Instituto Superior Técnico

Programação Avançada

Segundo Exame – 8/7/2008

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 *reificação*? Explique e exemplifique.

   A reificação é a criação de uma entidade que representa, no meta-sistema, uma entidadade do sistema. A reificação é condição para a reflexão.

   - ***"Qual é a classe desta instância?" implica a reificação de classes.***
   - ***"Quais são os métodos desta classe?" implica a reificação de métodos.***
   - ***"Qual foi a cadeia de invocações que conduziu à invocação desta função?" implica a reificação do* stack.**
   - ***"Quais são as variáveis livres desta função?" implica a reificação do ambiente léxico.***

2. (2.0) A linguagem Java possui várias capacidades de reflexão. Dê dois exemplos dessas capacidades e também dois exemplos de capacidades reflexivas que a linguagem Java não tem.

   Em Java é possível sabermos quais os métodos que são implementados por uma classe e é possível fazermos a invocação desses métodos a partir do seu nome e tipo dos parâmetros.

   Em Java não é possível saber que* errors handlers *estão activos num determinado instante nem é possível redefinir métodos.

3. (1.0) Considere o seguinte fragmento de programa Java:

```
MyClassLoader myClassLoader = new MyClassLoader();
Class boxClass = myClassLoader.loadClass("Box");
Object obj = boxClass.newInstance();
Box box = (Box)obj;
```

O que é que acontece se o executarmos? Explique.

O fragmento de programa cria um novo* class loader*, emprega-o para carregar a classe* `Box` *e cria uma instância dessa classe. Finalmente, tenta atribuir essa instância a uma variável do tipo* `Box`*, fazendo o* cast *necessário. Acontece que o tipo* `Box` *referido no* cast *foi carregado por um* class loader *diferente daquele que está a ser criado e, consequentemente, a classe* `Box` *para onde se está a fazer o* cast *é considerada diferente da classe* `Box` *que foi dinamicamente carregada e instanciada. Em resultado desta diferença, será gerado um erro.

4. (3.0) Traduza as seguintes expressões de Common Lisp para Java, de forma tão aproximada quanto lhe for possível (não precisa de se preocupar com o tratamento de excepções):

   (a) (1.0) `(make-instance 'foo)`

```
new Foo()
```

   (b) (1.0) `(make-instance foo)`

```
Class.forName(foo).newInstance()
```

   (c) (1.0) `(make-instance foo :bar "Baz")`

```
Class.forName(foo).
  getConstructor(new Class[] { String.class }).
    newInstance(new Object[] { "Baz" })
```

5. (1.0) A linguagem Common Lisp permite classes com herança múltipla. Neste contexto, explique o que é a *lista de precedências de classes* e refira para que serve.

A lista de precedências da classe C é uma ordenação total do conjunto contendo C e todas as suas superclasses, da mais específica para a menos específica. A ordenação da lista de precedências da classe C é consistente com a ordenação local de superclasses directas presente na definição de C e serve para a necessária ordenação dos métodos que é realizada pelas combinações de métodos empregue na criação de métodos efectivos.

6. (2.0) Considere o seguinte programa Java:

```
interface I1 { }

interface I2 { }

class Bar implements I1, I2 { }

public class Foo {

    public static void foo(I1 i1) {
        System.out.println(" Hello ");
    }

    public static void foo(I2 i2) {
        System.out.println(" World ");
    }

    public static void main(String[] args) {
        foo(new Bar());
    }
}
```

Infelizmente, o programa não é compilável. Porquê?

Que alterações faria à semântica da linguagem Java para que este programa fosse compilável? Explique.

Qual seria o resultado do programa para a semântica que propôs?

O programa não é compilável pois existe uma ambiguidade: ambos os métodos `foo` ***são aplicáveis e não existe um mais específico que o outro.***

Para que o programa fosse compilável seria necessário resolver a ambiguidade, por exemplo empregando uma lista de precedências de interfaces semelhante à que é empregue em CLOS.

Com esta semântica, o programa escreveria `Hello .`

7. (1.0) A linguagem CLOS especifica um conjunto de *procotolos de meta-objecto*. Para que servem? Escolha um desses protocolos e explique-o.

Os protocolos de meta-objecto servem para intermediar a manipulação dos objectos através dos correspondentes meta-objectos. Como exemplo de um protocolo de meta-objecto podemos considerar o acesso a um slot de um objecto que é feito pela função `slot-value` ***mas que está intermediada pela função genérica*** `slot-value-using-class` ***particularizada na metaclasse do objecto. Para implementar um comportamento diferente para o acesso a um slot, podemos definir uma nova metaclasse e particularizamos a função genérica*** `slot-value-using-class` ***para essa metaclasse de modo a implementar o comportamento pretendido.***

8. (1.0) Considere uma hipotética extensão da linguagem Java que emprega um mecanismo de combinação de métodos semelhante à *combinação standard de métodos* existente em CLOS. Neste mecanismo, os métodos não estáticos cujo nome tem "`before_`" ou "`after_`" como prefixo são considerados auxiliares e todos os restantes métodos não estáticos são considerados primários. A título de exemplo, considere o seguinte programa escrito nesta extensão de Java:

```
class Foo {
    void xpto() { System.out.println("1"); }
}
class Bar extends Foo {
    void xpto() { System.out.println("2"); }
    void before_xpto() { System.out.println("3"); }
}
class Baz extends Bar {
    void after_xpto() { System.out.println("4"); }
    void xpto() { System.out.println("5"); }
    void before_xpto() { System.out.println("6"); }
}
```

Tendo em conta a semântica da *combinação standard de métodos*, qual é o output do seguinte fragmento?

```
Foo[] foos = new Foo[] { new Foo(), new Bar(), new Baz() };
for (Foo foo : foos) {
    System.out.println(-----);
    foo.xpto();
}
```

```
-----
1
-----
3
2
-----
6
3
5
4
```

9. (2.0) Tendo em conta a semântica proposta na pergunta anterior, proponha um mecanismo de intercessão baseado em Javassist que permita implementar a combinação standard de métodos descrita. 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.

10. (3.0) Considere o seguinte aspecto definido em AspectJ:

```
aspect ShapeListening {
    private Vector<Screen> Shape.listeners = new Vector<Screen>();

    public void Shape.addListener(Screen sc) {
        listeners.add(sc);
    }

    pointcut changeShape(Shape s):
      call(void Shape+.set*(..)) && target(s);

    after(Shape sh): changeShape(sh) {
        for(Screen sc : sh.listeners) {
            sc.redraw(sh);
        }
    }
}
```

(a) (1.0) Em AspectJ, um aspecto agrupa vários conceitos. Diga quais são e explique-os, assinalando-os no programa anterior.

(b) (1.0) O *pointcut* descrito no programa anterior descreve um conjunto de pontos no fluxo de controle de um programa. Descreva esse conjunto de forma tão exaustiva quanto conseguir.

(c) (1.0) O *pointcut* descrito no programa anterior tem um parâmetro `s`. Para que serve? Onde é atribuído o valor desse parâmetro?

11. (1.0) O que é um *avaliador meta-circular*? Explique.

Um avaliador (ou interpretador) de uma linguagem é um procedimento que, quando aplicado a uma expressão dessa linguagem realiza as operações descritas nessa expressão.

Um avaliador meta-circular é um avaliador definido na mesma linguagem que ele avalia.

12. (2.0) Defina uma *macro* `when` em Scheme (usando `syntax-rules` ou `syntax-case`) que permita escrever expressões condicionais como a que se apresenta à esquerda e obter, como expansão, a forma que se apresenta à direita:

```
(when (negative? x)
  (display "Interesting!")
  (newline)
 else
  (newline)
  (display "D'oh!")
  (newline))
```

$\Longrightarrow$

```
(if (negative? x)
  (begin
    (display "Interesting!")
    (newline))
  (begin
    (newline)
    (display "D'oh!")
    (newline)))
```

Note que o símbolo `else` é um elemento sintático que separa as expressões do consequente das expressões da alternativa. Pode assumir que ocorre um e um só `else` com, pelo menos, uma expressão subsequente.

```
(define-syntax when
  (syntax-rules ()
    ((when condicao exprs ...)
     (when-if condicao (exprs ...) ()))))

(define-syntax when-if
  (syntax-rules (else)
    ((when-if condicao (else) (then-exprs ...))
     (if condicao
         (begin then-exprs ...)
         #f))
    ((when-if condicao (else else-exprs ...) (then-exprs ..
     (if condicao
         (begin then-exprs ...)
         (begin else-exprs ...)))
    ((when-if condicao (expr exprs ...) (then-exprs ...))
     (when-if condicao (exprs ...) (then-exprs ... expr)))))
```