Instituto Superior Técnico

Programação Avançada – 2007/2008

Exame – 17/6/2008

Número: _____

Nome: _____

Escreva o seu número em todas as folhas da prova. O tamanho das respostas deve ser limitado ao espaço fornecido para cada pergunta. Pode usar os versos das folhas para rascunho. A prova tem **8** páginas e a duração é de **2.0 horas**. A cotação de cada questão encontra-se indicada entre parêntesis. Boa sorte.

1. (1.0) O que é a reflexão? Quais são os graus de reflexão que conhece? Explique e exemplifique.

   Reflexão é a capacidade que um sistema computacional tem de manipular uma representação da sua estrutura e comportamento durante a sua própria execução.

   Existem dois* graus *de Reflexão:

   - ***A Introspecção é a capacidade de um sistema de* observar *a sua própria estrutura e comportamento. Por exemplo, saber "Quantos parâmetros tem a função*** `foo`***?" é introspecção.***
   - ***A Intercessão é a capacidade de um sistema de* modificar *a sua própria estrutura e comportamento. Por exemplo, "Mudar a classe desta instância para*** `Bar`***" é intercessão.***

2. (2.0) A linguagem Java disponibiliza algumas formas de reflexão que nos permitem escrever o seguinte fragmento de programa:

   ```
   Class cls = Class.forName("Foo");
   Method meth = cls.getMethod("bar", new Class[] {});
   meth.invoke(cls.newInstance(), new Object[] {});
   ```

   (a) (1.0) Reescreva o fragmento de modo a produzir o mesmo resultado mas sem empregar os mecanismos reflectivos da linguagem Java.

   ```
   new Foo().bar();
   ```

(b) (1.0) Que vantagens e desvantagens é que a sua versão tem sobre a anterior? Explique.

A versão sem reflexão tem a desvantagem de ser rígida em termos da classe instanciada e do método invocado, não podendo ser adaptada, durante a execução, para outras classes e/ou métodos. No entanto, tem a vantagem de ser mais fácil de ler, permitir detectar erros em tempo de compilação e ser mais eficiente.

3. (2.0) A linguagem Java permite o *overloading* de métodos enquanto que linguagem Common Lisp permite o *despacho múltiplo*. Quais as semelhanças e diferenças entre estas duas abordagens? Como é que poderia implementar despacho múltiplo em Java?

A semelhança é que ambas as abordagens se destinam a permitir que um mesmo nome possa designar diferentes computações dependendo do tipo dos argumentos.

A diferença é que no caso do despacho múltiplo essa escolha é determinada em tempo de execução e em função dos tipos dos argumentos enquanto que no overloading a escolha da computação a executar é determinada em tempo de compilação e em função dos tipos das expressões argumentos (tipos esses que podem não ser iguais aos tipos dos argumentos).

Para se implementar o despacho múltiplo em Java ou usava cascatas de `instanceof` ***com casts, ou usava encadeamentos do padrão de despacho duplo ou empregava reflexão para identificar dinamicamente o melhor método para um dado conjunto de argumentos.***

4. (2.0) Em Java, qualquer variável de um programa tem de possuir um valor. No caso de variáveis locais, o compilador verifica (de forma conservativa) que o programador inicializa ou atribui cada variável antes de a ler. No caso de variáveis de instância, estas são inicializadas automaticamente (por exemplo, a `null`, no caso de variáveis de tipo referência), permitindo que sejam lidas antes de o programador lhes atribuir um valor. Infelizmente, este comportamento da linguagem Java permite que um programa possa ler estas variáveis sem terem sido previamente inicializadas, "escondendo" assim potenciais erros.

   Proponha um mecanismo de intercessão baseado em Javassist que permita detectar e reportar, como erro, qualquer leitura de uma variável de instância que não tenha sido previamente atribuída. Não é necessário descrever exaustivamente a implementação do seu mecanismo mas inclua toda a informação que considerar relevante para que outra pessoa (conhecedora de Javassist) possa realizar essa implementação sem problemas.

   Através de um Translator, operamos sobre cada classe que é carregada.

   Para cada slot, acrescentamos um novo slot de tipo booleano inicializado a falso para indicar se o slot original já foi atribuído e cujo nome resulta univocamente do nome do slot original, por exemplo, acrescentando `$initializedP` ***ao nome do slot original. O slot meta-objecto é acrescentado a uma lista do translator para se saber quais os slots que estão a ser monitorizados.***

   Para cada behavior (método, construtor, inicializador), analisamos o código e, para cada acesso a um slot, verificamos se o slot referenciado está na lista dos slots monitorizados e, se estiver, então:

   - ***se o acesso é uma escrita, injectamos código após a escrita para mudar o valor do correspondente slot*** `$initializedP` ***para true***
   - ***se o acesso é uma leitura, injectamos código antes da leitura para verificar se o valor do correspondente slot*** `$initializedP` ***é true e para gerar uma excepção em caso contrário.***

5. (2.0) Distinga os seguintes conceitos da linguagem CLOS: função genérica, método, método primário, método auxiliar, método aplicável e método efectivo.

   Uma função genérica é uma função cujo comportamento depende dos tipos dos argumentos. A função genérica é implementada por um conjunto de métodos que correspondem a especializações da função genérica para diferentes tipos de argumentos e para diferentes papéis. Estes papéis permitem classificar os métodos em métodos primários (que produzem o valor a retornar) e métodos auxiliares (que modificam ou controlam o comportamento dos primários).

   Quando se aplica uma função genérica, os argumentos são usados para seleccionar o subconjunto de métodos da função cujos especializadores emparelham com os argumentos. Estes métodos aplicáveis são são ordenados e combinados tendo em conta a sua especificidade e o seu papel (como primários ou auxiliares) para produzir o método efectivo que vai ser de facto aplicado aos argumentos.

6. (1.0) O que é uma *metaclasse*? Qual a utilidade das metaclasses? Explique.

   A metaclasse de um objecto é a classe da classe desse objecto. Uma metaclasse é uma classe cujas instâncias são classes.

   A utilidade das metaclasses está no facto de permitirem:

   - ***determinar a forma de herança das classes que são suas instâncias.***
   - ***determinar a representação das instâncias das classes que são suas instâncias.***
   - ***determinar o acesso aos* slots *das instâncias.***

7. (1.0) Algumas linguagens de programação permitem que uma instância possa mudar de classe. Quais são as vantagens e desvantagens dessa possibilidade? Explique.

   As vantagens incluem:

   - ***Aumento de poder expressivo: permite escrever programas mais simples do que quando isso não é possível.***
   - ***Possibilidade de criar instâncias quando ainda não é bem conhecida a classe a que devem pertencer e poder especificar essa classe mais tarde.***
   - ***Permitir actualizar as instâncias já existentes sem ter de fazer* redeploy *da aplicação***

   As desvantagens incluem:

   - ***Implementação mais complexa da linguagem.***
   - ***Ineficiências no acesso às instâncias ou na mudança de classe.***
   - ***Dificuldades de optimização.***
   - ***Funcionamento do programa difícil de entender.***
   - ***Depuração mais difícil.***

8. (2.0) O que é um *cross-cutting concern*? Quais são os problemas causados pelos *cross-cutting concerns*? Explique.

   Um* cross-cutting concern *é um requisito de uma aplicação cuja implementação atravessa todo o sistema, afectando um largo conjunto de classes, objectos ou funções.

   Os problemas dos* cross-cutting concerns *são:

   - **Code Tangling *Um módulo implementa um* concern *fundamental e parte de outro que lhe é transversal.***
   - **Code Scattering *Um* concern *transveral é implementado por fragmentos espalhados em várias partes do sistema.***

   Destes problemas, resulta:

   - ***Baixa qualidade:* tangling *de código facilita erros.***
   - ***Baixa rastreabilidade: onde está o código que implementa um* concern*? A que* concern *diz respeito este código?***
   - ***Difícil reutilização: cada módulo inclui fragmentos que não têm a ver com a sua função principal.***
   - ***Difícil evolução: de módulos que estão espalhados pelo sistema e de módulos que incluem fragmentos de outros módulos.***

9. (2.0) A linguagem AspectJ é uma extensão da linguagem Java destinada à programação orientada a aspectos e introduz uma série de novos conceitos, em particular, *join point*, *pointcut*, *advice*, *inter-type declaration* e *aspect*. Explique estes conceitos.

   Um* join point *é um ponto bem definido no fluxo de execução de um programa (onde um aspecto se "junta").

   Um* pointcut *é um conjunto de pontos de junção e valores nesses pontos.

   Um* advice *é um fragmento de código que é executado quando um ponto de junção é atingido.

   Uma* inter-type declaration *é uma declaração que permite alterar a estrutura estática de um programa (membros de classes e relações entre classes).

   Um* aspect *é um módulo que agrupa secções (*pointcuts*), conselhos (*advices*) e declarações inter-tipo.

10. (2.0) Defina a *macro* `when` em Scheme (usando `syntax-rules` ou `syntax-case`) que permita escrever expressões condicionais da forma:

```
(when (negative? x)
  (newline)
  (display "Bad number: negative."))
```

e obter, como expansão:

```
(if (negative? x)
  (begin
    (newline)
    (display "Bad number: negative."))
  #f)
```

```
(define-syntax when
  (syntax-rules ()
    ((when condition form . forms)
     (if condition
       (begin form . forms)
       #f))))
```

11. (3.0) Considere o seguinte micro-avaliador:

```
(define (eval expr)
  (cond ((number? expr)
         expr)
        (else
         (apply
          (cond ((eq? (car expr) 'plus) +)
                ((eq? (car expr) 'times) *)
                ((eq? (car expr) 'smaller) <)
                (else (error 'eval "Unknown operator")))
          (list-of-values (cdr expr))))))

(define (list-of-values exprs)
  (if (null? exprs)
      (list)
      (cons (eval (car exprs))
            (list-of-values (cdr exprs)))))
```

Este micro-avaliador é capaz de avaliar expressões aritméticas simples como, por exemplo:

```
> (eval '(plus 1 (times 2 3) (times 3 4)))
19
```

(a) (1.0) Quantas invocações da função `eval` ocorrem em consequência da avaliação da expressão anterior?

8

(b) (1.0) Modifique o avaliador de modo a que a ordem de avaliação dos argumentos dos operadores seja **da direita para a esquerda**. Isto quer dizer que, no exemplo anterior, a sub-expressão `(times 3 4)` seria avaliada antes da sub-expressão `(times 2 3)` que seria avaliada antes da sub-expressão `1`.

```
(define (list-of-values exprs)
  (if (null? exprs)
      (list)
      (let ((rest-values (list-of-values (cdr exprs))))
        (cons (eval (car exprs))
              rest-values))))
```

(c) (1.0) Considere uma extensão à linguagem implementada pelo interpretador anterior que consiste na introdução do *operador de selecção* `if`. Com este novo operador, deverá ser possível obter a seguinte interacção:

```
> (eval '(plus 1
              (if (smaller (times 2 3) (plus 3 4))
                  (plus 4 5)
                  6)))
10
```

Redefina o avaliador de modo a que ele possa lidar com o operador de selecção `if`.

```
(define (eval expr)
  (cond ((number? expr)
         expr)
        ((eq? (car expr) 'if)
         (if (eval (cadr expr))
             (eval (caddr expr))
             (eval (cadddr expr))))
        (else
         (apply
          (cond ((eq? (car expr) 'plus) +)
                ((eq? (car expr) 'times) *)
                ((eq? (car expr) 'smaller) <)
                (else (error 'eval "Unknown operator")))
          (list-of-values (cdr expr))))))

(define (list-of-values exprs)
  (if (null? exprs)
      (list)
      (cons (eval (car exprs))
            (list-of-values (cdr exprs)))))
```