CENTRO UNIVERSITARIO UAEM ATLACOMULCO

MATERIA

PARADIGMAS DE LA PROGRAMACIÓN

DOCENTE

JULIO ALBERTO DE LA TEJA LÓPEZ

ALUMNO

ALEXIS VALENCIA MARTÍNEZ

CARRERA

LICENCIATURA EN INGENIERÍA EN SISTEMAS COMPUTACIONALES

ICO-27

FECHA DE ENTREGA

12/SEPTIEMBRE/2023

Ejercicio 1: Clases abstractas e interfaces de las figuras Geométricas

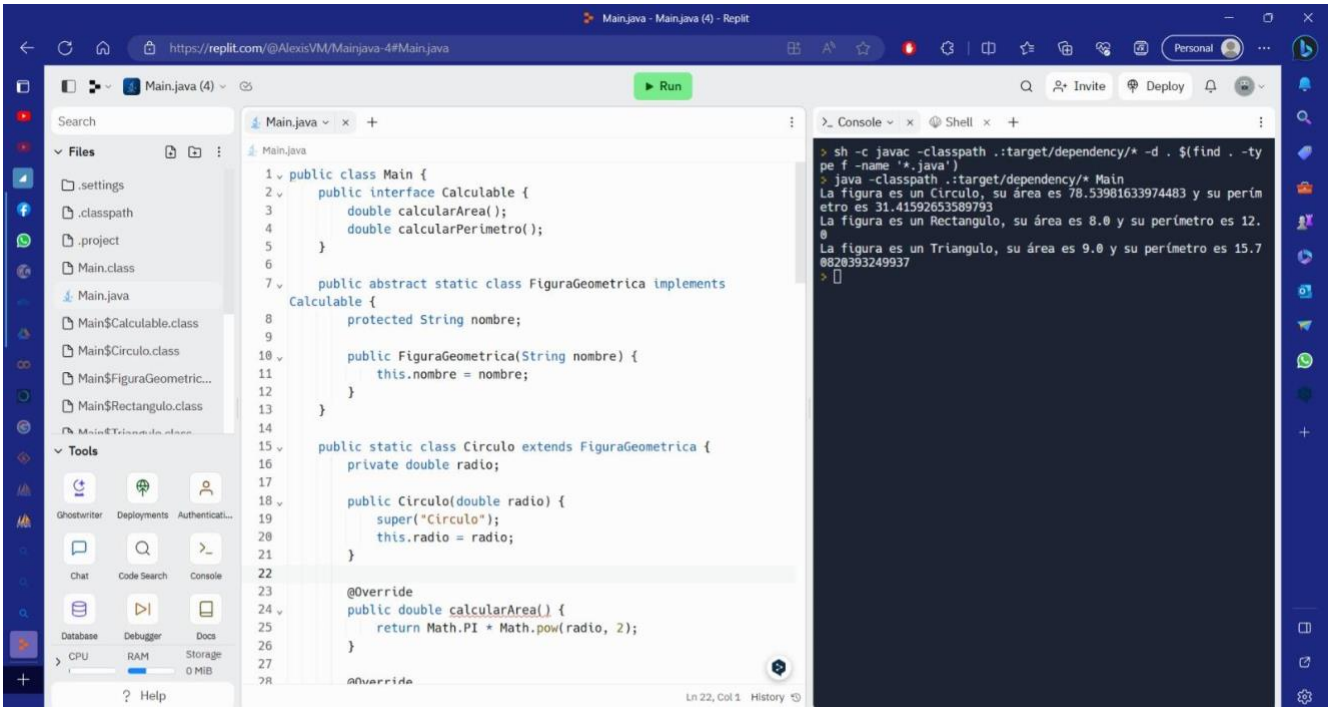
- CÓDIGO

```
Public class Main {
  Public interface Calculable {
    Double calcularArea();
    Double calcularPerimetro();
  }

  Public abstract static class FiguraGeometrica implements Calculable {
    Protected String nombre;

    Public FiguraGeometrica(String nombre) {
      This.nombre = nombre;
    }
  }

  Public static class Circulo extends FiguraGeometrica {
    Private double radio;

    Public Circulo(double radio) {
      Super(“Circulo”);
      This.radio = radio;
    }

    @Override
    Public double calcularArea() {
      Return Math.PI * Math.pow(radio, 2);
    }

    @Override
    Public double calcularPerimetro() {
      Return 2 * Math.PI * radio;
    }
  }

  Public static class Rectangulo extends FiguraGeometrica {
    Private double base;
    Private double altura;

    Public Rectangulo(double base, double altura) {
      Super(“Rectangulo”);
      This.base = base;
      This.altura = altura;
    }

    @Override
    Public double calcularArea() {
      Return base * altura;
```

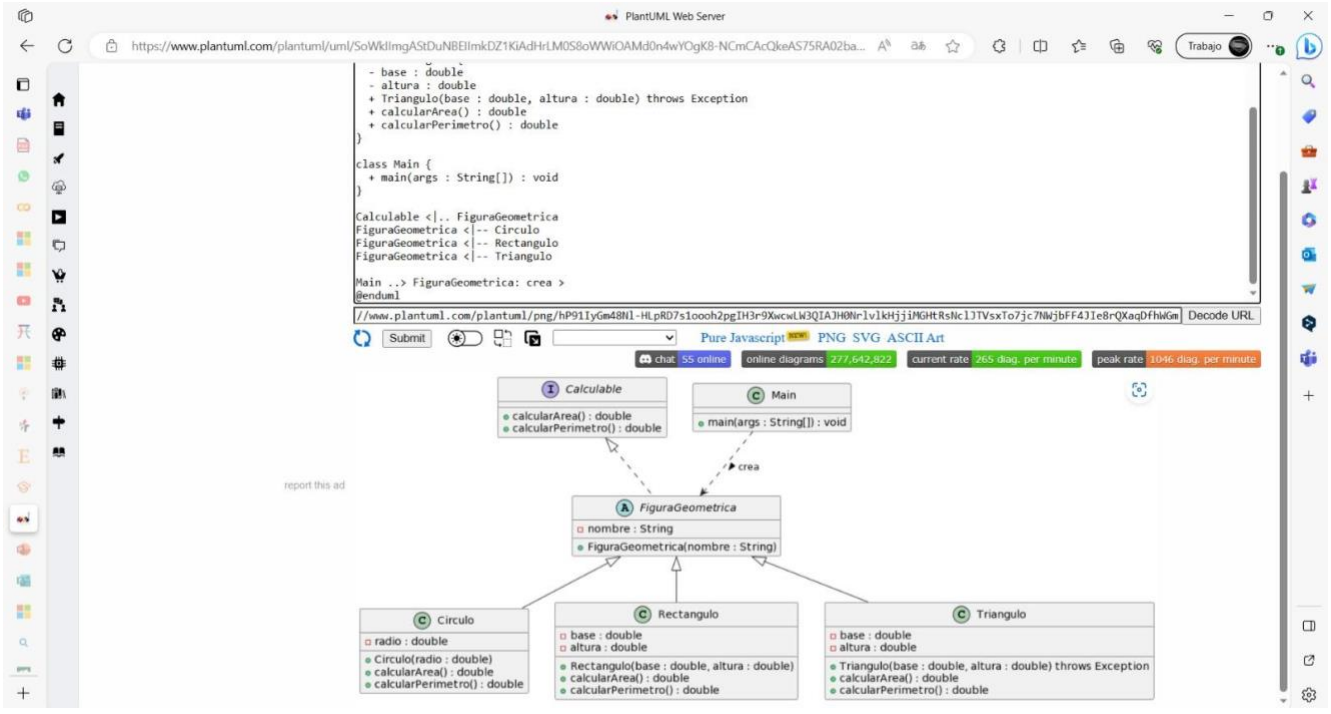
```
    }

    @Override
    Public double calcularPerimetro() {
      Return 2 * (base + altura);
    }
  }

  Public static class Triangulo extends FiguraGeometrica {
    Private double base;
    Private double altura;

    Public Triangulo(double base, double altura) throws Exception {
      Super("Triangulo");
      If (base <= 0 || altura <= 0) {
        Throw new Exception("No se puede crear un triángulo con base o altura menor o igual a cero");
      }
      This.base = base;
      This.altura = altura;
    }

    @Override
    Public double calcularArea() {
      Return (base * altura) / 2;
    }

    @Override
    Public double calcularPerimetro() {
      If (base <= 0 || altura <= 0) {
        System.out.println("Error: No se puede calcular el perímetro con base o altura menor o igual a cero");
        Return -1;
      }
      Return base + altura + Math.sqrt(Math.pow(base, 2) + Math.pow(altura, 2));
    }
  }

  Public static void main(String[] args) {
    Try {
      FiguraGeometrica[] figuras = new FiguraGeometrica[3];
      Figuras[0] = new Circulo(5);
      Figuras[1] = new Rectangulo(4, 2);
      Figuras[2] = new Triangulo(3, 6);

      For (FiguraGeometrica figura : figuras) {
        System.out.println("La figura es un " + figura.nombre + ", su área es " + figura.calcularArea() + " y su
perímetro es " + figura.calcularPerimetro());
      }
    } catch (Exception e) {
```

```
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

- SALIDA



- DIAGRAMA UML



Ejercicio 2: Clases abstractas e interfaces en personajes en un Videojuego

- CÓDIGO

```
// Definimos la clase abstracta “Personaje”
Abstract class Personaje {
   String nombre;
   Int nivel;

   Public Personaje(String nombre, int nivel) {
      If (nivel < 1) {
         Throw new IllegalArgumentException(“El nivel no puede ser menor que 1”);
      }
      This.nombre = nombre;
      This.nivel = nivel;
   }

   Public void atacar() {
      If (nivel < 5) {
         Throw new RuntimeException(“El nivel debe ser al menos 5 para realizar un ataque”);
      }
   }
}

// Definimos la interfaz “HabilidadesMágicas”
Interface HabilidadesMágicas {
   Void usarHabilidadEspecial();
}

// Definimos la interfaz “HabilidadesFísicas”
Interface HabilidadesFísicas {
   Void gritar();
```

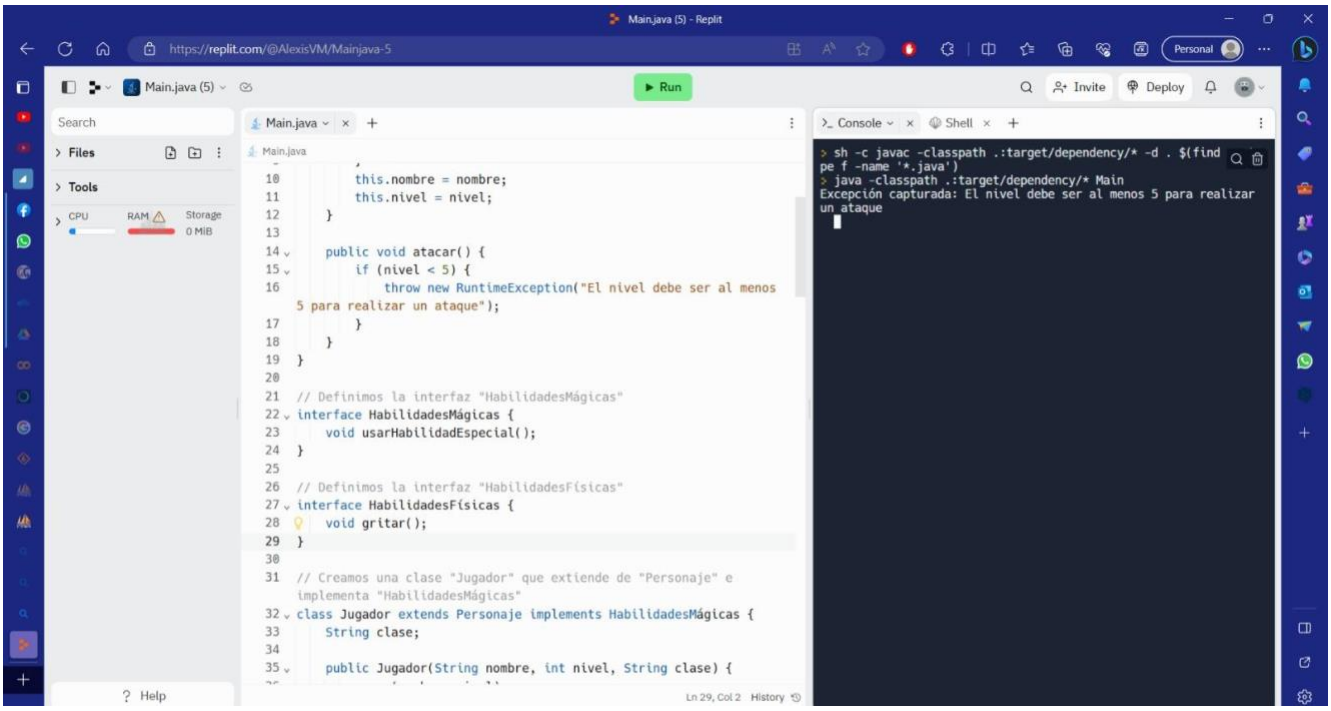
```
}

// Creamos una clase “Jugador” que extiende de “Personaje” e implementa “HabilidadesMágicas”
Class Jugador extends Personaje implements HabilidadesMágicas {
   String clase;

   Public Jugador(String nombre, int nivel, String clase) {
      Super(nombre, nivel);
      This.clase = clase;
   }

   @Override
   Public void atacar() {
      Super.atacar();
      System.out.println(nombre + “ ataca con su espada!”);
   }

   @Override
   Public void usarHabilidadEspecial() {
      System.out.println(nombre + “ usa su habilidad especial!”);
   }
}

// Creamos una clase “Enemigo” que extiende de “Personaje” e implementa “HabilidadesFísicas”
Class Enemigo extends Personaje implements HabilidadesFísicas {
   String tipo;

   Public Enemigo(String nombre, int nivel, String tipo) {
      Super(nombre, nivel);
      This.tipo = tipo;
```

```
    }

    @Override
    Public void atacar() {
        Super.atacar();
        System.out.println(nombre + " ataca con su garra!");
    }

    @Override
    Public void gritar() {
        System.out.println(nombre + " grita!");
    }
}

// Creamos una clase "Paladín" que extiende de "Personaje" e implementa ambas interfaces
Class Paladín extends Personaje implements HabilidadesMágicas, HabilidadesFísicas {
    String clase;

    Public Paladín(String nombre, int nivel, String clase) {
        Super(nombre, nivel);
        This.clase = clase;
    }

    @Override
    Public void atacar() {
        Super.atacar();
        System.out.println(nombre + " ataca con su espada y magia!");
    }

    @Override
```

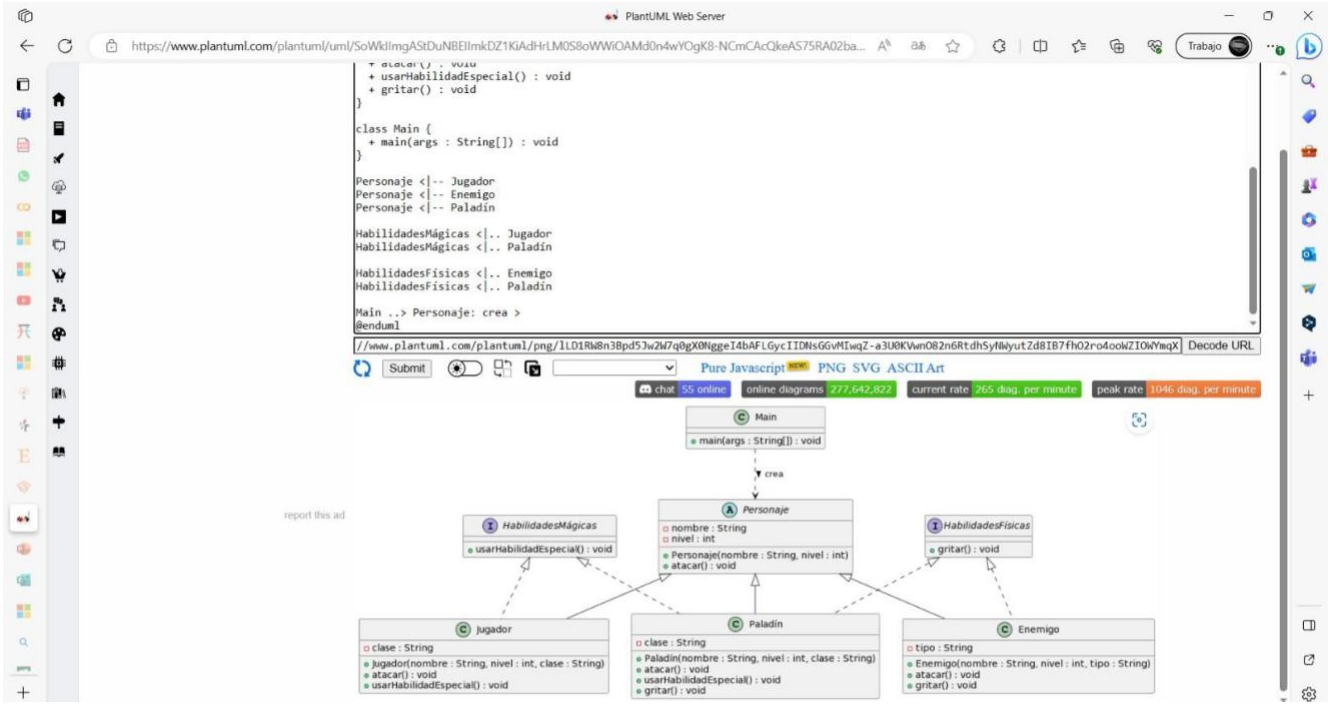
```
    Public void usarHabilidadEspecial() {
        System.out.println(nombre + " usa su habilidad especial mágica!");
    }

    @Override
    Public void gritar() {
        System.out.println(nombre + " grita con fuerza!");
    }
}

Public class Main {

  Public static void main(String[] args) {

      Try {
          Personaje personaje1 = new Jugador("Juan", 1, "Guerrero");
          Personaje1.atacar();
          ((Jugador) personaje1).usarHabilidadEspecial();

          Personaje personaje2 = new Enemigo("Esqueleto", 1, "Esqueleto");
          Personaje2.atacar();
          ((Enemigo)personaje2).gritar();

          Personaje personaje3 = new Paladín("Pedro", 1, "Paladín");
          Personaje3.atacar();
          ((Paladín) personaje3).usarHabilidadEspecial();
          ((Paladín) personaje3).gritar();
      } catch (RuntimeException e) { // Modificado aquí
          System.out.println("Excepción capturada: El nivel debe ser al menos 5 para realizar un ataque");
      }
```

```
    }
}
```

- SALIDA



- DIAGRAMA UML



Ejercicio 3: Clases abstractas e interfaces en una Paletería

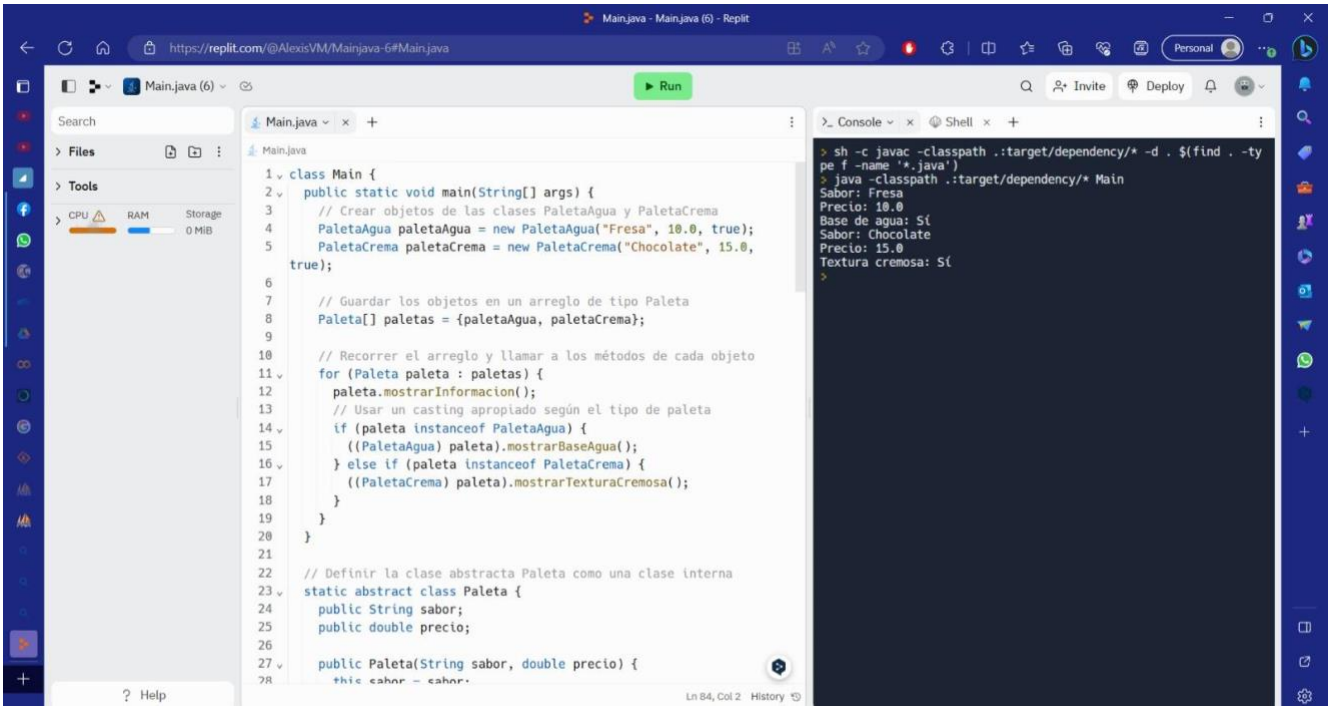
- CÓDIGO

```
Class Main {
  Public static void main(String[] args) {
    // Crear objetos de las clases PaletaAgua y PaletaCrema
    PaletaAgua paletaAgua = new PaletaAgua(“Fresa”, 10.0, true);
    PaletaCrema paletaCrema = new PaletaCrema(“Chocolate”, 15.0, true);

    // Guardar los objetos en un arreglo de tipo Paleta
    Paleta[] paletas = {paletaAgua, paletaCrema};

    // Recorrer el arreglo y llamar a los métodos de cada objeto
    For (Paleta paleta : paletas) {
      Paleta.mostrarInformacion();
      // Usar un casting apropiado según el tipo de paleta
      If (paleta instanceof PaletaAgua) {
        ((PaletaAgua) paleta).mostrarBaseAgua();
      } else if (paleta instanceof PaletaCrema) {
        ((PaletaCrema) paleta).mostrarTexturaCremosa();
      }
    }
  }

  // Definir la clase abstracta Paleta como una clase interna
  Static abstract class Paleta {
    Public String sabor;
    Public double precio;

    Public Paleta(String sabor, double precio) {
      This.sabor = sabor;
```

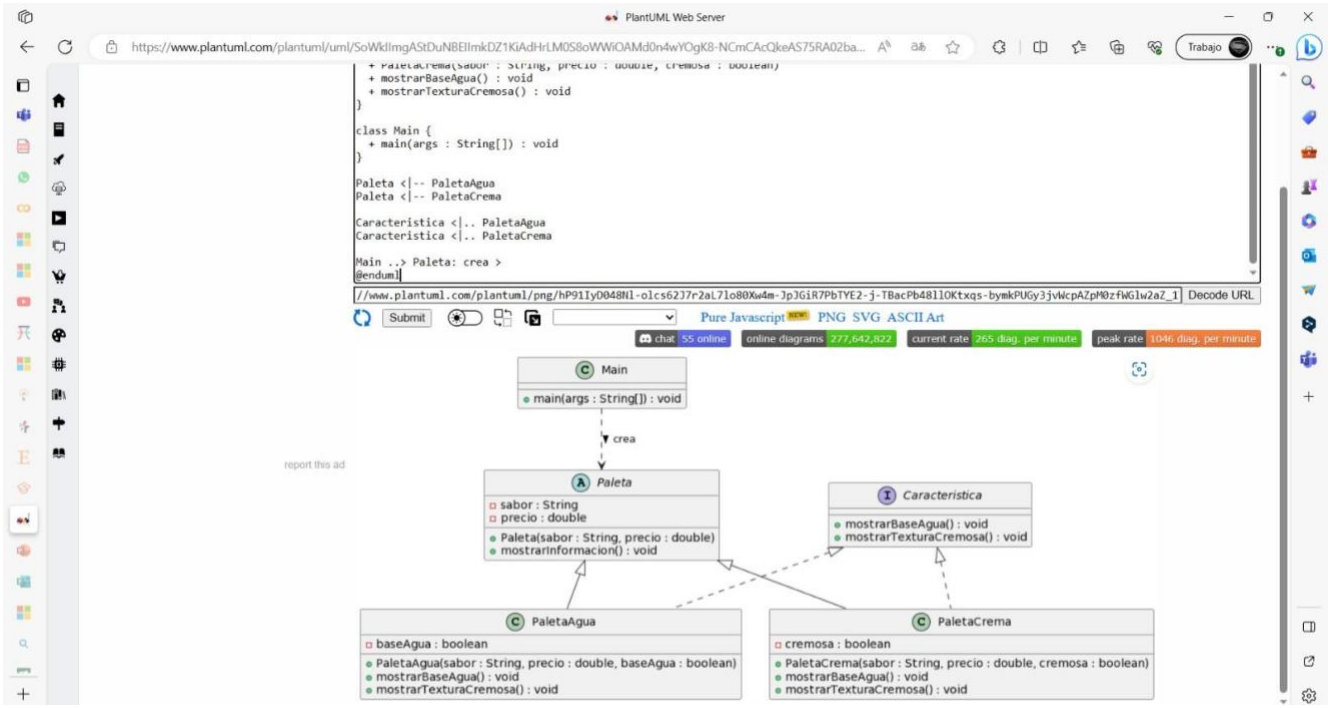
```
    This.precio = precio;
  }

  Public void mostrarInformacion() {
    System.out.println(“Sabor: “ + this.sabor);
    System.out.println(“Precio: “ + this.precio);
  }
}

// Definir la interfaz Caracteristica como una clase interna
Interface Caracteristica {
  // Declarar los métodos abstractos que deben implementar las clases que la usen
  Void mostrarBaseAgua();
  Void mostrarTexturaCremosa();
}

// Definir la clase PaletaAgua como una clase interna que hereda de la clase abstracta Paleta e implementa la interfaz Caracteristica
Static class PaletaAgua extends Paleta implements Caracteristica {
  Public boolean baseAgua;

  Public PaletaAgua(String sabor, double precio, boolean baseAgua) {
    Super(sabor, precio);
    This.baseAgua = baseAgua;
  }

  // Sobrescribir el método mostrarBaseAgua() según la característica de la paleta de agua
  Public void mostrarBaseAgua() {
    System.out.println(“Base de agua: “ + (this.baseAgua ? “Sí” : “No”));
  }
```

```
    // Sobrescribir el método mostrarTexturaCremosa() según la característica de la paleta de agua
    Public void mostrarTexturaCremosa() {
      System.out.println(“Textura cremosa: No”);
    }
  }

  // Definir la clase PaletaCrema como una clase interna que hereda de la clase abstracta Paleta e implementa la interfaz Caracteristica
  Static class PaletaCrema extends Paleta implements Caracteristica {
    Public boolean cremosa;

    Public PaletaCrema(String sabor, double precio, boolean cremosa) {
      Super(sabor, precio);
      This.cremosa = cremosa;
    }

    // Sobrescribir el método mostrarBaseAgua() según la característica de la paleta de crema
    Public void mostrarBaseAgua() {
      System.out.println(“Base de agua: No”);
    }

    // Sobrescribir el método mostrarTexturaCremosa() según la característica de la paleta de crema
    Public void mostrarTexturaCremosa() {
      System.out.println(“Textura cremosa: “ + (this.cremosa ? “Sí” : “No”));
    }
  }
}
```

- SALIDA



- DIAGRAMA UML



The reason I applied this concept is to take advantage of the advantages of inheritance and polymorphism in Java. By using an abstract class, I can define common attributes and methods of palettes and avoid code repetition. By using an interface, I can declare the methods that the classes that implement it must have and ensure the consistency of their behavior