Segunda Prova de MAB 225 — Computação II

Fabio Mascarenhas

6 de Julho de 2015

A prova é individual e sem consulta. Responda as questões na folha de respostas, a lápis ou a caneta. Se tiver qualquer dúvida consulte o professor.

Nome: _____

DRE: _____

Questão:	1	2	Total
Pontos:	7	3	10
Nota:			

1. Um *filtro* de linha de comando é um programa que lê linhas do console, faz algo com elas (reordena as linhas, elimina linhas duplicadas, converte para maiúsculas ou minúsculas, remove acentos, etc.), e escreve sua saída no console. Muitos programas de linha de comando são filtros, alguns bastante complexos.

   A classe *abstrata* a seguir modela filtros de linha de comando simples. Uma linha da entrada pode gerar zero ou mais linhas na saída, e o tipo de processamento fica a cargo da subclasse de `Filtro`:

```
class Filtro:
  def filtra(self):
    linha = raw_input()
    while True:
      for s in self.processa(linha):
        print s
      linha = raw_input()

  def processa(linha):
    raise NotImplementedError()
```

   (a) (2 pontos) A função `raw_input` lança uma exceção do tipo `EOFError` quando não existem mais linhas para processar. Conserte o método `filtra` para capturar esse erro, e fazer ele interromper a filtragem saindo do método.

   (b) (3 pontos) Crie uma subclasse *concreta* de `Filtro` chamada `FiltroUnico`, que elimina da saída linhas que já apareceram antes. Dica: use uma lista ou um dicionário para guardar as linhas que já foram vistas pelo filtro.

   (c) (2 pontos) Crie uma subclasse *abstrata* de `Filtro` chamada `FiltroRecupera` que redefine `filtra` para capturar outras exceções que aconteçam durante a filtragem, deixando o método `processa`

abstrato. Qualquer exceção deve ser passada para um novo método abstrato `erro`, que recebe o objeto da exceção e retorna `True` se a filtragem deve continuar a `False` se ela deve ser interrompida. Para obter 100% da questão a nova implementação de `filtra` em `FiltroRecupera` deve usar a implementação da superclasse `Filtro`.

2. (3 pontos) Em uma interface gráfica como a nossa biblioteca `gui` ou o `Tkinter`, qualquer computação demorada (como processar uma imagem, ou se comunicar com outro computador) que façamos trava a interface: enquanto o computador está ocupado ele não responde a eventos, animações param, corre-se até o risco do sistema operacional reclamar que a "aplicação não está respondendo". Uma solução é fazer esse tipo de computação com uma *tarefa assíncrona*, que executa a computação em sem travar a interface. A classe *abstrata* abaixo dá o esqueleto de uma classe para definir tarefas assíncronas:

```
class TarefaAssincrona:
  def rodar(self):
    # dispara uma outra linha de execução para
    # chamar o método executa, e não trava a interface
    # se o método executa terminar sem erros, o
    # método sucesso é chamado com o valor de retorno
    # de executa (ele pode, por exemplo, atualizar a interface)
    # se o método executa disparar uma exceção o método
    # falha é chamado com essa exceção

  def executa(self):
    # a execução desse método pode demorar o tempo que for preciso
    raise NotImplementedError()

  def sucesso(self, retorno):
    raise NotImplementedError()

  def falha(self, erro):
    raise NotImplementedError()
```

O método `rodar` é concreto, e sua implementação não foi dada pois ela depende de detalhes do sistema que estamos usando.

Defina uma classe *abstrata* `TarefaAssincronaSujeito` derivada de `TarefaAssincrona` na qual o método `executa` continua abstrato, mas os métodos `sucesso` e `falha` são concretos. Essa classe mantém internamente uma lista de *observadores*, e no método `sucesso` notifica todos eles chamando o método `mudou` de cada observador. O método `mudou` deve receber o valor do parâmetro `retorno`. O método `falha` deve lançar o parâmetro `erro`. Implemente também um método `observa` para cadastrar observadores, esse método recebe o observador que quer se cadastrar.

BOA SORTE!