7 Projeto e implementação

OBJETIVOS

Os objetivos deste capítulo são introduzir o projeto (*design*) de software orientado a objetos usando a UML e destacar as principais preocupações de implementação. Ao ler este capítulo, você:

- compreenderá as atividades mais importantes em um processo geral de projeto orientado a objetos;
- conhecerá alguns dos diferentes modelos que podem ser utilizados para documentar um projeto orientado a objetos;
- entenderá o conceito de padrões de projeto e como eles são uma maneira de reutilizar o conhecimento e a experiência de projeto;
- verá questões-chave que devem ser consideradas durante a implementação do software, incluindo o reúso e o desenvolvimento de software de código aberto (*open source*).

CONTEÚDO

O projeto (*design*) e implementação de software é o estágio no processo de engenharia de software em que é desenvolvido um sistema executável. Para alguns sistemas simples, a engenharia de software se limita a projeto e implementação, e todas as outras atividades de engenharia de software se fundem nesse processo. Entretanto, para sistemas grandes, projeto e implementação de software é apenas um processo dentre tantos outros (engenharia de requisitos, verificação e validação etc.).

As atividades de projeto e implementação são invariavelmente intercaladas. O projeto de software é uma atividade criativa na qual são identificados componentes de software e seus relacionamentos, com base nos requisitos de um cliente. Implementação é o processo de realizar o projeto na forma de um programa. Às vezes, existe uma fase de projeto separada, quando o projeto é modelado e documentado; outras vezes, um projeto está 'na cabeça' do programador ou esboçado em uma lousa

ou folhas de papel. Um projeto pretende solucionar um problema, então sempre há um processo de projeto. No entanto, nem sempre é necessário ou adequado descrevê-lo em detalhes usando a UML ou outra linguagem de descrição de projeto.

O projeto e a implementação estão intimamente relacionados e, normalmente, questões de implementação devem ser levadas em consideração quando se desenvolve um projeto. Por exemplo, usar a UML para documentar um projeto pode ser a coisa certa a fazer se se estiver programando em uma linguagem orientada a objetos, como Java ou C#. No entanto, penso que isso seja menos útil se linguagens dinamicamente tipadas, como Python, estiverem sendo utilizadas. Não faz sentido usar a UML se um sistema estiver sendo implementado pela configuração de um pacote de prateleira. Conforme discuti no Capítulo 3, os métodos ágeis costumam trabalhar a partir de esboços informais do projeto e deixam as decisões mais importantes para os programadores.

Uma das decisões de implementação mais importantes em um estágio inicial de um projeto de software é a de criar ou comprar a aplicação. Hoje, em muitos tipos de aplicação, é possível comprar sistemas de prateleira que podem ser adaptados e personalizados para os requisitos dos usuários. Por exemplo, se quiser implementar um sistema de registros médicos, é possível comprar um pacote já utilizado em hospitais. Normalmente, usar essa abordagem é mais barato e rápido do que desenvolver um novo sistema em uma linguagem de programação convencional.

Quando você desenvolve um sistema de aplicação reusando um produto de prateleira, o processo de projeto se concentra em como configurar o sistema para satisfazer os requisitos da aplicação. Modelos de projeto do sistema, como os modelos dos objetos e suas interações, não são desenvolvidos. Discutirei essa abordagem de desenvolvimento baseado em reúso no Capítulo 15.

Suponho que a maioria dos leitores tenha experiência com projeto e implementação de programas. Isso é algo que se adquire com o aprendizado de programação, dominando elementos de uma determinada linguagem, como Java ou Python. É provável que você tenha aprendido boas práticas de programação nas linguagens estudadas, e também sobre depuração dos programas desenvolvidos. Portanto, não abordarei esses tópicos aqui. Em vez disso, proponho dois objetivos:

1. Mostrar como a modelagem e o projeto de arquitetura do sistema (cobertos nos Capítulos 5 e 6) são postos em prática no desenvolvimento de um projeto de software orientado a objetos.
2. Introduzir questões importantes de implementação que normalmente não são cobertas nos livros de programação. Essas questões incluem reúso, gerenciamento de configuração e desenvolvimento de código aberto.

Como existe uma grande quantidade de plataformas de desenvolvimento diferentes, o capítulo não dá preferência a nenhuma linguagem de programação ou tecnologia de implementação em particular. Portanto, apresentarei todos os exemplos usando UML em vez de uma linguagem de programação, como Java ou Python.

7.1 PROJETO ORIENTADO A OBJETOS USANDO UML

Um sistema orientado a objetos é composto de objetos que interagem para manter seu próprio estado local e fornecer operações nesse estado. A representação

do estado é privada e não pode ser acessada diretamente de fora do objeto. Os processos de projeto orientados a objetos envolvem a criação de classes de objeto e os relacionamentos entre elas. Essas classes definem os objetos no sistema e suas interações. Quando o projeto se realiza como um programa executável, os objetos são criados dinamicamente a partir dessas definições de classe.

Os objetos incluem dados e operações para manipulá-los. Portanto, eles podem ser entendidos e modificados como entidades independentes. Mudar a implementação de um objeto ou acrescentar serviços é algo que não deve afetar outros objetos do sistema. Como os objetos estão associados a coisas, muitas vezes há um mapeamento claro entre as entidades do mundo real (como os componentes de hardware) e os objetos que as controlam no sistema. Isso facilita a compreensão e, portanto, a manutenibilidade do projeto.

Para desenvolver um projeto de sistema a partir do conceito até o projeto detalhado e orientado a objetos, é preciso:

1. compreender e definir o contexto e as interações externas com o sistema;
2. projetar a arquitetura do sistema;
3. identificar os objetos principais no sistema;
4. desenvolver modelos de projeto;
5. especificar interfaces.

Assim como todas as atividades criativas, o projeto não é um processo sequencial óbvio. Ele se desenvolve a partir de ideias, da proposição de soluções e do refinamento destas à medida que mais informações são disponibilizadas. Inevitavelmente, deve-se voltar atrás e tentar novamente quando surgirem problemas. Em alguns casos, as opções são exploradas em detalhes para ver se funcionam; em outros, os detalhes são ignorados até quase o final do processo. Algumas vezes, se usam notações, como a UML, precisamente para esclarecer aspectos do projeto; em outros momentos, elas são utilizadas de modo informal para estimular discussões.

Explico o projeto de software orientado a objetos desenvolvendo um projeto para parte do software embarcado para a estação meteorológica na natureza, que apresentei no Capítulo 1. As estações meteorológicas na natureza são instaladas em áreas remotas. Cada estação registra informações do clima local e as transfere periodicamente para um sistema de informações meteorológicas usando um *link* de satélite.

7.1.1 Contexto do sistema e interações

O primeiro estágio em qualquer processo de projeto de software é desenvolver uma compreensão dos relacionamentos entre o software que está sendo projetado e seu ambiente externo. Isso é essencial para decidir como fornecer a funcionalidade necessária para o sistema e como estruturá-lo para se comunicar com seu ambiente. Conforme discuti no Capítulo 5, compreender o contexto também permite estabelecer os limites do sistema.

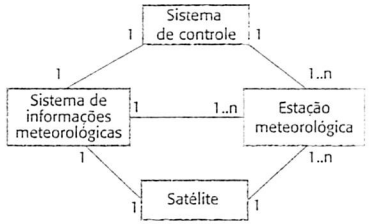
Definir os limites do sistema ajuda a decidir quais características serão implementadas no sistema que está sendo projetado e quais estarão em outros sistemas associados. Nesse caso, é preciso decidir como a funcionalidade será distribuída entre o sistema de controle para todas as estações meteorológicas e o software embarcado na própria estação.

Os modelos de contexto do sistema e os modelos de interação apresentam visões complementares dos relacionamentos entre um sistema e seu ambiente:

1. Um modelo de contexto do sistema é um modelo estrutural que apresenta os outros sistemas no ambiente do sistema que está sendo desenvolvido.
2. Um modelo de interação é um modelo dinâmico que mostra como o sistema interage com o seu ambiente à medida em que é utilizado.

O modelo de contexto de um sistema pode ser representado por meio de associações. Essas associações mostram simplesmente que existem alguns relacionamentos entre as entidades envolvidas. É possível documentar o ambiente do sistema usando um diagrama de blocos simples, que mostra as entidades no sistema e suas associações. A Figura 7.1 mostra que os sistemas no ambiente de cada estação meteorológica são um sistema de informações meteorológicas, um sistema de satélite a bordo e um sistema de controle. A informação de cardinalidade do *link* mostra que existe um único sistema de controle — mas várias estações meteorológicas —, um satélite e um sistema de informações meteorológicas gerais.

FIGURA 7.1 Contexto do sistema para a estação meteorológica.



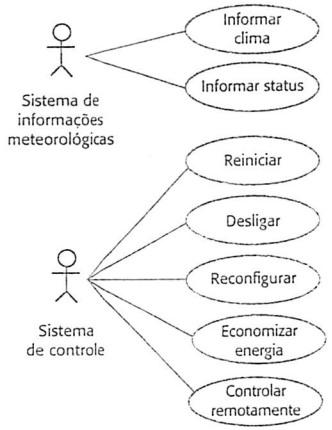
Ao modelar as interações de um sistema com o seu ambiente, deve-se usar uma abordagem abstrata que não inclua detalhes demais. Um modo de fazer isso é usar um modelo de caso de uso. Conforme discuti nos Capítulos 4 e 5, cada caso de uso representa uma interação com o sistema. Cada possível interação é identificada em uma elipse, e a entidade externa envolvida na interação é representada por um boneco palito.

Casos de uso da estação meteorológica

- Informar clima — envia dados climáticos para o sistema de informações meteorológicas.
- Informar status — envia informações de status para o sistema de informações meteorológicas.
- Reiniciar — se a estação meteorológica estiver desligada, reinicia o sistema.
- Desligar — desliga a estação meteorológica.
- Reconfigurar — reconfigura o software da estação meteorológica.
- Economizar energia — coloca a estação meteorológica em modo de economia de energia.
- Controlar remotamente — envia comandos de controle para qualquer subsistema da estação meteorológica.

O modelo de uso de caso da estação meteorológica é apresentado na Figura 7.2. Ele mostra que a estação interage com o sistema de informações meteorológicas para informar dados climáticos e o status do hardware da estação. Outras interações são com um sistema que pode emitir comandos de controle específicos para a estação meteorológica. O boneco palito é utilizado pela UML para representar outros sistemas e também usuários humanos.

FIGURA 7.2 Casos de uso da estação meteorológica.



Cada um desses casos de uso deve ser descrito em linguagem natural estruturada. Isso ajuda os projetistas a identificarem os objetos no sistema e dá a eles uma compreensão do que o sistema se destina a fazer. Para essa descrição, uso um formato padrão que identifica claramente quais são as informações trocadas, como começa a interação etc. Conforme explicarei no Capítulo 21, os sistemas embarcados frequentemente são modelados a partir da descrição de como eles respondem a estímulos internos ou externos. Portanto, os estímulos e as respostas associadas devem ser apresentados na descrição. A Figura 7.3 mostra a descrição do caso de uso 'Informar clima' da Figura 7.2, que se baseia nessa abordagem.

FIGURA 7.3 Descrição de caso de uso — Informar clima.

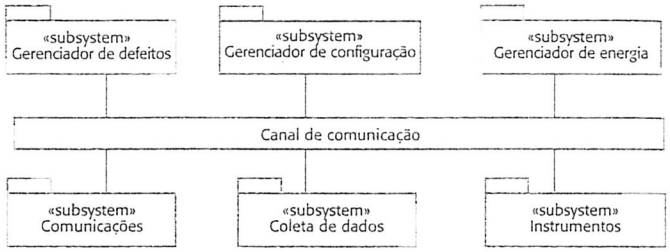
Sistema	Estação meteorológica
Caso de uso	Informar clima
Atores	Sistema de informações meteorológicas, estação meteorológica
Dados	A estação meteorológica envia para o sistema de informações meteorológicas um resumo dos dados que foram coletados pelos instrumentos no período de coleta. Os dados enviados são: a máxima, a mínima e a média de temperaturas do solo e do ar; a máxima, a mínima e a média das pressões atmosféricas; a máxima, a mínima e a média das velocidades do vento; a precipitação total; a direção do vento, conforme amostrado a cada 5 minutos.
Estímulo	O sistema de informações meteorológicas estabelece uma comunicação por satélite com a estação meteorológica e solicita a transmissão dos dados.
Resposta	Os dados resumidos são enviados para o sistema de informações meteorológicas.
Comentários	As estações meteorológicas normalmente são solicitadas a informar uma vez a cada hora, mas essa frequência pode variar de uma estação para outra e pode ser modificada no futuro.

7.1.2 Projeto de arquitetura

Depois de definidas as interações entre o sistema de software e seu ambiente, essas informações podem ser usadas como base para projetar a arquitetura do sistema. Naturalmente, é preciso combinar esse conhecimento com o conhecimento geral dos princípios de projeto da arquitetura e com conhecimentos mais detalhados sobre o domínio de aplicação. Primeiro, os principais componentes do sistema e suas interações são identificados. Depois, a organização do sistema pode ser projetada usando um padrão de arquitetura, como um modelo em camadas ou cliente-servidor.

O projeto de alto nível de arquitetura de software da estação meteorológica é exibido na Figura 7.4. Essa estação é composta de subsistemas independentes que se comunicam transmitindo mensagens em uma infraestrutura comum, exibida como **Canal de comunicação** na Figura 7.4. Cada subsistema escuta as mensagens nessa infraestrutura e seleciona as que são destinadas a ele. Esse 'modelo ouvinte' (*listener*) é um estilo de arquitetura frequentemente utilizado em sistemas distribuídos.

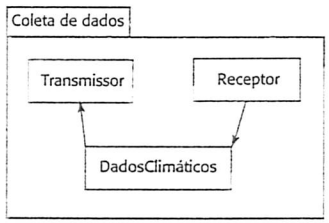
FIGURA 7.4 Arquitetura de alto nível da estação meteorológica.



Quando o subsistema de comunicações recebe um comando de controle, como 'desligar', esse comando é capturado por cada um dos outros subsistemas, que depois se desligam da maneira correta. O benefício fundamental dessa arquitetura é a facilidade de permitir diferentes configurações de subsistemas, pois o emissor da mensagem não precisa endereçá-la para um subsistema em particular.

A Figura 7.5 mostra a arquitetura do subsistema de coleta de dados incluído na Figura 7.4. Os objetos **Transmissor** e **Receptor** estão relacionados ao gerenciamento da comunicação e o objeto **DadosClimáticos** encapsula as informações que são coletadas pelos instrumentos e transmitidas para o sistema de informações meteorológicas. Esse arranjo segue o padrão produtor-consumidor, que será discutido no Capítulo 21.

FIGURA 7.5 Arquitetura do sistema de coleta de dados.



7.1.3 Identificação de classes

Nesse estágio do processo de projeto, já é possível ter algumas ideias a respeito dos objetos essenciais no sistema que está sendo projetado. À medida que aumenta a compreensão do projeto, refinam-se as ideias a respeito dos objetos do sistema. A partir da descrição do caso de uso 'Informar clima', é óbvio que será necessário implementar objetos representando os instrumentos que coletam os dados climáticos e outro objeto que represente o resumo dos dados climáticos. Normalmente, também será necessário um objeto (ou mais) de sistema de alto nível que encapsule as interações do sistema definidas nos casos de uso. Com esses objetos em mente, é possível começar a identificar as classes de objeto mais gerais no sistema.

À medida que o projeto orientado a objetos evoluiu ao longo dos anos 1980, foram sugeridas várias maneiras de identificar classes nos sistemas orientados a objetos, tais como:

1. Usar uma análise gramatical de uma descrição em linguagem natural do sistema a ser construído. Objetos e atributos seriam substantivos; e operações ou serviços, verbos (ABBOTT, 1983).
2. Usar entidades tangíveis (coisas) no domínio da aplicação, como aeronaves; papéis, como o de gerente; eventos, como uma solicitação; interações, como reuniões; locais, como escritórios; unidades organizacionais, como empresas etc. (WIRFS-BROCK; WILKERSON; WEINER, 1990).
3. Usar uma análise baseada em cenário em que são identificados e analisados vários cenários de uso do sistema, um de cada vez. À medida que cada cenário é analisado, a equipe responsável deve identificar objetos, atributos e operações necessários (BECK; CUNNINGHAM, 1989).

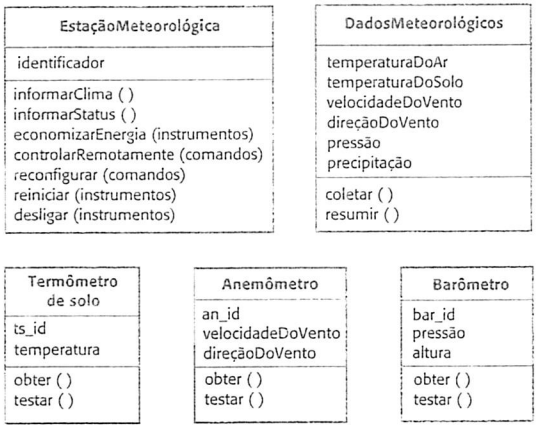
Na prática, deve-se usar várias fontes de conhecimento para descobrir as classes. Elas, os atributos e as operações do objeto — que são identificadas inicialmente a partir da descrição informal do sistema — podem ser o ponto de partida do projeto (*design*). Depois, as informações provenientes do conhecimento do domínio da aplicação e da análise do cenário podem ser usadas para refinar e estender os objetos iniciais. Essas informações podem ser coletadas em documentos de requisitos, em discussões com usuários ou em análises de sistemas existentes. Assim, com os objetos representando entidades externas ao sistema, também pode ser necessário projetar 'objetos de implementação', que são usados para fornecer serviços gerais, como busca e validação de dados.

Na estação meteorológica na natureza, a identificação dos objetos se baseia no hardware tangível do sistema. Não disponho de espaço para incluir aqui todos os objetos do sistema, mas mostro cinco classes na Figura 7.6. Os objetos **Termômetro de solo**, **Anemômetro** e **Barômetro** são do domínio da aplicação, e os objetos **EstaçãoMeteorológica** e **DadosMeteorológicos** foram identificados a partir da descrição do sistema e da descrição do cenário (caso de uso):

1. A classe **EstaçãoMeteorológica** fornece a interface básica da estação com o seu ambiente. Suas operações se baseiam nas interações mostradas na Figura 7.3. Uso uma única classe e ela inclui todas essas interações. Como alternativa, você poderia projetar a interface do sistema como várias classes diferentes, com uma classe por interação.

2. A classe DadosMeteorológicos é responsável por processar o comando Informar clima. Ela envia dados resumidos dos instrumentos da estação meteorológica para o sistema de informações meteorológicas.
3. As classes Termômetro de solo, Anemômetro e Barômetro estão diretamente relacionadas com os instrumentos no sistema. Elas refletem entidades de hardware tangíveis no sistema e as operações são relativas ao controle desse hardware. Esses objetos operam de forma autônoma para coletar dados na frequência especificada e armazenar os dados coletados localmente. Esses dados são fornecidos para o objeto DadosMeteorológicos mediante solicitação.

FIGURA 7.6 Classes da estação meteorológica.



O conhecimento do domínio da aplicação é usado para identificar outros objetos, atributos e serviços:

1. As estações meteorológicas frequentemente são instaladas em lugares remotos e incluem vários instrumentos que, às vezes, não funcionam bem. Em caso de falha, elas devem ser informadas automaticamente. Isso quer dizer que é preciso ter atributos e operações específicos para conferir o funcionamento correto dos instrumentos.
2. Existem muitas estações remotas, então cada estação meteorológica deve ter seu próprio identificador para que possa ser exclusivamente identificada nas comunicações.
3. Como as estações meteorológicas são instaladas em épocas diferentes, os tipos de instrumentos podem ser diferentes. Por essa razão, cada instrumento deve ser identificado unicamente, e um banco de dados de informações dos instrumentos deve ser mantido.

Nesse estágio do processo de projeto, o foco deve estar nos objetos em si, sem pensar sobre como eles poderiam ser implementados. Depois de identificar os objetos, é possível refinar o projeto desses objetos. Dá para procurar características comuns e depois projetar a hierarquia de classes do sistema. Por exemplo, é possível identificar uma superclasse Instrumento, que define as características comuns de todos os instrumentos, como um identificador, e as operações 'obter' e 'testar'. Também dá para acrescentar novos atributos e operações à superclasse, como um atributo que registra a frequência com que os dados devem ser coletados.

7.1.4 Modelos de projeto

Os modelos de projeto ou de sistema, conforme discuti no Capítulo 5, mostram os objetos ou classes de um sistema. Eles também mostram as associações e relações que existem entre essas entidades. Esses modelos são a ponte entre os requisitos do sistema e sua implementação. Eles precisam ser abstratos, para que os detalhes desnecessários não ocultem as relações entre eles e os requisitos do sistema. No entanto, eles também devem incluir detalhes suficientes, para que os programadores tomem decisões de implementação.

O nível de detalhe necessário em um modelo de projeto depende do processo utilizado. Os modelos abstratos podem ser suficientes nas situações em que há vínculos próximos entre engenheiros de requisitos, projetistas e programadores. Decisões de projeto específicas podem ser tomadas à medida que o sistema é implementado; e os problemas, resolvidos por meio de discussões informais. De modo similar, se for utilizado um método de desenvolvimento ágil, os modelos de projeto desenhados em uma lousa podem ser suficientes.

Entretanto, se for utilizado um processo de desenvolvimento dirigido por plano, talvez seja necessário obter modelos mais detalhados. Quando há vínculos distantes entre engenheiros de requisitos, projetistas e programadores (por exemplo, no caso de um sistema projetado em parte de uma organização, mas implementado em outro lugar), então são necessárias descrições de projeto precisas para que haja comunicação. Utilizam-se modelos detalhados, derivados de modelos abstratos de alto nível, para que todos os membros do time tenham a mesma compreensão do projeto.

Portanto, uma etapa importante no processo de projeto é decidir sobre os modelos de projeto necessários e sobre o nível de detalhes que eles devem ter. Isso depende do tipo de sistema que está sendo desenvolvido. Um sistema de processamento de dados sequencial é bem diferente de um sistema de tempo real embarcado, então é preciso usar diferentes tipos de modelos de projeto. A UML permite 13 tipos de modelos diferentes, mas, conforme discuti no Capítulo 5, muitos deles não são amplamente utilizados. Minimizar o número de modelos produzidos reduz os custos do projeto e o tempo necessário para concluí-lo.

Ao usar a UML para desenvolver um projeto, deve-se desenvolver dois tipos de modelo:

1. *Modelos estruturais*, que descrevem a estrutura estática do sistema usando classes e seus relacionamentos. Os relacionamentos importantes que podem ser documentados nesse estágio são os de generalização (herança), os do tipo usa/usado por e os de composição.
2. *Modelos dinâmicos*, que descrevem a estrutura dinâmica do sistema e mostram as interações previstas em tempo de execução entre os objetos do sistema. As interações que podem ser documentadas incluem a sequência de solicitações de serviço feitas pelos objetos e as mudanças de estado desencadeadas por essas interações.

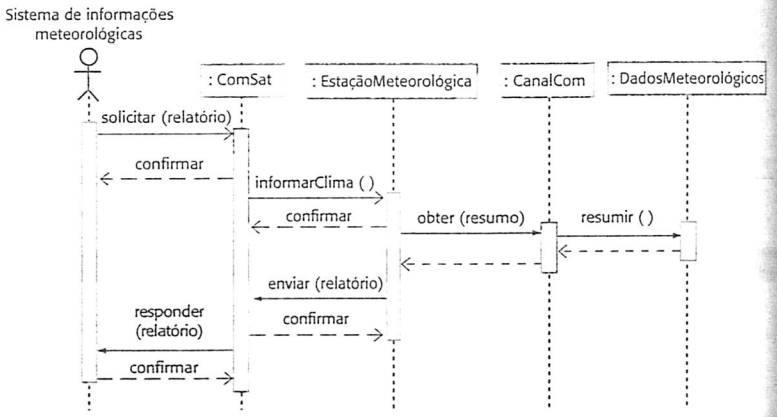
Acredito que três tipos de modelo em UML sejam particularmente úteis para acrescentar detalhes ao caso de uso e aos modelos de arquitetura:

1. *Modelos de subsistema*, que mostram os agrupamentos lógicos dos objetos em subsistemas coerentes. Esses agrupamentos são representados na forma de diagrama de classes, com cada subsistema exibido como um pacote com objetos incluídos. Os modelos de subsistema são modelos estruturais.
2. *Modelos de sequência*, que mostram a sequência de interações entre os objetos. São representados usando os diagramas de sequência ou de colaboração da UML. Os modelos de sequência são dinâmicos.
3. *Modelos de máquinas de estados*, que mostram como os objetos mudam seu estado em resposta aos eventos. Eles são representados em UML usando diagramas de máquina de estados. Os modelos de máquinas de estados são dinâmicos.

Um modelo de subsistema é um modelo estático útil que mostra como um projeto é organizado em grupos de objetos logicamente relacionados. Já mostrei esse tipo de modelo na Figura 7.4, para apresentar os subsistemas no sistema de mapeamento meteorológico. Assim como os modelos de subsistemas, também se pode projetar modelos de objeto detalhados, mostrando os objetos nos sistemas e suas associações (herança, generalização, agregação etc.). No entanto, há um perigo em fazer modelagem excessiva. Não se deve tomar decisões detalhadas a respeito da implementação, pois o melhor é deixar para quando o sistema for implementado.

Os modelos de sequência são dinâmicos e descrevem, para cada modo de interação, a sequência de interações entre objetos. Durante a documentação de um projeto, deve ser produzido um modelo de sequência para cada interação relevante. Caso tenha sido desenvolvido um modelo de caso de uso, então deve haver um modelo de sequência para cada caso de uso identificado.

FIGURA 7.7 Diagrama de sequência descrevendo a coleta de dados.



A Figura 7.7 é um exemplo de modelo de sequência exibido como um diagrama de sequência da UML. Esse diagrama mostra a sequência de interações que ocorrem quando um sistema externo solicita dados resumidos da estação meteorológica. Os diagramas de sequência são lidos de cima para baixo:

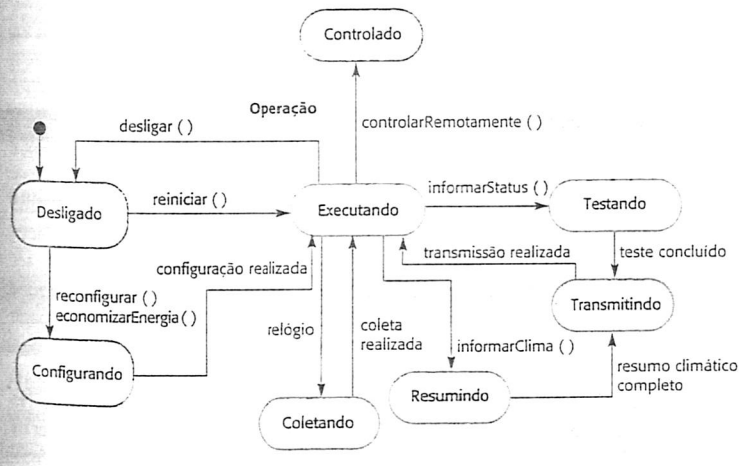
1. O objeto **ComSat** recebe uma solicitação do sistema de informações meteorológicas para coletar um relatório climático de uma estação meteorológica e confirma o recebimento. A seta de ponta aberta na mensagem enviada indica que o sistema externo não espera por uma resposta e já pode executar outro processamento.
2. **ComSat** envia uma mensagem para **EstaçãoMeteorológica**, por meio de um *link* via satélite, para criar um resumo dos dados meteorológicos coletados. Mais uma vez, a seta de ponta aberta indica que **ComSat** não suspende sua execução esperando por uma resposta.
3. **EstaçãoMeteorológica** envia uma mensagem para um objeto **CanalCom** para resumir os dados climáticos. Nesse caso, a seta de ponta fechada indica que a instância da classe **EstaçãoMeteorológica** espera uma resposta.
4. **CanalCom** invoca o método resumir no objeto **DadosMeteorológicos** e espera por uma resposta.
5. O resumo dos dados climáticos é calculado e devolvido à **EstaçãoMeteorológica** por meio do objeto **CanalCom**.
6. Depois, **EstaçãoMeteorológica** chama o objeto **ComSat** para transmitir os dados resumidos para o sistema de informações meteorológicas, por meio do sistema de comunicação por satélite.

Os objetos **ComSat** e **EstaçãoMeteorológica** podem ser implementados como processo simultâneos, cuja execução pode ser suspensa ou retomada. A instância do objeto **ComSat** escuta as mensagens do sistema externo, decodifica essas mensagens e inicia as operações da estação meteorológica.

Os diagramas de sequência são utilizados para modelar o comportamento combinado de um grupo de objetos, mas também é possível resumir o comportamento de um objeto ou subsistema em resposta às mensagens e eventos. Para isso, pode-se usar um modelo de máquina de estados que mostre como o objeto muda de estado, dependendo das mensagens que receber. Conforme discuti no Capítulo 5, a UML inclui diagramas de máquina de estados para descrever tais modelos.

A Figura 7.8 é um diagrama de máquina de estados do sistema da estação meteorológica que mostra como ele responde às solicitações de vários serviços.

FIGURA 7.8 Diagrama de máquina de estados da estação meteorológica.



É possível ler esse diagrama da seguinte maneira:

1. Se o estado do sistema for **Desligado**, então ele pode responder com uma mensagem de **reiniciar ( )**, de **reconfigurar ( )** ou de **economizarEnergia ( )**. A seta sem rótulo com uma bola preta indica que o estado **Desligado** é o estado inicial. Uma mensagem **reiniciar ( )** provoca uma transição para a operação normal. Tanto a mensagem **economizarEnergia ( )** quanto a **reconfigurar ( )** provocam uma transição para um estado no qual o sistema se reconfigura. O diagrama de estado mostra que a reconfiguração é permitida somente se o sistema for desligado.
2. No estado **Executando**, o sistema espera outras mensagens. Se for recebida uma mensagem **desligar ( )**, o objeto retorna para o estado **Desligado**.
3. Se for recebida uma mensagem **informarClima ( )**, o sistema passa para um estado **Resumindo**. Quando o resumo é terminado, o sistema passa para o estado **Transmitindo**, em que as informações são transmitidas para o sistema remoto. Depois ele retorna para o estado **Executando**.
4. Se for recebido um sinal do relógio, o sistema passa para o estado **Coletando**, no qual coleta dados dos instrumentos. Cada instrumento é instruído, por sua vez, a coletar seus dados dos sensores associados.
5. Se for recebida uma mensagem **controlarRemotamente ( )**, o sistema passa para um estado controlado, no qual responde a um conjunto diferente de mensagens da sala de controle remoto. Essas mensagens não são exibidas nesse diagrama.

Os diagramas de máquina de estados são modelos de alto nível úteis que retratam a operação de um sistema ou de um objeto. Entretanto, não há necessidade de um diagrama de máquina de estados para todos os objetos no sistema. Muitos desses objetos são simples e sua operação pode ser descrita facilmente sem um modelo de máquina de estados.

7.1.5 Especificação de interface

Uma parte importante de qualquer processo de projeto é a especificação das interfaces entre os componentes. É preciso especificar as interfaces para que os objetos e os subsistemas possam ser projetados em paralelo. Depois que uma interface foi especificada, os desenvolvedores dos outros objetos podem assumir que a interface será implementada.

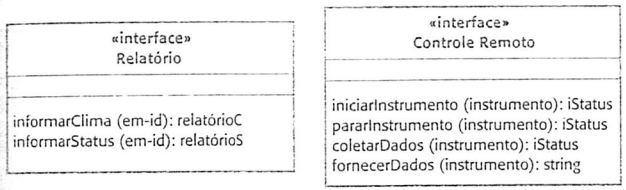
O projeto de interface se preocupa com a especificação dos detalhes da interface para um objeto ou grupo de objetos. Isso significa definir as assinaturas e a semântica dos serviços que são fornecidos pelo objeto ou por um grupo de objetos. As interfaces podem ser especificadas em UML usando a mesma notação de um diagrama de classes. No entanto, não há uma seção de atributos, e o estereótipo «interface» da UML deve ser incluído na parte do nome. A semântica da interface pode ser definida usando a linguagem de restrição de objetos (OCL, do inglês *object constraint language*). Discutirei o uso da OCL no Capítulo 16, no qual explicarei como ela pode ser utilizada para descrever a semântica dos componentes.

Os detalhes da representação de dados não devem ser incluídos em um projeto de interface, pois os atributos não são definidos em uma especificação da interface. No entanto, as operações devem ser incluídas para que seja possível acessar e atualizar os dados. Como a representação dos dados fica escondida, ela pode ser modificada facilmente sem afetar os objetos que usam esses dados. Isso leva a um projeto inerentemente mais manutenível. Por exemplo, uma representação em vetor de uma pilha pode ser trocada para uma lista sem afetar os objetos que usam essa pilha. Por outro lado, os atributos devem ser expostos normalmente em um modelo de objeto, pois essa é a maneira mais clara de descrever as características essenciais dos objetos.

Não existe uma relação 1:1 simples entre objetos e interfaces. O mesmo objeto pode ter várias interfaces, cada uma delas sendo um ponto de vista sobre os métodos que ela fornece. Isso tem suporte direto em Java, em que as interfaces são declaradas separadamente dos objetos, e esses objetos 'implementam' as interfaces. Do mesmo modo, um grupo de objetos pode ser acessado por meio de uma interface única.

A Figura 7.9 mostra duas interfaces que podem ser definidas para a estação meteorológica. A interface da esquerda é de relatório e define os nomes das operações que são utilizadas para gerar relatórios de clima e status. Elas correspondem diretamente às operações no objeto **EstaçãoMeteorológica**. A interface 'Controle Remoto' fornece quatro operações, que correspondem a um único método no objeto **EstaçãoMeteorológica**. Nesse caso, cada operação é codificada na *string* de comando associada ao método **controlarRemotamente**, exibido na Figura 7.6.

FIGURA 7.9 Interfaces da estação meteorológica.



7.2 PADRÕES DE PROJETO

Os padrões de projeto (*design patterns*) derivaram de ideias apresentadas por Christopher Alexander (1979), que sugeriu a existência de certos padrões comuns em projetos de prédios que eram inerentemente agradáveis e eficazes. O padrão é uma descrição do problema e a essência de sua solução, de modo que ela possa ser reutilizada em diferentes contextos. O padrão não é uma especificação detalhada. Em vez disso, é possível encará-lo como uma descrição da sabedoria e do conhecimento acumulados, uma solução testada e aprovada para um problema comum.

Uma citação proveniente do site do Hillside Group (hillside.net/patterns), dedicada a manter informações sobre os padrões, encapsula seu papel no reúso:

> *Padrões e Linguagens de Padrões são maneiras de descrever as melhores práticas, os bons projetos e capturar a experiência de uma maneira que viabilize a reutilização dessas experiências por outras pessoas.*[1]

Os padrões causaram um enorme impacto no projeto de software orientado a objetos. Além de serem soluções testadas para problemas comuns, eles se transformaram em um vocabulário para falar sobre projeto. Portanto, é possível explicar um projeto

1 The Hillside Group. Patterns. Site, 1994-2018. Disponível em: <hillside.net/patterns/>. Acesso em: 26 abr. 2018.

descrevendo os padrões utilizados nele. Isso vale particularmente para os padrões de projeto mais conhecidos, que foram descritos originalmente pela 'Gangue dos Quatro'[2] em seu livro sobre padrões, publicado em 1995 (GAMMA *et al.*, 1995). Outras descrições de padrões importantes são as publicadas em uma série de livros de autores da Siemens, uma grande empresa europeia do setor de tecnologia (BUSCHMANN *et al.*, 1996; SCHMIDT *et al.*, 2000; KIRCHER; JAIN, 2004; BUSCHMANN; HENNEY; SCHMIDT, 2007a, 2007b).

Os padrões são uma maneira de reutilizar o conhecimento e a experiência de outros projetistas. Os padrões de projeto normalmente estão associados ao projeto orientado a objetos. Os que foram publicados se baseiam quase sempre nas características de objetos, como herança e polimorfismo, para promover a generalidade. No entanto, o princípio geral de encapsular a experiência em um padrão é igualmente aplicável a qualquer tipo de projeto de software. Por exemplo, é possível ter padrões de configuração para instanciar sistemas de aplicação reusáveis.

A Gangue dos Quatro definiu em seu livro os quatro elementos essenciais dos padrões de projeto:

1. Um nome que seja uma referência significativa para o padrão.
2. Uma descrição da área do problema que explique quando o padrão pode ser aplicado.
3. Uma descrição das partes da solução de projeto, suas relações e responsabilidades. Isso não é uma descrição do projeto concreto. É um *template* para uma solução de projeto que pode ser instanciado de diferentes maneiras. Muitas vezes, se expressa graficamente e mostra os relacionamentos entre os objetos e suas classes na solução.
4. Uma declaração das consequências — os resultados e os *trade-offs* — de aplicar o padrão. Isso pode ajudar os projetistas a compreenderem se um padrão pode ou não ser utilizado em uma determinada situação.

Gamma e seus coautores decompõem a descrição do problema em motivação (uma descrição explicando porque o padrão é útil) e aplicabilidade (uma descrição das situações em que o padrão pode ser utilizado). Sob a descrição da solução, eles descrevem a estrutura do padrão, os participantes, as colaborações e a implementação.

FIGURA 7.10 O padrão Observer.

Nome do padrão:	Observer
Descrição:	Separa a exibição do estado de um objeto do próprio objeto, permitindo exibições alternativas. Quando o estado do objeto muda, todas as exibições são automaticamente notificadas e atualizadas para refletir a mudança.
Descrição do problema:	Em muitas situações, deve-se proporcionar múltiplas exibições da informação do estado, como uma visualização gráfica ou uma exibição em tabela. Nem todas podem ser conhecidas quando a informação é especificada. Todas as apresentações alternativas devem permitir interação e, quando o estado muda, todas as exibições devem ser atualizadas. Esse padrão pode ser utilizado tanto nas situações em que é necessário mais de um formato de exibição para a informação de estado, quanto naquelas em que não é necessário que o objeto mantenedor da informação de estado conheça os formatos de exibição específicos utilizados.

continua

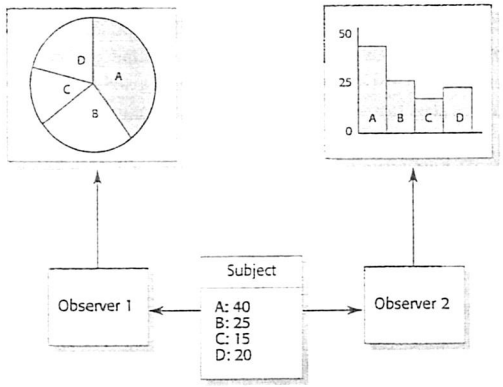
2 N. da T.: refere-se aos quatro autores do livro: Erich Gamma, John Vlissides, Ralph Johnson e Richard Helm.

continuação

Descrição da solução:	Isso envolve dois objetos abstratos, o Subject (sujeito) e o Observer (observador), e dois objetos concretos, ConcreteSubject (sujeito concreto) e ConcreteObserver (observador concreto), que herdam os atributos dos objetos abstratos relacionados. Os objetos abstratos incluem operações gerais, que são aplicáveis em todas as situações. O estado a ser exibido é mantido no ConcreteSubject, que herda operações do Subject, permitindo que ele inclua (*attach*) ou exclua (*detach*) Observers (cada observador corresponde a uma exibição) e emita uma notificação (*notify*) quando o estado mudar. O ConcreteObserver mantém uma cópia do estado do ConcreteSubject e implementa a interface update () do Observer que permite que essas cópias sejam mantidas harmonizadas, atualizando-as. O ConcreteObserver exibe automaticamente o estado e reflete as mudanças sempre que esse estado for atualizado. O modelo em UML do padrão é exibido na Figura 7.12.
Consequências:	O sujeito só conhece o Observer abstrato e não conhece detalhes da classe concreta. Portanto, há uma associação mínima entre esses objetos. Em razão da falta de conhecimento, as otimizações que melhoram o desempenho da exibição são impraticáveis. As mudanças no sujeito podem causar a geração de um conjunto de atualizações encadeadas nos observadores, algumas delas podendo ser desnecessárias.

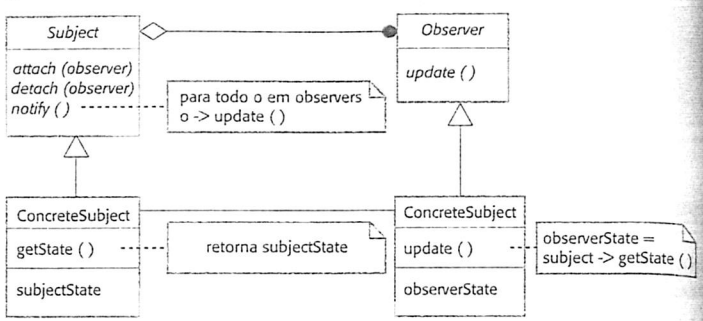
Para ilustrar a descrição do padrão, uso o padrão Observer (observador, em português), extraído do livro de padrões da Gangue dos Quatro e exibido na Figura 7.10. Em minha descrição, eu uso os quatro elementos essenciais da descrição e também incluo uma breve declaração do que o padrão pode fazer. Esse padrão pode ser utilizado em situações nas quais são exigidas diferentes apresentações de um estado do objeto. Ele separa o objeto que deve ser exibido das diferentes formas de apresentação. Isso é ilustrado na Figura 7.11, que mostra duas representações gráficas diferentes do mesmo conjunto de dados.

FIGURA 7.11 Múltiplas exibições.



As representações gráficas normalmente são utilizadas para ilustrar as classes envolvidas nos padrões e seus relacionamentos. Elas suplementam a descrição do padrão e acrescentam detalhes à descrição da solução. A Figura 7.12 é a representação em UML do padrão Observer.

FIGURA 7.12 Modelo em UML do padrão Observer.



Para usar padrões no projeto, é preciso reconhecer que, para um eventual problema enfrentado, pode haver um padrão associado que pode ser aplicado. Exemplos desses problemas, documentados no livro de padrões original da Gangue dos Quatro, incluem:

1. dizer a vários objetos que o estado de algum outro objeto mudou (padrão Observer);
2. arrumar as interfaces para uma série de objetos relacionados, que muitas vezes foram desenvolvidas de modo incremental (padrão Façade);
3. proporcionar uma maneira padrão de acessar os elementos em um conjunto, independentemente de como ele foi implementado (padrão Iterator);
4. permitir a possibilidade de estender a funcionalidade de uma classe existente em tempo de execução (padrão Decorator).

Os padrões apoiam o reúso do conceito em alto nível. Ao tentar reusar componentes executáveis, inevitavelmente se fica restrito às decisões detalhadas do projeto que foram tomadas pelos implementadores desses componentes. Essas limitações variam desde os algoritmos específicos que foram utilizados para implementar os componentes até os objetos e tipos em suas interfaces. Quando as decisões de projeto conflitam com os seus requisitos, o reúso do componente é impossível ou introduz ineficiências no seu sistema. Usar padrões significa reaproveitar as ideias e adaptar a implementação para adequá-la ao sistema que está sendo desenvolvido.

Quando se começa a projetar um sistema, pode ser difícil saber, de antemão, se há necessidade de um padrão particular. Portanto, usar padrões em um processo de projeto envolve desenvolver um projeto, vivenciar um problema e, então, reconhecer que um padrão pode ser utilizado. Certamente, isso é possível se focamos nos 23 padrões de propósito geral documentados no livro de padrões original. No entanto, se o problema for diferente, pode ser difícil encontrar algo apropriado entre as centenas de diferentes padrões que foram propostos.

Os padrões são uma ótima ideia, mas é preciso ter experiência em projeto de software para utilizá-los com eficácia, pois é necessário reconhecer as situações em que um padrão pode ser aplicado. Os programadores inexperientes, ainda que tenham lido livros sobre padrões, sempre acharão difícil decidir se eles podem reusar um padrão ou se precisam desenvolver uma solução específica.

7.3 QUESTÕES DE IMPLEMENTAÇÃO

A engenharia de software inclui todas as atividades envolvidas no desenvolvimento de software, desde os requisitos iniciais do sistema até a manutenção e o gerenciamento do sistema implantado. Evidentemente, uma etapa crítica desse processo é a implementação do sistema, em que se cria uma versão executável do software. A implementação pode tanto envolver o desenvolvimento de programas em linguagens de programação de alto ou baixo nível como adequar e adaptar sistemas genéricos de prateleira a fim de satisfazer requisitos específicos de uma organização.

Suponho que a maioria dos leitores entenda os princípios de programação e tenha alguma experiência no assunto. Como este capítulo se destina a oferecer uma abordagem independentemente da linguagem, não me concentrei nas questões das boas práticas de programação, pois isso exigiria a utilização de exemplos em linguagens específicas. Em vez disso, introduzi alguns aspectos da implementação que são particularmente importantes para a engenharia de software e que, muitas vezes, não são abordados nos textos sobre programação. São eles:

1. *Reúso.* A maior parte do software moderno é construída reusando componentes ou sistemas que já existem. Ao longo do desenvolvimento de software, deve-se utilizar o código existente o máximo possível.
2. *Gerenciamento de configuração.* Durante o processo de desenvolvimento, são criadas muitas versões diferentes de cada componente do software. Se elas não forem controladas por um sistema de gerenciamento de configuração, corre-se o risco de incluir as versões erradas desses componentes no sistema.
3. *Desenvolvimento host-target.* O software de produção normalmente não é executado no mesmo computador do ambiente de desenvolvimento de software. Em vez disso, você o desenvolve em um computador (*host*) e o executa em outro (*target*, ou sistema-alvo). Os sistemas *host* e *target* às vezes são do mesmo tipo, mas, na maioria das vezes, são completamente diferentes.

7.3.1 Reúso

Dos anos 1960 até os anos 1990, a maior parte do novo software foi desenvolvida do zero, com todo o código escrito em uma linguagem de programação de alto nível. O único reúso significativo de software era o aproveitamento das funções e objetos nas bibliotecas de linguagem de programação. No entanto, o custo e a pressão do cronograma significaram que essa abordagem se tornou cada vez mais inviável, especialmente para sistemas comerciais e baseados na internet. Por essa razão, uma abordagem para o desenvolvimento baseada no reúso de software existente passou a ser a norma usada para muitos tipos de desenvolvimento de sistemas. Uma abordagem baseada em reúso é empregada atualmente nos sistemas web de todos os tipos, no software científico e, cada vez mais, na engenharia de sistemas embarcados.

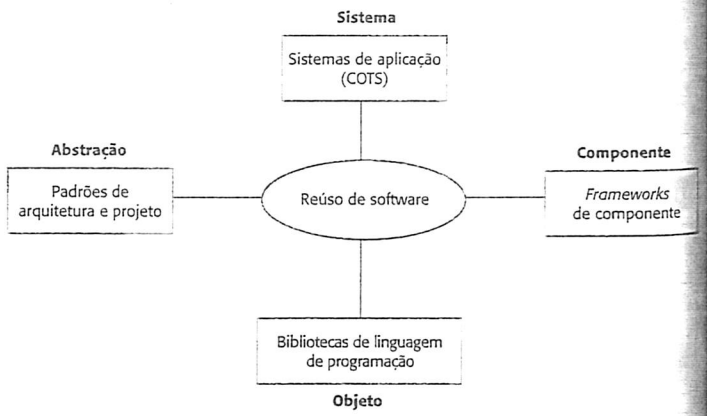
O reúso de software é possível em uma série de níveis diferentes, como mostra a Figura 7.13:

1. *No nível de abstração.* Nesse nível, o software não é reusado diretamente, mas, em vez disso, é usado o conhecimento das abstrações bem-sucedidas no projeto de

software. Os padrões de projeto e os padrões de arquitetura (cobertos no Capítulo 6) são maneiras de representar o conhecimento abstrato para reúso.

2. *No nível de objeto*. Nesse nível, os objetos de uma biblioteca são reusados diretamente em vez de o código ser escrito. Para implementar esse tipo de reúso, deve-se encontrar as bibliotecas apropriadas e descobrir se os objetos e métodos oferecem a funcionalidade necessária. Por exemplo, se é preciso processar mensagens de e-mail em um programa Java, podem ser utilizados objetos e métodos de uma biblioteca JavaMail.
3. *No nível de componente*. Os componentes são conjuntos de objetos e classes que operam juntos para fornecer funções e serviços relacionados. Muitas vezes, deve-se adaptar e estender o componente, adicionando algum código por conta própria. Um exemplo de reúso no nível do componente é a criação de uma interface com o usuário usando um *framework*. Trata-se de um conjunto de classes genéricas que implementam o tratamento de eventos, gerenciamento de exibição etc. Conexões são acrescentadas aos dados a serem exibidos e o código é escrito para definir detalhes específicos da exibição, como o *layout* e as cores da tela.
4. *No nível de sistema*. Nesse nível, são reusados sistemas de aplicação inteiros. Essa função, geralmente, envolve algum tipo de configuração desses sistemas. Isso pode ser feito ao adicionar e ao modificar código (caso uma linha de produtos de software esteja sendo reusada) ou ao usar a interface de configuração do próprio sistema. Atualmente, a maioria dos sistemas comerciais é criada dessa maneira, com sistemas de aplicação genéricos sendo adaptados e reusados. Às vezes, essa abordagem pode envolver a integração de vários sistemas de aplicação para criar um novo.

FIGURA 7.13 Reúso de software.



Reusando o software existente, é possível desenvolver novos sistemas mais rapidamente, com menos riscos de desenvolvimento e a um custo mais baixo. Como o software reusado foi testado em outras aplicações, ele deve ser mais confiável do que o novo software. Entretanto, existem custos associados ao reúso:

1. Os custos do tempo gasto na busca pelo software para reúso e na avaliação do nível de satisfação das suas necessidades por este software. Você pode ter de testar o software para certificar-se de que ele vai funcionar no seu ambiente, especialmente se for diferente do ambiente em que ele foi desenvolvido.
2. Os custos de comprar o software reusável, quando for aplicável. Nos grandes sistemas de prateleira, esse custo pode ser muito alto.
3. Os custos de adaptar e configurar os componentes do software ou do sistema reusável para refletir os requisitos do sistema que está sendo desenvolvido.
4. Os custos de integrar os elementos do software reusável com cada um dos demais elementos (caso esteja sendo usado software de fontes diferentes) e com o novo código desenvolvido. Pode ser difícil e caro integrar software reusável de diferentes fornecedores porque eles podem ter pressupostos conflitantes sobre como seu respectivo software será reusado.

A primeira coisa em que se deve pensar ao começar um projeto de desenvolvimento de software é como reusar o conhecimento e o software existentes. É preciso considerar a possibilidade de reúso antes de projetar o software em detalhes, já que pode ser necessário adaptar o projeto para reusar recursos de software existentes. Conforme discuti no Capítulo 2, em um processo de desenvolvimento orientado para reúso, elementos reusáveis são buscados e, então, seus requisitos e seu projeto são modificados para fazer o melhor uso desses elementos.

Em virtude da importância do reúso na engenharia de software moderna, dedicarei vários capítulos na Parte 3 deste livro exatamente a esse tópico (Capítulos 15, 16 e 18).

7.3.2 Gerenciamento de configuração

Em desenvolvimento de software, mudanças acontecem o tempo todo, então o gerenciamento da mudança é absolutamente essencial. Quando várias pessoas estão envolvidas no desenvolvimento de um sistema de software, é necessário assegurar que os membros do time não interfiram no trabalho uns dos outros. Em outras palavras, se duas pessoas estão trabalhando em um componente, suas mudanças precisam ser coordenadas. Senão, um programador pode fazer alterações e sobrescrever o trabalho do outro. Também é importante garantir que todos possam acessar as versões mais atualizadas dos componentes de software; senão, os desenvolvedores podem refazer o trabalho que já foi feito. Quando algo sai errado com uma nova versão de um sistema, deve ser possível voltar para uma versão funcional do sistema ou componente.

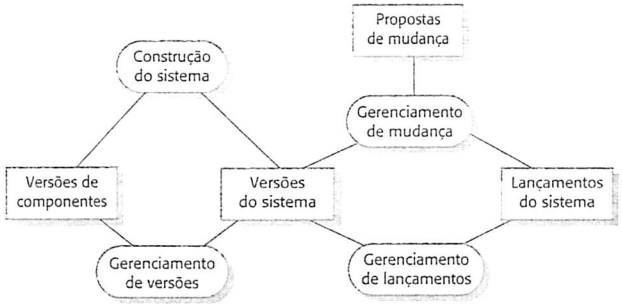
Gerenciamento de configuração é o nome dado ao processo geral de gerenciamento de um sistema de software em mudança. O objetivo do gerenciamento de configuração é o de apoiar o processo de integração do sistema para que todos os desenvolvedores possam acessar o código do projeto e seus documentos de maneira controlada, descobrir quais mudanças foram feitas, compilar e ligar os componentes para criar um sistema. Conforme a Figura 7.14, existem quatro atividades fundamentais no gerenciamento de configuração:

1. *Gerenciamento de versão*, em que o apoio é fornecido para controlar as diferentes versões dos componentes de software. Os sistemas de gerenciamento de versão incluem recursos para coordenar o desenvolvimento por vários programadores.

Eles impedem que um desenvolvedor sobrescreva o código que foi submetido para o sistema por outro desenvolvedor.

2. *Integração do sistema*, em que o apoio é fornecido para ajudar os desenvolvedores a definirem quais versões dos componentes são utilizadas para criar cada versão de um sistema. Essa descrição é utilizada para criar um sistema automaticamente, compilando e ligando os componentes necessários.
3. *Rastreamento de problemas*, em que o apoio é fornecido para permitir aos usuários relatarem defeitos e outros problemas, e permitir a todos os desenvolvedores ver quem está trabalhando nesses problemas e quando são consertados.
4. *Gerenciamento de lançamento* (*release*), em que novas versões do software são liberadas para os clientes. Essa tarefa está relacionada com o planejamento da funcionalidade dos novos lançamentos e com a organização do software para distribuição.

FIGURA 7.14 Gerenciamento de configuração.



As ferramentas de gerenciamento de configuração de software apoiam cada uma das atividades acima. Essas ferramentas geralmente são instaladas em um ambiente de desenvolvimento integrado, como o Eclipse. O gerenciamento de versões pode ser apoiado usando um sistema de gerenciamento de versões, como o Subversion (PILATO; COLLINS-SUSSMAN; FITZPATRICK, 2008) ou o Git (LOELIGER; MCCULLOUGH, 2012), que pode apoiar o desenvolvimento em múltiplos locais por múltiplos times. O apoio à integração do sistema pode ser embutido na linguagem ou se basear em um conjunto de ferramentas à parte, como o GNU *build system*. O rastreamento de defeitos ou os sistemas de acompanhamento de problemas, como o Bugzilla, são utilizados para informar defeitos e outros problemas e monitorar se foram ou não consertados. Um conjunto abrangente de ferramentas construído em torno do sistema Git está disponível no Github (github.com).

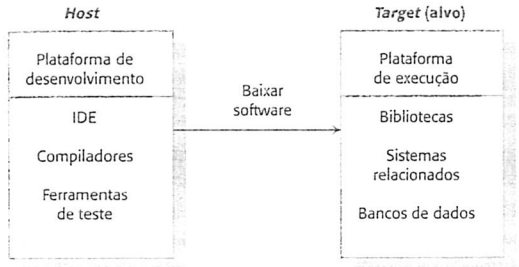
Em razão de sua importância na engenharia de software profissional, discutirei o gerenciamento de mudança e de configuração com mais detalhes no Capítulo 25.

7.3.3 Desenvolvimento *host-target*

A maior parte do desenvolvimento de software profissional se baseia no modelo *host-target* (Figura 7.15). O software é desenvolvido em um computador (o *host*), mas

executa em outra máquina (o *target*, ou alvo). Em termos mais gerais, podemos falar sobre uma plataforma de desenvolvimento (*host*) e uma plataforma de execução (*target*). A plataforma vai além do hardware. Ela inclui o sistema operacional instalado e outros softwares de apoio, como um sistema de gerenciamento de banco de dados ou, nas plataformas de desenvolvimento, um ambiente de desenvolvimento interativo.

FIGURA 7.15 Desenvolvimento *host-target*.



Às vezes, a plataforma de desenvolvimento e a plataforma de execução são a mesma, possibilitando o desenvolvimento do software e seu teste na mesma máquina. Portanto, se você desenvolve em Java, o ambiente-alvo é a Máquina Virtual Java. A princípio, ela é a mesma em todos os computadores, então os programas devem ser portáveis de uma máquina para outra. No entanto, particularmente nos sistemas embarcados e nos sistemas móveis, as plataformas de desenvolvimento e execução são diferentes. É preciso passar o software desenvolvido à plataforma de execução, seja para testar ou para executar um simulador em sua máquina de desenvolvimento.

Os simuladores são frequentemente utilizados durante o desenvolvimento dos sistemas embarcados. São simulados os dispositivos de hardware, como sensores, e os eventos no ambiente em que o sistema será implantado. Os simuladores aceleram o processo de desenvolvimento dos sistemas embarcados, já que cada desenvolvedor pode ter a sua própria plataforma de execução sem necessidade de baixar o software para o hardware-alvo. Porém, os simuladores têm um desenvolvimento caro e, portanto, geralmente só estão disponíveis para as arquiteturas de hardware mais populares.

Se o sistema-alvo tiver middleware ou outro software instalado que precise ser utilizado, então deve ser necessário testar o sistema usando esse software. Pode ser impraticável instalar o software em uma máquina de desenvolvimento, mesmo que seja a mesma da plataforma-alvo, em razão das restrições de licença. Se for o caso, será preciso transferir o código desenvolvido para a plataforma de execução para testar o sistema.

Uma plataforma de desenvolvimento de software deve fornecer uma gama de ferramentas para apoiar os processos de engenharia de software. Essas ferramentas podem incluir:

1. um compilador integrado e um sistema de edição dirigido pela sintaxe que permitam criar, editar e compilar o código;
2. um sistema de depuração da linguagem;
3. ferramentas gráficas de edição, como as ferramentas para editar modelos UML;

4. ferramentas de teste, como JUnit, que podem executar automaticamente um conjunto de testes em uma nova versão de um programa;
5. ferramentas para apoiar a refatoração e a visualização do programa;
6. ferramentas de gerenciamento de configuração para gerenciar versões do código-fonte e para integrar e construir os sistemas.

Além dessas ferramentas básicas, o sistema de desenvolvimento pode incluir ferramentas mais especializadas, como os analisadores estáticos (discutidos no Capítulo 12). Normalmente, os ambientes de desenvolvimento para times também incluem um servidor compartilhado que executa um sistema de gerenciamento de mudança e configuração e, talvez, um sistema para apoiar o gerenciamento de requisitos.

Hoje, as ferramentas de desenvolvimento de software estão instaladas em um ambiente de desenvolvimento integrado (IDE, do inglês *integrated development environment*). Um IDE é um conjunto de ferramentas de software que apoia diferentes aspectos do desenvolvimento de software dentro de algum *framework* e de uma interface de usuário comuns. Geralmente, os IDEs são criados para apoiar o desenvolvimento em uma linguagem de programação específica, como Java. O IDE da linguagem pode ser desenvolvido ou pode ser uma instanciação de um IDE de uso geral, com ferramentas de apoio específicas para a linguagem.

Um IDE de uso geral é um *framework* para abrigar ferramentas de software que proporcionam recursos de gerenciamento de dados para o software que está sendo desenvolvido e mecanismos de integração que permitem que as ferramentas trabalhem juntas. O IDE de propósito geral mais conhecido é o ambiente Eclipse (www.eclipse.org), que se baseia em uma arquitetura de *plug-in* para que possa ser especializado em diferentes linguagens, como Java, e domínios de aplicação. Portanto, é possível instalar o Eclipse e adaptá-lo a necessidades específicas adicionando *plug-ins*. Por exemplo, dá para adicionar um conjunto de *plug-ins* para permitir o desenvolvimento de sistemas em rede na linguagem Java (VOGEL, 2013) ou engenharia de sistemas embarcados usando a linguagem C.

Como parte do processo de desenvolvimento, é preciso tomar decisões sobre como o software desenvolvido será implantado na plataforma-alvo. Isso é simples nos sistemas embarcados, em que o alvo normalmente é um único computador. No entanto, nos sistemas distribuídos é preciso decidir sobre as plataformas específicas nas quais os componentes serão implantados. As questões que devem ser consideradas na tomada de decisão são:

1. *Os requisitos de hardware e software de um componente.* Se um componente for projetado para uma arquitetura de hardware específica ou depender de outro sistema de software, então, obviamente, ele deve ser implantado em uma plataforma que tenha o apoio necessário.
2. *Os requisitos de disponibilidade do sistema.* Sistemas de alta disponibilidade podem exigir que os componentes sejam implantados em mais de uma plataforma. Isso significa que, no caso de uma falha da plataforma, uma implementação alternativa do componente está disponível.
3. *Comunicação dos componentes.* Se houver muita comunicação entre os componentes, normalmente é melhor implantá-los na mesma plataforma ou em plataformas que sejam fisicamente próximas umas das outras. Isso reduz a latência da comunicação — o atraso entre o momento em que uma mensagem é enviada por um componente e recebida por outro.

É possível documentar as decisões sobre implantação de hardware e software usando diagramas de implantação da UML, que mostram como os componentes de software estão distribuídos entre as plataformas de hardware.

Caso esteja desenvolvendo um sistema embarcado, pode ser necessário levar em consideração as características-alvo, como o tamanho físico, a potência, a necessidade de respostas de tempo real a eventos de sensor, as características físicas de atuadores e de seu sistema operacional de tempo real. Discutirei a engenharia de sistemas embarcados no Capítulo 21.

Diagramas de implantação da UML

Os diagramas de implantação da UML mostram como os componentes de software são implantados fisicamente nos processadores. Ou seja, o diagrama de implantação mostra o hardware e o software no sistema e o middleware utilizado para conectar os diferentes componentes no sistema. Basicamente, você pode pensar nos diagramas de implantação como uma maneira de definir e documentar o ambiente-alvo.

7.4 DESENVOLVIMENTO DE CÓDIGO ABERTO (*OPEN SOURCE*)

O desenvolvimento de código aberto (*open source*) é uma abordagem para o desenvolvimento de software na qual o código-fonte de um sistema de software é publicado e voluntários são convidados para participar do processo de desenvolvimento (RAYMOND, 2001). Suas bases estão na Free Software Foundation (www.fsf.org), que defende que o código-fonte não deve ser proprietário, mas, sim, disponibilizado para que os usuários o examinem e modifiquem do modo que desejarem. Havia um pressuposto de que o código seria controlado e desenvolvido por um pequeno grupo principal, em vez de usuários do código.

O software de código aberto estendeu essa ideia, usando a internet para recrutar uma população muito maior de desenvolvedores voluntários. Muitos deles também são usuários do código. Pelo menos a princípio, qualquer colaborador de um projeto de código aberto pode informar e consertar defeitos, além de propor novas características e funcionalidades. No entanto, na prática, os sistemas de código aberto bem-sucedidos ainda se baseiam em um grupo principal de desenvolvedores que controlam as mudanças no software.

O software de código aberto é a espinha dorsal da internet e da engenharia de software. O sistema operacional Linux é o mais utilizado em servidores, assim como o servidor web Apache, que também é de código aberto. Outros produtos de código aberto importantes e aceitos universalmente são Java, o IDE Eclipse e o sistema de gerenciamento de banco de dados mySQL. O sistema operacional Android está instalado em milhões de celulares e tablets. Os grandes protagonistas do setor de computadores, como a IBM e a Oracle, apoiam o movimento de código aberto e baseiam seu software em produtos que seguem essa filosofia. Milhares de outros sistemas e componentes de código aberto menos conhecidos também podem ser utilizados.

Normalmente, adquirir software de código aberto é barato ou até mesmo gratuito — de maneira geral, é possível baixar softwares de código aberto sem pagar por eles. Entretanto, se quiser documentação e suporte, então pode ser necessário pagar por isso, muito embora os custos sejam razoavelmente baixos. O outro benefício principal

de usar produtos de código aberto é que os sistemas que seguem essa filosofia e que são amplamente utilizados também são bastante confiáveis. Eles têm uma grande população de usuários dispostos a consertar os problemas sozinhos em vez de informar esses problemas para o desenvolvedor e esperar por uma nova versão do sistema. Os defeitos são descobertos e consertados mais rapidamente do que é possível na maioria das vezes com um software proprietário.

Para uma empresa envolvida no desenvolvimento de software, existem duas questões do código aberto que precisam ser consideradas:

1. O produto que está sendo desenvolvido deve usar componentes de código aberto?
2. Uma abordagem de código aberto deve ser utilizada para o desenvolvimento do próprio software?

As respostas para essas perguntas dependem do tipo de software que está sendo desenvolvido, da formação e da experiência do time de desenvolvimento.

Se estiver desenvolvendo um produto de software para venda, então o *time to market* (tempo até o produto chegar ao mercado) e a redução de custos são críticos. Se estiver desenvolvendo software em um domínio no qual existem sistemas de código aberto com alta qualidade, é possível poupar tempo e dinheiro usando esses sistemas. No entanto, se estiver desenvolvendo software para um conjunto específico de requisitos organizacionais, então pode não ser uma opção usar componentes de código aberto. O software poderá ter que ser integrado a sistemas existentes que são incompatíveis com os sistemas de código aberto disponíveis. Entretanto, ainda assim poderia ser mais rápido e barato modificar o sistema de código aberto do que desenvolver novamente a funcionalidade da qual você necessita.

Hoje em dia, muitas empresas de desenvolvimento de software estão usando uma abordagem de código aberto, especialmente para sistemas especializados. Seu modelo de negócios não depende da venda de um produto de software, mas, sim, de vender suporte para o produto. Elas acreditam que envolver a comunidade de código aberto vai permitir que o software seja desenvolvido de forma mais barata e mais rápida, criando uma comunidade de usuários do software.

Algumas empresas acreditam que adotar uma abordagem de código aberto revelará conhecimentos confidenciais do negócio para seus concorrentes e, portanto, relutam em adotar este modelo de desenvolvimento. No entanto, se estiver trabalhando em uma pequena empresa e abrir o código-fonte de seu software, pode ser tranquilizador para os clientes de que eles serão capazes de dar suporte ao software se a empresa vir a falir.

Publicar o código-fonte de um sistema não significa necessariamente que as pessoas de uma comunidade mais ampla vão ajudar no seu desenvolvimento. Os produtos de código aberto mais bem-sucedidos têm sido as plataformas, e não os sistemas de aplicação. Existe uma quantidade limitada de desenvolvedores que poderiam se interessar por sistemas de aplicação especializados. Transformar um sistema de software em código aberto não garante o envolvimento da comunidade. Existem milhares de projetos de código aberto na Sourceforge e no GitHub que têm apenas um punhado de downloads. No entanto, se os usuários do seu software se preocuparem com a sua futura disponibilidade, transformar o software em código aberto significa que eles poderão ter sua própria cópia e tranquilizar-se de que não perderão o acesso a esse software.

7.4.1 Licenciamento de código aberto

Embora um princípio fundamental do desenvolvimento de código aberto seja que o código-fonte deve ser livremente disponível, isso não significa que qualquer um possa fazer o que quiser com ele. Legalmente, o desenvolvedor do código (uma empresa ou um indivíduo) o possui, e pode colocar restrições ao seu modo de utilização incluindo condições legalmente vinculantes em uma licença de software de código aberto (ST. LAURENT, 2004). Alguns desenvolvedores de código aberto acreditam que, se um componente de código aberto for utilizado para desenvolver um novo sistema, então esse sistema também deverá ser de código aberto. Outros estão dispostos a permitir que seu código seja utilizado sem essa restrição. Os sistemas desenvolvidos podem ser proprietários e vendidos como sistemas de código fechado.

A maioria das licenças de código aberto (CHAPMAN, 2010) consiste em variações de um dos três modelos gerais:

1. GNU General Public License (GPL): essa é a chamada licença recíproca que, de maneira simplificada, significa que, se você usar software de código aberto licenciado de acordo com a GPL, então seu software deverá ser publicado em código aberto.
2. GNU Lesser General Public License (LGPL): essa é uma variação da licença GPL, em que se pode escrever componentes vinculados ao código aberto sem ter que publicar a fonte desses componentes. No entanto, caso se mude o componente licenciado, deve-se publicar tal mudança como código aberto.
3. Berkley Standard Distribution (BSD) License: essa é uma licença não recíproca, significando que não há obrigação de republicar quaisquer modificações feitas no código aberto. É possível incluir o código nos sistemas proprietários que são vendidos. Ao usar componentes de código aberto, deve-se reconhecer o criador original do código. A licença MIT é uma variação da licença BSD com condições semelhantes.

As questões de licenciamento são importantes porque quem usa software de código aberto como parte de um produto de software pode ser obrigado, pelos termos da licença, a tornar o seu próprio produto de código aberto. Para tentar vender um software, pode ser preferível mantê-lo em segredo. Isso significa que pode ser melhor evitar o uso de software de código aberto com licença GPL em seu desenvolvimento.

As licenças não são um problema para quem está construindo um software a ser executado em uma plataforma de código aberto, mas que não reusa componentes de código aberto. No entanto, caso seja embutido software de código aberto no software sendo construído, poderá ser necessário controlar processos e ter bancos de dados para monitorar o que foi utilizado e sob quais condições de licenciamento. Para Bayersdorfer (2007), as empresas que gerenciam projetos e usam código aberto devem:

1. Estabelecer um sistema para manter informações sobre componentes de código aberto que são baixados e utilizados. Elas precisam manter uma cópia da licença de cada componente válida no momento em que ele foi utilizado. As licenças podem mudar, então é preciso conhecer as condições com as quais concordaram.
2. Estar cientes dos diferentes tipos de licenças e entender como um componente é licenciado antes de ser utilizado. Elas podem optar pelo uso de um

componente em um sistema, mas não em outro, já que planejam usar esses sistemas de maneiras diferentes.

3. Estar cientes das trajetórias de evolução dos componentes. Elas precisam saber um pouco sobre o projeto de código aberto em que os componentes são desenvolvidos para compreender como eles poderiam mudar no futuro.
4. Educar as pessoas a respeito do código aberto. Não basta ter procedimentos em vigor para garantir a observância das condições de licenciamento. Também é necessário educar os desenvolvedores sobre o código aberto e o licenciamento desse tipo de código.
5. Manter sistemas de auditoria. Os desenvolvedores, sujeitos a prazos apertados, poderiam ficar tentados a violar os termos de uma licença. Se for possível, as empresas devem ter software instalado para detectar e prevenir isso.
6. Participar da comunidade de código aberto. Se as empresas contam com produtos de código aberto, elas devem participar da comunidade e ajudar a apoiar o seu desenvolvimento.

A abordagem de código aberto é um dos vários modelos de negócio relacionados ao software. Nesse modelo, as empresas liberam o código-fonte do seu software e vendem serviços e consultoria associados a ele. Elas também podem vender serviços de software baseados na nuvem — uma opção atraente para os usuários que não têm expertise para gerenciar seu próprio sistema de código aberto e também versões especializadas do seu sistema para determinados clientes. Portanto, o código aberto tende a crescer em importância como uma maneira de desenvolver e distribuir software.

PONTOS-CHAVE

- O projeto e a implementação de software são atividades intercaladas. O nível de detalhe no projeto depende do tipo de sistema que está sendo desenvolvido e do fato de se utilizar uma abordagem ágil ou dirigida por plano.
- O processo de projeto orientado a objetos inclui atividades para projetar a arquitetura do sistema, identificar objetos no sistema, descrever o projeto usando diferentes modelos de objeto e documentar as interfaces dos componentes.
- Muitos modelos diferentes podem ser produzidos durante um processo de projeto orientado a objetos. Eles incluem os modelos estáticos (modelos de classe, modelos de generalização, modelos de associação) e os modelos dinâmicos (modelos de sequência, modelos de máquina de estados).
- As interfaces dos componentes devem ser definidas precisamente para que outros objetos possam usá-las. O estereótipo de interface da UML pode ser utilizado para definir interfaces.
- Durante o desenvolvimento de software, deve-se sempre considerar a possibilidade de reusar o software existente, seja como componentes, como serviços ou como sistemas completos.
- Gerenciamento de configuração é o processo de gerenciar mudanças em um sistema de software em desenvolvimento. Ele é essencial quando um time está cooperando para desenvolver um software.
- A maior parte do desenvolvimento é do tipo *host-target*. Um IDE é usado em uma máquina hospedeira (*host*) para desenvolver o software, que é transferido para uma máquina destino (*target* ou alvo) para execução.

- O desenvolvimento de código aberto (*open source*) envolve tornar disponível publicamente o código-fonte de um sistema. Isso significa que muitas pessoas podem propor mudanças e melhorias no software.

LEITURAS ADICIONAIS

Design patterns: elements of reusable object-oriented software. Esse é o manual original de padrões de software, que os apresentou para uma grande comunidade. GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. Addison-Wesley, 1995.

Applying UML and patterns: an introduction to object-oriented analysis and design and iterative development. 3. ed. Larman escreve claramente sobre projeto orientado a objetos e discute o uso da UML; essa é uma boa introdução ao uso dos padrões no processo de projeto. Embora tenha mais de dez anos, ainda é o melhor livro sobre o tema. LARMAN, C. Prentice-Hall, 2004.

Producing open source software: how to run a successful free software project. Este livro é um guia abrangente para o contexto do software de código aberto, questões de licenciamento e os aspectos práticos de tocar um projeto de desenvolvimento de código aberto. FOGEL, K. O'Reilly Media Inc., 2008.

Outras leituras sobre reúso de software serão sugeridas no Capítulo 15, e sobre gerenciamento de configuração, no Capítulo 25.

SITE[3]

Apresentações em PowerPoint para este capítulo disponíveis em: <http://software-engineering-book.com/slides/chap7/>.

Links para vídeos de apoio disponíveis em: <http://software-engineering-book.com/videos/implementation-and-evolution/>.

Mais informações sobre o sistema de informações meteorológicas disponíveis em: <http://software-engineering-book.com/case-studies/wilderness-weather-station/>.

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

EXERCÍCIOS

7.1 Usando a notação tabular exibida na Figura 7.3, especifique os casos de uso da estação meteorológica para 'Informar status' e 'Reconfigurar'. Você deve fazer suposições razoáveis quanto à funcionalidade exigida aqui.

7.2 Suponha que o sistema Mentcare esteja sendo desenvolvido usando uma abordagem orientada a objetos. Desenhe um diagrama de caso de uso mostrando pelo menos seis casos de uso possíveis para esse sistema.

7.3 Usando a notação gráfica da UML para classes, projete as seguintes classes, identificando atributos e operações. Use a sua própria experiência para decidir sobre os atributos e operações que devem estar associados a esses objetos.

- Um sistema de mensagens em um celular ou tablet.
- Uma impressora para um computador pessoal.
- Um sistema de música pessoal.
- Uma conta bancária.
- Um catálogo de biblioteca.

7.4 Usando como ponto de partida os objetos da estação meteorológica identificados na Figura 7.6, identifique outros objetos que possam ser utilizados nesse sistema. Projete uma hierarquia de herança para os objetos que você identificou.

7.5 Desenvolva o projeto da estação meteorológica para mostrar a interação entre o subsistema de coleta de dados e os instrumentos que coletam os dados climáticos. Use diagramas de sequência para mostrar essa interação.

7.6 Identifique possíveis objetos nos seguintes sistemas e desenvolva um projeto orientado a objetos para eles. Você pode fazer quaisquer suposições razoáveis sobre os sistemas quando derivar o seu projeto.

- Um diário de grupo e um sistema de gerenciamento de tempo se destinam a permitir o agendamento de reuniões e compromissos entre um grupo de colegas de trabalho. Quando deve ocorrer uma reunião envolvendo uma série de pessoas o sistema encontra uma vaga comum em cada um de seus diários e marca a reunião para aquele horário. Se não houver uma vaga comum, ele interage com o usuário para reorganizar seu diário pessoal e arrumar um horário para a reunião.
- Um posto de gasolina deve ser configurado para operação totalmente automatizada. Os motoristas passam seus cartões de crédito por uma leitora conectada à bomba; o cartão é verificado pela comunicação com um computador da operadora, sendo estabelecido um limite

de combustível. O motorista pode abastecer com o combustível necessário. Quando o abastecimento estiver concluído e a mangueira da bomba for devolvida ao suporte, a conta do cartão de crédito do motorista é debitada com o custo do combustível. O cartão de crédito é devolvido após ser debitado. Se o cartão for inválido, a bomba o devolve antes de liberar a saída do combustível.

7.7 Desenhe um diagrama de sequência mostrando as interações dos objetos em um sistema de diário de grupo quando um grupo de pessoas está marcando uma reunião.

7.8 Desenhe um diagrama de estado da UML mostrando as possíveis mudanças de estado no diário de grupo ou no sistema do posto de abastecimento de combustível.

7.9 Usando exemplos, explique por que o gerenciamento de configuração é importante quando um time está desenvolvendo um produto de software.

7.10 Uma pequena empresa desenvolveu um produto de software especializado que ela configura especialmente para cada cliente. Os clientes novos normalmente têm necessidades específicas a serem incorporadas ao seu sistema e eles pagam para que sejam desenvolvidas e integradas ao produto. A empresa de software tem uma oportunidade para disputar um novo contrato, o que mais do que dobraria a sua base de clientes. O novo cliente deseja ter algum envolvimento na configuração do sistema. Explique porque, nessas circunstâncias, poderia ser uma boa ideia a empresa dona do software transformá-lo em código aberto.

REFERÊNCIAS

ABBOTT, R. Program design by informal english descriptions. *Comm. ACM*, v. 26, n. 11, 1983. p. 882--894. doi:10.1145/182.358441.

ALEXANDER, C. *A Timeless way of building*. Oxford, UK: Oxford University Press. 1979.

BAYERSDORFER, M. Managing a project with open source components. *ACM Interactions*, v. 14, n. 6, 2007. p. 33-34. doi: 10.1145/1300655.1300677.

BECK, K.; CUNNINGHAM, W. A laboratory for teaching object-oriented thinking. In: *Proc. OOPSLA'89 (Conference on Object-Oriented Programming, Systems, Languages and Applications)*, 1-6. ACM Press. 1989. doi:10.1145/74878.74879.

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

_____. *Pattern-oriented software architecture volume 5*: on patterns and pattern languages. 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.

CHAPMAN, C. A short guide to open-source and similar licences. *Smashing Magazine*, 24 mar. 2010. Disponível em: <http://www.smashingmagazine.com/2010/03/24/a-short-guide-to-open-source-and-similar-licenses/>. Acesso em: 22 abr. 2018.

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

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

LOELIGER, J.; MCCULLOUGH, M. *Version control with Git*: powerful tools and techniques for collaborative software development. Sebastopol, CA: O'Reilly & Associates, 2012.

PILATO, C.; COLLINS-SUSSMAN, B.; FITZPATRICK, B. *Version control with Subversion*. Sebastopol, CA: O'Reilly & Associates, 2008.

RAYMOND, E. S. *The cathedral and the bazaar*: musings on Linux and open source by an accidental revolutionary. Sebastopol. CA: O'Reilly & Associates, 2001.

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

ST. LAURENT, A. *Understanding open source and free software licensing*. Sebastopol, CA: O'Reilly & Associates, 2004.

VOGEL, L. *Eclipse IDE*: a tutorial. Hamburg, Germany: Vogella Gmbh, 2013.

WIRFS-BROCK, R.; WILKERSON, B.; WEINER, L. *Designing object-oriented software*. Englewood Cliffs, NJ: Prentice-Hall, 1990.

8 Teste de software

OBJETIVOS

O objetivo deste capítulo é introduzir o teste de software e seus processos. Ao ler este capítulo, você:

- compreenderá os estágios de teste, desde o desenvolvimento até a aceitação pelos clientes do sistema;
- será apresentado às técnicas que ajudam a escolher os casos de teste concebidos para descobrir defeitos de programação;
- compreenderá o desenvolvimento com testes *a priori* (*test-first*), em que os testes são projetados antes da escrita do código e executados automaticamente;
- conhecerá três tipos de testes diferentes — teste de componentes, teste de sistemas e teste de lançamento (*release*);
- compreenderá as diferenças entre teste de desenvolvimento e teste de usuário.

CONTEÚDO

Os testes pretendem mostrar que um programa faz o que foi destinado a fazer e descobrir defeitos antes que ele seja colocado em uso. Em um teste de software, um programa é executado com uso de dados artificiais, e os resultados são conferidos em busca de erros, anomalias ou informações sobre os atributos não funcionais do programa.

Quem testa um software, tenta fazer duas coisas:

1. Demonstrar ao desenvolvedor e ao cliente que o software atende aos seus requisitos. No caso de software customizado, isso significa que deve haver pelo menos um teste para cada requisito no documento de requisitos. No caso de produtos de software genéricos, isso significa que deve haver testes para todas as características do sistema que serão incluídas no lançamento do produto. Também podem ser testadas combinações de características para averiguar a existência de interações indesejadas entre elas.