planos de processos que aumentem a produtividade dos desenvolvedores enquanto reduzem a ocorrência de falhas no desenvolvimento.
Até este momento a definição de software usada se refere à noção de plano de construção de máquina. Será esta a definição mais aceita para software entre os praticantes? Vejamos abaixo outra definição de programa (software) presente em uma licença de software da IBM (4).
'... O termo "programa" significa o programa original e todas as cópias completas ou parciais do mesmo. Um programa consiste em instruções legíveis por máquina, seus componentes, dados, conteúdo audiovisual (tal como imagens, texto, gravações ou figuras) e materiais licenciados relacionados.'.
Conforme esta definição, em geral adotada pela indústria de software, um software é mais do que as instruções interpretáveis por uma máquina. Digno de nota é a indicação de que conteúdo audiovisual (tal como imagens, texto, gravações ou figuras) tambem é parte do software - este aspecto extrapola o conceito de software inicialmente apresentado como meta-máquina, à medida que torna explícito o fato de que qualquer material escrito, impresso, apresentável em qualquer mídia de comunicação, de natureza textual, gráfica, audível, etc, que tem por objetivo descrever algo para o usuário ou sua máquina, também é parte do software.
Outra definição de software comumente aceita entre quem desenvolve software é a que prescreve que o resultado de quaisquer das atividades do processo produtivo de software também é software. Além de todas as mídias digitais, impressas, ou reproduzíveis de alguma forma, que foram reproduzidas e entregues ao cliente, são parte do software os subprodutos internos do processo produtivo, como planos de decomposições de software, especificações de linguagens, definições de prazos e custos limites, planos de testes, documentos formais de aceitação, etc. Cada um dos artefatos é parte de um plano de construção, não necessariamente compreensível por uma máquina computável, mas destinado a ser interpretado por um ser humano que participa da construção e evolução do software. Sendo assim, uma possível eliminação desses artefatos (ou do conhecimento neles contidos) de dentro da composição do software, sempre provoca prejuízos no processo de manutenção do mesmo.

Conclusões A prática do software emerge da interação entre múltiplos agentes coletivos, com interesses e necessidades distintas, que contribuem com pontos de vista complementares para usar e criar máquinas, linguagens e planos de construção de máquinas. Embora a satisfação primária provocada pelo uso do software seja resultante do efeito imediato de uma relação mecânica de interpretação efetuada por uma máquina computável, o contexto histórico-social-lingüístico de concepção do software o redefine como um artefato modularizado, interdependente e hierarquizado, constituído por mídias de diversas naturezas, concebidas por uma ampla gama de seres humanos com habilidades profissionais extremamente variadas, e destinadas não só à interpretação por máquinas computáveis, mas também por seres humanos.

Jorge Henrique Cabral Fernandes *é professor do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte*

Referências bibliográficas

1 Dawkins, R. *Clibimg mount improbable*. W.W Norton, 1996.
2 Lehmam,M.M. *Laws of software evolution revisited*. LNCS 1149, Springer, 1997.
3 SWEBOK Project. *Software engineering body of knowledge*. Ed. 0.9. 2001.
4 IBM software license

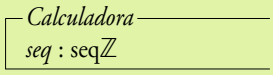
Refinamento: a essência da engenharia de software

Ana Cavalcanti

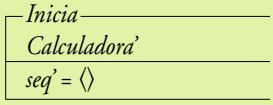
A noção de refinamento captura a essência das atividades diárias de engenheiros de software, que projetam sistemas baseados em especificações, e de programadores, que implementam estes projetos. Em ambos os casos, o principal objetivo é a construção de sistemas e programas de acordo com documentos que os definem. O produto final, acima de tudo, deve ser, ou tem que ser, correto.
Refinamento é a relação que existe entre uma especificação, seus projetos e implementações corretas, do ponto de vista funcional. Métodos de desenvolvimento de programa são baseados nesta noção de uma forma ou de outra. Uma técnica formal vai além, no sentido que ela provê uma base matemática para garantia de correção. Neste caso, a meta primordial é o refinamento de uma especificação inicial para obtenção de uma implementação aceitável. Critérios de aceitação podem incluir, por exemplo, eficiência, mas a garantia fornecida é que a especificação e a implementação estão relacionadas por refinamento.

Conceitos Básicos Inicialmente, refinamento foi estudado para programas seqüenciais, aonde o foco é a relação entre as entradas e saídas de um programa. Foi identificado que há basicamente duas formas de refinar uma especificação. A primeira é a introdução e a transformação de estruturas de programação e controle como atribuições, condições, e laços. Isto é chamado refinamento algorítmico.
A segunda forma de refinamento é relacionada com as estruturas de dados usadas no programa. Sistemas são especificados em termos de tipos de dados que são apropriados para descrever propriedades do domínio de aplicação; neste estágio do desenvolvimento, não se faz, por exemplo, considerações relacionadas à eficiência.
Decisões de projeto, no entanto, normalmente introduzem estruturas de dados mais elaboradas e apropriadas para implementação. A mudança de representação de dados envolvida nessa tarefa é chamada refinamento de dados.
O ponto de partida de qualquer método formal é uma especificação formal. Correção é uma noção relativa: dizemos que um programa é correto se ele implementa a sua especificação. Para garantir correção, nós precisamos de uma especificação formal do programa.
Há muitas linguagens e formalismos em uso hoje. Nós usaremos uma linguagem de especificação chamada Z para apresentar um exemplo. Uma especificação de sistema em Z consiste basicamente de uma definição de um estado e de uma coleção de operações. O estado é composto de variáveis que representam os dados usados e registrados no sistema. As operações recebem entradas e produzem saídas, possivelmente alterando o estado.
Tanto o estado quanto as operações são definidas por esquemas: uma notação gráfica para agrupar declarações de variáveis e suas propriedades.

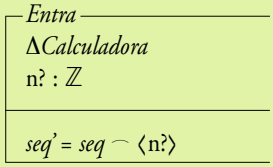
Exemplo Nós apresentamos a especificação de um sistema que calcula a média de uma seqüência de números recebidos como entrada. O estado deste sistema só tem um componente: a seqüência de inteiros.



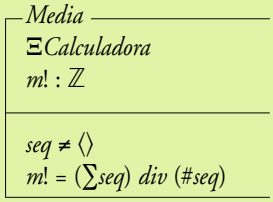
O nome do estado, *Calculadora* , é definido, e o seu único componente é declarado.
Este sistema tem três operações. A primeira, *Inicia*, inicializa o estado.



A referência à *Calculadora* indica que esta é uma operação sobre este estado. Em uma definição de operação, nós podemos nos referir à *seq*, que representa o valor do componente de estado antes da operação, e à *seq'*, o valor depois da operação. A definição de *Inicia* especifica que, inicialmente, *seq* é seqüência vazia: não foi dado entrada em nenhum número.
A segunda operação, *Entra*, registra um valor *n*? recebido como entrada.

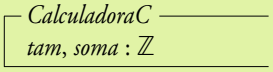


O Δ na frente de *Calculadora* indica que *Entra* muda o estado. A variável *n*? representa o número recebido como entrada. A nova seqüência *seq'* pode ser obtida pela inserção de *n*? ao final de *seq*; ⌢ é o operador de concatenação de seqüências e ⟨*n*?⟩ é a seqüência que só contêm *n*?.
A última operação, *Media*, calcula a média dos números recebidos como entrada até então.

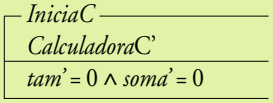


O Ξ indica que *Media* não muda o estado. A saída é representada pela variável *m*!. A especificação requer que a seqüência *seq* não seja vazia. Esta é a **pré-condição** da operação: se *Media* for executada quando esta condição não for satisfeita, seu resultado não é previsível. Se, no entanto, a pré-condição for satisfeita, a especificação de *Media* requer que a saída seja a soma dos elementos da seqüência dividida pelo seu tamanho.
Especificar um sistema é o primeiro passo para obtenção de uma implementação correta. Um método de desenvolvimento formal usa a especificação como uma base para a produção de uma implementação correta, que refina a especificação.
Refinamento é baseado na idéia de que uma especificação é um contrato entre o cliente e o desenvolvedor. O cliente não pode reclamar se, quando executadas em estados que satisfazem as suas pré-condições, as operações da implementação produzem saídas que satisfazem as propriedades estabelecidas na especificação. Neste caso, nós temos uma implementação correta.

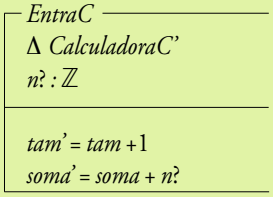
Refinamento de dados Tipicamente, a primeira oportunidade de refinamento é a mudança de representação dos componentes de estado. A especificação descreve a relação entre as entradas e as saídas, quando o sistema é inicializado e uma seqüência de operações é executada. Os valores dos componentes de estado, no entanto, não são visíveis.
No nosso exemplo, nós usamos uma seqüência para registrar os números recebidos; esta é uma forma natural de descrever o sistema. Entretanto, nós podemos economizar memória se nós registrarmos apenas a soma e o número de inteiros recebidos como entrada. Se as operações forem modificadas apropriadamente, esta é uma mudança de representação de estado válida.
Este tipo de mudança é conhecida como refinamento de dados; a especificação original é considerada abstrata e a nova especificação, concreta. Na verdade, a especificação concreta é um projeto do sistema, em que estruturas de dados mais apropriadas para a implementação são introduzidas.
A outra oportunidade de refinamento é o desenvolvimento de implementações para as operações. Como estas implementações são afetadas pelas mudanças no estado, nós consideramos refinamento de dados primeiro. Neste estágio, nós mudamos as operações apenas para adaptá-las ao novo tipo de dados. Em Z, nós escrevemos a especificação concreta no mesmo estilo usado para a especificação abstrata.
Exemplo O estado concreto sugerido acima pode ser definido como mostrado abaixo.



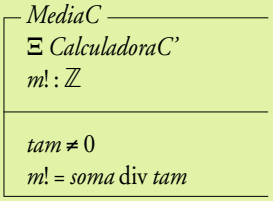
Há dois componentes: o tamanho *tam* da seqüência de números recebidos como entrada, e a sua *soma*.
As novas definições das operações são apresentadas a seguir. A inicialização, *IniciaC*, registra que nenhum número foi recebido.



A operação *EntraC,* que registra um número *n*? recebido como entrada, incrementa *tam* e adiciona a entrada a *soma*



A operação que calcula a média tem uma especificação muito mais simples.



Os valores necessário estão prontamente disponíveis em *soma* e *tam* .
Depois de prover a especificação concreta, nós temos que provar que ela satisfaz a propriedade mencionada anteriormente: os clientes que contrataram uma implementação da especificação abstrata não podem reclamar se receberem uma implementação da especificação concreta. A técnica mais utilizada para realização dessa prova é conhecida como **simulação**. Ela envolve a definição de uma relação entre os estados abstratos e concretos que especifica como a informação no estado abstrato é representada no estado concreto. Em Z, esta relação é conhecida como relação de recuperação.
Para o nosso exemplo, a relação apropriada pode ser especificada como segue.

Recupera
Calculadora
CalculadoraC

$$tam = \#\, seq \land soma = \sum seq$$

A inclusão dos estados abstrato e concreto reflete o fato que nós estamos especificando uma relação entre eles. Basicamente, um estado concreto está relacionado a um estado abstrato quando o valor de *tam* é de fato o tamanho (operador #) de *seq*, e *soma* é a soma dos números nesta seqüência.
Dada uma relação de recuperação, nós precisamos primeiro verificar que a nova inicialização é adequada: dado um estado inicial concreto, há um estrado abstrato inicial correspondente. No nosso exemplo, como o tamanho e a soma dos elementos da seqüência vazia é 0, nós temos o resultado requerido. Na prática, este resultado é formalizado através de um teorema matemático, que pode ser provado usando ferramentas computacionais apropriadas para Z.
Nós também precisamos provar que as operações da especificação concreta estão de acordo com as operações correspondentes da especificação abstrata. Há dois pontos que precisam ser verificados: **aplicabilidade** e **correção**.
Aplicabilidade requer que, sempre que a pré-condição da operação abstrata for válida em um determinado estado, os estados concretos relacionados satisfaçam a pré-condição da operação concreta. Em outras palavras, a operação concreta deve ter um comportamento bem definido, sempre que a operação abstrata tiver.
No nosso exemplo, as pré-condições de *Entra* e *EntraC* são ambas sempre satisfeitas, de forma que este requisito é trivialmente válido. A pré-condição de *Media* é que a seqüência não seja vazia; a de *MediaC* é $tam \neq 0$. Se a seqüência não é vazia, e *tam* é o seu comprimento, como estabelecido na relação de recuperação, então *tam* é diferente de zero, como requerido.
Quanto a correção, o requisito é que, toda vez que as operações abstrata e concreta forem executadas em estados relacionados, para qualquer estado resultante da execução da operação concreta, deve existir um estado abstrato relacionado que poderia ser obtido com a execução da operação abstrata. Na verdade, esta restrição deve valer apenas quando as operações forem executadas em situações em que suas pré-condições valem. Se a pré-condição da operação abstrata não for satisfeita, não há restrições sobre a operação concreta.

Por exemplo, no caso de *Media* e *MediaC*, nós temos que provar que, se a seqüência não for vazia, então dados *tam'* e *soma'* como definidos em *MediaC* , o *seq'* definido em *Media* está relacionado a eles. Isto é verdade porque, *tam'* é igual a *tam* mais 1, e *soma'* é igual a *soma* mais *n*?, aonde *tam* e *soma* são o comprimento e a soma dos elementos de *seq*. Assim sendo, *tam'* e *soma'* são o comprimento e a soma dos elementos de $seq \frown \langle n? \rangle$, que é a definição de *seq'* em *Media*.
Como antes, estas verificações podem ser formalizadas através de teoremas. A maioria deles são longos, mas simples.
Refinamento de dados também pode ser aplicado a módulos. Sempre que tivermos uma estrutura que encapsula informações, este tipo de modificação é possível.

REFINAMENTO ALGORÍTMICO Uma vez que decidimos os tipos de dados que devem ser usados no programa final, nós podemos prosseguir com a implementação das operações. Há duas abordagens: verificação e cálculo. No caso de refinamento de dados, nós propusemos uma nova especificação e então provamos que ela é satisfatória: nós **verificamos** que o refinamento proposto é correto.
No caso de refinamento algorítmico, as técnicas mais modernas sugerem o uso de uma abordagem baseada em cálculo. Nestas técnicas, chamadas cálculos de refinamentos, a especificação inicial é usada como ponto de partida para uma seqüência de transformações, cada uma formalizada como uma lei matemática, chamada de lei de refinamento. A cada passo, a especificação é gradualmente transformada em um programa.
A linguagem de um cálculo de refinamentos inclui construtores de especificação e programação. No caso do cálculo de Z, além dos esquemas, nós temos atribuições, condicionais, laços, e outros construtores. Durante o desenvolvimento, nós podemos ter, por exemplo, laços cujo corpo é um esquema.
Para a aplicação de uma técnica de refinamento algorítmico, os papéis diferenciados das pré-condições e das pós-condições são muito relevantes. Como esquemas não os distinguem sintaticamente, pode ser conveniente transformar um esquema num comando de especificação. Neste, os componentes de estado que podem ser modificados pela operação são listados, e a pré-condição e a pós-condição são especificados separadamente. Por exemplo, *EntraC* pode ser especificada pelo seguinte comando de especificação.

$$tam, soma : \left[true, \begin{pmatrix} tam' = tam + 1 \\ soma' = soma + n? \end{pmatrix} \right]$$

Ela modifica *tam* e *soma,* sua pré-condição é sempre satisfeita *(true),* e a sua pós-condição é $tam' = tam+1$ e $soma' = soma + n?$. No caso *MediaC,* nós temos o comando de especificação $m! : [\, tam \neq 0, m! = soma \text{ div } tam]$ .Aqui, a pré-condição não é trivial e está claramente separada da pós-condição, o que não ocorre no esquema. Uma lei de refinamento pode ser usada para transformar as especificações de *EntraC* e *MediaC* que usam esquemas nos comandos acima.
As leis de refinamento refletem e formalizam a intuição dos programadores. Por exemplo, é óbvio para um programador que a melhor maneira de implementar *MediaC* é através da atribuição $m! := soma \text{ div } tam$. De fato, há uma lei de refinamento que permite a transformação do comando de especificação que define *MediaC* nesta atribuição. A intuição do programador é que, nas situações em que *tam* é diferente de 0, a atribuição de $soma \text{ div } tam$ a *m!* fará com que a igualdade colocada na pós-condição de *MediaC* seja satisfeita. Este é exatamente o teorema que tem que ser provado quando aplicamos a lei de refinamento que transforma um comando de especificação em uma atribuição.

Desenvolvimentos mais interessantes geram um seqüência de aplicações de lei. Como um exemplo simples, nós consideramos o desenvolvimento de *EntraC*. Intuitivamente, vemos que precisamos de duas atribuições: uma a *tam* e outro a *soma*. A seqüência de comandos pode ser introduzida com a aplicação de uma lei que transforma *EntraC* no seguinte programa.

$$\begin{array}{l} tam, \\ soma \end{array} : \left[true, \begin{pmatrix} tam' = tam + 1 \\ soma' = soma \end{pmatrix} \right] ;$$
$$soma := soma + n?$$

Neste exemplo, nós temos uma combinação de construtores de programação (; e atribuição) com um comando de especificação. A segunda atribuição é introduzida, mas o primeiro comando da seqüência ainda é uma especificação, que precisa ser implementada. É claro que é possível ter uma lei que introduz as duas atribuições de uma só vez, mas a idéia do cálculo é introduzir os diferentes componentes do programa gradualmente.
Como a atribuição já atualiza a *soma*, a pós-condição do comando de especificação requer apenas que seu valor não seja alterado. Com outra aplicação de lei, podemos transformar esta especificação na atribuição *tam* = *tam* + 1. O programa gerado é o seguinte.

$$tam = tam + 1$$
$$soma := soma + n?$$

Com a prova dos teoremas associados a aplicação de cada lei, podemos ter certeza que a implementação obtida satisfaz a especificação inicial. Para este exemplo, o resultado é óbvio; para sistemas reais, o ganho é enorme.

LINGUAGENS ORIENTADAS A OBJETOS Em uma linguagem orientada a objetos como Java (3), programas são formados por **classes**. Como um sistema em Z, classes possuem componentes chamados **atributos** que registram informação usada e manipulada pela classe. Operações sobre atributos são definidas por procedimentos (ou funções) chamados **métodos**.
Uma classe é um módulo de programação, mas pode ser usada como um tipo de dados. Nós podemos declarar variáveis cujo tipo é uma classe; os valores que tais variáveis podem assumir são chamados objetos. Um **objeto** cujo tipo é uma classe `C` é semelhante a um registro que associa um valor a cada um dos atributos de `C`.
Objetos são manipulados através dos métodos de sua classe. O comando `x`.m é uma **chamada de método**; ele invoca o método m da classe de `x`, que age sobre os atributos de `x`.
Classes podem ser definidas através da extensão e modificação de classes existentes com o uso do mecanismo de **herança**. Quando nós definimos uma classe `C2`, nós podemos declarar que ela é uma **subclasse** de, por exemplo, `C1`. Nós dizemos também que `C1` é uma **superclasse** de `C2`. Neste caso, em linhas gerais, `C2` tem todos os atributos de `C1` e mais os atributos declarados em sua definição. Em outras palavras, os atributos de `C1` são herdados por `C2`. O mesmo se aplica para os métodos de `C2`, exceto pelo fato de que, além de declarar métodos adicionais, `C2` pode redefinir métodos de `C1`.
Uma variável `x` cujo tipo declarado é uma classe C, pode ter como valor objetos da classe C ou de qualquer uma de suas subclasses. Nós dizemos que C é o **tipo estático** de `x`, e o tipo do objeto que define o valor de `x` em um determinado instante é o seu **tipo dinâmico**.

Do ponto de vista de refinamento, o uso de classes como tipos de dados requer cuidados adicionais. O refinamento de uma classe não é muito diferente do refinamento de uma especificação em Z. No entanto, novas técnicas são necessárias.
A intuição geral é que se uma classe `C1` é refinada por uma classe `C2`, então usos de objetos de `C1` podem ser substituídos por usos de objetos de `C2`. Entretanto, testes e coerção de tipos podem ser usados para distinguir objetos de classes diferentes. Mesmo que duas classes tenham os mesmos atributos e métodos, mas nomes diferentes, testes de tipo (e coerção) podem ser usados para distinguir objetos destas classes.
Em Java, um teste de tipo pode ser feito com o uso da operação `instanceof:` se `C` é uma classe e `x` é uma variável cujo tipo estático é uma superclasse de `C`, então o teste `x instanceof C` é válido se o valor de `x` é um objeto da classe `C` ou de qualquer uma de suas subclasses. Usando um teste de tipo, nós podemos escrever o comando `if (x instanceof C2) while (true) {};` em Java. Suponha que `C1` não é uma subclasse de `C2`, ou vice-versa, mas que o tipo estático de `x` é uma superclasse de ambos. Se `x` é uma instância de `C1` e, portanto, não é uma instância de `C2`, o comando acima termina e não modifica nenhuma variável. Se, por outro lado, `x` é uma instância de `C2`, ele não termina. Claramente, objetos da classe `C1` não podem ser substituídos por objetos de `C2`, mesmo que estas classes tenham os mesmos atributos e métodos.
Em conclusão, é preciso cuidado quando comparamos classes com nomes diferentes ou com posições diferentes na hierarquia de herança. Em contextos aonde comandos como o mostrado acima estão presentes, isto não é possível. Em programas bem projetados, nós devemos ter apenas comandos c para os quais o bloco de comandos `{ C1 x = new C1(); c }`, que declara a variável `x` com o tipo estático `C1` e a inicializa, antes de executar c, é refinado pelo bloco de comandos `{C2 x= new C2(); c }`. Isto significa que c não faz uso do fato que `x` é uma instância de `C1` ou `C2`.
Para classes com o mesmo nome e posição na hierarquia de herança, refinamento é bem mais parecido com refinamento de dados tradicional. Simulação é uma técnica válida. No entanto, para estabelecer uma simulação, nós precisamos comparar os métodos das classes, da mesma forma que comparamos as operações abstratas e concretas de especificações em `Z`. Neste contexto, entretanto, nós temos chamadas de métodos; elas são um desafio.
Em uma linguagem imperativa, uma chamada de procedimento (ou função) necessariamente invoca a execução do comando na sua única definição: o corpo do procedimento. Para provar propriedades da chamada, nós podemos nos basear no corpo do procedimento. No caso de uma chamada de método `x.m`, o método m executado depende do tipo dinâmico de `x`. Se o tipo estático de `x` é uma classe `C`, então qualquer uma das definições de `m` em `C` e em suas subclasses pode ser invocada em tempo de execução. Para provar propriedades de uma chamada de método, nós temos que considerar todas as possibilidades.
Atualmente, o consenso é que refinamento de programas orientados a objetos só é prático se todas as classes estiverem relacionadas às suas subclasses através de uma relação de refinamento chamada subtipo comportamental. Em Java, e na maioria das linguagens orientadas a objetos com tipos fortes, a herança garante subtipos. Isto significa que, quando uma variável `x` de tipo estático `C` é usada para acesso a um atributo ou método declarado em `C`, nunca ocorre um erro em tempo de execução, mesmo que `x` tenha como valor um objeto de uma subclasse de `C`.

Subtipo, entretanto, não provê nenhuma garantia com relação ao comportamento dos métodos das subclasses. Para redefinir um método m de `C` em uma de suas subclasses, nós precisamos prover uma nova declaração para m que tem os mesmos parâmetros, mas não há restrições sobre o efeito do novo método. Por exemplo, se em `C` o método m incrementa o valor de um atributo a de um valor dado por um argumento `v`, em uma subclasse de `C`, para redefinir `m`, nós precisamos manter o argumento `v`, com o mesmo tipo, mas nós podemos usar este valor para decrementar a, ao invés de incrementar, por exemplo.
Esta prática torna mais difícil provar refinamento de programas, devido à ligação dinâmica. Toda vez que nós mudamos a definição de uma classe, ou adicionamos uma subclasse, nós temos que verificar que todas as propriedades das chamadas de métodos previamente provadas ainda valem. Uma vez que extensão e reuso são um ponto forte do paradigma de orientação a objetos, esta situação não é aceitável.
Nós precisamos de um técnica de refinamento que permita que provas sejam feitas com base no tipo estático das variáveis. Os resultados obtidos devem ser válidos para todos os tipos dinâmicos que essas variáveis podem vir a ter, e não devem ser afetados por mudanças e adições de subclasses. Para que isto seja possível, a estratégia sugerida é que as subclasses sejam sempre subtipos comportamentais. Isto requer que uma redefinição de método refine a definição original.
Outro desafio imposto pelas linguagens orientadas a objetos é o uso maciço de ponteiros e compartilhamento. A maioria dos trabalhos na área de refinamento de programas imperativos tradicionais não considera a possibilidade de compartilhamento. Neste contexto, compartilhamento ocorre apenas como consequência do uso explícito de ponteiros ou passagem de parâmetro por referência. Na grande maioria dos casos, compartilhamento é considerado uma má prática de programação. Por outro lado, se o uso de ponteiros não envolve compartilhamento, então refinamento não é afetado. Assim, ponteiros e compartilhamento são normalmente ignorados.
No caso de linguagens orientadas a objetos, o uso de ponteiros é muito mais comum. A prática adotada por essas linguagens usualmente leva a programas que fazem uso disciplinado de ponteiros, e compartilhamento não é considerado uma má prática. Para ilustrar o problema no tratamento de programas que fazem uso de compartilhamento, nós apresentamos o seguinte comando.

```
x = y; x.a = 1; y.a = 2;
```

A atribuição inicial de `y` a `x` faz com que estas variáveis apontem para o mesmo objeto. Quando nós atribuímos 1 ao atributo a de `x`, e então 2 ao mesmo atributo de `y`, nós estamos na verdade fazendo duas atribuições ao mesmo atributo do objeto apontado por `x` e `y`. Em outras palavras, este comando é equivalente a `x = y; y.a = 2;`.
Entretanto, isto não é verdade se a atribuição `x = y` cria uma cópia do objeto apontado por `y` e faz `x` apontar para esta cópia. Esta abordagem equivale a ignorar o uso de ponteiros e considerar que o valor de `y` é atribuído ao valor de `x`, que é a abordagem usada no tratamento de linguagens imperativas tradicionais. Os contextos diferentes requerem noções de refinamento e técnicas diferentes.

CONCORRÊNCIA No caso de sistemas concorrentes, como sistemas de controle de equipamentos, especificações e projetos se concentram principalmente nas interações com outros sistemas e com o ambiente. Funcionalidade não é caracterizada pela relação entre entradas e saídas, como no caso de sistemas seqüenciais, mas pela maneira em que comunicações e sincronizações podem ocorrer. Entradas e saídas são formas particulares de comunicação.
Terminação não é um requisito importante, uma vez que sistemas que não terminam, como sistemas operacionais, mas que continuamente interagem com o seu ambiente para atingir um determinado objetivo, são úteis. Neste contexto, refinamento deve considerar o comportamento dos sistemas em cada uma de suas interações.
Programas concorrentes são tipicamente formados por vários componentes, que nós chamamos de processos. Eles interagem entre si e com o ambiente externo. Especificação, projeto, e implementação de processos foram extensivamente estudados no contexto de um formalismo chamado CSP (3). Recentemente, tem havido esforços no sentido de enriquecer Java com as facilidades de CSP.
Uma interação é caracterizada em CSP como um evento. Na especificação de um processo, o primeiro elemento de interesse é o conjunto de eventos em que ele pode participar. A definição de um evento simplesmente declara o seu nome.

EXEMPLO Um processo que controla uma porta giratória pode ser caracterizado em termos dos eventos *chega, roda, sai* e *para*. O primeiro representa a chegada de alguém na área ao redor da porta; o segundo, o momento em que a porta começa a girar; *sai* é o evento que modela a saída de uma pessoa da área da porta; finalmente, *para* ocorre quando a porta para de mover.
Na especificação da porta, entradas e saídas não são uma preocupação; o ponto relevante é o modo com que a porta interage com o ambiente: as pessoas que usam a porta. A seguir, nós apresentamos a definição dos processos *Porta (i)*, onde *i* é o número de pessoas que estão usando a porta.
Se não há ninguém usando a porta, o único evento possível é a chegada de alguém; depois, a porta começa a rodar, e passa a se comportar como uma porta que está sendo usada por uma pessoa. Na especificação de *Porta* (0), nós usamos o operador prefixo $a \rightarrow P$, que especifica o único evento a em que o processo está preparado para participar, e um processo P que caracteriza o seu comportamento subseqüente.

$$Porta(0) = chega \rightarrow roda \rightarrow Porta(1)$$

Nós usamos prefixo duas vezes: primeiro, a porta está preparada para registrar o evento *chega,* depois o único evento possível é *roda,* e então a porta se comporta como *Porta* (1).
Se há uma pessoa usando a porta, então outra pessoa chega ou aquela pessoa sai. Nós usamos o operador de escolha de $P[]Q$ para especificar este comportamento: este processo está preparado para se comportar como P ou Q; a escolha é feita pelo ambiente.

$$\begin{aligned} Porta(1) = {} & chega \rightarrow Porta\,(2) \\ & []\, sai \rightarrow para \rightarrow Porta(0) \end{aligned}$$

Se o evento *chega* ocorre, então a porta se comporta como uma porta usada por duas pessoas. Se o evento *sai* ocorre, o único possível é *para,* e então nós temos o comportamento de *Porta* (0) de novo.
Portas com duas ou mais pessoas se comportam de maneira semelhante.

$$Porta(n) = chega \rightarrow Porta\,(n+1)$$
$$\quad []sai \rightarrow Porta\,(n-1) \text{ se } n > 1$$

Se alguém entra em uma porta com *n* pessoas para *n* maior do 1, então nós temos o comportamento de uma porta com *n*+1 pessoas. Se alguém sai, então o comportamento é o de uma porta com *n* –1 pessoas.
Em um prédio grande, nós normalmente temos várias portas giratórias. Elas trabalham em paralelo, mas independentemente. Nós definimos um processo *Entrada* com *m* portas da seguinte forma.

$$Entrada = |||\, i : 1..m \bullet i : Porta(0)$$

Neste processo, há *m* cópias de *Porta*(0) registrando eventos *i.chega, i.roda*, *i.sai*, e *i.para*, para *i* entre 1 e *m*. O conjunto de eventos de *Entrada* inclui todos os *m* events: 4 para cada uma das *m* portas.
Uma porta "educada" contém um componente adicional: um processo que detecta que alguém chegou e emite uma mensagem falada de boas vindas. Este processo pode ser especificado como mostrado abaixo.

$$Educada = chega \rightarrow bem\text{-}vindo \rightarrow Educada$$

O evento adicional *bem-vindo* registra a emissão da mensagem de boas-vindas. A porta educada poder ser caracterizada pela execução paralela da porta *Porta*(0) e *Educada.*

$$PEducada = Porta\,(0) \,||\, Educada$$

Neste processo paralelo, há uma interação entre os dois componentes; eles não são independentes como no exemplo anterior. Uma vez que *chega* é um evento tanto de *Porta* como de *Educada* , estes processos sincronizam nesse evento.Toda vez que alguém chega, *Porta*(0) e *Educada* agem conjuntamente; do ponto de vista de *PEducada*, apenas um evento ocorre.
Refinamento é baseado nas possíveis interações dos processos. Basicamente, as interações da implementação tem que ser interações em que a especificação poderia participar.
No caso do nosso exemplo, não é realístico assumir que um número arbitrário de pessoas pode usar uma porta ao mesmo tempo. Um possível projeto de *Porta*(0) pode ser obtido se nós consideramos que este número é no máximo *max*, e definirmos *Porta*(*max*) como se segue, aonde nós assumimos que *max* > 2.

$$Porta(max) = sai \rightarrow Porta(max-1)$$

Quando o número máximo de pessoas é atingido, a porta não aceita mais a chegada de novas pessoas. O único evento aceito é *sai*.
O número de pessoas que usam a porta é parte de seu estado e não é visível ao ambiente. Em CSP, cada processo encapsula seu estado; interação entre os processos ocorre através de eventos. Refinamento está relacionado apenas a eventos.
Assim sendo, nós podemos considerar refinamento de dados. Por exemplo, nós podemos usar os inteiros negativos para representar o número de pessoas usando uma porta, como mostrado abaixo.

$$PortaN(0) = chega \rightarrow \text{roda} \rightarrow \text{PortaN}\,(-1)$$
$$PortaN(-1) = chega \rightarrow PortaN\,(-2)$$
$$\quad []sai \rightarrow para \rightarrow PortaN(0)$$

$$PortaN(\text{-n}) =$$
$$\quad chega \rightarrow PortaN(-n-1)$$
$$\quad []sai \rightarrow para \rightarrow PortaN(-n+1)$$
$$\quad\quad se\ -n < -1$$

Os processos *Porta*(0) e *PortaN*(0) são equivalentes. Entretanto, este tipo de refinamento não é de interesse da comunidade de CSP porque a linguagem de definição de daods de CSP é muito simples.
Um outro ponto relacionado ao refinamento de processos concorrentes são os eventos em que um processo pode se recusar a participar, e as seqüências de eventos que podem levar a divergência. Por exemplo, a especificação da porta é um processo que não se recusa a reconhecer a chegada de pessoa em qualquer situacão; a implementação, por outro lado, pode ignorar este evento se a porta estiver cheia. Deste ponto de vista, nós não temos realmente uma implementação da porta. A especificação precisa ser mais flexível para permitir uma implementação realística.
Para concluir, técnicas de refinamento provêem justificativas organizadas e claras para a intuição que milhões de programadores usam todo dia para convencer a si mesmos e a outros que os seus projetos e implementações estão corretos. A falta de uma técnica formal têm sido uma das responsáveis por desenvolvimentos mal documentados, desnecessariamente complexos, ou simplesmente errados.
O entendimento de refinamento como a base científica para métodos de desenvolvimento pode ajudar bons engenheiros e programadores a desempenhar suas funções mais efetivamente. Conhecimento das propriedades da relação de refinamento na forma de, por exemplo, leis de refinamento, leva a uma maior habilidade nas tarefas de programação e projeto, mesmo que uma técnica formal de refinamento não seja aplicada.
Mais recentemente, linguagens integradas que combinam, por exemplo, Z e CSP, têm sido propostas; algumas destas linguagens incluem também aspectos de orientação a objetos. A idéia é tornar possível o tratamento dos problemas que nós consideramos aqui isoladamente: tipos de dados, orientação a objetos, e concorrência.

Ana Cavalcanti *é pesquisadora do Computing Laboratory, University of Kent at Canterbury, Inglaterra*

Bibliografia consultada

J. C. P. Woodcock and J. Davies. Using Z - *Specification, Refinement, and Proof*. Prentice-Hall, 1996.
J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
A. W. Roscoe. *The Theory and Practice of Concurrency. Prentice-Hall Series in Computer Science*. Prentice-Hall, 1998.