TÉCNICO LISBOA

Fundamentos da Programação

Segundo Teste

18 de Junho de 2014

15:00–16:30

1. (1.0) Diga quais as fases por que passa o desenvolvimento de um programa no modelo estudado e o que se faz em cada uma delas.

   Resposta:

   - *Análise do problema.* O programador, juntamente com o cliente, estuda o problema a resolver com o objectivo de determinar exactamente o que o programa deve fazer.
   - *Desenvolvimento de uma solução.* Determinação de como vai ser resolvido o problema. Desenvolvimento de um algoritmo e definição abstracta dos tipos de informaçãousados. Deve usar-se a metodologia do topo para a base.
   - *Codificação da solução.* Tradução do algoritmo para uma linguagem de programação, e implementação dos tipos de informação. Depuração, i.e., correcção de erros sintácticos e semânticos.
   - *Testes.* Definição de uma bateria de testes com o objectivo de "garantir" que o programa funciona correctamente em todas as situações possíveis.
   - *Manutenção.* Fase que decorre depois do programa estar em funcionamento. A manutenção é necessária por dois tipos de razões: a descoberta de erros ou a necessidadede introduzir modificações e actualizações nas especificações do programa.

2. (1.0) Explique o é a abstracção de dados e o que é violar as barreiras de abstracção.

   Resposta:

   A *abstracção de dados* consiste em separar o modo como os dados são utilizados do modo como os dados são representados. É definido um conjunto de operações básicas que conhecem e podem manipular a representação interna dos objectos de um tipo abstracto de informação. Todo o resto do programa deve usar esta operações para manipular os objectos do tipo, se o não fizerem e manipularem directamente a sua representação interna estão a violar as barreira de abstracção definidas pelo conjunto de operações básicas do tipo.

3. (1.0) Diga quais são as várias fases da metedologia de Tipos Abstractos de Informação.

   Resposta:

   Na metodologia dos tipos abstractos de informação são seguidos quatros passos sequenciais:

   (a) Identificação das operações básicas;

   (b) Axiomatização das operações básicas;

   (c) Escolha de uma representação para os elementos do tipo;

   (d) Realização das operações básicas para a representação escolhida.

4. (1.0) Considere as estruturas de informação cujo comportamento é caracterizado por LIFO e FIFO. Explique o significado destes termos e diga quais as estruturas de informação a que estão associados.

Resposta:

LIFO significa "Last In First Out" e está associado ao tipo pilha. FIFO significa "First In First Out" e está associado ao tipo fila.

5. (1.5) Escreva a função `f_linhas_com_pontos` cujo argumento é o nome de um ficheiro de texto, que devolve o número de linhas que terminam com um ponto final `'.'`. Por exemplo, se `'texto.txt'` corresponder ao nome do ficheiro com conteúdo:

```
Este é um ficheiro de texto com linhas com pontos.
E com linhas sem pontos
Fim.
```

podemos obter a interacção:

```
>>> f_linhas_com_pontos("texto.txt")
2
```

Resposta:

```
def f_linhas_com_pontos(fich):

    f = open(fich, 'r')
    linhas = f.readlines()
    f.close()

    res = 0
    for linha in linhas:
        if linha[-2:] == ".\n" or linha[-1] == ".":
            res = res + 1

    return res
```

6. (1.5) Escreva uma função em Python que que recebe uma cadeia de caracteres e calcula quantas vezes aparece cada um dos caracteres nessa cadeia de caracteres. A sua função deve devolver um dicionário cujas chaves são caracteres e os valores associados correspondem ao número de vezes que os caracteres aparecem na cadeia de caracteres. Por exemplo,

```
>>> conta_caracteres("Isto é um teste.")
{'u': 1, 't': 3, ' ': 3, 's': 2, 'm': 1, 'é': 1, 'o': 1, '.': 1, 'I': 1, 'e': 2}
```

Resposta:

```
def conta_caracteres(cadeia):

    res = {}
    for c in cadeia:
        if c in res:
            res[c] = res[c] +  1
        else:
            res[c] = 1

    return res
```

7. (1.5) Escreva uma função recursiva `numero_ocorrencias_lista` que recebe uma lista e um número, e devolve o número de vezes que o número ocorre na lista e nas suas sublistas, se existirem. Por exemplo:

```
>>> numero_ocorrencias_lista([4, 5, 6], 5)
1
>>> numero_ocorrencias_lista([3, [[3], 5], 7, 3, [2, 3]], 3)
4
```

Resposta:

```
def numero_ocorrencias_lista(lst, num):
    if lst == []:
        return 0

    elif lst[0] == num:
        return 1 + numero_ocorrencias_lista(lst[1:], num)

    elif  isinstance  (lst[0],  list):
        return (numero_ocorrencias_lista(lst[0], num) \
                + numero_ocorrencias_lista(lst[1:], num))

    else:
        return numero_ocorrencias_lista(lst[1:], num)
```

8. Escreva uma função `conta_pares_tuplo` que recebe um tuplo de inteiros e devolve o número de elementos pares no tuplo.

   a. (1.0) Usando recursão com operações adiadas;

   Resposta:

```
def conta_pares_tuplo(t):
    if t == ():
        return 0
    elif t[0] % 2 == 0:
        return 1 + conta_pares_tuplo(t[1:])
    else:
        return conta_pares_tuplo(t[1:])
```

   b. (1.0) Usando recursão de cauda;

   Resposta:

```
def conta_pares_tuplo(t):

    def conta_aux(t, cont):
        if t == ():
            return cont
        elif t[0] % 2 == 0:
            return conta_aux(t[1:], cont + 1)
        else:
            return conta_aux(t[1:], cont)

   return conta_aux(t, 0)
```

   c. (1.0) Usando um processo iterativo.

   Resposta:

```
def conta_pares_tuplo(t):
    cont = 0
    for el in t:
        if el % 2 == 0:
            cont = cont + 1

    return cont
```

9. Suponha que desejava criar o tipo *relogio* em Python. Suponha que o tipo é caracterizado por um número que representa as horas (um inteiro entre 0 e 23) e um segundo número que representa os minutos (um inteiro entre 0 e 59).

   a. (1,5) Especifique as operações básicas para o tipo *relogio*.

   Resposta:

```
(a) Construtor:
        • cria_relogio : {n ∈ Z : 0 ≤ n ≤ 23} × {n ∈ Z: 0 ≤ n ≤ 59} → relogio
          cria_relogio(h, m) tem como valor o relógio correspondente a h horas,
          e m minutos.
(b) Selectores:
        • horas : relogio → {n ∈ Z : 0 ≤ n ≤ 23}
          horas(rel) tem como valor o número de horas do relógio rel.
        • minutos : relogio -> {n ∈ Z : 0 ≤ n ≤ 59}
          minutos(rel) tem como valor o número de minutos do relógio rel.

(c) Reconhecedor:
        • e_relogio : universal → lógico
          e_relogio(arg) tem o valor verdadeiro se arg for do tipo relógio e
          falso caso contrário.

(d) Testes:
        • relogios_iguais : relogio × relogio → lógico
          relogios_iguais(r1, r2) tem o valor verdadeiro se os relógios r1 e r2
          representarem as mesmas horas e minutos e falso caso contrário.
```

   b. (0,5) Escolha uma representação interna para o tipo *relogio* usando dicionários.

   Resposta:

```
Um elemento do tipo relogio será representado por um dicionário com dois pares
chave/valor, em que à chave 'H' está associado o valor correspondente às horas e
à chave 'M' está associado o valor correspondente aos minutos.
```

   c. (1,0) Escreva em Python os construtores e os selectores de acordo com a representação escolhida.

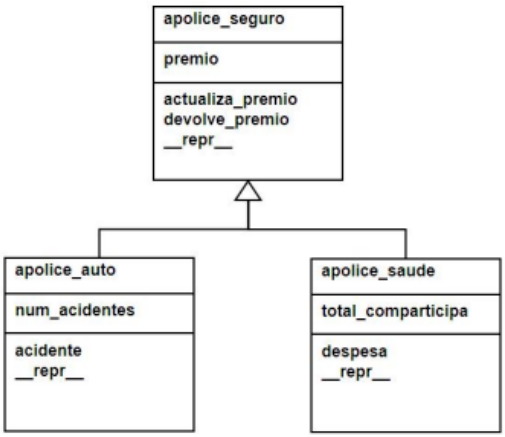
   Resposta:

```
def cria_relogio(h, m):
    if (isinstance (h, int) and isinstance(m, int) \
        and (0 <= h <= 23) and (0 <= m <= 59)):
        return {'H' : h, 'M' : m}
    else:
        raise ValueError \
              ('cria_relogio: args devem ser inteiros positivos entre \
                0 e 23 e entre 0 e 59')

def horas(rel):
    return rel['H']

def minutos(rel):
    return rel['M']
```

10. Considere a seguinte hierarquia de classes que indica que há dois tipos de apólices de seguro – as apólices do ramo automóvel e as apólices de saúde:



a. (2,0) Implemente a classe *apolice_seguro* de acordo com o diagrama apresentado e a descrição dos seus métodos:

- `apolice_seguro(valor)`, cria uma nova apólice com o valor indicado para o prémio;
- `actualiza_premio(valor)`, atribui o novo valor indicado para o prémio;
- `devolve_premio()`, devolve o valor do prémio da apólice;
- `__repr__`, método que o Python usa para apresentar a representação externa. Por exemplo,

```
>>> apolice_seguro(1000)
premio: 1000.0
```

- Considere o seguinte exemplo de interacção,

```
>>> ap_001 = apolice_seguro(1000)
>>> ap_001
premio: 1000.0
>>> ap_001.actualiza_premio(1200)
>>> ap_001.devolve_premio()
1200.0
```

Resposta:

```
class apolice_seguro :

    def __init__(self, valor):
        if isinstance (valor, (int, float)) and valor >= 0:
            self.premio = valor * 1.0
        else:
            raise ValueError('apolice_seguro: arg deve ser positivo')

    def actualiza_premio (self, valor):
        self.premio = valor * 1.0

    def devolve_premio(self):
        return self.premio

    def __repr__(self):
        return 'premio:' + str(self.premio)
```

b. (2,0) Implemente a classe *apolice_auto* de acordo com o diagrama apresentado e a descrição dos seus métodos:

- `acidente()`, permite a comunicação de um acidente, o que incrementa o número de acidentes e faz aumentar o prémio em 20 por cento.
- Método que o Python usa para apresentar a representação externa. Por exemplo,

```
>>>  apolice_auto(1000)
premio: 1000.0 num de acidentes: 0
```

- Considere o seguinte exemplo de interacção,

```
>>> ap_001 = apolice_auto(1000)
>>> ap_001
premio: 1000.0 num de acidentes: 0
>>> ap_001.acidente()
>>> ap_001
premio: 1200.0 num de acidentes: 1
>>> ap_001.acidente()
>>> ap_001
premio: 1440.0 num de acidentes: 2
```

Resposta:

```
class apolice_auto(apolice_seguro):

    def __init__(self, valor):
        if isinstance (valor, (int, float)) and valor >= 0:
            self.premio = valor * 1.0
            self.acidentes = 0
        else:
            raise ValueError('apolice_auto: arg deve ser positivo')

    def acidente(self):
        self.premio = self.premio * 1.2
        self.acidentes = self.acidentes + 1

    def __repr__(self):
        return   'premio: '    + str(self.premio) \
               + ' acidentes: ' + str(self.acidentes)
```

11. (1,5) Considere que tem disponíveis as operações básicas do tipo abstracto árvore: *nova_arv*, *cria_arv*, *raiz*, *arv_esq*, *arv_dir*, *arv_vazia*, *arv_iguais*, *escreve_arv*.

    Escreva uma função `arv_soma_pares` que recebe como argumento uma árvore de números e devolve a soma dos números pares representados na árvore. Por exemplo,

```
>>> arv1 = cria_arv(6, \
                    cria_arv(2, nova_arv(), nova_arv()), \
                    cria_arv(13, nova_arv(), nova_arv()))
>>> arv_soma_pares(arv1)
8
```

    Resposta:

```
def arv_soma_pares(arv):
    if arv_vazia(arv):
        return 0

    elif raiz(arv) % 2 == 0:
        return raiz(arv) + \
               arv_soma_pares(arv_esq(arv)) + \
               arv_soma_pares(arv_dir(arv))

    else:
        return arv_soma_pares(arv_esq(arv)) + \
               arv_soma_pares(arv_dir(arv))
```