

Programação Orientada Pelos Objectos

# **Critérios de avaliação TP1 e TP2**



2

# Avaliação dos trabalhos práticos

# Avaliação da funcionalidade e qualidade do código

3

- Mooshak avalia a funcionalidade do código
  - Pontos entre 1 e 100
- Docentes avaliam a qualidade do código
  - 100 pontos no Mooshak garantem positiva no trabalho prático
    - Em geral, 100 pontos garantem no mínimo 13 valores
    - Mas há excepções. E.g. se o programa tiver apenas a classe Main, 100 pontos só vão garantir 10 valores.
  - São usados 15 critérios de avaliação (+ javadoc):

Básicos								Main		Estruturação				
declaração de vars através de classes (1.a)	cópia de objectos (1.b)	código repetido (1.c)	código confuso (1.d)	Faltam constantes (1.e)	ids sem significado (1.f)	vars globais (1.g)	encapsulamento (1.h)	interacção mix lógica (2.a)	Falta método por comando (2.b)	Erro na hierarquia de classes (3.a)	Faltam comandos na interface de topo (3.b)	Faltam entidades (3.c)	Faltam interfaces por classe (3.d)	Abuso de strings (3.e)
0.00	0.00	0.00	1.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
1.00	0.00	1.00	0.75	0.00	0.00	0.00	1.00	0.00	0.00	0.50	0.00	0.25	1.00	0.00

# Boas práticas de programação

4

- Código com comentários Javadoc
  - Métodos públicos comentados na interface

```
/**
 * Uma pessoa, com accoes associadas e personalidade consequente
 * @author Miguel Goulao / Adriano Lopes / Carla Ferreira
 */

public interface Person {

    /**
     * Verifica se a pessoa tem a accao <code>description</code>
     * @param description - descricao da accao.
     * @return devolve <code>true</code> se a pessoa tem a accao <code>description</code>
     */
    boolean hasAction(String description);

    /**
     * Acrescenta uma accao <code>description</code> a pessoa
     * @pre !hasAction(description)
     * @param description - descricao da accao.
     */
    void addAction(String description);
    ...
}
```

# Boas práticas de programação

5

- Código com comentários Javadoc
  - Variáveis de instância, construtor e métodos privados comentados na classe

```
/**
 * Uma pessoa, com accoes associadas e personalidade consequente
 * @author Miguel Goulao / Adriano Lopes / Carla Ferreira
 */
public class PersonClass implements Person {
    /**
     * Dimensao do vector de accoes, por omissao.
     */
    private static final int DEFAULT = 20;

    /**
     * Nome da pessoa.
     */
    private String name;

    /**
     * Accoes da pessoa.
     */
    private Action[] myActions;
    ...
}
```

# Boas práticas de programação

6

- **1.a)** Utilização de interfaces para declarar os tipos dos objectos:

```
Person owner;
```



```
PersonClass owner;
```



```
Array<Action> myActions;
```



```
Array<ActionClass> myActions;
```



```
ArrayClass<Action> myActions;
```



```
ArrayClass<ActionClass> myActions;
```



# Boas práticas de programação

7

## ○ 1.b) Evitar duplicação de informação

- E.g., evitar cópias desnecessárias de objectos

```
public interface SocialNetwork {  
    /**  
     * Adiciona 'a pessoa de nome name o amigo com o nome friend.  
     * @pre hasPerson(name) && hasPerson(friend) && !hasFriendship(name, friend)  
     * @param name nome da pessoa a adicionar o amigo.  
     * @param friend nome do amigo.  
     */  
    void setFriendship(String name, String friend);  
    ...  
}
```

```
public void setFriendship(String name, String friend) {  
    Person newFriend = network[indexOf(friend)];  
    Person person = network[indexOf(name)];  
    person.addFriend(newFriend);  
}
```



```
public void setFriendship(String name, String friend) {  
    Person newFriend = new PersonClass(friend);  
    Person person = network[indexOf(name)];  
    person.addFriend(newFriend);  
}
```






# Boas práticas de programação

8

## ○ 1.c) Evitar repetição de código

```
public void init() {
    current = 0;
    switch (type) { // Initializes Iterator based on User type
        case C: while (current < counter && !(user[current] instanceof Customer))
                current++;
                break;
        case P: while (current < counter && !(user[current] instanceof Provider))
                current++;
                break;
    }
}

public User next() {
    User temp = user[current++];
    switch (type) { // Calculates next user based on its type
        case C: while (current < counter && !(user[current] instanceof Customer))
                current++;
                break;
        case P: while (current < counter && !(user[current] instanceof Provider))
                current++;
                break;
    }
    return temp;
}
```





# Boas práticas de programação

9

## ○ 1.d) Evitar código confuso

```
public Vehicle transport(int max, int x, int y, String type) {
    Vehicle[] aux = new Vehicle[vehicles.length];
    Vehicle temp;
    float distance = 0;
    int counterAux = 0;
    boolean getOut = false;
    for (int i = 0; i < counter; i++)
        if (vehicles[i].getMaxCap() >= max)
            switch (type) {
                case G: if (vehicles[i] instanceof VehicleGoods) aux[counterAux++] = vehicles[i];
                        break;
                case P: if (vehicles[i] instanceof VehiclePeople) aux[counterAux++] = vehicles[i];
                        break;
            }
    for (int i = 1; i < counterAux; i++)
        for (int j = counterAux - 1; j >= i; j--)
            if (aux[j - 1].getCoordinates().distanceFromCoordinates(x, y) >
                aux[j].getCoordinates().distanceFromCoordinates(x, y)) {
                Vehicle tmp = aux[j - 1]; aux[j - 1] = aux[j]; aux[j] = tmp;
            }
    temp = aux[0];
    distance = temp.getCoordinates().distanceFromCoordinates(x, y);
    for (int i = 1; !getOut && i < counterAux; i++)
        if (distance == aux[i].getCoordinates().distanceFromCoordinates(x, y))
            if (aux[i].getMaxCap() < temp.getMaxCap()
                || (aux[i].getMaxCap() == temp.getMaxCap() && aux[i].getID() < temp.getID())) {
                temp = aux[i];
                distance = temp.getCoordinates().distanceFromCoordinates(x, y);
            }
        else getOut = true;
    return temp;
}
```



# Boas práticas de programação

10

- **1.e)** Definição de constantes quando necessárias
  - E.g., mensagens de output e comandos

```
// Comandos do utilizador
private static final String QUIT = "SAIR";
private static final String NEW = "NOVO";
private static final String ADD_FRIEND = "ADICIONA";
private static final String REM_FRIEND = "REMOVE";
private static final String DO = "FAZ";
private static final String VOTE = "VOTA";
private static final String GOOD = "BEM";
private static final String BAD = "MAL";
private static final String LIST = "LISTA";
...
// Feedback dado pelo programa
private static final String OK = "Ok";
private static final String BYE = "Adeus";
private static final String ALREADY_EXISTS = " ja existe";
private static final String DOES_NOT_EXIST = " nao existe";
private static final String NOT = " nao ";
private static final String ALREADY = " ja ";
```



# Boas práticas de programação

11

- **1.f)** Escolha de nomes apropriados para as variáveis, parâmetros, métodos, etc.

```
int d;  
// elapsed time in days  
int ds;  
int dsm;  
int faid;
```



```
int elapsedTimeInDays;  
int daysSinceCreation;  
int daysSinceModification;  
int fileAgeInDays;
```



# Boas práticas de programação

12

## ○ 1.g) Não usar variáveis globais

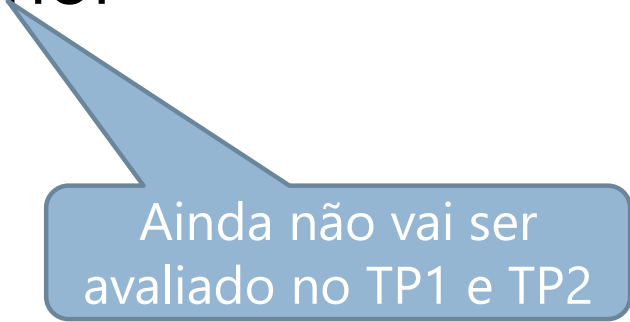
```
public class Main {  
    private static StupidFriendsBook fb = new StupidFriendsBookClass();  
  
    private static void commands() {  
        Scanner in = new Scanner(System.in);  
        String command = in.nextLine().toUpperCase();  
        while (!command.equals(QUIT)) {  
            switch (command) {  
                case Main.ADD_FRIEND: addFriend(in); break;  
                case Main.REM_FRIEND: removeFriend(in); break;  
                ...  
            }  
            command = in.nextLine().toUpperCase();  
        }  
        System.out.println(Main.BYE); in.close();  
    }  
    private static void addFriend(Scanner in) {  
        String name = in.nextLine();  
        if (!fb.hasFriend(name)) {  
            fb.addFriend(name);  
            System.out.println(Main.OK); }  
        else  
            System.out.println(name + Main.ALREADY_EXISTS);  
    }  
}
```



# Boas práticas de programação

13

- **1.h)** Garantir encapsulamento
  - Variáveis de instância privadas
  - Métodos não devem retornar membros de dados alteráveis do exterior



Ainda não vai ser  
avaliado no TP1 e TP2

# Boas práticas de programação

14

- **1.h) Garantir encapsulamento**
  - E.g. não ter métodos públicos que devolvem estruturas de dados

```
public interface IntSet {  
    public void insert(int x);  
    public void remove(int x);  
    public boolean subset(IntSet s);  
    public int size();  
    public Iterator<Integer> elements();  
}
```



```
public interface IntSet {  
    public void insert(int x);  
    public void remove(int x);  
    public boolean subset(IntSet s);  
    public int size();  
    public Array<Integer> elements();  
}
```




# Classe Main


15

## ○ 2.a) A interacção com o utilizador separada da lógica do domínio

```
private static void minArea(ShapesCollection shapes) {  
    if (shapes.isEmpty()) System.out.println(Main.IS_EMPTY);  
    else {  
        Shape shape = shapes.smallestArea();  
        System.out.printf(SHAPE_INFO, shape.getID(), ...);  
    }  
}
```



```
private static void minArea(ShapesCollection shapes) {  
    if (shapes.isEmpty()) System.out.println(Main.IS_EMPTY);  
    else {  
        Iterator it = shapes.allShapesIterator();  
        Shape min = it.next();  
        while (it.hasNext()) {  
            Shape shape = it.next();  
            if (shape.area() <= min) min = shape;  
        }  
        System.out.printf(SHAPE_INFO, min.getID(), ...);  
    }  
}
```





# Classe Main

16

- **2.a)** A interacção com o utilizador separada da lógica do domínio

## NOTA IMPORTANTE

O problema neste exemplo não é o uso do iterador, mas o facto de estarem a ser feitos cálculos na classe Main que deviam estar encapsulados num método da classe de topo.

```
private static void minArea(ShapesCollection shapes) {  
    if (shapes.isEmpty()) System.out.println(Main.IS_EMPTY);  
    else {  
        Iterator it = shapes.allShapesIterator();  
        Shape min = it.next();  
        while (it.hasNext()) {  
            Shape shape = it.next();  
            if (shape.area() <= min) min = shape;  
        }  
        System.out.printf(SHAPE_INFO,min.getID(),...);  
    }  
}
```



# Classe Main

17

- **2.b)** Correcta estruturação da classe Main
  - Método privado por cada comando

```
private static void commands() {
    StupidFriendsBook fb = new StupidFriendsBookClass();
    Scanner in = new Scanner(System.in);
    String command = in.nextLine().toUpperCase();
    while (!command.equals(QUIT)) {
        switch (command) {
            case Main.ADD_FRIEND: addFriend(in, fb); break;
            case Main.REM_FRIEND: removeFriend(in, fb); break;
            ...
        }
        command = in.nextLine().toUpperCase();
    }
    System.out.println(Main.BYE); in.close();
}

private static void addFriend(Scanner in, StupidFriendsBook fb) {
    String name = in.nextLine();
    if (!fb.hasFriend(name)) {
        fb.addFriend(name);
        System.out.println(Main.OK); }
    else System.out.println(name + Main.ALREADY_EXISTS);
}
```

# Qualidade do modelo de classes e interfaces

18

- **3.a)** Qualidade da hierarquia de classes (e.g. extensibilidade)
  - Critério dependente do trabalho prático
  - Usar, sempre que adequado, herança (de classes e interfaces) e classes abstractas

# Qualidade do modelo de classes e interfaces

19

- **3.b)** A interface de topo deve ter um método para cada comando
  - Devem também existir métodos na interface de topo para verificar as pré-condições

```
public interface SocialNetwork {  
    // Comandos  
    void registerPerson(String name, String email, String status);  
    void setFriendship(String name, String friend);  
    Iterator listFriends(String name);  
    String getStatus(String name);  
    void setStatus(String name, String status);  
    Iterator listPersons();  
    // Pre-condicoes  
    boolean hasPerson(String name);  
    boolean hasFriend(String name, String friend);  
}
```

# Qualidade do modelo de classes e interfaces

20

- **3.c)** Identificação das entidades necessárias
  - Critério dependente do trabalho prático

# Qualidade do modelo de classes e interfaces

21

- **3.d)** Em geral, deve existir uma interface para cada classe
  - Para as subclasses **pode** não ser necessária uma nova interface, caso estas não adicionem novos métodos.
  - Diferentes classes podem implementar uma mesma interface.

# Qualidade do modelo de classes e interfaces

22

- **3.e) Evitar abuso de Strings**
  - Não construir Strings nas classes do domínio
    - Strings são apenas construídas na classe Main
    - A exceção é o método toString, mas neste método devem ser usados métodos de consulta públicos
  - As listagens resultantes da execução de comandos da classe Main devem ser suportadas por iteradores