Capítulo 1

Conceitos básicos

Este capítulo tem por objetivo definir engenharia de software e explicar a sua importância para o desenvolvimento de sistemas de software. Para que esse objetivo seja atingido, são definidos outros conceitos importantes, tais como sistemas, software, produtos e processos de software.

1.1 Sistemas de software

A palavra *sistema* é definida no dicionário da língua portuguesa [FER 86] como "um conjunto de elementos concretos ou abstratos entre os quais se pode encontrar alguma relação". Bruno Maffeo [MAF 92] define um sistema em termos gerais como:

> Um conjunto, identificável e coerente, de elementos que interagem coesivamente, onde cada elemento pode ser um sistema.

Nessas duas definições, fica evidente a noção de relação estrutural que deve existir entre as partes componentes do sistema. Pode-se traçar uma fronteira conceitual separando o sistema do resto do universo e tratar o que está em seu interior como uma entidade relativamente autônoma e identificável. Os elementos que constituem o sistema, isto é, aqueles que estão dentro de seu contorno conceitual, têm entre eles interações fortes, quando comparadas às interações entre estes e os elementos do resto do universo. As interações entre os elementos constituintes do sistema e os demais elementos do universo devem ser suficientemente fracas para que possam ser desprezadas, quando se deseja considerar o sistema isolado. Por exemplo, em um estudo sobre ecologia, pode-se definir o ecossistema de uma certa espécie de inseto que passa toda a sua vida em uma única árvore de uma determinada espécie. A fronteira conceitual, nesse caso, envolveria o inseto (fisiologia, anatomia, hábitos reprodutivos, ciclo de vida etc.), a árvore, a fauna que habita a árvore e a caracterização do hábitat da árvore (que inclui clima, solo, vegetação na vizinhança etc.), como mostra a Figura 1.1.

Figura 1.1: Ecossistema de um inseto.

Se, por outro lado, o ecossistema fosse a floresta Amazônica, então a fronteira conceitual deveria envolver a fauna, a flora, a ocupação humana na floresta e em volta dela e a utilização que os seres humanos fazem da floresta, podendo inclusive chegar a aspectos atmosféricos.

Por analogia, quando se trata de *sistemas de software*, busca-se identificar componentes do sistema que interagem entre si para atender necessidades específicas do ambiente no qual estão inseridos. Existe também a possibilidade de haver um componente (sistêmico) humano. A escolha de uma fronteira conceitual adequada é um passo decisivo para o êxito do processo de concepção do sistema, pois a determinação dessa fronteira permitirá a separação entre os componentes pertencentes ao sistema, cujas informações devem ser detalhadamente estudadas, e aqueles pertencentes ao ambiente externo, que só têm interesse quanto a sua interação com o sistema.

Para exemplificar o conceito de fronteira conceitual de um sistema de software, pode-se considerar um *sistema de reservas de passagem aérea* de uma determinada companhia.

Para esse sistema, a fronteira conceitual só englobaria a própria companhia aérea e os dados sobre vôos (disponibilidade, reserva, cancelamento etc.). Se, por outro lado, o sistema de software incluísse reservas de hotel, locação de carros, ofertas de pacotes turísticos e assim por diante, a fronteira conceitual deveria ser muito mais abrangente, envolvendo informações sobre hotéis, locadoras de carros e agências de turismo. Sistemas de software entregues ao usuário com a documentação que descreve como instalar e usar o sistema são chamados *produtos de software* [SOM 96].

Existem duas categorias de produtos de software: (1) sistemas genéricos, produzidos e vendidos no mercado a qualquer pessoa que possa comprá-los; e (2) sistemas específicos, encomendados especialmente por um determinado cliente. O produto de software consiste em:

1) *instruções* (programas de computador) que, quando executadas, realizam as funções e têm o desempenho desejados;

2) *estruturas de dados* que possibilitam às instruções manipular as informações de forma adequada;

3) *documentos* que descrevem as operações e uso do produto.

Sistemas de software são produtos lógicos, não suscetíveis aos problemas do meio ambiente. No começo da vida de um sistema, há um alto índice de erros, mas, à medida que esses erros são corrigidos, o índice se estabiliza [PRE 92]. Com a introdução de mudanças, seja para corrigir erros descobertos após a entrega do produto, seja para adaptar o sistema a novas tecnologias de software e hardware, ou ainda para incluir novos requisitos do usuário, novos erros são também introduzidos, e o software começa a se deteriorar.

A maioria dos produtos de software é construída de acordo com as necessidades do usuário e não montada a partir de componentes já existentes, pois não existem catálogos de componentes de software; é possível comprar produtos de software, mas somente como uma unidade completa, não como componentes separados que podem ser utilizados na confecção de novos sistemas. Todavia, esta situação está mudando rapidamente, com a disseminação do conceito de software reutilizável, que privilegia a reutilização de componentes de produtos de software já existentes [SOM 96]. Uma grande vantagem deste tipo de abordagem é a redução dos custos de desenvolvimento como um todo, pois um menor número de componentes terá de ser desenvolvido e validado. Outras vantagens dessa abordagem são o aumento na confiabilidade do sistema, pois os componentes a ser reutilizados já foram testados em outros sistemas, e a redução dos riscos de desenvolvimento, pois existe menos incerteza sobre os custos de reutilização de software se comparados aos custos de desenvolvimento.

O *processo de desenvolvimento de software* envolve o conjunto de atividades e resultados associados a essas atividades, com o objetivo de construir o produto de software. Existem três atividades fundamentais, comuns a todos os processos de construção de software, apresentadas na Figura 1.2. São elas:

1) *desenvolvimento*: as funcionalidades e as restrições relativas à operacionalidade do produto são especificadas, e o software é produzido de acordo com essas especificações;

2) *validação*: o produto de software é validado para garantir que ele faça exatamente o que o usuário deseja;

3) *manutenção*: o software sofre correções, adaptações e ampliações para corrigir erros encontrados após a entrega do produto, atender os novos requisitos do usuário e incorporar mudanças na tecnologia.

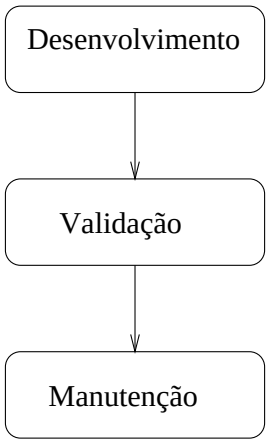

Figura 1.2: O processo de desenvolvimento de software.

O processo de desenvolvimento de software é complexo e envolve inúmeras atividades, e cada uma delas pode ser detalhada em vários passos a ser seguidos durante o desenvolvimento. *Modelos de processos* (também chamados *paradigmas de desenvolvimento*) especificam as atividades que, de acordo com o modelo, devem ser executadas, assim como a ordem em que devem ser executadas. Produtos de software podem ser construídos utilizando-se diferentes modelos de processos. No entanto, alguns modelos são mais adequados que outros para determinados tipos de aplicação, e a opção por um determinado modelo deve ser feita levando-se em consideração o produto a ser desenvolvido.

1.2 Engenharia de software

Durante as décadas de 60 e 70, o desafio primordial era desenvolver hardware que reduzisse o custo de processamento e de armazenamento de dados [PRE 92]. Durante a década de 80, avanços na microeletrônica resultaram em um aumento do poder computacional a um custo cada vez menor. Entretanto, tanto o processo de desenvolvimento como o software produzido ainda deixavam muito a desejar: cronogramas não eram cumpridos, custos excediam os previstos, o software não cumpria os requisitos estipulados e assim por diante. Portanto, o desafio primordial nas últimas duas décadas foi e continua sendo melhorar a qualidade e reduzir o custo do software produzido, através da introdução de disciplina no desenvolvimento, o que é conhecido como engenharia de software. Dessa forma, pode-se dizer que a engenharia de software é:

Uma disciplina que reúne metodologias, métodos e ferramentas a ser utilizados, desde a percepção do problema até o momento em que o sistema desenvolvido deixa de ser operacional, visando resolver problemas inerentes ao processo de desenvolvimento e ao produto de software.

O objetivo da engenharia de software é auxiliar no processo de produção de software, de forma que o processo dê origem a produtos de alta qualidade, produzidos mais rapidamente e a um custo cada vez menor. São muitos os problemas a ser tratados pela engenharia de software, pois tanto o processo quanto o produto de software possuem vários atributos que devem ser considerados para que se tenha sucesso, por exemplo, a complexidade, a visibilidade, a aceitabilidade, a confiabilidade, a manutenibilidade, a segurança etc. Por exemplo, para a especificação de sistemas de controle de tráfego aéreo e ferroviário, a confiabilidade é um atributo fundamental [FFO 00]. Já para sistemas mais simples, tais como os controladores embutidos em aparelhos eletrodomésticos, como lavadoras de roupa e videocassetes, o desempenho é o atributo mais importante a ser considerado.

A engenharia de software herda da engenharia o conceito de disciplina na produção de software, através de *metodologias*, que por sua vez seguem *métodos* que se utilizam de *ferramentas* automatizadas para englobar as principais atividades do processo de produção de software. Alguns métodos focalizam as funções do sistema; outros se concentram nos objetos que o povoam; outros, ainda, baseiam-se nos eventos que ocorrem durante o funcionamento do sistema.

Existem alguns princípios da engenharia de software que descrevem de maneira geral e abstrata as propriedades desejáveis para os processos e produtos de software. O desenvolvimento de software deve ser norteado por esses princípios, de forma que seus objetivos sejam alcançados.

1.3 Os princípios da engenharia de software

Existem vários princípios importantes e gerais que podem ser aplicados durante toda a fase de desenvolvimento do software [GUE 91]. Esses princípios se referem tanto ao processo como ao produto final e descrevem algumas propriedades gerais dos processos e produtos. O processo correto ajudará a produzir o produto correto, e o produto almejado também afetará a escolha do processo a ser utilizado. Entretanto, esses princípios por si sós não são suficientes para dirigir o desenvolvimento de software. Para aplicar esses princípios na construção de sistemas de software, o desenvolvedor deve estar equipado com as *metodologias* apropriadas e com os *métodos* e as *ferramentas* específicos que o ajudarão a incorporar as propriedades desejadas aos processos e produtos. Além disso, os princípios devem guiar a escolha das metodologias, métodos e ferramentas apropriados para o desenvolvimento de software. Alguns dos princípios a ser observados durante o desenvolvimento são descritos a seguir.

1.3.1 Formalidade

O desenvolvimento de software é uma atividade criativa e, como tal, tende a ser imprecisa e a seguir a "inspiração do momento" de uma maneira não estruturada. Através de um enfoque formal, pode-se produzir produtos mais confiáveis, controlar seu custo e ter mais confiança no seu desempenho. A formalidade não deve restringir a criatividade, mas melhorá-la através do aumento da confiança do desenvolvedor em resultados criativos, uma vez que eles são criticamente analisados à luz de uma avaliação formal. Em todo o campo da engenharia, o projeto acontece como uma seqüência de passos definidos com precisão. Em cada passo o desenvolvedor utiliza alguma metodologia que segue algum método, que pode ser formal, informal ou semiformal. Não há necessidade de ser sempre formal, mas o desenvolvedor tem de saber quando e como sê-lo. Por exemplo, se a tarefa atribuída ao engenheiro fosse projetar uma embarcação para atravessar de uma margem para a outra de um riacho calmo, talvez uma jangada fosse suficiente. Se, por outro lado, a tarefa fosse projetar uma embarcação para navegar através de um rio de águas turbulentas, na certa o engenheiro teria de fazer um projeto com especificações mais precisas para uma embarcação muito mais sofisticada. Finalmente, projetar uma embarcação para fazer uma viagem transatlântica demandaria maior formalidade na especificação do produto.

O mesmo acontece na engenharia de software. O desenvolvedor de software deve ser capaz de entender o nível de formalidade que deve ser atingido, dependendo da dificuldade conceitual da tarefa a ser executada. As partes consideradas críticas podem necessitar de uma descrição formal de suas funções, enquanto as partes mais bem entendidas e padronizadas podem necessitar de métodos mais simples. Tradicionalmente, o formalismo é usado somente na fase de programação, pois programas são componentes formais do sistema, com sintaxe e semântica totalmente definidas e que podem ser automaticamente manipulados pelos compiladores. Entretanto, a formalidade pode ser aplicada durante toda a fase de desenvolvimento do software, e seus efeitos benéficos podem ser sentidos na manutenção, reutilização, portabilidade e entendimento do software.

1.3.2 Abstração

Abstração é o processo de identificação dos aspectos importantes de um determinado fenômeno, ignorando-se os detalhes. Os detalhes a ser ignorados vão depender do objetivo da abstração. Por exemplo, para um telefone sem fio, uma abstração útil para o usuário seria um manual contendo a descrição dos efeitos de apertar os vários botões, o que permite que o telefone entre nos vários módulos de funcionalidade e reaja diferentemente às seqüências de comandos. Uma abstração útil para a pessoa ocupada em manter o telefone funcionando seria um manual contendo as informações sobre a caixa que deve ser aberta para substituir a pilha. Outras abstrações podem ser feitas para entender o funcionamento do telefone e as atividades necessárias para consertá-lo. Portanto, podem existir diferentes abstrações da mesma realidade, cada uma fornecendo uma *visão* diferente da realidade e servindo para diferentes objetivos. Quando requisitos de uma aplicação são analisados e especificados, desenvolvedores de software constroem modelos da aplicação proposta. Esses

modelos podem ser expressos de várias formas, dependendo do grau de formalidade desejado. As linguagens de programação, por exemplo, fornecem condições para que programas sejam escritos ignorando-se os detalhes, tais como o número de bits usado para representar números e caracteres ou o mecanismo de endereçamento utilizado. Isso permite que o programador se concentre no problema a ser resolvido e não na máquina. Os programas, por si sós, são abstrações das funcionalidades do sistema.

1.3.3 Decomposição

Uma das maneiras de lidar com a complexidade é subdividir o processo em atividades específicas, provavelmente atribuídas a especialistas de diferentes áreas. Podem-se separar essas atividades de várias formas, uma delas por tempo. Por exemplo, considere o caso de um médico-cirurgião que decide concentrar suas atividades cirúrgicas no período da manhã e seu atendimento aos pacientes no período da tarde, reservando as sextas-feiras para estudo e atualização. Essa separação permite o planejamento das atividades e diminui o tempo extra que seria gasto mudando de uma atividade para outra. No caso do software, pode-se separar, por exemplo, atividades relativas ao controle de qualidade do processo e do produto das atividades de desenvolvimento, como, por exemplo, especificação do projeto, implementação e manutenção, que são atividades bastante complexas. Além disso, cada uma dessas atividades pode ser decomposta, levando naturalmente à divisão das tarefas, possivelmente atribuídas a pessoas diferentes, com diferentes especialidades. A decomposição das atividades leva também à separação das preocupações. Por exemplo, pode-se lidar com a eficiência e correção de um dado produto de software separadamente, primeiro projetando-o de maneira cuidadosa e estruturada, de forma que garanta seu corretismo, e só então passando a reestruturá-lo parcialmente, de forma que melhore sua eficiência.

Além da decomposição do processo, também se pode decompor o produto em sub-produtos, definidos de acordo com o sistema que está sendo desenvolvido. Essa decomposição do produto traz inúmeras vantagens; por exemplo, permite que atividades do processo de desenvolvimento sejam executadas paralelamente. Além disso, dado que os componentes são independentes, eles podem ser reutilizados por outros componentes ou sistemas, e não haverá propagação de erros para outros componentes. A decomposição do produto pode ser feita, por exemplo, em termos dos objetos que povoam o sistema. Nesse caso, o produto será decomposto em um conjunto de objetos que se comunicam. Uma outra maneira de decompor o produto é considerando-se as funções que ele deve desempenhar. Nesse caso, o produto é decomposto em componentes funcionais que aceitam dados de entrada e os transformam em dados de saída. O objetivo maior, nos dois casos, é diminuir a complexidade.

1.3.4 Generalização

O princípio da generalização é importante pois, sendo mais geral, é bem possível que a solução para o problema tenha mais potencial para ser reutilizada. Também pode acontecer que através da generalização o desenvolvedor acabe projetando um componente que seja

utilizado em mais de um ponto do sistema de software desenvolvido, em vez de ter várias soluções especializadas. Por outro lado, uma solução generalizada pode ser mais custosa, em termos de velocidade de execução ou tempo de desenvolvimento. Portanto, é necessário avaliar os problemas de custo e eficiência para poder decidir se vale a pena desenvolver um sistema generalizado em vez de atender a especificação original. Por exemplo, supondo que seja necessário desenvolver um sistema de software para catalogar os livros de uma biblioteca pequena, em que cada livro tem um nome, autor, editor, data de publicação e um código específico. Além de catalogar os livros da biblioteca, deve ser possível fazer buscas baseadas na disponibilidade dos livros, por autor, por título, por palavras-chave etc. Em vez de especificar um conjunto de funcionalidades para o produto de software, envolvendo apenas essas funcionalidades, pode-se considerá-las como um caso especial de um conjunto mais geral de funcionalidades fornecidas por um sistema de biblioteca, como empréstimos, devoluções, aquisições etc. Esse sistema mais geral atenderia as necessidades do proprietário da pequena biblioteca e poderia interessar também a bibliotecas de médio porte, para as quais as outras funcionalidades são relevantes.

1.3.5 Flexibilização

O princípio da flexibilização diz respeito tanto ao processo como ao produto de software. O produto sofre constantes mudanças, pois em muitos casos a aplicação é desenvolvida enquanto seus requisitos ainda não foram totalmente entendidos. Isso ocorre porque o processo de desenvolvimento acontece passo a passo, de maneira incremental. Mesmo depois de entregue ao usuário, o produto pode sofrer alterações, seja para eliminar erros que não tenham sido detectados antes da entrega, seja para evoluir no sentido de atender as novas solicitações do usuário, seja para adaptar o produto de software às novas tecnologias de hardware e software. Os produtos são com freqüência inseridos em uma estrutura organizacional, o ambiente é afetado pela introdução do produto, e isso gera novos requisitos que não estavam presentes inicialmente.

O princípio da flexibilização é necessário no processo de desenvolvimento para permitir que o produto possa ser modificado com facilidade. O processo deve ter flexibilidade suficiente para permitir que partes ou componentes do produto desenvolvido possam ser utilizados em outros sistemas, bem como a sua portabilidade para diferentes sistemas computacionais.

A fim de alcançar esses princípios, a engenharia de software necessita de mecanismos para controlar o processo de desenvolvimento. Na próxima seção serão vistos três dos paradigmas mais utilizados no desenvolvimento de sistemas de software.

1.4 Paradigmas de engenharia de software

Paradigmas são modelos de processo que possibilitam: (a) ao gerente: controlar o processo de desenvolvimento de sistemas de software; e (b) ao desenvolvedor: obter a base para produzir, de maneira eficiente, software que satisfaça os requisitos preestabelecidos. Os paradigmas especificam algumas atividades que devem ser executadas, assim como a ordem em

que devem ser executadas. A função dos paradigmas é diminuir os problemas encontrados no processo de desenvolvimento do software, e, devido à importância desse processo, vários paradigmas já foram propostos. O paradigma é escolhido de acordo com a natureza do projeto e do produto a ser desenvolvido, dos métodos e ferramentas a ser utilizados e dos controles e produtos intermediários desejados. A seguir serão apresentados três dos paradigmas mais utilizados. São eles: *ciclo de vida clássico*, *evolutivo* e *espiral*.

1.4.1 Ciclo de vida clássico

É um paradigma que utiliza um método sistemático e seqüencial, em que o resultado de uma fase se constitui na entrada de outra. Devido à forma com que se dá a passagem de uma fase para outra, em ordem linear, esse paradigma também é conhecido como *cascata*. Cada fase é estruturada como um conjunto de atividades que podem ser executadas por pessoas diferentes, simultaneamente. Compreende as seguintes atividades, apresentadas na Figura 1.3: análise e especificação dos requisitos; projeto; implementação e teste unitário; integração e teste do sistema; e operação e manutenção. Existem inúmeras variações desse paradigma, dependendo da natureza das atividades e do fluxo de controle entre elas. Os estágios principais do paradigma estão relacionados às atividades fundamentais de desenvolvimento.

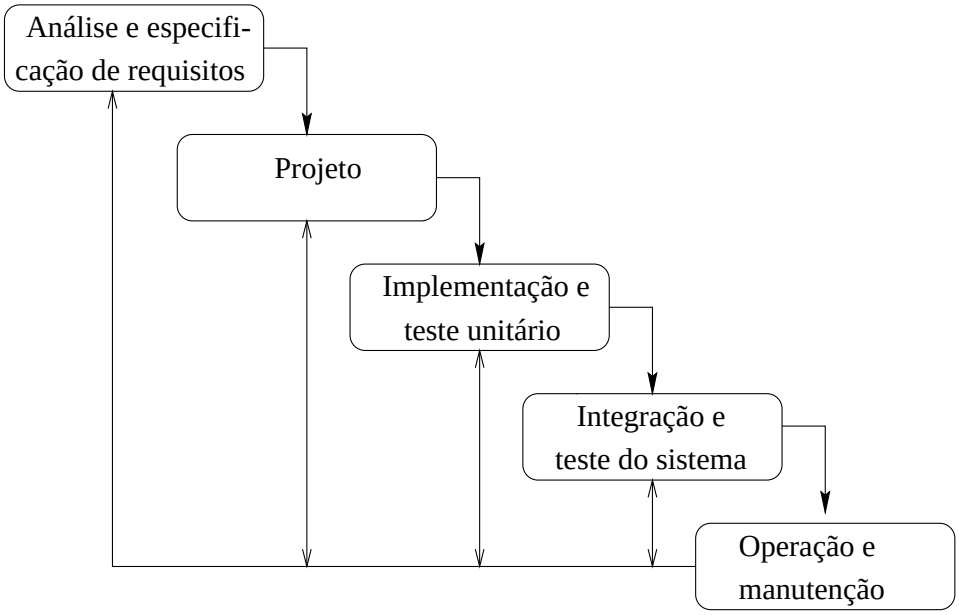

Figura 1.3: Os cinco passos do ciclo de vida clássico.

A opção pelo paradigma clássico vai depender da complexidade da aplicação. Aplicações simples e bem entendidas demandam menos formalidade; já aplicações maiores e mais complexas podem necessitar de uma maior decomposição do processo para garantir um melhor controle e obter os resultados almejados. Por exemplo, o desenvolvimento de uma aplicação

a ser utilizada por usuários não especialistas pode necessitar de uma fase na qual um material especial de treinamento é projetado e desenvolvido para tornar-se parte integrante do software; além disso, a fase em que o sistema entra em operação deve conter uma fase de treinamento. Por outro lado, se os usuários forem especialistas, a fase de treinamento pode não existir, sendo necessário somente o fornecimento de manuais técnicos. Outros detalhes podem ser fornecidos por telefone ou por um serviço de atendimento ao usuário.

Análise e especificação de requisitos

Durante essa fase do desenvolvimento, são identificados, através de consultas aos usuários do sistema, os serviços e as metas a ser atingidas, assim como as restrições a ser respeitadas. É, portanto, identificada a qualidade desejada para o sistema a ser desenvolvido, em termos de funcionalidade, desempenho, facilidade de uso, portabilidade etc. O desenvolvedor deve especificar *quais* os requisitos que o produto de software deverá possuir, sem especificar *como* esses requisitos serão obtidos através do projeto e da implementação. Por exemplo, ele deve definir quais funções o produto de software deverá desempenhar, sem dizer como uma certa estrutura ou algoritmo podem ajudar a realizá-las. Os requisitos especificados não devem restringir o desenvolvedor de software nas atividades de projeto e implementação; ele deve ter liberdade e responsabilidade para escolher a estrutura mais adequada, assim como para fazer outras escolhas relativas à implementação do software. O resultado dessa fase é um documento de especificação de requisitos, com dois objetivos: (1) o documento deve ser analisado e confirmado pelo usuário para verificar se ele satisfaz todas as suas expectativas; e (2) deve ser usado pelos desenvolvedores de software para obter um produto que satisfaça os requisitos.

Esse documento vai servir de instrumento de comunicação entre muitos indivíduos e, portanto, deve ser inteligível, preciso, completo, consistente e sem ambigüidades. Além disso, deve ser facilmente modificável, pois deve evoluir para acomodar a natureza evolutiva dos sistemas de software. As características da especificação podem variar, dependendo do contexto. Por exemplo, *precisão* pode significar formalidade para o desenvolvedor de software, ainda que especificações formais possam ser ininteligíveis para o usuário. Uma maneira de conciliar as necessidades do usuário e as do desenvolvedor é traduzir a especificação para um texto em língua natural e, talvez, complementar essa especificação com uma versão preliminar do manual do usuário, que descreva precisamente como o usuário deverá interagir futuramente com o sistema. Um outro produto da fase de análise e especificação de requisitos é o *plano de projeto*, baseado nos requisitos do produto a ser desenvolvido.

A fase de especificação de requisitos tem tarefas múltiplas e, para atingi-las, o desenvolvedor de software deve aplicar os princípios vistos na seção 1.3, especialmente *abstração*, *decomposição* e *generalização*. O software a ser produzido deve ser entendido e então descrito em diferentes níveis de abstração, dos aspectos gerais aos detalhes necessários. O produto deve ser dividido em partes, que possam ser analisadas separadamente. Uma possível lista do conteúdo do documento de especificação dos requisitos é a seguinte:

– *requisitos funcionais*: descrevem o que o produto de software faz, usando notações informais, semiformais, formais ou uma combinação delas;

– *requisitos não funcionais*: podem ser classificados nas categorias confiabilidade (disponibilidade, integridade, segurança), acurácia dos resultados, desempenho, problemas de interface homem–computador, restrições físicas e operacionais, questões de portabilidade etc.;

– *requisitos de desenvolvimento e manutenção*: incluem procedimentos de controle de qualidade — particularmente procedimentos de teste do sistema —, prioridades das funções desejadas, mudanças prováveis nos procedimentos de manutenção do sistema e outros.

Projeto

O projeto de software envolve a representação das funções do sistema em uma forma que possa ser transformada em um ou mais programas executáveis. Nessa fase, o produto é decomposto em partes tratáveis, e o resultado é um *documento de especificação do projeto*, que contém a descrição da arquitetura do software, isto é, o que cada parte deve fazer e a relação entre as partes. O processo de decomposição pode acontecer iterativamente e/ou pode ser feito através de níveis de abstração. Cada componente identificado em algum passo pode ser decomposto em subcomponentes. É comum distinguir entre projeto preliminar e detalhado, mas o significado desses termos pode variar. Para alguns, o projeto preliminar descreve a estrutura em termos de relações (por exemplo, *usa*, *é_composto_de*, *herda_de*), enquanto o projeto detalhado trata da especificação das interfaces. Outros usam esses termos para diferenciar entre a decomposição lógica (projeto em alto nível) e a decomposição física do programa em unidades de linguagem de programação. Outros, ainda, referem-se às principais estruturas de dados e aos algoritmos para cada módulo.

Implementação e teste unitário

Essa é a fase em que o projeto de software é transformado em um programa, ou unidades de programa, em uma determinada linguagem de programação. O resultado dessa fase é uma coleção de programas implementados e testados. A implementação também pode estar sujeita aos padrões da organização, que nesse caso pode definir o completo *layout* dos programas, tais como cabeçalhos para comentários, convenção para a utilização de nomes de variáveis e subprogramas, número máximo de linhas para cada componente e outros aspectos que a organização porventura achar importantes.

O teste unitário tem por objetivo verificar se cada unidade satisfaz suas especificações. Os testes são freqüentemente objeto de padronizações, que incluem uma definição precisa de um plano e critérios de testes a ser seguidos, definição do critério de completude (isto é, quando parar de testar) e o gerenciamento dos casos de teste. A depuração é uma atividade relacionada e também executada nessa fase. O teste das unidades é a principal atividade de controle de qualidade executada nessa fase, que ainda pode incluir inspeção de código para verificar se os programas estão de acordo com os padrões de codificação, estilo de programação disciplinada e outras qualidades do software, além da correção funcional (como, por exemplo, desempenho).

Integração e teste do sistema

Os programas ou unidades de programa são integrados e testados como um sistema completo para garantir que todos os seus requisitos sejam satisfeitos. A integração consiste na junção das unidades que foram desenvolvidas e testadas separadamente. Essa fase nem sempre é considerada separadamente da implementação; desenvolvimentos incrementais podem integrar e testar os componentes à medida que eles forem sendo desenvolvidos. Testes de integração envolvem testes das subpartes à medida que elas estão sendo integradas, e cada uma delas já deverá ter sido testada separadamente. Freqüentemente isso não é feito de uma só vez, mas progressivamente, em conjuntos cada vez maiores, a partir de pequenos subsistemas, até que o sistema inteiro seja construído. No final, são executados testes do sistema, e, uma vez que os testes tenham sido executados a contento, o produto de software pode ser colocado em uso. Padrões podem ser usados tanto na maneira como a integração é feita (por exemplo, *ascendente* ou *descendente*) como na maneira de projetar os dados de teste e documentar a atividade de teste. Depois de testado, o produto de software é entregue ao usuário.

Operação e manutenção

Normalmente esta é a fase mais longa do ciclo de vida. O sistema é instalado e colocado em uso. A entrega do software normalmente é feita em dois estágios. No primeiro, a aplicação é distribuída entre um grupo selecionado de usuários, para executar uma experiência controlada, obter *feedback* dos usuários e fazer alterações, se necessário, antes da entrega oficial. A manutenção é o conjunto de atividades executadas depois que o sistema é entregue aos usuários e consiste, basicamente, na correção dos erros remanescentes, que não foram descobertos nos estágios preliminares do ciclo de vida (manutenção corretiva), adaptação da aplicação às mudanças do ambiente (manutenção adaptativa), mudanças nos requisitos e adição de características e qualidades ao software (manutenção evolutiva).

Outras atividades

O paradigma clássico apresenta uma visão de desenvolvimento em fases. Algumas atividades, entretanto, são executadas antes que o ciclo de vida tenha início; outras, durante todo o ciclo de vida. Entre essas atividades estão: *estudo de viabilidade*, *documentação*, *verificação* e *gerenciamento*.

– **Estudo de viabilidade**:

  Esse estágio é crítico para o sucesso do resto do projeto, pois ninguém quer gastar tempo solucionando o problema errado. O conteúdo do estudo de viabilidade vai depender do tipo de desenvolvedor e do tipo de produto a ser desenvolvido. O objetivo dessa fase é produzir um *documento de estudo de viabilidade* que avalie os custos e benefícios da aplicação proposta. Para fazer isso, primeiro é necessário analisar o problema, pelo menos em nível global. Quanto melhor o problema for entendido, mais facilmente poderão ser identificados soluções alternativas, seus custos e potenciais benefícios para o usuário. Portanto, o ideal é fazer uma análise profunda do problema para produzir um estudo de viabilidade bem fundamentado. Na prática, entretanto, o estudo de

viabilidade é feito em um certo limite de tempo e sob pressão. Geralmente, o resultado desse estudo é uma oferta ao usuário potencial, e, como não se pode ter certeza de que a oferta será aceita, por razões econômicas não é possível investir muitos recursos nessa etapa. A identificação de soluções alternativas é baseada nessa análise preliminar, e, para cada solução, são analisados os custos e datas de entrega. Portanto, nessa fase é feita uma espécie de simulação do futuro processo de desenvolvimento, da qual é possível derivar informações que ajudem a decidir se o desenvolvimento vai valer a pena e, se for esse o caso, qual opção deve ser escolhida. O resultado do estudo de viabilidade é um documento que deve conter os seguintes itens: (a) a definição do problema; (b) soluções alternativas, com os benefícios esperados; e (c) as fontes necessárias, custos e datas de entrega para cada solução proposta.

– **Documentação**:

  Os produtos ou resultados de grande parte das fases são documentos; a mudança ou não de uma fase para outra vai depender desses documentos, e, normalmente, os padrões organizacionais definem a forma em que eles devem ser entregues.

– **Verificação e Validação**:

  Embora tenha sido dito que a verificação e a validação ocorrem em duas fases específicas (teste unitário e teste do sistema), elas são realizadas em várias outras fases. Em muitos casos, são executadas como um processo de controle de qualidade através de revisões e inspeções, com o objetivo de monitorar a qualidade do produto durante o desenvolvimento e não após a implementação. A descoberta e remoção de erros devem ser feitas o quanto antes para que a entrega do produto com erros seja evitada.

– **Gerenciamento**:

  A meta principal do gerenciamento é controlar o processo de desenvolvimento. Há três aspectos a ser considerados no gerenciamento. O primeiro diz respeito à *adaptação* do ciclo de vida ao processo, pois ele não deve ser tão rígido a ponto de ser aplicado exatamente da mesma forma a todos os produtos, indistintamente. Por exemplo, alguns procedimentos podem ser necessários para alguns produtos, mas excessivamente caros para aplicações mais simples. O segundo aspecto é a *definição de políticas*: como os produtos ou resultados intermediários vão ser armazenados, acessados e modificados, como as versões diferentes do sistema são construídas e quais as autorizações necessárias para acessar os componentes de entrada e saída do banco de dados do produto. Finalmente, o gerenciamento tem de lidar com todos os *recursos* que afetam o processo de produção de software, particularmente com os *recursos humanos*.

O paradigma do ciclo de vida clássico trouxe contribuições importantes para o processo de desenvolvimento de software, sendo as principais:

– o processo de desenvolvimento de software deve ser sujeito à disciplina, planejamento e gerenciamento;

– a implementação do produto deve ser postergada até que os objetivos tenham sido completamente entendidos.

Existem vários problemas com o paradigma clássico, sendo um deles a rigidez. O processo de desenvolvimento de software não é linear, envolve uma seqüência de iterações das atividades de desenvolvimento. Essas iterações estão representadas na Figura 1.3 pelas setas que retornam às atividades executadas anteriormente. O paradigma clássico, como proposto inicialmente, não contemplava essa volta às fases anteriores. Os resultados de uma fase eram "congelados" antes de se passar para a próxima fase. Dessa forma, o paradigma assumia que os requisitos e as especificações de projeto podiam ser "congelados" num estágio preliminar de desenvolvimento, quando o conhecimento sobre a aplicação pode ainda estar sujeito a mudanças.

Na prática, entretanto, todas essas fases se sobrepõem e fornecem informações umas para as outras. Durante a fase de projeto, podem ser identificados problemas com os requisitos; durante a implementação, podem surgir problemas com o projeto; durante a fase final do ciclo de vida, quando o software é instalado e posto em uso, erros e omissões nos requisitos originais podem ser descobertos, assim como erros de projeto e de implementação. Novas funcionalidades também podem ser identificadas, e modificações podem se tornar necessárias. Para fazer essas mudanças, pode ser necessário repetir alguns ou todos os estágios anteriores.

Todavia, a meta do paradigma clássico continua sendo tentar a linearidade, para manter o processo previsível e fácil de monitorar. Os planos são baseados nessa linearidade, e qualquer desvio é desencorajado, pois vai representar um desvio do plano original e, portanto, requerer um replanejamento.

Finalmente, nesse paradigma todo o planejamento é orientado para a entrega do produto de software em uma data única; toda a análise é executada antes do projeto e da implementação, e a entrega pode ocorrer muito tempo depois. Quando se cometem erros de análise e quando isto não é identificado durante as revisões, o produto pode ser entregue ao usuário com erros, depois de muito tempo e esforços terem sido gastos. Além disso, como o processo de desenvolvimento de sistemas complexos pode ser longo, demandando talvez anos, a aplicação pode ser entregue quando as necessidades do usuário já tiverem sido alteradas, o que vai requerer mudanças imediatas na aplicação.

Apesar de suas limitações, o paradigma do ciclo de vida clássico ainda é um modelo de processo bastante utilizado, especialmente quando os requisitos estão bem claros no início do desenvolvimento. Nas próximas seções, serão apresentados dois paradigmas alternativos que tentam sanar os problemas do ciclo de vida clássico, especialmente no que diz respeito ao "congelamento" dos requisitos.

1.4.2 O paradigma evolutivo

O paradigma evolutivo é baseado no desenvolvimento e implementação de um produto inicial, que é submetido aos comentários e críticas do usuário; o produto vai sendo refinado através de múltiplas versões, até que o produto de software almejado tenha sido desenvolvido.

As atividades de desenvolvimento e validação são desempenhadas paralelamente, com um rápido *feedback* entre elas. O paradigma evolutivo pode ser de dois tipos:

1) *Desenvolvimento exploratório*, em que o objetivo do processo é trabalhar junto do usuário para descobrir seus requisitos, de maneira incremental, até que o produto final seja obtido. O desenvolvimento começa com as partes do produto que são mais bem entendidas, e a evolução acontece quando novas características são adicionadas à medida que são sugeridas pelo usuário. O desenvolvimento exploratório é importante quando é difícil, ou mesmo impossível, estabeler uma especificação detalhada dos requisitos do sistema *a priori*. A Figura 1.4, adaptada de SOM 96, apresenta as atividades do desenvolvimento exploratório.

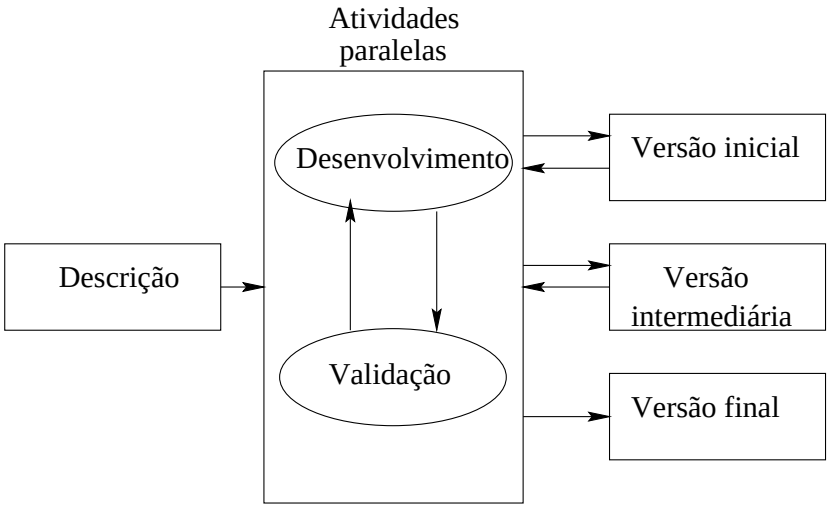

Figura 1.4: Desenvolvimento exploratório.

Primeiramente é desenvolvida uma versão inicial do produto, que é submetida a uma avaliação inicial do usuário. Essa versão é refinada, gerando várias versões, até que o produto almejado tenha sido desenvolvido.

2) *Protótipo descartável*, cujo objetivo é entender os requisitos do usuário e, conseqüentemente, obter uma melhor definição dos requisitos do sistema. O protótipo se concentra em fazer experimentos com os requisitos do usuário que não estão bem entendidos e envolve projeto, implementação e teste, mas não de maneira formal ou completa. Muitas vezes o usuário define uma série de objetivos para o produto de software, mas não consegue identificar detalhes de entrada, processamento ou requisitos de saída. Outras vezes, o desenvolvedor pode estar incerto sobre a eficiência de um algoritmo, a adaptação de um sistema operacional ou ainda sobre a forma da interação homem–máquina. Nessas situações, o protótipo pode ser a melhor opção. É um processo que possibilita ao desenvolvedor criar um modelo do software que será construído. Por um lado, o desenvolvedor pode perceber as reações iniciais do usuário e obter sugestões

para mudar ou inovar o protótipo; por outro, o usuário pode relacionar o que vê no protótipo diretamente com os seus requisitos. A Figura 1.5, adaptada de JAL 97, apresenta as atividades do paradigma do protótipo descartável.

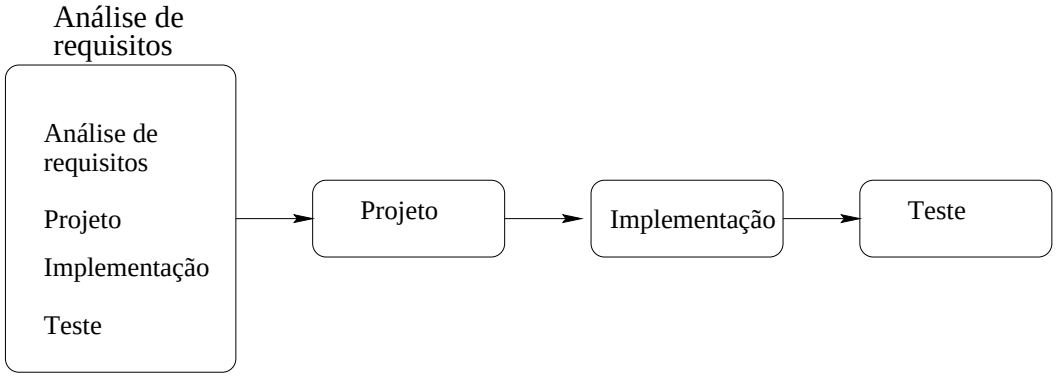

Figura 1.5: Protótipo descartável.

O desenvolvimento do protótipo descartável começa depois que uma versão preliminar da especificação de requisitos é obtida. Após o desenvolvimento do protótipo, é dada aos usuários a oportunidade de usá-lo e "brincar" com ele; baseados nessa experiência, eles dão opiniões sobre o que está correto, o que precisa ser modificado, o que está faltando, o que é desnecessário etc. Os desenvolvedores, então, modificam o protótipo para incorporar aquelas mudanças facilmente incorporáveis, e é dada aos usuários nova chance de utilizá-lo. Esse ciclo se repete até que os desenvolvedores concluam que o custo e o tempo envolvidos em novas mudanças não irão compensar as informações que porventura venham a ser obtidas. Baseados em todas as informações obtidas, os desenvolvedores, então, modificam os requisitos iniciais, com o objetivo de produzir a especificação de requisitos definitiva, que será usada para desenvolver o produto de software.

O paradigma evolutivo é normalmente mais efetivo do que o ciclo de vida clássico no desenvolvimento de produtos de software que atendam aos requisitos do usuário. Entretanto, sob a perspectiva de engenharia e do gerenciamento, o paradigma evolutivo apresenta os seguintes problemas:

- o processo não é visível, pois, como o desenvolvimento acontece de maneira rápida, não compensa produzir documentos que reflitam cada versão do produto de software. Entretanto, os gerentes de projeto precisam de documentos para medir o progresso no desenvolvimento;

- os sistemas são pobremente estruturados, pois as mudanças constantes tendem a corromper a estrutura do software. Portanto, a sua evolução provavelmente será difícil e custosa;

– quando um protótipo a ser descartado é construído, o usuário vê o que parece ser uma versão em funcionamento do produto de software, sem saber que o protótipo se mantém unido artificialmente e que, na pressa de colocá-lo em funcionamento, questões de qualidade e manutenção a longo prazo foram ignoradas. Quando informado de que o produto deve ser reconstruído, o usuário faz pressão para que algumas "pequenas" mudanças sejam feitas, para que o protótipo se transforme em produto final. Muitas vezes, o gerente de desenvolvimento de software cede;

– o desenvolvedor, muitas vezes, assume certos compromissos de implementação para garantir que o produto esteja funcionando rapidamente. Um sistema operacional ou uma linguagem inapropriados podem ser usados simplesmente porque estão disponíveis e são conhecidos; um algoritmo ineficiente pode ser implementado simplesmente para demonstrar as possibilidades do sistema. Depois de um certo tempo, o desenvolvedor pode se tornar familiarizado com essas escolhas e esquecer as razões pelas quais elas são inapropriadas. Uma escolha inapropriada pode se tornar parte integrante do sistema.

O desenvolvimento evolutivo é mais apropriado para sistemas pequenos, pois os problemas de mudanças no sistema atual podem ser contornados através da reimplementação do sistema todo quando mudanças substanciais se tornam necessárias. Uma outra vantagem deste tipo de abordagem é que os testes podem ser mais efetivos, visto que testar cada versão do sistema é provavelmente mais fácil do que testar o sistema todo no final.

1.4.3 O paradigma espiral

Também conhecido como paradigma de Boehm [BOE 91], foi desenvolvido para englobar as melhores características do ciclo de vida clássico e do paradigma evolutivo, ao mesmo tempo em que adiciona um novo elemento — a *análise de risco* — que não existe nos dois paradigmas anteriores.

Riscos são circunstâncias adversas que podem atrapalhar o processo de desenvolvimento e a qualidade do produto a ser desenvolvido. O paradigma espiral prevê a eliminação de problemas de alto risco através de um planejamento e projeto cuidadosos, em vez de tratar tanto problemas triviais como não triviais da mesma forma. Como o próprio nome sugere, as atividades do paradigma podem ser organizadas como uma espiral que tem muitos ciclos. Cada ciclo na espiral representa uma fase do processo de desenvolvimento do software. Portanto, o primeiro ciclo pode estar relacionado com o estudo de viabilidade e com a operacionalidade do sistema, isto é, com as funcionalidades e características do sistema e com o ambiente no qual será desenvolvido; o próximo ciclo pode estar relacionado com a definição dos requisitos, o próximo com o projeto do sistema e assim por diante. Não existem fases fixas neste paradigma. É durante o planejamento que se decide como estruturar o processo de desenvolvimento de software em fases. O paradigma, representado pela espiral da Figura 1.6, define quatro atividades principais representadas pelos quatro quadrantes da figura:

1) *Definição dos objetivos, alternativas e restrições*: os objetivos para a fase de desenvolvimento são definidos, tais como desempenho e funcionalidade, e são determinadas

alternativas para atingir esses objetivos; as restrições relativas ao processo e ao produto são também determinadas; um plano inicial de desenvolvimento é esboçado e os riscos de projeto são identificados. Estratégias alternativas, dependendo dos riscos detectados, podem ser planejadas, como, por exemplo, comprar o produto em vez de desenvolvê-lo.

2) *Análise de risco*: para cada um dos riscos identificados é feita uma análise cuidadosa. Atitudes são tomadas visando à redução desses riscos. Por exemplo, se houver risco de que os requisitos estejam inapropriados ou incompletos, um protótipo pode ser desenvolvido.

3) *Desenvolvimento e validação*: após a avaliação dos riscos, um paradigma de desenvolvimento é escolhido. Por exemplo, se os riscos de interface com o usuário forem predominantes, o paradigma de prototipagem evolutiva pode ser o mais apropriado.

4) *Planejamento*: o projeto é revisado, e a decisão de percorrer ou não mais um ciclo na espiral é tomada. Se a decisão for percorrer mais um ciclo, então o próximo passo do desenvolvimento do projeto deve ser planejado.

À medida que a volta na espiral acontece (começando de dentro para fora), versões mais completas do software vão sendo progressivamente construídas. Durante a primeira volta na espiral, são definidos objetivos, alternativas e restrições. Se a análise de risco indicar que há incerteza nos requisitos, a prototipagem pode ser usada para auxiliar o desenvolvedor e o usuário. Simulações e outros modelos podem ser usados para melhor definir o problema e refinar os requisitos. O processo de desenvolvimento tem início, e o usuário avalia o produto e faz sugestões de modificações. Com base nos comentários do usuário, ocorre então a próxima fase de planejamento e análise de risco. Cada volta na espiral resulta em uma decisão "continue" ou "não continue". Se os riscos forem muito grandes, o projeto pode ser descontinuado. Na maioria dos casos, o fluxo em volta da espiral continua, com cada passo levando os desenvolvedores em direção a um modelo mais completo do sistema e, em última instância, ao próprio sistema.

A diferença mais importante entre o paradigma espiral e os outros paradigmas é a análise de risco. O paradigma espiral possibilita ao desenvolvedor entender e reagir aos riscos em cada ciclo. Usa prototipagem como um mecanismo de redução de risco e mantém o desenvolvimento sistemático sugerido pelo ciclo de vida clássico. Incorpora, ainda, um componente iterativo que reflete o mundo mais realisticamente. Demanda uma consideração direta dos riscos envolvidos em todos os estágios do projeto e, se aplicado corretamente, reduz os riscos antes que eles se tornem problemáticos. Esse paradigma exige um especialista em análise de risco, e o sucesso em sua utilização reside exatamente no conhecimento desse especialista. Se um grande risco não for descoberto, problemas certamente ocorrerão. Os riscos podem ser diminuídos, ou mesmo sanados, através da descoberta de informações que venham a diminuir a incerteza com relação ao desenvolvimento.

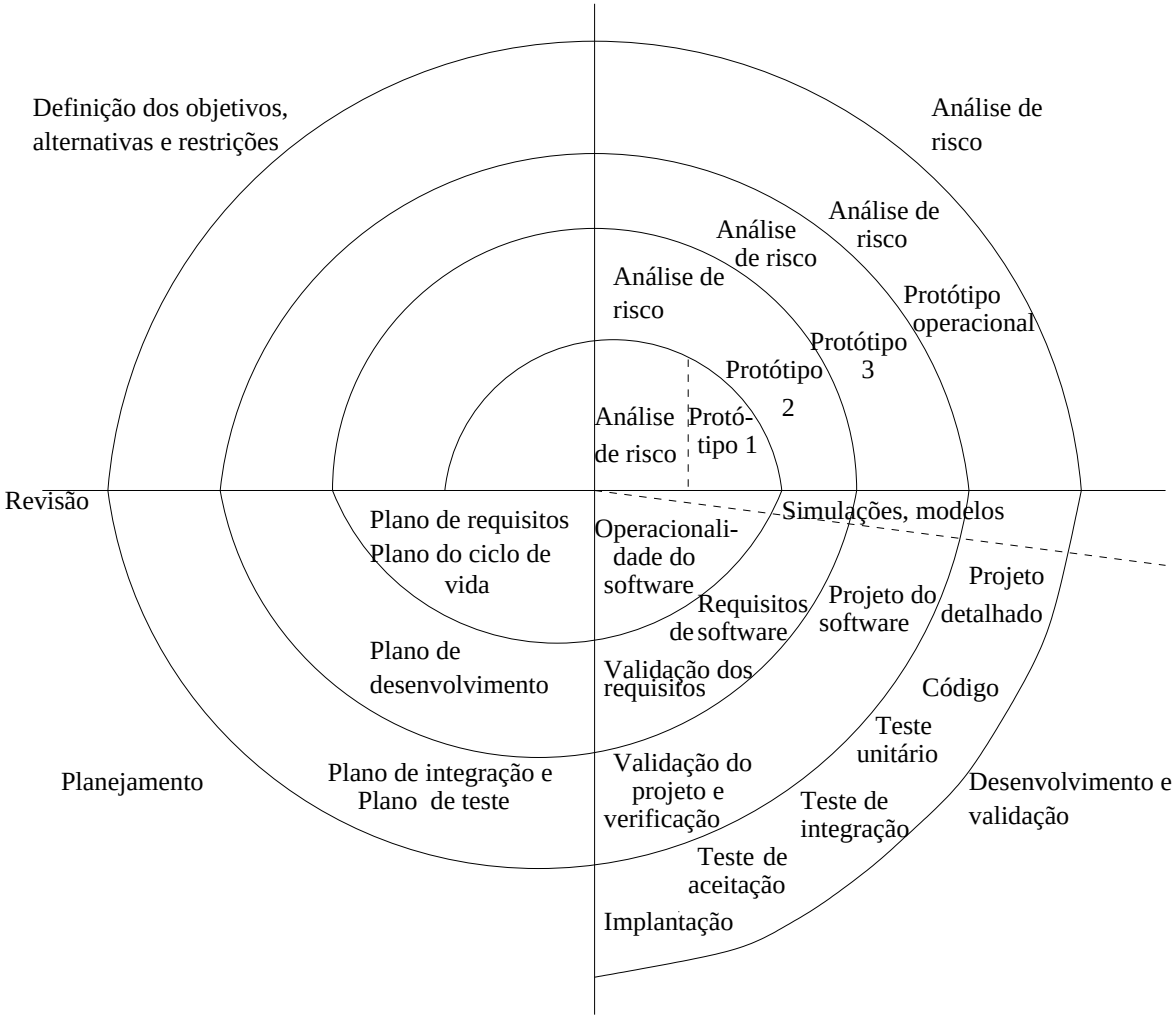

Figura 1.6: O paradigma espiral.

1.5 Engenharia de software influenciando e sendo influenciada por outras áreas dentro e fora da computação

A influência da engenharia de software pode ser notada em praticamente todas as áreas da computação e vice-versa. Entretanto, em algumas delas essa influência é mais óbvia. A seguir são apresentadas algumas dessas áreas.

A *teoria da computação* tem desenvolvido modelos que se tornaram ferramentas úteis para a engenharia de software, como, por exemplo, autômatos de estados finitos, que têm servido como base para o desenvolvimento de software. Autômatos têm sido utilizados, por exemplo, para especificação de sistemas operacionais, interfaces homem–computador e para a construção de processadores para tais especificações.

A *semântica formal* permite que se raciocine sobre as propriedades dos sistemas de software, e sua importância cresce proporcionalmente ao tamanho e complexidade do sistema que está sendo desenvolvido. Por exemplo, se os requisitos de um sistema de software que controla o fluxo em linhas de trem determinam que deve existir, no máximo, um trem em qualquer parte dos trilhos de uma determinada linha, então a semântica formal possibilita a produção de uma prova de que o software sempre terá esse comportamento [GEO 95].

As *linguagens de programação* são as principais ferramentas utilizadas no desenvolvimento de software e, por isso, têm uma influência muito grande no alcance dos objetivos da engenharia de software. Por outro lado, esses objetivos influenciam o desenvolvimento de linguagens de programação, visto que as linguagens mais modernas permitem a inclusão de características modulares, tais como a compilação independente e a separação entre especificação e implementação, para permitir o desenvolvimento de sistemas de software grandes, normalmente por uma equipe de desenvolvedores. Existem ainda outras linguagens que permitem o desenvolvimento de "pacotes", possibilitando a separação entre a interface e sua implementação, assim como bibliotecas de pacotes que podem ser usados como componentes no desenvolvimento de sistemas de software independentes.

O desenvolvimento dos *compiladores* modernos é feito de maneira modular e envolve basicamente dois componentes: análise da linguagem e geração do código propriamente dito. Essa decomposição permite a reutilização do segundo componente no desenvolvimento de outros compiladores. Essa técnica é usada na construção da família de compiladores GNU, por exemplo, para atender a diferentes arquiteturas e linguagens de programação de alto nível. A escrita do menor número possível de linhas de código só se torna viável através da reutilização de código, que é um conceito bastante utilizado em engenharia de software.

Os *sistemas operacionais* modernos fornecem ferramentas que possibilitam o gerenciamento da configuração do software, isto é, a manutenção e o controle das relações entre os diferentes componentes e versões do sistema de software. A grande vantagem do sistema operacional Unix [BOU 83] sobre seus antecessores é a sua organização como um conjunto de programas simples que podem ser combinados com grande flexibilidade, graças a uma "interface" comum (arquivos-texto).

Na área de *bancos de dados*, podem-se encontrar linguagens de consulta a bancos de dados, que permitem que aplicações usem os dados sem se preocupar com a representação interna deles. O banco de dados pode ser alterado para, por exemplo, melhorar o desempenho do sistema, sem nenhuma mudança na aplicação. Bancos de dados também podem ser usados como componentes de sistemas de software, visto que os primeiros já resolvem muitos dos problemas associados com o gerenciamento do acesso concorrente, por múltiplos usuários, a grandes quantidades de informações.

Sistemas multiagentes são sistemas complexos, cujo desenvolvimento envolve a decomposição do produto e a conseqüente separação das preocupações. Por exemplo, o desenvolvimento de um sistema multiagente para processamento de textos é um processo complexo, que pode ser decomposto em subprocessos (análise sintática, semântica, pragmática etc.) para resolver o problema em questão.

Sistemas especialistas são sistemas modulares, com dois componentes distintos: os fatos

conhecidos pelo sistema e as regras usadas para processar esses fatos, como, por exemplo, uma regra para decidir o curso de uma determinada ação. A idéia é que o conhecimento sobre uma determinada aplicação é dado pelo usuário, e os princípios gerais de como aplicar esse conhecimento a qualquer problema são fornecidos pelo próprio sistema.

Existem também outras áreas que têm tratado de problemas que não são específicos da engenharia de software, mas cujas soluções têm sido adaptadas a ela. Questões relativas ao gerenciamento, tais como estimativas de projeto, cronograma, planejamento dos recursos humanos, decomposição e atribuição de tarefas e acompanhamento do projeto, assim como questões pessoais envolvendo contratação e atribuição da tarefa certa à pessoa certa, estão diretamente relacionadas à engenharia de software. A ciência do gerenciamento estuda essas questões, e muitos dos modelos desenvolvidos nessa área podem ser aplicados à engenharia de software.

Outra área cujos estudos têm contribuído para a engenharia de software é a engenharia de sistemas, que estuda sistemas complexos, partindo da hipótese de que certas leis governam o comportamento de qualquer sistema complexo, que por sua vez é composto de muitos componentes com relações complexas. O software é normalmente um componente de um sistema muito maior, e, portanto, técnicas de engenharia de sistemas podem ser aplicadas no estudo desses sistemas.

1.6 Comentários finais

Em muitos casos, os paradigmas podem ser combinados de forma que os "pontos fortes" de cada um possam ser utilizados em um único projeto. O paradigma espiral já faz isso diretamente, combinando prototipagem e elementos do ciclo de vida clássico em um paradigma evolutivo. Entretanto, qualquer um desses pode servir como alicerce no qual outros paradigmas podem ser integrados. O processo sempre começa com a determinação dos objetivos, alternativas e restrições, que algumas vezes é chamada de obtenção de requisitos preliminar. Depois disso, qualquer caminho pode ser tomado. Por exemplo, os passos do ciclo de vida clássico podem ser seguidos se o sistema puder ser completamente especificado no começo. Se os requisitos não estiverem muito claros, um protótipo pode ser usado para melhor defini-los. Usando o protótipo como guia, o desenvolvedor pode retornar aos passos do ciclo de vida clássico (projeto, implementação e teste). Alternativamente, o protótipo pode evoluir para um sistema, retornando ao paradigma em cascata para ser testado. A natureza da aplicação é que vai determinar o paradigma a ser utilizado, e a combinação de paradigmas só tende a beneficiar o processo como um todo.

1.7 Exercício

1) Um gerente-geral de uma cadeia de lojas de presentes acredita que o único objetivo da construção de um protótipo é entender os requisitos do usuário e que depois esse protótipo será descartado. Portanto, ele acha bobagem gastar tempo e recursos em

algo que será desprezado mais tarde. Considerando essa relutância, resolva as seguintes questões:

(a) Compare brevemente o protótipo descartável com o desenvolvimento evolutivo, de forma que o gerente compreenda o que um protótipo pode significar.

(b) O gerente pensa em implementar o sistema, implantá-lo e testá-lo em uma loja e, depois, se obtiver sucesso, instalá-lo nas outras cinco lojas da cadeia. Diga qual método de prototipagem deve ser usado e justifique sua escolha.