Instituto Superior Técnico

Programação Avançada – 2008/2009

Segundo Exame – 21/7/2009

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 **6** páginas e a duração é de **2.0 horas**. A cotação de cada questão encontra-se indicada entre parêntesis. Boa sorte.

1. (1.0) Defina **reflexão** e **sistema reflexivo**.

   Reflexão é a capacidade de raciocinar e/ou actuar sobre si próprio.

   Um sistema reflexivo é um meta-sistema computacional que se tem a si próprio como sistema objecto.

2. (1.0) O que é uma **arquitectura reflexiva** e quais os requisitos necessários para a implementar? Explique.

   Uma arquitectura reflexiva é uma arquitectura que disponibiliza mecanismos para lidar explicitamente com a computação reflexiva. Como requisitos, é necessário providenciar estruturas de dados que representam o próprio sistema, existindo uma relação causal entre estes dados e os aspectos do sistema que eles representam, de tal modo que modificações nos dados causam modificações no estado e comportamento do sistema.

3. (2.0) Tendo em conta as várias dimensões da reflexão (introspecção, intercessão, estrutural, comportamental), classifique os seguintes mecanismos:

   (a) (0.5) Operação de rastreio de invocações de função que mostre os argumentos e o resultado de cada vez que a função é invocada.

   Introspecção comportamental.

(b) (0.5) Operação que permite saber qual o número de parâmetros de uma função.

Introspecção estrutural.

(c) (0.5) Operação que permite retomar a execução de um programa que foi interrompido por um erro produzido por uma invocação de função, através da imposição de um valor de retorno a essa invocação.

Intercessão comportamental.

(d) (0.5) Operação que permite mudar uma instância de forma a pertencer a uma classe diferente daquela que foi usada para a sua criação.

Intercessão estrutural.

4. (3.0) Durante a execução de um programa Java, a cadeia de invocações de métodos que conduziram ao estado actual, bem como os seus argumentos e os valores das variáves locais, são guardadas no *stack*. Infelizmente, a linguagem Java não permite facilmente a um programa aceder ao *stack*.

   Proponha um mecanismo de intercessão baseado em Javassist que permita implementar uma API para aceder à cadeia de invocações de métodos e, para cada um, saber quais os argumentos com que foi invocado. 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.

5. (1.5) Descreva, de forma mais exaustiva que lhe for possível, cada um dos seguintes *pointcuts* de AspectJ.

   (a) (0.5) `call(void *.foo(Bar))`

   Invocações do método `foo`, definido em qualquer classe, cujo tipo de retorno é `void` e com um único parâmetro do tipo `Bar`.

   (b) (0.5) `execution(public void Foo.bar(Baz))`

   Execuções do método `public` `bar` definido na classe `Foo` que tem um parâmetro do tipo `Baz` e tipo de retorno `void`.

   (c) (0.5) `cflow(call(* Foo.bar(..)))`

   Pontos de junção que ocorram no fluxo de controle de invocações a todos os métodos `bar` da classe `Foo` (qualquer qualquer tipo de argumentos e resultado), incluíndo a própria invocação.

6. (1.0) Alguns autores criticam algumas implementações da programação orientada a aspectos (e, em particular, AspectJ) por os aspectos dependerem, em grande medida, de detalhes concretos do código onde vão intervir. Comente esta afirmação, relacionando-a com as metodologias modernas de desenvolvimento de programas, como a "Extreme Programming," que promovem frequentes mudanças nos programas.

   Uma vez que os aspectos dependem de detalhes concretos da código, cada vez que o código muda é possível que um aspecto, que era aplicável num determinado ponto, deixe de o ser e que um aspecto, que não era aplicável num determinado ponto, passe a ser. Sem termos conhecimento completo dos requisitos de um aspecto, torna-se difícil fazer alterações ao programa que continuem a obedecer aos objectivos desse aspecto. Uma metodologia que promove mudanças frequentes ao código amplifica este problema. A sua solução exige ferramentas, como o Eclipse, que mostrem quais os aspectos cuja aplicabilidade é afectada pelas alterações ao programa.

7. (1.0) As versões recentes da linguagem Java disponibilizam *anotações*. Explique o que são e refira a sua utilidade, quer para o Javassist, quer para o AspectJ.

8. (1.0) A linguagem Java permite o carregamento dinâmico de classes. No entanto, sempre que uma classe é carregada, é-lhe imediatamente associado o *class loader* que a carregou. Que problema é que se pretendeu resolver com esta abordagem? Explique.

9. (2.0) Considere o seguinte programa CLOS:

```
(defmethod foo ((num number))
  (print 1))

(defmethod foo ((num rational))
  (print 2))

(defmethod foo :after ((num integer))
  (print 3))

(defmethod foo :before ((num (eql 5)))
  (print 4))

(defmethod foo :before ((num real))
  (print 5))
```

Sabendo que `integer` é subtipo de `rational` que é subtipo de `real` que é subtipo de `number`, diga o que é que é escrito por cada uma das seguintes invocações:

(a) (0.5) `(foo 1)`

```
5 2 3
```

(b) (0.5) `(foo 0.5)`

```
5 1
```

(c) (0.5) `(foo 1/2)`

```
5 2
```

(d) (0.5) `(foo 5)`

```
4 5 2 3
```

10. (1.0) Contrariamente ao que acontece em Java, em CLOS os métodos não pertencem às classes. Porquê? Explique.

Despacho múltiplo.

11. (1.5) O que é uma metaclasse? Quais são as responsabilidades das metaclasses em CLOS?

Uma metaclasse é uma classe cujas instâncias são classes.

As responsabilidades de uma metaclasse são:

- ***A metaclasse determina o processo de herança que é usado pelas classes que são suas instâncias.***
- ***A metaclasse determina a representação das instâncias das classes que são suas instâncias.***
- ***A metaclasse determina o acesso aos* slots *das instâncias das classes que são suas instâncias***

12. (2.0) Diferentes linguagens de programação adoptam diferentes abordagem no que diz respeito ao *âmbito* dos nomes, podendo este âmbito ser *léxico* ou *dinâmico*.

    (a) (1.0) Em que condições é que estes dois âmbitos podem produzir resultados diferentes? Explique.

    Na presença de funções de ordem superior e funções com variáveis livres.

(b) (1.0) Explique as alterações que é necessário fazer na implementação de um avaliador de âmbito dinâmico para que passe a fazer avaliação de âmbito léxico.

As funções passam a ter associado o ambiente onde foram criadas. A aplicação de funções passa a extender esse ambiente (com as associações entre parâmetros formais e actuais) ao invés do ambiente dinâmico.

13. (2.0) A *avaliação preguiçosa* (de *lazy evaluation*), também conhecida por *avaliação atrasada* é usada por várias modernas linguagens de programação, como Haskell, O'Caml e F#.

(a) (1.0) O que é a *avaliação preguiçosa*? Explique.

(b) (1.0) Como é que pode implementar avaliação preguiçosa num avaliador metacircular? Explique.

O avaliador metacircular deverá usar âmbito léxico e, de cada vez que invocamos uma função não primitiva, "embrulhamos" cada um dos argumentos numa estrutura que associa a expressão correspondente ao ambiente em que era suposto ser avaliada. Sempre que invocamos uma função primitiva, os argumentos que foram "embrulhados" são "desembrulhados" através da avaliação da expressão no ambiente que lhe foi associado. Para evitar múltiplas avaliações, convém que o "embrulho" possa ser manipulado para registar o resultado da primeira avaliação.