```
public interface DocumentAccess {
 public void saveDocument(Document b);
 public Document loadDocument(String tittle);
}
```

```
public class X {
   public static DocumentAccess createPersistenceAccess(String type) {
      switch (type) {
         case "serialization":
            return new DocumentFilePersistence();
         case "mySql":
            return new DocumentMySQLPersistence();
         default:
            throw new IllegalArgumentException(" this type  does not exist");
      }
   }
}
```

```
public class Main {

   public static void main(String[] args) {
      DocumentAccess documentsData = X.createPersistenceAccess("mySql");
      Document b2 = new Document("Rua Sesamo3", "Ana");
      Document b3 = new Document("Rua Sesamo1", "Tomas");
      Document b4 = new Document("Rua Sesamo6", "Luis");
      Document b5 = new Document("Rua Sesamo5", "Ana");
      documentsData.saveDocument(b2);
      documentsData.saveDocument(b3);
      documentsData.saveDocument(b4);
      documentsData.saveDocument(b5);
      Document c = documentsData.loadDocument("Rua Sesamo1");
      System.out.println(c);
       }
 }
```

Figura 1

```
public interface Graph<V, E> {
   public int numVertices();
   public int numEdges();
   public Collection<Vertex<V>> vertices();
   public Collection<Edge<E, V>> edges();
   public Collection<Edge<E, V>> incidentEdges(Vertex<V> v)
         throws InvalidVertexException;
  public Vertex<V> opposite(Vertex<V> v, Edge<E, V> e)
   public boolean areAdjacent(Vertex<V> u, Vertex<V> v)
         throws InvalidVertexException;
   public Vertex<V> insertVertex(V vElement)
         throws InvalidVertexException;
   public Edge<E, V> insertEdge(Vertex<V> u, Vertex<V> v, E edgeElement)
         throws InvalidVertexException, InvalidEdgeException;
   public Edge<E, V> insertEdge(V vElement1, V vElement2, E edgeElement)
         throws InvalidVertexException, InvalidEdgeException;;
   public V removeVertex(Vertex<V> v) throws InvalidVertexException;
   public E removeEdge(Edge<E, V> e) throws InvalidEdgeException;
   public V replace(Vertex<V> v, V newElement) throws InvalidVertexException;
   public E replace(Edge<E, V> e, E newElement) throws InvalidEdgeException;
}
```

Figura 2

```
public class FactoryMethodMain {

    public static void main(String[] args) {
        Loja loja = new LojaAmerica();
        Pizza p1 = loja.make("type1", "pp1");
        Pizza p2 = loja.make("type2", "pp2");
        p1.applyPromotion(5);
        p2.applyPromotion(5);
        System.out.println(p1);
        System.out.println(p2);
    }

}
```

Figura 3

```
ALGORITHM FX
Input: bt BinaryTree
Output:NONE
BEGIN
IF NOT(isEmpty(bt)) THEN
     FX(getLeftTree(bt))
     WRITE(getRoot(bt))
     FX(getRightTree(bt))
ENDIF
END
```

Figura 4

```
private class TreeNode {

    private E element;
    private TreeNode left;
    private TreeNode right;

    public TreeNode(E element, TreeNode left, TreeNode right) {
      this.element = element;
      this.left = left;
      this.right = right;
    }
    public TreeNode(E element) {
                          A
  }
    // Retorna true se o nó for interno
    public boolean isInternal(){
                          B
    }
  }
}
```

Figura 5

```
    public Person metodoX() {
        if (network.numVertices() == 0) {
            return null;
        }
        Vertex<Person> x = null;
        int m = -1;

            if (n > m) {
                x = v;
                m = n;
            }
        }
        return x.element();
      }
}
```

Figura 6

```
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);
}
```

Figura 7

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

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

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


}
```

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

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

    public  void  addItem(Product  prod,  int
quant)    {
        items.add(new Item(prod,quant));
    }

    public int getTotal(){
        total=0;
        for(Item item: items)

               total+=item.getQuantity()*
               item.getProd().getPrice();
        return total;
    }

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

            }

}
```

Figura 8

```
public interface IntegerElement {
    int value();
}
public class BSTImplementation<T extends IntegerElement> {
    private TreeNode root;
    //...
        private class TreeNode {
        T element;
        TreeNode left, right;
    }    //...
}
```

Figura 9

```
public class BagOfIntegers {
  private ArrayList<Integer> bag;
  private String name;

  public BagOfIntegers(String name) {
    this.name = name;
    this.bag = new ArrayList<>();
  }

  public void setName(String name) { this.name = name; }

  public void put(int number) { bag.add(number); }

  public String toString() {
    return String.format("%s | %s \n", name, bag.toString());
  }

  public Memento createMemento() {
    /* ... */
  }

  public void setMemento(Memento state) {
    /* ... */
  }

  private class MyMemento implements Memento {
    /* ... */
  }
}
```

```
public interface Memento { /* empty*/ }

public class CareTaker {
  private final BagOfIntegers bag;
  /* ... */

  public CareTaker(BagOfIntegers bag) { this.bag = bag; }

  public void saveState() {  /* ... */  }
  public void restoreState() { /* ... */ }
}

public class Main {
  public static void main(String[] args) {
    BagOfIntegers b = new BagOfIntegers("Empty");
    CareTaker caretaker = new CareTaker(b);
    careTaker.saveState();
    b.put(1);
    b.put(3);
    b.setName("Not empty.");
    careTaker.saveState();
    b.put(7);
    careTaker.restoreState();
    System.out.println(b);
  }
}
```

Figura 10

```
public class Dice {
  private int currentValue = 1;

  public void roll() {
    currentValue = (int)Math.random() * 6;
  }
}

public class DiceView extends VBox {
  private final Label diceValue;

  public DiceView() {

    initComponents();
  }

  private void initComponents() {
    Label text = new Label("Dice Value:");
    diceValue = new Label("?");
    this.getChildren().addAll(text, diceValue);
  }

  public void setTriggers(MiddleLayer c) {
    /* ... */
  }
}
```

```
public class MiddleLayer {
  private final Dice model;
  private final DiceView view;

  public MiddleLayer(Dice model, DiceView view) {
    this.model = model; this.view = view;

  }
}

public class Main extends Application {
    public void start(Stage primaryStage) {

        Dice dice = new Dice();
        DiceView view = new DiceView(dice);
        MiddleLayer controller = new MiddleLayer(dice, diceView);
        BorderPane window = new BorderPane();
        window.setCenter(view);
        Scene scene = new Scene(window, 300, 250);
        primaryStage.setTitle("Roll the dice!");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
    public static void main(String[] args) {
        launch(args);
    }
}
```

Figura 11

```
public class Inventory {
  private String[] productNames;
  private double[] productPrices;
  private int size;
  private int cheapestIndex;

  public Inventory() {
    productNames = new String[1000];
    productPrices = new double[1000];
    size = 0;
  }

  public boolean addProduct(String name, double price) {
    if(exists(name)) return false;
    productNames[size] = name;
    productPrices[size] = price;
    size++;
    return true;
  }

  public boolean updatePrice(String name, double price) {
    if(exists(name)) return false;

    for(int i=0; i<size; i++) {
      if(name.compareToIgnoreCase(productNames[i] == 0) {
        productPrices[i] = price;
        return true;
      }
    }
    return false;
  }
```

```
public String getCheapestProduct() {
    if(size == 0) return "None";
    double min = productPrices[0];
    cheapestIndex = 0;
    for(int i=0; i<size; i++) {
      if(productPrices[i] < min) {
        min = productPrices[i];
        cheapestIndex = i;
      }
    }
    return productNames[cheapestIndex];
  }

  private boolean exists(String name) {
    for(int i=0; i<size; i++) {
      if(name.compareToIgnoreCase(productNames[i] == 0)) return true
;
    }
    return false;
  }

}
```

Figura 12