenvolvida rapidamente para verificar as necessidades do cliente e a viabilidade de algumas decisões de projeto. Esse processo previne mudanças, já que permite aos usuários experimentarem o sistema antes da entrega e, então, refinarem seus requisitos. O número de propostas de mudanças de requisitos a ser feito após a entrega é, portanto, suscetível de ser reduzido.

2. Entrega incremental, em que incrementos do sistema são entregues aos clientes para comentários e experimentação. Essa abordagem dá suporte tanto para a prevenção de mudanças quanto para a tolerância a mudanças. Também evita o comprometimento prematuro com requisitos para todo o sistema e permite que mudanças sejam incorporadas nos incrementos posteriores com um custo relativamente baixo.

A noção de refatoração, ou seja, melhoria da estrutura e organização de um programa, também é um importante mecanismo que suporta mudanças. Discuto esse assunto no Capítulo 3, que abrange métodos ágeis.

2 3.1 Prototipação

Um protótipo é uma versão inicial de um sistema de software, usado para demonstrar conceitos, experimentar opções de projeto e descobrir mais sobre o problema e suas possíveis soluções. O desenvolvimento rápido e iterativo do protótipo é essencial para que os custos sejam controlados e os *stakeholders* do sistema possam experimentá-lo no início do processo de software.

Um protótipo de software pode ser usado em um processo de desenvolvimento de software para ajudar a antecipar as mudanças que podem ser requisitadas:

1. No processo de engenharia de requisitos, um protótipo pode ajudar na elicitação e validação de requisitos de sistema.
2. No processo de projeto de sistema, um protótipo pode ser usado para estudar soluções específicas do software e para apoiar o projeto de interface de usuário.

Protótipos do sistema permitem aos usuários ver quão bem o sistema dá suporte a seu trabalho. Eles podem obter novas ideias para requisitos e encontrar pontos fortes e fracos do software; podem, então, propor novos requisitos do sistema. Além disso, o desenvolvimento do protótipo pode revelar erros e omissões nos requisitos propostos. A função descrita em uma especificação pode parecer útil e bem definida. No entanto, quando essa função é combinada com outras, os usuários muitas vezes percebem que sua visão inicial foi incorreta ou incompleta. A especificação do sistema pode então ser modificada para refletir o entendimento dos requisitos alterados.

Enquanto o sistema está em projeto, um protótipo do sistema pode ser usado para a realização de experimentos de projeto visando à verificação da viabilidade da proposta. Por exemplo, um projeto de banco de dados pode ser prototipado e testado para verificar se suporta de modo eficiente o acesso aos dados para as consultas mais comuns dos usuários. Prototipação também é uma parte essencial do processo de projeto da interface de usuário. Devido à natureza dinâmica de tais interfaces, descrições textuais e diagramas não são bons o suficiente para expressar seus requisitos. Portanto, a prototipação rápida com envolvimento do usuário final é a única maneira sensata de desenvolver interfaces gráficas de usuário para sistemas de software.

Um modelo de processo para desenvolvimento de protótipos é a Figura 2.9. Os objetivos da prototipação devem ser explicitados desde o início do processo. Estes podem ser o desenvolvimento de um sistema para prototipar a interface de usuário, o desenvolvimento de um sistema para validação dos requisitos funcionais do sistema ou o desenvolvimento de um sistema para demonstrar aos gerentes a viabilidade da aplicação. O mesmo protótipo não pode cumprir todos os objetivos. Se os objetivos não são declarados, a gerência ou os usuários finais podem não entender a função do protótipo. Consequentemente, eles podem não obter os benefícios que esperavam do desenvolvimento do protótipo.

O próximo estágio do processo é decidir o que colocar e, talvez mais importante ainda, o que deixar de fora do sistema de protótipo. Para reduzir os custos de prototipação e acelerar o cronograma de entrega, pode-se deixar alguma funcionalidade fora do protótipo. Você pode optar por relaxar os requisitos não funcionais, como tempo

Figura 2.9 O processo de desenvolvimento de protótipo

de resposta e utilização de memória. Gerenciamento e tratamento de erros podem ser ignorados, a menos que o objetivo do protótipo seja estabelecer uma interface de usuário. Padrões de confiabilidade e qualidade de programa podem ser reduzidos.

O estágio final do processo é a avaliação do protótipo. Durante esse estágio, provisões devem ser feitas para o treinamento do usuário, e os objetivos do protótipo devem ser usados para derivar um plano de avaliação. Os usuários necessitam de um tempo para se sentir confortáveis com um sistema novo e para se situarem em um padrão normal de uso. Uma vez que estejam usando o sistema normalmente, eles descobrem erros e omissões de requisitos.

Um problema geral com a prototipação é que o protótipo pode não ser necessariamente usado da mesma forma como o sistema final. O testador do protótipo pode não ser um usuário típico do sistema ou o tempo de treinamento durante a avaliação do protótipo pode ter sido insuficiente, por exemplo. Se o protótipo é lento, os avaliadores podem ajustar seu modo de trabalho e evitar os recursos do sistema que têm tempos de resposta lentos. Quando equipados com melhores respostas no sistema final, eles podem usá-lo de forma diferente.

Às vezes, os desenvolvedores são pressionados pelos gerentes para entregar protótipos descartáveis, especialmente quando há atrasos na entrega da versão final do software. No entanto, isso costuma ser desaconselhável:

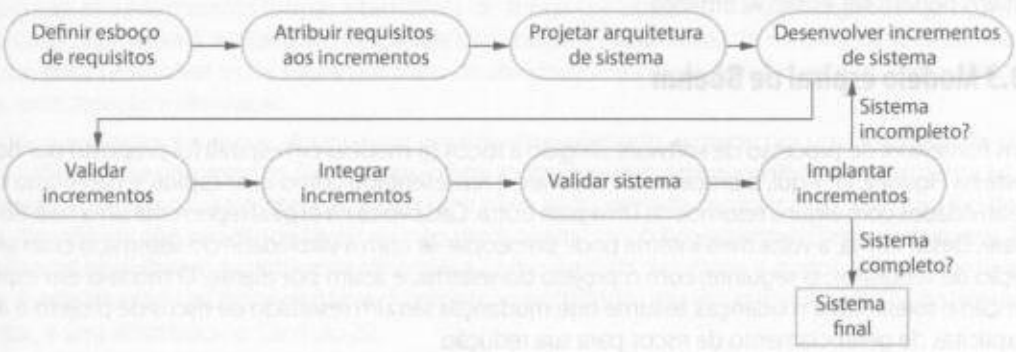
1. Pode ser impossível ajustar o protótipo para atender aos requisitos não funcionais, como requisitos de desempenho, proteção, robustez e confiabilidade, que foram ignorados durante o desenvolvimento do protótipo.
2. Mudanças rápidas durante o desenvolvimento inevitavelmente significam que o protótipo não está documentado. A única especificação de projeto é o código do protótipo. Para a manutenção a longo prazo, isso não é bom o suficiente.
3. As mudanças durante o desenvolvimento do protótipo provavelmente terão degradado a estrutura do sistema. O sistema será difícil e custoso de ser mantido.
4. Padrões de qualidade organizacional geralmente são relaxados para o desenvolvimento do protótipo.

Protótipos não precisam ser executáveis para serem úteis. Maquetes em papel da interface de usuário do sistema (RETTIG, 1994) podem ser eficazes em ajudar os usuários a refinar o projeto de interface e trabalhar por meio de cenários de uso. Estes são muito baratos de se desenvolver e podem ser construídos em poucos dias. Uma extensão dessa técnica é o protótipo Mágico de Oz, no qual apenas a interface de usuário é desenvolvida. Os usuários interagem com essa interface, mas suas solicitações são passadas para uma pessoa que os interpreta e produz a resposta adequada.

2.3.2 Entrega incremental

Entrega incremental (Figura 2.10) é uma abordagem para desenvolvimento de software na qual alguns dos incrementos desenvolvidos são entregues ao cliente e implantados para uso em um ambiente operacional. Em um processo de entrega incremental os clientes identificam, em linhas gerais, os serviços a serem fornecidos pelo sistema. Eles identificam quais dos serviços são mais e menos importantes para eles. Uma série de incrementos de entrega são, então, definidos, com cada incremento proporcionando um subconjunto da funcionalidade do sistema. A atribuição de serviços aos incrementos depende da ordem de prioridade dos serviços — os serviços de mais alta prioridade são implementados e entregues em primeiro lugar.

Figura 2.10 Entrega incremental



Uma vez que os incrementos do sistema tenham sido identificados, os requisitos dos serviços a serem entregues no primeiro incremento são definidos em detalhes, e esse incremento é desenvolvido. Durante o desenvolvimento, podem ocorrer mais análises de requisitos para incrementos posteriores, mas mudanças nos requisitos do incremento atual não são aceitas.

Quando um incremento é concluído e entregue, os clientes podem colocá-lo em operação. Isso significa aceitar a entrega antecipada de uma parte da funcionalidade do sistema. Os clientes podem experimentar o sistema, e isso os ajuda a compreender suas necessidades para incrementos posteriores. Assim que novos incrementos são concluídos, eles são integrados aos incrementos existentes para que a funcionalidade do sistema melhore com cada incremento entregue.

A entrega incremental tem uma série de vantagens:

1. Os clientes podem usar os incrementos iniciais como protótipos e ganhar experiência, a qual informa seus requisitos para incrementos posteriores do sistema. Ao contrário de protótipos, trata-se, aqui, de partes do sistema real, ou seja, não existe a necessidade de reaprendizagem quando o sistema completo está disponível.
2. Os clientes não necessitam esperar até que todo o sistema seja entregue para obter ganhos a partir dele. O primeiro incremento satisfaz os requisitos mais críticos de maneira que eles possam usar o software imediatamente.
3. O processo mantém os benefícios do desenvolvimento incremental, o que deve facilitar a incorporação das mudanças no sistema.
4. Quanto maior a prioridade dos serviços entregues e, em seguida, incrementos integrados, os serviços mais importantes recebem a maioria dos testes. Isso significa que a probabilidade de os clientes encontrarem falhas de software nas partes mais importantes do sistema é menor.

No entanto, existem problemas com a entrega incremental:

1. A maioria dos sistemas exige um conjunto de recursos básicos, usados por diferentes partes do sistema. Como os requisitos não são definidos em detalhes até que um incremento possa ser implementado, pode ser difícil identificar recursos comuns, necessários a todos os incrementos.
2. O desenvolvimento iterativo também pode ser difícil quando um sistema substituto está sendo desenvolvido. Usuários querem toda a funcionalidade do sistema antigo e, muitas vezes, ficam relutantes em experimentar um novo sistema incompleto. Portanto, é difícil obter *feedbacks* úteis dos clientes.
3. A essência do processo iterativo é a especificação ser desenvolvida em conjunto com o software. Isso, contudo, causa conflitos com o modelo de compras de muitas organizações, em que a especificação completa do sistema é parte do contrato de desenvolvimento do sistema. Na abordagem incremental, não há especificação completa do sistema até que o último incremento seja especificado, o que requer uma nova forma de contrato, à qual os grandes clientes, como agências governamentais, podem achar difícil de se adaptar.

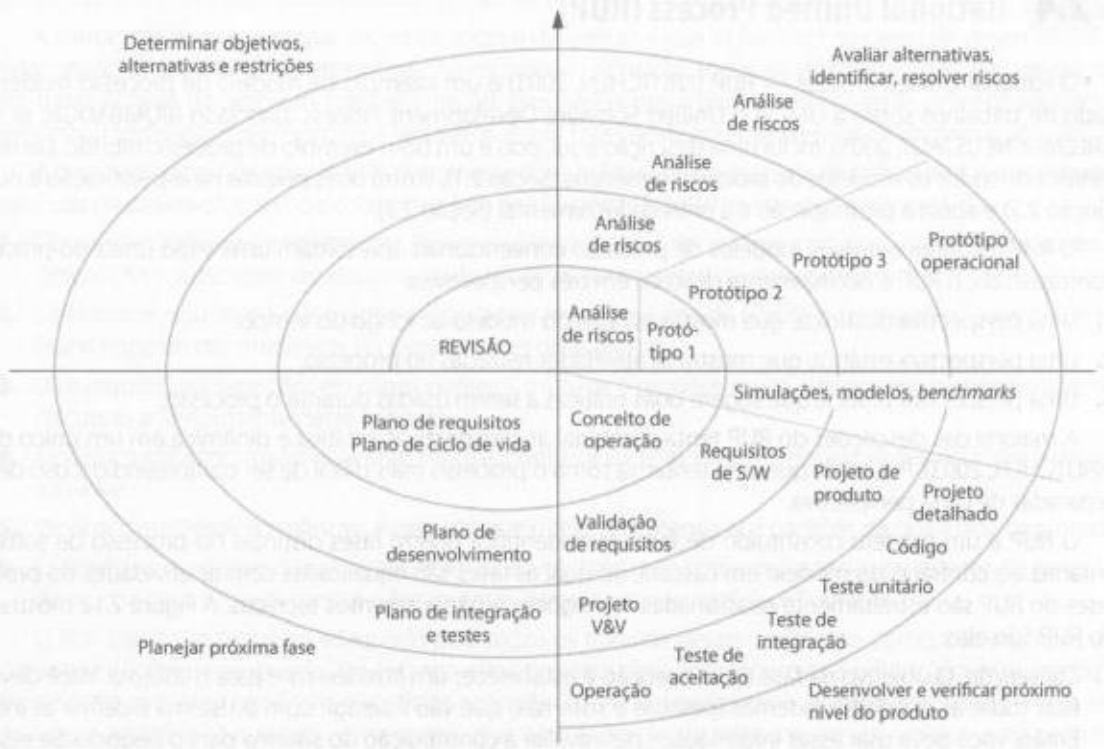
Existem alguns tipos de sistema para os quais o desenvolvimento e a entrega incrementais não são a melhor abordagem. Esses sistemas são muito grandes, de modo que o desenvolvimento pode envolver equipes trabalhando em locais diferentes, além de alguns sistemas embutidos, em que o software depende do desenvolvimento de hardware, e de alguns sistemas críticos, em que todos os requisitos devem ser analisados na busca por interações capazes de comprometer a proteção ou a segurança do sistema.

Tais sistemas, naturalmente, sofrem com os mesmos problemas de requisitos incertos e mutáveis. Portanto, para resolver esses problemas e obter alguns dos benefícios do desenvolvimento incremental, pode ser usado um processo no qual um protótipo de sistema é desenvolvido de forma iterativa e usado como uma plataforma para experimentos com os requisitos e projeto do sistema. Com a experiência adquirida a partir do protótipo, requisitos definitivos podem ser, então, acordados.

2.3.3 Modelo espiral de Boehm

Um *framework* de processo de software dirigido a riscos (o modelo em espiral) foi proposto por Boehm (1988). Isso está na Figura 2.11. Aqui, o processo de software é representado como uma espiral, e não como uma sequência de atividades com alguns retornos de uma para outra. Cada volta na espiral representa uma fase do processo de software. Dessa forma, a volta mais interna pode preocupar-se com a viabilidade do sistema; o ciclo seguinte, com definição de requisitos; o seguinte, com o projeto do sistema, e assim por diante. O modelo em espiral combina prevenção e tolerância a mudanças, assume que mudanças são um resultado de riscos de projeto e inclui atividades explícitas de gerenciamento de riscos para sua redução.

Figura 2.11 Modelo em espiral de processo de software de Boehm (©IEEE 1988)



Cada volta da espiral é dividida em quatro setores:

1. *Definição de objetivos*. Objetivos específicos para essa fase do projeto são definidos; restrições ao processo e ao produto são identificadas, e um plano de gerenciamento detalhado é elaborado; os riscos do projeto são identificados. Podem ser planejadas estratégias alternativas em função desses riscos.
2. *Avaliação e redução de riscos*. Para cada um dos riscos identificados do projeto, é feita uma análise detalhada. Medidas para redução do risco são tomadas. Por exemplo, se houver risco de os requisitos serem inadequados, um protótipo de sistema pode ser desenvolvido.
3. *Desenvolvimento e validação*. Após a avaliação dos riscos, é selecionado um modelo de desenvolvimento para o sistema. Por exemplo, a prototipação descartável pode ser a melhor abordagem de desenvolvimento de interface de usuário se os riscos forem dominantes. Se os riscos de segurança forem a principal consideração, o desenvolvimento baseado em transformações formais pode ser o processo mais adequado, e assim por diante. Se o principal risco identificado for a integração de subsistemas, o modelo em cascata pode ser a melhor opção.
4. *Planejamento*. O projeto é revisado, e uma decisão é tomada a respeito da continuidade do modelo com mais uma volta da espiral. Caso se decida pela continuidade, planos são elaborados para a próxima fase do projeto.

A principal diferença entre o modelo espiral e outros modelos de processo de software é seu reconhecimento explícito do risco. Um ciclo da espiral começa com a definição de objetivos, como desempenho e funcionalidade. Em seguida, são enumeradas formas alternativas de atingir tais objetivos e de lidar com as restrições de cada um deles. Cada alternativa é avaliada em função de cada objetivo, e as fontes de risco do projeto são identificadas. O próximo passo é resolver esses riscos por meio de atividades de coleta de informações, como análise mais detalhada, prototipação e simulação.

Após a avaliação dos riscos, algum desenvolvimento é efetivado, seguido por uma atividade de planejamento para a próxima fase do processo. De maneira informal dizemos que o risco significa, simplesmente, algo que pode dar errado. Por exemplo, se a intenção é usar uma nova linguagem de programação, um risco é o de os compiladores disponíveis não serem confiáveis ou não produzirem um código-objeto eficiente o bastante. Riscos levam a mudanças no software e problemas de projeto, como estouro de prazos e custos. Assim, o gerenciamento de riscos é uma atividade muito importante do projeto, constituindo uma das partes essenciais do gerenciamento de projetos, e será abordado no Capítulo 22.

2.4 Rational Unified Process (RUP)

O Rational Unified Process — RUP (KRUTCHEN, 2003) é um exemplo de modelo de processo moderno, derivado de trabalhos sobre a UML e o Unified Software Development Process associado (RUMBAUGH, et al., 1999; ARLOW e NEUSTADT, 2005). Incluí uma descrição aqui, pois é um bom exemplo de processo híbrido. Ele reúne elementos de todos os modelos de processo genéricos (Seção 2.1), ilustra boas práticas na especificação e no projeto (Seção 2.2) e apoia a prototipação e a entrega incremental (Seção 2.3).

O RUP reconhece que os modelos de processo convencionais apresentam uma visão única do processo. Em contrapartida, o RUP é normalmente descrito em três perspectivas:

1. Uma perspectiva dinâmica, que mostra as fases do modelo ao longo do tempo.
2. Uma perspectiva estática, que mostra as atividades relizadas no processo.
3. Uma perspectiva prática, que sugere boas práticas a serem usadas durante o processo.

A maioria das descrições do RUP tenta combinar as perspectivas estática e dinâmica em um único diagrama (KRUTCHEN, 2003). Por achar que essa tentativa torna o processo mais difícil de ser compreendido, uso descrições separadas de cada perspectiva.

O RUP é um modelo constituído de fases que identifica quatro fases distintas no processo de software. No entanto, ao contrário do modelo em cascata, no qual as fases são equalizadas com as atividades do processo, as fases do RUP são estreitamente relacionadas ao negócio, e não a assuntos técnicos. A Figura 2.12 mostra as fases do RUP. São elas:

1. *Concepção*. O objetivo da fase de concepção é estabelecer um *business case* para o sistema. Você deve identificar todas as entidades externas (pessoas e sistemas) que vão interagir com o sistema e definir as interações. Então, você deve usar essas informações para avaliar a contribuição do sistema para o negócio. Se essa contribuição for pequena, então o projeto poderá ser cancelado depois dessa fase.
2. *Elaboração*. As metas da fase de elaboração são desenvolver uma compreensão do problema dominante, estabelecer um *framework* da arquitetura para o sistema, desenvolver o plano do projeto e identificar os maiores riscos do projeto. No fim dessa fase, você deve ter um modelo de requisitos para o sistema, que pode ser um conjunto de casos de uso da UML, uma descrição da arquitetura ou um plano de desenvolvimento do software.
3. *Construção*. A fase de construção envolve projeto, programação e testes do sistema. Durante essa fase, as partes do sistema são desenvolvidas em paralelo e integradas. Na conclusão dessa fase, você deve ter um sistema de software já funcionando, bem como a documentação associada pronta para ser entregue aos usuários.
4. *Transição*. A fase final do RUP implica transferência do sistema da comunidade de desenvolvimento para a comunidade de usuários e em seu funcionamento em um ambiente real. Isso é ignorado na maioria dos modelos de processo de software, mas é, de fato, uma atividade cara e, às vezes, problemática. Na conclusão dessa fase, você deve ter um sistema de software documentado e funcionando corretamente em seu ambiente operacional.

No RUP, a iteração é apoiada de duas maneiras. Cada fase pode ser executada de forma iterativa com os resultados desenvolvidos de forma incremental. Além disso, todo o conjunto de fases também pode ser executado de forma incremental, como indicado pela seta curva de 'transição' para concepção, na Figura 2.12.

A visão estática do RUP prioriza as atividades que ocorrem durante o processo de desenvolvimento. Na descrição do RUP, essas são chamadas *workflows*. Existem seis *workflows* centrais, identificadas no processo, e três *workflows* de apoio. O RUP foi projetado em conjunto com a UML, assim, a descrição do *workflow* é orientada em

Figura 2.12 Fases no Rational Unified Process

torno de modelos associados à UML, como modelos de sequência, modelos de objetos etc. Os *workflows* centrais de engenharia e de apoio estão descritos na Tabela 2.1.

A vantagem de proporcionar visões estáticas e dinâmicas é que as fases do processo de desenvolvimento não estão associadas a *workflows* específicos. Ao menos em princípio, todos os *workflows* do RUP podem estar ativos em todas as fases do processo. Nas fases iniciais, provavelmente, maiores esforços serão empenhados em *workflows*, como modelagem de negócios e requisitos, e, nas fases posteriores, no teste e na implantação.

A perspectiva prática sobre o RUP descreve as boas práticas da engenharia de software que são recomendadas para uso no desenvolvimento de sistemas. Seis boas práticas fundamentais são recomendadas:

1. *Desenvolver software iterativamente.* Planejar os incrementos do sistema com base nas prioridades do cliente e desenvolver os recursos de alta prioridade no início do processo de desenvolvimento.
2. *Gerenciar os requisitos.* Documentar explicitamente os requisitos do cliente e acompanhar suas mudanças. Analisar o impacto das mudanças no sistema antes de aceitá-las.
3. *Usar arquiteturas baseadas em componentes.* Estruturar a arquitetura do sistema em componentes, conforme discutido anteriormente neste capítulo.
4. *Modelar o software visualmente.* Usar modelos gráficos da UML para apresentar visões estáticas e dinâmicas do software.
5. *Verificar a qualidade do software.* Assegurar que o software atenda aos padrões de qualidade organizacional.
6. *Controlar as mudanças do software.* Gerenciar as mudanças do software, usando um sistema de gerenciamento de mudanças e procedimentos e ferramentas de gerenciamento de configuração.

O RUP não é um processo adequado para todos os tipos de desenvolvimento, como, por exemplo, desenvolvimento de software embutido. No entanto, ele representa uma abordagem que potencialmente combina os três modelos de processo genéricos discutidos na Seção 2.1. As inovações mais importantes do RUP são a separação de fases e *workflows* e o reconhecimento de que a implantação de software em um ambiente do usuário é parte do processo. As fases são dinâmicas e têm metas. Os *workflows* são estáticos e são atividades técnicas que não são associadas a uma única fase, mas podem ser utilizadas durante todo o desenvolvimento para alcançar as metas específicas.

Tabela 2.1 *Workflows* estáticos no Rational Unified Process

WORKFLOW	DESCRIÇÃO
Modelagem de negócios	Os processos de negócio são modelados por meio de casos de uso de negócios.
Requisitos	Atores que interagem com o sistema são identificados e casos de uso são desenvolvidos para modelar os requisitos do sistema.
Análise e projeto	Um modelo de projeto é criado e documentado com modelos de arquitetura, modelos de componentes, modelos de objetos e modelos de sequência.
Implementação	Os componentes do sistema são implementados e estruturados em subsistemas de implementação. A geração automática de código a partir de modelos de projeto ajuda a acelerar esse processo.
Teste	O teste é um processo iterativo que é feito em conjunto com a implementação. O teste do sistema segue a conclusão da implementação.
Implantação	Um *release* do produto é criado, distribuído aos usuários e instalado em seu local de trabalho.
Gerenciamento de configuração e mudanças	Esse *workflow* de apoio gerencia as mudanças do sistema (veja o Capítulo 25).
Gerenciamento de projeto	Esse *workflow* de apoio gerencia o desenvolvimento do sistema (veja os capítulos 22 e 23).
Meio ambiente	Esse *workflow* está relacionado com a disponibilização de ferramentas apropriadas para a equipe de desenvolvimento de software.

PONTOS IMPORTANTES

- Os processos de software são as atividades envolvidas na produção de um sistema de software. Modelos de processos de software são representações abstratas desses processos.
- Modelos gerais de processo descrevem a organização dos processos de software. Exemplos desses modelos gerais incluem o modelo em cascata, o desenvolvimento incremental e o desenvolvimento orientado a reúso.
- Engenharia de requisitos é o processo de desenvolvimento de uma especificação de software. As especificações destinam-se a comunicar as necessidades de sistema dos clientes para os desenvolvedores do sistema.
- Processos de projeto e implementação estão relacionados com a transformação das especificações dos requisitos em um sistema de software executável. Métodos sistemáticos de projeto podem ser usados como parte dessa transformação.
- Validação de software é o processo de verificação de que o sistema está de acordo com sua especificação e satisfaz às necessidades reais dos usuários do sistema.
- Evolução de software ocorre quando se alteram os atuais sistemas de software para atender aos novos requisitos. As mudanças são contínuas, e o software deve evoluir para continuar útil.
- Processos devem incluir atividades para lidar com as mudanças. Podem envolver uma fase de prototipação, que ajuda a evitar más decisões sobre os requisitos e projeto. Processos podem ser estruturados para o desenvolvimento e a entrega iterativos, de forma que mudanças possam ser feitas sem afetar o sistema como um todo.
- O Rational Unified Process (RUP) é um moderno modelo genérico de processo, organizado em fases (concepção, elaboração, construção e transição), mas que separa as atividades (requisitos, análises, projeto etc.) dessas fases.

LEITURA COMPLEMENTAR

Managing Software Quality and Business Risk. Esse é essencialmente um livro sobre gerenciamento de software, mas que inclui um excelente capítulo (Capítulo 4) sobre os modelos de processo. (OULD, M. *Managing Software Quality and Business Risk*. John Wiley and Sons Ltd., 1999.)

"Process Models in Software Engineering". Essa é uma excelente visão geral de uma vasta gama de modelos de processo de engenharia de software que têm sido propostos. (SCACCHI, W. "Process Models in Software Engineering". In: MARCINIAK, J. J. (Orgs.). *Encyclopaedia of Software Engineering*. John Wiley and Sons, 2001.) Disponível em: <http://www.ics.uci.edu/~wscacchi/Papers/SE-Encyc/Process-Models-SE- Encyc.pdf>.

The Rational Unified Process – An Introduction (3rd Edition). Esse é o livro sobre RUP mais lido no momento. Krutchen descreve bem o processo, mas eu gostaria de ter lido mais sobre as dificuldades práticas do uso do processo. (KRUTCHEN, P. *The Rational Unified Process – An Introduction*. 3.ed. Addison-Wesley, 2003.)

EXERCÍCIOS

2.1 Justificando sua resposta com base no tipo de sistema a ser desenvolvido, sugira o modelo genérico de processo de software mais adequado para ser usado como base para a gerência do desenvolvimento dos sistemas a seguir:

Um sistema para controlar o antibloqueio de frenagem de um carro.

Um sistema de realidade virtual para dar apoio à manutenção de software.

Um sistema de contabilidade para uma universidade, que substitua um sistema já existente.

Um sistema interativo de planejamento de viagens que ajude os usuários a planejar viagens com menor impacto ambiental.

2.2 Explique por que o desenvolvimento incremental é o método mais eficaz para o desenvolvimento de sistemas de software de negócios. Por que esse modelo é menos adequado para a engenharia de sistemas de tempo real?

2.3 Considere o modelo de processo baseado em reúso da Figura 2.3. Explique por que, nesse processo, é essencial ter duas atividades distintas de engenharia de requisitos.

2.4 Sugira por que é importante, no processo de engenharia de requisitos, fazer uma distinção entre desenvolvimento dos requisitos do usuário e desenvolvimento de requisitos de sistema.

2.5 Descreva as principais atividades do processo de projeto de software e as saídas dessas atividades. Usando um diagrama, mostre as possíveis relações entre as saídas dessas atividades.

2.6 Explique por que, em sistemas complexos, as mudanças são inevitáveis. Exemplifique as atividades de processo de software que ajudam a prever as mudanças e fazer com que o software seja desenvolvido mais tolerante a mudanças (desconsidere prototipação e entrega incremental).

2.7 Explique por que os sistemas desenvolvidos como protótipos normalmente não devem ser usados como sistemas de produção.

2.8 Explique por que o modelo em espiral de Boehm é um modelo adaptável, que apoia tanto as atividades de prevenção de mudanças quanto as de tolerância a mudanças. Na prática, esse modelo não tem sido amplamente usado. Sugira as possíveis razões para isso.

2.9 Quais são as vantagens de proporcionar visões estáticas e dinâmicas do processo de software, assim como no Rational Unified Process?

2.10 Historicamente, a introdução de tecnologia provocou mudanças profundas no mercado de trabalho e, pelo menos temporariamente, deixou muitas pessoas desempregadas. Discuta se a introdução da automação extensiva em processos pode vir a ter as mesmas consequências para os engenheiros de software. Se sua resposta for não, justifique. Se você acha que sim, que vai reduzir as oportunidades de emprego, é ética a resistência passiva ou ativa, pelos engenheiros afetados, à introdução dessa tecnologia?

REFERÊNCIAS

ARLOW, J.; NEUSTADT, I. *UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design*. 2.ed. Boston: Addison-Wesley, 2005.

BOEHM, B.; TURNER, R. *Balancing Agility and Discipline: A Guide for the Perplexed*. Boston: Addison-Wesley, 2003.

BOEHM, B. W. "A Spiral Model of Software Development and Enhancement". *IEEE Computer*, v. 21, n. 5, 1988, p. 61-72.

BUDGEN, D. *Software Design*. 2.ed. Harlow, Reino Unido: Addison-Wesley, 2003.

KRUTCHEN, P. *The Rational Unified Process — An Introduction*. Reading, MA: Addison-Wesley, 2003.

MASSOL, V.; HUSTED, T. *JUnit in Action*. Greenwich, Conn.: Manning Publications Co., 2003.

RETTIG, M. "Practical Programmer: Prototyping for Tiny Fingers". *Comm. ACM*, v. 37, n. 4, 1994, p. 21-7.

ROYCE, W. W. "Managing the Development of Large Software Systems: Concepts and Techniques". *IEEE WESTCON*. Los Angeles, CA, 1970, p. 1-9.

RUMBAUGH, J.; JACOBSON, I.; BOOCH, G. *The Unified Software Development Process*. Reading, Mass: Addison-Wesley, 1999.

SCHMIDT, D. C. "Model-Driven Engineering". *IEEE Computer*, v. 39, n. 2, 2006, p. 25-31.

SCHNEIDER, S. *The B Method*. Houndmills, Reino Unido: Palgrave Macmillan, 2001.

WORDSWORTH, J. *Software Engineering with B*. Wokingham: Addison-Wesley, 1996.