Programação Orientada pelos Objetos B

Departamento de Informática – FCT/UNL

2º semestre, 2013-14

Versão 3.0 (2014-04-24)

1. Classes

Apresentamos de seguida um conjunto de problemas introdutórios à programação orientada pelos objectos, no pressuposto de que os alunos estão familiarizados com conceitos básicos de programação.

1.1 Analogia com dispositivos físicos

Qualquer dispositivo físico suporta um conjunto de operações que lhe são específicas, o que permite o seu manuseamento por um utilizador interessado. O que permite manusear o dispositivo é a sua *interface*, ou seja, o conjunto de operações a que o dispositivo dá suporte.

Consideremos por exemplo a interface de um leitor de música MP3. As operações usuais são as seguintes: Previous, Play, Next, Back, Stop, Ok, Playing e Memory. Algumas das operações são acções que alteram o estado interno do objecto (dispositivo). Por exemplo, Next – muda para a música seguinte e Stop – pára a música corrente. Outras operações permitem apenas consultar o estado do objecto. É o caso de Playing – mostra no écran a música corrente, e Memory – mostra no ecrã a memória livre.

Em geral, todos os objectos apresentam dois tipos de operações na sua interface: as operações *modificadoras* e as operações *de consulta*.

Proponha um conjunto de operações, também conhecido como *interface*, para cada um dos seguintes dispositivos físicos:

1. Micro-ondas.
2. Máquina de operações bancárias (Multibanco).

Em ambos os casos indique claramente:

- Quais são as operações modificadoras.
- Quais são as operações de consulta.
- Que informação de estado julga ser necessário armazenar.

1.2 TwitterHappy

Defina em Java uma classe TwitterHappy, cujos objectos são registos de mensagens enviadas para o Twitter. Considera-se que uma pessoa está feliz, enquanto utilizador do Twitter, se enviar pelo menos 24 mensagens por dia, ou seja, em média uma por hora. Claro que ninguém tem tempo para estar sempre a mandar mensagens, mas o importante é manter o nível médio (se tiver 8 mensagens de manhã, 8 à tarde, e 8 à noite, não haverá problema).

Programe a sua classe e construa um programa principal para teste. O programa principal deve testar vários objectos da classe TwitterHappy de modo a confirmar se os objectos se comportam tal como esperado. As operações reconhecidas são as seguintes:

```
/* Começa um novo dia. */
 public void reset()
```

```
/* Lança mais uma mensagem no twitter de manhã. */
 public void twitMorning()

/* Lança mais uma mensagem no twitter à tarde. */
 public void twitAfternoon()

/* Lança mais uma mensagem no twitter à noite. */
 public void twitEvening()

/* Devolve o número de twits enviados de manhã. */
 public int morningTwits()

/* Devolve o número de twits enviados à tarde. */
 public int afternoonTwits()

/* Devolve o número de twits enviados à noite. */
 public int  eveningTwits()

/* Devolve true se há, cumulativamente, pelo menos 8 mensagens
enviadas de manhã, 8 à tarde e 8 à noite. */
 public boolean isTwitterHappy()
```

Por exemplo: uma pessoa faz reset. Depois disto, tem 0 twits de manhã, 0 à tarde, 0 à noite. Não está TwitterHappy. Depois, faz 325 twits de manhã (faz 325 operações!). Continua a não estar TwitterHappy. Depois, faz mais 2 twits de manhã. Depois disto, tem 327 twits de manhã, 0 à tarde, 0 à noite. Depois, faz 10 twits à tarde. Depois disto, tem 327 twits de manhã, 10 à tarde, 0 à noite. Continua a não estar TwitterHappy. Depois, faz 12 twits à noite. Depois disto, tem 327 twits de manhã, 10 à tarde, 12 à noite. Está TwitterHappy. Faz reset. Passa a ter 0 twits de manhã, 0 à tarde, 0 à noite. Não está TwitterHappy.

1.3 Parque de estacionamento (*Problema extra*)

Um parque de estacionamento cobra o tempo de permanência do automóvel no par- que de acordo com a seguinte tabela:

- Até 1 hora inclusive : 0,60 euros / hora.
-] 1ª hora, 4ª hora] : 0,80 euros / hora.
- Horas seguintes : 1 euro / hora.

Para efeitos de cálculo, o tempo de permanência é fraccionado em intervalos de 15 minutos. Considera-se ainda que o tempo de permanência de um automóvel não excede 24h e que a contagem começa e acaba no mesmo dia.

Defina uma classe Parking, cujos objectos são parques de estacionamento para automóveis.

As operações reconhecidas são as seguintes:

```
/* Determina o tempo de permanência do automóvel no parque, dado o
momento de entrada e o momento de saída.
O resultado é devolvido em horas (por exemplo, se permanecer 1h30m,
o resultado devolvido será de 1.5). */
 public double
elapsedTime(int inHour, int inMin, int outHour, int outMin)
```

```
/* Procede ao pagamento da permanência do automóvel no parque, dado
o momento de entrada e o momento de saída.
Devolve o montante pago. */
 public double
pay (int inHour, int inMin, int outHour, int outMin)

/* Devolve o número de veículos que estiveram estacionados no parque
ao longo do dia (número de veículos que pagaram). */
public  int  getVehiclesOut()

/* Devolve a quantia já facturada pelo parque durante o dia. */
 public double getCash()

/* Inicia um novo dia no parque, apagando a informação existente
sobre o dia anterior. */
 public void reset()
```

Teste a classe implementada com vários objectos e verifique se se comportam tal como esperado.

1.4 Círculos (Problema extra)

Um círculo define-se por um ponto no plano (par de coordenadas <x, y>) – o centro e o comprimento do raio.

Defina em Java uma classe Circle cujos objectos têm a funcionalidade que se apresenta a seguir.

A construção do círculo pode ser feita de várias formas:

- Dados o seu centro e raio.
- Dado o seu raio e considerando o centro localizado na origem do plano (ponto de coordenadas (0,0)).
- Sem nenhum indicação em especial, o que significa que o centro será na origem do plano e o raio será unitário.

Note que é possível definir na mesma classe quantos construtores quisermos, para cobrir as várias maneiras pretendidas de criar os objectos. Todos os construtores têm o mesmo nome (o nome da classe) mas são distinguidos pelos parâmetros que possuem.

O conjunto de operações reconhecidas são as seguintes:

```
/* Devolve o perímetro do círculo. */
public double getPerimeter()

/* Devolve a área do círculo. */
public double getArea()

/* Devolve o comprimento do raio do círculo. */
public double getRadius()

/* Devolve a abcissa do centro. */
 public double getXCenter()

/* Devolve a ordenada do centro. */
public double getYCenter()

/* Devolve a menor abcissa de um ponto do círculo. */
public double getLeftMostX()
```

```
/* Devolve a maior abcissa de um ponto do círculo. */
public double getRightMostX()

/* Devolve a menor ordenada de um ponto do círculo. */
public double getBottomMostY()

/* Devolve a maior ordenada de um ponto do círculo. */
public double getTopMostY()

/* Indica se o ponto (x,y) se encontra no círculo. */
public boolean ptInCircle(double x, double y)

/* Desloca o círculo segundo o vector <dx,dy>. */
public void translate(double dx, double dy)

/* Devolve true se a circunferência intersecta a
circunferência de raio radius, centrada em (cx, cy). */
public boolean
intersects(double cx, double cy, double radius)

/* Devolve true se o círculo de raio radius, centrado em
(cx, cy) está contido no objecto. */
public boolean
includes(double cx, double cy, double radius)

/* Roda a circunferência em torno da origem dos eixos
coordenados. */
public void rotateAroundOrigin (double degrees)

/* Mudança de escala, aplicando o factor rate ao raio e
também às coordenadas do centro. */
public void scale(double rate)

/* Constrói e devolve uma String informativa sobre o
círculo. O formato da string é: Circulo centrado no nn
quadrante, no ponto (xx,yy) e de raio rr. */
public String toString()
```

Programe a sua classe e construa um programa principal para teste. O programa principal deve testar vários objectos da classe de modo a confirmar se os objectos se comportam tal como esperado.

1.5 Archer

Defina em Java uma classe Archer cujos objectos são os arqueiros do lendário Robin Hood. Estes arqueiros assaltam os homens do xerife de Nottingham, mas têm de comprar as flechas para os atacar.

Programe a sua classe e construa um programa principal para testar. O programa principal deve testar vários objectos da classe de modo a confirmar se os objectos se comportam tal como esperado.

Cada arqueiro caracteriza-se pelo número de flechas que ainda tem disponível, a sua pontaria (um número real), e pelo dinheiro que lhe resta. O preço das flechas é constante e igual a 5 euros.

Quando o arqueiro é criado, se não dissermos nada, o arqueiro começa com 0 flechas, 1.0 de pontaria e 20 euros. Em alternativa, podemos indicar os valores iniciais para o número de flechas, a pontaria e o dinheiro.

O arqueiro pode disparar flechas contra alvos que se encontram a uma determinada distância (leia-se, contra os malvados cobradores do Xerife de Nottingham). Se acertar, recebe um prémio em dinheiro (ou seja, fica com o dinheiro que cobrador do Xerife leva) e aumenta a sua pontaria; se falhar, não aumenta nem o dinheiro nem a pontaria.

As operações reconhecidas são as seguintes:

```
/* Consulta o número de flechas que restam. */
 public int getArrows()

/* Consulta a pontaria do arqueiro. */
public float getAccuracy()

/* Consulta o dinheiro que resta. */
 public int getMoney()

/* Determina quantas flechas pode comprar com o dinheiro que tem.
Lembre-se que as flechas custam 5 euros cada. Use uma constante. */
 public int howManyArrowsCanBuy()

/* Compra flechas na quantidade indicada. Cada flecha custa 5 Euros.
*/
 public void buyArrows(int howMany)

/* Calcula o sucesso do tiro. Um tiro tem sucesso se a razão entre a
pontaria do arqueiro e a distância for maior ou igual a 0.5f. A
função retorna false se o tiro falhou, true se acertou. */
public boolean successfulShot(int distance)

/* Dispara a flecha a uma determinada distância do alvo. Cada tiro
custa 1 flecha. Para saber se o tiro é ou não certeiro, use a
operação successfulShot(). Se acertar, adiciona o prémio, em
dinheiro, à sua bolsa e aumenta a sua pontaria em metade da
distância para o alvo. Quer acerte, quer falhe, gasta sempre uma
flecha. */
public void fireArrow(int distance, int prizeMoney)

/* Indica se o arqueiro ainda tem flechas para disparar. */
public boolean canFireArrow()

/* Indica se o arqueiro pode continuar na sua actividade de
salteador. A função deve retornar true se o arqueiro ainda tiver
flechas, ou dinheiro suficiente para as comprar. Caso contrário,
retorna false. */
public boolean canContinuePlaying()
```

1.6 Mapa do tesouro (Problema extra)

Todos conhecemos histórias de piratas e mapas do tesouro. Em geral, há um cofre enterrado algures numa ilha e são fornecidas indicações que podem levar à sua descoberta. Por exemplo:

1. Coloque-se ao monumento do Pirata Desconhecido virado para norte.
2. Dê 10 passos (para norte).
3. Vire à direita e dê 4 passos (para leste).
4. Vire à direita e dê 2 passos (para sul).
5. Chegou ao seu destino. Comece a escavar, o tesouro está debaixo dos seus pés

Neste exercício, queremos que programe um sistema deste género. Para o efeito, defina em Java uma classe TreasureMap de acordo com a especificação que se segue. Programe a sua classe e construa um programa principal para testar. O programa principal deve testar vários objectos da classe de modo a confirmar se os objectos se comportam tal como esperado.

Quando é criado um objecto da classe, o utilizador do mapa do tesouro está na posição x=0, y=0, virado para norte. O construtor deve receber dois argumentos, com as coordenadas <x, y> da localização do tesouro.

As operações reconhecidas são as seguintes:

```
/* Caminha steps passos, na direcção para norte. */
public void walkNorth (int steps)

/* Caminha steps passos, na direcção para leste. */
public void walkEast (int steps)

/* Caminha steps passos, na direcção para sul. */
public void walkSouth (int steps)

/* Caminha steps passos, na direcção para oeste. */
public void walkWest (int steps)

/* Escava no local onde se encontra. Devolve true, se estiver
precisamente no local do tesouro, ou false, caso contrário.
*/
public boolean dig()

/* Devolve a coordenada x da posição onde se encontra. Como já
deve ter reparado, este mapa funciona num tablet equipado com GPS
em que as coordenadas são dadas em passos. */
public int getXPos()

/* Devolve a coordenada y da posição onde se encontra. */
public int getYPos()

/* Devolve o sentido para onde se encontra virado: 0 = norte; 1
= leste; 2 = sul 3 = oeste. */
public int getCurrentDirection()
```

1.7 Mars Rover

Defina em Java uma classe MarsRover cujos objectos têm a funcionalidade a seguir indicada.

O MarsRover é um veículo de exploração do planeta Marte que se desloca num plano imaginário sobreposto à superfície do planeta, em que cada posição é indicada por um par de coordenadas (números reais). Uma das funcionalidades do MarsRover é medir as distâncias em linha recta entre vários pontos no terreno.

Cada objecto MarsRover conhece a sua posição no terreno e a sua direcção de deslocação, esta última definida por um ângulo α. Quando é criado, cada objecto MarsRover encontra-se na origem do plano (X=0,Y=0) e virado para leste ($\alpha = 0$ graus.)

Programe a sua classe e construa um programa principal para testar. O programa principal deve testar vários objectos da classe de modo a confirmar se os objectos se comportam tal como esperado.

As operações reconhecidas são as seguintes:

```
/* O Rover avança distance unidades na direcção corrente. */
public void moveForward(double distance)

/* O Rover vira-se para a direcção absoluta angle, com o
respectivo valor em graus. */
public void setHeading(double angle)

/* O Rover regista o ponto corrente como ponto base. */
public void mark()

/* Devolve o valor da coordenada X. */
public double getXPos()

/* Devolve o valor da coordenada Y. */
public double getYPos()
/* Consulta o valor da orientação do Rover. */
public double getHeading()

/* O Rover indica a distância (em linha recta) entre o último
ponto base marcado e a sua posição corrente. */
public double getDistance()
```

1.8 Sócio (Problema extra)

Construa uma aplicação para que uma colectividade possa gerir a realização de eleições para os seus órgãos sociais. Considere os seguintes requisitos:

- Os sócios têm um número de votos diferenciado consoante a sua categoria, de acordo com a tabela 1.1.
- Cada sócio deposita todos os seus votos numa única lista (assuma que existem apenas 3 listas, designadas por X, Y e Z).
- Ganha a lista com mais votos.
- Também pretendemos saber o número de votantes em cada lista.
- O sistema deve registar o número de votantes irregulares (que não pertencem a uma categoria existente).
- O sistema deve registar o número de votos nulos (votos em listas inexistentes).

Tabela 1.1: Número de votos por categoria.

A	B	C	D	E	F	G	H	I
25	22	19	16	13	10	7	4	1

Cada eleitor deve indicar a sua categoria e a lista em que vota. No final de todos os votantes exercerem o seu direito de voto, o programa escreve na consola:

- A votação das listas concorrentes X, Y e Z.
- O número de votantes nas listas concorrentes X, Y e Z.
- O número de votos nulos.
- O número de eleitores irregulares (os votos destes não se contabilizam, como é óbvio).
- A lista vencedora.

Teste todas as operações implementadas.

2. Múltiplas Classes

2.1 Agenda de contactos

Desenvolva um programa para manipulação de uma agenda de contactos. Para isso, implemente uma classe ContactBook cujos objectos criados são agendas de contactos. Uma agenda deve conter informação sobre contactos, o que significa que deve também definir e implementar uma classe Contact.

Considere que cada contacto caracteriza-se por um nome, um telefone e um email. Para criar objectos da classe Contact são necessários três argumentos: uma String que representa o nome do contacto; um int que representa o número de telefone; e uma String que representa o email.

A interface de utilização do programa deve ser feita através de comandos, vul- garmente designado por interpretador de comandos. O programa principal deve pois dar ao utilizador a possibilidade de execução de diversas operações numa agenda e quantas vezes quiser.

Os comandos de interacção com o utilizador são os seguintes:

> AC (adiciona um contacto à agenda, se ainda não existir)

> RC (remove um contacto da agenda, se este existir)

> GP (consulta o telefone de um contacto, se este existir)

> GE (consulta o email de um contacto, se este existir)

> SP (actualiza o telefone de um dado contacto, se este existir)

> SE (actualiza o e-mail de um dado contacto, se este existir)

> LC (lista todos os contactos existentes na agenda)

> Q (sair)

Estas operações pressupõem a identificação inequívoca do contacto em causa, através do respectivo nome. Por isso, o conjunto de operações suportadas não permite alterar o nome de um contacto, o que quer dizer que não haverá contactos iguais na agenda.

Agenda de Contactos. O conjunto de operações reconhecidas na classe ContactBook são as seguintes:

```
/* Devolve o telefone de um contacto associado a um nome. */
public int getPhone(String name)

/* Devolve o e-mail de um contacto associado a um nome. */
public String getEmail(String name)
/* Insere um novo contacto, dado o nome, o telefone e o email.
*/
public void addContact(String name, int phone, String email)

/* Remove o contacto associado a um nome. */
public void deleteContact(String name)
```

```
/* Altera o telefone de um contacto associado a um nome. */
public void setPhone(String name, int phone)

/* Altera o email de um contacto associado a um nome. */
public void setEmail(String name, String email)

/* Indica se existe ou não um contacto associado a um nome. */
public boolean hasContact(String name)

/* Devolve o número de contactos na agenda. */
public int numberOfContacts()
```

Contacto. O conjunto de operações reconhecidas na classe Contact são as seguintes:

```
/* Devolve o nome do contacto. */
public String getName()

/* Devolve o telefone do contacto. */
public int getPhone()

/* Devolve o email do contacto. */
public String getEmail()

/* Altera o telefone do contacto para phone. */
public void setPhone(int phone)

/* Altera o email do contacto para email. */
public void setEmail(String email)

/* Devolve a descrição do contacto (nome, telefone e email). */
public String toString()
```

Teste o programa. A título indicativo, apresentamos o seguinte exemplo de interacção:

```
> AC
> Joana Dias > 99999999
> Joana@fct.unl.pt Contact added.
> AC
> Joana Dias
> 22222222
> JoanaDias@fct.unl.pt Contact already exists.
> AC
> Joana Horas
> 91999999
> Joana@gmail.com Contact added.
> RC
> Joana Dias Contact removed.
> RC
> Joana Dias
Cannot remove contact.
> AC
> Joana Dias
> 99999999
> Joana@fct.unl.pt Contact added.
> GP
> Joana Dias 99999999
```

```
> GP
> Joana Meses
Contact does not exist.
> GE
> Joana Dias Joana@fct.unl.pt
> GE
> Joana Meses
Contact does not exist.
> SP
> Joana Dias
> 253253253
Contact updated.
> SP
> Joana Meses
> 253253253
Contact does not exist.
> GP
> Joana Dias 253253253
> SE
> Joana Dias
> JoanaEu@tu.ele.pt Contact updated.
> SE
> Joana Meses
> JoanaEu@tu.ele.pt Contact does not exist.
> GE
> Joana Dias JoanaEu@tu.ele.pt
> LC
Joana Dias; JoanaEu@tu.ele.pt; 253253253 Joana Horas; Joana@gmail.com; 91999999
> RC
> Joana Dias Contact removed.
> RC
> Joana Horas Contact removed.
> LC
Contact book empty.
```

Por fim, chama-se a atenção para os seguintes aspectos relacionados com a implementação:

1. Para guardar n contactos de uma agenda, sugerimos que defina variáveis associadas a: (i) um vector de contactos com uma determinada capacidade máxima;
   (ii) um número que referencia o número de contactos existentes no vector.

2. Todas as operações da agenda necessitam de pesquisar um contacto dado o seu nome, pelo que deve existir uma operação auxiliar com esse objectivo, com utilização restrita aos métodos da classe. Sugerimos que implemente um método com visibilidade privada e tendo como argumento uma String com o nome a pesquisar. Como resultado, esse método devolverá a posição no vector de contactos do contacto associado ao nome (devolverá -1 se o nome não existir).
3. Consulte a classe String para obter informação sobre o método equals, tendo em vista a comparação de strings.

3. Máquina de café

3.1 Descrição do problema

Pretende-se desenvolver uma aplicação para simular o funcionamento de uma máquina de café em cápsulas, ainda que rudimentar se compararmos com as verdadeiras máquinas de café.

A aplicação deve gerir o funcionamento da máquina. Cada cápsula de café contém um de três tipos de café comercializados: *Espresso*, *Decaffeinato* ou *Lungo*. Exceptuando o caso do *Decaffeinato*, cada cápsula contém uma determinada quantidade de cafeína: 70 mg para um *Espresso* e 80 mg para um *Lungo*.

Para além do tipo de café e da quantidade de cafeína existente no interior da cápsula, existem mais dois elementos característicos de uma cápsula: o nome comercial e a cor.

Os cafés podem ser servidos curtos (25 ml), normais (40 ml) ou longos (110 ml). A máquina pode ser configurada para servir cafés usando qualquer uma destas quantidades, independentemente da cápsula utilizada (tipo *Espresso*, *Decaffeinato* ou *Lungo*).

Por motivos de reciclagem ambiental, pretendemos também inferir o valor residual de cafeína existente nas cápsulas após a sua utilização. Esse valor depende do valor de cafeína inicial, isto é, antes da utilização da cápsula, e da quantidade de água utilizada no respectivo café. Se o café foi curto então o café servido continha apenas 85% da cafeína existente na cápsula. Ou seja, consideramos que a cafeína residual retida na cápsula é de 15% do valor inicial existente na cápsula. Se o café foi normal, a cafeína utilizada foi de 90%. No caso do café longo, essa percentagem é de 95%.

A capacidade do reservatório de água da máquina é de 1 litro. O nível de água no reservatório diminui na proporção da água utilizada no café. Em qualquer momento, o reservatório pode ser cheio de água até à sua capacidade máxima. É claro que um café só pode ser servido se existir água suficiente na máquina.

Considere que, no máximo, a máquina só pode ser servir 1000 cafés. Considere também que, inicialmente, o reservatório de água está vazio e a quantidade usada para servir um café é de 40 ml (normal).

Concluindo, a aplicação deve permitir:

1. Servir um café. São indicados o tipo de café da cápsula, o nome comercial e a cor da cápsula. O café é servido com a quantidade de água previamente estabelecida na máquina. A operação falha se o tipo de café não corresponder a um dos três tipos acima referidos; se não existir água suficiente no reservatório; ou se a máquina já serviu o número máximo de cafés. Em caso de sucesso, deve ser mostrada uma mensagem indicando tal facto.

2. Estabelecer o nível de água curto para cafés. A operação é sempre bem sucedida. A operação mostra a quantidade de água a usar para cafés curtos.

3. Estabelecer o nível de água normal para cafés. A operação é sempre bem sucedida. A operação mostra a quantidade de água a usar para cafés normais.

4. Estabelecer o nível de água longo para cafés. A operação é sempre bem sucedida. A operação mostra a quantidade de água a usar para cafés longos.

5. Consultar o nível de água a usar em cada café. A operação é sempre bem sucedida.

6. Encher o reservatório de água até à sua capacidade máxima. A operação retorna a quantidade de água utilizada para o efeito, e é sempre bem sucedida.

7. Consultar o nível de água existente no reservatório. A operação é sempre bem sucedida.

8. Listar individualmente todas as cápsulas utilizadas, pela ordem pela qual os cafés foram servidos. Para cada uma das cápsulas em causa, é indicado o tipo de café e o nome comercial, bem como a quantidade de cafeína residual existente na cápsula.

9. Listar individualmente todas as cápsulas utilizadas de um determinado tipo de café, pela ordem pela qual os cafés foram servidos. Para cada uma das cápsulas em causa, é indicado o nome comercial e a quantidade de cafeína residual existente na cápsula.

3.2 Exemplo de interacção com a aplicação

Desenvolva a sua aplicação para que esta garanta, pelo menos, o modelo de interacção ilustrado no exemplo seguinte:

```
> CAPSULAS TODAS
Nao existem cafes para listar.
> RESERVATORIO
O reservatorio de agua contem 0 ml.
```

3.2. EXEMPLO DE INTERACÇÃO COM A APLICAÇÃO

> CAFE Lungo Fortissio verde
Nao e possivel servir o cafe indicado.
> AGUA
Foram adicionados 1000 ml de agua.
> CAFE Special Cosi azul
Tipo de cafe desconhecido.
> CURTO
Quantidade de agua a usar por cafe: 25 ml.
> CAFE Espresso Ristretto preto
Espresso Ristretto servido.
> CAFE Espresso Cosi castanho
Espresso Cosi servido.
> CAFE Decaffeinato Intenso vermelho
Decaffeinato Intenso servido.
> RESERVATORIO
O reservatorio de agua contem 925 ml.
> NORMAL
Quantidade de agua a usar por cafe: 40 ml.
> CAFE Espresso Ristretto preto
Espresso Ristretto servido.
> CAFE Espresso Volluto amarelo
Espresso Volluto servido.
> AGUA
Foram adicionados 155 ml de agua.
> LONGO
Quantidade de agua a usar por cafe: 110 ml.
> CAFE Lungo Fortissio verde
Lungo Fortissio servido.
> CAPSULAS TODAS
A sequencia das (6) capsulas utilizadas e a seguinte:
Espresso Ristretto preto. Cafeina residual: 11 mg.
Espresso Cosi castanho. Cafeina residual: 11 mg.
Decaffeinato Intenso vermelho. Cafeina residual: 0 mg.
Espresso Ristretto preto. Cafeina residual: 7 mg.
Espresso Volluto amarelo. Cafeina residual: 7 mg. Lungo
Fortissio verde. Cafeina residual: 4 mg.
> CAPSULAS Continente
Tipo de cafe desconhecido.
> CAPSULAS Espresso
A sequencia das (4) capsulas do tipo Espresso utilizadas e a seguinte:
Ristretto. Cafeina residual: 11 mg.
Cosi. Cafeina residual: 11 mg.
Ristretto. Cafeina residual: 7 mg.
Volluto. Cafeina residual: 7 mg.

> SAIR
Fim de execucao do programa.

3.3 Desenvolvimento

Desenvolva a sua aplicação de acordo com as seguintes fases:

1. Desenvolva o(s) interface(s) de suporte à aplicação. Utilize o conceito de polimorfismo. Desenhe um diagrama de classes e interfaces para ilustrar a proposta de modelação.
2. Implemente a aplicação.

Adicionalmente, apresente uma nova versão para a aplicação recorrendo aos conceitos de herança e de classe abstracta.