5.9 Desenhe diagramas de máquina de estados do software de controle para:

- uma lavadora automática com diferentes programas para diferentes tipos de roupas;
- o software de um DVD player;
- o software de controle da câmera em seu celular. Ignore o flash se você tiver um em seu celular.

5.10 Você é um gerente de engenharia de software e um membro experiente de sua equipe propõe que a engenharia dirigida por modelos deveria ser utilizada para desenvolver um novo sistema. Quais fatores você deveria levar em conta quando decidir se deve ou não introduzir essa abordagem para o desenvolvimento de software?

REFERÊNCIAS

AMBLER, S. W. *The object primer*: agile model-driven development with UML 2.0. 3. ed. Cambridge: Cambridge University Press, 2004.

AMBLER, S. W.; JEFFRIES, R. *Agile modeling*: effective practices for extreme programming and the unified process. New York: John Wiley & Sons, 2002.

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. *The Unified Modeling Language user guide*. 2. ed. Boston: Addison-Wesley, 2005.

BRAMBILLA, M.; CABOT, J.; WIMMER, M. *Model-driven software engineering in practice*. San Rafael: Morgan Claypool, 2012.

DEN HAAN, J. "Why there is no future for Model Driven Development." *The Enterprise Architect*, 25 jan. 2011. Disponível em: <http://www.theenterprisearchitect.eu/archive/2011/01/25/why-there-is-no-future-for-model-driven-development/>. Acesso em: 22 mai. 2018.

ERICKSON, J.; SIAU, K. "Theoretical and Practical Complexity of Modeling Methods." *Comm. ACM*, v. 50, n. 8, 2007. p. 46-51. doi:10.1145/1278201.1278205.

HAREL, D. "Statecharts: a visual formalism for complex systems." *Sci. Comput. Programming*, v. 8, n. 3, 1987. p. 231-274. doi:10.1016/0167-6423(87)90035-9.

HULL, R.; KING, R. "Semantic database modeling: survey, applications and research issues." *ACM Computing Surveys*, v. 19, n. 3, 1987. p. 201-260. doi:10.1145/45072.45073.

HUTCHINSON, J.; ROUNCEFIELD, M.; WHITTLE, J. "Model-Driven Engineering Practices in Industry." In: *34ª Int. Conf. on Software Engineering*, 2012. p. 633-642. doi:10.1145/1985793.1985882.

JACOBSON, I.; CHRISTERSON, M.; JONSSON, P.; OVERGAARD, G. *Object-oriented software engineering*. Wokingham: Addison-Wesley, 1993.

KOEGEL, M. "EMF Tutorial: What Every Eclipse Developer Should Know about EMF." 2012. Disponível em: <http://eclipsesource.com/blogs/tutorials/emf-tutorial/>. Acesso em: 11 abr. 2018.

MELLOR, S. J.; BALCER, M. J. *Executable UML*. Boston: Addison-Wesley, 2002.

_______.; SCOTT, K.; WEISE, D. *MDA distilled*: principles of model-driven architecture. Boston: Addison-Wesley, 2004.

OMG - Object Management Group. "Model-driven architecture: success stories." 2012. Disponível em: <http://www.omg.org/mda/products_success.htm>. Acesso em: 11 abr. 2018.

RUMBAUGH, J.; JACOBSON, I.; BOOCH, G. *The Unified Modelling Language Reference Manual*. 2. ed. Boston: Addison-Wesley. 2004.

STAHL, T.; VOELTER, M. *Model-Driven Software Development*: Technology, Engineering, Management. New York: John Wiley & Sons. 2006.

ZHANG, Y.; PATEL, S. "Agile Model-Driven Development in Practice." *IEEE Software*, v. 28, n. 2, 2011. p. 84-91. doi:10.1109/MS.2010.85.

6 | Projeto de arquitetura

OBJETIVOS

O objetivo deste capítulo é introduzir os conceitos de arquitetura de software e projeto de arquitetura. Ao ler este capítulo, você:

- compreenderá por que o projeto de arquitetura de software é importante;
- compreenderá as decisões que precisam ser tomadas a respeito da arquitetura de software durante o projeto;
- será apresentado à ideia de padrões de arquitetura, que são maneiras experimentadas de organizar as arquiteturas de software que podem ser reusadas nos projetos de sistemas;
- compreenderá como podem ser utilizados padrões de arquitetura específicos para aplicações no processamento de transações e nos sistemas de processamento de linguagens.

CONTEÚDO

6.1 Decisões de projeto de arquitetura

6.2 Visões de arquitetura

6.3 Padrões de arquitetura

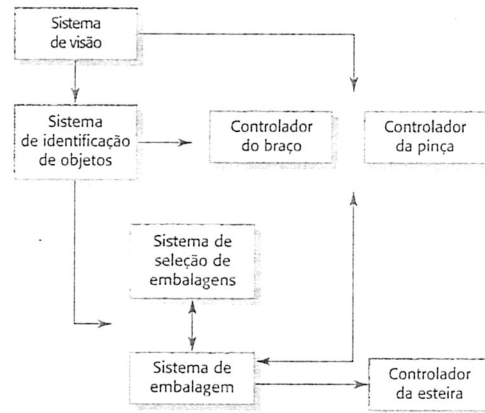
6.4 Arquiteturas de aplicações

O projeto de arquitetura visa compreender como um sistema de software deve ser organizado e projetar a estrutura geral desse sistema. No modelo do processo de desenvolvimento de software que descrevi no Capítulo 2, o projeto de arquitetura é o primeiro estágio no processo de projeto (*design*) do software. É o vínculo fundamental entre o projeto e a engenharia de requisitos, já que identifica os principais componentes estruturais em um sistema e as relações entre eles. A saída do processo de projeto de arquitetura é um modelo que descreve como o sistema está organizado como um conjunto de componentes que se comunicam.

Nos processos ágeis, é geralmente aceito que um estágio inicial do processo de desenvolvimento ágil esteja focado na criação do projeto da arquitetura geral do sistema. O desenvolvimento incremental das arquiteturas normalmente não é bem-sucedido. É relativamente fácil refatorar os componentes em resposta às mudanças. No entanto, é caro refatorar a arquitetura do sistema porque é possível ter de modificar a maior parte dos componentes do sistema para adaptá-los às mudanças da arquitetura.

Para ajudar a entender o que quero dizer com arquitetura de sistema, basta observar a Figura 6.1. Esse diagrama mostra um modelo abstrato da arquitetura de um sistema de embalagem robotizado, que consegue embalar diferentes tipos de objetos. Ele utiliza um componente de visão computacional para escolher os objetos em uma esteira, identificá-los e selecionar o tipo correto de embalagem. Depois, o sistema move os objetos da esteira de entrega para serem embalados e os coloca em outra esteira. O modelo de arquitetura mostra esses componentes e os vínculos entre eles.

FIGURA 6.1 A arquitetura de um sistema de controle de embalagem robotizado.



Na prática, há uma sobreposição significativa entre os processos de engenharia de requisitos e o projeto de arquitetura. Em condições ideais, uma especificação do sistema não deveria incluir qualquer informação de projeto. Contudo, esse ideal não é realista, exceto quando tratamos de sistemas muito pequenos. É preciso identificar os principais componentes da arquitetura, pois eles refletem as características de alto nível do sistema. Portanto, como parte integrante do processo de engenharia, é possível propor uma arquitetura do sistema abstrata, que associa grupos de funções ou características do sistema a componentes ou subsistemas de larga escala. Depois, essa decomposição é usada para discutir com *stakeholders* os requisitos e características mais detalhadas do sistema.

É possível projetar as arquiteturas de software em dois níveis de abstração, que eu chamo de arquiteturas em pequena e grande escala:

1. A *arquitetura em pequena escala* está relacionada com a arquitetura de um programa individual. Nesse nível, estamos preocupados com a maneira como cada programa é decomposto em seus componentes. Este capítulo tem a ver basicamente com as arquiteturas de programas.
2. A *arquitetura em grande escala* está relacionada com a arquitetura de sistemas corporativos complexos, que incluem outros sistemas, programas e componentes de programas. Esses sistemas corporativos podem ser distribuídos por diferentes computadores, os quais podem pertencer e ser gerenciados por diferentes empresas. (A arquitetura em grande escala será coberta nos Capítulos 17 e 18.)

A arquitetura de software é importante porque afeta o desempenho, a robustez, a capacidade de distribuição e a manutenibilidade de um sistema (BOSCH, 2000). Como explica Bosch, os componentes individuais implementam os requisitos funcionais do sistema, mas a influência dominante nas características não funcionais desse sistema é sua arquitetura. Chen, Ali Babar e Nuseibeh (2013) confirmaram isso em um estudo dos 'requisitos arquiteturalmente relevantes', no qual constataram que os requisitos não funcionais surtiram o efeito mais significativo na arquitetura do sistema.

Bass, Clements e Kazman (2012) sugerem que projetar e documentar explicitamente a arquitetura de software tem três vantagens:

1. *Comunicação com* stakeholders. A arquitetura é uma apresentação de alto nível do sistema que pode ser usada como foco de discussão por vários *stakeholders*.
2. *Análise de sistema.* Tornar explícita a arquitetura de sistema em um estágio inicial no desenvolvimento de sistemas exige alguma análise. As decisões de projeto de arquitetura têm um efeito profundo no cumprimento ou não dos requisitos críticos, como o desempenho, a confiabilidade e a manutenibilidade.
3. *Reúso em larga escala.* Um modelo de arquitetura é uma descrição compacta e gerenciável de como um sistema é organizado e de como os componentes operam entre si. A arquitetura do sistema costuma ser a mesma nos sistemas com requisitos parecidos e, por isso, consegue apoiar o reúso de software em larga escala. Conforme explicarei no Capítulo 15, as arquiteturas de linha de produto são uma abordagem na qual a mesma arquitetura é reusada por toda uma gama de sistemas relacionados.

As arquiteturas de sistema são frequentemente modeladas de modo informal, com diagramas de bloco simples sendo usados, como na Figura 6.1. Cada caixa no diagrama representa um componente. As caixas dentro de caixas indicam que o componente foi decomposto em subcomponentes. As setas significam que os dados e os sinais de controle são passados de um componente para outro na direção das setas. É possível ver muitos exemplos desse tipo de modelo de arquitetura no manual de arquitetura de software de Booch (2014).

Os diagramas de bloco apresentam uma imagem de alto nível da estrutura do sistema, que pessoas de diferentes disciplinas envolvidas no processo de desenvolvimento do sistema podem compreender imediatamente. Apesar de seu uso generalizado, Bass, Clements e Kazman (2012) não gostam dos diagramas de bloco informais para descrever uma arquitetura, pois dizem que são representações pobres da arquitetura, já que não mostram o tipo de relacionamento entre os componentes do sistema e nem suas propriedades externas visíveis.

As aparentes contradições entre a teoria de arquitetura e a prática industrial surgem porque existem duas maneiras de utilizar o modelo de arquitetura de um programa:

1. *Como forma de encorajar as discussões sobre o projeto do sistema.* Uma visão de alto nível da arquitetura de um sistema é útil para a comunicação com os *stakeholders* e para o planejamento do projeto, pois ela não está lotada de detalhes. Os *stakeholders* podem se referir a ela e entender uma visão abstrata do sistema e, depois, discutir o sistema como um todo sem se confundir com detalhes. O modelo de arquitetura identifica os componentes-chave que devem ser desenvolvidos, então os gerentes podem começar a designar pessoas para planejar o desenvolvimento desses sistemas.
2. *Como forma de documentar uma arquitetura que foi projetada.* O objetivo aqui é produzir um modelo completo do sistema que mostre seus diferentes componentes, suas interfaces e suas conexões. O argumento para esse tipo de

modelo é que uma descrição detalhada da arquitetura facilita a compreensão e o desenvolvimento do sistema.

Os diagramas de bloco são uma boa maneira de apoiar a comunicação entre as pessoas envolvidas no processo de projeto (*design*) do software. Eles são intuitivos, e tanto os especialistas no domínio quanto os engenheiros de software podem se referir a eles e participar das discussões sobre o sistema. Os gerentes consideram esses diagramas úteis no planejamento do projeto. Em muitos projetos, os diagramas de bloco são a única descrição da arquitetura.

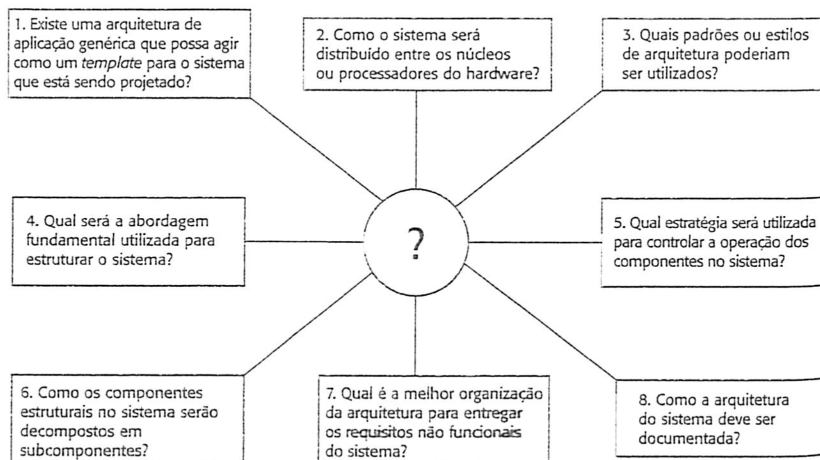
Em condições ideais, se a arquitetura de um sistema tiver de ser documentada em detalhes, é melhor usar uma notação mais rigorosa para a descrição da arquitetura. Várias linguagens de descrição de arquitetura foram desenvolvidas (BASS; CLEMENTS; KAZMAN, 2012) com essa finalidade. Uma descrição mais detalhada e completa significa que há menos espaço para equívocos nos relacionamentos entre os componentes da arquitetura. No entanto, o desenvolvimento de uma descrição detalhada é um processo caro e demorado. É praticamente impossível saber se é econômico ou não, então essa abordagem não é muito utilizada.

6.1 DECISÕES DE PROJETO DE ARQUITETURA

O projeto de arquitetura é um processo criativo no qual se projeta a organização de um sistema que vai satisfazer os seus requisitos funcionais e não funcionais. Não existe um processo de projeto de arquitetura estereotipado. Ele depende do tipo do sistema que está sendo desenvolvido, da experiência do arquiteto do sistema e dos requisitos específicos desse sistema. Consequentemente, é melhor considerar o projeto de arquitetura como uma série de decisões a serem tomadas, em vez de uma sequência de atividades.

Durante o processo de projeto de arquitetura, os arquitetos do sistema têm de tomar uma série de decisões estruturais que afetam profundamente o sistema e seu processo de desenvolvimento. Com base em seu conhecimento e experiência, eles devem considerar as questões fundamentais exibidas na Figura 6.2.

FIGURA 6.2 Decisões de projeto de arquitetura.



Embora cada sistema de software seja único, os sistemas no mesmo domínio de aplicação costumam ter arquiteturas similares, que refletem os conceitos fundamentais do domínio. Por exemplo, as linhas de produto de aplicação são aplicações criadas em torno de uma arquitetura central, com variantes que satisfazem requisitos específicos do cliente. Ao projetar uma arquitetura de sistemas, é necessário decidir o que o sistema e as classes de aplicação mais amplas têm em comum e quanto conhecimento é possível reaproveitar dessas arquiteturas de aplicações.

Nos sistemas embarcados e nas aplicações projetadas para computadores pessoais e dispositivos móveis, não é necessário projetar uma arquitetura distribuída para o sistema. No entanto, a maioria dos sistemas grandes consiste em sistemas distribuídos no quais o software do sistema é distribuído por muitos computadores diferentes. A escolha da arquitetura de distribuição é uma decisão-chave, que afeta o desempenho e a confiabilidade do sistema. Esse tópico é importante por si só, e será coberto no Capítulo 17.

A arquitetura de um sistema de software pode se basear em um determinado padrão ou estilo de arquitetura (esses termos acabaram ganhando o mesmo significado). Um padrão de arquitetura é uma descrição de uma organização de sistema (GARLAN; SHAW, 1993), como uma organização cliente-servidor ou uma arquitetura em camadas. Os padrões de arquitetura capturam a essência de uma arquitetura que tem sido utilizada em diferentes sistemas de software. Ao tomar decisões sobre a arquitetura de um sistema, é necessário estar a par dos padrões comuns, onde eles podem ser utilizados e quais são seus pontos fortes e fracos. Tratarei, na Seção 6.3, de vários padrões frequentemente utilizados.

O conceito de Garlan e Shaw, de um estilo de arquitetura, abrange as questões 4 a 6 na lista de questões fundamentais da Figura 6.2. Deve-se escolher a estrutura mais adequada — cliente-servidor ou em camadas, por exemplo —, que permitirá o atendimento dos requisitos do sistema. Para decompor as unidades estruturais do sistema, deve-se decidir por uma estratégia para decompor os componentes em subcomponentes. Finalmente, no processo de modelagem do controle, desenvolve-se um modelo geral de relacionamentos de controle entre as várias partes do sistema e decide-se como a execução dos componentes é controlada.

Em virtude do relacionamento entre as características não funcionais do sistema e sua arquitetura, a escolha do estilo de arquitetura e da estrutura deve depender dos requisitos não funcionais do sistema:

1. *Desempenho*. Se o desempenho for um requisito crítico, a arquitetura deve ser projetada para localizar as operações críticas dentro de um pequeno número de componentes, com esses componentes implantados no mesmo computador, em vez de distribuídos pela rede. Isso pode significar a necessidade de usar alguns componentes relativamente grandes, em vez de pequenos e mais refinados. Usar componentes grandes reduz o volume de comunicação entre os componentes, já que a maioria das interações entre funcionalidades relacionadas ocorre dentro de um mesmo componente. Também é possível considerar organizações do sistema em tempo de execução que permitem que o sistema seja replicado e executado em diferentes processadores.
2. *Segurança da informação* (security). Se a segurança da informação for um requisito crítico, deve ser utilizada uma estrutura em camadas na arquitetura,

com os ativos mais críticos protegidos nas camadas mais internas e um alto nível de validação de segurança da informação aplicado a essas camadas.

3. *Segurança* (safety). Se a segurança for um requisito crítico, a arquitetura deve ser projetada para que as operações relacionadas à segurança fiquem juntas em um mesmo componente ou em um pequeno número de componentes. Isso reduz os custos e os problemas de validação da segurança e permite o fornecimento de sistemas de proteção relacionados, que, em caso de falha, possam desligar o sistema de maneira segura.
4. *Disponibilidade*. Se a disponibilidade for um requisito crítico, a arquitetura deve ser projetada para incluir componentes redundantes para que seja possível substituir e atualizar os componentes sem parar o sistema. No Capítulo 11, descreverei as arquiteturas de sistema tolerantes a defeitos para sistemas de alta disponibilidade.
5. *Manutenibilidade*. Se a manutenibilidade for um requisito crítico, a arquitetura do sistema deve ser projetada usando componentes pequenos e autocontidos que possam ser facilmente modificados. Os produtores de dados devem ser separados dos consumidores, e as estruturas de dados compartilhadas devem ser evitadas.

Obviamente, há um possível conflito entre algumas dessas arquiteturas. Por exemplo, usar componentes grandes melhora o desempenho e usar componentes pequenos melhora a manutenibilidade. Entretanto, se o desempenho e a manutenibilidade forem requisitos importantes do sistema, então deve haver algum acordo. Às vezes, é possível fazer isso usando diferentes padrões e estilos de arquitetura para diferentes partes do sistema. Hoje, a segurança da informação (*security*) quase sempre é um requisito crítico, e deve-se projetar uma arquitetura que satisfaça, ao mesmo tempo, este e outros requisitos não funcionais.

É difícil avaliar um projeto de arquitetura porque o verdadeiro teste de uma arquitetura é determinado por quão bem o sistema atende seus requisitos funcionais e não funcionais quando é utilizado. Entretanto, é possível fazer alguma avaliação comparando o projeto com arquiteturas de referência ou com padrões mais genéricos. A descrição de Bosch (2000) das características não funcionais de alguns padrões de arquitetura pode ajudar nessa avaliação.

6.2 VISÕES DE ARQUITETURA

Expliquei na introdução a este capítulo que os modelos de arquitetura de um sistema de software podem ser utilizados para focar a discussão sobre os requisitos ou o projeto (*design*) de software. Por outro lado, eles podem ser usados para documentar um projeto de modo que ele possa ser utilizado como base para o projeto detalhado e a implementação do sistema. Nesta seção, discutirei duas questões relevantes para ambas as utilizações:

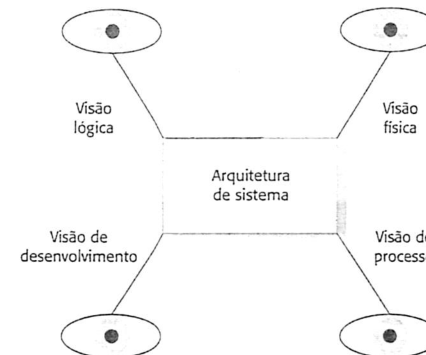
1. Quais visões ou perspectivas são úteis durante o projeto e documentação da arquitetura de um sistema?
2. Quais notações devem ser utilizadas para descrever os modelos de arquitetura?

É impossível representar em um único diagrama todas as informações relevantes a respeito da arquitetura de um sistema, pois um modelo gráfico só consegue mostrar uma visão ou perspectiva do sistema. Ele poderia mostrar como um sistema é decomposto em módulos, como os processos interagem em tempo de execução ou de quais maneiras distintas componentes de sistema se distribuem por uma rede. Como tudo isso é útil em momentos diferentes, tanto no projeto quanto na documentação, normalmente é preciso apresentar várias visões de arquitetura de software.

Existem diversas opiniões quanto às visões necessárias. Kruchten (1995), em seu conhecido modelo de arquitetura de software 4+1, sugere que deveria haver quatro visões fundamentais de arquitetura, que podem ser ligadas por meio de casos de uso e de cenários comuns (Figura 6.3). Ele sugere as seguintes:

1. *Uma visão lógica*, que mostra as abstrações fundamentais do sistema como objetos ou classes. Nesse tipo de visão, deve ser possível relacionar os requisitos do sistema às suas entidades.
2. *Uma visão de processo*, que mostra como, no tempo de execução, o sistema é composto de processos que interagem. Essa visão é útil para fazer julgamentos sobre características não funcionais do sistema, como o desempenho e a disponibilidade.
3. *Uma visão de desenvolvimento*, que mostra como o software é decomposto para desenvolvimento; isto é, mostra a divisão do software em componentes que são implementados por um único desenvolvedor ou time de desenvolvimento. Essa visão é útil para gerentes e programadores de software.
4. *Uma visão física*, que mostra o hardware do sistema e como os componentes de software estão distribuídos pelos processadores no sistema. Essa visão é útil para os engenheiros de sistema que estão planejando uma implantação do sistema.

FIGURA 6.3 Visões de arquitetura.



Hofmeister, Nord e Soni (2000) sugerem o uso de visões similares, mas acrescentam a ideia de uma visão conceitual. Essa é uma visão abstrata do sistema, que pode ser a base para decompor requisitos de alto nível em especificações mais detalhadas, ajudando os engenheiros a tomarem decisões sobre os componentes que podem ser reusados, e pode representar uma linha de produto (discutida no Capítulo 15) em vez de um único sistema. A Figura 6.1, que descreve a arquitetura de um robô empacotador, é um exemplo de visão conceitual do sistema.

Na prática, as visões conceituais da arquitetura de um sistema são quase sempre desenvolvidas durante o processo de projeto. Elas são utilizadas para explicar a arquitetura do sistema para os *stakeholders* e informar a tomada de decisão de arquitetura. Durante o processo de projeto, algumas das outras visões também podem ser desenvolvidas quando são discutidos diferentes aspectos do sistema, mas raramente é necessário desenvolver uma descrição completa de todas as perspectivas. Também pode ser possível associar padrões de arquitetura, que serão discutidos na próxima seção, com as diferentes visões de um sistema.

Existem diferentes visões quanto aos arquitetos de software deverem ou não usar a UML para descrever e documentar arquiteturas de software. Um levantamento feito em 2006 (LANGE; CHAUDRON; MUSKENS, 2006) mostrou que, quando a UML foi utilizada, ela foi aplicada principalmente de maneira informal. Os autores desse artigo argumentaram que isso não foi bom.

Eu discordo dessa opinião. A UML foi concebida para descrever sistemas orientados a objeto e, no estágio de projeto de arquitetura, muitas vezes se deseja descrever sistemas em um nível mais alto de abstração. As classes são próximas demais da implementação para serem úteis em uma descrição de arquitetura. Não acho que a UML seja útil durante o processo de projeto em si e prefiro notações informais que são mais rápidas de escrever e que podem ser facilmente desenhadas em uma lousa. A UML é mais valiosa quando se está documentando uma arquitetura em detalhes ou usando o desenvolvimento dirigido por modelos, conforme discutido no Capítulo 5.

Vários pesquisadores (BASS; CLEMENTS; KAZMAN, 2012) propuseram o uso de linguagens de descrição de arquitetura (ADLs, do inglês *architectural description languages*) mais especializadas para descrever arquiteturas de sistema. Os elementos básicos das ADLs são componentes e conectores, e elas incluem regras e diretrizes para arquiteturas bem formadas. No entanto, como as ADLs são linguagens especializadas, os especialistas do domínio e da aplicação acham difícil de entendê-las e de usá-las. Pode haver algum valor em usar ADLs específicas para domínios como parte do desenvolvimento dirigido por modelos, mas não acho que elas se tornarão parte da prática convencional de engenharia de software. Os modelos e as notações informais, como a UML, continuam a ser as maneiras mais utilizadas de documentar a arquitetura de sistema.

Os usuários dos métodos ágeis reivindicam que a documentação de projeto detalhada praticamente não é utilizada. Portanto, desenvolver esses documentos é desperdício de tempo e dinheiro. Concordo em grande parte com essa visão e acho que, exceto para sistemas críticos, não vale a pena desenvolver uma descrição de arquitetura detalhada a partir das quatro perspectivas de Kruchten. As visões que devem ser desenvolvidas são as que forem úteis para a comunicação, independentemente se a documentação de arquitetura está completa ou não.

6.3 PADRÕES DE ARQUITETURA

A ideia dos padrões como uma maneira de apresentar, compartilhar e reutilizar conhecimento sobre sistemas foi adotada em uma série de áreas da engenharia de software. O gatilho para isso foi a publicação de um livro sobre padrões de projeto orientado a objetos (GAMMA *et al.*, 1995). Isso suscitou o desenvolvimento de outros tipos de padrões, como os padrões de projeto organizacional (COPLIEN; HARRISON, 2004), de usabilidade (THE USABILITY GROUP, 1998), de interação cooperativa (MARTIN; SOMMERVILLE, 2004) e de gerenciamento de configuração (BERCZUK; APPLETON, 2002).

Os padrões de arquitetura foram propostos nos anos 1990 com o nome 'estilos de arquitetura' (SHAW; GARLAN, 1996). Uma série bem detalhada de manuais, em cinco volumes, sobre arquitetura de software orientada a objetos, foi publicada entre 1996 e 2007 (BUSCHMANN *et al.*, 1996; SCHMIDT *et al.*, 2000; BUSCHMANN; HENNEY; SCHMIDT, 2007a, 2007b; KIRCHER; JAIN, 2004).

Nesta seção, apresentarei os padrões de arquitetura e descreverei resumidamente uma seleção dos padrões mais utilizados. Esses padrões podem ser descritos usando uma mistura de descrição narrativa e de diagramas (Figuras 6.4 e 6.5). Para informações mais detalhadas sobre padrões e seu uso, vale a pena consultar os manuais de padrões publicados.

FIGURA 6.4 O padrão MVC (Modelo-Visão-Controlador).

Nome	MVC (Modelo-Visão-Controlador)
Descrição	Separa a apresentação e a interação dos dados do sistema. O sistema é estruturado em três componentes lógicos que interagem entre si. O componente Modelo gerencia os dados do sistema e as operações a eles associadas. O componente Visão define e gerencia como os dados são apresentados ao usuário. O componente Controlador gerencia a interação do usuário (por exemplo, pressionamento de teclas, cliques de mouse etc.) e passa essas interações para Visão e Modelo. Ver Figura 6.5.
Exemplo	A Figura 6.6 mostra a arquitetura de uma aplicação web, organizada com o uso do padrão MVC.
Quando é utilizado	É utilizado quando há várias maneiras de visualizar e interagir com os dados. Também é utilizado quando os requisitos futuros para interação e apresentação dos dados são desconhecidos.
Vantagens	Permite que os dados sejam alterados independentemente de sua representação e vice-versa. Apoia a apresentação dos mesmos dados de maneiras diferentes, exibindo as alterações feitas em uma representação em todas as demais.
Desvantagens	Pode envolver mais código e aumentar sua complexidade quando o modelo de dados e as interações forem simples.

É possível pensar em um padrão de arquitetura como uma descrição estilizada, abstrata, das práticas recomendadas que foram testadas e aprovadas em diferentes subsistemas e ambientes. Então, um padrão de arquitetura deve descrever a organização de um sistema que foi bem-sucedida em sistemas anteriores. Ela deve incluir tanto informações sobre quando é apropriado usar esse padrão e também detalhes de pontos fortes e fracos do padrão.

A Figura 6.4 descreve o conhecido padrão MVC (Modelo-Visão-Controlador), que é a base do gerenciamento da interação em muitos sistemas web, sendo suportado pela maioria dos *frameworks*. A descrição estilizada do padrão inclui o nome do padrão, uma descrição resumida, um modelo gráfico e um exemplo do tipo de sistema no qual o padrão é utilizado. Devem ser incluídas informações sobre quando o padrão é utilizado e suas vantagens e desvantagens.

Os modelos gráficos da arquitetura associada ao padrão MVC são mostrados nas Figuras 6.5 e 6.6. Eles apresentam a arquitetura a partir de diferentes visões: a Figura 6.5 é uma visão conceitual e a Figura 6.6 mostra uma arquitetura de sistema em tempo de execução quando esse padrão é utilizado para gerenciamento da interação em um sistema web.

FIGURA 6.5 A organização do MVC (Modelo-Visão-Controlador).

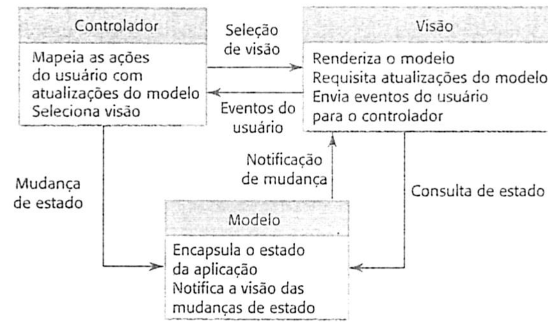
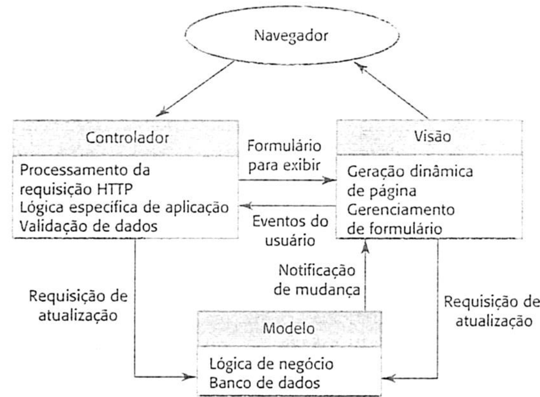


FIGURA 6.6 Arquitetura de aplicação web usando o padrão MVC.



Nesse pequeno espaço, é impossível descrever todos os padrões genéricos que podem ser utilizados no desenvolvimento de software. Em vez disso, apresento alguns exemplos selecionados de padrões amplamente utilizados e que capturam bons princípios de projeto de arquitetura.

6.3.1 Arquitetura em camadas

Os conceitos de separação e de independência são fundamentais para o projeto de arquitetura porque permitem que as mudanças sejam localizadas. O padrão MVC, exibido na Figura 6.4, separa os elementos de um sistema, permitindo que sejam alterados de maneira independente. Por exemplo, adicionar uma nova visão ou mudar uma visão existente pode ser feito sem quaisquer alterações nos dados subjacentes ao modelo. O padrão de arquitetura em camadas é outra maneira de alcançar a separação e a independência. Esse padrão é exibido na Figura 6.7. Aqui, a funcionalidade do sistema é organizada em camadas separadas e cada uma se baseia apenas nos recursos e serviços oferecidos pela camada imediatamente abaixo dela.

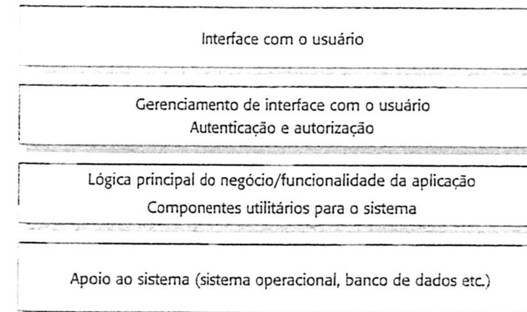
FIGURA 6.7 O padrão de arquitetura em camadas.

Nome	Arquitetura em camadas
Descrição	Organiza o sistema em camadas, com funcionalidade associada a cada uma. Uma camada fornece serviços para a camada acima dela, então as camadas nos níveis mais inferiores representam os serviços essenciais que tendem a ser utilizados em todo o sistema (Figura 6.8).
Exemplo	Um modelo em camadas de um sistema de aprendizagem digital para apoiar a aprendizagem de todas as disciplinas nas escolas (Figura 6.9).
Quando é utilizado	Utilizado quando se cria novos recursos em cima de sistemas existentes; quando o desenvolvimento é distribuído por vários times, cada um deles responsável por uma camada de funcionalidade; quando há necessidade de segurança da informação (*security*) em múltiplos níveis.
Vantagens	Permite a substituição de camadas inteiras, contanto que a interface seja mantida. Recursos redundantes, como a autenticação, podem ser fornecidos em cada camada para aumentar a dependabilidade do sistema.
Desvantagens	Na prática, muitas vezes é difícil proporcionar uma separação clara entre as camadas, de modo que camadas dos níveis mais altos podem ter de interagir diretamente com as dos níveis mais baixos em vez das imediatamente inferiores a elas. O desempenho pode ser um problema por causa dos múltiplos níveis de interpretação de uma requisição de serviço à medida que essa requisição é processada em cada camada.

Essa abordagem em camadas apoia o desenvolvimento incremental de sistemas. À medida que uma camada é desenvolvida, alguns dos serviços fornecidos por ela podem ser disponibilizados aos usuários. A arquitetura também é mutável e móvel. Se a sua interface permanecer inalterada, uma nova camada com funcionalidade ampliada pode substituir uma camada existente sem mudar outras partes do sistema. Além disso, quando as interfaces de camada mudam ou quando novos recursos são acrescentados a uma camada, apenas a camada adjacente é afetada. Como os sistemas em camadas localizam dependências de máquina, isso facilita o fornecimento de implementações multiplataforma de um sistema de aplicação. Somente as camadas dependentes de máquina precisam ser implementadas novamente para levar em conta os recursos de um sistema operacional ou banco de dados diferente.

A Figura 6.8 é um exemplo de arquitetura em camadas, contando com quatro camadas. A camada de nível mais baixo inclui software de apoio ao sistema — tipicamente, apoio ao banco de dados e ao sistema operacional. A próxima camada é a de aplicação, que inclui os componentes relacionados à funcionalidade da aplicação e os componentes utilitários aproveitados por outros componentes da aplicação.

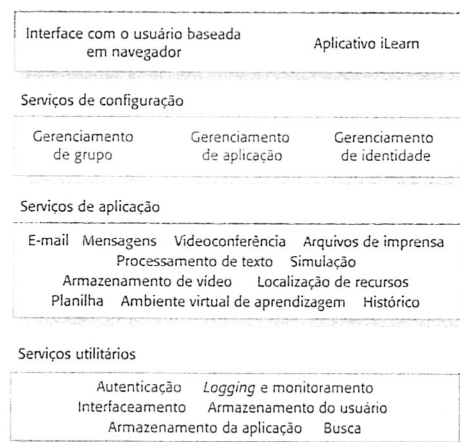
FIGURA 6.8 Uma arquitetura genérica em camadas.



A terceira camada está relacionada ao gerenciamento da interface com o usuário e ao fornecimento de autenticação e autorização de usuário, com a camada superior fornecendo recursos de interface com o usuário. Naturalmente, o número de camadas é arbitrário. Qualquer uma das camadas na Figura 6.6 poderia ser dividida em duas ou mais.

A Figura 6.9 mostra que o sistema de aprendizado digital iLearn, introduzido no Capítulo 1, possui uma arquitetura de quatro camadas que segue esse padrão. Há outro exemplo de padrão de arquitetura em camadas na Figura 6.19 (na Seção 6.4, que mostra a organização do sistema Mentcare).

FIGURA 6.9 A arquitetura do sistema iLearn.



6.3.2 Arquitetura de repositório

O padrão de arquitetura em camadas e o padrão MVC são exemplos nos quais a visão apresentada é a organização conceitual de um sistema. Meu próximo exemplo, o padrão repositório (Figura 6.10), descreve como um conjunto de componentes que interagem pode compartilhar dados.

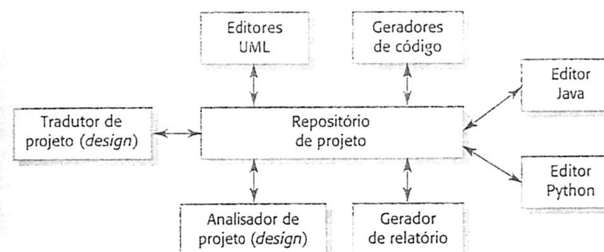
A maioria dos sistemas que usa grandes quantidades de dados é organizada em torno de um banco de dados ou repositório compartilhado. Portanto, esse modelo é adequado para aplicações nas quais dados são gerados por um componente e usados por outro. Exemplos desse tipo de sistema incluem sistemas de comando e controle, sistemas de gerenciamento de informações, sistemas CAD (projeto assistido por computador, do inglês *computer aided design*) e ambientes interativos de desenvolvimento de software.

A Figura 6.11 ilustra uma situação em que se pode utilizar um repositório. Esse diagrama mostra um IDE que inclui diferentes ferramentas para apoiar um desenvolvimento dirigido por modelos. O repositório, nesse caso, pode ser um ambiente com controle de versão (conforme será discutido no Capítulo 25), que monitora as mudanças no software e permite a restauração de versões anteriores.

FIGURA 6.10 Padrão repositório.

Nome	Repositório
Descrição	Todos os dados em um sistema são gerenciados em um repositório central que é acessível a todos os componentes do sistema. Os componentes não interagem diretamente, apenas por meio do repositório.
Exemplo	A Figura 6.11 é um exemplo de IDE cujos componentes usam um repositório de informações de projeto (*design*) do sistema. Cada ferramenta de software gera informações que depois são disponibilizadas para uso de outras ferramentas.
Quando é utilizado	Esse padrão deve ser usado em um sistema no qual são gerados grandes volumes de informação que precisam ser armazenados por muito tempo. Também é possível usá-lo nos sistemas dirigidos por dados, cuja inclusão no repositório dispara uma ação ou uma ferramenta.
Vantagens	Os componentes podem ser independentes; eles não precisam saber da existência dos outros componentes. As mudanças feitas por um componente podem ser propagadas para todos os demais. Todos os dados podem ser gerenciados de modo consistente (por exemplo, *backups* feitos ao mesmo tempo), já que estão todos em um só lugar.
Desvantagens	O repositório é um único ponto de falha, então os problemas no repositório afetam o sistema inteiro. Pode haver ineficiências em organizar toda a comunicação por meio do repositório. Pode ser difícil distribuir o repositório por vários computadores.

FIGURA 6.11 Uma arquitetura de repositório para um IDE.



Organizar as ferramentas em torno de um repositório é uma maneira eficiente de compartilhar uma grande quantidade de dados. Não é necessário transmitir os dados explicitamente de um componente para outro. No entanto, os componentes devem operar em torno de um modelo aprovado de repositório de dados. Inevitavelmente, esse é um acordo entre as necessidades específicas de cada ferramenta e pode ser difícil ou impossível integrar novos componentes se os modelos de dados não se encaixarem no esquema aprovado. Na prática, pode ser difícil distribuir o repositório por uma série de máquinas. Embora seja possível distribuir um repositório logicamente centralizado, isso envolve manter várias cópias dos dados. Manter esses dados coerentes e atualizados sobrecarrega o sistema.

Na arquitetura de repositório exibida na Figura 6.11, o repositório é passivo e o controle é de responsabilidade dos componentes que usam o repositório. Uma abordagem alternativa, que foi derivada dos sistemas de inteligência artificial (IA), usa um modelo de 'quadro-negro' que dispara componentes quando determinados dados ficam disponíveis. Isso é conveniente quando os dados no repositório não são estruturados. As decisões sobre qual ferramenta deve ser ativada só podem ser tomadas quando os dados forem analisados. Esse modelo foi introduzido por Nii (1986), e Bosch (2000) inclui uma boa discussão sobre como esse estilo está relacionado aos atributos de qualidade do sistema.

6.3.3 Arquitetura cliente-servidor

O padrão de repositório preocupa-se com a estrutura estática de um sistema e não mostra a sua organização em tempo de execução. Meu próximo exemplo, o padrão cliente-servidor (Figura 6.12), ilustra uma organização em tempo de execução utilizada frequentemente em sistemas distribuídos. Um sistema que segue o padrão cliente-servidor é organizado como um conjunto de serviços e servidores associados e de clientes que acessam e usam esses serviços. Os principais componentes desse modelo são:

1. Um conjunto de servidores que oferecem serviços para outros componentes. Os exemplos incluem os servidores de impressão, que oferecem serviços de impressão; os servidores de arquivos, que oferecem serviços de gerenciamento de arquivos; e um servidor de compilação, que oferece serviços de compilação de linguagem de programação. Os servidores são componentes de software, e vários deles podem ser executados no mesmo computador.
2. Um conjunto de clientes que demanda os serviços oferecidos pelos servidores. Normalmente haverá várias instâncias de um programa cliente sendo executado simultaneamente em computadores diferentes.
3. Uma rede que permite que os clientes acessem esses serviços. Os sistemas cliente-servidor normalmente são implementados como sistemas distribuídos, conectados por protocolos da internet.

FIGURA 6.12 O padrão cliente-servidor.

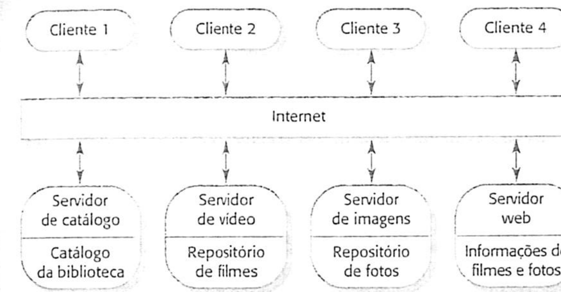
Nome	Cliente-servidor
Descrição	Em uma arquitetura cliente-servidor, o sistema é apresentado como um conjunto de serviços, e cada serviço é fornecido por um servidor separado. Os clientes são usuários desses serviços e acessam os servidores para usá-los.
Exemplo	A Figura 6.13 é um exemplo de biblioteca de filmes e vídeos/DVDs organizada como um sistema cliente-servidor.
Quando é utilizado	Utilizado quando os dados em um banco de dados compartilhado têm de ser acessados a partir de diversos locais. Como os servidores podem ser replicados, eles também podem ser utilizados quando a carga em um sistema for variável.
Vantagens	A principal vantagem desse modelo é que os servidores podem ser distribuídos em rede. A funcionalidade geral (por exemplo, um serviço de impressão) pode estar disponível para todos os clientes e não precisa ser implementada por todos os serviços.
Desvantagens	Cada serviço é um único ponto de falha e, portanto, é suscetível a ataques de negação de serviço ou a falhas no servidor. O desempenho pode ser imprevisível porque depende da rede e também do sistema. Podem surgir problemas de gerenciamento se os servidores forem de propriedade de organizações diferentes.

As arquiteturas cliente-servidor normalmente são encaradas como arquiteturas de sistemas distribuídos, mas o modelo lógico de serviços independentes sendo executados em servidores diferentes pode ser implementado em um único computador. Mais uma vez, os benefícios importantes são a separação e a independência. Os serviços e servidores podem ser modificados sem afetar outras partes do sistema.

Os clientes podem ser obrigados a saber os nomes dos servidores disponíveis e os serviços que eles fornecem. No entanto, os servidores não precisam saber qual a identidade dos clientes ou quantos clientes estão acessando seus serviços. Os clientes acessam os serviços fornecidos por um servidor por meio de chamadas remotas de procedimento usando um protocolo de requisição-resposta (como o HTTP), no qual um cliente faz uma solicitação a um servidor e deve esperar até receber a resposta.

A Figura 6.13 é um exemplo de sistema baseado no modelo cliente-servidor. Esse sistema web é multiusuário e controla uma biblioteca de filmes e de fotos. Nele, vários servidores gerenciam e exibem diferentes tipos de mídia. Os quadros do vídeo precisam ser transmitidos rapidamente e em sincronia, mas em uma resolução relativamente baixa. Eles podem ser comprimidos em um repositório, então o servidor de vídeo pode lidar com a compressão e descompressão de vídeo em diferentes formatos. As imagens estáticas, porém, devem ser mantidas em uma resolução alta, então é conveniente mantê-las em um servidor separado.

FIGURA 6.13 Arquitetura cliente-servidor de uma biblioteca de filmes.



O catálogo deve ser capaz de lidar com uma série de consultas e de fornecer *links* para o sistema de informação na web, que inclui dados sobre filmes e videoclipes, e um sistema de *e-commerce* que permita a venda de fotografias, filmes e videoclipes. O programa cliente é simplesmente uma interface de usuário integrada, construída usando um navegador web para acessar esses serviços.

A vantagem mais importante do modelo cliente-servidor é que se trata de uma arquitetura distribuída. O uso eficaz pode ser feito nos sistemas em rede que contam com muitos processadores distribuídos. É fácil adicionar um novo servidor e integrá-lo ao resto do sistema ou atualizar os servidores transparentemente sem afetar as outras partes do sistema. Cobrirei as arquiteturas distribuídas no Capítulo 17, no qual explicarei com mais detalhes o modelo cliente-servidor e suas variações.

6.3.4 Arquitetura duto e filtro

Meu exemplo final de um padrão de arquitetura genérico é o duto e filtro (*pipe and filter* — Figura 6.14), um modelo de organização de um sistema em tempo de execução no qual transformações funcionais processam suas entradas e produzem saídas. Os dados fluem de um para outro e são transformados enquanto passam pela sequência. Cada etapa de processamento é implementada como uma transformação. Os dados de entrada fluem por essas transformações até serem convertidos para saída. As transformações podem ser executadas sequencial ou paralelamente. Os dados podem ser processados por cada transformação, item a item ou em um único lote.

FIGURA 6.14 O padrão duto e filtro.

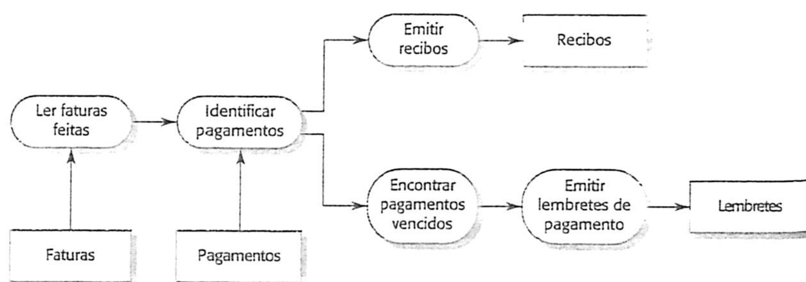
Nome	Duto e filtro *(pipe and filter)*
Descrição	O processamento dos dados em um sistema é organizado de modo que cada componente de processamento (filtro) é discreto e executa um tipo de transformação dos dados. Os dados fluem (como em um duto) de um componente para outro para serem processados.
Exemplo	A Figura 6.15 é um exemplo de sistema duto e filtro utilizado para processar pedidos.
Quando é utilizado	Utilizado frequentemente nas aplicações de processamento de dados (tanto baseadas em lotes quanto em transações), em que as entradas são processadas em estágios separados, gerando saídas relacionadas.
Vantagens	Fácil de entender e permite o reúso de transformações. O estilo do fluxo de trabalho corresponde à estrutura de muitos processos de negócio. A evolução por meio da adição de transformações é direta. Pode ser implementado como um sistema sequencial ou concorrente.
Desvantagens	O formato da transferência de dados tem de ser acordado entre as transformações que se comunicam. Cada transformação deve analisar sua entrada e devolver a saída para a forma acordada. Isso aumenta a sobrecarga do sistema e pode significar que é impossível reusar componentes arquiteturais que usam estruturas de dados incompatíveis.

O nome 'duto e filtro' vem do sistema Unix original, no qual era possível vincular processos usando *pipes*. Esses *pipes* passavam um fluxo de texto de um processo para outro. Os sistemas que seguem esse modelo podem ser implementados combinando comandos Unix, usando *pipes* e recursos de controle do Unix *shell*. O termo *filtro* (*filter*) é utilizado porque uma transformação 'filtra' os dados que pode processar a partir de seu fluxo de dados de entrada.

Variações desse padrão têm sido utilizadas desde que os computadores foram introduzidos no processamento automático de dados. Quando as transformações são sequenciais com dados processados em lotes, esse modelo de arquitetura duto e filtro se transforma em um modelo sequencial em lotes — uma arquitetura comum nos sistemas de processamento de dados, como os sistemas de cobrança. A arquitetura de um sistema embarcado também pode ser organizada como um *pipeline* de processo, com cada processo executando concorrentemente. No Capítulo 21, cobrirei o uso desse padrão nos sistemas embarcados.

Um exemplo desse tipo de arquitetura de sistema, utilizado em uma aplicação de processamento em lotes, é exibido na Figura 6.15. Uma organização enviou faturas aos clientes. Uma vez por semana, os pagamentos feitos são reconciliados com as faturas; para as faturas pagas, um recibo é emitido; para as que estão em atraso, um lembrete é emitido.

FIGURA 6.15 Exemplo da arquitetura duto e filtro.



Os sistemas duto e filtro são mais adequados para sistemas de processamento em lotes e sistemas embarcados nos quais há interação limitada do usuário. Os sistemas interativos são difíceis de escrever usando o modelo duto e filtro em razão da necessidade de processar um fluxo de dados. Embora a entrada e a saída textuais simples possam ser modeladas dessa maneira, as interfaces gráficas com o usuário têm formatos de E/S mais complexos e uma estratégia de controle baseada em eventos, como cliques de mouse ou seleções de menu. É difícil implementar isso como um fluxo sequencial em conformidade com o modelo duto e filtro.

Padrões de arquitetura para controle

Existem padrões de arquitetura específicos que refletem as maneiras mais utilizadas para organizar o controle em um sistema. Esses padrões incluem controle centralizado, baseado em um componente invocando outros componentes; e o controle baseado em evento, no qual o sistema reage a eventos externos.

6.4 ARQUITETURAS DE APLICAÇÕES

Os sistemas de aplicação se destinam a satisfazer a necessidade de um negócio ou de uma empresa. Todas as empresas têm muito em comum — elas precisam contratar pessoas, emitir faturas, manter contas e assim por diante. As empresas que operam no mesmo setor usam aplicações comuns específicas para o setor. Portanto, assim como as funções comerciais mais genéricas, todas as empresas de telefonia precisam de sistemas para conectar e medir a duração das chamadas, gerenciar sua rede e emitir faturas para seus clientes. Consequentemente, os sistemas de aplicação utilizados por essas empresas também têm muito em comum.

Essas semelhanças levaram ao desenvolvimento de arquiteturas de software que descrevem a estrutura e a organização de determinados tipos de sistemas de software. As arquiteturas de aplicação encapsulam as características principais de uma classe de sistemas. Por exemplo, em sistemas de tempo real, pode haver modelos de arquitetura genéricos de diferentes tipos de sistemas, como os de coleta de dados ou os de monitoramento. Embora as instâncias desses sistemas sejam diferentes em seus detalhes, a estrutura de arquitetura comum pode ser reusada durante o desenvolvimento de novos sistemas do mesmo tipo.

A arquitetura de aplicação pode ser reimplementada durante o desenvolvimento de novos sistemas. No entanto, em muitos sistemas de negócio, ao configurar sistemas genéricos para a criação de uma nova aplicação, o reúso da arquitetura de aplicação está implícito. Vemos isso no uso generalizado dos sistemas ERP e dos sistemas de prateleira configuráveis, como os de contabilidade e os de controle de estoque, todos eles sistemas com arquitetura e componentes padrões. Os componentes são configurados e adaptados para criar uma aplicação comercial específica. Por exemplo, um sistema para gerenciamento da cadeia de suprimento pode ser adaptado para diferentes tipos de fornecedores, bens e arranjos contratuais.

Arquiteturas de aplicação

Existem vários exemplos de arquiteturas de aplicação no site do livro. Esses exemplos incluem descrições de sistemas de processamento em lotes, sistemas de alocação de recursos e sistemas de edição baseados em eventos.

Um projetista de software pode usar modelos de arquiteturas de aplicação de várias maneiras:

1. *Como ponto de partida para o processo de projeto de arquitetura.* Caso não haja familiaridade com o tipo de aplicação que está sendo desenvolvido, é possível basear o projeto inicial em uma arquitetura de aplicação genérica. Depois, essa arquitetura pode ser especificada para o sistema que estiver sendo desenvolvido.
2. *Como um* checklist *do projeto (*design*).* Ao desenvolver um projeto de arquitetura para um sistema de aplicação, é possível compará-lo a uma arquitetura de aplicação mais genérica. Isso permite verificar se o projeto está coerente com a arquitetura genérica.
3. *Como uma maneira de organizar o trabalho do time de desenvolvimento.* As arquiteturas de aplicação identificam características estruturais estáveis das arquiteturas de sistema e, em muitos casos, é possível desenvolver essas características paralelamente. É possível distribuir o trabalho entre os membros de um time para a implementação de diferentes componentes da arquitetura.
4. *Como um meio de avaliar os componentes para reúso.* Caso haja componentes passíveis de reúso, é possível compará-los com as estruturas genéricas para ver se há componentes similares na arquitetura de aplicação.
5. *Como um vocabulário para falar sobre aplicações.* Ao discutir uma aplicação específica ou comparar aplicações, é possível usar os conceitos identificados na arquitetura genérica para falar sobre essas aplicações.

Existem muitos tipos de sistemas de aplicação e, em alguns casos, eles podem parecer muito diferentes. No entanto, aplicações diferentes na superfície podem ter muito em comum e, assim, compartilhar uma arquitetura de aplicação abstrata. Ilustro isso descrevendo as arquiteturas de dois tipos de aplicação:

1. *Aplicações de processamento de transações.* São aplicações centradas em bancos de dados, que processam as solicitações de informação feitas pelos usuários e as atualizam em um banco de dados. Esses são os tipos mais comuns de sistemas de negócio interativos e são organizados de modo que as ações do usuário não interfiram umas nas outras e que a integridade do banco de dados seja mantida. Essa classe de sistema inclui os sistemas bancários interativos, os de *e-commerce*, os de informação e os de reservas.
2. *Sistemas de processamento de linguagem.* São sistemas nos quais as intenções do usuário são expressas em uma linguagem formal, como uma linguagem de programação. O sistema processa essa linguagem e a transforma em um formato interno, interpretando posteriormente essa representação interna. Os sistemas de processamento de linguagem mais conhecidos são os compiladores, que traduzem programas em linguagem de alto nível em código de máquina. No entanto, os sistemas de processamento de linguagem também são utilizados para interpretar linguagens de comando para bancos de dados e sistemas de informação, bem como linguagens de marcação, como XML.

Escolhi esses tipos particulares de sistemas porque uma grande quantidade de sistemas web de negócio são sistemas de processamento de transações, e todo desenvolvimento de software depende dos sistemas de processamento de linguagem.

6.4.1 Sistemas de processamento de transações

Os sistemas de processamento de transações são concebidos para processar requisições de um usuário por informação em um banco de dados ou requisições para atualizar um banco de dados (LEWIS; BERNSTEIN; KIFER, 2003). Tecnicamente, uma transação de banco de dados faz parte de uma sequência de operações que é tratada como uma única unidade (uma unidade atômica). Todas as operações em uma transação devem ser realizadas antes de as mudanças no banco de dados se tornarem permanentes. Isso garante que a falha das operações dentro de uma transação não leve a incoerências no banco de dados.

Na perspectiva do usuário, uma transação é qualquer sequência de operações coerente que satisfaz uma meta, como 'encontrar os horários de voos de Londres para Paris'. Se a transação do usuário não exigir uma alteração no banco de dados, então não é necessário que seja empacotada como uma transação técnica de banco de dados.

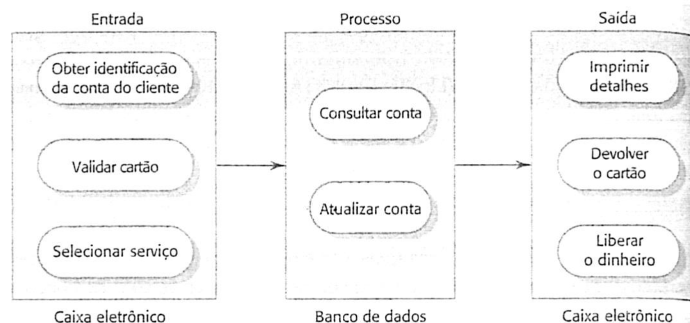
Um exemplo de transação de banco de dados é a solicitação de um cliente para retirar dinheiro de uma conta bancária usando um caixa eletrônico. Isso envolve conferir se há saldo na conta do cliente, modificar o saldo relativo à quantidade retirada e enviar comandos para o caixa eletrônico entregar o dinheiro. Enquanto todas essas etapas não forem concluídas, a transação está incompleta e o banco de dados das contas de cliente não é alterado.

Os sistemas de processamento de transações normalmente são interativos, nos quais os usuários fazem solicitações assíncronas de serviço. A Figura 6.16 ilustra a estrutura de arquitetura conceitual das aplicações de processamento de transações. Primeiramente, um usuário faz um pedido ao sistema por meio de um componente de processamento de E/S. Esse pedido é processado por uma lógica específica do sistema. Uma transação é criada e passada para um gerenciador de transações, que normalmente está embutido no sistema de gerenciamento do banco de dados. Após o gerenciador de transações garantir que a transação foi concluída adequadamente, a aplicação é avisada de que o processamento terminou.

FIGURA 6.16 Estrutura das aplicações de processamento de transações.

Os sistemas de processamento de transações podem ser organizados como uma arquitetura duto e filtro, em que os componentes do sistema são responsáveis pela entrada, pelo processamento e pela saída. Por exemplo, um sistema bancário, que permite aos clientes consultarem suas contas e retirarem dinheiro de um caixa eletrônico, é composto de dois componentes de software cooperativos: o software do caixa eletrônico e o software de processamento das contas no servidor de banco de dados da instituição bancária. Os componentes de entrada e de saída são implementados como software no caixa eletrônico, e o componente de processamento faz parte do servidor de banco de dados da instituição bancária. A Figura 6.17 mostra a arquitetura desse sistema, ilustrando as funções dos componentes de entrada, de processamento e de saída.

FIGURA 6.17 Arquitetura de software de um sistema de caixa eletrônico.

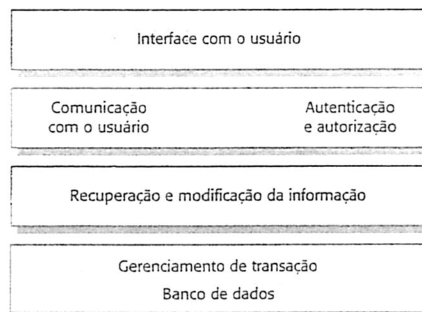


6.4.2 Sistemas de informação

Todos os sistemas que envolvem interação com um banco de dados compartilhado podem ser considerados sistemas de informação baseados em transação. Um sistema de informação permite o acesso controlado a uma grande base de informações, como um catálogo de uma biblioteca, uma tabela de horários de voos ou os registros dos pacientes em um hospital. Os sistemas de informação quase sempre são sistemas web, cuja interface com o usuário é acessada por meio de um navegador.

A Figura 6.18 apresenta um modelo bem genérico de sistema de informação. O sistema é modelado usando uma abordagem em camadas (discutida na Seção 6.3), em que a camada superior apoia a interface com o usuário e a camada inferior é o banco de dados do sistema. A camada de comunicação com o usuário lida com todas as entradas e saídas da interface com o usuário, e a camada de recuperação de informações inclui a lógica da aplicação para acessar e atualizar o banco de dados. Nesse modelo, as camadas podem ser mapeadas diretamente nos servidores, em um sistema web distribuído.

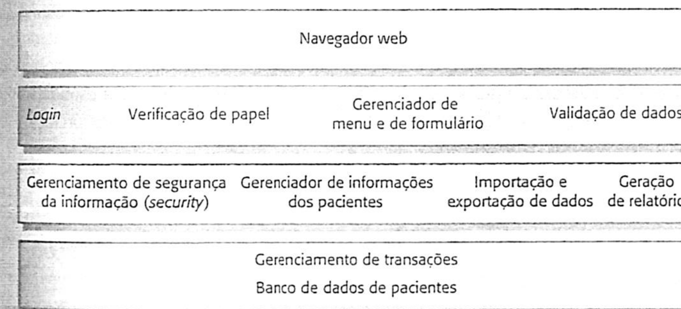
FIGURA 6.18 Arquitetura de sistema de informação em camadas.



Como exemplo de uma instanciação desse modelo em camadas, a Figura 6.19 mostra a arquitetura do sistema Mentcare. Vale lembrar que esse sistema mantém e gerencia detalhes dos pacientes que consultam especialistas em problemas de saúde mental. Acrescentei detalhes a cada camada do modelo, identificando os componentes que oferecem apoio às comunicações com o usuário e à recuperação e ao acesso às informações:

1. A camada superior é uma interface com o usuário baseada em navegador.
2. A segunda camada tem a funcionalidade da interface de usuário que é fornecida por meio do navegador web. Ela inclui componentes para permitir que os usuários acessem o sistema e componentes de verificação para garantir que as operações que eles utilizam são permitidas de acordo com seu papel. Essa camada inclui componentes de gerenciamento de formulários e de menu, que apresentam informações para os usuários, e componentes de validação de dados, que verificam a coerência das informações.
3. A terceira camada implementa a funcionalidade do sistema e fornece componentes que implementam a segurança da informação (*security*) do sistema, a criação e a atualização de informações de pacientes, a importação e a exportação de dados de pacientes de outros bancos de dados, além dos geradores de relatório que criam relatórios gerenciais.
4. Finalmente, a camada mais baixa, que é construída usando um sistema de gerenciamento de banco de dados comercial, proporciona a supervisão das transações e o armazenamento persistente dos dados.

FIGURA 6.19 Arquitetura do sistema Mentcare.



Os sistemas de informação e de gerenciamento de recursos às vezes também são sistemas de processamento de transações. Por exemplo, os sistemas de *e-commerce* são sistemas web de gerenciamento de recursos que aceitam pedidos eletrônicos de bens ou serviços e depois providenciam sua entrega ao cliente. Em um sistema de *e-commerce*, as camadas específicas para a aplicação incluem outra funcionalidade, que é um 'carrinho de compras' no qual os usuários podem adicionar uma série de itens em transações separadas e, em seguida, pagar por eles em uma única transação.

A organização dos servidores nesses sistemas normalmente reflete o modelo genérico de quatro camadas apresentado na Figura 6.18. Esses sistemas são implementados frequentemente como sistemas distribuídos, com uma arquitetura cliente-servidor multicamadas:

1. O servidor web é responsável por todas as comunicações com o usuário, e a interface é acessada por meio de um navegador web.
2. O servidor de aplicação é responsável por implementar a lógica específica da aplicação e também os pedidos de armazenamento e recuperação de informações.
3. O servidor de banco de dados move as informações de/para o banco de dados e lida com o gerenciamento de transações.

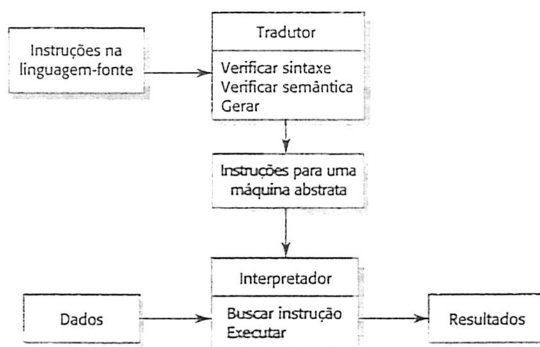
Usar vários servidores permite uma alta vazão (*throughput*) e possibilita o manuseio de milhares de transações por minuto. À medida que a demanda aumenta, servidores podem ser acrescentados em cada nível para lidar com o processamento extra envolvido.

6.4.3 Sistemas de processamento de linguagem

Os sistemas de processamento de linguagem traduzem uma linguagem para uma representação alternativa e, no caso das linguagens de programação, também podem executar o código resultante. Os compiladores traduzem uma linguagem de programação em código de máquina. Outros sistemas de processamento de linguagem podem traduzir uma descrição de dados em XML em comandos para consultar um banco de dados ou em uma representação XML alternativa. Os sistemas de processamento de linguagem natural podem traduzir uma linguagem em outra, como do francês para o norueguês.

A Figura 6.20 ilustra uma possível arquitetura para um sistema de processamento de linguagem. As instruções da linguagem-fonte definem o programa a ser executado, e um tradutor converte isso em instruções para uma máquina abstrata. Essas instruções são interpretadas por outro componente que as busca e executa usando, se necessário, dados do ambiente. A saída desse processo é o resultado da interpretação das instruções nos dados de entrada.

FIGURA 6.20 Arquitetura de um sistema de processamento de linguagem.



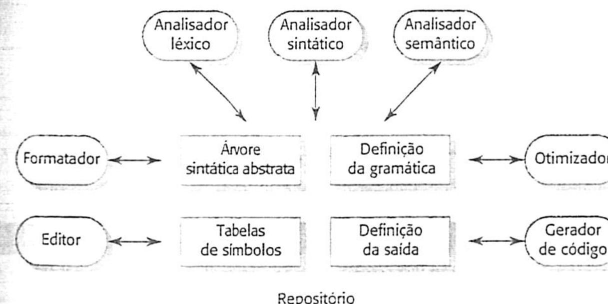
Em muitos compiladores, o interpretador é o hardware do sistema que processa instruções de máquina, e a máquina abstrata é um processador real. No entanto, nas linguagens dinamicamente tipadas, como Ruby e Python, o interpretador é um componente de software.

Os compiladores das linguagens de programação que fazem parte de um ambiente de programação mais geral têm uma arquitetura genérica (Figura 6.21) que inclui os seguintes componentes:

1. Um analisador léxico, que pega os *tokens* da linguagem de entrada e os converte em um formato interno.
2. Uma tabela de símbolos, que guarda informações sobre os nomes das entidades (variáveis, nomes de classe, nomes de objetos etc.) utilizados no texto que está sendo traduzido.
3. Um analisador sintático, que verifica a sintaxe da linguagem que está sendo traduzida. Ele usa uma gramática da linguagem definida e cria uma árvore de sintaxe.
4. Uma árvore de sintaxe, que é uma estrutura interna representando o programa que está sendo compilado.
5. Um analisador semântico, que usa as informações da árvore de sintaxe e a tabela de símbolos para verificar a correção semântica do texto da linguagem de entrada.
6. Um gerador de código, que 'caminha' pela árvore de sintaxe e gera código de máquina abstrata.

Outros componentes também poderiam ser incluídos para analisar e transformar a árvore de sintaxe e para melhorar a eficiência e remover a redundância do código de máquina gerado.

FIGURA 6.21 Arquitetura de repositório para um sistema de processamento de linguagem.



Arquiteturas de referência

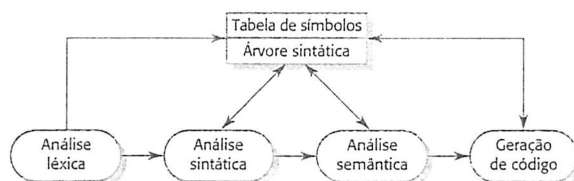
As arquiteturas de referência capturam características importantes das arquiteturas de sistema em um domínio. Essencialmente, elas incluem tudo o que poderia estar em uma arquitetura de aplicação, apesar de, na realidade, ser muito improvável que qualquer aplicação viesse a incluir todas as características exibidas em uma arquitetura de referência. A principal finalidade das arquiteturas de referência é avaliar e comparar propostas de projeto e educar as pessoas a respeito das características de arquitetura naquele domínio.

Em outros tipos de sistema de processamento de linguagem, como um tradutor de linguagem natural, haverá outros componentes, como um dicionário. A saída do sistema é a tradução do texto de entrada.

A Figura 6.21 ilustra como um sistema de processamento de linguagem pode fazer parte de um conjunto integrado de ferramentas de apoio à programação. Nesse exemplo, a tabela de símbolos e a árvore sintática agem como um repositório central de informações. Ferramentas ou fragmentos de ferramenta se comunicam por meio desse sistema. Outras informações, que às vezes estão embutidas nas ferramentas, como a definição da gramática e a definição do formato de saída do programa, foram tiradas das ferramentas e colocadas no repositório. Portanto, um editor dirigido por sintaxe consegue verificar se a sintaxe de um programa está correta enquanto se digita nele. Um formatador de programa consegue criar listagens do programa que realçam os elementos sintáticos diferentes e, portanto, são mais fáceis de ler e entender.

Padrões de arquitetura alternativos podem ser utilizados em um sistema de processamento de linguagem (GARLAN; SHAW, 1993). Compiladores podem ser implementados usando uma composição de repositório e modelo duto e filtro. Em uma arquitetura de compilador, a tabela de símbolos é um repositório para dados compartilhados. As fases da análise léxica, sintática e semântica são organizadas sequencialmente, conforme a Figura 6.22, e se comunicam por meio da tabela de símbolos compartilhada.

FIGURA 6.22 Arquitetura de compilador duto e filtro.



Esse modelo duto e filtro de compilação de linguagem é eficaz nos ambientes em lote, nos quais os programas são compilados e executados sem interação com o usuário; por exemplo, na tradução de um documento XML para outro. Ele é menos eficaz quando um compilador é integrado a outras ferramentas de processamento de linguagem, como um sistema de edição estruturada, um depurador interativo ou um formatador de programa. Nessa situação, as mudanças de um componente precisam ser refletidas imediatamente nos outros componentes. É melhor organizar o sistema em torno de um repositório, conforme a Figura 6.21, caso seja implementado um ambiente de programação geral orientado à linguagem.

PONTOS-CHAVE

- Arquitetura de software é uma descrição de como se organiza um sistema de software. As propriedades de um sistema, como desempenho, segurança da informação (*security*) e disponibilidade, são influenciadas pela arquitetura utilizada.
- As decisões de projeto de arquitetura incluem decisões sobre o tipo de aplicação, a distribuição do sistema, os estilos de arquitetura a serem utilizados e os modos como a arquitetura deve ser documentada e avaliada.

- As arquiteturas podem ser documentadas a partir de várias perspectivas ou visões diferentes. Possíveis visões incluem a visão conceitual, a visão lógica, a visão de processo, a visão de desenvolvimento e a visão física.
- Os padrões de arquitetura são um meio de reutilizar o conhecimento a respeito de arquiteturas de sistema genéricas. Eles descrevem a arquitetura, explicam quando ela pode ser utilizada e apontam suas vantagens e desvantagens.
- Os padrões de arquitetura mais utilizados incluem o modelo-visão-controlador, a arquitetura em camadas, o repositório, o cliente-servidor e o duto e filtro.
- Os modelos genéricos de arquiteturas de sistemas de aplicação nos ajudam a compreender a operação das aplicações, comparar aplicações do mesmo tipo, validar projetos de sistema de aplicação e avaliar os componentes de larga escala para reúso.
- Os sistemas de processamento de transações são interativos, permitindo que as informações em um banco de dados sejam acessadas remotamente e modificadas por uma série de usuários. Os sistemas de informação e de gerenciamento de recursos são exemplos de sistemas de processamento de transações.
- Os sistemas de processamento de linguagem são utilizados para traduzir textos de uma linguagem para outra e executar as instruções especificadas na linguagem de entrada. Eles incluem um tradutor e uma máquina abstrata que executa a linguagem gerada.

LEITURAS ADICIONAIS

Software architecture: perspectives on an emerging discipline. Este foi o primeiro livro sobre arquitetura de software e tem uma boa discussão, que ainda é relevante, sobre os diferentes estilos arquiteturais. SHAW, M.; GARLAN, D. Prentice-Hall, 1996.

"The golden age of software architecture". Este artigo pesquisa o desenvolvimento da arquitetura de software desde seu início nos anos 1980 até o seu uso no século XXI. Não é um conteúdo muito técnico, mas sim uma visão geral histórica interessante. SHAW, M.; CLEMENTS, P. *IEEE Software*, v. 21, n. 2, mar./abr. 2006. doi:10.1109/MS.2006.58.

Software architecture in practice. 3. ed. Essa é uma discussão prática das arquiteturas de software, que não promove exageradamente os benefícios do projeto de arquitetura. O texto fornece uma lógica comercial clara, explicando porque as arquiteturas são importantes. BASS, L.; CLEMENTS, P.; KAZMAN, R. Addison-Wesley, 2012.

Handbook of software architecture. Esse é um trabalho em andamento executado por Grady Booch, um dos primeiros evangelistas da arquitetura de software. Ele vem documentando as arquiteturas de uma série de sistemas de software de modo que se possa ver a realidade em vez da abstração acadêmica. Disponível na web e destinado a ser publicado em livro. BOOCH, G. 2014. Disponível em: <http://www.handbookofsoftwarearchitecture.com/>. Acesso em: 21 abr. 2018.

SITE[1]

Apresentações em PowerPoint para este capítulo disponíveis em: <http://software-engineering-book.com/slides/chap6/>.
Links para vídeos de apoio disponíveis em: <http://software-engineering-book.com/videos/requirements-and-design/>.

1 Todo o conteúdo disponibilizado na seção *Site* de todos os capítulos está em inglês.

EXERCÍCIOS

6.1 Explique por que, ao descrever um sistema, talvez você tenha de começar o projeto da arquitetura do sistema antes de concluir a especificação dos requisitos.

6.2 Pediram que você prepare e entregue uma apresentação para um gerente não técnico a fim de justificar a contratação de um arquiteto de sistema para um novo projeto. Escreva uma lista de itens com pontos-chave em sua apresentação e explique a importância da arquitetura de software.

6.3 Explique por que podem surgir conflitos durante o projeto de uma arquitetura para a qual os requisitos de disponibilidade e segurança da informação (*security*) são os requisitos não funcionais mais importantes.

6.4 Desenhe diagramas mostrando uma visão conceitual e uma visão de processo das arquiteturas dos seguintes sistemas:
 - Uma máquina de bilhetes utilizada por passageiros em uma estação ferroviária.
 - Um sistema de videoconferência controlado por computador permitindo que vídeo, áudio e dados de computador fiquem visíveis para vários participantes ao mesmo tempo.
 - Um aspirador robótico destinado a limpar espaços relativamente desobstruídos, como corredores. O aspirador deve ser capaz de sentir as paredes e outros obstáculos.

6.5 Explique por que normalmente você usa vários padrões de arquitetura quando projeta a arquitetura de um sistema grande.

6.6 Sugira uma arquitetura de um sistema (como o iTunes) que é utilizado para vender e distribuir música na internet. Quais padrões de arquitetura são a base para a sua arquitetura proposta?

6.7 Um sistema de informações deve ser desenvolvido para manter as informações sobre ativos de propriedade de uma empresa de serviços públicos, como prédios, veículos e equipamentos. A intenção é que ele seja atualizável pelo pessoal que trabalha em campo usando dispositivos móveis à medida que as informações sobre novos ativos ficarem disponíveis. A empresa tem vários bancos de dados de ativos que devem ser integrados por esse sistema. Projete uma arquitetura em camadas para esse sistema de gerenciamento de ativos baseados na arquitetura genérica de sistemas de informação exibida na Figura 6.18.

6.8 Usando o modelo genérico de um sistema de processamento de linguagem apresentado aqui, projete a arquitetura de um sistema que aceite comandos em linguagem natural e traduz esses comandos em consultas a banco de dados em uma linguagem como SQL.

6.9 Usando o modelo básico de um sistema de informações, conforme apresentado na Figura 6.18, sugira os possíveis componentes de um aplicativo para um dispositivo móvel que exiba informações sobre voos chegando e saindo de um determinado aeroporto.

6.10 Deveria haver uma profissão distinta, a de 'arquiteto de software', cujo papel seria trabalhar independentemente com um cliente para projetar a arquitetura do sistema de software? Outra empresa de software, então, implementaria o sistema. Quais poderiam ser as dificuldades de estabelecer esse tipo de profissão?

REFERÊNCIAS

BASS, L.; CLEMENTS, P.; KAZMAN, R. *Software architecture in practice*. 3. ed. Boston: Addison-Wesley, 2012.

BERCZUK, S. P.; APPLETON, B. *Software configuration management patterns*: effective teamwork, practical integration. Boston: Addison-Wesley, 2002.

BOOCH, G. Handbook of software architecture. 2014. Disponível em: <http://handbookofsoftwarearchitecture.com/>. Acesso em: 21 abr. 2018.

BOSCH, J. *Design and use of software architectures*. Harlow, UK: Addison-Wesley, 2000.

BUSCHMANN, F.; HENNEY, K.; SCHMIDT, D. C. *Pattern-oriented software architecture*: a pattern language for distributed computing. v. 4. New York: John Wiley & Sons, 2007a.

______. *Pattern-oriented software architecture*: on patterns and pattern languages. v. 5. New York: John Wiley & Sons, 2007b.

BUSCHMANN, F.; MEUNIER, R.; ROHNERT, H.; SOMMERLAD, P. *Pattern-oriented software architecture*: a system of patterns. v. 1. New York: John Wiley & Sons, 1996.

CHEN, L.; ALI BABAR, M.; NUSEIBEH, B. "Characterizing architecturally significant requirements." *IEEE Software*, v. 30, n. 2, 2013. p. 38-45. doi:10.1109/MS.2012.174.

COPLIEN, J. O.; HARRISON, N. B. *Organizational patterns of agile software development*. Englewood Cliffs, NJ: Prentice-Hall, 2004.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. *Design patterns*: elements of reusable object-oriented software. Reading, MA: Addison-Wesley, 1995.

GARLAN, D.; SHAW, M. An introduction to software architecture. In: AMBRIOLA, V.; TORTORA, G. *Advances in software engineering and knowledge engineering*, v. 2. London: World Scientific Publishing Co., 1993. p. 1-39.

HOFMEISTER, C.; NORD, R.; SONI, D. *Applied software architecture*. Boston: Addison-Wesley, 2000.

KIRCHER, M.; JAIN, P. *Pattern-oriented software architecture*: patterns for resource management. v. 3. New York: John Wiley & Sons, 2004.

KRUCHTEN, P. The 4+1 view model of software architecture. *IEEE Software*, v. 12, n. 6, 1995. p. 42-50. doi:10.1109/52.469759.

LANGE, C. F. J.; CHAUDRON, M. R. V.; MUSKENS, J. "UML software architecture and design description." *IEEE Software*, v. 23, n. 2, 2006. p. 40-46. doi:10.1109/MS.2006.50.

LEWIS, P. M.; BERNSTEIN, A. J.; KIFER, M. *Databases and transaction processing*: an application-oriented approach. Boston: Addison-Wesley, 2003.

MARTIN, D.; SOMMERVILLE, I. Patterns of cooperative interaction: linking ethnomethodology and design. *ACM transactions on computer-human interaction*, v. 11, n. 1, 1 mar. 2004. p. 59-89. doi:10.1145/972648.972651.

NII, H. P. Blackboard systems, parts 1 and 2. *AI Magazine*, v. 7, n. 1 e 2, 1986. p. 38-53; 62-69. Disponível em: <http://www.aaai.org/ojs/index.php/aimagazine/article/view/537/473>. Acesso em: 21 abr. 2018.

SCHMIDT, D.; STAL, M.; ROHNERT, H.; BUSCHMANN, F. *Pattern-oriented software architecture*: patterns for concurrent and networked objects. v. 2. New York: John Wiley & Sons, 2000.

SHAW, M.; GARLAN, D. *Software architecture*: perspectives on an emerging discipline. Englewood Cliffs, NJ: Prentice-Hall, 1996.

THE USABILITY GROUP. The brighton usability pattern collection. University of Brighton, UK: 1998. Disponível em: <http://www.it.bton.ac.uk/research/patterns/home.html>. Acesso em: 21 abr. 2018.