UNIVERSIDAD DE LAS FUERZAS ARMADAS - ESPE

Nombres:

- Macas Karol
- Pilataxi Diego
- CampoVerde Carlos
- Lara Nicole

Grupo: 3

INFORME DE PATRONES DE DISEÑO ORIENTADO A OBJETOS

Introducción

En este informe se analiza la implementación de tres patrones de diseño (creación, estructural y comportamiento) utilizados en el proyecto de una aplicación de boxeadores. A continuación, se detalla el análisis de cada patrón seleccionado, su UML y el código de las clases correspondientes.

1. PATRONES DE CREACIÓN

ANÁLISIS

Permite la creación de un objeto complejo, a partir de una variedad de partes que contribuyen individualmente a la creación y ensamblaje del objeto mencionado.

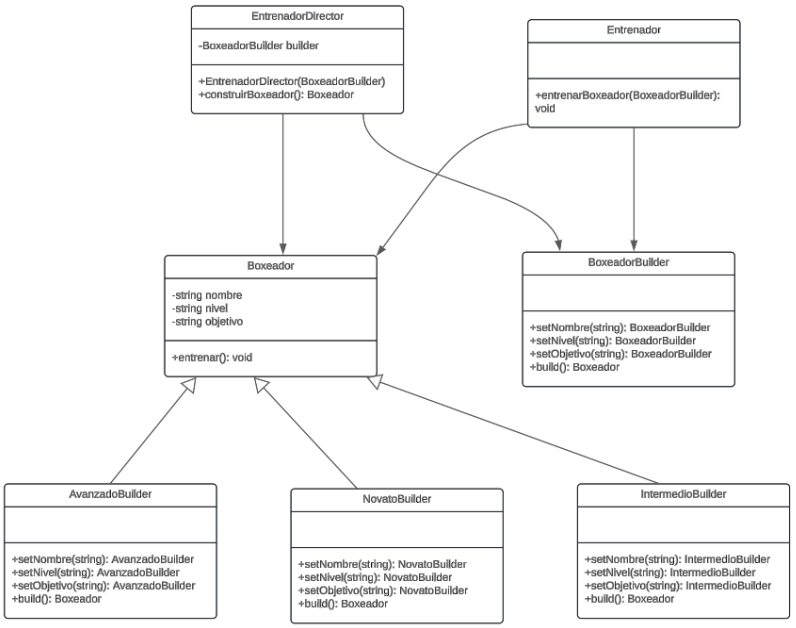
El patrón Builder nos permite crear un objeto complejo, en este caso, un boxeador, a partir de una variedad de partes que se ensamblan individualmente. Este patrón centraliza el proceso de creación, permitiendo crear diferentes representaciones del objeto construido de manera dinámica.

El uso del patrón Builder en nuestro sistema ofrece varios beneficios clave:

- **Flexibilidad**:Podemos crear diferentes representaciones del boxeador según sus necesidades específicas.
- **Reutilización:**La lógica de construcción está separada y es modular, permitiendo reutilizar y ajustar partes individuales sin afectar el sistema completo.
- **Mantenimiento:** Facilita el mantenimiento y la escalabilidad del sistema, ya que las implementaciones de BoxeadorBuilder pueden cambiar independientemente.

- **Centralización:** El EntrenadorDirector centraliza y coordina el proceso de construcción, asegurando consistencia y eficiencia en la creación de boxeadores.

UML



- **Boxeador:** Es un objeto complejo que se está construyendo.
- **BoxeadorBuilder**: Es la interfaz que define el algoritmo de construcción, permitiendo diferentes representaciones del objeto construido.
- **NovatoBuilder, IntermedioBuilder, AvanzadoBuilder:** Son las implementaciones concretas del Builder que configuran y ensamblan las partes del Boxeador de diferentes maneras.
- **EntrenadorDirector:** Centraliza y coordina el proceso de construcción utilizando un BoxeadorBuilder.

IMPLEMENTACIÓN DEL PATRÓN BUILDER EN JAVASCRIPT (Node.js)

Clase Boxer (boxer.js)

Define el objeto Boxer con sus propiedades type, name, y objective.

```
class Boxer {
 // Constructor de la clase Boxer
```

```
  constructor(type, name, objective) {
    // Propiedades de la clase Boxer
    this.type = type;
    this.name = name;
    this.objective = objective;
  }
}

module.exports = Boxer;
```

Clase BoxerBuilder (boxerBuilder.js)

```
const Boxer = require('./boxer');

class BoxerBuilder {
  // Constructor inicializa un objeto vacío de boxeador
  constructor() {
    this.type = '';
    this.name = '';
    this.objective = '';
  }

  // Método para establecer el tipo de boxeador
  setType(type) {
    this.type = type;
    return this; // Retorna this para encadenar métodos
  }

  // Método para establecer el nombre del boxeador
  setName(name) {
    this.name = name;
    return this;
  }

  // Método para establecer el objetivo del boxeador
  setObjective(objective) {
    this.objective = objective;
    return this;
  }
```

```
  // Método para construir para crear la instancia de Boxer
  build() {
    return new Boxer(this.type, this.name, this.objective);
  }
}

module.exports = BoxerBuilder;
```

Proporciona métodos para establecer las propiedades de un Boxer y un método build para crear la instancia de Boxer.

Clase CoachDirector (coachDirector.js)

```
class CoachDirector {
  //El constructor recibe un builder de boxeador usando BoxerBuilder
  constructor(builder) {
    this.builder = builder;
  }

  //Método que recibe los parámetros para construir un boxeador
  constructBoxer(type, name, objective) {
    this.builder.setType(type);  //Establece el tipo
    this.builder.setName(name);  // Establece el nombre
    this.builder.setObjective(objective); //Establece el objetivo
    return this.builder.build();  //Devuelve el boxeador construido
  }
}

module.exports = CoachDirector;
```

Orquesta la construcción de un Boxer usando BoxerBuilder. Define los pasos específicos necesarios para construir el Boxer.

Clase CRUDBoxer (crudBoxer.js)

```
 const CoachDirector = require('./coachDirector');
const BoxerBuilder = require('./boxerBuilder');

class CRUDBoxer {
```

```
  // Constructor inicializa un array vacío para almacenar boxeadores
  constructor() {
    this.boxers = [];
  }

  //manejamos las owperaciones CRUD de los boxeadores

  // Método para crear un nuevo boxeador
  createBoxer(type, name, objective) {
    const builder = new BoxerBuilder();        // Crea una instancia de BoxerBuilder
    const director = new CoachDirector(builder); // Crea una instancia de
CoachDirector con el builder
    const boxer = director.constructBoxer(type, name, objective); // Usa el director para
construir el boxeador
    this.boxers.push(boxer);                // Agrega el boxeador al array de boxeadores
  }

  // Método para obtener todos los boxeadores
  getBoxers() {
    return this.boxers;
  }

  // Método para obtener todos los boxeadores (mismo método que getBoxers)
  getAllBoxers() {
    return this.getBoxers();
  }

  // Método para eliminar un boxeador por nombre
  deleteBoxer(name) {
    this.boxers = this.boxers.filter(boxer => boxer.name !== name); // Filtra el array
para excluir el boxeador con el nombre dado
  }
}

module.exports = CRUDBoxer;
```

Maneja las operaciones CRUD. Utiliza CoachDirector para crear nuevos Boxers.
Archivo Principal app.js

```
const express = require('express');
```

```
const bodyParser = require('body-parser');
const cors = require('cors');
const CRUDBoxer = require('./crudBoxer');

// Crea una aplicación Express
const app = express();
const crudBoxer = new CRUDBoxer(); // Crea una instancia de CRUDBoxer

app.use(bodyParser.json()); // Middleware para parsear JSON en el cuerpo de las solicitudes
app.use(cors());            // Middleware para permitir CORS

// Ruta para crear un nuevo boxeador
app.post('/boxers', (req, res) => {
  const { builderType, name, objective } = req.body; // Extrae datos del cuerpo de la solicitud
  crudBoxer.createBoxer(builderType, name, objective); // Llama al método para crear un boxeador
  res.status(201).send('Boxer created'); // Responde con un estado 201 (creado)
});

// Ruta para obtener todos los boxeadores
app.get('/boxers', (req, res) => {
  const boxers = crudBoxer.getAllBoxers(); // Llama al método para obtener todos los boxeadores
  res.json(boxers);                        // Responde con la lista de boxeadores en formato JSON
});

// Ruta para eliminar un boxeador por nombre
app.delete('/boxers/:name', (req, res) => {
  const { name } = req.params;             // Extrae el nombre de los parámetros de la ruta
  crudBoxer.deleteBoxer(name);             // Llama al método para eliminar el boxeador
  res.status(200).send('Boxer deleted');   // Responde con un estado 200 (OK)
});

// Inicia el servidor en el puerto 3002
app.listen(3002, () => {
  console.log('Server running on port 3002');
```

```
});
```

Configura y ejecuta un servidor Express que maneja las solicitudes HTTP para crear, obtener y eliminar boxeadores. Utiliza CRUDBoxer para manejar las operaciones CRUD en respuesta a las solicitudes HTTP.

2. **PATRONES ESTRUCTURALES: Aplicativo de sesiones de entrenamiento personalizado.**

ANÁLISIS

Por qué es el más adecuado:

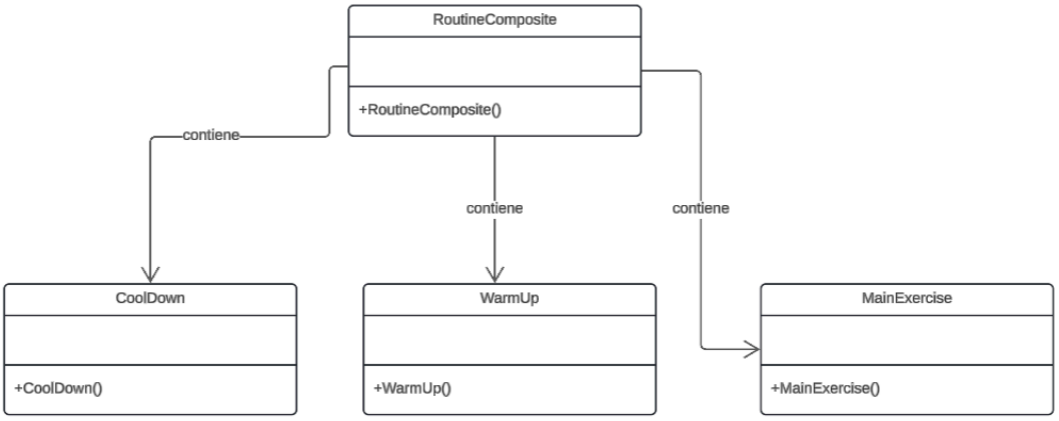
Composite es ideal cuando tienes que tratar a grupos de objetos de manera uniforme como si fueran un solo objeto. En el proyecto se maneja para organizar las rutinas de entrenamiento en componentes como calentamiento, ejercicio principal y enfriamiento, permitiendo que las rutinas complejas se componen de rutinas más simples.

Calentamiento: 15 de caminadora, descanso 5 min, estiramiento 5 min

Por qué no los demás

- **Adapter:** Hace compatible una interfaz existente con otra, pero no maneja jerarquías de objetos.
- **Bridge:** Separa una abstracción de su implementación, útil pero no esencial si la estructura de clases no es tan compleja.
- **Decorator:** Agrega funcionalidad a objetos dinámicamente, pero no es tan adecuado para manejar grupos de objetos como Composite.
- **Facade:** Proporciona una interfaz simplificada a un subsistema complejo, pero no maneja jerarquías de objetos ni permite tratar grupos de objetos de manera uniforme.

UML



CÓDIGO

CoolDown.js

```jsx
import React from 'react';

const CoolDown = () => {
  return (
    <div className="cool-down">
      <h3>Enfriamiento</h3>
      <p>Ejecutando rutina de enfriamiento...</p>
    </div>
  );
};

export default CoolDown;
```

MainExercise.js

```jsx
import React from 'react';

const MainExercise = () => {
  return (
    <div className="main-exercise">
      <h3>Ejercicio Principal</h3>
      <p> Ejecutando rutina principal...</p>
    </div>
  );
};

export default MainExercise;
```

RoutineComposite.js

```
import React from 'react';
import WarmUp from './WarmUp';
import MainExercise from './MainExercise';
import CoolDown from './CoolDown';


const RoutineComposite = () => {
  return (
    <div className="routine-composite">
      <h2>Patron Composite</h2>
      <h3>Rutina Diaria</h3>
      <WarmUp />
      <MainExercise />
      <CoolDown />
    </div>
  );
};

export default RoutineComposite;
```

WarmUp.js

```
import React from 'react';

const WarmUp = () => {
  return (
    <div className="warm-up">
      <h3>Calentamiento</h3><br />
      <p> Ejecutando rutina de calentamiento...</p><br />
    </div>
  );
};

export default WarmUp;
```

3. **PATRONES DE COMPORTAMIENTO: Aplicativo de sesiones de entrenamiento personalizado.**

ANÁLISIS

Por qué es el más adecuado:

El patrón Strategy es útil cuando tienes varias formas de realizar una tarea específica y necesitas cambiar estos algoritmos dinámicamente. En este proyecto lo aplicamos

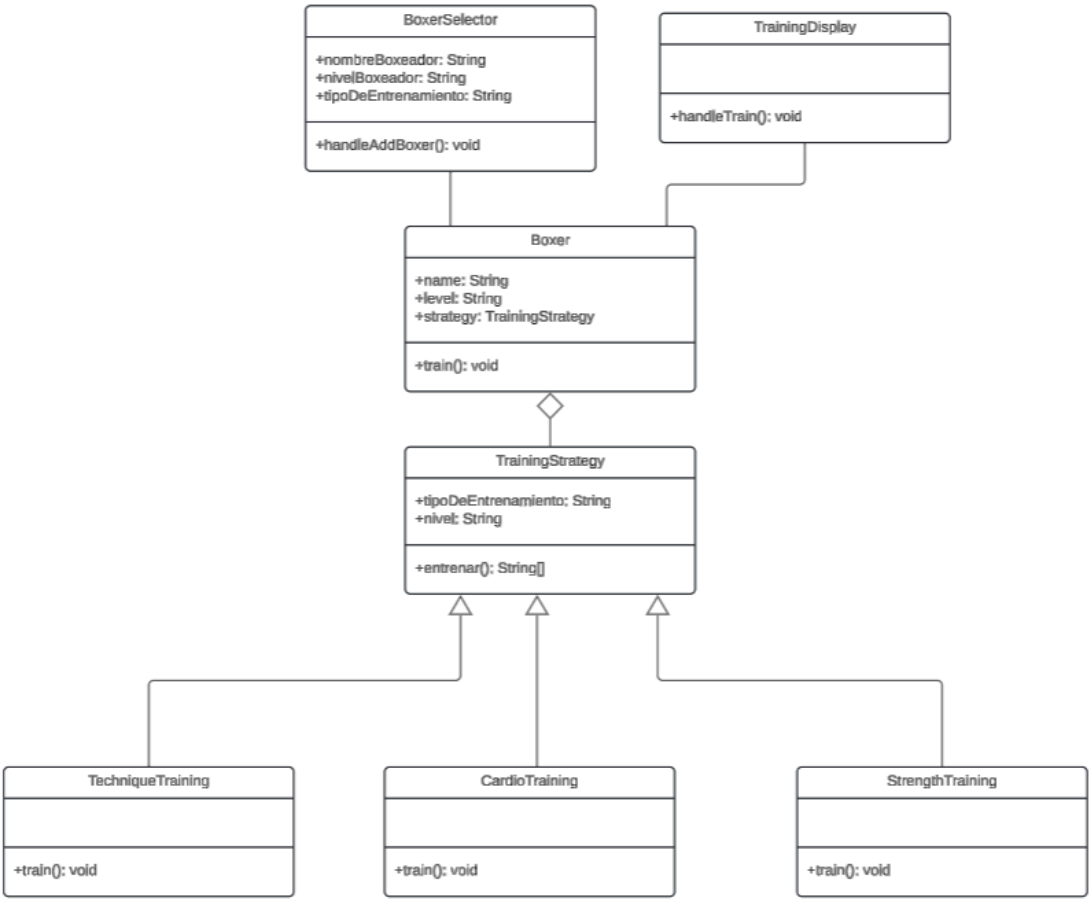
para tener diferentes estrategias de entrenamiento basadas en el nivel del Boxeador (novato, intermedio, avanzado) y sus objetivos específicos (resistencia, fuerza y técnica).

Este patrón permite encapsular las estrategias de entrenamiento en clases separadas y cambiar entre ellas según sea necesario, facilitando la extensión y el mantenimiento del código.

Por qué no los demás

- **Chain of Responsibility:** Útil para pasar una solicitud a lo largo de una cadena de manejadores, no tan relevante para tu caso donde la elección de la estrategia no sigue un flujo secuencial.
- **Command:** Útil para encapsular solicitudes como objetos, pero no se adapta bien a la necesidad de cambiar dinámicamente entre diferentes estrategias de entrenamiento.
- **Interpreter**: Útil para interpretar lenguajes, no directamente aplicable a tu caso.
- **Iterator:** Útil para recorrer colecciones de objetos, no se adapta a la necesidad de cambiar estrategias de entrenamiento.
- **Mediator:** Útil para reducir las dependencias entre objetos, pero no se adapta bien a la necesidad de seleccionar estrategias de entrenamiento.
- **Memento:** Útil para guardar y restaurar el estado de un objeto, no es relevante para cambiar estrategias.
- **Observer:** Útil para notificar cambios a múltiples objetos, pero no es necesario para la selección de estrategias de entrenamiento.
- **State:** Útil para cambiar el comportamiento de un objeto cuando su estado cambia, pero las estrategias de entrenamiento son más específicas y cambiantes que el estado.
- **Template Method:** Útil para definir el esqueleto de un algoritmo, pero en tu caso necesitas más flexibilidad para cambiar completamente la estrategia.
- **Visitor:** Útil para operar sobre una estructura de objetos, no se adapta bien a la selección dinámica de estrategias de entrenamiento.

UML



CÓDIGO

Boxer.js

```
class Boxer {
  constructor(name, level, strategy) {
    this.name = name;
    this.level = level;
    this.strategy = strategy;
  }

  train() {
    if (this.strategy && typeof this.strategy.train === 'function') {
      this.strategy.train();
    } else {
      console.error('Strategy does not have a train method');
    }
  }
}
```

```
export default Boxer;
```

BoxerSelector.jsx

```
// src/components/BoxerSelector.jsx
import React, { useState } from 'react';
import Boxer from './Boxer';
import TrainingStrategy from './TrainingStrategy';

const BoxerSelector = ({ onSelect }) => {
  const [nombreBoxeador, setNombreBoxeador] = useState('');
  const [nivelBoxeador, setNivelBoxeador] = useState('');
  const [tipoDeEntrenamiento, setTipoDeEntrenamiento] = useState('');

  const handleAddBoxer = () => {
              if    (nombreBoxeador.trim()    &&    nivelBoxeador    &&
tipoDeEntrenamiento) {
       const nuevoBoxeador = new Boxer(nombreBoxeador, nivelBoxeador,
new TrainingStrategy(tipoDeEntrenamiento, nivelBoxeador));
      onSelect(nuevoBoxeador);
    }
  };

  return (
    <div>

      <h2>Patron Strategy</h2>
      <input
        type="text"
        placeholder="Ingresa el nombre del boxeador"
        value={nombreBoxeador}
        onChange={(e) => setNombreBoxeador(e.target.value)}
      />
                  <select   value={nivelBoxeador}   onChange={(e)   =>
setNivelBoxeador(e.target.value)}>
        <option value="">Selecciona el nivel</option>
        <option value="Novato">Novato</option>
        <option value="Intermedio">Intermedio</option>
        <option value="Avanzado">Avanzado</option>
      </select>
            <select  value={tipoDeEntrenamiento}  onChange={(e)  =>
setTipoDeEntrenamiento(e.target.value)}>
        <option value="">Selecciona el tipo de entrenamiento</option>
        <option value="Cardio">Cardio</option>
        <option value="Fuerza">Fuerza</option>
      </select>
      <button onClick={handleAddBoxer}>Seleccionar Boxeador</button>
```

```
    </div>
  );
};

export default BoxerSelector;
```

CardioTraining.jsx

```
// src/components/CardioTraining.jsx
import TrainingStrategy from './TrainingStrategy';

export default class CardioTraining extends TrainingStrategy {
  train() {
    console.log("Cardio training...");
  }
}
```

StrengthTraining.jsx

```
// src/components/StrengthTraining.jsx
import TrainingStrategy from './TrainingStrategy';

export default class StrengthTraining extends TrainingStrategy {
  train() {
    console.log("Strength training...");
  }
}
```

TechniqueTraining.jsx

```
// src/components/TechniqueTraining.jsx
import TrainingStrategy from './TrainingStrategy';

export default class TechniqueTraining extends TrainingStrategy {
  train() {
    console.log("Technique training...");
  }
}
```

TrainingDisplay.jsx

```
// src/components/TrainingDisplay.jsx
import React from 'react';
```

```
const TrainingDisplay = ({ boxer }) => {
  const handleTrain = () => {
    if (boxer) {
      const recomendaciones = boxer.strategy.entrenar();
      console.log(recomendaciones);
                alert(recomendaciones.join('\n'));   //   Mostrar   las
recomendaciones en una alerta
    }
  };

  return (
    <div>
      {boxer ? (
        <div>
          <h3>Boxeador: {boxer.name}</h3>
          <p>Nivel: {boxer.level}</p>
          <button onClick={handleTrain}>Entrenar</button>
        </div>
      ) : (
        <p>No se ha seleccionado ningún boxeador.</p>
      )}
    </div>
  );
};

export default TrainingDisplay;
```

TrainingStrategy.jsx

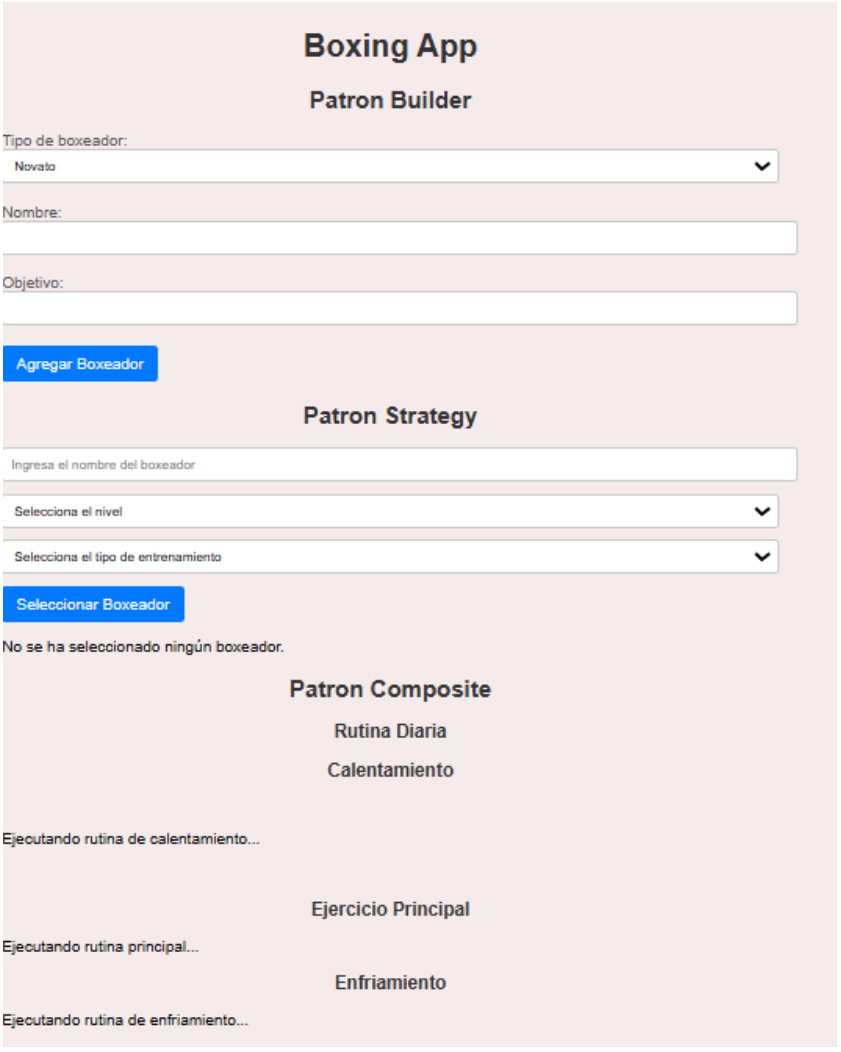
```
export default class TrainingStrategy {
  constructor(tipoDeEntrenamiento, nivel) {
    this.tipoDeEntrenamiento = tipoDeEntrenamiento;
    this.nivel = nivel; // Añadido para usar en las recomendaciones
  }

  entrenar() {
    // Definir recomendaciones basadas en el tipo de entrenamiento y
el nivel
    const recomendaciones = {
      "Novato": {
        "Cardio": [
          "Trota a ritmo ligero durante 20 minutos.",
          "Salta la cuerda durante 10 minutos."
        ],
        "Fuerza": [
          "Ejercicios básicos con peso corporal durante 15 minutos.",
          "Sentadillas y flexiones con el peso corporal."
```

```
        ]
      },
      "Intermedio": {
        "Cardio": [
          "Correr con intervalos durante 30 minutos.",
             "Entrenamiento en intervalos de alta intensidad (HIIT)
durante 20 minutos."
        ],
        "Fuerza": [
          "Entrenamiento con pesas moderadas durante 30 minutos.",
          "Ejercicios compuestos como sentadillas y press de banca."
        ]
      },
      "Avanzado": {
        "Cardio": [
              "Correr o andar en bicicleta intensamente durante 45
minutos.",
          "HIIT avanzado con sprints y intervalos intensos."
        ],
        "Fuerza": [
              "Entrenamiento con pesas pesadas con alta intensidad
durante 45 minutos.",
           "Enfoque en levantamiento de pesas y técnicas avanzadas de
fuerza."
        ]
      }
    };

     const recomendacionesPorNivel = recomendaciones[this.nivel] ||
{};
                            const       recomendacionesPorTipo       =
recomendacionesPorNivel[this.tipoDeEntrenamiento]    ||    ["No    hay
recomendaciones disponibles."];

    return recomendacionesPorTipo;
  }
}
```

- Interfaz de Estrategia (**TrainingStrategy**): Define un método train() que debe ser implementado por todas las estrategias concretas.
- Estrategias Concretas **(CardioTraining, StrengthTraining, TechnicalTraining**): Cada una implementa la interfaz TrainingStrategy y proporciona su propia versión del método train().

- Contexto (**Boxer**): Contiene una referencia a una estrategia de entrenamiento (TrainingStrategy) y permite cambiarla mediante el método setTrainingStrategy(). El método train() ejecuta la estrategia actual.

CAPTURAS DEL SISTEMA



CONCLUSIÓN:

El uso de estos patrones de diseño (Builder, Composite y Strategy) en el proyecto de la aplicación de boxeadores ha permitido una mejor organización del código, flexibilidad y facilidad de mantenimiento. Cada patrón ha resuelto problemas específicos de creación, estructuración y comportamiento, mejorando así la calidad del software.