Ficha 1

```
/* Snippet 1*/
public Node {
    private int x, y;
    ....
    public move(int dx, int dy) {
        x += dx;
        y += dy;
    }
    public resetOrigin() { x = y = 0; }
}
public NodeMemento {
    int x, y;
    public NodeMemento(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public getX() { return x; }
    public getY() { return y; }
}
```

1. Relativo ao código em *Snippet 1*, em particular na classe `NodeMemento`, qual o *code smell* presente?

    - **A.** Primitive Obsession
    - **B.** Lazy Class
    - **C.** Data Class
    - **D.** Data Clump

Data Class – "A data class refers to a class that contains only fields and crude methods for accessing them (getters and setters). These are simply containers for data used by other classes. These classes don't contain any additional functionality and can't independently operate on the data that they own."

2. Relativo à resposta em **1** como deve proceder?

    - **A.** Remover a classe `NodeMemento`
    - **B.** Aplicar a técnica Inline Class integrando a classe `NodeMemento` na classe `Node`.
    - **C.** Não fazer nada, porque é um *code smell* que não representa perigo.
    - **D.** Adicionar o método `toString` à classe `NodeMemento`.

```
/* Snippet 2*/
public class A {
    public doSomething(int[] arr) {
        int x = 0;
        for(int i=0; i<arr.length; i++) {
            x += arr[i];
        }
        int r = x / arr.length;
        return arr[r];
    }
}
public class B {
    public doSomething(int[] arr) {
        int x = 0;
        for(int i=0; i<arr.length; i++) {
            x += arr[i];
        }
        int r = (int)(Math.random()*arr.length);
        return arr[r];
    }
}
```

3. Relativo ao código em *Snippet 2*, qual o *code smell* presente no código?

   - **A.** Duplicate code
   - **B.** Temporary Field
   - **C.** Primitive Obsession
   - **D.** Switch Statements

Duplicate code – "Two code fragments look almost identical."

4. Relativo ao código em *Snippet 2*, qual a técnica de refactoring que deve aplicar?

   - **A.** Replace Type Code with Strategy
   - **B.** Replace Conditional with Polymorphism
   - **C.** Form Template Method
   - **D.** Replace Delegation with Inheritance

Form Template Method – "If the duplicate code is similar but not completely identical, use Form Template Method."

```
/* Snippet 3*/
public FlightPlanner {
    public Graph<City, Flight> graph;
    ....
}
public static void main(String[] args) {
    FlightPlanner p = new FlightPlanner();
    Vertex<City> lisbon = p.graph.insertVertex(new City("Lisbon"));
    Vertex<City> oporto = p.graph.insertVertex(new City("Oporto"));
    p.graph.insertEdge(lisbon, oporto, new Flight("TP1024"));
}
```

5. Relativo ao código em *Snippet 3*, qual o *code smell* presente no código?

    - **A.** Primitive Obsession
    - **B.** Duplicate code
    - **C.** Temporary Field
    - **D.** Inappropriate intimacy

Inappropriate intimacy – "Keep a close eye on classes that spend too much time together. Good classes should **know as little about each other as possible**. Such classes are easier to maintain and reuse."

6. Relativo ao código em *Snippet 3*, qual a técnica de refactoring que deve aplicar?

    - **A.** Extract Class
    - **B.** Inline Method
    - **C.** Move Method
    - **D.** Hide Delegate

Hide Delegate - Another solution is to use **Extract Class** and **Hide Delegate** on the class to make the code relations "official".

```
/* Snippet 4*/
public Inventory /* v1 */* {
    private String[] names;
    private double[] prices;
    int count;
    ...
    public addItem(String name, double price) {
        names[count] = name;
        prices[count++] = price;
    }
}
public Inventory /* v2 */ {
    private Item[] items;
    int count;
    ...
    public addItem(String name, double price) {
        items[count++] = new Item(name, price);
    }
}
public Inventory /* v3 */{
    private List<Item> items;
    ...
    public addItem(String name, double price) {
        items.add( new Item(name, price) );
    }
}
```

7. Relativo ao código em *Snippet 4*, em particular na classe `Inventory` qual o *code smell* presente na versão *V1* que levou à versão *V2*?

    - **A.** Temporary Field
    - **B.** Data Clump
    - **C.** Data Class
    - **D.** Lazy Class

Data clump – "If you want to make sure whether or not some data is a data clump, just delete one of the data values and see whether the other values still make sense. If this is not the case, this is a good sign that this group of variables should be combined into an object."

8. Relativo ao código em *Snippet 4* e à versão *V2* da classe `Inventory`, foi sugerido a presença do *code smell* **primitive obsession**. Qual a técnica de refactoring que foi aplicada para chegar a *V3* ?

    - **A.** Replace Inheritance with Delegation
    - **B.** Replace Data with Object
    - **C.** Extract Class
    - **D.** Nenhuma das anteriores

Nenhuma das anteriores - "If there are arrays among the variables, use Replace Array with Object.". Array de items passa para uma coleção (objecto).

```
/* Snippet 5*/
public class Deque<T> extends LinkedList<T> {
    ...
    public insertFirst(T elem) {
        add(0, elem);
    }
}
public class Deque_Refactored<T> {
    private LinkedList<T> adaptee;
    ...
    public insertFirst(T elem) {
        adaptee.add(0, elem);
    }
}
```

9. Relativo ao código em *Snippet 5* qual o *code smell* que motivou o processo de refactoring?

    - **A.** Refused Bequest
    - **B.** Data Clump
    - **C.** Primitive Obsession
    - **D.** Innapropriate Intimacy

Refused Bequest – "If a subclass uses only some of the methods and properties inherited from its parents, the hierarchy is off-kilter. The unneeded methods may simply go unused or be redefined and give off exceptions."

10. Relativo ao código em *Snippet 5*, qual a técnica de refactoring que foi aplicada?
    - **A.** Replace Data with Object
    - **B.** Replace Inheritance with Delegation
    - **C.** Extract Class
    - **D.** Nenhuma das anteriores

Replace Inheritance with Delegation – "If inheritance makes no sense and the subclass really does have nothing in common with the superclass, eliminate inheritance in favor of Replace Inheritance with Delegation."

Ficha 2

```
/* Snippet 1*/
public int getScore() {
  int res1, res2;
  res1 = (int)(Math.random()*6) + 1;
  do{
  res2 = (int)(Math.random()*6) + 1;
    } while (res1==res2)

  return res1+res2;
}
//REFATORED
public int getScore() {
  int res1, res2;
  res1 = rollDie()
  do{
  res2 = rollDie();
    } while (res1==res2)

    return res1+res2;
}
```

1. Qual o BAD SMELL detetado no código apresentado ?

    - **A.** Primitive Obsession
    - **B.** Temporary Field
    - **C.** Duplicate code
    - **D.** Nenhum dos anteriores

Duplicate code – "Two code fragments look almost identical."

2. Qual a Técnica de Refactoring Apresentada?

    - **A.** Inline Method
    - **B.** Move Method
    - **C.** Extract Method
    - **D.** Nenhuma das anteriores

Extract Method – "You have a code fragment that can be grouped together."

```
/* Snippet 2*/
public class Shape {
  private static final int CIRCLE = 1;
  private static final int SQUARE = 2;
  private static final int TRIANGLE = 3;
  //...
  double getArea() {
    switch(type) {
      case CIRCLE:
         return PI * r * r;
      case SQUARE:
        return l * l;
      case TRIANGLE:
        return  l * b/2;
      default: throw new RunTimeException("Invalid Type");
}
  }
```

3. Qual o BAD SMELL detetado no código apresentado ?

- **A.** Primitive Obsession
- **B.** Switch Statements
- **C.** Refused Bequest
- **D.** Temporary Field

Switch Statements – "You have a complex `switch` operator or sequence of `if` statements."

4. Qual a Técnica de Refactoring que deve aplicar?

- **A.** Form Template Method
- **B.** Replace Type Code with Strategy
- **C.** Replace Conditional with Polymorphism
- **D.** Replace Delegation with Heritance

Replace Type Code with Strategy – "After specifying the inheritance structure, use Replace Conditional with Polymorphism."

```
/* Snippet 3*/
public class Group extends HashSet<Person> {
    private String name;
    private int maxNumber;
    private Person manager;

    public Group(String name, int maxNumber, Person manager) {
        this.name = name;
        this.maxNumber = maxNumber;
        this.manager = manager;
    }

    public boolean remove(Person p) {
      if(p.equals(manager))
          return false;
      else
          return super.remove(p);
     }
}
```

5. Qual o BAD SMELL detetado no código apresentado?

    - **A.** Primitive Obsession
    - **B.** Innapropriate Intimacy
    - **C.** Refused Bequest
    - **D.** Data Clump

Refused Bequest – "If a subclass uses only some of the methods and properties inherited from its parents, the hierarchy is off-kilter. The unneeded methods may simply go unused or be redefined and give off exceptions."

6. Qual a Tecnica de Refatoring que deve aplicar?

    - A. Replace Data with Object
    - B. Extract Class
    - C. Extract Method
    - **D.** Replace Heritance with Delegation

Replace Inheritance with Delegation – "If inheritance makes no sense and the subclass really does have nothing in common with the superclass, eliminate inheritance in favor of Replace Inheritance with Delegation."

```
public class Product {
    private String name;
    private int cod;
    private int price;
    private int tax;

public Product(String name, int cod, int preco, int iva) {
        this.name = name;
        this.cod = cod;
        this.price = preco;
        this.tax = iva;
    }

    public String getName() {
        return name;
    }
    public int getCod() {
        return cod;
    }
    public int getPrice() {
        return price;
    }
    public int getTax() {
        return tax;
    }
}

public class Item {
    private Product prod;
    private int quantity;

    public Item(Product prod, int quantity) {
        this.prod = prod;
        this.quantity = quantity;
    }
    public Product getProd() {
        return prod;
    }
    public int getQuantity() {
        return quantity;
    }
}
```

```
public class Order {
    private Date date;
    private ArrayList<Item> items;
    private double total;

    public Order(String customer, Date date) {
        this.customer = customer;
        this.date = date;
        this.items = new ArrayList();
    }

    public void addItem(Item item)    {
        items.add(item);
    }

    public double getTotal(){
        for(Item item: items)
              total+=item.getQuantity()*
                    item.getProd().getPrice()*(1+item.getProd().getTax()/100.0);
      return total;
    }

   public Item getItem(int i) {
     return items.get(i);
    }
}
```

7. Identifique o BAD SMELL que se conseguem claramente identificar na classe Produto

    - **A.** Primitive Obsession
    - **B.** Lazy Class
    - **C.** Data Clump
    - **D.** Data Class

Lazy Class – "Perhaps a class was designed to be fully functional but after some of the refactoring it has become ridiculously small.

Or perhaps it was designed to support future development work that never got done."

8. Qual o BAD SMELL identificado no método getTotal() da classe Order.

    - **A.** Data Clump
    - **B.** Permitive Obsesssion
    - **C.** Duplicate Code
    - **D.** Message Chains

Message Chains – "In code you see a series of calls resembling `$a->b()->c()->d()`".

9. Para resolver o problema detetado em 7 a melhor opção é:

    - **A.** Remover a classe Product
    - **B.** Aplicar a técnica Inline Class integrando a classe Product na Classe Item.
    - **C.** Não fazer nada, porque é um BAD SMELL que não representa perigo.
    - **D.** Adicionar o método ToString à classe Produto.

Inline Class – "Move all features from the class to another one."

1. Para resolver o problema detetado em P8 a melhor opção é:

    - **A.** Definir novos métodos na classe Item e Produto, usando a técnica Hide Delegate.
    - **B.** Definir um novo método na classe Produto – calculateFinalPrice() que calcula o preço liquido de um produto – usando a técnica Hide Delegate.
    - **C.** Usar a técnica extract method, para definir um método calculateFinalPrice(Product p), na classe Order, e assim, resolver o problema de duplicate code.
    - **D.** Usar a técnica de MoveMethod, movendo o método getTotal para a classe Item.

Ficha 3

```
/* Snippet 1*/
public class Employee {
    int INTERN = 1;
    int FULLTIME = 2;
    int PARTTIME = 3;
    //...
    double getSalary() {
        switch(type) {
            case INTERN:
            // Long calculation 1
            return res1;
            case FULLTIME:
            // Long calculation 2
            return res2;
            case PARTTIME:
            // Long calculation 3
            return res3;
        }
    throw new RuntimeException();
    }
}
```

1. Relativo ao código em Snippet 1 P1 - Qual das técnicas de refactoring é a mais adequada para lidar com os vários casos no método getSalary()?

    - A. Extract method
    - B. Replace conditional with polymorphism
    - C. Replace delegation with heritance.
    - D. Nenhuma das anteriores

```
/* Snippet 2*/
    public int getScore1() {
        int res;
        res = (int)(Math.random()*6) + 1;
        dice[0].setFaceValue(res);
        res = (int)(Math.random()*6) + 1;
        dice[1].setFaceValue(res);
        int score = dice[0].getFaceValue() +
        dice[1].getFaceValue();
        return score;
    }
    public int getScore2() {
        int res;
        res = rollDie();
        dice[0].setFaceValue(res);
        res = rollDie();
        dice[1].setFaceValue(res);
        int score = dice[0].getFaceValue() +
        dice[1].getFaceValue();
        return score;
    }
```

2. Qual das técnicas de refactoring foi utilizada no método getScore1 de modo a obter o método getScore2?

- A. Extract method
- B. Inline method
- C. Remove duplicated code
- D. Nenhuma das anteriores

```
/* Snippet 3*/
    public class Student {
        String name;
        int id;
        // Student methods
    }
    public class Course {
        ArrayList<Student> studentList;
        // Course methods
    }

    public void main() {
        Student s1 = new Student("Ana");
        Student s2 = new Student("Zé");
        Course pa = new Course();
        pa.studentList.add(s1);
        pa.studentList.add(s2);
}
```

Relativo ao código em Snippet 3

3. Qual o BAD SMELL detetado no código apresentado ?

- A. Refused Bequest
- B. Middle man
- C. Inappropriate intimacy
- D. Nenhuma das anteriores

4. Assinale a armação incorreta?
   - A. Antes de iniciar o refactoring a um módulo de código, deve existir uma suite de testes.
   - B. O refactoring influencia (e modifica) o comportamento externo de uma classe.
   - C. O refactoring influencia (e modifica) a legibilidade do código.
   - D. O refactoring influencia (e modifica) a arquitectura interna de uma classe.

5. Qual das seguintes actividades deve ser considerada como refactoring?

   - A. Adição de novas funcionalidades.
   - B. Melhorar a performance da aplicação.
   - C. Implementação de uma suite de testes.
   - D. Nenhuma das anteriores.

```
/* Snippet 4*/
    public void imprimeCondutoresInicial(char inicial) {
        for (Condutor condutor : listCondutores) {
            if (condutor.getNome().charAt(0) == inicial) {
                System.out.println(" Condutor " + condutor.getNome());
                System.out.println(" Camiao " +
condutor.getCamiao().getMatricula().getId());
            }
        }
    }
```

6. Qual o BAD SMELL detetado no código apresentado ?
   - A. Refused Bequest
   - B. Message chain
   - C. Long Method
   - D. Nenhuma das anteriores

7. Qual a técnica mais adequada para lidar com o Bad SMELL apresentado ?
   - A. Extract Class
   - B. Replace Bidirecional Association with Unidirecional Association
   - C. Hide Delegation
   - D. Nenhuma das anteriores

```
/* Snippet 5*/
    public class Circle {
    private Point center;
    private int radius;

    public Circle() {
        this.center = new Point(0,0);
        this.radius = 1;
    }
   //getters e setters
    @Override
    public String toString() {
        return "Circle{" +
                "center=" + center +
                ", radius=" + radius +
                '}';
    }
```

```
}

public class Main {

    public static void main(String[] args) {
      Circle c= new Circle();
      for(int i=0; i< 5;i++) {
          c.setCenter(new Point(c.getCenter().getX()+1, c.getCenter().getY()+1));
          System.out.println(c);
      }
   }
}
```

```
/* Snippet 5.1*/
public static final int REPEAT=5;
    public static void main1(String[] args) {
        Circle c= new Circle();
        for(int i=0; i< REPEAT;i++) {
            c.setCenter(new Point(c.getCenter().getX()+1, c.getCenter().getY()+1));
            System.out.println(c);
        }
    }
}
```

```
/* Snippet 5.2*/
public class Circle {
    private Point center;
    private int radius;

    public Circle() {
        this.center = new Point(0,0);
        this.radius = 1;
    }

    public void moveCenter(int i, int i1) {
        Point p= getCenter();
        p.setX(p.getX()+i);
        p.setY(p.getY()+i);
        setCenter(p);
    }

    @Override
    public String toString() {
        return "Circle{" +
                "center=" + center +
                ", radius=" + radius +
                '}';
    }
}
public class Main {
   public static void main(String[] args) {
        Circle c= new Circle();
        for(int i=0; i< 5;i++) {
            c.moveCenter(1,1);
            System.out.println(c);
        }
    }
```

```
 /* Snippet 5.3*/
public class Main {
    public static void main3(String[] args) {
        Circle c= new Circle();
        for(int i=0; i< 5;i++) {
            c.setCenter(new Point(c.getCenter().getX()+i, c.getCenter().getY()+i));
            System.out.println(c);
        }
        }
    }
}
```

P8 - Qual dos fragmentos de código não é o resultado da aplicação de refatoring?

- A. Snippet 1
- B. Snippet 2
- C. Snippet 3
- D. Todos correspondem à aplicação de uma ou mais técnicas de refatoring