



# ESPE

UNIVERSIDAD DE LAS FUERZAS ARMADAS  
INNOVACIÓN PARA LA EXCELENCIA



Departamento de Computación  
Programación Integrativa de Componentes

Nombre: Carlos Pogo  
NRC: 16496

Tarea 1.6: Pattern Desing with javascript

Bridge Pattern

Al hacer clic en los botones, se dibujarán círculos en el canvas en lugar de solo mostrar valores de texto. Cada implementación de `DrawAPI` (`RedCircle` y `GreenCircle`) dibuja un círculo en el canvas con el color correspondiente. Además, se ha añadido una función para limpiar el canvas antes de dibujar un nuevo círculo.

**Código y explicación:**

index.html

```

1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <meta name="viewport" content="width=device-width, initial-scale=1.0">
6      <title>Bridge Pattern Demo</title>
7      <link rel="stylesheet" href="css/style.css">
8  </head>
9  <body>
10     <div class="container">
11         <h1>Bridge Pattern Demo</h1>
12         <div>
13             <button id="drawRedCircle">Draw Red Circle</button>
14             <button id="drawGreenCircle">Draw Green Circle</button>
15         </div>
16         <canvas id="canvas" width="500" height="500"></canvas>
17     </div>
18     <script src="js/app.js"></script>
19     <script src="js/script.js"></script>
20 </body>
21 </html>
22

```

HTML Básico: Se configura una estructura básica de HTML con un título, dos botones para dibujar círculos y un elemento `<canvas>` para mostrar los gráficos.

CSS y JavaScript: Se enlazan los archivos de estilos (`style.css`) y scripts (`script.js` y `app.js`) necesarios para la funcionalidad y los estilos.

Interfaz DrawAPI:

Define una interfaz con el método `drawCircle`. Cualquier implementación de esta interfaz debe proporcionar una implementación de este método.

```

1 // Interfaz DrawAPI
2 class DrawAPI {
3     drawCircle(ctx, radius, x, y) {
4         throw "¡Este método debe anularse!";
5     }
6 }

```

### Implementaciones Concretas de DrawAPI:

RedCircle y GreenCircle implementan DrawAPI y proporcionan una implementación específica del método drawCircle para dibujar un círculo rojo y un círculo verde, respectivamente.

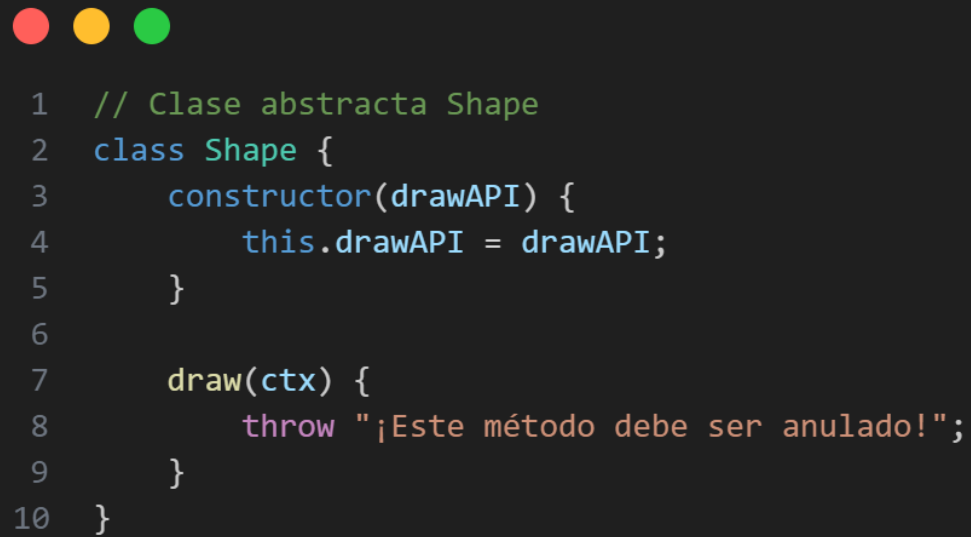
```

1 // Implementación concreta de DrawAPI - RedCircle
2 class RedCircle extends DrawAPI {
3     drawCircle(ctx, radius, x, y) {
4         ctx.beginPath();
5         ctx.arc(x, y, radius, 0, 2 * Math.PI, false);
6         ctx.fillStyle = 'red';
7         ctx.fill();
8         ctx.lineWidth = 2;
9         ctx.strokeStyle = '#003300';
10        ctx.stroke();
11    }
12 }
13
14 // Implementación concreta de DrawAPI - GreenCircle
15 class GreenCircle extends DrawAPI {
16     drawCircle(ctx, radius, x, y) {
17         ctx.beginPath();
18         ctx.arc(x, y, radius, 0, 2 * Math.PI, false);
19         ctx.fillStyle = 'green';
20         ctx.fill();
21         ctx.lineWidth = 2;
22         ctx.strokeStyle = '#003300';
23         ctx.stroke();
24     }
25 }

```

### Clase Abstracta Shape:

Define una clase abstracta Shape que contiene una referencia a DrawAPI. Esto permite a Shape delegar el trabajo de dibujar a la implementación de DrawAPI.



```
1  // Clase abstracta Shape
2  class Shape {
3      constructor(drawAPI) {
4          this.drawAPI = drawAPI;
5      }
6
7      draw(ctx) {
8          throw "¡Este método debe ser anulado!";
9      }
10 }
```

### Implementación Concreta de Shape - Circle::

Circle extiende Shape y usa una implementación de DrawAPI para dibujar un círculo.

```

1 // Implementación concreta de Shape - Circle
2 class Circle extends Shape {
3     constructor(x, y, radius, drawAPI) {
4         super(drawAPI);
5         this.x = x;
6         this.y = y;
7         this.radius = radius;
8     }
9
10    draw(ctx) {
11        this.drawAPI.drawCircle(ctx, this.radius, this.x, this.y);
12    }
13 }
14

```

## Lógica de Interacción y Manejo del Canvas

app.js

```

const canvas = document.getElementById('canvas');
const ctx = canvas.getContext('2d');

function clearCanvas() {
    ctx.clearRect(0, 0, canvas.width, canvas.height);
}

document.getElementById('drawRedCircle').addEventListener('click', () => {
    clearCanvas();
    const redCircle = new Circle(250, 250, 50, new RedCircle());
    redCircle.draw(ctx);
});

document.getElementById('drawGreenCircle').addEventListener('click', () => {
    clearCanvas();
    const greenCircle = new Circle(250, 250, 50, new GreenCircle());
    greenCircle.draw(ctx);
});

```

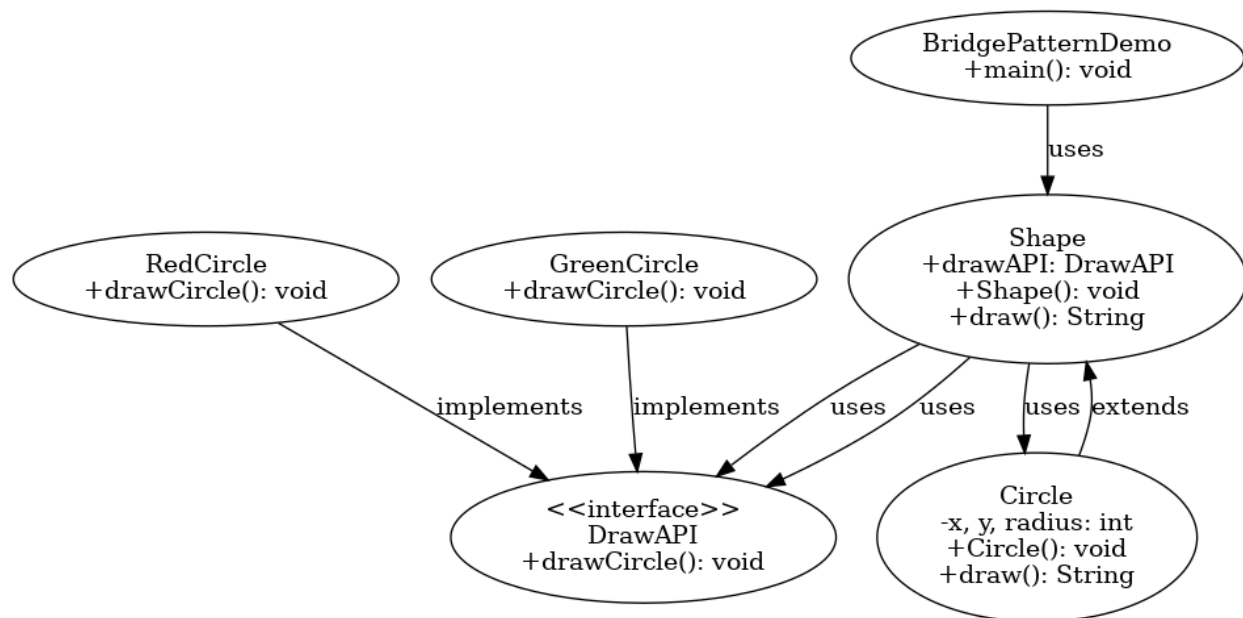
Se obtiene el elemento `<canvas>` y su contexto 2D para poder dibujar en él.

`clearCanvas` borra el contenido del canvas antes de dibujar un nuevo círculo.

Se añaden event listeners a los botones para dibujar círculos rojos y verdes.

Al hacer clic en un botón, se crea una instancia de `Circle` con la implementación adecuada de `DrawAPI` y se llama a su método `draw`.

Diagrama:



State Pattern

Codigo:

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>State Pattern Demo</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>
```

```
<div class="container">
  <h1>State Pattern Demo</h1>
  <div>
    <button id="startState">Set Start State</button>
    <button id="stopState">Set Stop State</button>
  </div>
  <div id="output"></div>
</div>
<script src="script.js"></script>
<script src="app.js"></script>
</body>
</html>
```

style.css

```
body {
  font-family: Arial, sans-serif;
}

.container {
  text-align: center;
  margin-top: 50px;
}

button {
  margin: 10px;
  padding: 10px 20px;
  font-size: 16px;
}

#output {
  margin-top: 20px;
  font-size: 18px;
}
```

script.js

```
// Interfaz State
class State {
  doAction(context) {
    throw "This method must be overridden!";
  }
}

// Implementación concreta de State - StartState
```

```

class StartState extends State {
  doAction(context) {
    console.log("Player is in start state");
    context.setState(this);
  }

  toString() {
    return "Start State";
  }
}

// Implementación concreta de State - StopState
class StopState extends State {
  doAction(context) {
    console.log("Player is in stop state");
    context.setState(this);
  }

  toString() {
    return "Stop State";
  }
}

// Clase Context
class Context {
  constructor() {
    this.state = null;
  }

  setState(state) {
    this.state = state;
  }

  getState() {
    return this.state;
  }
}

```

app.js

```

const context = new Context();

document.getElementById('startState').addEventListener('click', () => {
  const startState = new StartState();
  startState.doAction(context);
  document.getElementById('output').innerText = `Current State: ${context.getState()}`;
});

```



```
document.getElementById('stopState').addEventListener('click', () => {  
  const stopState = new StopState();  
  stopState.doAction(context);  
  document.getElementById('output').innerText = `Current State: ${context.getState()}`;  
});
```

**Interfaz State:** Define el método `doAction` que debe ser implementado por todos los estados concretos.

- **Estados Concretos:**
  - **StartState:** Implementa el método `doAction` para el estado de inicio.
  - **StopState:** Implementa el método `doAction` para el estado de parada.
- **Contexto:** Mantiene una referencia al estado actual y permite cambiar de estado.

El State Pattern permite que un objeto cambie su comportamiento cuando su estado interno cambia. En este ejemplo:

- **Interfaz State:** Define un método `doAction` que debe ser implementado por todos los estados concretos.
- **Estados Concretos (StartState y StopState):** Implementan `doAction` para cambiar el estado del contexto.
- **Contexto (Context):** Mantiene el estado actual y permite cambiar entre estados.

Diagrama:

