Terceira Prova de MAB 240 2012.2 — Computação II

Fabio Mascarenhas

13 de Março de 2013

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:	4	6	10
Nota:			

1. A interface parametrizada abaixo representa sequências de itens de qualquer tipo. O método `proximo` retorna o próximo item da sequência, ou `null` se não há mais elementos:

```
public interface Seq<T> {
  T proximo();
}
```

A interface parametrizada abaixo representa funções de uma variável, quaisquer que sejam seus tipos de entrada e de saída:

```
public interface Funcao<X,Y> {
  Y aplica(X x);
}
```

(a) (2 pontos) Escreva a classe `Naturais` que implementa `Seq` e representa uma sequência de números naturais começando em um natural n passado ao construtor. A sequência é infinita. Um exemplo de uso dessa classe:

```
Naturais nats = new Naturais(1);
// Imprime os números de 1 a 10
for(int i = 0; i < 10; i++)
  System.out.println(nats.proximo());
// Imprime 11
System.out.println(nats.proximo());
```

(b) (2 pontos) Escreva a classe `FiltroSeq` que implementa `Seq` e filtra uma outra sequência de acordo com uma função filtro (implementação de `Funcao`), ambas passadas no construtor. A sequência filtrada só produz elementos que passam pelo filtro. Um exemplo de uso:

```
Naturais nats = new Naturais(1);
Funcao<Integer,Boolean> impar = new Funcao<Integer,Boolean>() {
  public Boolean aplica(Integer x) { return x % 2 == 1; }
}
FiltroSeq<Integer> impares = new FiltroSeq<Integer>(nats, impar);
// Imprime 1, 3, 5, 7, 9
for(int i = 0; i < 5; i++)
  System.out.println(impares.proximo());
```

O elemento `null` não deve ser passado à função filtro, e sim retornado diretamente.

2. Uma regra de negócio é um predicado que diz se um objeto qualquer se aplica àquela regra, que pode ser combinado com outras regras usando as operações booleanas *e* e *ou*, ou transformada na negação da regra. Podemos representar uma regra de negócio usando a classe *abstrata* `Regra<T>`, parametrizada pelo tipo de objeto sobre o qual a regra se aplica.

```
public abstract class Regra<T> {
  public abstract boolean seAplica(T obj);

  public Regra<T> e(Regra<T> outra) {
    return new RegraE<T>(this, outra);
  }

  public Regra<T> ou(Regra<T> outra) {
    return new RegraOu<T>(this, outra);
  }

  public Regra<T> nao() {
    return new RegraNao<T>(this);
  }
}
```

(a) (3 pontos) Implemente as classes *concretas* `RegraE`, `RegraOu` e `RegraNao`, usadas pela definição de `Regra`.

(b) (3 pontos) Considere a classe `Pagamento` abaixo, que representa um parte de um pagamento em um sistema comercial:

```
public class Pagamento {
  public boolean atrasado;
  public int notificacoes;
  public boolean noSPC;
}
```

Implemente as classes `Atrasado`, `Notificado`, e `NoSPC`, que são regras de negócio que respectivamente se aplicam a pagamentos em atraso, pagamentos com um número de notificações maior ou igual a n (passado no construtor de `Notificado`, e pagamentos que estão no SPC. Por exemplo, uma regra de negócio que se aplica a pagamentos que estão atrasados, com pelo menos três notificações de atraso, mas que ainda não estão no SPC, poderia ser construída com a seguinte expressão:

```
Regra<Pagamento> atrasado = new Atrasado();
Regra<Pagamento> notif3 = new Notificado(3);
Regra<Pagamento> nospc = new NoSPC();
Regra<Pagamento> regra = atrasado.e(notif3).e(nospc.nao());
```

BOA SORTE!