

CONCEITOS- -CHAVE

certificação	499
especificação de estrutura de caixa ..	492
especificação funcional	493
linguagem de especificação Z ...	508
linguagens de especificações formais	504
modelo de processo sala limpa	493
object constraint language (OCL) ...	505
projeto sala limpa ..	496
refinamento do projeto	496
verificação de correção	492

Diferentemente das revisões e testes que começam logo que os modelos e os códigos de software são desenvolvidos, a modelagem formal e a verificação incorporam métodos de modelagem especializados que são integrados com abordagens prescritas de verificação. Sem a abordagem apropriada, a verificação não pode ser feita.

Neste capítulo, discutem-se dois métodos formais de modelagem e verificação — *engenharia de software sala limpa* e *métodos formais*. Ambos demandam abordagem de verificação especializada e a cada um se aplica um método único de verificação. Ambos são muito rigorosos e nenhum deles é usado amplamente pela comunidade de engenharia de software. Mas para criar um software “à prova de balas”, esses métodos podem ser muito úteis. Vale a pena conhecê-los.

Engenharia de software sala limpa é uma abordagem que enfatiza a necessidade de agregar precisão ao software enquanto está sendo desenvolvido. Em vez do ciclo clássico de análise, codificação, teste e depuração, a abordagem sala limpa sugere um diferente ponto de vista [Lin94b]:

A filosofia por trás da engenharia de software sala limpa é evitar a dependência de processos onerosos de remoção de defeitos, escrevendo incrementos de código corretos já na primeira vez e verificando sua precisão antes do teste. Seu modelo de processo incorpora a certificação estatística de qualidade dos incrementos de código à medida que vão se acumulando no sistema.

PANORAMA

O que é? Quantas vezes você já ouviu alguém dizer, “Faça direito já na primeira vez”? Se aplicássemos isso ao software, seriam gastos bem menos esforços com retrabalho desnecessário de software. Dois métodos avançados de engenharia de software — engenharia de software sala limpa e métodos formais — ajudam a equipe de software a “fazer direito já na primeira vez” proporcionando uma abordagem matemática baseada na modelagem de programa e na habilidade de verificar se o modelo está correto. A engenharia de software de sala limpa enfatiza a verificação matemática da correção antes de começar a construção do programa e a certificação da confiabilidade do software como parte da atividade de teste. Os métodos formais usam a teoria dos conjuntos e a notação lógica para criar uma definição clara dos fatos (requisitos) que podem ser analisados para melhorar (ou mesmo provar) a correção e consistência. O limite inferior para ambos os métodos é a criação de software com taxas de falhas extremamente baixas.

Quem realiza? Um engenheiro de software especialmente treinado.

Por que é importante? Erros criam retrabalho. O retrabalho consome tempo e aumenta os custos. Não seria maravilhoso se pudéssemos reduzir significativamente a quantidade de erros (defei-

tos, bugs) introduzidos à medida que o software é projetado e construído? Essa é a premissa da modelagem e verificação formais.

Quais são as etapas envolvidas? Modelos de requisitos e de projeto são criados usando notação especializada que é favorável à verificação matemática. A engenharia de software de sala limpa utiliza representação de estrutura de caixas que encapsula o sistema (ou algum aspecto do sistema) em um nível específico de abstração. É aplicada a verificação da correção uma vez completo o projeto de estrutura de caixas. Depois de verificada a correção para cada estrutura de caixas, inicia-se o teste de uso estatístico. Métodos formais traduzem os requisitos do software para uma representação mais formal aplicando a notação e heurística de conjuntos para definir a invariante dos dados, estados e operações para uma função do sistema.

Qual é o artefato? É desenvolvido um modelo especializado e formal de requisitos. São registrados os resultados das provas de correção e dos testes estatísticos de uso.

Como garantir que o trabalho foi realizado corretamente? É aplicada uma prova formal de correção ao modelo de requisitos. Os testes estatísticos de uso experimentam cenários de uso para assegurar que os erros na funcionalidade de usuário sejam descobertos e corrigidos.

De muitas formas, a abordagem sala limpa eleva a engenharia de software a outro nível, enfatizando a necessidade de *provar* a precisão.

Modelos desenvolvidos por meio de *métodos formais* são descritos usando sintaxe e semântica formais que especificam a função e o comportamento do sistema. A especificação é na forma matemática (por exemplo, o cálculo de predicado pode ser usado como base para uma linguagem de especificação formal). Em sua introdução aos métodos formais, Anthony Hall [Hal90] faz um comentário que se aplica igualmente aos métodos sala limpa:

Métodos formais [e a engenharia de software sala limpa] são controvertidos. Seus defensores dizem que podem revolucionar o desenvolvimento [de software]. Seus críticos acham que são impossivelmente difíceis. Enquanto isso, para a maioria das pessoas, métodos formais [e engenharia de software sala limpa] são tão estranhos que é difícil julgar os argumentos a favor ou contra.

Neste capítulo, exploraremos os métodos formais de modelagem e verificação e examinaremos seu impacto sobre a engenharia de software nos próximos anos.

21.1 ESTRATÉGIA SALA LIMPA

"A única maneira de os erros ocorrerem em um programa é serem colocados lá pelo autor. Não há outros mecanismos conhecidos... A prática correta procura evitar a inserção de erros e, se isso falhar, deve-se removê-los antes de testar ou de utilizar qualquer outro programa."

Harlan Mills

WebRef

Uma excelente fonte de informações e recursos para engenharia de software sala limpa pode ser encontrada em www.cleantsoft.com.

"A engenharia de software sala limpa consegue um controle estatístico de qualidade sobre o desenvolvimento de software separando estritamente o processo de projeto do processo de teste em uma sequência de desenvolvimento incremental de software."

Harlan Mills

A engenharia de software sala limpa usa uma versão especializada do modelo incremental de software introduzido no Capítulo 2. Uma "sequência de incrementos de software" [Lin94b] é desenvolvida por pequenas equipes de software independentes. À medida que cada incremento é certificado, ele é integrado no todo. Daí o motivo pelo qual o sistema cresce com o tempo.

A sequência das tarefas sala limpa para cada incremento está ilustrada na Figura 21.1. De acordo com a sequência para incrementos sala limpa, ocorrem as seguintes tarefas:

Planejamento de incremento. É desenvolvido um plano de projeto que adota a estratégia incremental. São criados a funcionalidade de cada incremento, seu tamanho projetado e um cronograma de desenvolvimento sala limpa. Deve-se ter cuidado especial para garantir que os incrementos certificados serão integrados no seu devido tempo.

Coleta dos requisitos. Usando técnicas similares àquelas introduzidas no Capítulo 5, é desenvolvida uma descrição mais detalhada dos requisitos no nível do cliente (para cada incremento).

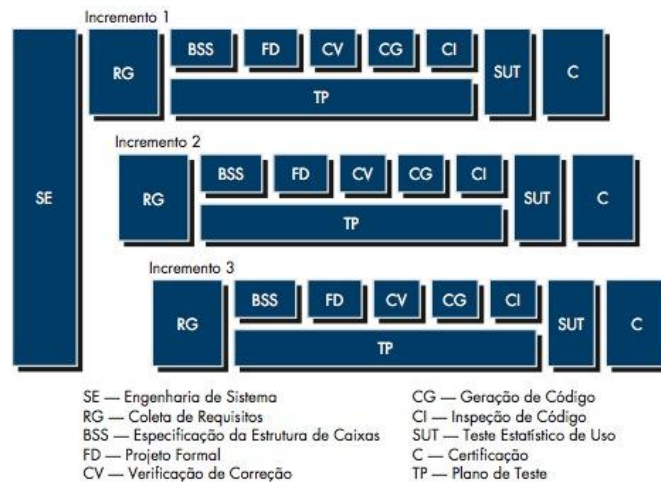
Especificação da estrutura de caixa. É usado um método de especificação que emprega *estruturas de caixas* para descrever a especificação funcional. As estruturas de caixas "isolam e separam a definição criativa de comportamento, dados e procedimentos em cada nível de refinamento" [Hev93].

Projeto formal. Por meio da abordagem de estrutura de caixas, o projeto sala limpa é uma extensão natural e contínua da especificação. Embora seja possível fazer uma clara distinção entre as duas atividades, as especificações (chamadas de *caixas-pretas*) são refinadas iterativamente (dentro de um incremento) para se tornarem análogas aos projetos arquitetônicos e no nível de componentes (chamados de *caixas de estado* e *caixas-claras*, respectivamente).

Verificação de correção. A equipe sala limpa executa uma série de atividades de verificação rigorosa da correção sobre o projeto e depois sobre o código. A verificação (Seção 21.3.2) começa com a estrutura de caixas de nível mais alto (especificação) e move-se em direção ao detalhe de projeto e código. O primeiro nível de verificação de correção ocorre aplicando-se uma série de "perguntas de correção" [Lin88]. Se essas perguntas não demonstrarem que a especificação está correta, utilizam-se métodos de verificação mais formais (matemáticos).

Geração de código, inspeção e verificação. As especificações de estrutura de caixa, representadas em uma linguagem especializada, são traduzidas em linguagem de programação apropriada. São usadas revisões técnicas (Capítulo 15) para garantir a conformidade

FIGURA 21.1
O modelo de
processo
sala limpa



da semântica do código e das estruturas de caixa e correção sintática do código. Então é executada a verificação de correção para o código fonte.

Planejamento do teste estatístico. É analisado o uso projetado do software, e é planejada e projetada uma série de casos de teste que experimentam uma “distribuição de probabilidades” de uso (Seção 21.4). De acordo com a Figura 21.1, essa atividade sala limpa é executada em paralelo à especificação, verificação e geração de código.

Teste estatístico de uso. Recordando que o teste exaustivo de software é algo impossível (Capítulo 18), é sempre necessário projetar um número finito de casos de teste. As técnicas estatísticas de uso [Poo88] executam uma série de testes derivados de uma amostragem estatística (a distribuição de probabilidades que mencionamos antes) de todas as execuções de programa possíveis por todos os usuários de uma população-alvo (Seção 21.4).

Certificação. Uma vez completadas a verificação, inspeção e teste de uso (e todos os erros forem corrigidos), o incremento é certificado como pronto para a integração.

As quatro primeiras atividades no processo sala limpa preparam o cenário para as atividades de verificação formal que vêm em seguida. Por essa razão, começa-se a discussão da abordagem sala limpa com as atividades de modelagem que são essenciais para a verificação formal a ser aplicada.

21.2 ESPECIFICAÇÃO FUNCIONAL

“Há uma coisa curiosa sobre a vida: se você se recusa a aceitar qualquer coisa que não seja o melhor, frequentemente acabará obtendo o melhor.”

W. Somerset
Maugham

A abordagem de modelagem em engenharia de software sala limpa usa um método chamado de *especificação de estrutura de caixa*. Uma “caixa” encapsula o sistema (ou algum aspecto do sistema) em algum nível de detalhe. Por meio de um processo de elaboração ou refinamento passo a passo, as caixas são refinadas em uma hierarquia em que cada caixa tem transparência referencial. Isto é, “O conteúdo de informações da especificação de cada caixa é suficiente para definir seu refinamento, sem depender da implementação de qualquer outra caixa” [Lin94b]. Isso habilita o analista a particionar um sistema hierarquicamente, mudando da representação essencial no topo para o detalhe específico de implementação na base. São usados três tipos de caixas:

Como se consegue um refinamento como parte de uma especificação de estrutura de caixa?

Caixa-preta. A caixa-preta especifica o comportamento de um sistema ou uma parte de um sistema. O sistema (ou parte) responde a estímulos específicos (eventos) aplicando um conjunto de regras de transição que mapeiam os estímulos em uma resposta.

Caixa de estado. A caixa de estado encapsula dados de estado e serviços (operações) de maneira análoga a objetos. Nessa visão de especificação, são representadas entradas para a caixa de estado (estímulos) e saídas (respostas). A caixa de estado representa o “histórico de estímulo” da caixa-preta, isto é, os dados encapsulados na caixa de estado que devem ser retidos entre as transições inferidas.

Caixa-clara. As funções de transição inferidas pela caixa de estado são definidas na caixa-clara. Em outras palavras, uma caixa-clara contém o projeto procedimental para a caixa de estado.

A Figura 21.2 ilustra a abordagem de refinamento usando a especificação de estrutura de caixas. Uma caixa-preta (BB1) define as respostas para um conjunto completo de estímulos. BB1 pode ser refinado em um conjunto de caixas-pretas, BB1.1 até BB1.n, cada uma das quais lida com uma classe de comportamento. O refinamento continua até que seja identificada uma classe coerente de comportamento (por exemplo, BB1.1.1). Uma caixa de estado (SB1.1.1) é então definida para a caixa-preta (BB1.1.1). Nesse caso, SB1.1.1 contém todos os dados e serviços necessários para implementar o comportamento definido por BB1.1.1. Por fim, SB1.1.1 é refinada em caixas-claras (CB1.1.1.n) e são especificados os detalhes do projeto procedimental.

À medida que ocorre cada um desses passos de refinamento, ocorre também a verificação de correção. As especificações de caixa de estado são verificadas para garantir que cada uma esteja conforme o comportamento definido pela especificação da caixa-preta pai. De maneira semelhante, as especificações caixas-claras são verificadas em relação à caixa de estado pai.

FIGURA 21.2

Refinamento da estrutura de caixas

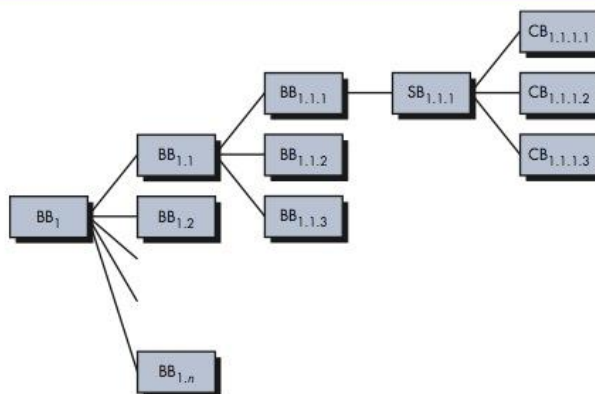
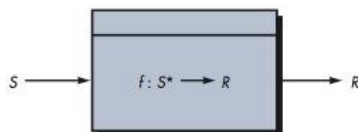


FIGURA 21.3

Uma especificação caixa-preta



21.2.1 Especificação caixa preta

A especificação *caixa-preta* descreve uma abstração, estímulos e respostas usando a notação mostrada na Figura 21.3 [Mil88]. A função f é aplicada a sequência S^* de entradas (estímulos) S e as transforma em uma saída (resposta) R . Para componentes simples de software, f pode ser uma função matemática, mas em geral, f é descrita por meio de linguagem natural (ou linguagem de especificação formal).

Muitos dos conceitos introduzidos para sistemas orientados a objetos são também aplicáveis à caixa-preta. As abstrações de dados e as operações que manipulam as abstrações são encapsuladas pela caixa-preta. Como em hierarquia de classe, a especificação caixa-preta pode exibir hierarquias de uso nas quais as caixas de baixo nível herdam as propriedades daquelas caixas de mais alto nível na estrutura.

21.2.2 Especificação caixa de estado

A *caixa de estado* é uma simples generalização de uma máquina de estado [Mil88]. Lembrando da discussão da modelagem comportamental e diagramas de estado no Capítulo 7, um estado é algum modo observável do comportamento de um sistema. À medida que ocorre o processamento, um sistema responde a eventos (estímulos) fazendo uma transição do estado atual para algum novo estado. Enquanto é feita a transição, pode ocorrer uma ação. A caixa de estado utiliza a abstração de dados para determinar a transição para o próximo estado e a ação (resposta) que ocorrerá em consequência da transição.

Observando a Figura 21.4, a caixa de estado incorpora uma caixa-preta g . O estímulo S , colocado na caixa-preta, chega de alguma fonte externa e de um conjunto de estados internos de sistema T . Mills [Mil88] fornece uma descrição matemática da função f da caixa-preta contida na caixa de estado:

$$g : S^* \times T^* \rightarrow R \times T$$

em que g é uma subfunção ligada a um estado específico t . Quando considerada coletivamente, os pares estado-subfunção (t, g) definem a função caixa-preta f .

21.2.3 Especificação caixa clara

A especificação caixa-clara está estreitamente relacionada com o projeto procedimental e programação estruturada. Essencialmente, a subfunção g dentro da caixa de estado é substituída pelas construções de programação estruturada que implementam g .

Como exemplo, considere a caixa-clara da Figura 21.5. A caixa-preta g , da Figura 21.3, é substituída por uma sequência de construções que incorpora uma condicional.

FIGURA 21.4
Especificação
caixa-preta

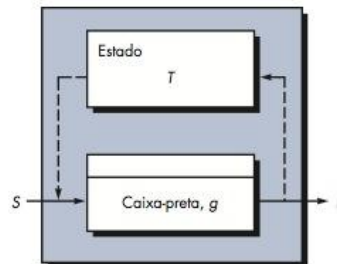
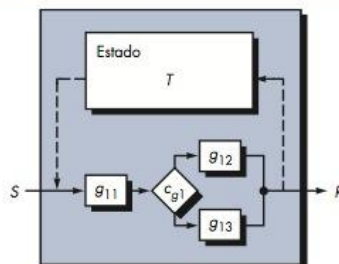


FIGURA 21.5
Especificação
caixa-clara



Estas, por sua vez, podem ser refinadas em caixas-claras em um nível mais baixo, à medida que prossegue o refinamento passo a passo.

É importante notar que a especificação procedimental descrita na hierarquia caixa-clara pode ter sua correção provada. Esse tópico é considerado na Seção 21.3.

21.3 PROJETO SALA LIMPA

A engenharia de software sala limpa usa intensamente a filosofia de programação estruturada (Capítulo 10). Mas neste caso, a programação estruturada é aplicada com muito mais rigor.

Funções básicas de processamento (descritas durante os primeiros refinamentos da especificação) são refinadas usando uma “expansão passo a passo de funções matemáticas em estruturas de conectivos lógicos [por exemplo, if-then-else] e subfunções, em que a expansão [é] executada até que todas as subfunções identificadas possam ser diretamente declaradas na linguagem de programação empregada para implementação” [Dye92].

A abordagem de programação estruturada pode ser usada eficazmente para refinar uma função, mas e o projeto de dados? Aqui entram em cena muitos conceitos fundamentais de projeto (Capítulo 8). Dados de programação são encapsulados como um conjunto de abstrações atendidas por subfunções. Utilizam-se os conceitos de encapsulamento de dados, ocultamento de informações e tipos de dados para criar o projeto de dados.

21.3.1 Refinamento de projeto

Cada especificação caixa-clara representa o projeto de um procedimento (subfunção) necessário para obter uma transição caixa de estado. Na caixa-clara, construções de programação estruturada e refinamento passo a passo são usados para representar o detalhe procedimental. Por exemplo, uma função de programa f é refinada em uma sequência de subfunções g e h . Estas, por sua vez, são refinadas em construções condicionais (por exemplo, if-then-else e do-while). O refinamento continua até que haja detalhe procedimental suficiente para criar o componente em questão.

Em cada nível de refinamento, a equipe sala limpa¹ executa uma *verificação formal de correção*. Para tanto, uma série de condições genéricas de correção é anexada às construções de programação estruturada. Se uma função f é expandida em uma sequência g e h , a condição de precisão para toda a entrada a f é

- g seguido por h faz f ?

¹ Como a equipe inteira está envolvida no processo de verificação, é pouco provável que seja produzido um erro na condição da própria verificação.

? Que condições são aplicadas para provar a correção das construções estruturadas?

Quando uma função p é refinada para uma condicional da forma, $\text{if } \langle c \rangle \text{ then } q, \text{ else } r$, a condição de correção para toda entrada em p é

- Sempre que a condição $\langle c \rangle$ for verdadeira, q faz p ; e sempre que $\langle c \rangle$ for falsa, r faz p ?

Quando a função m é refinada como um laço, as condições de correção para toda a entrada a m são

- O término está garantido?
- Sempre que $\langle c \rangle$ for verdadeiro, n seguido de m faz m ; e sempre que $\langle c \rangle$ for falsa, a saída do ciclo ainda faz m ?

A cada vez que uma caixa-clara for refinada para o próximo nível de detalhe, essas condições de correção são aplicadas.



Se você se limitar a apenas construções estruturadas à medida que desenvolve um projeto procedimental, a prova de correção é simples. Se você violar as construções, as provas de correção tornam-se difíceis ou impossíveis.

21.3.2 Verificação de projeto

Você deve observar que o uso de construções de programação estruturada limita o número de testes de correção que devem ser feitos. É verificada uma única condição para as sequências; são testadas duas condições para if-then-else e verificadas três condições para laços.

Para ilustrarmos a verificação de correção para um projeto procedimental, usamos um exemplo simples introduzido por Linger, Mills e Witt [Lin79]. O objetivo é projetar e verificar um pequeno programa que encontra a parte inteira y de uma raiz quadrada de um número inteiro x . O projeto procedimental é representado usando-se o fluxograma da Figura 21.6.²

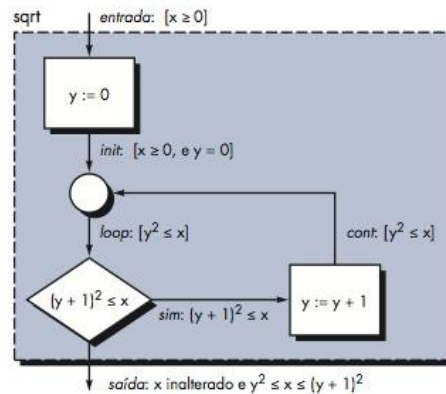
Para verificar a correção desse projeto, são acrescentadas condições de entrada e saída conforme mostra a Figura 21.6. A condição de entrada menciona que x deve ser maior ou igual a 0. A condição de saída requer que x permaneça inalterado e que y satisfaça a expressão da Figura. Para provar que o projeto está correto, é necessário provar que as condições *init*, *loop*, *cont*, *sim* e *saída* mostradas na Figura 21.6 são verdadeiras em todos os casos. Às vezes chama-se isso de subprovas.

1. A condição *init* demanda que $[x \geq 0 \text{ e } y = 0]$. Baseado nos requisitos do problema, assume-se que a condição de entrada é correta.³ Portanto, a primeira parte da condição *init*, $x \geq 0$, é satisfeita. Observando o fluxograma, a instrução que precede imediatamente a condição *init* faz $y = 0$. Portanto, a segunda parte da condição *init* também é satisfeita. Daí, *init* é verdadeira.

FIGURA 21.6

Calculando a parte inteira de uma raiz quadrada

Fonte: [Lin79]



² A Figura 21.6 foi adaptada de [Lin94]. Usada com permissão.

³ Um valor negativo para a raiz quadrada não tem significado nesse contexto.

PONTO-CHAVE

Para provar que um projeto está correto, você precisa primeiro identificar todas as condições e então provar que cada uma delas assume um valor booleano apropriado. Elas são chamadas de subprovas.

2. A condição *loop* pode ser encontrada em uma de duas formas: (1) diretamente de *init* (neste caso, a condição *loop* é satisfeita diretamente) ou via fluxo de controle que passa através da condição *cont*. Como a condição *cont* é idêntica à condição *loop*, *loop* é verdadeira independentemente do caminho que leva até ela.
3. A condição *cont* é encontrada apenas depois que o valor de *y* é incrementado de 1. Além disso, o caminho do fluxo de controle que leva a *cont* pode ser invocado somente se a condição *sim* também for verdadeira. Daí, se $(y + 1)^2 \leq x$, segue-se que $y^2 \leq x$. A condição *cont* é satisfeita.
4. A condição *sim* é testada na lógica condicional mostrada. Logo, a condição *sim* deve ser verdadeira quando o fluxo de controle se move ao longo do caminho mostrado.
5. A condição *saída* primeiro demanda que *x* permaneça inalterado. Um exame do projeto indica que *x* não aparece em nenhum lugar à esquerda de um operador de atribuição. Não há chamadas de função que use *x*. Portanto, ele fica inalterado. Como o teste condicional $(y + 1)^2 \leq x$ deve falhar em atingir a condição de *saída*, segue-se que $(y + 1)^2 \leq x$. Além disso, a condição *loop* deve ainda ser verdadeira (isto é, $y^2 \leq x$). Portanto, $(y + 1)^2 > x$ e $y^2 \leq x$ podem ser combinados para satisfazer a condição de saída.

Você deve ainda garantir que o laço termine. Um exame da condição *loop* indica que, porque *y* é incrementado e $x \geq 0$, o laço deve eventualmente terminar.

Os cinco passos que acabamos de descrever são uma prova da correção do projeto do algoritmo mostrado na Figura 21.6. Você tem certeza agora que o projeto irá, de fato, calcular a parte inteira de uma raiz quadrada.

É possível uma abordagem matemática mais rigorosa da verificação de projeto. No entanto, uma discussão desse tópico está além do escopo deste livro. Se você tiver interesse, consulte [Lin79].

21.4 TESTE SALA LIMPA

“A qualidade não é um ato, é um hábito.”

Aristóteles

A estratégia e as táticas de teste sala limpa são fundamentalmente diferentes das abordagens de teste convencionais (Capítulos 17 até 20). Métodos convencionais derivam uma série de caso de testes para descobrir erros de projeto e codificação. O objetivo do teste sala limpa é validar requisitos de software demonstrando que uma amostragem estatística de casos de uso (Capítulo 5) foi executada de forma bem-sucedida.

21.4.1 Teste estatístico de uso

O usuário de um programa de computador raramente precisa entender os detalhes técnicos do projeto. O comportamento do programa visível ao usuário é controlado por entradas e eventos que muitas vezes são produzidos pelo usuário. Mas em sistemas complexos, o espectro possível de entradas e eventos (os casos de uso) pode ser extremamente amplo. Qual subconjunto de casos de uso verificará adequadamente o comportamento do programa? Essa é a primeira questão tratada no teste estatístico de uso.

O teste estatístico de uso “resume-se no teste do software da maneira que os usuários pretendem usá-lo” [Lin94b]. Para tanto, as equipes de teste sala limpa (também chamadas de *equipes de certificação*) devem determinar a distribuição de probabilidade de uso para o software. A especificação (caixa-preta) para cada incremento do software é analisada para determinar uma série de estímulos (entradas ou eventos) que fazem o software mudar seu comportamento. Com base em entrevistas com usuários em potencial, na criação de cenários de uso e em uma compreensão geral do domínio de aplicação, é atribuída a cada estímulo uma probabilidade de uso.

São gerados casos de testes para cada conjunto de estímulos⁴ de acordo com a distribuição de probabilidade de uso. Para ilustrar, considere o sistema *CasaSegura* já discutido neste



Mesmo que você decida ser contra a abordagem sala limpa, vale a pena considerar o teste estatístico de uso como parte integral de sua estratégia de teste.

⁴ Podem ser usadas ferramentas automáticas para conseguir isso. Para mais informações, veja [Dye92].

livro. A engenharia de software sala limpa está sendo usada para desenvolver um incremento de software que controle a interação do usuário com o teclado do sistema de segurança. Para esse incremento identificaram-se cinco estímulos. A análise indica a distribuição porcentual de probabilidade de cada estímulo. Para facilitar a seleção de casos de testes, essas probabilidades são mapeadas em intervalos numerados de 1 a 99 [Lin94] e ilustradas na tabela a seguir:

Estímulo ao programa	Probabilidade	Intervalo
Armar/Desarmar (AD)	50%	1–49
Definir zona (ZS)	15%	50–63
Consultar (Q)	15%	64–78
Testar (T)	15%	79–94
Alarme de pânico	5%	95–99

Para gerar uma sequência de casos de testes de uso que esteja de acordo com a distribuição de probabilidade de uso, geram-se números aleatórios entre 1 e 99. Cada número aleatório corresponde a um intervalo na distribuição de probabilidades mencionada. Portanto, a sequência de casos de testes de uso é definida aleatoriamente, mas corresponde a uma probabilidade apropriada de ocorrência de estímulo. Por exemplo, suponha que sejam geradas as seguintes sequências de números aleatórios:

13-94-22-24-45-56
81-19-31-69-45-9
38-21-52-84-86-4

Selecionando os estímulos apropriados com base no intervalo de distribuição mostrado na tabela, geram-se os seguintes casos de uso:

AD-T-AD-AD-AD-ZS
T-AD-AD-AD-Q-AD-AD
AD-AD-ZS-T-T-AD

A equipe de teste executa esses casos de uso e verifica o comportamento do software em relação à especificação para o sistema. São registrados os tempos para os testes de maneira que podem ser determinados os intervalos de tempo. Usando os intervalos de tempo, a equipe de certificação pode calcular o tempo médio para falhas (*mean-time-to-failure*, MTTF). Se for executada uma longa sequência de testes sem falha, o MTTF é baixo e a confiabilidade do software pode ser considerada alta.

21.4.2 Certificação

As técnicas de verificação e teste discutidas anteriormente neste capítulo levam a componentes de software (e incrementos completos) que podem ser certificados. No contexto da abordagem de engenharia de software sala limpa, a *certificação* implica que a confiabilidade (medida pelo MTTF) pode ser especificada para cada componente.

O impacto provável de componentes de software certificáveis vai muito além de um simples projeto sala limpa. Componentes de software reutilizáveis podem ser armazenados com seus cenários de uso, estímulos de programa e distribuições de probabilidade. Cada componente teria uma confiabilidade certificada sob o cenário de uso e regime de teste descrito. Essas informações são valiosas para aqueles que quiserem usar os componentes.

A abordagem de certificação envolve cinco passos [Woh94]: (1) devem ser criados cenários de uso, (2) é especificado um perfil de uso, (3) são gerados casos de testes com base no perfil, (4) são executados os testes, os dados de falhas são registrados e analisados, e (5) a confiabilidade



é calculada e certificada. Os passos 1 a 4 foram discutidos em uma seção anterior. A certificação para engenharia de software sala limpa requer a criação de três modelos [Poo93]:

Modelo de amostragem. O teste de software executa m casos de testes aleatórios e é certificado se não ocorrerem falhas ou se ocorrer um número especificado de falhas. O valor de m é derivado matematicamente para garantir que a confiabilidade desejada seja obtida.

Modelo de componente. Um sistema formado por n componentes deve ser certificado. O modelo de componente permite que o analista determine a probabilidade do componente i falhar antes de completar.

Modelo de certificação. A confiabilidade global do sistema é projetada e certificada.

Ao término do teste estatístico de uso, a equipe de certificação tem as informações necessárias para entregar um software com um MTTF certificado e computado usando cada um desses modelos. Se você estiver interessado em detalhes adicionais, veja [Cur86], [Mus87], ou [Poo93].

21.5 CONCEITOS DE MÉTODOS FORMAIS

A enciclopédia *The Encyclopedia of Software Engineering* [Mar01] define métodos formais da seguinte maneira:

Métodos formais usados no desenvolvimento de sistemas de computadores são técnicas baseadas matematicamente para descrever propriedades de sistemas. Esses métodos formais proporcionam uma estrutura dentro da qual as pessoas podem especificar, desenvolver e verificar sistemas de maneira sistemática em vez de *ad hoc*.

“Os métodos formais têm um tremendo potencial para melhorar a clareza e precisão das especificações de requisitos e para localizar erros sutis.”

Steve Easterbrook et al.

As propriedades desejadas de uma especificação formal — consistência, totalidade e ausência de ambiguidade — são os objetivos de todos os métodos de especificação. No entanto, a linguagem de especificação com base matemática, usada para métodos formais, resulta em uma possibilidade muito maior de obter essas propriedades. A sintaxe formal de uma linguagem de especificação (Seção 21.7) possibilita que os requisitos ou projeto sejam interpretados de uma única maneira, eliminando a ambiguidade que frequentemente ocorre quando uma linguagem natural (por exemplo, inglês) ou uma notação gráfica (por exemplo, UML) deve ser interpretada por um leitor. Os recursos descritivos da teoria dos conjuntos e notação lógica possibilitam definição clara dos requisitos. Para serem consistentes, os requisitos especificados em um ponto em uma especificação não podem ser contrariados em outro ponto. A consistência é obtida⁵ provando-se matematicamente que os fatos iniciais podem ser mapeados de maneira formal (usando regras de inferência) em declarações posteriores dentro da especificação.

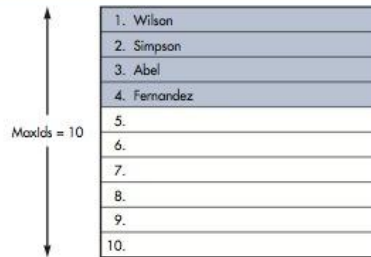
Para introduzirmos os conceitos básicos de métodos formais, vamos considerar alguns exemplos simples para ilustrar o uso da especificação matemática, sem aprofundar em detalhes da matemática.

Exemplo 1: Uma tabela de símbolos. Um programa é usado para manter uma tabela de símbolos. Emprega-se esse tipo de tabela frequentemente em diferentes tipos de aplicações. Ela consiste em uma coleção de itens sem qualquer duplicação. Um exemplo de uma típica tabela de símbolos está na Figura 21.7. Ela representa a tabela usada por um sistema operacional para manter os nomes dos usuários do sistema. Outros exemplos de tabelas incluem a coleção de nomes dos funcionários em um sistema de folha de pagamento, a coleção de nomes dos computadores em um sistema de comunicações em rede e a coleção de locais de destino em um sistema que gera as tabelas de horários de um sistema de transportes.

⁵ Na realidade, a totalidade é difícil de garantir, mesmo quando se utilizam métodos formais. Alguns aspectos de um sistema podem ser deixados indefinidos quando as especificações estão sendo criadas; outras características podem ser omitidas propositalmente para permitir que os projetistas tenham certa liberdade na escolha de abordagem de implementação; e, por fim, é impossível considerar cada cenário operacional em um sistema grande e complexo. Coisas podem ser omitidas por engano.

FIGURA 21.7

Uma tabela de símbolos



1. Wilson
2. Simpson
3. Abel
4. Fernandez
5.
6.
7.
8.
9.
10.

PONTO-CHAVE

Invariante de dados é um conjunto de condições que são verdadeiras ao longo de toda a execução do sistema que contém uma coleção de dados.



Outra maneira de entender o conceito de estado é dizer que os dados determinam o estado. Ou seja, você pode examinar os dados para ver em que estado o sistema está.

Suponha que a tabela apresentada neste exemplo seja formada apenas por nomes *MaxIds*. Essa afirmação, que impõe uma restrição na tabela, é o componente de uma condição conhecida como *invariante de dados*, uma condição verdadeira através de toda a execução do sistema que contém uma coleção de dados. A invariante de dados que se aplica à tabela de símbolos que acabamos de discutir tem duas componentes: (1) que a tabela terá apenas *MaxIds* e nada mais além disso e (2) que não haverá nomes duplicados na tabela. No caso do programa da tabela de símbolos, isso significa que não importa quando a tabela de símbolos será examinada durante a execução do sistema, ela sempre terá apenas *MaxIds* e nada mais, e não conterá duplicatas.

Outro conceito importante é o de *estado*. Muitas linguagens formais, como OCL (Seção 21.7.1), usam a noção de estado como discutido no Capítulo 7; isto é, um sistema pode estar em um dentre vários estados, cada um representando um modo de comportamento observável externamente. No entanto, uma definição diferente é usada na linguagem Z (Seção 21.7.2). Em Z (e linguagens relacionadas), o estado de um sistema é representado pelos dados do sistema armazenados (logo, Z sugere um número muito maior de estados, representando cada configuração possível dos dados). Usando a última definição no exemplo do programa da tabela de símbolos, o estado é a tabela de símbolos.

O conceito final é o de *operação*. Essa é uma ação que tem lugar em um sistema e lê ou escreve dados. Se o programa da tabela de símbolos está cuidando de adicionar ou remover nomes da tabela de símbolos, ele estará associado a duas operações: a operação *acrescentar()* um nome especificado à tabela de símbolos e a operação *remover()* da tabela um nome existente.⁶ Se o programa proporciona a facilidade de verificar se um nome específico está ou não contido na tabela, haverá uma operação que retornará alguma indicação sobre se o nome está ou não na tabela.

Três tipos de condições podem ser associadas a operações: invariantes, pré-condições e pós-condições. O *invariante* define o que está garantido que não mudará. Por exemplo, a tabela de símbolos tem uma invariante que diz que o número de elementos é sempre menor do que ou igual a *MaxIds*. Uma *pré-condição* define as circunstâncias nas quais uma operação em particular é válida. Por exemplo, a pré-condição para uma operação que acrescenta um nome a uma tabela de símbolos identificadores de funcionários é válida somente se o nome a ser adicionado não está contido na tabela e também se houver menos do que *MaxIds* identificadores de funcionários na tabela. A *pós-condição* de uma operação define o que é garantido que é verdadeiro após completar uma operação. Isso é definido por seu efeito sobre os dados. Para a operação *acrescentar()*, a pós-condição especificaria matematicamente que a tabela foi aumentada com o novo identificador.

⁶ Deve-se observar que a adição de um nome não pode ocorrer no estado *full* (cheio) e a exclusão de um nome é impossível no estado *empty* (vazio).

Exemplo 2: Um tratador de blocos. Uma das partes mais importantes de um sistema operacional simples é o subsistema que mantém os arquivos criados pelos usuários. Parte do subsistema de arquivos é o *tratador de blocos*. Os arquivos no sistema de armazenamento de arquivos são formados por blocos de armazenamento mantidos em um dispositivo de armazenamento. Durante a operação do computador, arquivos serão criados e excluídos, e isso requer a aquisição e liberação de blocos de armazenamento. Para enfrentar a tarefa, o subsistema de arquivamento deverá manter um reservatório de blocos não utilizados (livres) e manter controle dos blocos que estão correntemente em uso. Quando os blocos são liberados porque um arquivo foi excluído, eles normalmente são acrescentados a uma fila de blocos à espera de serem acrescentados ao reservatório de blocos não utilizados. Isso é apresentado na Figura 21.8. Nessa figura, há vários componentes: o reservatório de blocos não utilizados, os blocos que no momento formam os arquivos administrados pelo sistema operacional e aqueles blocos que estão esperando para ser acrescentados ao reservatório. Os blocos em espera são mantidos em fila, em que cada elemento da fila contém um conjunto de blocos de um arquivo excluído.



Técnicas de brainstorming podem funcionar bem quando você deve desenvolver uma invariante de dados para uma função razoavelmente complexa. Peça aos membros da equipe de software que escrevam os limites, as restrições e limitações para a função e então combinem e editem.

Para esse subsistema, o estado é a coleção de blocos livres, a coleção de blocos utilizados e a fila de blocos devolvidos. A invariante de dados, expressa em linguagem natural, é

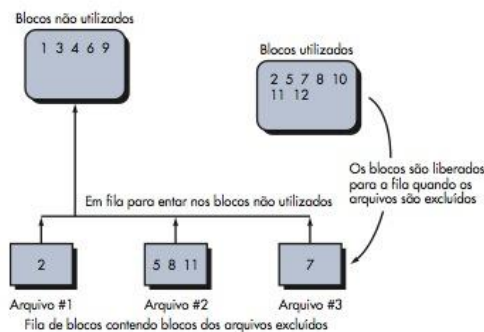
- Nenhum bloco será marcado como não usado e usado ao mesmo tempo.
- Todos os conjuntos de blocos mantidos na fila serão subconjuntos da coleção de blocos usados correntemente.
- Nenhum elemento da fila terá o mesmo número de blocos.
- A coleção formada por blocos usados e blocos não usados será a coleção total dos blocos que formam os arquivos.
- A coleção de blocos não usados não terá número de blocos duplicados.
- A coleção de blocos usados não terá número de blocos duplicados.

Algumas das operações associadas a esses dados são: *acrescentar()* uma coleção de blocos ao fim da fila, *remover()* uma coleção de blocos usados da frente da fila e colocá-los em uma coleção de blocos não usados e *verificar()* se uma fila de blocos está vazia.

A pré-condição de *acrescentar()* é que os blocos a ser adicionados devem estar na coleção de blocos usados. A pós-condição é que a coleção de blocos agora se encontra no fim da fila. A pré-condição de *remover()* é que a fila deve ter pelo menos um item nela. A pós-condição é que os blocos devem ser adicionados à coleção de blocos não usados. A operação *verificar()* não tem pré-condição. Isso significa que a operação é sempre definida, independentemente do valor do estado. A pós-condição fornece o valor *true* (verdadeiro) se a fila estiver vazia e *false* (falso) caso contrário.

FIGURA 21.8

Tratador de blocos



Nos exemplos apresentados nesta seção, introduzi os conceitos fundamentais de especificação formal, mas sem ênfase à matemática necessária para tornar a especificação formal. Na Seção 21.6, considerarei como a notação matemática pode ser usada para especificar formalmente algum elemento de um sistema.

21.6 APLICANDO NOTAÇÃO MATEMÁTICA⁷ PARA ESPECIFICAÇÃO FORMAL

Para ilustrarmos o uso da notação matemática na especificação formal de um componente de software, examinamos novamente o exemplo de tratador de blocos apresentado na Seção 21.5. Para recordar, um componente importante do sistema operacional de um computador mantém arquivos que foram criados pelos usuários. O tratador de blocos mantém um reservatório de blocos não usados e manterá o controle de blocos que estão correntemente em uso. Quando os blocos são liberados de um arquivo excluído, eles são normalmente acrescentados à fila de blocos que estão à espera de ser adicionados ao reservatório de blocos não usados. Isso foi mostrado esquematicamente na Figura 21.8.

? Como posso representar estados e invariantes de dados usando um conjunto e operadores lógicos?

Um conjunto denominado *BLOCKS* será formado por todos os números de bloco. *AllBlocks* é um conjunto de blocos que estão entre 1 e *MaxBlocks*. O estado será modelado por dois conjuntos e uma sequência. Os dois conjuntos são *used* e *free*. Ambos contêm blocos — o conjunto *used* contém os blocos que estão correntemente em uso em arquivos, e o conjunto *free* contém blocos que estão disponíveis para novos arquivos. A sequência irá conter conjuntos de blocos que estão prontos para ser liberados dos arquivos que foram excluídos. O estado pode ser descrito como

used, free: $\mathbb{P} \text{ BLOCKS}$

BlockQueue: $\text{seq } \mathbb{P} \text{ BLOCKS}$

Isso é muito semelhante à declaração de variáveis de programa. Estamos dizendo que *used* e *free* serão conjuntos de blocos e que *BlockQueue* será uma sequência, em que cada elemento será um conjunto de blocos. O invariante de dados pode ser escrito como

$\text{used} \cap \text{free} = \emptyset \wedge$

$\text{used} \cup \text{free} = \text{AllBlocks} \wedge$

$\forall i: \text{dom } \text{BlockQueue} \bullet \text{BlockQueue } i \subseteq \text{used} \wedge$

$\forall i, j: \text{dom } \text{BlockQueue} \bullet i \neq j \Rightarrow \text{BlockQueue } i \cap \text{BlockQueue } j = \emptyset$

Os componentes matemáticos do invariante de dados correspondem a quatro dos componentes de linguagem natural destacados, descritos anteriormente. A primeira linha do invariante de dados informa que não haverá blocos comuns nas coleções de blocos usados e blocos livres. A segunda linha diz que a coleção de blocos usados e blocos livres sempre será igual à coleção total de blocos no sistema. A terceira linha indica que o *i*-ésimo elemento na fila de blocos sempre será um subconjunto dos blocos usados. A linha final diz que, para quaisquer dois elementos da fila de blocos que não sejam o mesmo bloco, não haverá blocos comuns nesses dois elementos. Os dois componentes finais de linguagem natural do invariante de dados são implementados em virtude do fato de que *used* e *free* são conjuntos e, portanto, não contêm duplicatas.

A primeira operação a ser definida é aquela que remove um elemento do início da fila de blocos. A precondição é que deve haver pelo menos um item na fila:

$\# \text{BlockQueue} > 0,$

WebRef

Informações mais detalhadas sobre métodos formais podem ser encontradas em www.afm.sbu.ac.uk.

⁷ Escrevi essa seção baseando-me na hipótese de que você está familiarizado com a notação matemática associada a conjuntos e sequências e a notação lógica usada no cálculo de predicado. Se você precisar de uma revisão, há um breve resumo apresentado como recurso suplementar no site da 7a. edição. Para informações mais detalhadas, veja [Jec06] ou [Pot04].

A pós-condição é que o início da fila deve ser removido e colocado na coleção de blocos livres e a fila deve ser ajustada para mostrar a remoção:

$$\begin{aligned} used' &= used \setminus head\ BlockQueue \wedge \\ free' &= free \cup head\ BlockQueue \wedge \\ BlockQueue' &= tail\ BlockQueue \end{aligned}$$

Uma convenção usada em muitos métodos formais é que o valor de uma variável após uma operação recebe uma indicação por meio de um apóstrofo. Portanto, o primeiro componente da expressão anterior diz que os novos blocos usados (*used'*) será igual aos blocos usados antigos menos os blocos removidos. O segundo componente diz que os novos blocos livres (*free'*) serão os blocos livres antigos com o início da fila de blocos acrescentado a eles. O terceiro componente diz que a nova fila de blocos será igual à cauda do valor antigo da fila de blocos, isto é, todos os elementos na fila exceto o primeiro. Uma segunda operação adiciona uma coleção de blocos, *Ablocks*, à fila de blocos. A precondição é que *Ablocks* seja correntemente um conjunto de blocos usados:

$$Ablocks \subseteq used$$

A pós-condição é que o conjunto de blocos seja acrescentado ao fim da fila de blocos e que o conjunto de blocos usados e livres permaneça inalterado:

$$\begin{aligned} BlockQueue' &= BlockQueue \smallfrown (Ablocks) \wedge \\ used' &= used \wedge \\ free' &= free \end{aligned}$$

Não há dúvida de que a especificação matemática da fila de blocos é consideravelmente mais rigorosa do que uma narrativa em linguagem natural ou um modelo gráfico. O rigor adicional requer esforço, mas os benefícios obtidos da melhor consistência e completude podem ser justificados para alguns domínios de aplicação.

Como
represento
pré e pós-
condições?

21.7 LINGUAGENS DE ESPECIFICAÇÃO FORMAL

Uma linguagem de especificação formal em geral é composta por três componentes primários: (1) uma sintaxe que define a notação específica com a qual a especificação é representada, (2) semânticas para ajudar a definir um “universo de objetos” [Win90] que será usado para descrever o sistema, e (3) um conjunto de relações que definem as regras que indicam quais objetos satisfazem adequadamente à especificação.

O domínio sintático de uma linguagem de especificação formal muitas vezes é baseado em sintaxe derivada da notação padrão de teoria dos conjuntos e do cálculo de predicado. O *domínio semântico* de uma linguagem de especificação indica como a linguagem representa os requisitos do sistema.

É possível usar diferentes abstrações semânticas para descrever o mesmo sistema de distintas maneiras. Fizemos isso de modo menos formal nos Capítulos 6 e 7; informações, funções e comportamentos foram representados. Diferentes notações de modelagem podem ser usadas para representar o mesmo sistema. As semânticas de cada representação proporcionam visões complementares do sistema. Para ilustrar essa abordagem quando são usados métodos formais, suponha que se utilize uma linguagem de especificação formal para descrever o conjunto de eventos responsável por um estado particular em um sistema. Outra relação formal mostra todas as funções que ocorrem em determinado estado. A intersecção dessas duas relações proporciona a indicação dos eventos responsáveis por funções específicas.

Há em uso hoje uma variedade de linguagens de especificação formal. OCL [OMG03b], Z [ISO02], LARCH [Gut93] e VDM [Jon91] são linguagens de especificação formal representativas que possuem as características já mencionadas. Neste capítulo, uma breve discussão da OCL e Z é apresentada.

TABELA 21.1 RESUMO DA NOTAÇÃO OCL FUNDAMENTAL

$x.y$	Obtém a propriedade y do objeto x . Uma propriedade pode ser um atributo, o conjunto de objetos no fim de uma associação, o resultado da avaliação de uma operação ou outros itens dependendo do tipo de diagrama UML. Se x for um Set, y será aplicado a todo elemento de x ; os resultados são reunidos em um novo Set.
$c \rightarrow f()$	Aplica a operação f interna de OCL à própria Collection c (ao contrário de cada um dos objetos em c). A seguir estão listados exemplos de operações nativas.
$\text{and}, \text{or}, =, <>$	Operações lógicas and , or , equals , not-equals .
$p \text{ implies } q$	Verdadeiro se q for verdadeiro ou p for falso.

Exemplos de operações sobre Collection (incluindo Sets e Sequences)

$C \rightarrow \text{size}()$	O número de elementos na Collection c .
$C \rightarrow \text{isEmpty}()$	Verdadeiro se c não tiver elementos, falso caso contrário.
$c1 \rightarrow \text{includesAll}(c2)$	Verdadeiro se todo elemento de $c2$ for encontrado em $c1$.
$c1 \rightarrow \text{excludesAll}(c2)$	Verdadeiro se nenhum elemento de $c2$ for encontrado em $c1$.
$C \rightarrow \text{forAll}(\text{elem} \mid \text{boolexpr})$	Verdadeiro se boolexpr for verdadeira quando aplicada a todo elemento de c . Quando um elemento está sendo avaliado, ele é ligado ao elemento variável, que pode ser usado em boolexpr . Isso implementa a quantificação universal, discutida anteriormente.
$C \rightarrow \text{forAll}(\text{elem1}, \text{elem2} \mid \text{boolexpr})$	O mesmo que o anterior, exceto que boolexpr é avaliada para todo possível par de elementos tomados de c , incluindo casos em que o par consiste no mesmo elemento.
$C \rightarrow \text{isUnique}(\text{elem} \mid \text{expr})$	Verdadeiro se expr é avaliada a um valor diferente quando aplicada a cada elemento de c .

Exemplos de operações em Sets

$s1 \rightarrow \text{intersection}(s2)$	O conjunto daqueles elementos encontrados em $s1$ e também em $s2$.
$s1 \rightarrow \text{union}(s2)$	O conjunto daqueles elementos encontrados em $s1$ ou $s2$.
$s1 \rightarrow \text{excluding}(x)$	O conjunto $s1$ com o objeto x omitido.

Exemplo de operação específica para Sequences

$\text{Seq} \rightarrow \text{first}()$	O objeto que é o primeiro elemento na sequência seq .
---	--

21.7.1 Object Constraint Language (OCL)⁸

Object Constraint Language (OCL) é uma notação formal desenvolvida de forma que os usuários da UML possam adicionar mais precisão às suas especificações. Todos os poderes da lógica e da matemática discreta estão disponíveis na linguagem. No entanto, os projetistas da OCL decidiram que apenas os caracteres ASCII (em vez da notação matemática convencional) deveriam ser usados em instruções OCL. Isso torna a linguagem mais amigável àqueles menos inclinados à matemática e mais facilmente processada pelo computador. Mas isso torna a OCL um pouco prolixa em certas ocasiões.

Para usar a OCL, começa-se com um ou mais diagramas UML — mais comumente diagramas de classe, estado ou atividade (Apêndice 1). São acrescentadas as expressões OCL e declaram-se fatos sobre elementos dos diagramas. Essas expressões são chamadas de *restrições* (*constraints*); qualquer implementação derivada do modelo deve assegurar que cada uma das restrições permaneça sempre verdadeira.

⁸ Essa seção teve a contribuição do professor Timothy Lethbridge, da Universidade de Ottawa, e foi apresentada aqui com sua permissão.

Como uma linguagem de programação orientada a objeto, uma expressão OCL envolve operadores operando sobre objetos. No entanto, o resultado de uma expressão completa deve ser sempre booleano, isto é, verdadeiro ou falso. Os objetos podem ser instâncias da classe **Collection** da OCL, da qual **Set** e **Sequence** são duas subclasses.

O objeto **self** é o elemento do diagrama UML em cujo contexto a expressão OCL está sendo avaliada. Outros objetos podem ser obtidos por meio de *navegação* usando o símbolo **.** (dot) do objeto **self**. Por exemplo:

- Se **self** é a classe **C**, com atributo **a**, **self.a** avalia para objeto armazenado em **a**.
- Se **C** tem uma associação um-para-muitos chamada de *assoc* com outra classe **D**, **self.assoc** avalia para um **Set** cujos elementos são do tipo **D**.
- Por fim (e um pouco mais sutilmente), se **D** tem atributo **b**, a expressão **self.assoc.b** avalia para o conjunto de todos os **bs** pertencentes a todos os **Ds**.

A OCL proporciona operações internas implementando operadores set e logic, especificação construtiva e matemática relacionada. Um pequeno exemplo é apresentado na Tabela 21.1.

Para ilustrar o uso da OCL na especificação, reexaminamos o exemplo do tratador de blocos, introduzido na Seção 21.5. O primeiro passo é desenvolver um modelo UML (Figura 21.9). Esse diagrama de classe especifica muitas relações entre os objetos envolvidos. No entanto, são acrescentadas expressões OCL para permitir que os implementadores do sistema saibam mais precisamente o que deve permanecer verdadeiro enquanto o sistema processa.

As expressões OCL que suplementam o diagrama de classes correspondem às seis partes do invariante discutido na Seção 21.5. No exemplo a seguir, o invariante é repetido em inglês e então é escrita a expressão OCL correspondente. É aconselhável inserir texto de linguagem natural com a lógica formal; isso ajuda a entender a lógica e auxilia os revisores a descobrir os erros, por exemplo, situações em que não há correspondência entre o inglês e a lógica.

1. Nenhum bloco será marcado como não usado e usado.

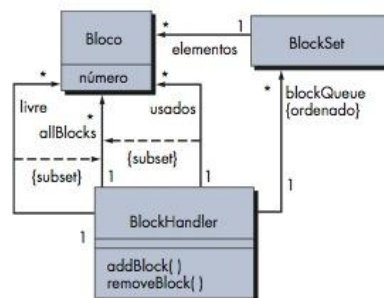
context BlockHandler inv:

(self.used -> intersection(self.free)) -> isEmpty()

Note que cada expressão começa com a palavra-chave **context**. Isso indica o elemento do diagrama UML que a expressão restringe. Como alternativa, você poderia colocar a restrição diretamente no diagrama UML, entre chaves {}. A palavra-chave **self** aqui se refere à instância de **BlockHandler**; no próximo item, como permite a OCL, vamos omitir o **self**.

FIGURA 21.9

Diagrama de classe para um tratador de blocos



2. Todos os conjuntos de blocos mantidos na fila serão subconjuntos da coleção de blocos usados correntemente.

```
context BlockHandler inv:
  blockQueue -> forAll(aBlockSet | used -> includesAll(aBlockSet ))
```

3. Nenhum elemento da fila terá o mesmo número de bloco.

```
context BlockHandler inv:
  blockQueue -> forAll(blockSet1, blockSet2 |
    blockSet1 <> blockSet2 implies
    blockSet1.elements.number -> excludesAll(blockSet2.elements.number))
```

A expressão antes de **implies** é necessária para assegurar que ignoramos pares nos quais ambos os elementos são o mesmo bloco.

4. A coleção de blocos usados e blocos que não são usados será a coleção total de blocos que compõem os arquivos.

```
context BlockHandler inv:
  allBlocks = used -> union(free)
```

5. A coleção de blocos não usados não terá números de bloco duplicados.

```
context BlockHandler inv:
  free -> isUnique(aBlock | aBlock.number)
```

6. A coleção de blocos usados não terá números de bloco duplicados.

```
context BlockHandler inv:
  used -> isUnique(aBlock | aBlock.number)
```

A OCL também pode ser utilizada para especificar pré e pós-condições de operações. Por exemplo, a notação a seguir descreve operações que removem e acrescentam conjuntos de blocos à fila. Observe que a notação **x@pre** indica o objeto **x** da maneira como ele existe *antes* da operação; isso é o oposto da notação matemática discutida anteriormente, em que é o **x** *depois* da operação que é especialmente designado (como **x'**).

```
context BlockHandler::removeBlocks()
pre: blockQueue -> size() > 0
post: used = used@pre-blockQueue@pre -> first() and
      free = free@pre -> union(blockQueue@pre -> first()) and
      blockQueue = blockQueue@pre -> excluding(blockQueue@pre -> first)

context BlockHandler::addBlocks(aBlockSet :BlockSet)
pre: used -> includesAll(aBlockSet.elements)
post: (blockQueue.elements = blockQueue.elements@pre
      -> append (aBlockSet.elements) and
      used = used@pre and
      free = free@pre
```

A OCL é uma linguagem de modelagem, mas tem todos os atributos de uma linguagem formal. A OCL permite a expressão de várias restrições, pré e pós-condições, proteções e outras características relacionadas com objetos representados em vários modelos UML.

21.7.2 A linguagem de especificação Z

Z (que se pronuncia como “zed”) é uma linguagem de especificação amplamente usada na comunidade de métodos formais. A linguagem Z aplica conjuntos tipados, relações e funções no contexto da lógica de predicados de primeira ordem para criar *esquemas* — um meio de estruturar a especificação formal.

As especificações Z são organizadas como um conjunto de esquemas — uma estrutura de linguagem que introduz variáveis e especifica a relação entre essas variáveis. Um esquema é essencialmente a especificação formal análoga do componente de linguagem de programação. Esquemas são usados para estruturar uma especificação formal da mesma maneira que os componentes são utilizados para estruturar um sistema.

Um esquema descreve os dados armazenados que um sistema acessa e altera. No contexto de Z, isso é chamado de “estado”. O uso do termo *estado* em Z é ligeiramente diferente do empregado no restante do livro.⁹ Além disso, o esquema identifica as operações aplicadas para mudar o estado e as relações que ocorrem no sistema. A estrutura genérica de um esquema toma a forma:

```

Nome do esquema
declarações

invariante

```

em que declarações identificam as variáveis que formam o estado do sistema e a invariante impõe restrições sobre a maneira pela qual o estado pode evoluir. A Tabela 21.2 apresenta um resumo da notação da linguagem Z.

O exemplo a seguir de um esquema descreve o estado de um tratador de blocos e a invariante de dados:

```

BlockHandler
used, free :  $\mathbb{P} \text{ BLOCKS}$ 
BlockQueue : seq  $\mathbb{P} \text{ BLOCKS}$ 
used  $\cap$  free =  $\emptyset \wedge$ 
used  $\cup$  free = AllBlocks  $\wedge$ 
 $\forall i: \text{dom BlockQueue} \bullet \text{BlockQueue } i \subseteq \text{used} \wedge$ 
 $\forall i, j: \text{dom BlockQueue} \bullet i \neq j \Rightarrow \text{BlockQueue } i \cap \text{BlockQueue } j = \emptyset$ 

```

Conforme observamos, o esquema consiste em duas partes. A parte acima da linha central representa as variáveis do estado, enquanto a parte abaixo da linha central descreve a invariante de dados. Sempre que o esquema especificar operações que mudam o estado, ele é precedido pelo símbolo (Δ). O exemplo a seguir de um esquema descreve a operação que remove um elemento da fila de blocos:

```

RemoveBlocks
 $\Delta \text{ BlockHandler}$ 

#BlockQueue > 0,
used' = used  $\setminus$  head BlockQueue  $\wedge$ 
free' = free  $\cup$  head BlockQueue  $\wedge$ 
BlockQueue' = tail BlockQueue

```

⁹ Lembre-se de que em outros capítulos, *estado* foi empregado para identificar um modo de comportamento observável externamente para um sistema.

TABELA 21.2 RESUMO DA NOTAÇÃO Z

A notação Z baseia-se na teoria de conjuntos típica e na lógica de primeira ordem. Z proporciona uma construção, chamada de esquema, para descrever o espaço de estado e operações de uma especificação. Um esquema agrupa declarações de variáveis com uma lista de predicados que restringem o valor possível de uma variável. Em Z, o esquema X é definido pela forma

X
declarações
predicados

Funções globais e constantes são definidas pela forma

declarações
predicados

A declaração fornece o tipo da função ou constante, enquanto o predicado dá seu valor. Nesta tabela há apenas um conjunto abreviado de símbolos Z.

Conjuntos:

$S : P X$	S é declarada como um conjunto de X s.
$x \in S$	x é um membro de S .
$x \notin S$	x não é um membro de S .
$S \subseteq T$	S é um subconjunto de T : todo membro de S está também em T .
$S \cup T$	A união de S e T : contém todos os membros de S ou T ou ambos.
$S \cap T$	A intersecção de S e T : contém todos os membros de S e T .
$S \setminus T$	A diferença de S e T : contém todos os membros de S exceto aqueles que também estão em T .
\emptyset	Conjunto vazio: não contém membros.
$\{x\}$	Conjunto unitário: contém somente x .
\mathbb{N}	O conjunto de números naturais $0, 1, 2, \dots$
$S : F X$	S é declarado como um conjunto finito de X s.
$\max(S)$	O máximo do conjunto não vazio de números S .

Funções:

$f: X \mapsto Y$	f é declarada como uma injeção parcial de X para Y .
$\text{dom } f$	O domínio de f : o conjunto de valores x para os quais $f(x)$ é definida.
$\text{ran } f$	O intervalo de f : o conjunto de valores tomados por $f(x)$ quando x varia sobre o domínio de f .
$f \oplus \{x \mapsto y\}$	Uma função que concorda com f exceto que x é mapeado para y .
$\{x\} \triangleleft f$	Uma função como f , exceto que x é removida de seu domínio.

Lógica:

$P \wedge Q$	P e Q : é verdadeira se P e Q forem ambos verdadeiros.
$P \Rightarrow Q$	P implica Q : é verdadeira se Q for verdadeiro ou P for falso.
$\theta S' = \theta S$	Nenhum componente do esquema S muda em uma operação.

A inclusão do Δ *BlockHandler* resulta que todas as variáveis que formam o estado tornam-se disponíveis para o esquema *RemoveBlocks* e assegura que a invariante de dados será mantida antes e depois que a operação tiver sido executada.

A segunda operação, que acrescenta uma coleção de blocos ao fim da fila, é representada por

—AddBlocks—

Δ *BlockHandler*
 Ablocks? : BLOCKS

$Ablocks? \subseteq used$
 $BlockQueue' = BlockQueue \langle Ablocks? \rangle \wedge$
 $used' = used \wedge$
 $free' = free$

Por convenção em Z, uma variável de entrada que é lida, mas não forma parte do estado, é encerrada por um ponto de interrogação. Assim, Ablocks?, que age como um parâmetro de entrada, é encerrada por um ponto de interrogação.



Métodos formais

Objetivo: o objetivo das ferramentas de métodos formais é ajudar uma equipe de software na especificação e verificação da correção.

Mecânica: a mecânica varia. Em geral, as ferramentas ajudam na especificação de uma prova automática da correção, usualmente definindo uma linguagem especializada para prova de teorema. Muitas ferramentas não são comercializadas e foram desenvolvidas para fins de pesquisa.

FERRAMENTAS DO SOFTWARE

Ferramentas representativas:¹⁰

ACL2, desenvolvida na Universidade do Texas (www.cs.utexas.edu/users/moore/acl2/), é "tanto uma linguagem de programação na qual você pode modelar sistemas de computadores quanto uma ferramenta para ajudá-lo a provar propriedades daqueles modelos".

EVES, desenvolvida pela ORA Canada (www.ora.on.ca/eves.html), implementa a linguagem Verdi para especificação formal e é um gerador automático de provas.

Uma extensa lista de mais de 90 ferramentas de métodos formais pode ser encontrada em <http://vl.fhnet.info/>.

21.8 RESUMO

A engenharia de software sala limpa é uma abordagem formal para o desenvolvimento de software que pode levá-lo a uma qualidade notavelmente alta. Ela usa a especificação de estrutura de caixa para análise e modelagem de projeto e dá ênfase à verificação da correção, em vez do teste, como mecanismo primário para localizar e remover erros. O teste de uso estatístico é aplicado para desenvolver as informações necessárias de taxa de falhas para certificar a confiabilidade do software fornecido.

A abordagem sala limpa começa com a análise e modelos de projeto que usam uma representação de estrutura de caixa. Uma "caixa" encapsula o sistema (ou algum aspecto do sistema) a um nível específico de abstração. Caixas-pretas são usadas para representar o comportamento de um sistema observável externamente. Caixas de estado encapsulam dados de estado e operações. Caixas-claras são usadas para modelar o projeto procedimental inferido pelos dados e operações de uma caixa de estado.

¹⁰ As ferramentas aqui apresentadas não significam um aval, mas sim uma amostra dessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

A verificação de correção é aplicada uma vez completo o projeto da estrutura de caixa. O projeto procedimental para um componente de software é particionado em uma série de subfunções. Para provar a correção das subfunções, definem-se condições de saída para cada subfunção e aplica-se uma série de subprovas. Se cada condição de saída é satisfeita, o projeto deve estar correto.

Uma vez completada a verificação da correção, inicia-se o teste estatístico de uso. Diferentemente do teste convencional, a engenharia de software sala limpa não enfatiza o teste de unidade ou de integração. Em vez disso, o software é testado definindo-se um conjunto de cenários de uso, determinando a probabilidade de uso para cada cenário e, por fim, definindo testes aleatórios que se sujeitam às probabilidades. Os registros de erros resultantes são combinados com os modelos de amostragem, componente e certificação para possibilitar a computação matemática da confiabilidade projetada para o componente de software.

Os métodos formais usam recursos descritivos da teoria dos conjuntos e notação lógica para possibilitar ao engenheiro de software criar uma definição clara dos fatos (requisitos). Os conceitos subjacentes que governam os métodos formais são: (1) a invariante de dados, uma condição verdadeira através de toda a execução do sistema que contém uma coleção de dados; (2) o estado, uma representação do modo de comportamento do sistema observável externamente ou (em Z e linguagens relacionadas) os dados armazenados que um sistema acessa e altera; e (3) a operação, uma ação que tem lugar em um sistema e lê ou escreve dados para um estado. A operação está associada a duas condições: uma pré e pós-condição.

Será que a engenharia de software sala limpa ou os métodos formais serão algum dia usados amplamente? A resposta é "provavelmente não". Eles são mais difíceis de aprender do que os métodos de engenharia de software convencionais e representam um "choque cultural" significativo para alguns profissionais. Mas na próxima vez em que você ouvir alguém se lamentando, "Por que não podemos escrever esse software certo já na primeira vez?", saberá que há técnicas que podem ajudá-lo a conseguir exatamente isso.

PROBLEMAS E PONTOS A PONDERAR

21.1. Se você tivesse de escolher um aspecto da engenharia de software sala limpa que a torna radicalmente diferente das abordagens de engenharia de software convencionais orientadas a objeto, qual seria ele?

21.2. Como um modelo de processo incremental e certificação funcionam em conjunto para produzir software de alta qualidade?

21.3. Por meio da especificação de estrutura de caixa, desenvolva modelos de análise e projeto "primeira-passada" para o sistema *CasaSegura*.

21.4. Um algoritmo de ordenação da bolha (bubble-sort) é definido da seguinte maneira:

```

procedure bubblesort;
var i, t, integer;
begin
  repeat until = 5 a[1]
    t := a[1];
    for j := 2 to n do
      if a[j-1] > a[j] then begin
        t := a[j-1];
        a[j-1] := a[j];
        a[j] := t;
      end
    endrep
  end

```

Particione o projeto em subfunções e defina um conjunto de condições que lhe possibilitariam provar que o algoritmo está correto.

21.5. Documente uma prova de verificação de correção para o bubble sort discutido no Problema 21.4.

21.6. Selecione um programa que você use regularmente (por exemplo, um programa de e-mail, um processador de texto, um programa de planilha). Crie um conjunto de cenários de uso para o programa. Defina a probabilidade do uso de cada cenário e depois desenvolva uma tabela de distribuição de probabilidades e estímulos de programa similar à apresentada na Seção 21.4.1.

21.7. Para a tabela de distribuição de estímulos e probabilidades de programa desenvolvida no Problema 21.6, use um gerador de números aleatórios para desenvolver um conjunto de caso de testes para um teste estatístico de uso.

21.8. Descreva o objetivo da certificação no contexto de engenharia de software sala limpa.

21.9. Você foi designado para uma equipe que está desenvolvendo software para um fax modem. A sua tarefa é desenvolver a parte da “lista telefônica” da aplicação. A função lista telefônica permite armazenar até *MaxNomes* de pessoas associadas ao nome da empresa, o número do fax e outras informações relacionadas. Usando linguagem natural, defina

- a. a invariante de dados.
- b. o estado.
- c. as operações possíveis.

21.10. Você foi designado para uma equipe de software que está desenvolvendo um programa chamado MemoryDoubler, que fornece para o PC uma memória aparentemente maior do que a memória física. Isso é conseguido identificando, coletando e reatribuindo blocos de memória que foram atribuídos a uma aplicação existente, mas que não estão sendo usados. Os blocos não usados são reatribuídos a aplicações que requerem memória adicional. Adotando as hipóteses apropriadas e usando linguagem natural, defina

- a. a invariante de dados.
- b. o estado.
- c. as operações possíveis.

21.11. Usando a notação OCL ou Z apresentadas na Tabela 21.1 ou 21.2, selecione alguma parte do sistema de segurança *CasaSegura* descrita anteriormente neste livro e tente especificá-la com OCL ou Z.

21.12. Usando uma ou mais das fontes de informações citadas nas referências deste capítulo ou em *Leituras e fontes de informação complementares*, desenvolva uma apresentação de meia hora sobre a sintaxe e semânticas básicas de uma linguagem de especificação formal que não seja OCL ou Z.

LEITURAS E FONTES DE INFORMAÇÃO COMPLEMENTARES

Nos anos recentes publicaram-se relativamente poucos livros sobre técnicas avançadas de especificação e verificação. No entanto, vale considerar algumas novas adições à literatura. Um livro editado por Gabbar (*Modern Formal Methods and Applications*, Springer, 2006) apresenta fundamentos, novos desenvolvimentos e aplicações avançadas. Jackson (*Software Abstractions*, The MIT Press, 2006) traz todos os fundamentos básicos e uma abordagem que ele chama de “métodos formais leves”. Monin e Hinchey (*Understanding Formal Methods*, Springer, 2003) proporcionam uma excelente introdução ao assunto. Butler e outros editores (*Integrated Formal Methods*, Springer, 2002) apresentam uma variedade de trabalhos sobre tópicos de métodos formais.

Além dos livros citados neste capítulo, Prowell e seus colegas (*Cleanroom Software Engineering: Technology and Process*, Addison-Wesley, 1999) fornecem um tratamento profundo de todos os aspectos importantes da abordagem sala limpa. Discussões úteis dos tópicos sala limpa foram editadas por Poore e Trammell (*Cleanroom Software Engineering: A Reader*, Blackwell Publishing, 1996). Becker e Whittaker (*Cleanroom Software Engineering Practices*, Idea Group Publishing, 1996) apresentam uma excelente visão geral para aqueles que não estão familiarizados com as práticas sala limpa.

O Cleanroom Pamphlet (Software Technology Support Center, Hill AF Base, April 1995) contém reimpressões de vários artigos importantes. The Data and Analysis Center for Software (DACS) (www.dacs.dtic.mil) fornece muitas publicações úteis, guias e outras fontes de informação sobre engenharia de software sala limpa.

A verificação de projeto através da prova de correção está no coração da abordagem sala limpa. Livros de Cupillari (*The Nuts and Bolts of Proofs*, 3d ed., Academic Press, 2005), Solow (*How to Read and Do Proofs*, 4th ed., Wiley, 2004), Eccles (*An Introduction to Mathematical Reasoning*, Cambridge University Press, 1998) fornecem excelentes introduções aos fundamentos básicos matemáticos. Staveland (*Toward Zero-Defect Software*, Addison-Wesley, 1998), Baber (*Error-Free Software*, Wiley, 1991) e Schulmeyer (*Zero Defect Software*, McGraw-Hill, 1990) discutem prova de correção em detalhe considerável.

No domínio dos métodos formais, livros de Casey (*A Programming Approach to Formal Methods*, McGraw-Hill, 2000), Hinchey e Bowan (*Industrial Strength Formal Methods*, Springer-Verlag, 1999), Hussmann (*Formal Foundations for Software Engineering Methods*, Springer-Verlag, 1997) e Sheppard (*An Introduction to Formal Specification with Z and VDM*, McGraw-Hill, 1995) proporcionam diretrizes úteis. Além disso, livros específicos sobre linguagem como Warmer and Kleppe (*Object Constraint Language*, Addison-Wesley, 1998), Jacky (*The Way of Z: Practical Programming with Formal Methods*, Cambridge University Press, 1997), Harry (*Formal Methods Fact File: VDM and Z*, Wiley, 1997) e Cooper e Barden (*Z in Practice*, Prentice-Hall, 1995) fornecem informações úteis para métodos formais, bem como uma variedade de linguagens de modelagem.

Uma ampla variedade de fontes de informações sobre engenharia de software sala limpa e métodos formais está disponível na Internet. Uma lista atualizada das referências relevantes na Web sobre modelagem formal e verificação pode ser encontrada no site www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm.