CAPÍTULO

8 CONCEITOS DE PROJETO

CONCEITOS-CHAVE

A atividade de projeto de software[1] engloba o conjunto de princípios, conceitos e práticas que levam ao desenvolvimento de um sistema ou produto com alta qualidade. Os princípios de projeto estabelecem uma filosofia que prevalece sobre as atitudes e ações do desenvolvimento, orientando as atividades para realizar o projeto. Os conceitos de projeto devem ser estabelecidos e entendidos antes de aplicar a prática de projeto, que deve levar à criação de várias representações do software que servem como um guia para a atividade de construção que se segue.

A atividade projeto é crítica para o êxito em engenharia de software. No início dos anos 1990, Mitch Kapor, o criador do Lotus 1-2-3, apresentou um "manifesto de projeto de software" no *Dr. Dobbs Journal*. Disse ele:

> O que é projeto? É onde você fica com o pé em dois mundos — o mundo da tecnologia e o mundo das pessoas e dos propósitos do ser humano — e você tenta unir os dois...
>
> O crítico de arquitetura romano, Vitruvius, lançou a noção de que prédios bem projetados eram aqueles que apresentavam solidez, comodidade e deleite. O mesmo poderia ser dito em relação a software de boa qualidade. *Solidez*: um programa não deve apresentar nenhum bug que impeça seu funcionamento. *Comodidade*: um programa deve ser adequado aos propósitos para os quais foi planejado. *Deleite*: a experiência de usar o programa deve ser prazerosa. Temos aqui os princípios de uma teoria de projeto de software.

PANORAMA

O que é? Projeto é o que quase todo engenheiro quer fazer. É o lugar onde a criatividade impera — onde os requisitos dos interessados, as necessidades da aplicação e considerações técnicas se juntam na formulação de um produto ou sistema. O projeto cria uma representação ou modelo do software, mas diferentemente do modelo de requisitos (que se concentra na descrição do "O que" é para ser feito: dos dados, função e comportamento necessários), o modelo de projeto indica "O Como" fazer, fornecendo detalhes sobre a arquitetura de software, estruturas de dados, interfaces e componentes fundamentais para implementar o sistema.

Quem realiza? Os engenheiros de software conduzem cada uma das tarefas de projeto.

Por que é importante? O projeto permite que se modele o sistema ou produto a ser construído. O modelo pode ser avaliado em termos de qualidade e aperfeiçoado ANTES de o código ser gerado, ou de os testes serem realizados, ou de os usuários finais se envolverem com grandes números. Projeto é o lugar onde a qualidade do software é estabelecida.

Quais são as etapas envolvidas? O projeto representa o software de várias formas diferentes. Primeiramente, a arquitetura do sistema ou do produto tem de ser representada. Em seguida, são modeladas as interfaces que conectam o software aos usuários finais a outros sistemas e a dispositivos, bem como seus próprios componentes internos. Por fim, os componentes de software usados para construir o sistema são projetados. Cada uma dessas visões representa uma diferente ação de projeto, mas todas devem estar de acordo com um conjunto de conceitos básicos de projeto que orientam o trabalho de projeto de software.

Qual é o artefato? Um modelo de projeto que engloba representações de arquitetura, interface, no nível de componentes e de utilização é o principal artefato gerado durante o projeto de software.

Como garantir que o trabalho foi realizado corretamente? O modelo de projeto é avaliado pela equipe de software em um esforço para determinar se ele contém erros, inconsistências ou omissões; se existem alternativas melhores; e se o modelo pode ser implementado de acordo com as restrições, prazo e orçamento estabelecidos.

1 N. de R.T.: o termo em inglês dessa atividade é *design*. Refere-se à atividade de engenharia cujo foco é definir "como" os requisitos estabelecidos do projeto devem ser implementados no software. É uma fase que se apresenta de maneira similar nas diversas especializações da engenharia como a Civil, Naval, Química e Mecânica.

O objetivo da atividade projetar é gerar um modelo ou representação que apresente solidez, comodidade e deleite. Para tanto, temos de praticar a diversificação e, depois, a convergência. Belady [Bel81] afirma que "diversificação é a aquisição de um repertório de alternativas, a matéria-prima do projeto: componentes, soluções de componentes e conhecimento, todos contidos em catálogos, livros-textos e na mente". Assim que esse conjunto diversificado de informações for montado, temos de escolher elementos do repertório que atendam os requisitos definidos pela engenharia de requisitos e pelo modelo de análise (Capítulos 5 ao 7). À medida que isso ocorre, são consideradas e rejeitadas alternativas e convergimos para "uma particular configuração de componentes e, portanto, a criação do produto final" [Bel81].

Diversificação e convergência combinam intuição e julgamento baseado na experiência de construção de entidades similares, um conjunto de princípios e/ou heurística que orientam a maneira por meio da qual o modelo evolui, um conjunto de critérios que permitem que a qualidade seja avaliada e um processo de iteração que, ao fim, leva a uma representação final do projeto. O projeto de software muda continuamente à medida que novos métodos, melhor análise e entendimento mais abrangente evoluem.[2] Mesmo hoje em dia, a maioria das metodologias de projeto de software carece da profundidade, flexibilidade e natureza quantitativa que normalmente estão associadas às disciplinas mais clássicas de engenharia de projeto. Entretanto, existem efetivamente métodos para projeto de software, critérios para qualidade de projeto estão disponíveis e notação de projeto pode ser aplicada. Neste capítulo, exploraremos os conceitos e princípios fundamentais aplicáveis a todos os projetos de software, os elementos do modelo de projeto e o impacto dos padrões no processo de projeto. Nos Capítulos 9 a 13 apresentaremos uma série de métodos de projeto de software à medida que são aplicados ao projeto da arquitetura, de interfaces e de componentes, bem como as abordagens de projeto baseada em padrões e orientada para a Web.

8.1 Projeto no Contexto da Engenharia de Software

"O milagre mais comum da engenharia de software é a transição da análise para o projeto e do projeto para o código."

Richard Due'

O projeto de software sempre deve começar levando em consideração os dados — a base para todos os demais elementos do projeto. Após estabelecida a base, a arquitetura tem de ser extraída. Só então devem se realizar outras tarefas de projeto.

O projeto de software reside no núcleo técnico da engenharia de software e é aplicado independentemente do modelo de processos de software utilizado. Iniciando assim que os requisitos de software tiverem sido analisados e modelados, o projeto de software é a última ação da engenharia de software da atividade de modelagem e prepara a cena para a **construção** (geração de código e testes).

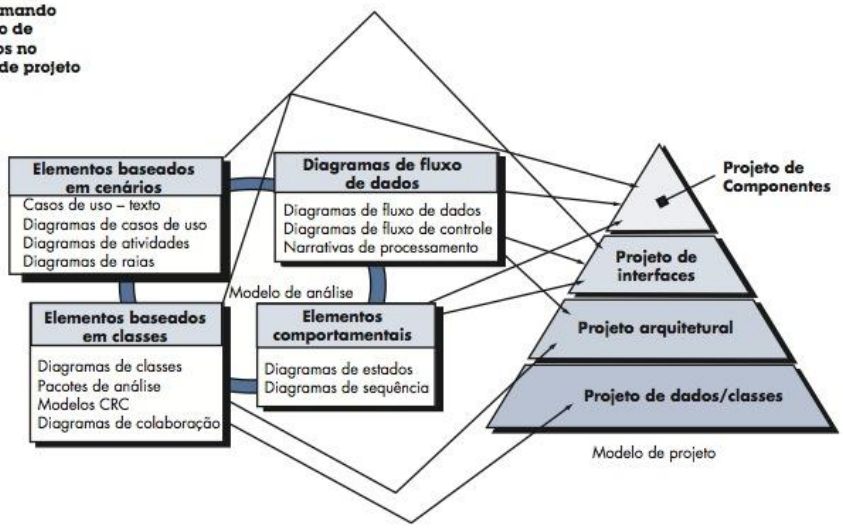
Cada um dos elementos do modelo de requisitos (Capítulos 6 e 7) fornece informações necessárias para criar os quatro modelos de projeto necessários para uma especificação completa. O fluxo de informações durante o projeto de software é ilustrado na Figura 8.1. O modelo de requisitos, manifestado por elementos baseados em cenários, baseados em classes, orientado a fluxos e comportamentais, alimenta a tarefa de projeto. Usando a notação de projeto e os métodos de projeto discutidos em capítulos posteriores, o projeto gera um projeto de dados/classes, um projeto de arquitetura, um projeto de interfaces e um projeto de componentes.

O projeto de dados/classes transforma os modelos de classes (Capítulo 6) em realizações de classes de projeto e nas estruturas de dados dos requisitos necessárias para implementar o software. Os objetos e os relacionamentos definidos nos cartões CRC e no conteúdo detalhado dos dados representados por atributos de classes e outra notação fornecem a base para a realização do projeto de dados. Parte do projeto de classes pode ocorrer com o projeto da arquitetura de software. O projeto de classe mais detalhado ocorre à medida que cada componente de software é projetado.

O projeto arquitetural define os relacionamentos entre os principais elementos estruturais do software, os estilos arquiteturais e padrões de projeto que podem ser usados para satisfazer os requisitos definidos para o sistema e as restrições que afetam o modo pelo qual a arquitetura

2 Aqueles leitores com maior interesse na filosofia do projeto de software talvez se interessem pela intrigante discussão de Philippe Kruchen sobre o projeto "pós-moderno" [Kru05a].

FIGURA 8.1
Transformando o modelo de requisitos no modelo de projeto



pode ser implementada [Sha96]. A representação do projeto arquitetural — a organização da solução técnica de um sistema baseado em computador — é derivada do modelo de requisitos.

O projeto de interfaces descreve como o software se comunica com sistemas que interromperam com ele e com as pessoas que o utilizam. Uma interface implica fluxo de informações (por exemplo, dados e/ou controle) e um tipo de comportamento específico. Consequentemente, modelos comportamentais e de cenários de uso fornecem grande parte das informações necessárias para o projeto de interfaces.

O projeto de componentes transforma elementos estruturais da arquitetura de software em uma descrição procedural dos componentes de software. As informações obtidas dos modelos baseados em classes, modelos de fluxo e modelos comportamentais servem como base para o projeto de componentes.

Durante o projeto tomamos decisões que, em última análise, afetarão o sucesso da construção do software e, igualmente importante, a facilidade de manutenção do software. Mas por que o projeto é tão importante?

A importância do projeto de software pode ser definida em uma única palavra — *qualidade*. Projeto é a etapa em que a qualidade é incorporada na engenharia de software. O projeto nos fornece representações de software que podem ser avaliadas em termos de qualidade. Projeto é a única maneira em que podemos traduzir precisamente os requisitos dos interessados em um produto ou sistema de software finalizado. O projeto de software serve como base para todas as atividades de apoio e da engenharia de software que seguem. Sem um projeto, corremos o risco de construir um sistema instável — um sistema que falhará quando forem feitas pequenas alterações; um sistema que talvez seja difícil de ser testado; um sistema cuja qualidade não pode ser avaliada até uma fase avançada do processo de software, quando o tempo está se esgotando e muito dinheiro já foi gasto.

> "Há duas maneiras de construir um projeto de software. Uma delas é fazê-lo tão simples que, obviamente, não existirão deficiências, e a outra maneira é fazê-lo tão complicado que não há nenhuma deficiência óbvia. O primeiro método é bem mais difícil."
>
> **C. A. R. Hoare**

CasaSegura

Projeto versus codificação

Cena: Sala do Ed, enquanto a equipe se prepara para traduzir os requisitos para o projeto.

Atores: Jamie, Vinod e Ed — todos membros da equipe de engenharia de software do *CasaSegura*.

Conversa:

Jamie: Você sabe, Doug [o gerente da equipe] está obcecado pelo projeto. Tenho que ser honesto; o que eu realmente adoro fazer é programar. Dê-me o C++ ou Java e ficarei feliz.

Ed: Nada... Você gosta de projetar.

Jamie: Você não está me ouvindo; programar é o canal.

Vinod: Acredito que aquilo que o Ed quis dizer é que você realmente não gosta de programar; você gosta de projetar e expressar isto na forma de código de programa. Código é a linguagem que você usa para representar um projeto.

Jamie: E o que há de errado nisso?

Vinod: Nível de abstração.

Jamie: Hã?

Ed: Uma linguagem de programação é boa para representar detalhes como estruturas de dados e algoritmos, mas não é tão boa para representar a colaboração componente-componente ou a arquitetura... Coisas do tipo.

Vinod: E uma arquitetura ferrada pode arruinar até mesmo o melhor código.

Jamie (pensando por um minuto): Portanto, você está dizendo que não posso representar arquitetura no código... Isso não é verdade.

Vinod: Certamente você pode envolver arquitetura no código, mas na maioria das linguagens de programação, é bem difícil ter uma visão geral rápida da arquitetura examinando-se o código.

Ed: E é isso que queremos antes de começar a programar.

Jamie: OK, talvez projetar e programar sejam coisas diferentes, mas ainda assim prefiro programar.

8.2 O Processo de Projeto

O projeto de software é um processo iterativo através do qual os requisitos são traduzidos em uma "planta" para construir o software. Inicialmente, a planta representa uma visão holística do software. O projeto é representado em um alto nível de abstração — um nível que pode ser associado diretamente ao objetivo específico do sistema e aos requisitos mais detalhados de dados, funcionalidade e comportamento. À medida que ocorrem as iterações do projeto, refinamento subsequente leva a representações do projeto em níveis de abstração cada vez mais baixos. Estes ainda podem ser associados aos requisitos, mas a conexão é mais sutil.

8.2.1 Diretrizes e atributos da qualidade de software

"Escrever um trecho de código inteligente que funcione é uma coisa; projetar algo que possa dar suporte a negócios duradouros é outra totalmente diferente."

C. Ferguson

Ao longo do processo de projeto, a qualidade do projeto que evolui é avaliada com uma série de revisões técnicas discutidas no Capítulo 15. McGlaughlin [McG91] sugere três características que servem como um guia para a avaliação de um bom projeto:

- O projeto deve implementar todos os requisitos explícitos contidos no modelo de requisitos e deve acomodar todos os requisitos implícitos desejados pelos interessados.
- O projeto deve ser um guia legível, compreensível para aqueles que geram código e para aqueles que testam e, subsequentemente, dão suporte ao software.
- O projeto deve dar uma visão completa do software, tratando os domínios de dados, funcional e comportamental sob a perspectiva de implementação.

Cada uma dessas características é, na verdade, uma meta do processo de projeto. Mas como cada uma é alcançada?

Diretrizes de qualidade. Para avaliar a qualidade da representação de um projeto, você e outros membros da equipe de software devem estabelecer critérios técnicos para um bom projeto. Na Seção 8.3, discutimos conceitos de projeto que também servem como critérios de qualidade de software. Por enquanto, consideremos as seguintes diretrizes:

Quais são as características de um bom projeto?

1. Um projeto deve exibir uma arquitetura que (1) foi criada usando estilos ou padrões arquiteturais reconhecíveis, (2) seja composta por componentes que apresentam boas características de projeto (discutidos mais adiante neste capítulo) e (3) possa ser implementada de uma forma evolucionária[3] facilitando, portanto, a implementação e os testes.
2. Um projeto deve ser modular; o software deve ser logicamente particionado em elementos ou subsistemas, de modo que seja fácil de testar e manter.
3. Um projeto deve conter representações distintas de: dados, arquitetura, interfaces e componentes.
4. Um projeto deve levar as estruturas de dados adequadas às classes a ser implementadas e baseadas em padrões de dados reconhecíveis.
5. Um projeto deve levar a componentes que apresentem características funcionais independentes (baixo acoplamento).
6. Um projeto deve levar a interfaces que reduzam a complexidade das conexões entre os componentes e o ambiente externo (encapsulamento).
7. Um projeto deve ser obtido usando-se um método repetível, isto é, dirigido por informações obtidas durante a análise de requisitos de software.
8. Um projeto deve ser representado usando-se uma notação que efetivamente comunique seu significado.

Essas diretrizes não são atingidas por acaso. Elas são alcançáveis por meio da aplicação de princípios de projeto fundamentais, de metodologia sistemática e de revisão.

"Qualidade não é algo que se coloque sobre os assuntos e objetos como um enfeite em uma árvore de Natal."

Robert Pirsig

Atributos de qualidade. A Hewlett-Packard [Gra87] desenvolveu um conjunto de atributos de qualidade de software ao qual foi atribuído o acrônimo FURPS — *functionality* (funcionalidade), *usability* (usabilidade), *reliability* (confiabilidade), *performance* (desempenho) e *supportability* (facilidade de suporte). Os atributos de qualidade FURPS representam uma meta para todo projeto de software:

- *A funcionalidade* é avaliada pela observação do conjunto de características e capacidades do programa, a generalidade das funções que são entregues e a segurança do sistema como um todo.

INFORMAÇÕES

Avaliação da qualidade do projeto — a revisão técnica

O projeto é importante porque permite à equipe de software avaliar a qualidade[4] do software antes de ser implementado — em um momento em que os erros, as omissões ou as inconsistências são fáceis e baratas de ser corrigidos. Mas como avaliar a qualidade durante um projeto? O software não pode ser testado, pois não existe nenhum software executável para testar. O que fazer então?

Durante um projeto, a qualidade é avaliada realizando-se uma série de revisões técnicas (technical reviews, TRs). As TRs são discutidas em detalhes no Capítulo 15[5] mas vale fazer um resumo neste momento. A revisão técnica é uma reunião conduzida por membros da equipe de software. Normalmente duas, três ou quatro pessoas participam, dependendo do escopo das informações de projeto a ser revisadas. Cada pessoa desempenha um papel: um *líder de revisão* planeja a reunião, estabelece uma agenda e conduz a reunião; o *registrador* toma notas de modo que nada seja perdido; o *produtor* é a pessoa cujo artefato (por exemplo, o projeto de um componente de software) está sendo revisado. Antes de uma reunião, cada pessoa da equipe de revisão recebe uma cópia do artefato do projeto e lhes é solicitada sua leitura, procurando encontrar erros, omissões ou ambiguidade. Quando a reunião começa, o intuito é perceber todos os problemas com o artefato, de modo que possam ser corrigidos antes da implementação começar. A TR dura, tipicamente, entre 90 minutos e 2 horas. Na conclusão, a equipe de revisão determina se outras ações são necessárias por parte do produtor antes de o artefato do projeto ser aprovado como parte do modelo de projeto final.

3 Evolucionária: incremental, com entregas por partes. Para sistemas menores, algumas vezes o projeto pode ser desenvolvido linearmente.

4 Os fatores de qualidade discutidos no Capítulo 23 podem ajudar a equipe de revisão à medida que avalia a qualidade.

5 Você deve considerar consulta do Capítulo 15 neste momento. As revisões técnicas são parte crítica do processo de projeto e é um importante mecanismo para atingir qualidade em um projeto.

Os projetistas de software tendem a se concentrar no problema a ser resolvido. Apenas não se esqueça de que os atributos de qualidade FURPS sempre fazem parte do problema. Eles têm de ser considerados.

- *A usabilidade* é avaliada considerando-se fatores humanos (Capítulo 11), estética, consistência e documentação como um todo.
- *A confiabilidade* é avaliada medindo-se a frequência e a severidade das falhas, a precisão dos resultados gerados, o tempo médio entre defeitos (*mean-time-to-failure*, MTTF), a capacidade de se recuperar de uma falha e a previsibilidade do programa.
- *O desempenho* é medido considerando-se a velocidade de processamento, o tempo de resposta, o consumo de recursos, vazão (*throughput*) e eficiência.
- *A facilidade de suporte* combina a habilidade de estender o programa (extensibilidade), a adaptabilidade e a reparabilidade — esses três atributos representam um termo mais comum, *facilidade de manutenção* — e, além disso, a facilidade de realizar testes, a compatibilidade, a facilidade de configurar (a habilidade de organizar e controlar elementos da configuração do software, Capítulo 22), a facilidade com a qual um sistema pode ser instalado, bem como a facilidade com a qual os problemas podem ser localizados.

Nem todo atributo de qualidade de software tem o mesmo peso à medida que o projeto é desenvolvido. Uma aplicação poderia enfatizar a funcionalidade com ênfase especial na segurança. Outra poderia demandar desempenho com particular ênfase na velocidade de processamento. Um terceiro foco poderia ser na confiabilidade. Independentemente do peso dado, é importante notar que esses atributos de qualidade devem ser considerados quando o projeto se inicia, e *não* após o projeto estar completo e a construção tiver começado.

8.2.2 A evolução de um projeto de software

"Um projetista sabe que seu projeto atingiu a perfeição não quando não resta mais nada a ser acrescentado, mas quando não há mais nada a ser eliminado."

Antoine de St-Exupéry

A evolução de um projeto de software é um processo contínuo que já atinge quase seis décadas. Os trabalhos iniciais concentravam-se em critérios para o desenvolvimento de programas modulares [Den73] e de métodos para refinamento das estruturas de software de uma forma *topdown* [Wir71]. Aspectos procedurais da definição de um projeto evoluíram e convergiram para uma filosofia denominada *programação estruturada* [Dah72], [Mil72]. Trabalhos posteriores propuseram métodos para a tradução de fluxos de dados [Ste74] ou da estrutura de dados (por exemplo, [Jac75], [War74]) em uma definição de projeto. Abordagens de projeto mais recentes (por exemplo, [Jac92], [Gam95]) propuseram uma abordagem orientada a objetos para a derivação do projeto. Atualmente, a ênfase em projeto de software tem sido na arquitetura de software [Kru06] e nos padrões de projeto que podem ser utilizados para implementar arquiteturas de software e níveis de abstração de projeto mais baixos (por exemplo, [Hol06] [Sha05]). Tem crescido a ênfase em métodos orientados a aspectos (por exemplo, [Cla05], [Jac04]), no desenvolvimento dirigido a modelos [Sch06] e dirigido a testes [Ast04] que enfatizam técnicas para se atingir uma modularidade e estrutura arquitetural mais efetiva nos projetos criados.

Quais características são comuns a todos os métodos de projeto?

Uma série de métodos de projetos, originários dos trabalhos citados, está sendo aplicada em toda a indústria. Assim como os métodos de análise apresentados nos Capítulos 6 e 7, cada método de projeto de software introduz heurística e notação únicas, bem como uma visão um tanto provinciana daquilo que caracteriza a qualidade de um projeto. Mesmo assim, todos os métodos possuem uma série de características comuns: (1) um mecanismo para a tradução do modelo de requisitos em uma representação de projeto, (2) uma notação para representar componentes funcionais e suas interfaces, (3) heurística para refinamento e particionamento e (4) diretrizes para avaliação da qualidade.

Independentemente do método de projeto utilizado, devemos aplicar um conjunto de conceitos básicos ao projeto de dados, de arquitetura, de interface e dos componentes. Tais conceitos são considerados nas seções a seguir.

CONJUNTO DE TAREFAS

Conjunto de tarefas genéricas para projeto

1. Examinar o modelo do domínio de informação e projetar estruturas de dados apropriadas para objetos de dados e seus atributos.
2. Usar o modelo de análise, selecionar um estilo de arquitetura apropriado ao software.
3. Dividir o modelo de análise em subsistemas de projeto e alocá-los na arquitetura:
   Certificar-se de que cada subsistema seja funcionalmente coeso.
   Projetar interfaces de subsistemas.
   Alocar classes ou funções de análise para cada subsistema.
4. Criar um conjunto de classes ou componentes de projeto:
   Traduzir a descrição de classes de análise em uma classe de projeto.
   Verificar cada classe de projeto em relação aos critérios de projeto; considerar questões de herança.
   Definir métodos e mensagens associadas a cada classe de projeto.
   Avaliar e selecionar padrões de projeto para uma classe ou um subsistema de projeto.
   Rever as classes de projeto e revisar quando necessário.
5. Projetar qualquer interface necessária para sistemas ou dispositivos externos.
6. Projetar a interface do usuário:
   Revisar os resultados da análise de tarefas.
   Especificar a sequência de ações baseando-se nos cenários de usuário.
   Criar um modelo comportamental da interface.
   Definir objetos de interface, mecanismos de controle.
   Rever o projeto de interfaces e revisar quando necessário.
7. Conduzir o projeto de componentes.
   Especificar todos os algoritmos em um nível de abstração relativamente baixo.
   Refinar a interface de cada componente.
   Definir estruturas de dados dos componentes.
   Revisar cada componente e corrigir todos os erros descobertos.
8. Desenvolver um modelo de implantação.

8.3 Conceitos de Projeto

Um conjunto de conceitos fundamentais de projeto de software evoluiu ao longo da história da engenharia de software. Embora o grau de interesse em cada conceito tenha variado ao longo dos anos, cada um resistiu ao tempo. Esses conceitos fornecem ao projetista de software uma base a partir da qual métodos de projeto mais sofisticados podem ser aplicados. Ajudam-nos a responder as seguintes questões:

- Quais critérios podem ser usados para particionar o software em componentes individuais?
- Como os detalhes de função ou estrutura de dados são separados de uma representação conceitual do software?
- Quais critérios uniformes definem a qualidade técnica de um projeto de software?

M. A. Jackson [Jac75] disse uma vez: "O princípio da sabedoria para um [engenheiro de software] é reconhecer a diferença entre fazer um programa funcionar e fazer com que ele funcione corretamente". Os conceitos fundamentais de projeto de software fornecem a organização necessária para estruturá-la e para "fazer com que ele funcione corretamente".

Nas seções a seguir, apresentamos uma breve visão de importantes conceitos de projeto de software que englobam tanto o desenvolvimento de software tradicional quanto o orientado a objetos.

"Abstração é uma das maneiras fundamentais como nós, seres humanos, lidamos com a complexidade."
Grady Booch

8.3.1 Abstração

Ao se considerar uma solução modular para qualquer problema, muitos níveis de abstração podem se apresentar. No nível de abstração mais alto, uma solução é expressa em termos abrangentes usando a linguagem do domínio do problema. Em níveis de abstração mais baixos, uma descrição mais detalhada da solução é fornecida. A terminologia do domínio do problema é associada à terminologia de implementação para definir uma solução. Por fim, no nível de

AVISO

Como projetista, trabalhe arduamente para derivar tanto as abstrações procedurais quanto a de dados que atendam ao problema em questão. Se elas puderem atender um domínio inteiro dos problemas, tanto melhor.

abstração mais baixo, a solução técnica do software é expressa de maneira que pode ser diretamente implementada.

À medida que diferentes níveis de abstração são alcançados, usa-se a combinação de abstrações procedurais e de dados. Uma *abstração procedural* refere-se a uma sequência de instruções que possuem uma função específica e limitada. O nome de uma abstração procedural implica sua função, porém os detalhes específicos são omitidos. Um exemplo de abstração procedural tem como nome *abrir* para uma porta. *Abrir* implica uma longa sequência de etapas procedurais (por exemplo, dirigir-se até a porta, alcançar e agarrar a maçaneta, girar a maçaneta e puxar a porta, afastar-se da porta em movimento etc.).[6]

A *abstração de dados* é um conjunto de dados com nome que descreve um objeto de dados. No contexto da abstração procedural *abrir,* podemos definir uma abstração de dados chamada **porta.** Assim como qualquer objeto de dados, a abstração de dados para **porta** englobaria um conjunto de atributos que descrevem a porta (por exemplo, tipo de porta, mudar de direção, mecanismo de abertura, peso, dimensões). Daí decorre que a abstração *abrir* faria uso de informações contidas nos atributos da abstração de dados **porta**.

8.3.2 Arquitetura

WebRef

Uma discussão mais aprofundada de arquitetura de software pode ser encontrada em **www.sei.cmu.edu/ata/ata_init.html**.

A arquitetura de software refere-se à "organização geral do software e aos modos pelos quais disponibiliza integridade conceitual para um sistema" [Sha95a]. Em sua forma mais simples, arquitetura é a estrutura ou a organização de componentes de programa (módulos), a maneira através da qual esses componentes interagem e a estrutura de dados são usadas pelos componentes. Em um sentido mais amplo, entretanto, os componentes podem ser generalizados para representar os principais elementos de um sistema e suas interações.

Uma meta do projeto de software é derivar um quadro da arquitetura de um sistema. Esse quadro representa a organização a partir da qual atividades mais detalhadas de projeto são conduzidas. Um conjunto de padrões de arquitetura permite a um engenheiro de software reusar soluções-padrão para problemas similares.

"Arquitetura de software é o produto resultante do desenvolvimento que dá o maior retorno sobre o investimento em relação à qualidade, prazos e custo."

Len Bass et al.

Shaw e Garlan [Sha95a] descrevem um conjunto de propriedades que devem ser especificadas como parte de um projeto de arquitetura:

Propriedades estruturais. Esse aspecto do projeto da representação da arquitetura define os componentes de um sistema (por exemplo, módulos, objetos, filtros) e a maneira pela qual os componentes são empacotados e interagem entre si. Por exemplo, objetos são empacotados para encapsularem dados e processamento que manipula esses dados e interagem por meio da chamada dos métodos.

Propriedades não funcionais. A descrição do projeto de arquitetura deve tratar a maneira pela qual o projeto da arquitetura atinge os requisitos de desempenho, capacidade, confiabilidade, segurança, adaptabilidade e outras características do sistema, que não representam as funcionalidades diretamente acessadas pelos usuários.

Famílias de sistemas. O projeto de arquitetura deve tirar proveito de padrões reusáveis comumente encontrados no projeto de famílias de sistemas similares. Em essência, o projeto deve ter a habilidade de reutilizar os componentes que fazem parte da arquitetura.

AVISO

Não deixe que a arquitetura aconteça ao acaso. Ao fazer isso, você consumirá o restante do projeto adequando-o para a forma forçada ao projeto. Projete a arquitetura explicitamente.

Dada a especificação dessas propriedades, o projeto da arquitetura pode ser representado usando-se um ou mais modelos diferentes [Gar95]. Os *modelos estruturais* representam a arquitetura como um conjunto organizado de componentes de programa. Esses modelos aumentam o nível de abstração do projeto tentando identificar projetos arquiteturais reusáveis encontrados em tipos de aplicações similares. Os *modelos dinâmicos* tratam dos aspectos comportamentais da arquitetura do programa, indicando como a estrutura ou configuração do sistema pode mudar em função de eventos externos. Os *modelos de processos* concentram-se no projeto do

6 Deve-se notar, entretanto, que um conjunto de operações pode ser substituído por outro, desde que a função implicada pela abstração procedural permaneça a mesma. Consequentemente, as etapas necessárias para implementar *abrir* mudariam dramaticamente se a porta fosse automática e ligada a um sensor.

processo técnico ou do negócio que o sistema deve atender. Por fim, os *modelos funcionais* podem ser utilizados para representar a hierarquia funcional de um sistema.

Desenvolveu-se uma série de *linguagens de descrição de arquitetura* (*architectural description languages*, ADLs) diferente para representar esses modelos [Sha95b]. Embora tenham sido propostas diversas ADLs, a maioria fornece mecanismos para descrever componentes de sistema e a maneira através da qual estão conectados entre si.

Você deve notar que há certo debate em torno do papel da arquitetura no projeto. Alguns pesquisadores argumentam que a obtenção da arquitetura de software deve ser separada do projeto e ocorre entre as ações da engenharia de requisitos e ações de projeto mais convencionais. Outros acreditam que a obtenção da arquitetura é parte integrante do processo de projeto. A maneira pela qual a arquitetura de software é caracterizada e seu papel no projeto são discutidos no Capítulo 9.

"Cada padrão descreve um problema que ocorre repetidamente em nosso ambiente, e então descreve o núcleo da solução para esse problema, de uma forma tal que podemos usar a solução milhões de vezes, sem jamais fazê-lo da mesma forma."

Christopher Alexander

8.3.3 Padrões

Brad Appleton define *padrão de projeto* da seguinte maneira: "Padrão é parte de um conhecimento consolidado já existente que transmite a essência de uma solução comprovada para um problema recorrente em certo contexto, em meio a preocupações concorrentes" [App00]. Em outras palavras, um padrão de projeto descreve uma estrutura de projeto que resolve uma particular categoria de problemas de projeto em um contexto específico e entre "forças" que direcionam a maneira pela qual o padrão é aplicado e utilizado.

O intuito de cada padrão de projeto é fornecer uma descrição que permite a um projetista determinar (1) se o padrão se aplica ou não ao trabalho em questão, (2) se o padrão pode ou não ser reutilizado (e, portanto, poupando tempo) e (3) se o padrão pode servir como um guia para desenvolver um padrão similar, mas funcional ou estruturalmente diferente. Os padrões de projeto são discutidos de forma detalhada no Capítulo 12.

8.3.4 Separação por interesses (por afinidades)

A separação por interesses é um conceito de projeto [Dij82] que sugere que qualquer problema complexo pode ser tratado mais facilmente se for subdividido em trechos a ser resolvidos e/ou otimizados independentemente. *Interesse* se manifesta como uma característica ou comportamento especificados como parte do modelo de requisitos do software. Por meio da separação por interesses em blocos menores e, portanto, mais administráveis, um problema toma menos tempo para ser resolvido.

O argumento para a separação por interesses pode se estender muito mais. Se dividirmos um problema em um número excessivo de problemas muito pequenos, resolver cada um deles será fácil, porém, juntá-los em uma solução — integração — talvez seja muito difícil.

Para dois problemas, p_1 e p_2, se a complexidade percebida de p_1 for maior do que a percebida para p_2, segue que o esforço necessário para solucionar p_1 é maior do que o esforço necessário para solucionar p_2. Como caso geral, esse resultado é intuitivamente óbvio. Leva mais tempo para resolver um problema difícil.

Segue também que a complexidade percebida de dois problemas, quando estes são combinados, normalmente é maior do que soma da complexidade percebida quando cada um deles é tomado separadamente. Isso nos leva a uma estratégia dividir-para-conquistar — é mais fácil resolver um problema complexo quando o subdividimos em partes gerenciáveis. Isso tem implicações importantes em relação à modularidade do software.

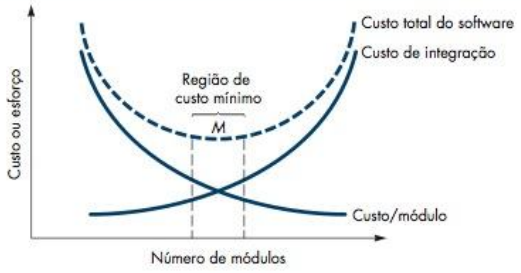
A separação por interesses manifesta-se em outros conceitos relacionados ao projeto: modularidade, aspectos, independência funcional e refinamento.

8.3.5 Modularidade

Modularidade é a manifestação mais comum da separação por interesses. O software é dividido em componentes separadamente especificados e endereçáveis, algumas vezes denominados *módulos*, que são integrados para satisfazer os requisitos de um problema.

Afirmou-se que "modularidade é o único atributo de software que possibilita que um programa seja intelectualmente gerenciável" [Mye78]. Software monolítico (um grande programa

FIGURA 8.2
Modularidade e custo do software



composto de um único módulo) não pode ser facilmente entendido por um engenheiro de software. O número de caminhos de controle, abrangência de referência, número de variáveis e complexidade geral tornaria o entendimento próximo do impossível. Em quase todos os casos, devemos dividir o projeto em vários módulos, para facilitar a compreensão e, consequentemente, reduzir o custo necessário para construir o software.

Recapitulando nossa discussão sobre separação por interesses, é possível concluir que, se subdividirmos o software indefinidamente, o esforço exigido para desenvolvê-lo seja pequeno! Infelizmente, outros fatores entram em jogo, invalidando essa conclusão. Referindo-se à Figura 8.2, o esforço (custo) para desenvolver um módulo de software individual realmente diminui à medida que o número total de módulos cresce. Dado o mesmo conjunto de requisitos, também com mais módulos, significa tamanho individual menor. Entretanto, à medida que o número de módulos aumenta, o esforço (custo) associado à integração dos módulos também cresce. Essas características levam a um custo total ou curva de esforço mostrada na figura. Existe um número *M* de módulos que resultaria em um custo de desenvolvimento mínimo, porém não temos a sofisticação suficiente para prever *M* com certeza.

? Qual o número exato de módulos para um dado sistema?

As curvas da Figura 8.2 nos dão uma útil orientação qualitativa quando a modularidade é considerada. Devemos modularizar, mas tomar cuidado para permanecer nas vizinhanças de *M*. *Devemos evitar* modularizar a menos ou a mais. Mas como saber a vizinhança de *M*? Quão modular devemos fazer com que um software seja? As respostas a essas perguntas exigem um entendimento de outros conceitos de projeto considerados posteriormente neste capítulo.

Modularizamos um projeto (e o programa resultante) de modo que o desenvolvimento possa ser planejado mais facilmente; incrementos de software possam ser definidos e entregues; as mudanças possam ser mais facilmente acomodadas; os testes e depuração possam ser conduzidos de forma mais eficaz e a manutenção no longo prazo possa ser realizada sem efeitos colaterais sérios.

8.3.6 Encapsulamento[7] de informações

O conceito de modularidade nos leva a uma questão fundamental: "Como decompor uma solução de software para obter o melhor conjunto de módulos?". O princípio de encapsulamento de informações [Par72] sugere que os módulos sejam "caracterizados por decisões de projeto que ocultem (cada uma delas) de todas as demais". Em outras palavras, os módulos devem ser especificados e projetados de modo que as informações (algoritmos e dados) contidas em um módulo sejam inacessíveis por parte de outros módulos que não necessitam tais informações, disponibilizando apenas os itens que interessam aos outros módulos.

7 N. de T.: Encapsular é uma técnica de engenharia largamente usada. Por exemplo, um componente de hardware digital é projetado da seguinte forma: esconde aspectos que não interessam aos demais componentes e publica aspectos úteis aos demais componentes.

PONTO-CHAVE

O intuito de encapsulamento de informações é esconder os detalhes das estruturas de dados e de processamento procedural que estão por trás da interface de acesso a um módulo. Os detalhes não precisam ser conhecidos por usuários do módulo.

Encapsular implica que efetiva modularidade pode ser conseguida por meio da definição de um conjunto de módulos independentes que passam entre si apenas as informações necessárias para realizar determinada função do software. A abstração ajuda a definir as entidades procedurais (ou informativas) que constituem o software. Encapsulamento define e impõe restrições de acesso tanto a detalhes procedurais em um módulo quanto em qualquer estrutura de dados local usada pelo módulo [Ros75].

O uso de encapsulamento de informações como critério de projeto para sistemas modulares fornece seus maiores benefícios quando são necessárias modificações durante os testes e, posteriormente, durante a manutenção do software. Como a maioria dos detalhes procedurais e de dados são ocultos para outras partes do software, erros introduzidos inadvertidamente durante a modificação em um módulo têm menor probabilidade de se propagar para outros módulos ou locais dentro do software.

8.3.7 Independência funcional

O conceito de independência funcional é um resultado direto da separação por interesses, da modularidade e dos conceitos de abstração e encapsulamento de informações. Em artigos marcantes sobre projeto de software, Wirth [Wir71] e Parnas [Par72] tratam técnicas de refinamento que aumentam a independência entre módulos. Trabalho posterior de Stevens, Myers e Constantine [Ste74] solidificaram o conceito.

A independência funcional é atingida desenvolvendo-se módulos com função "única" e com "aversão" à interação excessiva com outros módulos. Em outras palavras, devemos projetar software de modo que cada módulo atenda um subconjunto específico de requisitos e tenha uma interface simples quando vista de outras partes da estrutura do programa. É razoável perguntar por que a independência é importante.

? Por que devemos nos esforçar para criar módulos independentes?

Software com efetiva modularidade, isto é, módulos independentes, é mais fácil de ser desenvolvido, pois a função pode ser compartimentalizada e as interfaces simplificadas (considere as consequências quando o desenvolvimento é conduzido por uma equipe). Módulos independentes são mais fáceis de ser mantidos (e testados), pois efeitos colaterais provocados por modificação no código ou projeto são limitados, a propagação de erros é reduzida e módulos reutilizáveis são possíveis. Em suma, a independência funcional é a chave para um bom projeto, e projeto é a chave para a qualidade de um software.

PONTO-CHAVE

Coesão é uma indicação qualitativa do grau com o qual um módulo se concentra em fazer apenas uma coisa.

A independência é avaliada usando-se dois critérios qualitativos: coesão e acoplamento. A *coesão* indica a robustez funcional relativa de um módulo. O *acoplamento* indica a interdependência relativa entre os módulos.

A coesão é uma extensão natural do conceito do encapsulamento de informações descrito na Seção 8.3.6. Um módulo coeso realiza uma única tarefa, exigindo pouca interação com outros componentes em outras partes de um programa. De forma simples, um módulo coeso deve (idealmente) fazer apenas uma coisa. Embora você sempre deva tentar ao máximo obter uma alta coesão (funcionalidade única), muitas vezes é necessário e recomendável fazer com que um componente de software realize várias funções. Entretanto, componentes "esquizofrênicos" (módulos que realizam muitas funções não relacionadas) devem ser evitados caso se queira um bom projeto.

PONTO-CHAVE

Acoplamento é uma indicação qualitativa do grau com o qual um módulo está conectado a outros módulos e com o mundo externo.

O acoplamento é uma indicação da interconexão entre os módulos em uma estrutura de software e depende da complexidade da interface entre os módulos, do ponto onde é feito o acesso a um módulo e dos dados que passam pela interface. Em projeto de software, você deve se esforçar para obter o menor grau de acoplamento possível. A conectividade simples entre módulos resulta em software mais fácil de ser compreendido e menos sujeito a "reação em cadeia" [Ste74], provocada quando ocorrem erros, em um ponto, que se propagam por todo o sistema.

AVISO

Existe uma tendência de ir imediatamente até o último detalhe, pulando as etapas de refinamento. Isso induz a erros e omissões e torna o projeto muito mais difícil de ser revisado. Realize o refinamento gradual.

8.3.8 Refinamento

O refinamento gradual é uma estratégia de projeto *top-down* proposta originalmente por Niklaus Wirth [Wir71]. Um programa é desenvolvido refinando-se sucessivamente níveis de detalhes procedurais. É desenvolvida uma hierarquia através da decomposição de uma declaração macroscópica da função (uma abstração procedural) de forma gradual até que as declarações da linguagem de programação sejam atingidas.

Refinamento é, na verdade, um processo de *elaboração*. Começamos com um enunciado da função (ou descrição de informações) definida em um alto nível de abstração. O enunciado descreve a função ou informações conceitualmente, mas não fornece nenhuma informação sobre o funcionamento interno da função ou da estrutura interna das informações. Em seguida, elaboramos a declaração original, fornecendo cada vez mais detalhes à medida que ocorre cada refinamento (elaboração) sucessivo.

Abstração e refinamento são conceitos complementares. A abstração nos permite especificar procedimento e dados internamente, mas suprimir a necessidade de que "estranhos" tenham conhecimento de detalhes de baixo nível. O refinamento nos ajuda a revelar detalhes menores à medida que o projeto avança. Ambos os conceitos permitem que criemos um modelo de projeto completo à medida que o projeto evolui.

"É difícil ler de cabo a rabo um livro sobre princípios de mágica sem, de tempos em tempos, dar uma olhada na capa para ter certeza de que não é um livro sobre projeto de software."

Bruce Tognazzini

8.3.9 Aspectos

À medida que ocorre análise de requisitos, vai sendo revelado um conjunto de "interesses (ou afinidades)". Entre tais interesses "temos os requisitos, os casos de uso, as características, as estruturas de dados, questões de qualidade de serviço, variações, limites de propriedade intelectual, colaborações, padrões e contratos" [AOS07]. Idealmente, um modelo de requisitos pode ser organizado de forma que lhe permita isolar grupos de interesses (requisitos) para que possam ser considerados independentemente. Na prática, entretanto, alguns desses interesses abrangem o sistema inteiro e não pode ser facilmente dividido em compartimentos.

Quando um projeto se inicia, os requisitos são refinados em uma representação de projeto modular. Consideremos dois requisitos, *A* e *B*. O requisito *A intersecciona* o requisito *B* "se tiver sido escolhida uma decomposição [refinamento] de software em que *B* não pode ser satisfeito sem levar em conta *A*" [Ros04].

PONTO-CHAVE

Preocupação em comum é alguma característica do sistema que se aplica a vários requisitos diferentes.

Consideremos, por exemplo, dois requisitos para a WebApp **CasaSeguraGarantida.com**. O requisito *A* é descrito por meio do caso de uso **AVC-EVC** discutido no Capítulo 6. O refinamento de um projeto poderia se concentrar naqueles módulos que permitiriam a um usuário registrado acessar imagens de vídeo de câmeras distribuídas em um ambiente. O requisito *B* é um requisito de segurança genérico que afirma que *um usuário registrado tem de ser validado antes de usar* **CasaSeguraGarantida.com**. Esse requisito se aplica a todas as funções disponíveis para os usuários registrados de *CasaSegura*. À medida que ocorre o refinamento de projeto, *A** é uma representação de projeto para o requisito *A*, e *B** é uma representação de projeto para o requisito *B*. Consequentemente, *A** e *B** são representações de interesses, e *B* tem intersecção com A**.

Aspecto é uma representação de um interesse em comum. Consequentemente, a representação de projeto, *B**, do requisito *um usuário registrado tem de ser validado antes de poder usar* **CasaSeguraGarantida.com**, é um aspecto da WebApp *CasaSegura*. É importante identificar aspectos de modo que o projeto possa acomodá-los apropriadamente à medida que ocorre o refinamento e a modularização. Em um contexto ideal, um aspecto é implementado como um módulo (componente) separado, em vez de fragmentos de software que estão "espalhados" ou "emaranhados" através de vários componentes [Ban06]. Para tanto, a arquitetura de projeto deve oferecer suporte a um mecanismo para definição de aspectos — um módulo que possibilite que um interesse seja implementado e atenda aos demais interesses que ele interseccione.

8.3.10 Refatoração

WebRef
Excelentes recursos sobre refabricação podem ser encontrados em **www.refactoring.com**.

Uma importante atividade sugerida por diversos métodos ágeis (Capítulo 3), a *refatoração* é uma técnica de reorganização que simplifica o projeto (ou código) de um componente sem mudar sua função ou comportamento. Fowler [Fow00] define refatoração da seguinte maneira: "Refatoração é o processo de mudar um sistema de software de tal forma que não altere o comportamento externo do código [projeto], embora melhore sua estrutura interna".

WebRef
Uma série de padrões de refatoração podem ser encontrados em **http://c2.com/cgi/wiki?RefactoringPatterns**.

Quando um software é refabricado, o projeto existente é examinado em termos de redundância, elementos de projeto não utilizados, algoritmos ineficientes ou desnecessários, estruturas de dados mal construídas ou inapropriadas, ou qualquer outra falha de projeto que possa ser corrigida para produzir um projeto melhor. Por exemplo, uma primeira iteração de projeto poderia gerar um componente que apresentasse baixa coesão (realizar três funções que possuem apenas relacionamento limitado entre si). Após cuidadosa consideração, talvez decidamos que o componente devesse ser refabricado em três componentes distintos, cada um apresentando alta coesão.

O resultado será um software mais fácil de se integrar, testar e manter.

CASASEGURA

Conceitos de projeto

Cena: Sala do Vinod, quando começa a modelagem de projetos.

Atores: Jamie, Vinod e Ed — todos membros da equipe de engenharia de software do *CasaSegura*. Também participa Shakira, novo membro da equipe.

Conversa:

[Todos os quatro membros da equipe acabaram de voltar de um seminário intitulado "Aplicação de Conceitos Básicos de Projeto", oferecido por um professor de computação.]

Vinod: Vocês tiveram algum proveito do seminário?

Ed: Já conhecia grande parte do que foi falado, mas não é uma má ideia ouvir novamente, suponho.

Jamie: Quando era aluno de Ciências da Computação, nunca realmente entendi por que o encapsulamento de informações era tão importante como diziam.

Vinod: Porque... Essencialmente... É uma técnica para reduzir a propagação de erros em um programa. Na verdade, a independência funcional também realiza a mesma coisa.

Shakira: Eu não fiz Ciências da Computação, portanto, um monte de coisas que o professor mencionou é novidade para mim. Sou capaz de gerar bom código e rapidamente. Não vejo por que isso é tão importante.

Jamie: Vi seu trabalho, Shak, e sabe de uma coisa, você já faz grande parte do que foi dito naturalmente... É por isso que seus projetos e códigos funcionam.

Shakira (sorrindo): Bem, sempre realmente tento subdividir o código, mantê-lo concentrado em algo, manter as interfaces simples e restritas, reutilizar código sempre que posso... Esse tipo de coisa.

Ed: Modularidade, independência funcional, encapsulamento, padrões... Sabe.

Jamie: Ainda me lembro do primeiro curso de programação que fiz... Eles nos ensinaram a refinar o código iterativamente.

Vinod: O mesmo pode ser aplicado ao projeto, sabe.

Vinod: Os únicos conceitos que ainda não havia ouvido falar foram "aspectos" e "refabricação".

Shakira: Isso é usado em *Extreme Programming*, acho que foi isso que ele disse.

Ed: Exato. Não é muito diferente do refinamento, apenas que você o faz depois que o projeto ou código esteja completo. Acontece uma espécie de otimização no software, se você quer saber.

Jamie: Retornemos ao projeto *CasaSegura*. Imagino que devamos colocar esses conceitos em nossa lista de controle de revisão à medida que desenvolvermos o modelo de projeto para o *CasaSegura*.

Vinod: concordo. Mas tão importante quanto, vamos todos nos comprometer a pensar nelas enquanto desenvolvemos o projeto.

8.3.11 Conceitos de projeto orientado a objetos

O paradigma orientado a objetos (*object oriented*, OO) é largamente utilizado na engenharia de software moderna. O Apêndice 2 é dirigido àqueles que talvez não estejam familiarizados com conceitos de projeto OO como classes e objetos, herança, mensagens e polimorfismo, entre outros.

8.3.12 Classes de projeto

O modelo de requisitos define um conjunto de classes de análise (Capítulo 6). Cada um descreve algum elemento do domínio do problema, concentrando-se nos aspectos do problema

visíveis ao usuário. O nível de abstração de uma classe de análise é relativamente alto. À medida que o modelo de projeto evoluir, definiremos um conjunto de *classes de projeto* que refinem as classes de análise, fornecendo detalhes de projeto que permitirão que as classes sejam implementadas, e implementem uma infraestrutura de software que suporte a solução do negócio. Podem ser desenvolvidos cinco tipos diferentes de classes de projeto, cada um deles representando uma camada diferente da arquitetura de projeto [Amb01]:

Quais tipos de classes o projetista cria?

- *Classes de interfaces do usuário* definem todas as abstrações necessárias para a interação humano-computador (*human-computer interaction*, HCI). Em muitos casos, a HCI acontece no contexto de uma *metáfora* (por exemplo, um talão de cheques, um formulário de pedidos, uma máquina de fax) e as classes de projeto para uma interface poderiam ser representações visuais dos elementos da metáfora.
- *Classes de domínio de negócio* normalmente são refinamentos das classes de análise definidas anteriormente. As classes identificam os atributos e serviços (métodos) necessários para implementar algum elemento do domínio de negócio.
- *Classes de processos* implementam as abstrações de aplicação de baixo nível necessárias para a completa gestão das classes de domínio de negócio.
- *Classes persistentes* representam repositórios de dados (por exemplo, um banco de dados) que persistirá depois da execução do software.
- *Classes de sistema* implementam funções de gerenciamento e controle de software que permitam ao sistema operar e comunicar em seu ambiente computacional e com o mundo exterior.

À medida que a arquitetura se forma, o nível de abstração é reduzido enquanto cada classe de análise é transformada em uma representação de projeto. As classes de análise representam objetos de dados (e serviços associados aplicados a eles) usando o jargão do domínio de negócio. As classes de projeto apresentam significativamente mais detalhes técnicos como um guia para a implementação.

Arlow e Neustadt [Arl02] sugerem que cada classe de projeto seja revista para garantir que seja "bem formada". Eles definem quatro características de uma classe de projeto bem formada:

O que é uma classe de projeto "bem formada"?

Completa e suficiente. Uma classe de projeto deve ser o encapsulamento completo de todos os atributos e métodos que podem ser razoavelmente esperados (baseado em uma interpretação inteligente do nome da classe) para a classe. Por exemplo, a classe **Cena** definida para software de edição de vídeo é completa apenas se contiver todos os atributos e métodos que podem ser razoavelmente associados com a criação de uma cena de vídeo. Suficiência garante que a classe de projeto contenha apenas os métodos suficientes para atingir o objetivo da classe, nem mais nem menos.

Primitivismo. Os métodos associados a uma classe de projeto deveriam se concentrar na realização de um serviço para a classe. Assim que o serviço tivesse sido implementado com um método, a classe não deveria realizar de outra maneira a mesma coisa. Por exemplo, a classe **VideoClipe** para um software de edição de vídeo poderia ter atributos **ponto de início** e **ponto final** para indicar os pontos de início e fim do clipe (note que uma fita virgem carregada no sistema talvez seja mais longa do que o clipe que é usado). Os métodos, *estabelecerPontoInício()* e *estabelecerPontoFinal()*, fornecem os únicos meios para estabelecer os pontos de início e fim do clipe.

Alta coesão. Uma classe de projeto coesa tem um conjunto de responsabilidades pequeno e focado e de forma resoluta aplica atributos e métodos para implementar essas responsabilidades. Por exemplo, a classe **VideoClipe** poderia conter um conjunto de métodos para editar o videoclipe. Desde que cada método se concentra exclusivamente nos atributos associados ao videoclipe, a coesão é mantida.

Baixo acoplamento. No modelo de projeto, é necessário para as classes colaborarem entre si. Entretanto, a colaboração deveria ser mantida em um mínimo aceitável. Se um modelo de

projeto for altamente acoplado (todas as classes de projeto colaboram com todas as demais classes de projeto), o sistema é difícil de implementar, testar e manter ao longo do tempo. Em geral, as classes de projeto em um subsistema deveriam ter apenas um conhecimento limitado das demais classes. Essa restrição, chamada *Lei de Demeter* [Lie03], sugere que um método deveria enviar mensagens apenas para métodos em classes vizinhas.[8]

CasaSegura

Refinamento de uma classe de análise em uma classe de projeto

Cena: Sala do Ed, quando começa o modelamento de projetos.

Atores: Vinod e Ed — membros da equipe de engenharia de software do *CasaSegura*.

Conversa:

[Ed está trabalhando na classe **Planta** (veja discussão no quadro da Seção 6.5.3 e na Figura 6.10) e a refinou para o modelo de projeto.]

Ed: Então, você se lembra da classe **Planta**, certo? Ela é usada como parte das funções de vigilância e gestão da casa.

Vinod (confirmando com a cabeça): É isso mesmo, parece que nós a usamos como parte de nossas discussões CRC para gestão da casa.

Ed: Usamos. De qualquer maneira, estou refinando-a para o projeto. Gostaria de mostrar como realmente implementaremos a classe **Planta**. Minha ideia é implementá-la como um conjunto de listas ligadas [uma estrutura de dados específica] Portanto... Eu tinha de refinar a classe de análise **Planta** (Figura 6.10) e, na verdade, simplificá-la.

Vinod: A classe de análise mostrava coisas apenas no domínio do problema, bem, na verdade na tela do computador, o que era visível para o usuário final, certo?

Ed: Isso, mas para a classe de projeto **Planta**, tive que acrescentar algumas coisas específicas da implementação. Precisava mostrar que **Planta** é uma agregação de segmentos — daí a classe **Segmento** — e que a classe **Segmento** é composta por listas de segmentos de parede, janelas, portas e assim por diante. A classe **Câmera** colabora com **Planta** e, obviamente, podem existir muitas câmeras na planta.

Vinod: Uía, vejamos uma figura desta nova classe de projeto **Planta**. [Ed mostra a Vinod o desenho apresentado na Figura 8.3.]

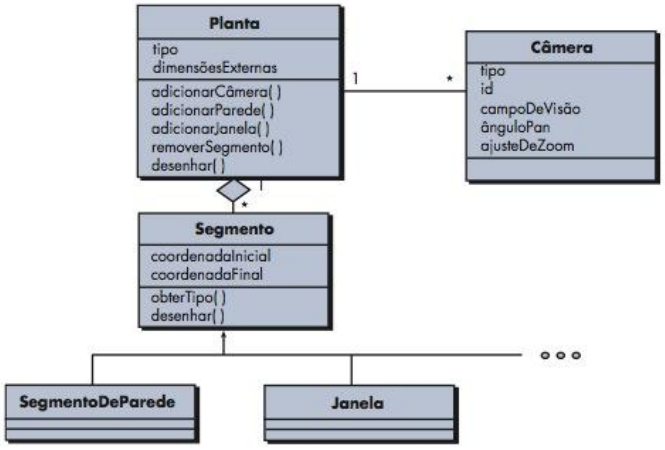
Vinod: Ok, vejo que você está tentando fazer. Isso lhe permite modificar facilmente a planta, pois novos itens podem ser acrescentados ou eliminados da lista — a agregação — sem quaisquer problemas.

Ed (meneando a cabeça): É isso aí, acho que vai funcionar.

Vinod: Eu também.

Figura 8.3

Classe de projeto para a Planta e agregação composta para a classe (veja no quadro de discussão)



8 Uma maneira menos formal de expressar a Lei de Demeter seria: "Cada unidade deve conversar apenas com seus amigos; não converse com estranhos".

8.4 O MODELO DE PROJETO

PONTO-CHAVE

O modelo de projeto possui quatro elementos principais: dados, arquitetura, componentes e interface.

"Questões como se o projeto é necessário ou acessível não vêm ao caso: o projeto é inevitável. A alternativa para um bom projeto é um projeto ruim, e não nenhum projeto em absoluto."

Douglas Martin

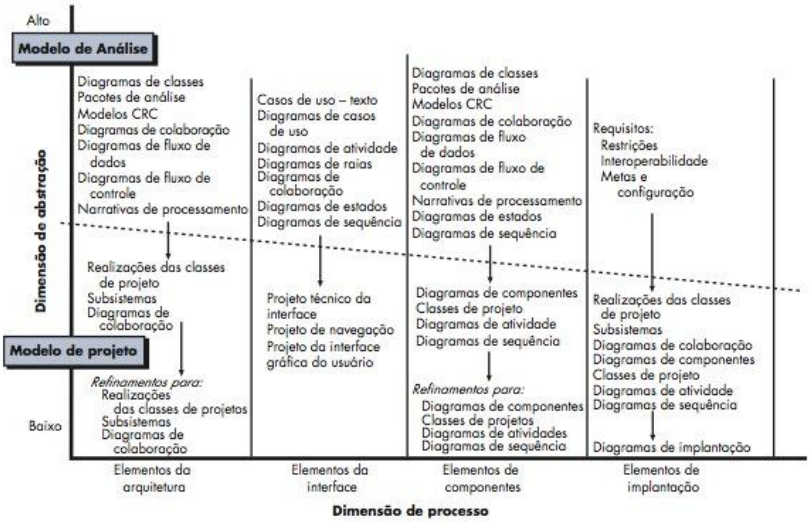
O modelo de projeto pode ser visto em duas dimensões diferentes conforme ilustrado na Figura 8.4. A *dimensão processo* indica uma evolução do modelo de projeto à medida que as tarefas de projeto são executadas como parte do processo do software. A *dimensão da abstração* representa o nível de detalhe à medida que cada elemento do modelo de análise é transformado em um equivalente de projeto e então refinado iterativamente. Referindo-se à Figura 8.4, a linha tracejada indica o limite entre os modelos de análise e de projeto. Em alguns casos, é possível ter uma clara distinção entre os modelos de análise e de projeto. Em outros, o modelo de análise vai lentamente se misturando ao de projeto e essa distinção clara é menos evidente.

Os elementos do modelo de projeto usam vários dos diagramas[9] UML utilizados no modelo de análise. A diferença é que esses diagramas são refinados e elaborados como parte do projeto; são fornecidos detalhes mais específicos à implementação e enfatizados a estrutura e o estilo da arquitetura, os componentes que residem nessa arquitetura, bem como as interfaces entre os componentes e com o mundo exterior.

Deve-se notar, entretanto, que os elementos de modelo indicados ao longo do eixo horizontal nem sempre são desenvolvidos de maneira sequencial. Na maioria dos casos, um projeto de arquitetura preliminar prepara o terreno e é seguido pelos projetos de interfaces e projeto de componentes, que normalmente ocorrem em paralelo. O modelo de implantação em geral é retardado até que o projeto tenha sido completamente desenvolvido.

Podemos aplicar padrões de projeto (Capítulo 12) em qualquer ponto durante o projeto. Estes possibilitam a utilização de conhecimentos adquiridos em projetos anteriores a problemas de domínios específicos encontrados e solucionados por outros.

FIGURA 8.4
Dimensões do modelo de projeto



9 O Apêndice 1 apresenta um tutorial sobre conceitos básicos e notação da UML.

8.4.1 Elementos de projeto de dados

PONTO-CHAVE

No nível da arquitetura (aplicação), o projeto de dados se concentra em arquivos ou bancos de dados; no nível dos componentes, o projeto de dados considera as estruturas de dados necessárias para implementar os objetos de dados locais.

Assim como ocorre com outras atividades da engenharia de software, o projeto de dados (também conhecido como *arquitetura de dados*) cria um modelo de dados e/ou informações que é representado em um nível de abstração elevado (a visão do cliente/usuário dos dados). Esse modelo é então refinado em representações cada vez mais específicas da implementação que podem ser processadas pelo sistema baseado em computador. Em muitas aplicações de software, a arquitetura dos dados terá uma profunda influência na arquitetura do software que deve processá-los.

A estrutura de dados sempre foi uma importante parte do projeto de software. No nível dos componentes de programa, o projeto das estruturas de dados e os algoritmos associados necessários para manipulá-los é essencial para a criação de aplicações de alta qualidade. No nível da aplicação, a transformação de um modelo de dados (obtido como parte da engenharia de necessidades) em um banco de dados é fundamental para atingir os objetivos de negócio de um sistema. No nível de negócio, o conjunto de informações armazenadas em bancos de dados diferentes e reorganizadas em um "depósito de dados" possibilita o *data mining* ou descoberta de conhecimento que pode ter um impacto no sucesso do negócio em si. Em qualquer caso, o projeto de dados desempenha um importante papel e é discutido com mais detalhes no Capítulo 9.

8.4.2 Elementos de projeto de arquitetura

"Você pode usar uma borracha enquanto ainda estiver na prancheta ou uma marreta na obra depois."

Frank Lloyd Wright

O *projeto de arquitetura* para software é o equivalente à planta baixa para uma casa. A planta baixa representa a distribuição dos cômodos; seus tamanhos, formas e relacionamentos entre si e as portas e janelas que possibilitam o deslocamento para dentro e fora dos cômodos. A planta baixa nos dá uma visão geral da casa. Os elementos de projeto de arquitetura nos dão uma visão geral do software.

O modelo de arquitetura [Sha96] é obtido de três fontes: (1) informações sobre o domínio de aplicação do software a ser construído; (2) elementos específicos de modelo de requisitos como os diagramas de fluxo de dados ou as classes de análise, seus relacionamentos e colaborações para o problema em questão; e (3) a disponibilidade de estilos de arquitetura (Capítulo 9) e padrões (Capítulo 12).

"O público está mais acostumado com projetos ruins do que bons projetos. Ele está, de fato, condicionado a preferir projetos ruins, pois é com isso que ele convive. O novo representa uma ameaça, o antigo, um sentimento de tranquilidade."

Paul Rand

O projeto dos elementos de arquitetura é normalmente representado como um conjunto de subsistemas interligados, em geral derivados dos pacotes de análise contidos no modelo de requisitos. Cada subsistema pode ter sua própria arquitetura (por exemplo, uma interface gráfica do usuário poderia ser estruturada de acordo com um estilo de arquitetura preexistente para interfaces do usuário). Técnicas para obtenção de elementos específicos do modelo de arquitetura são apresentadas no Capítulo 9.

8.4.3 Elementos de projeto de interfaces

O projeto de interfaces para software é análogo a um conjunto de desenhos detalhados (e especificações) para portas, janelas e ligações externas de uma casa. Esses desenhos representam o tamanho e a forma das portas e janelas, a maneira por meio da qual funcionam, a maneira pela qual as ligações de serviços públicos (por exemplo, água, energia elétrica, gás, telefone) entram na casa e são distribuídas entre os cômodos representados na planta. Eles nos informam onde se encontra a campainha, se deve ser usado ou não um porteiro eletrônico para anunciar a presença de um visitante e como um sistema de segurança deve ser instalado. Em essência, os desenhos detalhados (e especificações) para portas, janelas e ligações externas nos notificam como as coisas e as informações fluem para dentro e para fora da casa e no interior dos cômodos que fazem parte da planta. Os elementos de projeto de interfaces para software representam fluxos de informação que entram e saem do sistema e como são transmitidos entre os componentes definidos como parte da arquitetura.

PONTO-CHAVE

Há três partes para o elemento de projeto de interfaces: a interface do usuário, interfaces com os sistemas externos à aplicação e interfaces com componentes internos à aplicação.

Há três importantes elementos de projeto de interfaces: (1) a interface do usuário (*user interface*, UI); (2) interfaces externas para outros sistemas, dispositivos, redes ou outros produtores ou consumidores de informação; e (3) interfaces internas entre vários componentes de projeto. Esses elementos de projeto de interfaces possibilitam que o software se comunique externamente e que a comunicação interna e a colaboração entre os componentes preencham a arquitetura de software.

O projeto da UI (cada vez mais chamado *projeto de usabilidade*) é uma importante ação da engenharia de software e é considerado em detalhes no Capítulo 11. O projeto de usabilidade incorpora elementos estéticos (por exemplo, layout, cor, imagens, mecanismos de interação), elementos ergonômicos (por exemplo, o layout e a colocação de informações, metáforas, navegação da UI) e elementos técnicos (por exemplo, padrões UI, componentes reutilizáveis). Em geral, a UI é um subsistema exclusivo da arquitetura da aplicação geral.

O projeto de interfaces externas requer informações definitivas sobre a entidade para as quais as informações são enviadas ou recebidas. Em todos os casos, essas informações devem ser coletadas durante a engenharia de requisitos (Capítulo 5) e verificadas assim que o projeto de interface for iniciado.[10] O projeto de interfaces externas deve incorporar a verificação de erros e (quando necessário) características de segurança apropriadas.

O projeto de interfaces internas está intimamente ligado ao projeto dos componentes (Capítulo 10). As realizações de projeto das classes de análise representam todas as operações e os esquemas de troca de mensagens necessários para habilitar a comunicação e a colaboração entre as operações em várias classes. Cada mensagem deve ser desenvolvida para acomodar a transferência de informações requeridas e os requisitos funcionais específicos da operação solicitada. Se for escolhida a abordagem clássica de entrada-processo-saída para o projeto, a interface de cada componente de software é projetada tomando como base as representações do fluxo de dados e a funcionalidade descrita em uma narrativa de processamento.

Em alguns casos, uma interface é modelada de forma bastante parecida com a de uma classe. Na UML, a interface é definida da seguinte maneira [OMG03a]: "Interface é um especificador para as operações [públicas] visíveis externamente de uma classe, componente ou outro classificador (incluindo subsistemas), sem a especificação da estrutura interna". De maneira mais simples, interface é um conjunto de operações que descreve alguma parte do comportamento de uma classe e dá acesso a essas operações.

Por exemplo, a função de segurança do *CasaSegura* faz uso de um painel de controle que possibilita a um proprietário de imóvel controlar certos aspectos da função de segurança. Em uma versão mais avançada do sistema, as funções do painel de controle poderiam ser implementadas por meio de um PDA sem fio ou telefone celular.

A classe **PainelControle** (Figura 8.5) fornece o comportamento associado com um teclado e, consequentemente, deve implementar as operações *lerTeclaPressionada ()* e *decodificarTecla()*. Se essas operações tiverem de ser fornecidas para outras classes (no caso, **PDASemFio** e **TelefoneMóvel**), é útil definir uma interface conforme mostra a figura. A interface, chamada **Teclado**, é apresentada como um estereótipo <<interface>> ou como um pequeno círculo identificado ligado à classe por meio de uma linha. A interface é definida sem nenhum atributo e conjunto de operações necessários para atingir o comportamento de um teclado.

A linha tracejada com um triângulo com fundo branco em sua ponta (Figura 8.5) indica que a classe **PainelControle** fornece operações de **Teclado** como parte de seu comportamento. Na UML, isso é caracterizado como uma *realização*. Ou seja, parte do comportamento de **PainelControle** será implementada realizando as operações de **Teclado**. Essas operações serão fornecidas a outras classes que acessam a interface.

"De tempos em tempos dê uma volta, relaxe um pouco, de modo que ao voltar para o trabalho seu julgamento seja mais seguro. Afaste-se um pouco; o trabalho parecerá menor e grande parte dele poderá ser vista com um pequeno exame, e a falta de harmonia e proporção serão visualizadas mais facilmente."

Leonardo DaVinci

WebRef

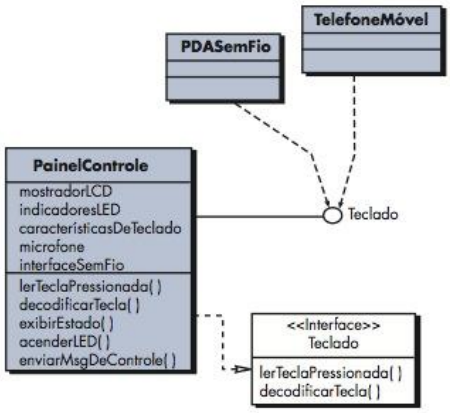
Informações extremamente valiosas no projeto da UI podem ser encontradas em **www.useit.com.**

"Um erro comum que as pessoas cometem ao tentarem projetar algo completamente infalível é subestimar a criatividade de completos idiotas."

Douglas Adams

10 As características das interfaces podem mudar ao longo do tempo. Consequentemente, um projetista deve assegurar que a especificação para uma interface seja precisa e completa.

FIGURA 8.5
Representação da interface para PainelControle



8.4.4 Elementos de projeto de componentes

O projeto de componentes para o software equivale a um conjunto de desenhos detalhados (e especificações) para cada cômodo de uma casa. Esses desenhos representam a fiação e o encanamento dentro de cada cômodo, a localização das tomadas e interruptores, torneiras, pias, chuveiros, banheiras, ralos, armários e banheiros. Eles também descrevem o piso a ser usado, as molduras a ser aplicadas e qualquer outro detalhe associado a um cômodo. O projeto de componentes para software descreve completamente os detalhes internos de cada componente de software. Para tanto, o projeto no nível de componente define estruturas de dados para todos os objetos de dados locais e detalhes algorítmicos para todo o processamento que ocorre em um componente e uma interface que dá acesso a todas as operações de componentes (comportamentos).

"Os detalhes não são detalhes. Eles fazem o projeto."
Charles Eames

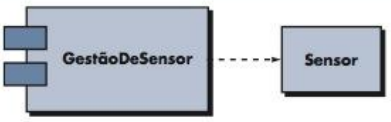
No contexto da engenharia de software orientada a objetos, um componente é representado em forma esquemática em UML conforme mostra a Figura 8.6. Nessa figura, é representado um componente chamado **GestãoDeSensor** (parte da função de segurança do *CasaSegura*). Uma seta pontilhada conecta o componente a uma classe chamada **Sensor** que é atribuída a ele. O componente **GestãoDeSensor** realiza todas as funções associadas aos sensores do *CasaSegura*, incluindo seu monitoramento e configuração. Uma discussão mais abrangente sobre diagramas de componentes é apresentada no Capítulo 10.

Os detalhes de projeto de um componente não podem ser modelados em muitos níveis de abstração diferentes. Um diagrama de atividades UML pode ser utilizado para representar processamento lógico. O fluxo procedural detalhado para um componente pode ser representado usando pseudocódigo (uma representação similar a uma linguagem de programação descrita no Capítulo 10) ou alguma outra forma esquemática (por exemplo, fluxograma ou diagrama de blocos). A estrutura algorítmica segue as regras estabelecidas para a programação estruturada (um conjunto de construções procedurais restritas). As estruturas de dados escolhidas tomando como base a natureza dos objetos de dados a ser processados normalmente são modeladas usando pseudocódigo ou a linguagem de programação para implementação.

8.4.5 Elementos de projeto de implantação

Os elementos de projeto de implantação indicam como os subsistemas e a funcionalidade de software serão alocados no ambiente computacional físico que irá suportar o software. Por

FIGURA 8.6
Um diagrama de componentes UML



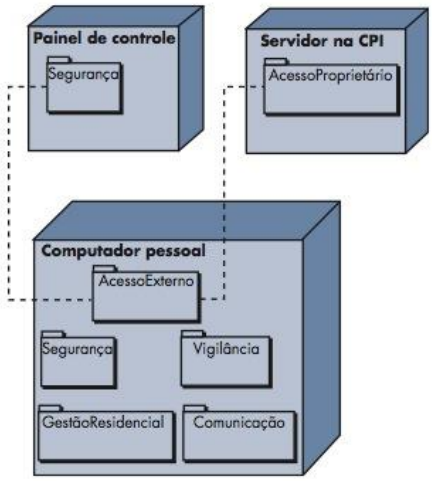
PONTO-CHAVE
Os diagramas de disponibilização começam na forma de descritores, em que o ambiente de disponibilização é descrito em termos gerais. Posteriormente, é usada a forma de instância, e os elementos da configuração são descritos explicitamente.

exemplo, os elementos do produto *CasaSegura* são configurados para operar dentro de três ambientes computacionais principais — um PC localizado na casa, o painel de controle *CasaSegura* e um servidor localizado na CPI Corp. (fornecendo acesso ao sistema via Internet).

Durante o projeto, um diagrama de implantação UML é desenvolvido e refinado conforme mostra a Figura 8.7. Na figura, são apresentados três ambientes computacionais (na verdade, existiriam outros com a inclusão de sensores, câmeras e outros dispositivos). Os subsistemas (funcionalidade) abrigados em cada elemento computacional são indicados. Por exemplo, o PC abriga subsistemas que implementam funções de segurança, vigilância, gestão residencial e de comunicação. Além disso, um subsistema de acesso externo foi projetado para gerenciar todas as tentativas para acessar o sistema *CasaSegura* de uma fonte externa. Cada subsistema seria elaborado para indicar os componentes que ele implementa.

O diagrama apresentado na Figura 8.7 se encontra na *forma de descritores*. Isso significa que o diagrama de disponibilização mostra o ambiente computacional, mas não indica explicitamente detalhes de configuração. Por exemplo, o "computador pessoal" não tem uma identificação adicional. Ele poderia ser um Mac ou um PC com Windows, uma estação de trabalho da Sun ou um computador com Linux. Esses detalhes são fornecidos quando o diagrama de implantação é revisitado na *forma de instância* durante os últimos estágios do projeto ou quando começa a construção. Cada instância da disponibilização (uma configuração de hardware com nome e específica) é identificada.

FIGURA 8.7
Um diagrama de implantação UML



8.5 Resumo

O projeto de software começa quando a primeira iteração da engenharia de requisitos chega a uma conclusão. O intuito do projeto de software é aplicar um conjunto de princípios, conceitos e práticas que levam ao desenvolvimento de um sistema ou produto de alta qualidade. O objetivo do projeto é criar um modelo de software que irá implementar corretamente todos os requisitos do cliente e trazer satisfação àqueles que o usarem. Os projetistas de software devem examinar completamente muitas alternativas de projeto e convergir para uma solução que melhor atenda às necessidades dos interessados no projeto.

O processo de projeto passa de uma visão macro do software para uma visão mais estreita que define os detalhes necessários para implementar um sistema. O processo começa concentrando-se na arquitetura. São definidos subsistemas; são estabelecidos mecanismos de comunicação entre os subsistemas; são identificados componentes; e é desenvolvida uma descrição detalhada de cada componente. Além disso, são projetadas interfaces externas, internas e para o usuário.

Os conceitos de projeto evoluíram ao longo dos primeiros 60 anos do trabalho da engenharia de software. Eles descrevem atributos de software que devem estar presentes independentemente do processo de engenharia de software escolhido, dos métodos de projeto aplicados ou das linguagens de programação usadas. Em essência, os conceitos de projeto enfatizam a necessidade da abstração como um mecanismo para a criação de componentes de software reutilizáveis; a importância da arquitetura como forma para melhor entender a estrutura geral de um sistema; os benefícios da engenharia baseada em padrões como técnica para desenvolvimento de software com capacidades já comprovadas; o valor da separação de preocupações e da modularidade eficaz como forma de tornar o software mais compreensível, mais fácil de ser testado e mantido; as consequências do encapsulamento de informações como um mecanismo para reduzir a propagação de efeitos colaterais quando da real ocorrência de erros; o impacto da independência funcional como critério para a construção de módulos eficazes; o uso do refinamento como mecanismo de projeto; a consideração de aspectos que interseccionem as necessidades do sistema; a aplicação da refabricação na otimização do projeto que é obtido; e a importância das classes orientadas a objetos e as características a elas relacionadas.

O modelo de projeto engloba quatro elementos distintos; à medida que cada um é desenvolvido, evolui uma visão mais completa do projeto. O elemento de arquitetura usa informações extraídas do domínio de aplicação, do modelo de requisitos e de catálogos disponíveis para padrões e estilos para obter uma representação estrutural completa do software, seus subsistemas e componentes. Elementos de projeto de interfaces modelam interfaces internas e externas, bem como a interface do usuário. Elementos de componentes definem cada um dos módulos (componentes) que preenchem a arquitetura. Por fim, os elementos de disponibilização alocam a arquitetura, seus componentes e as interfaces para a configuração física que abrigará o software.

Problemas e Pontos a Ponderar

8.1. Você projeta software ao "escrever" um programa? O que torna o projeto de software diferente da codificação?

8.2. Se um projeto de software não é um programa (e não é mesmo), então o que ele é?

8.3. Como avaliar a qualidade de um projeto de software?

8.4. Examine o conjunto de tarefas apresentado para o projeto. Em que momento a qualidade é avaliada em um conjunto de tarefas? Como se consegue isso? Como os atributos de qualidade discutidos na Seção 8.2.1 são atingidos?

8.5. Dê exemplos de três abstrações de dados e as abstrações procedurais que podem ser usadas para manipulá-las.

8.6. Descreva arquitetura de software com suas próprias palavras.

8.7. Sugira um padrão de projeto que você encontra em uma categoria das coisas cotidianas (por exemplo, eletrônica de consumo, automóveis, aparelhos domésticos). Descreva brevemente o padrão.

8.8. Descreva a separação de preocupações com suas próprias palavras. Existe um caso em que a estratégia dividir-para-conquistar não poderia ser apropriada? Como um caso desses poderia afetar o argumento da modularidade?

8.9. Quando um projeto modular deve ser implementado como um software monolítico? Como isso pode ser obtido? O desempenho é a única justificativa para a implementação de software monolítico?

8.10. Discuta a relação entre o conceito de encapsulamento de informações como um atributo de modularidade eficaz e o conceito da independência de módulos.

8.11. Como os conceitos de acoplamento e portabilidade de software estão relacionados? Dê exemplos para apoiar sua discussão.

8.12. Aplique uma "abordagem de refinamento gradual" para desenvolver três níveis diferentes de abstrações procedurais para um ou mais dos seguintes programas: (a) desenvolver um preenchedor de cheques que, dada uma quantia numérica, imprimirá a quantia por extenso como exigido no preenchimento de cheques; (b) encontrar iterativamente as raízes de uma equação transcendente; (c) desenvolver um algoritmo de cronograma de tarefas simples para um sistema operacional.

8.13. Considere o software necessário para implementar um recurso completo de navegação (usando GPS) em um dispositivo de comunicação móvel portátil. Descreva duas ou três preocupações em comum que estariam presentes. Discuta como você representaria uma dessas preocupações na forma de um aspecto.

8.14. "Refabricação" significa que modificamos todo o projeto iterativamente? Em caso negativo, o que significa?

8.15. Descreva brevemente cada um dos quatro elementos do modelo de projeto.

Leituras e Fontes de Informação Complementares

Donald Norman escreveu dois livros (*The Design of Everyday Things,* Doubleday, 1990 e *The Psychology of Everyday Things,* Harpercollins, 1988) que se tornaram clássicos na literatura de projeto e uma leitura "obrigatória" para qualquer um que projete qualquer coisa que os seres humanos usam. Adams (*Conceptual Blockbusting,* 3. ed., Addison-Wesley, 1986) é o autor de um texto essencial para os projetistas que querem alargar sua maneira de pensar. Por fim, um clássico de Polya (*How to Solve It,* 2. ed., Princeton University Press, 1988) fornece um processo genérico para resolução de problemas que podem ajudar os projetistas de software ao depararem com problemas complexos.

Seguindo a mesma tradição, Winograd et al. (*Bringing Design to Software,* Addison-Wesley, 1996) discutem projetos de software que funcionam, aqueles que não funcionam e o porquê. Um livro fascinante editado por Wixon e Ramsey (*Field Methods Casebook for Software Design,* Wiley, 1996) sugere métodos de pesquisa de campo (muito parecido com aqueles usados por antropólogos) para entender como os usuários finais realizam o trabalho deles e depois projetam software que atenda suas necessidades. Beyer e Holtzblatt (*Contextual Design: A Customer-Centered Approach to Systems Designs,* Academic Press, 1997) oferece uma outra visão do projeto de software que integra o cliente/usuário em todos os aspectos do processo de projeto de software. Bain (*Emergent Design,* Addison-Wesley, 2008) acopla padrões, refabricação e desenvolvimento dirigido por testes em uma abordagem de projeto eficaz.

Um tratado completo de projeto no contexto da engenharia de software é apresentado por Fox (*Introduction to Software Engineering Design,* Addison-Wesley, 2006) e Zhu (*Software Design Methodology,* Butterworth-Heinemann, 2005). McConnell (*Code Complete,* 2. ed.,

Microsoft Press, 2004) traz uma excelente discussão dos aspectos práticos para projetar software de alta qualidade. Robertson (*Simple Program Design*, 3. ed., Boyd and Fraser Publishing, 1999) apresenta uma discussão introdutória do projeto de software que é útil para aqueles que estão iniciando seus estudos do assunto. Budgen (*Software Design*, 2. ed., Addison-Wesley, 2004) introduz uma série de métodos de projeto populares, comparando e contrastando cada um deles. Fowler e seus colegas (*Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999) discutem técnicas para a otimização incremental de projetos de software. Rosenberg e Stevens (*Use Case Driven Object Modeling with UML*, Apress, 2007) abordam o desenvolvimento de projetos orientados a objetos usando casos de uso como base.

Uma excelente pesquisa histórica de projeto de software está contida em uma antologia editada por Freeman e Wasserman (*Software Design Techniques*, 4; ed., IEEE, 1983). Esse tutorial reapresenta diversos artigos clássicos que formaram a base para as tendências correntes em projeto de software. Medidas da qualidade de projeto, apresentadas tanto sob as perspectivas técnica como de gerenciamento, são consideradas por Card e Glass (*Measuring Software Design Quality*, Prentice-Hall, 1990).

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