2
Contratos

Este capítulo trata dos fundamentos e definições relativas aos contratos de software, além das suas principais características e sua relação com o desenvolvimento de software orientado a objetos e baseado em componentes.

2.1
Fundamentos

Entre os fundamentos que servem de base para a teoria de contratos de software, o trabalho realizado por Hoare [9] foi de grande influência por estabelecer os conceitos e a notação matemática usada para especificar o comportamento de um artefato de software. As fórmulas de correção propostas por Hoare, também conhecidas como *Hoare Triples*, são expressões da forma:

$$\{P\}\, Q\, \{R\}$$

Onde Q é um programa ou rotina de software, P é a pré-condição para que Q possa ser executada, e R expressa o efeito da execução de Q (P e R são assertivas booleanas). Informalmente, a expressão pode ser interpretada da seguinte forma: "Se P é verdadeira antes da execução de Q, então R será verdadeira após o seu término". Caso não haja pré-condição para a execução de Q, então P assume o valor *true*, ou seja, a pré-condição é sempre satisfeita.

Numa observação mais formal, se o ambiente em que a rotina Q é executada garante que a assertiva P é verdadeira, então R será verdadeira caso Q termine. Ou seja, caso Q não termine, R não será necessariamente verdadeira. No trabalho de Hoare, alguns axiomas para operações como atribuições e iterações são apresentados, servindo de base para a prova da correção de Q nestes casos.

A correção de uma rotina pode ser parcial ou total. *Correção parcial*: se P é satisfeita e Q termina, então R é satisfeita; *Correção total*: se P é satisfeita, então Q termina e R é satisfeita.

Para provar a correção total, é necessário provar que Q termina, o que é um problema indecidível. Cabe ressaltar também que a prova da correção de um programa é sempre relativa a premissas, tais como a correção do compilador utilizado e do hardware subjacente.

Para um sistema de médio porte desenvolvido nas linguagens mais usadas atualmente, a prova da sua correção ainda é muito complexa, o que a torna inviável para uso geral. Portanto, a correção de um programa é usualmente dada pela consistência do seu comportamento em relação à sua especificação, através de testes funcionais. Como descrito adiante, os contratos de software representam uma maneira de aumentar o formalismo da especificação do software sem no entanto exigir um conhecimento profundo de mecanismos de prova de correção de software.

Outra grande influência sobre a teoria de contratos é o conceito de tipo abstrato de dados, ou *abstract data type* (ADT), introduzido na década de 70 [10]. Ele estabeleceu as bases para a modelagem de software orientada a objetos e influenciou o projeto de muitas linguagens de programação usadas atualmente. Desde então, o paradigma de orientação a objetos tem sido amplamente adotado nos projetos de software visando a obtenção de uma modelagem adequada das abstrações envolvidas e, conseqüentemente, uma melhor qualidade da implementação do produto final.

Contratos

Baseado nestes fundamentos, o conceito genérico de contrato de software pode ser definido como um conjunto de assertivas lógicas que especificam o comportamento da interface de um artefato de software. Um contrato básico é formado por três tipos de assertivas: pré-condições, pós-condições e invariantes. As duas primeiras são derivadas diretamente da noção matemática concebida na fórmula de Hoare, enquanto as invariantes são assertivas que expressam o estado de consistência do artefato em questão, e aplicam-se a todas as operações definidas na sua especificação. A motivação para a definição de contratos é aumentar o formalismo da especificação de uma interface de software, visando torná-la mais precisa e, conseqüentemente, menos ambígua, reduzindo a incidência de erros.

O conceito de contrato de software foi introduzido por Bertrand Meyer [11] sob o termo *Design By Contract*. Meyer é também o criador da linguagem Eiffel [12], que possui suporte nativo ao uso de contratos. Devido à patente estabelecida para o termo *Design By Contract*, é comum encontrar na literatura referências a este conjunto de técnicas sob o nome de *programming by contract* ou *contract-first development*.

Embora o conceito de contrato tenha forte influência do paradigma de orientação a objetos, é possível também aplicá-lo a outros modelos de desenvolvimento como, por exemplo, programas procedimentais e baseados em componentes. Assim, dependendo do contexto em que é utilizado, os contratos podem ser associados a diferentes artefatos de software. No caso de programas orientados a objetos, um contrato se aplica a interfaces, classes e métodos; para programas procedurais, módulos e procedimentos ou funções; para componentes, o contrato se aplica às operações da sua interface e outras características específicas do modelo de componentes em questão. Assim, ao longo do texto são feitas referências a contratos de *objetos* de forma genérica, sem necessariamente se limitar ao uso no paradigma de orientação a objetos.

2.2 Classificação em Níveis

Em [13], os autores apresentam uma classificação para contratos de software em quatro níveis, que variam na ordem crescente da possibilidade de negociação das suas propriedades. A figura 2.1, baseada no mesmo artigo, ilustra esta classificação. A seguir, as principais características de cada nível serão brevemente descritas.

Nível 1 - Sintático

O nível sintático define as informações mínimas que o cliente da interface de um objeto necessita para usá-la de forma correta: o nome da interface, os nomes e tipos dos seus métodos, os nomes e tipos dos parâmetros formais de cada método e as exceções lançadas por eles.

Estas informações permitem que o compilador de uma linguagem de tipos estáticos verifique se as chamadas aos métodos da interface estão corretas do ponto de vista sintático. Para uma linguagem de tipagem dinâmica, a

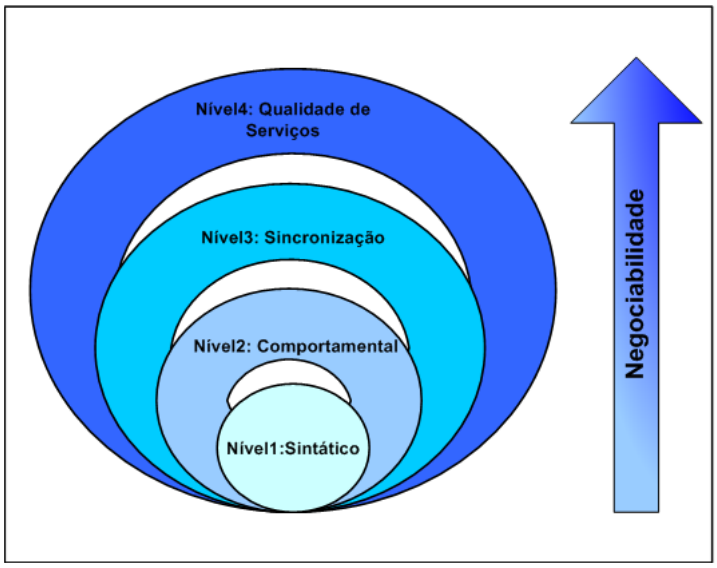

Figura 2.1: Classificação de contratos em níveis

verificação dos tipos deve ser feita em tempo de execução. Porém, em ambos os casos, este tipo de contrato permite que cliente e objeto se comuniquem, viabilizando as chamadas de operações do objeto.

Um exemplo de contrato de nível sintático é o arquivo de definição de interface de um objeto CORBA, escrito em IDL (*Interface Definition Language*). Outro exemplo é a definição de uma interface na linguagem Java.

Este é o nível de menor negociabilidade dos contratos pois, uma vez definidos os seus termos, poucas são as possibilidades de alterar os atributos de uma operação, ou seja, seu nome, parâmetros e tipos envolvidos. Uma exceção seria o caso de algumas linguagens que permitem especificar tipos diferentes de parâmetros e retorno na redefinição de funções, seguindo os princípios da covariância e contravariância de tipos [14].

Nível 2 - Comportamental

O nível sintático não permite que sejam especificados os efeitos de cada operação da interface, nem mesmo os requisitos para que estas possam ser invocadas com sucesso. Por este motivo, o nível comportamental tem como objetivo adicionar esta semântica à especificação da interface, permitindo expressar o comportamento de um objeto com relação às condições e efeitos das chamadas de suas operações.

Baseado no conceito de tipo abstrato de dados, um contrato de nível

comportamental é implementado através das assertivas de pré e pós-condições e invariantes. Este é o nível em que se encontram os contratos propostos por Meyer na teoria de *Design by Contract*, além da maioria das implementações existentes. É também neste nível que se concentra grande parte do trabalho realizado nesta pesquisa.

Em comparação com o nível sintático, os contratos de nível comportamental apresentam maior possibilidade de negociação. Por exemplo, é possível redefinir um contrato comportamental de modo a "relaxar" as suas condições, sem que isso comprometa o funcionamento correto do sistema.

Nível 3 - Sincronização

O nível comportamental não trata de aspectos de concorrência entre as chamadas de operações de uma interface, e pressupõe que todas ocorrem de forma atômica, como numa transação.

Porém, uma grande parte das situações reais de modelagem de interfaces de software precisa considerar os aspectos de sincronização entre as chamadas de um objeto, principalmente quando se trata de uma interface de componente em ambiente distribuído. O nível 3, contratos de sincronização, procura especificar quais operações devem ter exclusão mútua, de forma isolada ou mesmo entre duas ou mais operações.

É possível citar como exemplo de contrato de sincronização a definição de um método *Synchronized* em Java. Embora limitado quando comparado com outros mecanismos, este recurso da linguagem permite definir aspectos de sincronização ainda na definição de interfaces, o que representa um avanço em relação ao uso explícito de *locks*, que são uma fonte constante de erros.

Este nível de contrato permite que os atributos de sincronização sejam definidos na especificação, possibilitando a implementação automática desses mecanismos pelo ambiente de execução do objeto ou componente. Por outro lado, é possível que em determinados casos a granularidade dos mecanismos de exclusão mútua não seja a mais adequada para a solução em questão, o que pode se refletir em problemas de desempenho devido ao uso excessivo de *locks*, que geralmente possuem um custo alto quando usados com muita freqüência.

Alguns aspectos do nível de sincronização serão abordados no capítulo 4, onde será descrita a implementação do estudo de caso.

Nível 4 - Qualidade de Serviço

O nível 4, ou nível de qualidade de serviço (QoS – *Quality of Service*), define alguns aspectos não funcionais do contrato de um objeto de software, voltados principalmente para questões de desempenho, uso de recursos ou precisão de processamento. São exemplos de parâmetros de contratos de nível 4: tempo máximo de resposta, precisão do resultado de uma operação matemática, limite de uso de CPU ou memória, entre outros.

Os contratos de qualidade de serviço apresentam a maior flexibilidade relativa à negociação de suas propriedades, pois é possível variar o nível de exigência das condições impostas pelo contrato para adaptar a sua verificação a mudanças no ambiente. Por exemplo, é possível aumentar no contrato o parâmetro do limite de tempo de resposta, sem que isso altere o comportamento ou a correção do programa sendo executado.

Entretanto, estes contratos impõem uma maior dificuldade prática de implementação, pois em muitos casos dependem de terceiros, que nem sempre podem garantir as condições impostas, ou que não implementam o mesmo tipo de verificação de contratos nos seus componentes ou ambientes de execução. A falta de padronização para o suporte a este tipo de mecanismo dificulta mais ainda a sua adoção.

2.3 Características

Atualmente, a maior parte das implementações de contratos de software está voltada para o nível comportamental, seguindo os conceitos estabelecidos por Meyer com o *Design by Contract*. Da mesma forma, o trabalho desenvolvido nesta pesquisa concentra-se no nível comportamental, embora também aborde questões do nível de sincronização.

Esta seção trata dos principais aspectos que envolvem a utilização de contratos nestes níveis, relacionando-os com a programação orientada a objetos e descrevendo os princípios básicos para a sua elaboração.

2.3.1 Especificação

Segundo Meyer [15], um contrato protege as duas partes envolvidas. Por um lado, protege o contratante (cliente) por especificar o quanto deve ser feito, garantindo um resultado mínimo. Por outro lado, protege o contratado (provedor) por especificar o mínimo aceitável, ou seja, ele fica isento de executar tarefas que estejam fora do escopo previsto no contrato. Como obrigações, o cliente que deve satisfazer as pré-condições da operação, e o provedor deve garantir que as pós-condições sejam verdadeiras. Daí vem a analogia com os contratos legais, onde ambas as partes possuem direitos e deveres a serem observados.

Do ponto de vista do cliente, quanto mais "fraca" for a pré-condição de chamada de uma operação, mais fácil é o seu trabalho, pois este precisa oferecer poucas garantias para o objeto chamado. Uma assertiva a é mais "fraca" que outra assertiva b se b implica a, ou seja, a é menos restritiva que b. Um exemplo seria $a : (x > 10)$ e $b : (x > 20)$, ou seja, $b \Rightarrow a$. O caso extremo de uma pré-condição fraca é a assertiva "true", pois qualquer estado em que a chamada é feita a satisfaz.

Para o provedor da operação, uma pré-condição fraca significa que este deve assumir menos premissas em relação a ela, e portanto deve fazer mais verificações quanto à sua consistência.

Por outro lado, quanto mais "forte" a pós-condição, melhor para o cliente, que tem um resultado da operação mais preciso. Para o provedor do objeto, a situação se inverte novamente. Quanto mais mais forte a pós-condição, mais garantias ele precisa oferecer para o cliente ao final da operação.

Para ilustrar a especificação de um contrato de software, o código a seguir apresenta uma interface escrita na linguagem Eiffel. Ela é a principal referência no uso de contratos por possuir suporte nativo à sua especificação, dispensando o uso de pré-processadores ou ferramentas adicionais. Com ela, a descrição de contratos é parte da implementação e permite, inclusive, a geração automática da documentação da classe a partir do seu conteúdo. A listagem 2.1 representa uma interface denominada SIMPLE_STACK e seu contrato, adaptado de [16].

No exemplo da listagem 2.1, as operações da interface estão acompanhadas de um comentário com a sua descrição, e as pré-condições (*require*) e pós-condições (*ensure*) estão em destaque. As assertivas que contêm o modifi-

Listagem 2.1: Exemplo de contrato em Eiffel

```
class interface
   SIMPLE_STACK[G]
creation
   initialize
feature -- 1. Basic queries
    count: INTEGER -- The number of items on the stack
    item_at(i:INTEGER):G   -- The item at position i
        require
            i_big_enough: i >= 1;
            i_small_enough: i <= count

feature -- 2. Derived queries
    is_empty: BOOLEAN -- Does the stack contain no items ?
        ensure
            consistent_with_count: Result = (count=0)
    item: G      -- The item at the top of the stack
        require
            stack_not_empty: count > 0
        ensure
            consistent_with_item_at: Result = item_at(count)

feature -- 3. Creation commands
    initialize  -- Initialize the stack to be empty
        ensure
            stack_is_empty: count = 0

feature -- 4. Other commands
    put(g:G)     -- Push g onto the stack
        ensure
            count_increased: count = old count + 1
            g_on_top: item_at(count) = g
    remove       -- Delete the top item
        require
            stack_not_empty: count > 0
        ensure
            count_decreased: count = old count - 1
```

cador *old* referem-se ao valor da expressão no início da execução do método, o que possibilita à pós-condição referenciar o valor que a expressão possuía no momento anterior à execução da operação.

Em Eiffel, as assertivas estão associadas a rótulos (e.g. *stack_not_empty*) que permitem que o compilador emita mensagens mais inteligíveis para o desenvolvedor, pois em geral esse rótulo dá uma indicação semântica de qual condição deveria ter sido satisfeita e não foi. As cláusulas *require* e *ensure* podem ser compostas por uma ou mais assertivas, conectadas logicamente pelo operador AND, ou seja, todas as assertivas devem ser verdadeiras para que a pré ou pós-condição da operação seja satisfeita.

Em geral, as linguagens de especificação de contratos possuem características declarativas, não permitindo construções procedimentais (e.g. iterações) que façam do contrato um programa à parte, pois estas podem torná-lo complexo e aumentar a chance de que o contrato tenha seus próprios defeitos. Existem linguagens de contratos [17, 18] que permitem o uso de construções de lógica de primeira ordem, como quantificadores universais (*forall*) e existen-

ciais (*exists*). Estes recursos oferecem uma maior expressividade às assertivas, principalmente em relação ao uso de coleções como vetores ou listas. Um outro exemplo de linguagem com estas características que é usada como base para alguns sistemas de contratos é a *Object Constraint Language* (OCL), que faz parte da especificação da UML [19].

O uso de quantificadores de lógica de primeira ordem contribui para a expressividade dos contratos pois dá um caráter declarativo às assertivas que se aplicam a um conjunto de objetos. Embora esta seja uma característica presente em diversas implementações de suporte a contratos, a linguagem Eiffel padrão não oferece esta facilidade, sendo necessário o uso de pré-processadores específicos para poder utilizá-la. Em [11], Meyer argumenta que estes quantificadores foram deixados de fora do projeto da linguagem porque o seu funcionamento pode ser facilmente obtido por meio de uma chamada de função e, por outro lado, não resolvem alguns casos em que a lógica de primeira ordem não é suficiente, sendo necessária uma ordem mais alta. Como exemplo, ele cita o caso em que é necessário expressar as condições para garantir que um grafo seja acíclico.

O uso de funções pode ser uma ferramenta importante para permitir a expressão de uma condição mais complexa num contrato. Porém, por outro lado, o uso de funções representa um risco à integridade do próprio mecanismo de verificação de contratos, pois aumenta a chance de ocorrência de efeitos colaterais sobre o estado do objeto em questão. Portanto, é necessário que o código das funções usadas em contratos seja tão simples quanto possível, e livre de chamadas que possam alterar o estado do objeto em questão, para evitar efeitos colaterais e a introdução de defeitos no contrato.

Embora a especificação de contratos em Eiffel seja embutida no próprio código, nem sempre esta característica é desejável. Muitas vezes, é interessante que se possa especificar a interface e seu contrato separadamente da implementação, principalmente no que diz respeito ao uso de componentes de software. A separação de interface e implementação favorece a especificação mais formal da interface e evita a introdução de informação extra no código que a implementa.

2.3.2 Verificação

A verificação de contratos pode ocorrer, basicamente, em dois momentos: durante a compilação ou durante a execução. No caso de linguagens interpretadas, normalmente a verificação ocorre em tempo de execução.

A verificação durante a compilação oferece a vantagem de descobrir defeitos no software mais cedo durante o processo de desenvolvimento, quando comparada com a verificação na execução. Porém, a análise estática de software é um problema de difícil tratamento e a prova de correção de programas escritos na maioria das linguagens em uso atualmente não é viável. Além disso, a verificação estática pode demandar o conhecimento de métodos formais de prova de correção de software, o que dificulta a sua adoção de forma mais ampla por exigir um alto nível de especialização do desenvolvedor e o uso de ferramentas complexas, como provadores de teoremas.

A verificação em tempo de execução é a forma mais utilizada pelos mecanismos de suporte a contratos existentes, por ser mais simples de especificar e implementar. Embora os defeitos do software sejam apontados apenas no momento da sua execução, ainda assim há um ganho significativo na descoberta de defeitos através desta técnica. Diferentemente da verificação estática, ela pode ser adotada em sistemas de grande porte sem que a complexidade cresça na mesma proporção.

Grande parte das críticas da verificação de contratos em tempo de execução referem-se à perda de desempenho imposta pelo código extra embutido no sistema. Dependendo do caso, o custo da verificação dos contratos pode ser inviável no ambiente de produção em situação de alta carga. Por isso, diversos mecanismos de suporte a contratos permitem que a verificação seja desativada de forma seletiva, ou seja, por interface, por operação, por tipo (pré/pós-condição ou invariante), ou por objeto. Porém, com a evolução da tecnologia de hardware esse problema tende a ser menos significativo diante do benefício obtido pelo aumento da robustez na execução do software.

Uma possível abordagem para reduzir a influência de um contrato sobre o desempenho de um sistema é considerar que um componente, ao ser adicionado ao sistema, deve passar por um "estágio probatório", durante o qual o seu contrato será verificado rigorosamente. Após um período de testes, as condições do contrato poderiam ser relaxadas à medida que o componente seja considerado estável de acordo com a sua especificação.

Embora a monitoração de contratos seja uma importante aliada na detecção de defeitos de software, ela apenas prova que existem violações quando estas ocorrem, porém a não ocorrência de violações não significa que elas não possam ocorrer em outros contextos. Embora a monitoração de contratos seja uma importante aliada na detecção de defeitos de software, ela apenas prova que existem violações quando estas ocorrem, porém a não ocorrência de violações não significa que elas não possam ocorrer em outros contextos.

A implementação da verificação de contratos nas linguagens de programação é normalmente baseada no suporte do próprio compilador ou no uso de pré-processadores e *proxies*. A subseção 2.3.7 apresenta alguns exemplos de linguagens com suporte a contratos e algumas das suas características.

2.3.3 Notificação

Para tornar efetiva a detecção da violação de um contrato de software, é necessário que haja um mecanismo adequado de notificação desta violação. Existem algumas variantes nas implementações de contratos disponíveis, e em geral a forma de implementar depende da linguagem em questão e do ambiente de execução em que ela é utilizada.

Normalmente, o comportamento de um sistema de verificação de contratos ao detectar uma violação é representado por uma das opções abaixo, ou por uma combinação delas:

- Continuar: o sistema ignora a violação e continua a sua execução. Embora válida, é de pouca utilidade.
- Registrar em log: registro da ocorrência da violação num arquivo de *log* de eventos. Em geral, serve de complemento para as demais opções.
- Abortar: o sistema aborta a sua execução, com uma mensagem indicando a violação. É uma forma extrema de interromper o programa, sem possibilidade de contornar o problema e continuar executando.
- Lançar exceção: o sistema lança uma exceção indicando que ocorreu uma falha na verificação do contrato.
- Delegar: é possível que o mecanismo de suporte a notificação permita registrar um ou mais objetos responsáveis pelo tratamento da violação, o que o torna extensível. Neste caso, o objeto "observador" chamado

pode adotar alguma das estratégias anteriores, porém esse método abre a possibilidade de tratamentos mais específicos para a notificação.

– Bloquear: no caso de contratos que especificam as características de sincronização da interface, quando uma assertiva não é satisfeita pode haver o bloqueio da execução para aguardar que a condição seja satisfeita. Este tipo de comportamento será abordado na subseção 2.3.4 e no capítulo 4.

A notificação deve conter o máximo de informações que indiquem a origem da falha e em que ponto da execução ela ocorreu. A informação contida irá auxiliar o desenvolvedor a identificar a causa e a localização do defeito que originou tal falha. De acordo com a origem da violação, em geral é possível apontar a parte envolvida no contrato onde está localizado o defeito, ou seja, uma falha na pré-condição indica um problema no código do cliente da operação, enquanto que uma falha na pós-condição indica que o erro está na implementação do objeto chamado, assim como ocorre no caso de erros nas invariantes.

Vale lembrar que, no caso do objeto chamado, a falha pode estar localizada em alguma das suas dependências, ou mesmo no hardware subjacente. Falhas nas pré e pós-condições também podem significar inconsistências na especialização de métodos de interfaces herdadas, como será visto mais adiante na subseção 2.3.5.

2.3.4 Concorrência

O controle da concorrência no acesso a objetos ou componentes pode ser especificado por meio de um contrato de sincronização, conforme visto na seção 2.2. Embora seja comum que a própria implementação da interface faça este controle, a definição dos parâmetros de sincronização no contrato pode contribuir bastante para o funcionamento correto do software em questão, pois reduz a chance de haver ambigüidades na especificação que levem à introdução de defeitos relacionados à concorrência.

O controle da concorrência no contrato utiliza a granularidade por operação (ou método) para os mecanismos de sincronização. Este nível nem sempre é o mais apropriado, pois se utilizadas com uma freqüência maior que a necessária, as primitivas de sincronização (*mutex*, semáforo, *critical section*, etc.)

podem provocar uma redução no desempenho do sistema em virtude do custo que este tipo de operação impõe durante a execução. Por outro lado, se utilizadas num nível de granularidade muito alto, estas primitivas também causam problemas devido ao tempo de bloqueio dos recursos compartilhados que elas controlam. Dessa forma, os contratos de sincronização podem tornar a especificação mais precisa, mas também limitam a "sintonia fina"que pode ser feita quando as primitivas de sincronização permeiam o código da implementação em questão.

Em contratos de nível de sincronização, é comum a pré-condição de uma operação assumir um papel de controle da concorrência, e não apenas de verificação da correção [20]. Por exemplo, suponhamos uma interface de pilha de tamanho limitado que seja submetida a acesso concorrente. A pré-condição de sincronização da operação *push* deve possuir uma condição de guarda para garantir que ela só possa ser executada se a pilha não estiver cheia (*notFull*). Caso contrário, a chamada deve ficar bloqueada esperando a condição *notFull* se tornar verdadeira.

Um outro aspecto importante de concorrência em contratos está relacionado à implementação da verificação, mesmo quando não há contratos de sincronização envolvidos. A concorrência nas chamadas de operações do objeto impõem a necessidade de garantir que o contexto de cada chamada seja preservado pelo subsistema de contratos, ou seja, é necessário evitar que as estruturas usadas na verificação sejam corrompidas por conta de chamadas concorrentes. Portanto, é importante preservar o contexto de cada chamada sendo tratada, principalmente quando o mecanismo de verificação de contratos é independente do código de implementação. É o caso, por exemplo, quando o contrato é verificado através da interceptação da chamada num contêiner de execução de componentes.

A concorrência também afeta o funcionamento do mecanismo de verificação de contratos no que diz respeito às condições de corrida que podem ocorrer entre a avaliação das pré-condições, a execução da operação e a avaliação da pós-condição. A suspensão da tarefa corrente em qualquer ponto deste processo pode invalidar as condições requeridas para a operação, gerando resultados inconsistentes, ou então provocar a violação do contrato. Portanto, é necessário implementar mecanismos que permitam que o contrato de operações sujeitas à concorrência sejam avaliados corretamente. O problema aumenta quando as condições do contrato envolvem chamadas a objetos remotos, pois o bloqueio destes para a manutenção do estado corrente durante a execução da operação pode ser inviável ou sujeito a bloqueios perpétuos.

2.3.5
Orientação a Objetos

Como visto na seção 2.1, os fundamentos da teoria de contratos de software estão diretamente relacionados ao conceito de programação orientada a objetos. Portanto, é importante avaliar uso de contratos com relação a alguns conceitos deste paradigma, como herança e correção de classes.

No contexto do desenvolvimento de software orientado a objetos, as pré e pós-condições descrevem individualmente as propriedades de cada método que integra a interface sendo especificada, enquanto as invariantes referem-se às propriedades globais da interface, isto é, elas se aplicam a todas as suas operações. As invariantes podem ser temporariamente violadas devido a estados transientes do objeto durante a execução de um método. Porém, sempre ao final de cada operação as invariantes devem ser satisfeitas para que esta seja considerada correta do ponto de vista do contrato. Deve-se notar que estas restrições aplicam-se somente a operações públicas da interface, pois a implementação de uma determinada operação pode ser composta de uma seqüência de operações intermediárias (não públicas) que nem sempre garantem a consistência do objeto antes do seu final. Portanto, a correção de uma interface em relação ao seu contrato consiste em verificar se as pré e pós-condições e invariantes são satisfeitas para cada operação existente.

Como visto na seção 2.1, pode-se usar a notação da fórmula de Hoare para representar a correção da especificação de uma interface I. A expressão abaixo deve ser satisfeita para cada operação P disponível:

$$\{INV_I \; and \; PRE_P\} \; P \; \{ \; INV_I \; and \; POST_P\}$$

Onde INV_I é a invariante da interface I, PRE_P é o conjunto de assertivas das pré-condições da operação P e $POST_P$ é o conjunto de assertivas das pós-condições de P. Caso alguma pré/pós-condição ou invariante não seja definida no contrato, esta deve ser tratada como uma assertiva "true", pois deve ser satisfeita em qualquer verificação, uma vez que não foi especificada.

A formulação da correção de uma classe apresenta restrições adicionais quando há herança, ou seja, a avaliação do contrato de uma classe derivada deve levar em conta o contrato da sua classe base. Em geral, o mecanismo de subtipos presentes nas linguagens orientadas a objetos verificam apenas a sintaxe das classes que compõem uma hierarquia, de modo que os métodos das classes derivadas devem obedecer a regras específicas de cada linguagem no

tocante a nomes, parâmetros e tipos quando ocorre a redefinição de um método da classe base. O mesmo vale para a herança de interfaces nas linguagens que a suportam, como Java.

Para especificar a semântica da correção de uma classe quando há herança, pode haver variações no rigor da verificação das assertivas na hierarquia. Em [21], o autor cita duas definições para a correção da hierarquia de classes: "correção fraca" (*weak correctness*), referenciando Meyer [20], e "correção forte" (*strong correctness*), baseada no trabalho de Findler [22].

Em ambos os casos, o princípio de "exigir menos, entregar mais" na definição de subtipos orienta o mecanismo de verificação da consistência da hierarquia, seguindo a teoria de *behavioral subtyping* [23], na qual um objeto de uma classe derivada deve poder ser usado em toda operação em que um objeto da classe base estiver sendo esperado.

Segundo a definição de Meyer, a correção de uma classe derivada C a partir do seu contrato é dada pela fórmula abaixo, que deve ser satisfeita para cada rotina exportada r e qualquer conjunto válido de argumentos X:

$$\{(PRE_r(X)\ or\ \forall C_{super} : C_{super}.PRE_r(X))\ and\ (INV\ and\ \forall C_{super} : C_{super}.INV)\}$$

$$\{Rotina\ r\}$$

$$\{(POST_r(X) and \forall C_{super} : C_{super}.POST_r(X)) and (INV and \forall C_{super} : C_{super}.INV)\}$$

Onde PRE_r e $POST_r$ são as pré e pós-condições de r, INV é a invariante da classe, C_{super} é a classe da qual C deriva. Numa linguagem menos formal, a expressão acima pode ser traduzida nas seguintes regras a serem verificadas para a classe C:

- as pré-condições adicionadas por C a uma operação herdada devem ser verificadas com as pré-condições da operação nas superclasses numa operação lógica de disjunção (OR)
- as pós-condições adicionadas por C a uma operação herdada devem ser verificadas com as pós-condições da operação nas superclasses por meio da operação lógica de conjunção (AND)
- as invariantes de C e de todas as classes das quais C deriva também devem ser satisfeitas (AND)

Na definição acima, a correção de uma subclasse só se verifica quando as pré-condições são mais "fracas" em relação à classe base, enquanto as pós-condições são mais "fortes", ou seja, as pré-condições da classe base devem

implicar logicamente as pré-condições da classe derivada, e as pós-condições da classe derivada devem implicar as pós-condições da classe base.

O uso de disjunção na avaliação das pré-condições e de conjunção nas pós-condições se aproveita das tautologias $A \Rightarrow (A\ or\ B)$ e $(A\ and\ B) \Rightarrow A$ para forçar as implicações esperadas entre as assertivas das classes base e derivada, eliminando possíveis erros nos contratos da hierarquia estabelecida. Porém, este mecanismo pode ocultar inconsistências na criação de subtipos.

A definição de "correção forte" consiste em verificar a hierarquia através da análise das implicações lógicas entre as assertivas das classes base e derivada. Ao contrário do mecanismo anterior, não são efetuadas conjunções e disjunções das pré e pós-condições. A formulação da correção forte pode ser expressa como abaixo:

$$\{(PRE_r(X) \Rightarrow \forall C_{sub} : C_{sub}.PRE_r(X))\ and\ (INV\ \Rightarrow\ \forall C_{sub} : C_{sub}.INV)\}$$

$$\{Rotina\ r\}$$

$$\{(POST_r(X) \Rightarrow \forall C_{super} : C_{super}.POST_r(X))\ and\ (INV\ \Rightarrow\ \forall C_{sub} : C_{sub}.INV)\}$$

Onde C_{sub} denota uma subclasse de C, e X, PRE_r, $POST_r$, INV e C_{super} têm o mesmo significado que na definição anterior. Informalmente, a fórmula expressa as seguintes regras:

- as pré-condições das operações de C devem implicar logicamente as pré-condições das operações correspondentes em suas subclasses
- as pós-condições das operações de C devem implicar logicamente as pós-condições de suas superclasses
- as invariantes de C devem implicar logicamente as invariantes de suas subclasses

A maior parte das implementações de contratos adota a estratégia da correção fraca devido à dificuldade de se se provar automaticamente as implicações existentes na correção forte, deixando para a especificação do contrato a análise crítica das condições definidas para a classe derivada.

Alguns trabalhos [24] questionam estes mecanismos de verificação de contratos numa hierarquia de tipos, argumentando que as implicações requeridas entre as pré e pós-condições possuem um rigor excessivo em alguns casos. Assim, a verificação destas implicações poderia gerar falsos negativos, dando como inválidas hierarquias que seriam perfeitamente aceitáveis dentro da teoria de subtipos comportamentais.

2.3.6
Componentes

Um componente de software é, em geral, uma unidade de abstração mais complexa que um objeto, pois normalmente oferece diversos serviços ou facetas, cada um representado por uma interface, que podem ser implementados por meio de objetos.

Cada interface de uma faceta de componente possui uma especificação própria, que pode ser complementada por um contrato. Assim, os aspectos abordados na subseção 2.3.5 se aplicam às interfaces de componentes que são tipicamente implementadas como objetos.

As características de um componente são definidas pelo conjunto de interfaces que o constituem e pela relação estabelecida entre elas. É necessário representar, num nível mais alto, a especificação resultante da utilização destas interfaces para gerar um artefato de software mais complexo. Portanto, os contratos devem expressar a semântica que o conjunto de interfaces do componente oferece aos clientes dos serviços.

A especificação de componentes também pode tratar das suas dependências externas, representadas por meio de "receptáculos" que definem os pontos onde outros componentes devem ser conectados para que as suas funções possam ser desempenhadas. As propriedades destas conexões também podem ser especificadas nos contratos, determinando os critérios da sua compatibilidade e enriquecendo a semântica do componente em questão com relação às suas dependências.

O capítulo 4 apresenta mais detalhes sobre as características de contratos para componentes, através da descrição da implementação do estudo de caso.

2.3.7
Suporte nas linguagens

Atualmente, o suporte ao uso de assertivas pontuais nas linguagens de programação é algo bastante comum, sendo geralmente representadas por rotinas de nome *assert* ou similar. Ela pode ser considerada uma maneira simples de implementar verificações semelhantes às de um contrato de nível comportamental. Porém, quando se trata do suporte a contratos de forma mais ampla, existe um número limitado de linguagens que o oferece, seja diretamente

ou através de alguma extensão ou pré-processador.

Em geral, a implementação de contratos nas linguagens ocorre através de uma ou mais formas relacionadas abaixo:

- suporte nativo: ocorre quando a linguagem já oferece na sua sintaxe algumas construções que permitem a especificação de contratos . Esse é o caso das linguagens Eiffel e Spec#[25], que estão entre os exemplos mais conhecidos. A principal vantagem deste tipo de implementação é a integração das assertivas com a própria linguagem e com as suas ferramentas (compilador, *debugger*, etc.).
- pré-processamento: esta é uma forma bastante comum de se adicionar contratos a linguagens que não os oferecem nativamente. As assertivas são adicionadas ao programa original por um pré-processador, e depois o código instrumentado é compilado para gerar o programa final. Desvantagens deste esquema são a alteração do programa original e a falta de integração com as ferramentas de desenvolvimento da linguagem (e.g. *debugger*). Um exemplo dessa abordagem é o iContract [26], baseado na linguagem Java.
- metaprogramação: este mecanismo é comumente usado em linguagens de tipagem dinâmica e interpretadas, e permite que a interpretação da própria linguagem seja alterada para considerar o uso de contratos, através do uso de técnicas de reflexão. Dois exemplos de sistemas de contratos com estas características são o jContractor [18] e o *DbC for Python* [27].
- *proxies*: técnica usada principalmente em linguagens interpretadas, e que consiste na intermediação da chamada do objeto original através da aplicação do padrão *proxy*, inserindo as assertivas de pré e pós-condições antes e depois da chamada da operação original, respectivamente. O uso de *proxies* dinâmicos permite que a sua construção se dê em tempo de execução, o que evita qualquer necessidade de construção prévia de um *proxy* específico para cada objeto com contrato associado. Um exemplo desta abordagem é dado em [28].
- interceptação: usado em ambientes de execução de *middlewares*, que normalmente oferecem recursos de interceptação de chamadas efetuadas sobre objetos remotos ou componentes, como é o caso de CORBA. Essa técnica permite que o código da verificação de contratos seja isolado do código da implementação dos objetos ou componentes. Esta técnica é

usada em alguns dos trabalhos apresentados no capítulo 3, e também no estudo de caso que realizamos.

Além da forma de implementação dos contratos, uma característica que diferencia as linguagens é a possibilidade de evitar que uma assertiva no contrato referencie métodos com efeitos colaterais sobre o objeto. Uma vez que seja possível efetuar chamadas de métodos nas assertivas de um contrato, deve-se evitar a chamada daqueles que alteram o estado interno do objeto.

Um exemplo de linguagem com suporte a contratos que permite a especificação de métodos "puros" (sem efeitos colaterais) é Spec# [25], que oferece o modificador *[Pure]*. Embora não possua suporte a contratos, a linguagem C++ oferece um mecanismo básico para reduzir efeitos colaterais através do modificador *const*, que impede a chamada de um método constante associada a um objeto não constante.

2.3.8 Testes

Além dos benefícios sobre a especificação de interfaces, o uso de contratos complementa a disciplina de teste de software, uma vez que desempenha um papel importante na busca por defeitos.

As invariantes representam um reforço aos testes de unidade, pois auxiliam na definição formal do objeto e na verificação permanente de sua consistência. As pós-condições, por sua vez, garantem que o resultado de cada operação cumpra com a especificação dada.

Caso um defeito seja introduzido durante manutenção do código existente, provavelmente o retorno de algum método ou a consistência do objeto serão alterados, causando uma violação do contrato que irá notificar o ocorrido. Assim, contratos possuem um papel importante no auxílio a testes de regressão, pois ajudam a identificar defeitos nas alterações da base de código existente.

As pré-condições ajudam na verificação dos clientes que utilizam uma determinada interface, enquanto as pós-condições e invariantes monitoram a qualidade da sua implementação. Portanto, os contratos extrapolam a identificação de problemas apenas num objeto ou unidade de código, reforçando

os testes de integração, pois o relacionamento entre os objetos é continuamente verificado.

2.3.9 Princípios para a especificação

Em [16], os autores apresentam alguns princípios básicos para a definição de contratos de uma classe, com exemplos em Eiffel. Apesar de usar a terminologia da linguagem, os princípios são bastante genéricos e servem como guia de boas práticas para a elaboração das assertivas de um contrato.

Os tópicos a seguir contêm uma breve descrição de alguns princípios e boas práticas para a definição de contratos, com base na publicação citada e com algumas adaptações.

Princípio I: Separação entre consultas e comandos

Usando a terminologia da linguagem Eiffel, "consultas" (*queries*) são atributos ou métodos de uma classe que apenas retornam um valor sem no entanto alterar o estado do objeto. Por outro lado, comandos são os métodos que alteram o estado do objeto, mudando o valor de algum atributo interno. Como descrito na subseção 2.3.7, as consultas são os métodos isentos de efeitos colaterais durante a avaliação de uma assertiva. A figura 2.2 ilustra as características de um objeto e a sua divisão entre consultas e comandos.

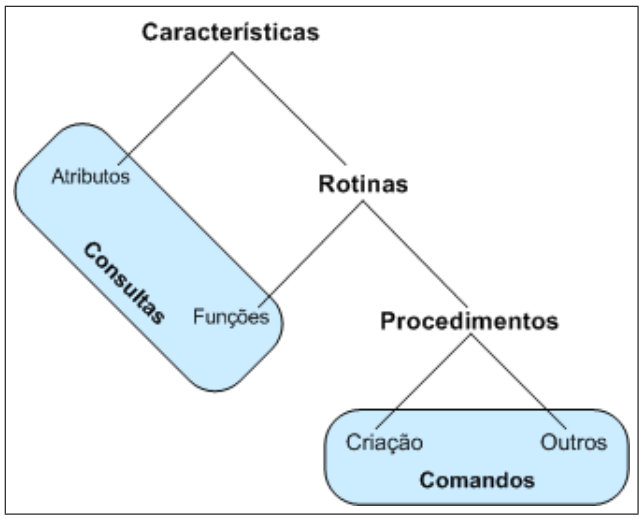

Figura 2.2: Consultas e Comandos

O primeiro princípio prega a separação entre os dois tipos de operação para evitar que, durante a avaliação do contrato, uma chamada de operação sobre o objeto cause a alteração do mesmo. Portanto, o ideal é evitar que métodos que alterem o estado do objeto (comandos) também funcionem como consulta. Quando essa situação for difícil de ser evitada, deve-se prover também a funcionalidade mais básica, ou seja, deve ser criada uma consulta que apenas retorna a mesma informação que o comando associado.

Um exemplo desta situação seria uma operação *pop* na interface de uma estrutura de pilha. Ela remove o elemento do topo da pilha, e pode também retorná-lo para o cliente, funcionando como comando e consulta. Porém, é importante definir uma operação *top* para que seja possível consultar o elemento do topo sem no entanto removê-lo e, conseqüentemente, sem alterar o estado do objeto.

Princípio II: Separação entre consultas básicas e derivadas

Uma consulta derivada é aquela que depende de uma ou mais consultas básicas, sendo definida em termos de outras consultas. Assim, este princípio prega a separação entre os dois tipos, e a especificação das consultas derivadas em função das consultas básicas.

A idéia por trás deste princípio é facilitar a definição das assertivas de uma operação de forma que, se esta depende de uma consulta derivada, a sua dependência em relação às consultas mais básicas estará automaticamente definida por conta do encadeamento entre elas.

Princípio III: Definir pós-condições de consultas derivadas em termos de consultas básicas

Seguindo a definição do Principio II, se uma consulta derivada depende de uma ou mais básicas, é interessante que se defina a sua pós-condição em função dos valores da consulta básica.

Seja, por exemplo, uma classe que implementa uma coleção e que possui na interface os métodos *isEmpty* e *count*, que retornam se a coleção está vazia e a sua quantidade de elementos, respectivamente. A definição de *isEmpty* deve se dar em função da consulta mais básica, *count*. Com isso, podemos especificar a pós-condição da primeira em função do retorno da segunda, com

uma assertiva do tipo: $(RESULT == (count == 0))$. A assertiva assegura na pós-condição que o resultado de *isEmpty* só será verdadeiro se *count* for igual a zero (neste exemplo, $RESULT$ representa o retorno da operação e '==' é o operador relacional de igualdade).

Princípio IV: Definir pós-condições de comandos em função das consultas básicas

Dado que um comando altera o estado observável do objeto, é importante definir na sua pós-condição como ele afeta o resultado das consultas básicas. Portanto, o ideal é que cada consulta básica influenciada pelo comando tenha o seu valor determinado na pós-condição.

Associando-se este princípio com o que foi definido no Principio III, obtém-se em cascata o efeito das pós-condições de um comando sobre as consultas derivadas.

Tomando o exemplo usado no princípio anterior, se a interface possui um comando *put* que insere um elemento na coleção, a pós-condição deve incluir o resultado deste comando sobre a consulta básica *count*. Neste caso, a assertiva poderia ser $(count == oldcount + 1)$, o que indica que o valor de *count* deve ser incrementado pela inclusão do novo elemento. Como *isEmpty* é definida em termos de *count*, o efeito de *put* sobre esta consulta é transitivamente definido.

Princípio V: Selecionar pré-condições adequadamente

Ao definir a pré-condição de uma consulta ou comando, quanto mais restritiva ela for, mais simples é a sua implementação, porém mais difícil fica a sua chamada por parte do cliente. Dependendo do caso, o relaxamento das restrições na pré-condição pode ser positivo por permitir um uso mais amplo da operação, como numa biblioteca de uso genérico, por exemplo.

É importante levar em consideração na especificação de um método o contexto em que este será usado. Por exemplo, ainda no exemplo anterior, se a coleção possui na interface um comando *remove*, que exclui um elemento em função de sua chave, a pré-condição pode exigir que este elemento faça parte do conjunto. Porém, uma alternativa mais tolerante é simplesmente relaxar essa restrição e ignorar quando o elemento especificado não existe na coleção, sem impor a tarefa desta verificação ao cliente. Neste caso, a complexidade

de implementação varia pouco e o trabalho de todos os clientes torna-se mais simples.

Por outro lado, em alguns casos a pré-condição deve ser mais restritiva para evitar comportamentos inesperados por parte da implementação. Por exemplo, no caso de um método para calcular a raiz quadrada de um número, se o seu retorno é um número real o parâmetro passado deve ser não negativo, e isto deve constar na pré-condição.

Outro ponto importante é o uso de restrições "físicas", ou seja, aquelas que levam em conta a tecnologia sendo utilizada. Nesta categoria entram verificações de valor nulo de parâmetros, por exemplo. Caso sejam parte da pré-condição, a passagem de um parâmetro nulo pode causar uma violação, o que pode ser desejável ou não dependendo da interface. Assim como no exemplo do *remove*, em certos casos um parâmetro nulo pode receber um tratamento tolerante e ser ignorado, sem que uma violação seja sinalizada.

É importante evitar que as pré-condições possuam assertivas de grande impacto no desempenho, uma vez que elas servem para verificar a correção da chamada do cliente à operação em questão. Dessa forma, é comum deixar apenas as pré-condições habilitadas quando o software encontra-se em produção, já que ela protege a implementação dos clientes "mal comportados".

Em resumo, não existe uma regra geral, e este princípio prega que cada caso deve ser analisado de acordo com a necessidade, evitando pré-condições muito restritivas e ao mesmo tempo muito liberais.

Princípio VI: Definir invariantes para as propriedades imutáveis dos objetos

A definição de invariantes para uma interface favorece a compreensão do modelo conceitual da abstração em questão por parte do cliente que a utiliza. Além disso, ela é importante para detectar inconsistências na implementação associada, principalmente durante as etapas de desenvolvimento e testes.

Mesmo após a implantação do software, o uso de invariantes pode ser útil na descoberta de defeitos ainda obscuros, sendo também importante do ponto de vista de documentação da interface para os clientes que venham a utilizá-la.

Princípio VII: Evitar efeitos colaterais

Um dos principais requisitos para a implementação de contratos é a sua transparência, ou seja, o uso de contratos não deve alterar o funcionamento normal do software, a não ser em caso de violação do contrato, visto que esta deve ser sinalizada, muitas vezes por meio de exceções.

É importante garantir que as assertivas e chamadas contidas no contrato sejam isentas de efeitos colaterais, isto é, não alterem o estado observável do objeto, pois isto significaria uma intervenção na implementação subjacente. Como visto na subseção 2.3.7, algumas linguagens permitem especificar operações sem efeitos colaterais. Nos casos em que esta facilidade não encontra-se disponível, deve-se ter em mente a orientação deste princípio ao especificar um contrato.

Uma alternativa para auxiliar no controle dos efeitos colaterais seria colocar na pós-condição de uma operação algumas assertivas que comparem os retornos das consultas básicas com os seus valores prévios, para garantir que não tenham sido alterados inadvertidamente durante a chamada. Porém, dependendo da interface, essa verificação pode prejudicar a compreensão do contrato devido ao excesso de assertivas extras.