Patterned NodeJS

A JavaScript Chess Engine

Lourdes Morente Gutiérrez

José Manuel Ramos Valderrama

Luis Boto Fernández

Tutor: Luis Fernández Muñoz

Índice

Introducción

- JavaScript como lenguaje versátil.
- Objetivo principal:

> *El objetivo principal es la elaboración de un producto de software en NodeJS, manteniendo un estilo de código unificado y aplicando los patrones de diseño clásicos de la programación orientada a objetos.*

Estilos de Programación

- I - Clases
- II - Closures
- III - Función Constructora
- IV - Función Factoría

I - Clases

- Incorporación reciente (ECMAScript 11)
- Públicas por defecto
- Azúcar sintáctico

```
class ClassA {

   #privateAttribute; //Private attributes must be defined prior to use

   constructor () {
      this.#privateAttribute = "This is a private attribute.";
   }

   publicMethod() {
       console.log("This is a public method. ");
       this.#privateMethod();
   }

   #privateMethod() {
       console.log("This is a private method.");
       console.log(this.#privateAttribute);
   }
}
```

I - Clases

Pros	Limitaciones
Similitud con lenguajes POO	Poco aceptado por la comunidad
Herencia "nativa"	Puramente sintácticas
Uso de "static"	Potencialmente verbosas

II – Closures

- Funciones como objetos de primera clase
- Uso previo a las clases
- Encapsulación

```
function ClosureA() {

   let privateAttribute = "This is a private attribute.";

   function publicMethod() {
       console.log("This is a public method. ");
       privateFunction();
   }

   function privateFunction() {
       console.log("This is a private function. ");
       console.log(privateAttribute);
   }

   return {
       publicMethod,
       publicMethod2: function() {
           console.log("This is another public method. ");
       }
   }
}
```

II – Closures

Pros	Limitaciones
Encapsulación	Los atributos públicos no son posibles
	Funciones privadas

III – Función Constructora

- Creación de objetos con la palabra reservada *new*
- *This* para referenciar el objeto creado

```
function ConstructorA(parameter) {

   this.publicAttribute = parameter;

   this.publicMethod = function() {
       console.log("This is a public method. ");
       privateFunction(this.publicAttribute);
   }

   this.publicMethod2 = function() {
       console.log("This is another public method. ");
   }

   function privateFunction(param) {
       console.log("This is a private function.");
       console.log(param);
   }
}
```

III – Función Constructora

Pros	Limitaciones
Sencillas	No soporta encapsulación
Acceso a contexto 'this' del objeto	

IV– Función Factoría

- No usa la palabra reservada *new*
- Sintaxis flexible
- Sintácticamente similares a clases

```
function CreateA() {
   return {
       publicAttribute: "A This is a public attribute.",
       publicMethod: function () {
           console.log("A This is a public method.");
       },
       publicMethod2: function () {
           console.log("This is another public method. ");
       }
   }
}
```

IV– Función Factoría

Pros	Limitaciones
Buena legibilidad	No soportan encapsulación

Herencia

- Clases
  - "extends ClassA"
- Operador spread
  - "...myObjectA"
- Función factoría y constructora
  - "let parent = new ConstructorA()"
- Cadena de prototipos
  - "ConstructorB.prototype = Object.create(ConstructorA.prototype)"

Decisión de estilo

- Closures
- Herencia con operador spread

```
function ClosureB() {
    let sup = ClousureA();

    function publicMethod() {
        console.log("This is an overriden inherited public method.");
    }

    return {
        ...sup,
        ...{
            publicMethod
        }
    }
}
```

Patrones de Diseño

- Patrones creacionales
  - Singleton
  - Builder
- Patrones de comportamiento
  - Template Method
  - Strategy
  - Memento
- Patrones estructurales
  - Composite
  - Decorator

Singleton

- Asegurar que una clase sólo puede tener una instancia y proveer un punto de acceso a ella.
- Implementado a través de **Patrón del Módulo.**
- Se ha implementado en la creación de los siguientes objetos:
  - Turn
  - BoardView
  - RestClient
  - MessageManager
  - RandomPlayer

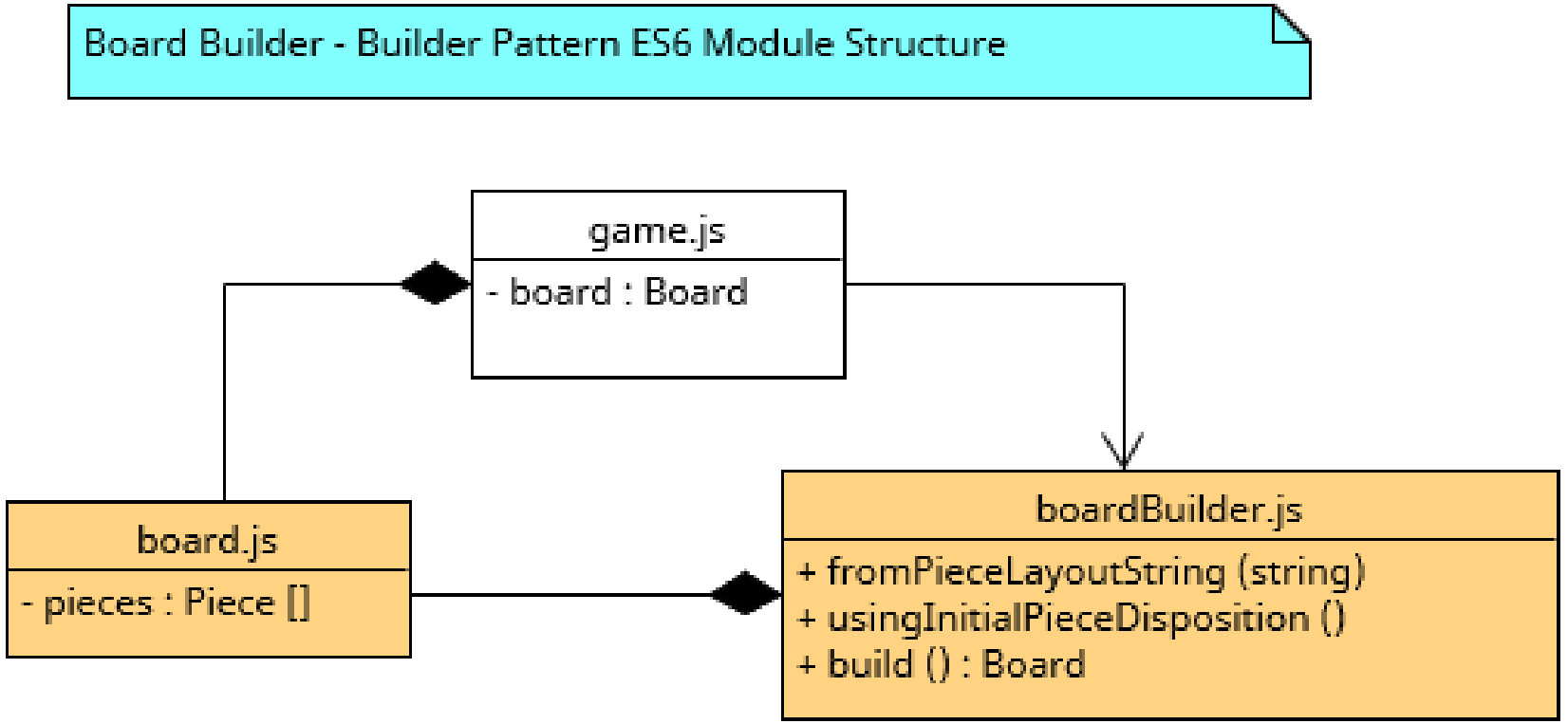
Singleton

```
const singleInstance = constructor();

function constructor() {
  function publicMethod1(){
      // method 1
  }

  function publicMethod2(errorMessage){
      // method 2
  }

  return {
      publicMethod1,
      publicMethod2
  }
}

export {
  singleInstance
}
```

Builder

- Separa la construcción de un objeto de su representación.
- En el chess:
  - El tablero precisa crearse con una disposición de piezas

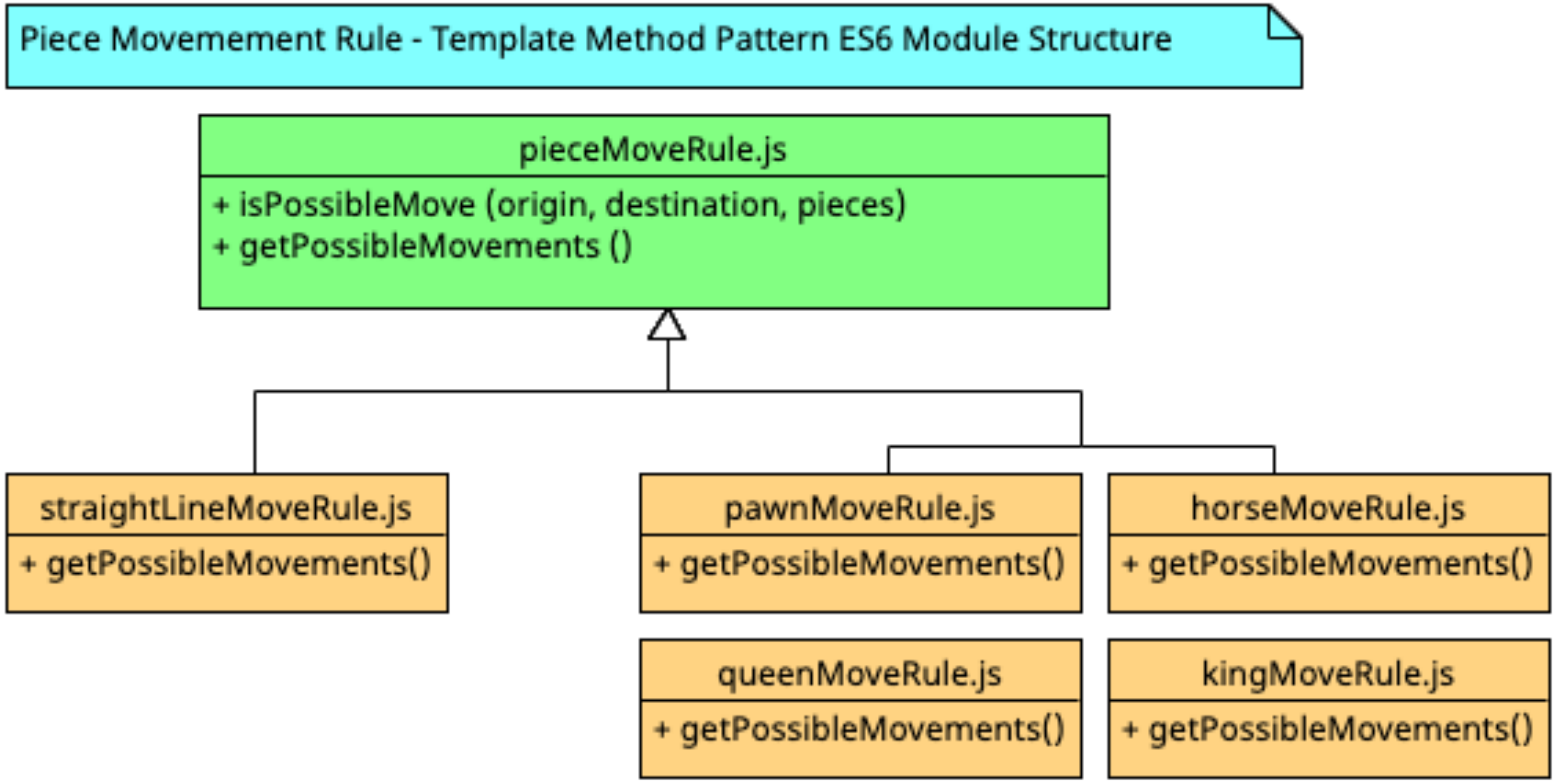
Builder


Board Builder - Builder Pattern ES6 Module Structure
game.js
- board : Board
board.js
- pieces : Piece []
boardBuilder.js
+ fromPieceLayoutString (string)
+ usingInitialPieceDisposition ()
+ build () : Board

Template method

- Creación del esqueleto de un algoritmo aplazando pasos a sus subclases.
- En el chess:
  - El método ***isPossibleMove()*** de la clase padre usa ***getPossibleMovements()*** implementado en las clases derivadas.

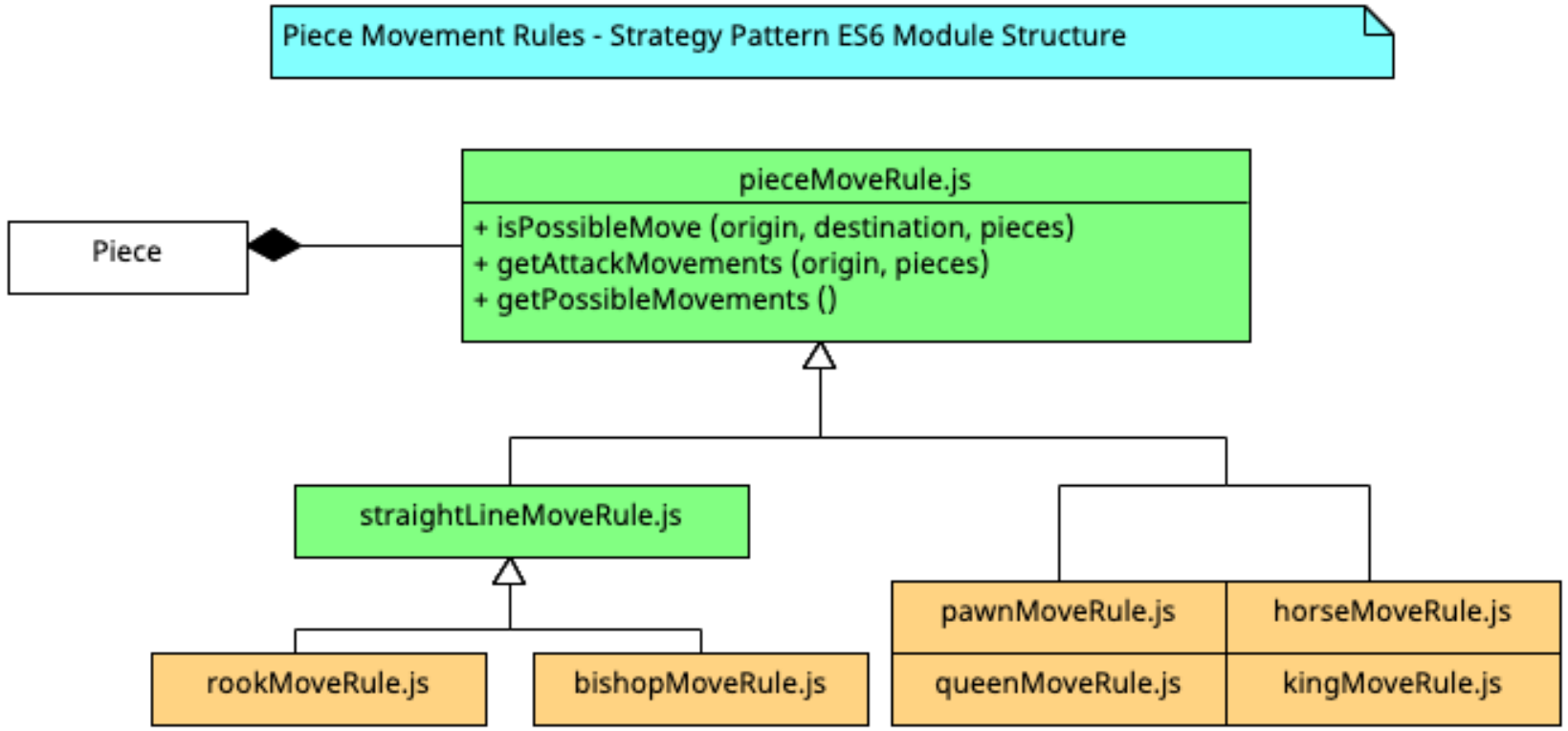
Template method



Strategy

- Define una familia de algoritmos, encapsulándolos y haciéndolos intercambiables en tiempo de ejecución.
- En el chess:
  - Se ha implementado para asignar a una pieza su algoritmo de movimiento.
  - Muy útil a la hora de coronar un peón cambiando su estrategia de movimiento en tiempo de ejecución.

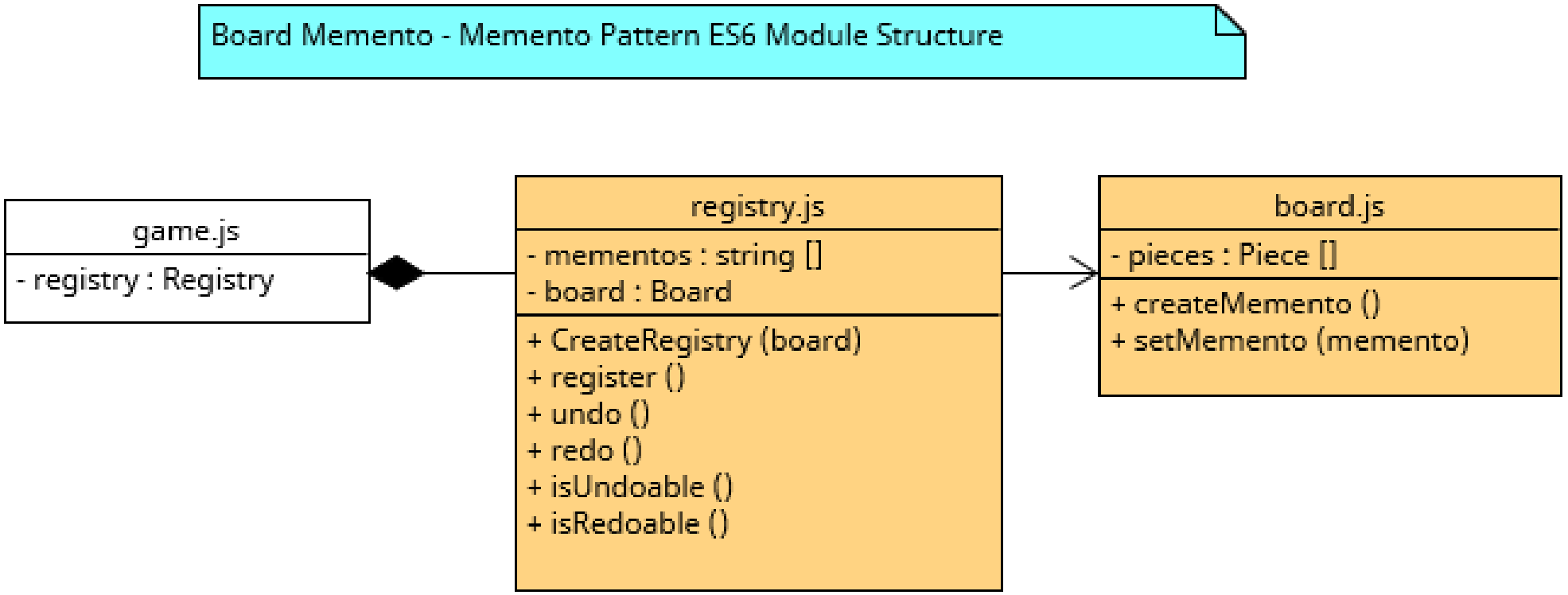
Strategy


Piece Movement Rules - Strategy Pattern ES6 Module Structure
pieceMoveRule.js
+ isPossibleMove (origin, destination, pieces)
+ getAttackMovements (origin, pieces)
+ getPossibleMovements ()
Piece
straightLineMoveRule.js
rookMoveRule.js
bishopMoveRule.js
pawnMoveRule.js
horseMoveRule.js
queenMoveRule.js
kingMoveRule.js

Memento

- Captura y externalizar la información de un objeto para restaurar su estado
- En el chess:
  - Undo (Deshacer último turno).
  - Redo (Rehacer último turno).

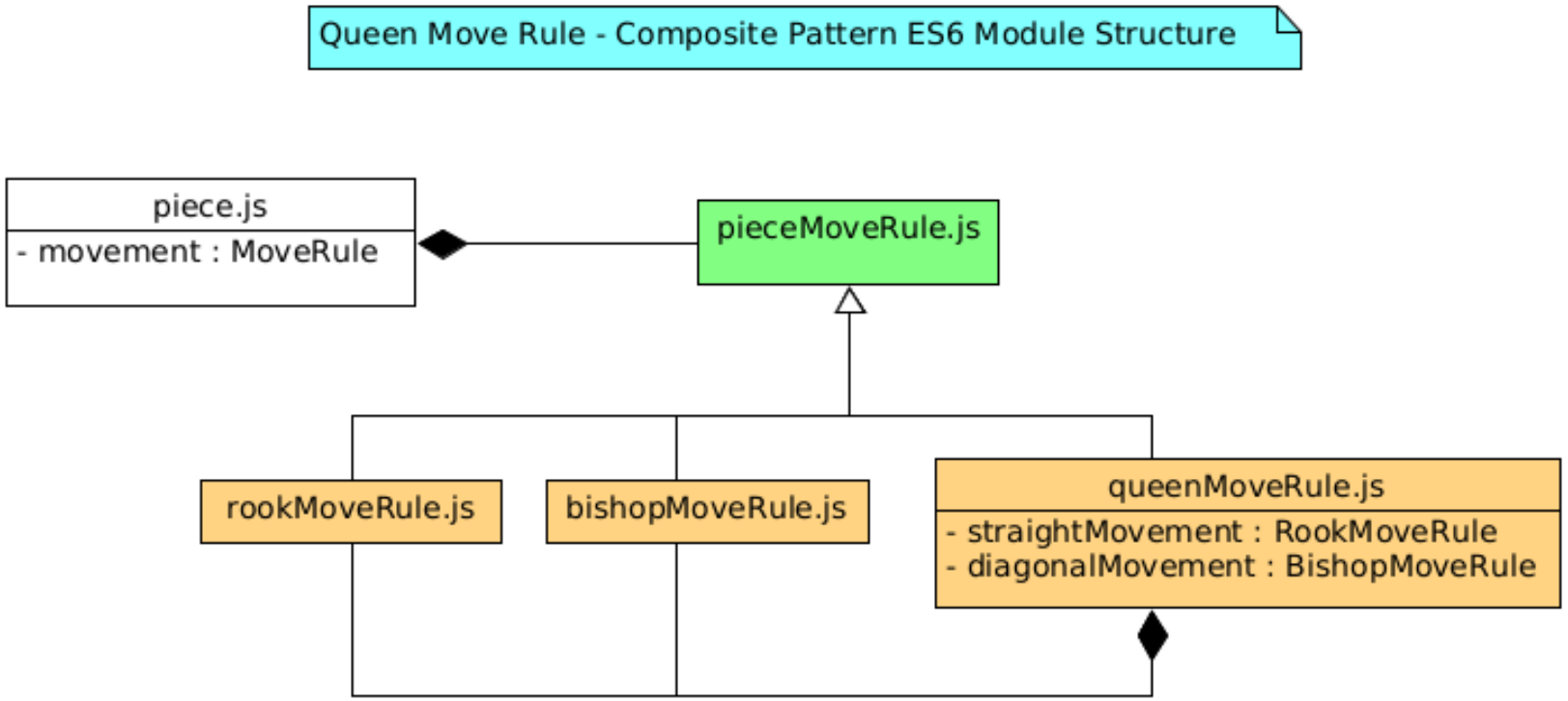
Memento


Board Memento - Memento Pattern ES6 Module Structure
game.js
- registry : Registry
registry.js
- mementos : string []
- board : Board
+ CreateRegistry (board)
+ register ()
+ undo ()
+ redo ()
+ isUndoable ()
+ isRedoable ()
board.js
- pieces : Piece []
+ createMemento ()
+ setMemento (memento)

Composite

- Facilita la creación de objetos con estructura de árbol.
- En el chess:
  - La lógica de movimiento de la pieza reina utiliza este patrón combinando un alfil y una torre.

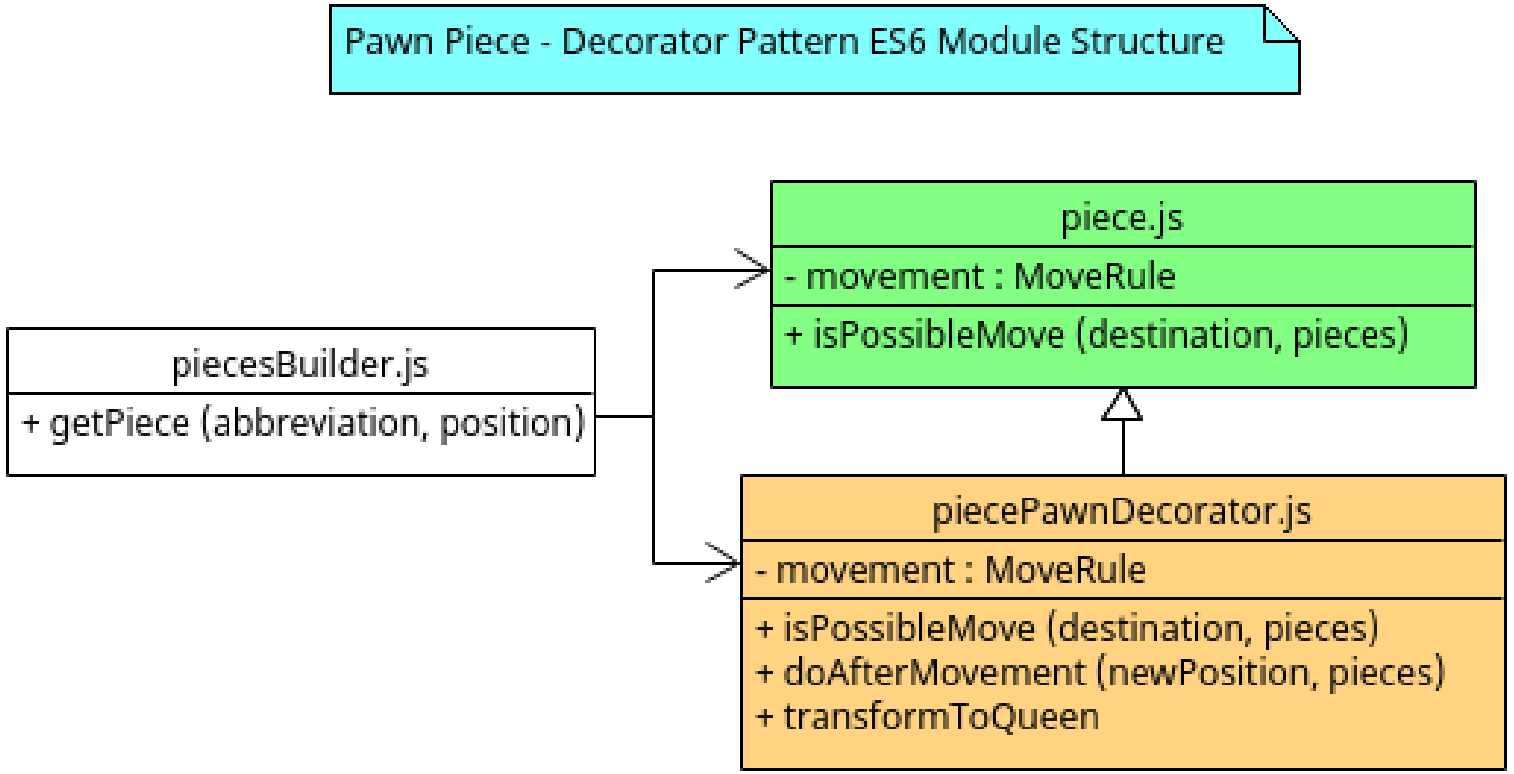
Composite


Queen Move Rule - Composite Pattern ES6 Module Structure
piece.js
- movement : MoveRule
pieceMoveRule.js
rookMoveRule.js
bishopMoveRule.js
queenMoveRule.js
- straightMovement : RookMoveRule
- diagonalMovement : BishopMoveRule

Decorator

- Permite agregar responsabilidades dinámicamente a un objeto.
- En el chess:
  - En una primera versión se utilizó para la coronación del peón.
  - Sustituido por el método **getNextMoveRule().**

Decorator


Pawn Piece - Decorator Pattern ES6 Module Structure
piece.js
- movement : MoveRule
+ isPossibleMove (destination, pieces)
piecesBuilder.js
+ getPiece (abbreviation, position)
piecePawnDecorator.js
- movement : MoveRule
+ isPossibleMove (destination, pieces)
+ doAfterMovement (newPosition, pieces)
+ transformToQueen

Pruebas de Software

- Test unitarios
  - Caja blanca:
    - Cobertura
  - Caja negra:
    - Clases de equivalencia
    - Valores límite

Componentes	Declaraciones	Ramas	Funciones	Líneas
main	97.18%	94.34%	96.83%	97.18%
moveRule	100%	100%	94.83%	100%
piece	100%	100%	100%	100%
Total	**98.85%**	**97.87%**	**96.73%**	**98.85%**

Integración Continua

- Ejecutar tests automáticos en los *PR*

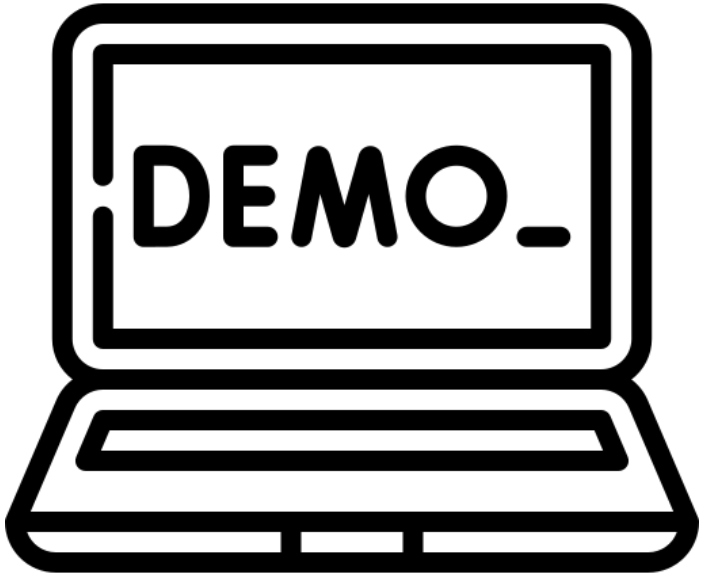
```yaml
name: Develop workflow

on:
  pull_request:
    branches: [ main ]
  workflow_dispatch:

jobs:
  test:
    name: Run jest tests
    runs-on: ubuntu-22.04
    steps:
    - name: Clone repo
      uses: actions/checkout@v3
    - name: Set up node
      uses: actions/setup-node@v3
      with:
        node-version: 16
    - name: Install dependencies
      run: npm ci
    - name: Execute tests
      run: npm test
```

Entrega Continua

- Ejecuta los tests
- Generación de imagen docker
- Publicación en *GitHub Packages*

Demo

- Ejecución del motor de ajedrez desde Docker
- Flujo de juego



Conclusión

- Se cumple objetivo principal
- Conclusiones sobre POO en JavaScript
- Trabajo futuro:
  - Modo multijugador y patrón State
  - Patrón Command para movimientos
  - Más capacidades del jugador CPU

Gracias

- ¿Preguntas?