Máster en Cloud Apps

Desarrollo y despliegue de aplicaciones en la nube

Curso académico 2021/2022

Trabajo de Fin de Máster

Patterned NodeJS

Autores: Lourdes Morente, José Manuel Ramos, Luis Boto

Tutor: Luis Fernández

Índice

Resumen

JavaScript se ha convertido en uno de los lenguajes de programación más utilizados en el desarrollo web, originalmente en el lado del cliente y más recientemente en el lado del servidor. Sus características de tipado débil y adecuación tanto a la orientación a objetos como a la programación funcional le dotan de una amplia versatilidad. Sin embargo, su gran variabilidad ha derivado en prácticas poco ortodoxas y desacuerdos que se han extendido en la comunidad, discrepando respecto al estilo y las pautas óptimas de programación que se deben seguir. Un ejemplo notorio de estas controversias es la no utilización de clases, aún en el marco de la programación orientada a objetos.

La motivación de este trabajo es explorar la implementación de los patrones de diseño clásicos definidos por la *Gang of Four* en JavaScript. La referida discrepancia en estilos provoca que la aplicación de estos patrones no sea trivial, dado que la programación orientada a objetos se articulará prescindiendo de la utilización de clases; base tradicional para estos patrones.

La aplicación de los patrones se ha realizado sobre sobre un **motor de ajedrez** **jugable**, desarrollado ad-hoc y basado en **NodeJS** como plataforma. Se ha realizado un análisis de los principales **estilos de programación** en JavaScript y un estudio de los **patrones de diseño clásicos** de la programación orientada a objetos. Se ha asegurado un nivel adecuado de robustez y funcionalidad del software mediante la realización de **pruebas unitarias**, diseñando un conjunto de casos de prueba fundamentados en una estrategia eficaz y razonable para la detección prematura de errores. Se han utilizado los procesos de **Integración Continua y Entrega Continua** para la automatización y la integración del código entre el equipo, apoyándose en la ejecución de los tests de software para garantizar el funcionamiento del sistema en cada versión.

Esta memoria se estructura en cuatro secciones: una preliminar, *Introducción y objetivos*, que establece las metas del trabajo; una sección principal, descriptiva del desarrollo llevado a cabo y constituida a su vez por cuatro capítulos claramente diferenciados (*I - Orientación a Objetos en JavaScript, II - Patrones de diseño en JavaScript, III - Pruebas de software e IV - Integración Continua y Entrega Continua*); un análisis sobre los resultados del trabajo y evoluciones sugeridas en *Conclusiones y trabajo a futuro*; y una última sección de *Bibliografía*, que recoge las referencias utilizadas. Por completitud se ha incluido un Anexo que documenta de manera concisa las pruebas unitarias.

Introducción y objetivos

JavaScript se caracteriza por su pragmatismo y naturaleza híbrida, contando con una gran versatilidad en su codificación. Se ha convertido en el lenguaje de programación web más empleado en la actualidad, NodeJS ha contribuido a su auge proporcionando un entorno para su ejecución del lado del servidor. No obstante, este crecimiento ha estado acompañado por las controversias dentro de la comunidad, la exploración de estilos de orientación a objetos de este trabajo surge debido a la disputa sobre el uso de clases.

El diseño de un software orientado a objetos reutilizable y flexible es complejo, es necesario establecer las jerarquías de herencia adecuadas y la identificación de las relaciones clave entre ellas, a la vez que se evita el rediseño o al menos se minimiza. Persiguiendo este fin, se aplican los patrones clásicos de la programación orientada a objetos definidos por la *Gang of Four* como el conjunto de soluciones habituales a los problemas comunes de diseño (Gamma et al., 1995).

En este trabajo, debido al estilo de codificación escogido, se ha introducido una dificultad añadida a la implementación de los patrones clásicos respecto a un estilo estructurado en clases. Se ha explorando la mejor implementación alcanzada gracias a las facilidades que nos aporta JavaScript. Se debe considerar que estas variaciones son derivadas tanto del estilo de código utilizado como del entorno débilmente tipado.

El **objetivo principal** del proyecto 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.

Para alcanzar este propósito se plantean los siguientes objetivos secundarios: analizar los estilos de codificación más prominentes en la comunidad de JavaScript, implementar los patrones de diseño requeridos por la necesidad del software, estudiar la diferencia entre la implementación aplicada y la estructura clásica en aquellos que sea de especial interés, garantizar la correcta funcionalidad siguiendo un diseño de pruebas eficaz y automatizar las etapas de integración y de entrega durante el desarrollo de software.

Las metas planteadas se abordan con: la guía de estilos de JavaScript, recogiendo las alternativas más prominentes y permitiendo una fácil transformación de un estilo a otro; el motor de ajedrez desarrollado en NodeJS donde se pone en práctica la implementación de los patrones de diseño y la propia documentación recogida en esta memoria.

Para la consecución de los objetivos marcados se pretende elaborar: una guía de estilos de JavaScript con las alternativas de mayor impacto, un motor de ajedrez donde se pone en práctica la implementación de los patrones de diseño acompañados de una batería de pruebas unitarias y dos workflows que implementen la integración y entrega continua.

La siguiente sección compone la parte central del documento, estructurada en cuatro capítulos diferenciados. Los dos primeros capítulos aportan la base sobre la que se apoya el diseño del software desarrollado. En el primer capítulo se analizan las fortalezas y limitaciones de las principales aproximaciones de la orientación a objetos en JavaScript,

escogiendo el uso de **módulos de ES6** y **closures** para los patrones de diseño aplicados. En el segundo capítulo se presentan los patrones clásicos implementados en el ajedrez, tales como el *Builder*, *Strategy* o el *Composite*.

El tercer capítulo describe las técnicas utilizadas en la disciplina de pruebas para garantizar una estrategia razonable que permita validar y verificar la funcionalidad del sistema. Sobre estas metodologías se respalda el conjunto de **pruebas unitarias** desarrolladas en el motor de ajedrez.

Por último, en el cuarto capítulo se describen brevemente los procesos de Integración Continua y Entrega Continua enfocados en la automatización de las etapas de integración y entrega de código. Incluyendo las herramientas empleadas para conseguir esta finalidad: *GitHub Actions y GitHub Packages*.

I - Orientación a objetos en JavaScript

Inicialmente, JavaScript surge como una implementación del lado del cliente en los navegadores, consiguiendo que las páginas web evolucionaran hasta ser interactivas, gracias a su capacidad de modificar dinámicamente el Modelo de Objeto del Documento (Document Object Model – DOM).

Posteriormente, con la llegada de NodeJS se habilitó un entorno multiplataforma para la ejecución de JavaScript de manera asíncrona con entrada y salida de datos en una arquitectura orientada a eventos en el lado del servidor (Du Preez, 2020). Esta herramienta no solo ha contribuido a su auge y evolución, además ha supuesto un cambio en el paradigma de la programación web permitiendo un único lenguaje de programación en los distintos componentes de la aplicación (Casciaro, 2014).

JavaScript se define como un lenguaje **interpretado de alto nivel con tipado débil**, destacando por su pragmatismo y naturaleza híbrida, a medio camino entre la programación orientada a objetos y el paradigma funcional. Gracias a estas características permite una amplia versatilidad en el diseño de código.

Sin embargo, la principal ventaja del lenguaje es a la vez su mayor debilidad. La gran variabilidad ha derivado en prácticas poco ortodoxas y controversiales que se han extendido en la comunidad de JavaScript. Discrepando respecto al estilo y las pautas óptimas de programación, llegando a la discusión actual sobre el uso de clases. Se debe considerar que cada estilo de programación puede cambiar radicalmente el aspecto y la dirección de un desarrollo pese a ser equivalentes funcionalmente.

La **Programación Orientada a Objetos (POO)** es un paradigma basado en la correspondencia intuitiva entre la simulación de software y el mundo físico. Por analógica, los componentes de software se denominan objetos y se crean con el fin de manipular los datos a la vez que se ofrecen sus funcionalidades especiales (Abadi & Cardelli, 1996).

JavaScript soporta la orientación a objetos, a menudo esta afirmación sorprende a los desarrolladores que lo han analizado. Si partimos de la definición de un objeto como una colección de propiedades conocidas como atributos y funciones propias denominadas métodos cualquier fragmento de código en JavaScript podría ser un objeto, incluso las propias funciones lo son por tener propiedades y métodos asociados (Stefanov, 2010).

La inusualidad de JavaScript como un lenguaje que soporta la POO reside en que **hasta la llegada de ECMAScript 6 en junio de 2015 no existían las clases**, pese a esto se sigue desaconsejando su uso dentro de la comunidad. Sus detractores se basan en argumentos como: *“no usar clases dota al lenguaje de mayor flexibilidad por evitar el uso de jerarquías de clases y sus restricciones de sintaxis”, “permite que los programas sean mucho más cortos por no necesitar una clase para la creación de objetos”* o “*las clases en JavaScript no dejan de tratarse de azúcar sintáctico”.* Este último argumento, se debe a que por debajo implementan funciones constructoras ya utilizadas en versiones anteriores.

Estilos de codificación

Tomando en consideración las alternativas a la orientación a objetos más prominentes en la comunidad de JavaScript se ha realizado un estudio y una comparativa partiendo de un mismo objeto creado con: *Clases*, *Closures*, *Funciones Constructoras* y *Función Factoría.* Se analizan las ventajas, las limitaciones y la jerarquía de herencia en cada una de ellas.

El código representado en los ejemplos se encuentra disponible en el repositorio de este proyecto bajo el directorio ***/codingStyles***.

Este capítulo no debe ser interpretado como una documentación complementaria al trabajo presentado, sino como la evaluación previa en la búsqueda de un estilo unificado con el que construir el motor de ajedrez. Adicionalmente, profundizar en estas opciones de codificación permite que el propio software pueda ser fácilmente adaptado de un estilo a otro.

Clases

Se inicia el estudio con la representación clásica para la creación de objetos en la POO, la clase se va a utilizar como base comparativa al resto de estilos sin contemplarse como una de las opciones empleadas debido a su controversia.

Métodos y atributos públicos

Los métodos y los atributos de la clase, por defecto son públicos, de manera que su creación es intuitiva y sencilla.

```
class ClassA {

   constructor () {
       this.publicAttribute = "This is a public attribute.";
    }

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

let objA = new ClassA();
console.log(objA.publicAttribute);
objA.publicMethod();
```

Cualquier objeto instanciado a partir de la clase *ClassA* puede acceder a los atributos y métodos de acceso público declarados en la clase.

Métodos y atributos privados

Las propiedades privadas no son nativas en JavaScript, su incorporación es relativamente reciente. Surgen en la versión de **ECMAScript 11 publicada en junio de 2020 aportando consigo la encapsulación a las clases**.

Anteriormente, existía una convención para la definición de atributos de acceso privado, consistía en agregar un guión bajo antes del nombre de la propiedad (*Ejemplo: this._name*). No se debe olvidar que no es una funcionalidad del lenguaje y fuera del contexto de la clase se puede modificar su valor.

Los atributos de acceso privado son declarados antes de su uso dentro de la propia clase. A partir de la versión ES11 se pueden definir estas propiedades anteponiendo el carácter # a su nombre, igualmente se usa esta misma referencia para su acceso.

```
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);
   }
}
```

Si se intenta acceder fuera del contexto de la clase a una propiedad privada se obtendrá un error, igualmente, tampoco será posible el alcance desde una clase derivada. Se debe implementar específicamente métodos públicos para su acceso o modificación, como pueden ser los **getter / setter**, mejorando así la encapsulación de la estructura.

Métodos y atributos estáticos

La definición de propiedades estáticas en la clase se realiza mediante la palabra reservada **static**.

```
class ClassA {

   static STATIC_FIELD = "This is a static field";
   static #PRIVATE_STATIC_FIELD = "This is a private static field.";

   constructor () {
   }

   static publicStaticMethod() {
       console.log("This is a static method.");
   }

   static #privateStaticMethod(){
       console.log("This is a private static method.")
   }
}

console.log(ClassA.STATIC_FIELD);
ClassA.publicStaticMethod();
```

Closures

JavaScript no presenta una sintaxis especial para definir las propiedades protegidas o públicas a diferencia de otros lenguajes de programación como Java e incluso las propiedades privadas son una incorporación muy reciente. No obstante, ya se podía conseguir la encapsulación fácilmente a partir de las **closures**. Una función de primer nivel crea una closure y cualquier **variable que sea parte de su ámbito no estará expuesta públicamente** en el constructor (Stefanov, 2010) quedando así su acceso limitado y concediendo una manera confiable de **encapsulación**.

Una función en JavaScript es considerada una closure cuando se definen sus atributos como variables locales y sus métodos públicos como funciones, **solamente los métodos tendrán acceso privilegiado a sus atributos**, no como copias sino a ellos mismos (*Douglas Crockford, 2008).* Una closure es la combinación de una función junto a la referencia de su entorno inmediato, es decir, su **entorno léxico**, (*Closures - JavaScript | MDN*, 2022).

Métodos públicos

Los métodos públicos en las closures se definen como funciones y para conseguir el acceso público se incluyen en el *return* de la función de primer nivel.

Atributos y funciones privados

Sus atributos por definición son privados, se definen dentro del contexto de la closure y no deben ser retornados, quedando su acceso limitado al contexto de la función.

Aquellas funciones no incluidas en el *return* tiene un acceso privado y pueden ser invocadas directamente en el ámbito de la closure. Las funciones privadas tienen acceso a los atributos al igual que sucede con los métodos públicos.

```
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. ");
       }
   }
}
```

Métodos y atributos estáticos

En JavaScript fuera del ámbito de la clase **no se permite el uso** de la palabra reservada **static**, por tanto, los atributos y métodos estáticos se añaden como propiedades del objeto función. De esta manera se definen las propiedades estáticas **tanto en las closures como en el resto de estilos de codificación** de esta documentación,

```
ClosureA.STATIC_FIELD = "This is a static field";
console.log(ClosureA.STATIC_FIELD);

ClosureA.publicStaticMethod = function() {
   console.log("This is a static method.");
}
```

Ventajas

Las closures se plantean como una solución a la encapsulación en JavaScript definiendo propiedades privadas en su contexto, sin hacer uso de la sintaxis especial utilizada en las clases. Además este estilo admite la creación de métodos públicos.

Limitaciones

Partiendo de la propia definición de las closures no es posible la creación de atributos públicos a los que se acceda fuera del ámbito de la propia función.

El acceso y modificación de los atributos definidos dentro de la closure se puede solventar mediante la creación de métodos *getter* / *setter* específicos para ello. De este modo, es posible trabajar con sus atributos desde un contexto externo a la propia closure.

Uso del *That*

Las closures no tienen métodos privados, la lógica de acceso privado queda limitada a la utilización de funciones. Sin embargo, si al concepto de closure añadimos el uso del ***that*****,** entendido como un objeto que contiene la parte privada, las funciones se comportan como métodos siendo parte del contexto del *that*. Se trata, por tanto, de una alternativa a *this* en las invocaciones dentro del ámbito de la closure.

```
function closureA() {
  let that = {
      privateAttribute: "This is a private attribute.",en
      privateMethod: function () {
          console.log("This is a private method.");
          console.log(that.privateAttribute);
      }
  }

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

  return {
      publicMethod
  }
}
```

Función constructora

Una función constructora en JavaScript define el prototipo de propiedades de un objeto, **invocando a la función con** la palabra reservada **new se crean objetos** del mismo tipo.

La palabra clave ***this* adquiere especial relevancia** en las funciones constructoras. Ésta contiene el valor del objeto en el contexto de ejecución de un fragmento de código. Particularmente, dentro de la función constructora, *this* ejerce como sustituto del nuevo objeto. Posteriormente, *this* almacenará la dirección del objeto cuando se ejecute la función constructora.

Métodos y atributos públicos

En la función constructora los atributos y métodos públicos se definen dentro de *this*, cuando se ejecute la función el objeto podrá acceder a dichas propiedades.

Funciones privadas

Las funciones constructoras **no permiten la encapsulación de atributos ni métodos**, únicamente pueden declarar funciones privadas. Las funciones privadas en este contexto no pueden acceder a *this* ni al resto de sus propiedades.

```
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);
   }
}
```

Métodos y atributos estáticos

Las propiedades estáticas, al igual que en las closures, se añaden como propiedades del objeto función.

Ventajas

Las funciones constructoras permiten la creación de objetos con propiedades públicas de una manera sencilla.

Limitaciones

En este estilo de codificación no es posible la encapsulación de propiedades, dado que no se pueden declarar atributos ni métodos privados y las funciones privadas quedan limitadas por no poder acceder al resto de propiedades del ámbito de la función constructora.

Adaptación hacia closures

La limitación de las funciones constructoras, para el uso de la encapsulación, se puede solventar aplicando una aproximación hacia las closures, como se muestra en el siguiente

fragmento de código. De esta manera, **se seguirá usando *this* y *new***, pero **teniendo la posibilidad de definir atributos limitando su acceso privado**.

```
function Clazz(privateAttributeX) {
   let privateAttributeY = 0;

   this.publicMethod = function () {
           privateFunction();
           console.log(`privateAttributeX: ${privateAttributeX} -
privateAttributeY: ${privateAttributeY}`);
   }

   function privateFunction() {
       privateAttributeX++;
       privateAttributeY++;
   }
}
```

Uso de prototype

Cada vez que se invoca una función constructora **se crean los métodos del objeto en memoria**, pudiendo resultar ineficiente (Stefanov, 2010). Este inconveniente está presente en todos los estilos de codificación mencionados hasta ahora.

Los prototipos aportan una solución para mejorar el rendimiento. El uso de prototipos permite declarar los métodos de la función constructora como propiedades del objeto prototipo en lugar de en cada objeto instanciado, definiendo tantos atributos como métodos de acceso público contenga la función. Además, la cadena de prototipos es muy práctica **a la hora de implementar la jerarquía de herencia** en JavaScript.

```
Clazz.prototype.publicMethod = function () {
   privateFunction(this);
   console.log(`publicAttributeX: ${this.publicAttributeX} -
publicAttributeY: ${this.publicAttributeY}`);

   function privateFunction(object) {
       object.publicAttributeX++;
       object.publicAttributeY++;
   }
}
```

Funciones factoría

Las funciones factoría en JavaScript son similares al constructor de una clase, sin embargo, no requieren para su construcción la utilización del operador *new* ni el uso de *this* para definir sus propiedades. La propia función factoría se encarga de la creación del objeto y de su retorno.

Métodos y atributos públicos

El objeto instanciado con la función factoría se define en el *return* de la función partiendo sus métodos y atributos, sus propiedades tendrán un acceso público.

```
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. ");
       }
   }
}
```

Métodos y atributos estáticos

Las propiedades estáticas se añaden como propiedades del objeto función, al igual que en el caso de las closures y las funciones constructoras.

Ventajas

Las funciones factoría, al igual que las funciones constructoras, permiten crear objetos con propiedades públicas de manera legible y sencilla. La diferencia con el estilo anterior radica en que las funciones factoría no hacen uso del operador *new* para la creación de objetos.

Limitaciones

En estas funciones, de la misma manera que en las funciones constructoras, no es posible la realización de la encapsulación de propiedades.

Adaptación hacia closures y uso de prototipos

Las funciones factoría también pueden adaptar su estilo hacia las closures para definir propiedades de acceso privado.

Su invocación es igualmente ineficiente por la creación de los métodos del objeto en memoria, este inconveniente se solventa de manera sencilla mediante el uso de prototipos (*Ejemplo: Object.create()*).

Herencia

En la Programación Orientada a Objetos (POO) la herencia es el mecanismo que permite la creación de clases a partir de una clase o jerarquía de clases preexistente. La clase derivada hereda las operaciones o estructura interna de la superclase. Entre las ventajas de la herencia destaca la reutilización de código (Johnson & Foote, 1988).

Clases

La herencia en las clases puede implementarse de una manera intuitiva, gracias al azúcar sintáctico que facilita JavaScript, **haciendo uso de la palabra reservada *extends*** del mismo modo que en otros lenguajes de programación. Se puede extender la funcionalidad a otra clase derivada, sobrescribir, invocar y añadir métodos de la superclase.

```
class ClassB extends ClassA {

   constructor() {
       super();
   }

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

   publicMethod2() {
       console.log("This is an augmented inherited public method.");
       super.publicMethod2();
   }

   publicMethod3() {
       console.log("This is a new child-exclusive public method.");
   }
}
```

Closures

Las closures pueden implementar la **herencia mediante el uso del operador *spread***. Este operador permite retornar un objeto como resultado de la concatenación de otros objetos, pudiendo así obtener un objeto derivado de la closure padre cuyas funciones redefinidas son sobreescritas.

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

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

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

Funciones factoría

Las funciones factoría pueden **establecer la jerarquía de herencia declarando el objeto padre y añadiendo nuevas propiedades**, de este modo, se consigue extender la funcionalidad del primer objeto y sobrescribir sus métodos. Otra alternativa, es asignar un método del objeto padre a un método del objeto hijo.

```
function CreateB() {
   let B = CreateA();

   B.publicAttribute = "B This is an overriden public attribute."

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

   B.publicMethod3 = function () {
       console.log("This is a new child-exclusive public method.");
   }

   return B;
}
```

Función constructora

La función constructora aplica la jerarquía de herencia del mismo modo descrito en las funciones factoría. Ambos estilos cuentan con otra alternativa más para la realización de la herencia mediante el uso de prototipos.

Cadena de prototipos

En cuanto a la herencia, en JavaScript cada objeto tiene una propiedad privada que contiene un enlace a otro objeto denominado su **prototipo**. El objeto prototipo tiene un prototipo propio, y así sucesivamente hasta llegar a un objeto con null como prototipo. Este mecanismo es conocido como **cadena de prototipos y aporta gran facilidad a la implementación de la jerarquía de herencia**, siendo posible mutar cualquier miembro de la cadena en tiempo de ejecución. Las clases en JavaScript implementan este mismo mecanismo por debajo, siendo transparente para el programador.

```
function ConstructorB() {
   ConstructorA.call(this);
}
ConstructorB.prototype = Object.create(ConstructorA.prototype);
ConstructorB.prototype.constructor = ConstructorB;
ConstructorB.prototype.publicMethod = function () {
   console.log("This is an overriden inherited public method.");
};
```

Guía de estilo

Seguidamente del análisis de las alternativas más prominentes a las orientación a objetos en JavaScript, se decide aplicar **las closures** para la construcción de objetos.

Las closures **permiten la encapsulación** de una manera sencilla creando un contexto privado para las propiedades y responsabilidades propias de cada modelo que no deban estar accesibles o ser modificadas por otros componentes. Adicionalmente, permite el **acceso público de los métodos** que se incluyan en el *return* de la función de primer nivel. La **jerarquía de herencia** se implementa mediante el uso del operador spread en el *return* de la función principal.

La limitación del acceso y modificación de atributos públicos fuera del contexto de la closure puede ser solventado mediante la creación de métodos públicos específicos para ello, definiendo métodos **getter** / **setter** cuando sea pertinente.

Se ha empleado una arquitectura basada en los **módulos de ES6** favoreciendo el uso de la encapsulación de una manera más efectiva, ya que solo quedarán accesibles por el resto de componentes aquellos objetos o funciones que se exporten explícitamente. Esta estructura ha sido de gran relevancia para la encapsulación de **closures** y **singletons**.

Definición de módulos

Se deberá crear un **archivo por cada módulo**, **su nombre será un sustantivo** representativo del modelo del dominio y / o funcionalidad **seguido de la extensión .js.**

Partiendo de la convención de estilo mencionada en el libro *JavaScript : the good parts de 2008, "Todas las funciones constructoras se nombran con una mayúscula inicial y nada más se escribirá con mayúscula*". Dado que en el software desarrollado se hace uso de la orientación a objetos basada en closure, se nombran todas las funciones con la primera letra en minúscula siguiendo el formato ***lowerCamelCase*** como se muestra en el siguiente fragmento.

```
function functionNameOnLowerCaseCamelCase (parameter1, parameter2) {
   // Function code
}
```

La exportación de elementos de un módulo **se realizará al final** del mismo, **mediante la palabra reservada *export*.** No estará permitido la exportación de funciones o variables utilizando *default export* o *export* en su propia declaración.

```
export {
   function1,
   function2,
   constant1,
}
```

La importación de elementos se efectuará mediante la palabra clave ***import,*** evitando el uso de *require*.

```
import { function1, function2 } from './ExampleModule.js';
```

Enumerados

En JavaScript, a diferencia de otros lenguajes de programación, no existe una estructura propia para la definición de enumerados. No obstante, **se pueden implementar gracias a la combinación conjunta de las closure y la propiedad freeze() de los objetos.** Se han definido los enumerados siguiendo el estilo indicado en el siguiente fragmento.

```
const SeasonEnum = Object.freeze({
   Winter: createSeason("Winter"),
   Summer: createSeason("Summer"),
   Autumn: createSeason("Autumn"),
   Spring: createSeason("Spring"),
});

function createSeason(season){
   let name = season;
   function methodA(){
       return "Season: " + name;
   }
   return{
       methodA
   }
}
```

II - Patrones de diseño en JavaScript

Muchos sistemas orientados a objetos cuentan con patrones recurrentes entre sus clases, enfocados a resolver los problemas específicos del software permitiendo diseños más flexibles, elegantes y en última instancia reutilizables. Motivado por ello, se definen los patrones de diseño como un conjunto de soluciones habituales a los problemas comunes en el diseño de software. Cada patrón es una guía general que puede personalizarse para resolver los problemas concretos con los que se encuentra cada desarrollador (Gamma et al., 1995).

En este proyecto se ha elaborado un **motor de ajedrez** en JavaScript ejecutado en NodeJS. Se ha desarrollado aplicando los patrones de diseño clásicos de la programación orientada a objetos definidos por la *Gang of Four,* haciendo uso de la POO basada en closures.

Se han contemplado los tres tipos de patrones: **Patrones Creacionales, Patrones Estructurales y Patrones de Comportamiento.** La decisión de su implementación ha sido motivada por el diseño de las distintas funcionalidades del juego, con el objeto de solventar los problemas surgidos debido a las particularidades del dominio.

En este capítulo se presentará brevemente la estructura general de la aplicación, proporcionando el contexto sobre el cual han sido utilizados los patrones, se ahondará en ellos incluyendo un pequeño análisis de su implementación en JavaScript.

Esquema general de la aplicación

El ajedrez se ha elaborado siguiendo la arquitectura software **Modelo-Vista-Controlador** (MVC), comúnmente empleado para implementar interfaces de usuario y lógica de control, manteniendo por separado la lógica de negocio y la visualización. El MVC nos ha permitido desacoplar la aplicación en dos componentes principales: backend y frontend**, ambos codificados en JavaScript, corriendo en el lado del servidor con NodeJS.**

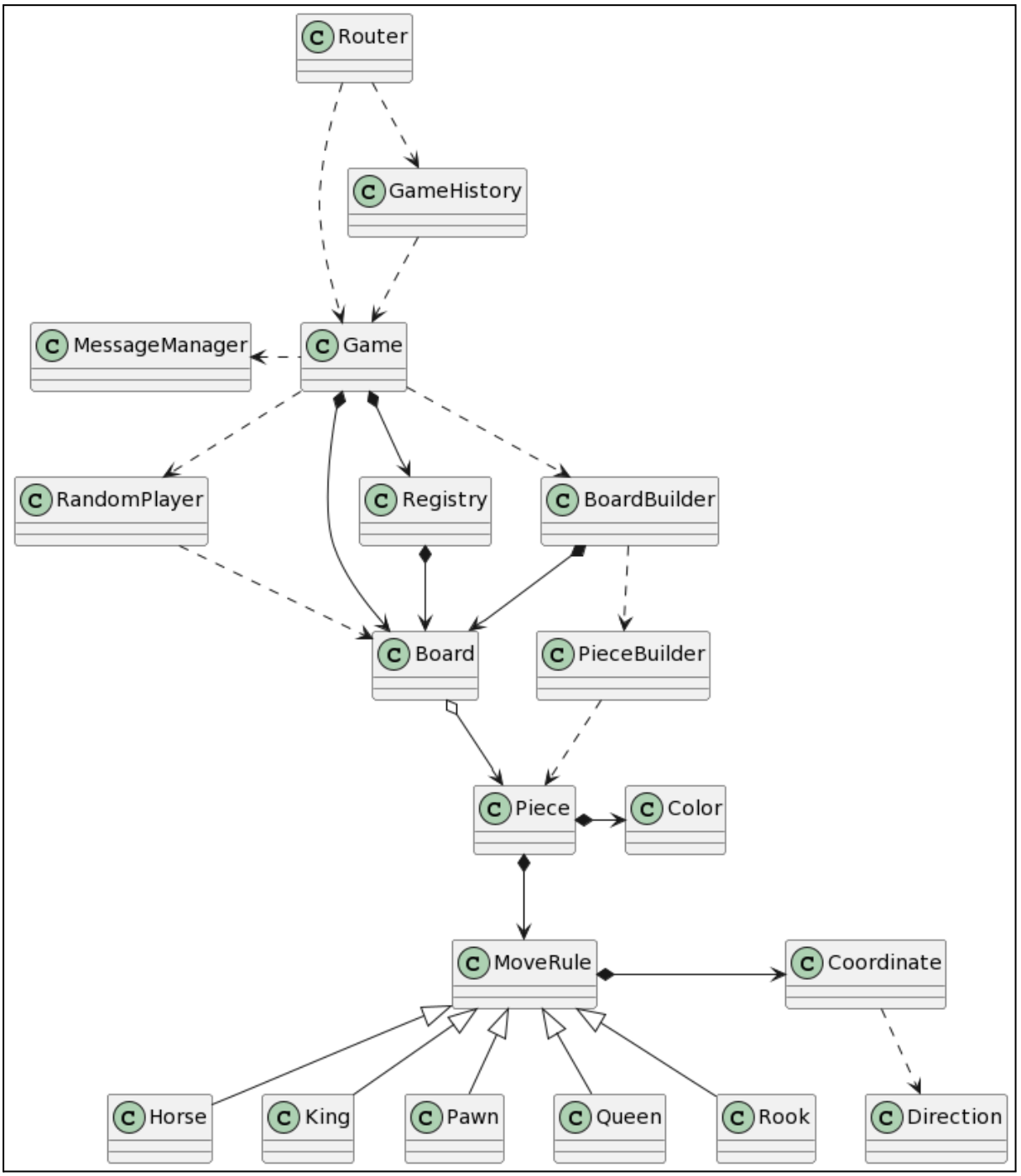
Backend

En el backend diferenciamos dos partes de la arquitectura MVC: **modelo** y **controlador**.

El modelo es la capa con la lógica de negocio de la aplicación, se encarga de la gestión de la información, contando con los mecanismos necesarios para acceder a ella y actualizar su estado. Este componente corresponde a los módulos propios que conforman el modelo del dominio del ajedrez (tablero, juego, coordenada…).

El controlador se encarga de gestionar los eventos y acciones del usuario, actuando como un intermediario entre la vista y el modelo. El backend es invocado desde la vista, a través del consumo de las operaciones REST definidas en el enrutador (*router*), permitiendo acciones como la creación del juego, el movimiento de piezas o el undo / redo entre otras. A su vez, el controlador redirige estas acciones al modelo, siendo este elemento quien realiza la lógica del juego y determina su estado.

En el repositorio del proyecto se ha facilitado una pequeña documentación en la que se puede observar con más detalle la relación entre los módulos del backend: Diagrama de clases del backend.



Frontend

En el frontend encontramos la **vista** de la arquitectura MVC, encargada de generar la interfaz gráfica con la que interactúa el usuario. Esta capa se ha dividido en tres componentes: **GameView**, **PlayerView** y **BoardView**, cada una de ellas representando su correspondiente elemento del modelo del dominio.

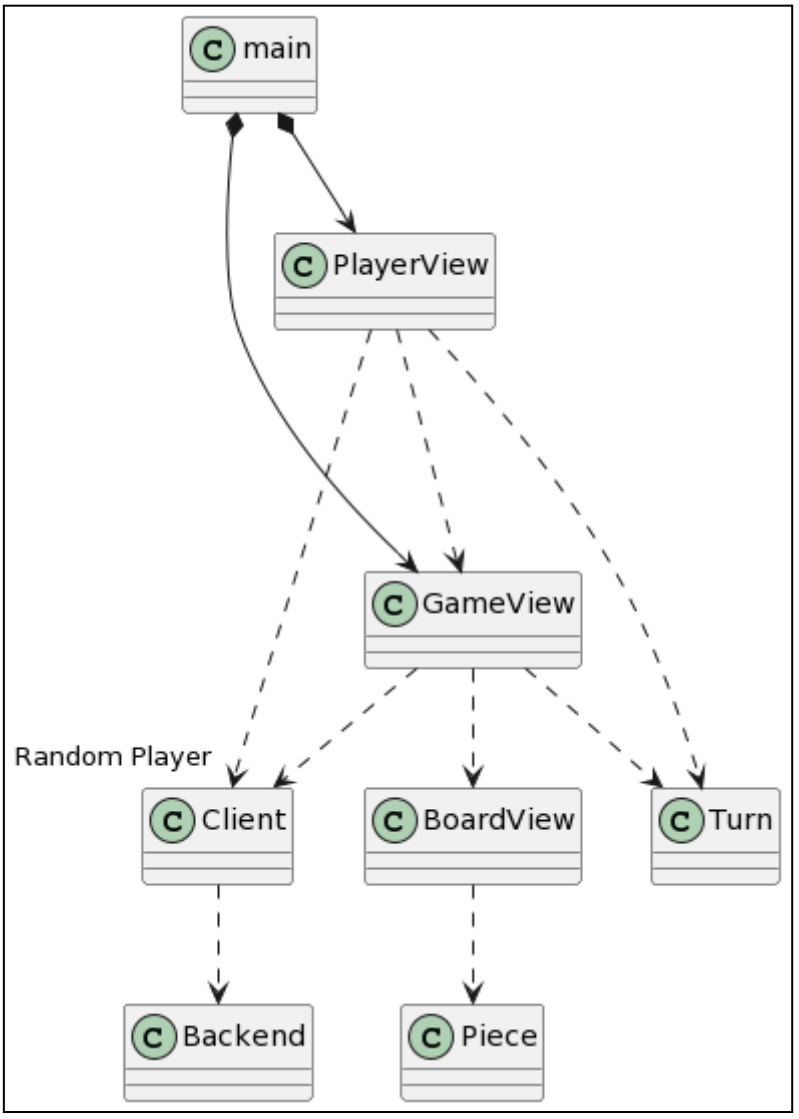
El **GameView** se responsabiliza de invocar a las acciones principales del juego: inicialización del tablero, realización de un movimiento o el undo / redo. Cuando esta vista obtiene la respuesta del backend, delega la acción de pintar el tablero al **BoardView**.

El **BoardView** realiza las operaciones frontend de más bajo nivel, orientadas a pintar el tablero, mostrar los mensajes de error y los títulos durante la evolución de la partida.

El **PlayerView** interactúa con el **GameView** para mover las piezas, verifica el turno e invoca al backend para realizar los movimientos de la CPU.

Por último, el módulo **main** se relaciona con las vistas y el Modelo de Objeto del documento (DOM) definiendo los elementos, como las casillas y los botones, con los que el usuario puede interactuar. Cuando el usuario realiza una acción en la interfaz el elemento accionado dirige la llamada a la vista correspondiente.

A continuación se muestra el diagrama del Diagrama de clases del frontend.



Patrones creacionales

Los patrones creacionales abstraen el proceso de creación de las instancias, ayudando a que un sistema sea independiente de cómo se crean, se componen o se representan sus objetos. Este tipo de patrones se vuelve más importante a medida que los sistemas evolucionan para depender más de la composición de objetos que de la herencia de clases (Gamma et al., 1995).

Singleton

El patrón Singleton se basa en asegurar que una clase contiene una única instancia y provee un único punto de acceso global a ella (Gamma et al., 1995).

En la implementación clásica se define mediante una clase con un método estático de acceso público que retorna la única instancia de la clase. La primera vez que se invoca este método se crea el objeto mediante el uso del operador *new*, mientras que en el resto de ocasiones se devuelve la instancia previamente creada. El constructor de la clase es privado siendo únicamente posible el acceso a la instancia a través del método estático descrito, la estructura del patrón puede observarse en el siguiente fragmento:

```
public Class() {
   private static Class instance;

   private Class() {
       // Private constructor
   }

   public static Class getInstance() {
       if(instance == null) {
           instance = new Class();
       }
       return instance;
   }
}
```

JavaScript permite una aproximación a esta estructura evitando el uso de variables estáticas y del método *getInstance()* como se muestra a continuación.

```
public Class() {
   constructor() {
            if (typeof Class.instance === "object") {
               return Class.instance;
           }
           Class.instance = this;
           return this;
   }
}
```

Existe un patrón ampliamente respaldado por la comunidad de JavaScript, conocido como el **Patrón del Módulo**. Este patrón es capaz de ofrecer una implementación eficaz, limpia y fundamentada en el uso de las closures, que puede utilizarse en combinación con el ***Singleton*** mejorando su eficacia o como una alternativa del propio patrón.

Patrón del Módulo cómo alternativa al Singleton

El **Patrón del Módulo** define una función que a su vez declara otras variables y funciones, permitiendo que el resto de estructuras del componente puedan acceder únicamente a un subconjunto de estas propiedades. Se sirve del ámbito de las **closures** para facilitar el desarrollo de su contexto privado y poder exportar las instancias únicas en lugar de las funciones de primer nivel. La función de primer nivel es declarada pero no exportada, aportando su entorno léxico durante la ejecución pero no permitiendo la creación de más instancias. La utilización de este patrón promueve la ocultación, produce objetos seguros, además de permitir una encapsulación eficaz de aplicaciones y singletons. (Crockford, 2008).

En el software se ha implementado el Patrón del Módulo cómo se indica a continuación: se ha definido una constante que tiene asignada la instancia de la closure. Esta constante es el único elemento exportado del módulo, por lo que permite el acceso público de la variable pero a su vez evita que puedan crearse otras instancias fuera del módulo.

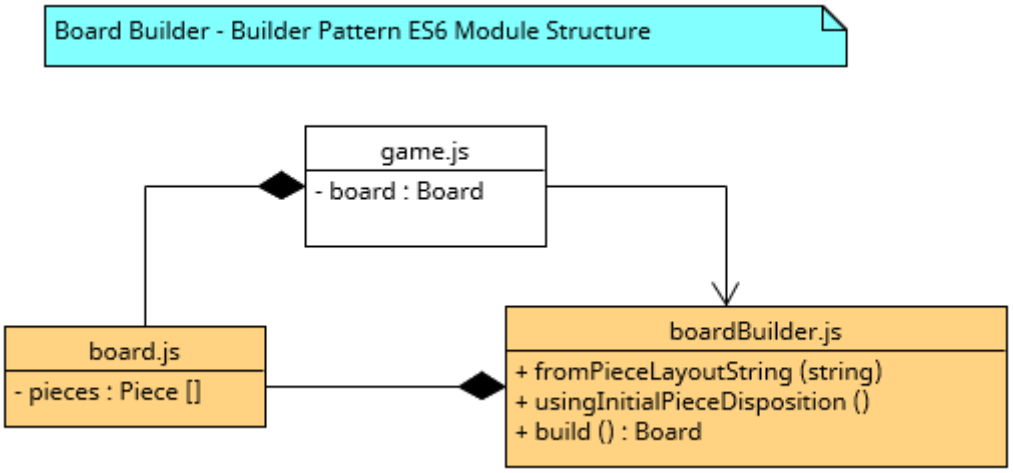
```
const singleInstance = constructor();

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

  function publicMethod2(errorMessage){
      // method 2
  }

  return {
      publicMethod1,
      publicMethod2
  }
}

export {
  singleInstance
}
```

El patrón se ha implementado en el motor de ajedrez para la declaración de los siguientes objetos: turn (fronted), boardView, restClient, messageManager y randomPlayer.

Builder

El patrón Builder separa la construcción de un objeto complejo de su representación, de tal forma que un mismo proceso de construcción puede crear representaciones distintas del objeto (Gamma et al., 1995).

En el software se ha aplicado el *Builder* para la construcción de la instancia de la clase **Board** mediante su constructor específico **BoardBuilder**. El constructor puede crear el tablero con la disposición de piezas al inicio de la partida o a partir de un string de inicialización con la distribución deseada. La responsabilidad de creación del tablero, **Board,** queda delegada a la clase **BoardBuilder**.



Patrones de comportamiento

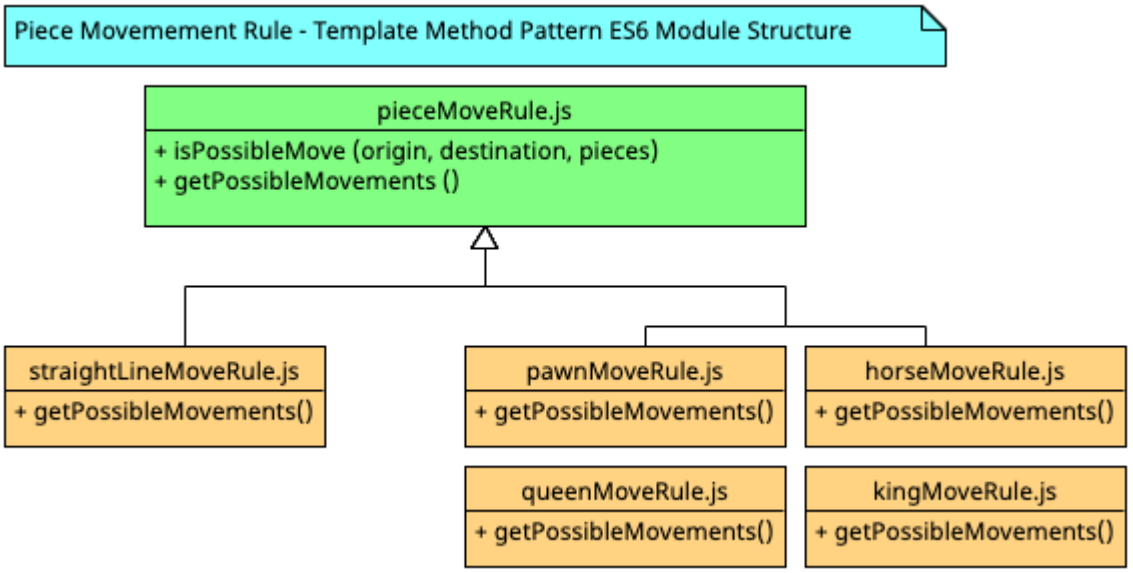
Los patrones de comportamiento se relacionan con los algoritmos y la asignación de responsabilidades entre objetos. Estos patrones no sólo describen objetos o clases, sino la comunicación existente entre ellos, permitiendo caracterizar un flujo de control completo (Gamma et al., 1995).

Template method

La intención del Patrón Template method es la creación del esqueleto de un algoritmo aplazando algunos pasos a sus subclases. Las clases derivadas redefinen estos pasos sin modificar la estructura general del algoritmo (Gamma et al., 1995).

Se ha aplicado el *Template method* en las reglas de movimiento que sigue cada tipo de pieza, los movimientos de cada pieza derivan de la clase **pieceMoveRule**. La clase padre contiene un método implementado **isPossibleMove()** a partir del método abstracto **getPossibleMovements()**. Cada clase derivada implementa este método abstracto devolviendo, en función de sus características, el listado de movimientos posibles.

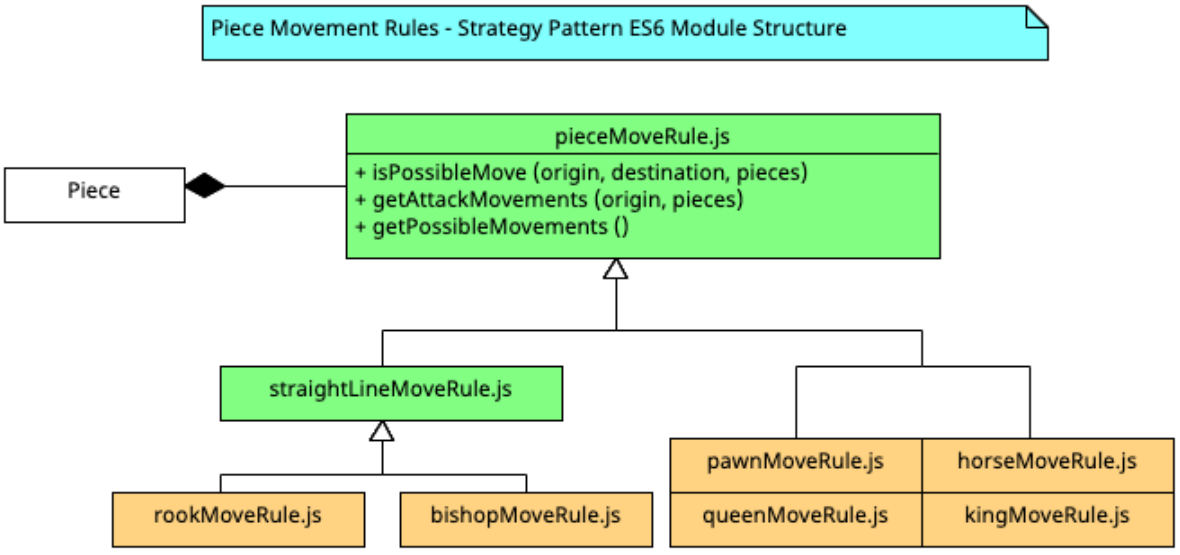
De esta manera a través del esqueleto común que proporciona el método **isPossibleMove(),** las clases hijas aportan su parte variable en la implementación del método **getPossibleMovements().**



Strategy

El patrón Strategy define una familia de algoritmos, encapsulando cada uno de ellos y haciéndolos intercambiables. Permite que estos varíen independientemente de los clientes que lo usen (Gamma et al., 1995). Presenta la siguiente estructura: separa cada uno de los algoritmos en una clase distinta, el cliente tendrá la referencia de cada estrategia y las podrá intercambiar en tiempo de ejecución.

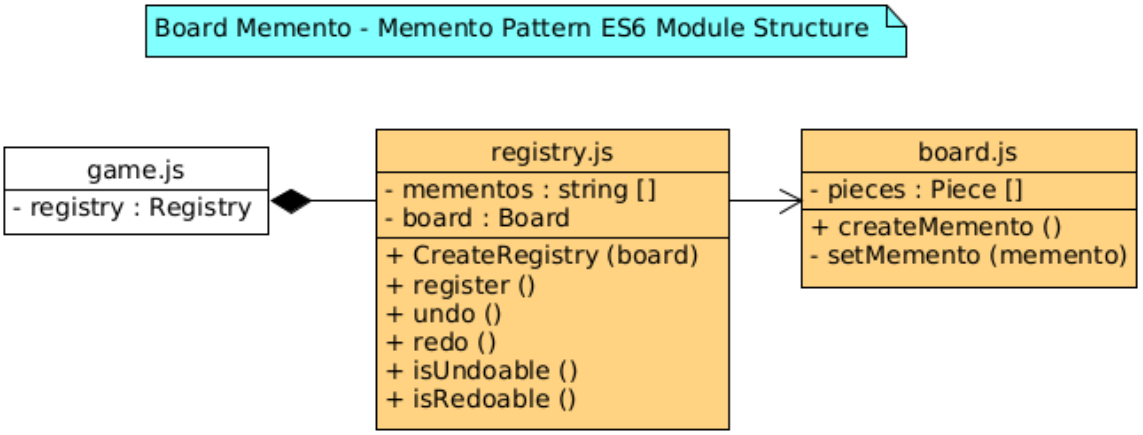
Se ha utilizado el *Strategy* para calcular los movimientos de cada pieza, dependiendo del tipo de pieza se asigna en tiempo de ejecución una regla de movimiento u otra. Esta versatilidad ha resultado muy útil a la hora de implementar la coronación del peón, ya que necesita cambiar de un pawnMoveRule a queenMoveRule en tiempo de ejecución.



Memento

El patrón Memento permite, sin violar la encapsulación, capturar y externalizar la información de un objeto para restaurar su estado más adelante (Gamma et al., 1995).

La funcionalidad de undo / redo y la simulación de movimientos en el cálculo de las posibilidades de jaque mate se han implmentado mediante el uso de este patrón. El módulo **registry** se encarga de almacenar en un array el estado del tablero en cada turno, así como de realizar undo / redo. Al final de cada movimiento, el tablero crea una copia de la disposición de las piezas.



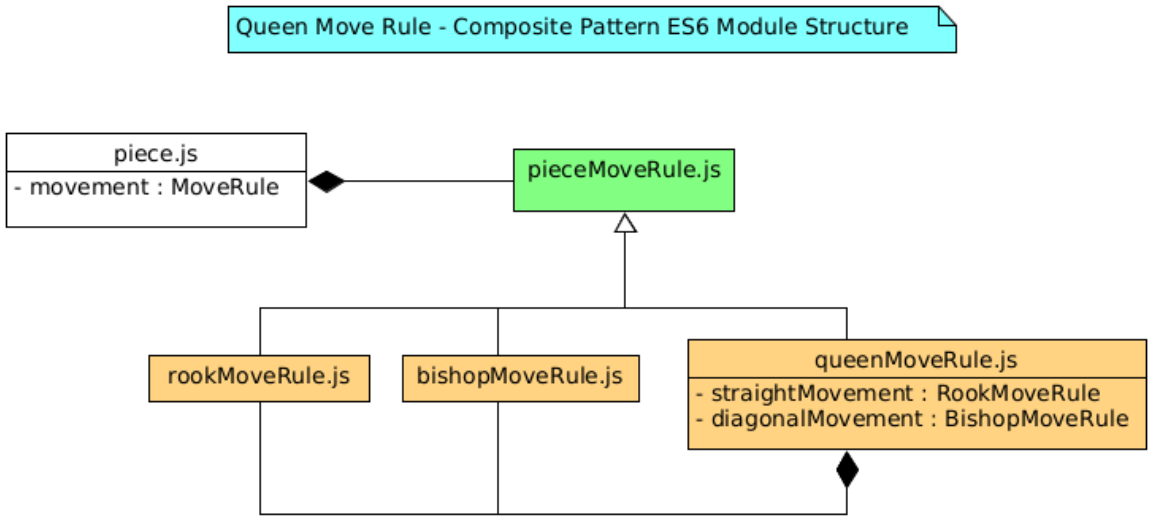
Patrones estructurales

Los patrones estructurales se encargan de la composición de clases y objetos para formar estructuras de mayor tamaño, apoyándose en la herencia. Estos patrones incorporan una flexibilidad añadida a la composición de objetos, derivada de la capacidad de cambiar la composición en tiempo de ejecución (Gamma et al., 1995).

Composite

El patrón Composite facilita la creación objetos con estructura de árbol, representado jerarquías de parte-todo. Este patrón permite a los clientes trabajar con estas estructuras como si fueran objetos individuales (Gamma et al., 1995).

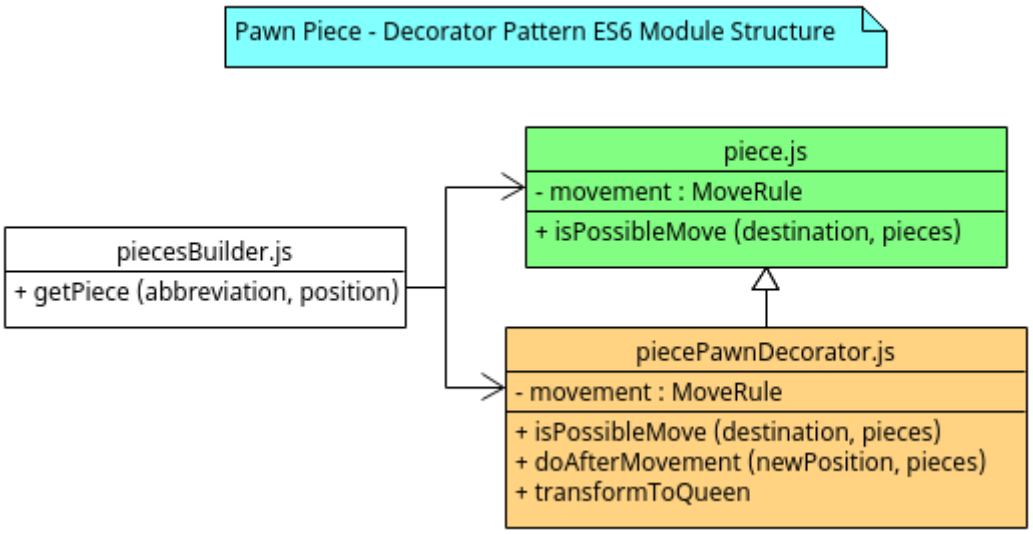
El patrón Composite se ha implementado en la regla de movimiento particular de la Reina, queenMoveRule. Esta estrategía se crea como un movimiento compuesto de la regla propia de la Torre rookMoveRule y del Alfil bishopMoveRule, manteniendo en la Reina el mismo tratamiento que en el resto de piezas.



Decorator

El patrón Decorator se basa en agregar dinámicamente responsabilidades a un objeto, proporcionando una alternativa flexible a la creación de subclases. Este patrón captura las relaciones de clase y objeto agregando una nueva responsabilidad (Gamma et al., 1995).

El Decorator se usó inicialmente para la coronación del peón, el objeto decorador piecePawnDecorator hereda de la regla de movimiento e implementa nuevos métodos. Esta transformación es realizada dinámicamente en tiempo de ejecución, sin embargo, cabe destacar que una vez realizado el cambio el peón no vuelve a su estado inicial. El motivo no es otro que las propias reglas del ajedrez.



La primera versión completa del motor de ajedrez incluyó este patrón, pero posteriormente fue retirado en favor de un método **getNextMoveRule** que permite a las estrategias devolver un 'sucesor' tras el movimiento realizado, simplificando el código y evitando el sobrediseño de la versión inicial.

III - Pruebas de software

Las pruebas de software se definen como el conjunto de procesos que pretenden validar y verificar la funcionalidad, así como asegurar la estabilidad de la aplicación durante su ciclo de vida. Pueden ejecutarse automáticamente en cualquier momento del desarrollo.

En esta sección nos centraremos en las **pruebas unitarias y en las estrategías de diseño de casos de prueba** que se han seguido para la realización del *testing* del backend, los test se encuentran bajo el directorio **/test**.

Pruebas Unitarias

Las **pruebas unitarias** (*Unit Testing*) son procesos automatizados que **verifican el funcionamiento de los componentes individuales de un sistema**, comprobando su funcionalidad de manera aislada y rápida (Badgett et al., 2011).

Los dobles son de gran utilidad en la elaboración de pruebas unitarias, permiten sustituir una dependencia por un objeto sobre el que se tiene control total de su comportamiento, facilitando el análisis entre las interacciones del *sujeto bajo pruebas (System under test - SUT)* y sus colaboradores (Khorikov, 2020).

Se ha desarrollado una batería de **pruebas unitarias** garantizando el cumplimiento de la funcionalidad esperada y reduciendo la posibilidad de errores en el ajedrez. El *testing* se realiza **bajo el marco del framework Jest**. Esta herramienta proporciona una biblioteca de afirmaciones utilizada para la comprobación de los valores esperados, el soporte de mocks y el propio entorno de ejecución de las pruebas.

Diseño de casos de prueba

Las metodologías de diseño de casos de prueba facilitan **el subconjunto de casos más eficaces**, pretenden ser lo más completo posible consiguiendo la mayor probabilidad de detectar el máximo número de errores existentes en el sistema. Persiguiendo este objetivo, las pruebas menos eficaces serían partir de valores aleatorios, mientras que una **estrategía razonable podría ser la combinación de pruebas orientadas de caja blanca y de caja negra** (Badgett et al., 2011).

Pruebas de caja blanca	Pruebas de caja negra
Selección de entradas al sistema a partir del análisis del código partiendo del conjunto de valores para ejecutar cada sentencia.	Selección de entradas y salidas al sistema partiendo de la experiencia previa y el conocimiento del dominio.
Cobertura de enunciados	Clases de equivalencias
Cobertura de decisiones	Análisis de valores límite
Cobertura de condiciones	Gráficos de causa-efecto
Cobertura de condiciones múltiples	Adivinación de errores

En esta tabla se resume la base de las pruebas de caja blanca y de caja negra, en cada una se presentan las metodologías de diseño que componen cada una de ellas.

El diseño de casos de prueba se ha llevado a cabo mediante la combinación de pruebas de caja blanca y de caja negra. Concretamente se ha tenido en cuenta **la cobertura de código, las clases de equivalencia y el análisis de los valores límite.**

En el resto del capítulo se describen las tres metodologías empleadas, mostrando algunos ejemplos de casos analizados en el *testing* del ajedrez, para obtener un mayor detalle de las casuísticas estudiadas se recomienda ver el **Anexo I - Diseño de casos de prueba.**

Cobertura

Las pruebas de caja blanca se basan **en el grado en que los casos ejercitan o cubren la lógica del sistema**, evaluando el conjunto de casuísticas que ejecutan al menos una vez cada sentencia de código y obteniendo los resultados posibles de cada decisión.

Se ha empleado la medida de **cobertura porcentual** del código fuente de la aplicación que ha sido ejercitado durante la ejecución del *testing*.

En la siguiente tabla se resume la cobertura de las declaraciones, ramas, funciones y líneas de código de los distintos componentes del backend, para obtener una mayor detalle se puede consultar el informe de cobertura en el siguiente enlace análisis de cobertura del backend.

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%**

Clases de equivalencia

Para cada entrada debe definirse **el subconjunto finito de valores** que el sistema **maneja de manera equivalente en el** ejercicio del sujeto bajo pruebas, **SUT**. La partición en clases de equivalencia permite suponer que el resultado obtenido para cualquiera de los valores de una misma clase implica un tratamiento similar.

Análisis de valores límite

Es común que **los errores de cualquier sistema se encuentren en los valores límite**, por lo que es impredecible analizarlos, de este modo se podría realizar una detección prematura de los errores de software.

El análisis de los valores límite con frecuencia es complementado con la partición en clases de equivalencia, escogiendo los valores frontera de cada subconjunto.

A continuación, se presentan algunos ejemplos de las casuísticas de diseño de pruebas del ajedrez apoyado en la técnica de análisis de valores límite y partición en clases de equivalencia:

*En la creación de la **coordenada** se diferencia dos grupos: **valores válidos y valores no válidos de un tablero de ajedrez**. Dentro de los valores válidos, se ha tenido en cuenta el análisis del límite, superior e inferior, tanto de la columna como de la fila. A partir de estas coordenadas se valida la funcionalidad de sus diferentes métodos.*

*Para el **movimiento de cada pieza** se ha considerado **todos los tipos de movimientos posibles**, por ejemplo, en la validación del peón se identifican dos tipos de comportamiento: el peón blanco avanza hacía el norte mientras que el peón negro avanza hacia el sur. Los test desarrollados en esta pieza se realizan por duplicado en cada color, sin embargo, en el resto de piezas no existe esta diferencia, por lo que en ellas se ha partido de una pieza de color blanco. Otro aspecto importante, es **la evaluación de los movimientos posibles en distintos escenarios,** estudiando el desplazamiento de cada pieza situada en los extremos del tablero: límite inferior, límite superior, límite derecho y límite izquierdo; así como en una posición intermedia.*

IV - Integración Continua y Entrega Continua

Se ha automatizado la etapa de integración del código entre el equipo y la fase de entrega, basándose en los procesos de **Integración Continua y Entrega Continua**. Con este fin se ha utilizado *GitHub Actions* implementando dos workflows.

GitHub Actions es un sistema ofrecido por *GitHub* para agilizar el desarrollo software, permitiendo automatizar el proceso de construcción, testeo y despliegue de una aplicación. Esta herramienta, se ha empleado para la ejecución de los tests y la generación de una imagen docker del motor de ajedrez.

Integración Continua

El desarrollo de software es una tarea con un componente social muy destacable, dado que habitualmente se realiza en equipo. Este proyecto se ha abordado de manera colaborativa compartiendo el código del motor de ajedrez en un repositorio central de *GitHub.* Ha sido primordial garantizar que el sistema siga funcionando correctamente en la integración de cada funcionalidad.

La Integración Continua *(Continuous Integration - CI)* **supone una solución para** un escenario cooperativo en el que se desea **garantizar el funcionamiento del sistema durante el desarrollo de software**. La CI consiste en integrar el código paulatinamente en lugar de hacerlo una sola vez por entrega, determinando que en cada incorporación de nueva funcionalidad en la línea principal se debe realizar una compilación y ejecución de los test asegurando la validez del software en cada versión.

En este proyecto para la aplicación de la CI se ha desarrollado un ***workflow***, su ejecución es previa a la integración del nuevo código con la rama principal del repositorio. La acción de abrir un *pull requests* a la rama *main* dispara el *workflow* encargado de lanzar las pruebas de la aplicación bajo el comando *npm test.* Este proceso automático ayuda a garantizar la validación del código antes de fusionarlo con la línea principal del software.

Entrega Continua

La Entrega Continua *(Continuous Delivery - CD)* es conocida como el siguiente paso o la extensión de la Integración Continua, **su objetivo principal es conseguir que la publicación** de nueva funcionalidad o la actualización **del software se aplique de una manera ágil y fiable.**

La CD automatiza las etapas comprendidas entre la integración del código en el repositorio central y la entrega del sistema al usuario, verificando el código actualizado y generando un *artefacto* usado a posteriori en el despliegue del software en un entorno productivo. La aplicación de esta técnica se ha llevado a la práctica utilizando la herramienta ***GitHub Packages***.

GitHub Packages es una plataforma de alojamiento de software empaquetado que permite la publicación de artefactos de manera privada o pública, soportando numerosos sistemas de empaquetado: Maven, NPM, Docker, RubyGems o Nuget entre otros.

Es posible centralizar todo el proceso de CI / CD creando flujos *DevOps* end-to-end utilizando conjuntamente las herramientas de *GitHub Packages, GitHub Actions* y los *webhooks.*

Partiendo de este sistema se han implementado los dos *workflows*, el primero de ellos encomendado al proceso de CI y el segundo encargado de la generación y publicación de una imagen Docker ejecutable de la aplicación. Una vez que el nuevo desarrollo es validado mediante la ejecución de los tests, se genera una imagen etiquetada con la versión de software definida en el archivo *package.json*.

Dockerfile

Docker **automatiza la creación de imágenes** a partir de las instrucciones definidas en un archivo denominado *Dockerfile*. El *Dockerfile* no es más que un documento de texto con los comandos que un usuario podría utilizar en línea de comandos para el ensamblaje de la imagen.

En el juego del ajedrez se ha apostado por un *Dockerfile* utilizando la técnica *Multi-stage*, permitiendo aplicar más fácilmente los procesos relacionados con la optimización, lectura y mantenimiento de este fichero. Se han definido las siguientes instrucciones:

- Paso 1: Descarga de las dependencias bajo el comando *npm install*.
- Paso 2: Creación de la imagen final de la aplicación a partir de los módulos descargados en el paso anterior.

Conclusiones y trabajo a futuro

A modo de recapitulación, a continuación se listan los principales resultados obtenidos, producto del trabajo llevado a cabo:

1. Se ha realizado un estudio comparativo de las alternativas más prominentes de la orientación a objetos en JavaScript, elaborando una guía apta para la adaptación de un estilo a otro.
2. Se ha conseguido implementar algunos de los patrones de diseño clásicos de la *Gang of four* en Javascript, detallando la estructura utilizada en este lenguaje. Adicionalmente, se ha explorado una alternativa al patrón *Singleton*, conocida como Patrón del Módulo.
3. Se ha elaborado una batería de test unitarios fundamentada en las metodologías de la disciplina de pruebas.
4. Se han desarrollado dos workflows con el fin de automatizar los procesos de Integración Continua y Entrega Continua.

Se puede observar, contrastando los objetivos secundarios presentados en la Introducción y objetivos que existe una relación uno a uno entre ellos y los resultados obtenidos, asegurando así el cumplimiento de las metas propuestas.

De este análisis se deriva que se ha alcanzado el objetivo principal del proyecto. Se entrega un producto de **software desarrollado en NodeJS con un estilo** de codificación unificado **basado en closures aplicando los patrones clásicos de la programación orientada a objetos (POO),** tales como: *Builder*, *Template method*, *Strategy*, *Memento* y *Composite*. Incluyendo el patrón *Decorator* utilizado en una versión preliminar y posteriormente descartado debido al sobrediseño que suponía esta solución.

En base al trabajo realizado y los resultados conseguidos, se puede concluir que **el uso de clases en la POO,** aunque está justificada y es ampliamente utilizada en distintos lenguajes de programación, **no es imprescindible en JavaScript para la creación de piezas de software de calidad, alta legibilidad y buen diseño**, como se ha demostrado concretamente con el uso de las closures. Las closures - respaldadas por una parte de la comunidad, aunque cada vez menos utilizadas desde la inclusión de la clase en el lenguaje - **son una alternativa natural y efectiva, capaz de adaptarse a los patrones de diseño clásicos de la *Gang of Four*** y, por tanto, de beneficiarse de todas sus ventajas.

Trabajo a futuro

Durante el desarrollo del ajedrez se han identificado algunos aspectos de la aplicación que admiten evoluciones y pueden ser potencialmente aprovechados como líneas de trabajo futuros. Se enfoca la implementación de patrones adicionales o la incorporación de nueva funcionalidad complementaria.

Las líneas de trabajo a futuro detectadas de manera preliminar son:

1. Incorporación de un modo multijugador. El backend desarrollado permite fácilmente este evolutivo, el cual se considera que podría beneficiarse del uso del patrón *State.*
2. Nuevo enfoque del registro de movimientos del juego. En una futura versión, se puede abordar el movimiento mediante el uso del patrón *Commad*, incluyendo nueva funcionalidad complementaria: listar los movimientos posibles o mostrar las fichas afectadas otorgando una sinergia de interés con el patrón *Memento* ya implementado en la versión actual.
3. Agregar grados de habilidad al jugador CPU, pues la versión actual simplemente actúa de manera rudimentaria realizando movimientos aleatorios. Es también una posibilidad interesante permitir al usuario escoger al inicio de la partida el color de las fichas, en lugar de utilizar por defecto las fichas blancas.

Bibliografía

References

Abadi, M., & Cardelli, L. (1996). *A theory of objects*. Springer New York.

Badgett, T., Myers, G. J., & Sandler, C. (2011). *The Art of Software Testing*. Wiley.

Casciaro, M. (2014). *Node.js Design Patterns*. Packt Publishing.

Classes - JavaScript | MDN. (2022, October 30). MDN Web Docs. Retrieved November 7, 2022, from https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes

Closures - JavaScript | MDN. (2022, September 14). MDN Web Docs. Retrieved October 9, 2022, from https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures

Closures - JavaScript | MDN. (2022, October 30). MDN Web Docs. Retrieved November 7, 2022, from https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures

Crockford, D. (2008). *JavaScript : the good parts*. O'Reilly.

Dockerfile reference. (n.d.). Docker Documentation. Retrieved November 21, 2022, from https://docs.docker.com/engine/reference/builder/

Du Preez, O. J. (2020). *JavaScript for Gurus: Use JavaScript programming features, techniques and modules to solve everyday problems*. BPB Publications.

Ellis, A. (n.d.). *Multi-stage builds*. Docker Documentation. Retrieved November 21, 2022, from https://docs.docker.com/build/building/multi-stage/

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns*. Addison-Wesley.

Hallie, L., & Osmani, A. (2020). *Learning Patterns*. Patterns.dev. https://www.patterns.dev/book/

How Inheritance works in Constructor Functions in JavaScript. (2022, 2 15). GeeksForGeeks. Retrieved 11 7, 2022, from https://www.geeksforgeeks.org/how-inheritance-works-in-constructor-functions-in-javascript/

JavaScript Constructors. (n.d.). W3Schools. Retrieved November 7, 2022, from https://www.w3schools.com/JS/js_object_constructors.asp

JavaScript Factory Functions. (2020, 3 26). JavaScript Tutorial. Retrieved 11 7, 2022, from https://www.javascripttutorial.net/javascript-factory-functions

JavaScript y Python mantienen su reinado en la programación ante el rápido crecimiento de Rust. (2022, May 23). *Europa Press*.

https://www.europapress.es/portaltic/software/noticia-javascript-python-mantienen-reinado-programacion-rapido-crecimiento-rust-20220523150208.html

Johnson, R. E., & Foote, B. (1988, June/July). Designing Reusable Classes. *Journal of Object-Oriented Programming*, *1*(2), 22-35. https://www.researchgate.net/publication/215446177_Designing_Reusable_Classes

Khorikov, V. (2020). *Unit Testing Principles, Practices, and Patterns: Effective Testing Styles, Patterns, and Reliable Automation for Unit Testing, Mocking, and Integration Testing with Examples in C#*. Manning.

Learn GitHub Packages. (n.d.). GitHub Docs. Retrieved November 21, 2022, from https://docs.github.com/en/packages/learn-github-packages

Mur, J. (2019, February 24). *Singleton Pattern con Javascript. En mi artículo anterior ya os hable de… | by Jesús Mur Fontanals*. Medium. Retrieved November 12, 2022, from https://medium.com/@jesusmurfontanals/singleton-pattern-con-javascript-3eb1c03f184e

Rehkopf, M. (n.d.). *¿En qué consiste la integración continua?* Atlassian. Retrieved November 21, 2022, from https://www.atlassian.com/es/continuous-delivery/continuous-integration

Stefanov, S. (2010). *JavaScript Patterns*. O'Reilly Media, Incorporated.

What are factory functions in JavaScript ? (2021, 6 24). GeeksForGeeks. Retrieved 11 7, 2022, from https://www.geeksforgeeks.org/what-are-factory-functions-in-javascript

Anexo I - Diseño de casos de prueba

Reglas de movimiento - *Move Rule*

Modelo	Grupo de pruebas	Descripción	Métodos validados	Casos de Uso
Coordenada	Acceso / modificación	Se analizan: métodos de acceso y modificación de atributos, métodos de obtención del número / letra de la columna o fila en una posición concreta.	setPosition getPosition getColumn getColumnLetter getRow	Se verifican en coordenadas creadas con valores válidos (límites e intermedios) y valores no válidos.
	Cálculo de la siguiente coordenada	Se analiza el cálculo de la siguiente coordenada.	getNextCoordinate	Se verifica en las 8 direcciones de movimiento con valores válidos del tablero (límites e intermedios).
Dirección	Cálculo de la siguiente coordenada	Se analiza el cálculo de la siguiente coordenada.	getNextCoordinate getRow getColumn	Se verifica en las 8 direcciones de movimiento con valores válidos del tablero. Se hace uso de los métodos para obtener la columna y la fila.
Regla de movimiento	Acceso / modificación	Se analiza la actualización / obtención de la posición actual y piezas del tablero.	updateCurrentPosition getCurrentPostion getBoardPieces	Se verifica partiendo de un tablero definido en el test sus valores válidos (límites e intermedios).
	Identificación de la coordenada	Se analiza la identificación de una coordenada vacia o de color opuesto.	isEmptyCoordinate isOpposingColor	Se verifica la identificación partiendo de un tablero definido en el test.
	Siguiente regla de movimiento	Se analiza la obtención de la siguiente regla.	getNextMoveRule	Se verifica obtener ella misma como siguiente regla de movimiento.
Regla de movimiento del peón	Obtención de movimientos posibles	Se analiza la obtención de los movimientos posibles de una pieza.	getPossibleMovements	Se verifican los movimientos de ataque y avance para un peón (blanco o negro) en distintos escenarios: situado en posiciones límite, rodeado de aliados o de enemigos.
	Validación de un movimiento	Se analiza la comprobación de un movimiento.	isPossibleMove	Se verifican en los escenarios anteriores las posiciones válidas / no válidas de desplazamiento.
	Validación de ataque	Se analiza la obtención de movimientos de ataque.	getAttackMovements	Se verifican rodeado de aliados / enemigos las posiciones válidas / no válidas de ataque.
	Siguiente regla de movimiento	Se analiza la obtención de la siguiente regla.	getNextMoveRule	Se verifica en una posición intermedia y al final del tablero.

Modelo	Grupo de pruebas	Descripción	Métodos validados	Casos de Uso
Regla de movimiento: rey, caballo y lineal	Obtención de movimientos posibles	Se analiza la obtención de los movimientos posibles de una pieza.	getPossibleMovements	Se verifican en cada tipo los movimientos en los escenarios: situado en medio, en el límite superior, inferior, derecho e izquierdo del tablero, rodeado de aliados / enemigos. En el caballo se estudia con aliados / enemigos en las posiciones de ataque.
	Validación de un movimiento	Se analiza la comprobación de un movimiento.	isPossibleMove	Se verifican en los escenarios anteriores las posiciones válidas / no válidas de desplazamiento.
	Validación de ataque	Se analiza la obtención de movimientos de ataque.	getAttackMovements	Se verifica en las casuísticas de aliados / enemigos las posiciones válidas / no válidas de ataque.
Regla de movimiento: reina, alfil y torre	Obtención de movimientos posibles	Se analiza la obtención de movimientos de ataque.	getPossibleMovements	Al evaluar el detalle de la superclase lineal, se verifican sólo los movimientos en las 8 direcciones de la reina y en las 4 de alfil y torre situado en medio del tablero.

Piezas - *Pieces*

Modelo	Grupo de pruebas	Descripción	Métodos validados	Casos de Uso
Pieza	Acceso / modificación	Se analiza los métodos de acceso y modificación de atributos.	getAbbreviation setAbbreviation getFullName setFullName getPosition setPosition getMovementRule setMovementRule	Se válida el acceso en los tres tipos de pieza (blanca, negra y vacía).
	Movimientos	Se analiza los métodos relacionados con el movimiento.	updateCurrentPosition isPossibleMove getPossibleMovements getAttackPositions getMovementError	Se validan los valores válidos / no válidos del método *isPossibleMove,* actualización de la posición, obtención de movimientos y errores.
	Identificación de la pieza	Se analiza la identificación de tipo de pieza (color / vacío).	isWhite, isOpposingColor, isOfColor, isEmpty	Se valida la identificación para los tres tipos de piezas (blanca, negra y vacía).
Constructor de piezas	Construir piezas	Se analiza la construcción de piezas a partir de una plantilla del tablero.	buildFromLayout	Se valida a partir de un tablero vacío y con cada tipo de pieza.
	Obtener pieza	Se analiza la construcción de piezas a partir de su abreviatura.	getPiece	Se valida para las piezas: vacía, peón blanco y peón negro.

Juego - *Game*

Modelo	Grupo de pruebas	Descripción	Métodos validados	Casos de uso
Juego	Acceso	Se analiza los métodos de acceso a atributos.	getBoard getUuid	Se valida el acceso a atributos del juego: tablero y uuid.
	Undo / Redo	Se analiza los métodos de undo / redo.	undo redo	Se valida la interacción de undo / redo con el registro.
	Movimiento aleatorio del jugador CPU	Se analiza el método de movimiento del jugador aleatorio.	getRandomMovement	Se valida la interacción con el jugador aleatorio para la realización de un movimiento.
	Jugar	Se analiza la lógica del juego.	play	Se valida la lógica del juego en distintas situaciones: movimiento, en un turno incorrecto, sin piezas en el tablero, finalizando (en tablas o en jaque mate) e intentando un nuevo movimiento en una partida finalizada.
	Estado del juego	Se analiza la identificación del estado y evaluación de juego finalizado.	isGameFinished getStatus	Se valida la comprobación de juego acabado en los dos estados posibles y la obtención del estado del juego (finalizado y en marcha).
	Respuesta del tablero	Se analiza la interacción con la respuesta del tablero.	getBoardResponse	Se valida la interacción con la respuesta del tablero.
	Posibilidad de Undo / Redo	Se analiza el mensaje creado para informar de la posibilidad de undo / redo.	undoableRedoable	Se valida el mensaje creado para informar la posibilidad undo / redo: ambos ciertos, ambos falsos y las dos posibilidades de solo uno cierto.
Histórico del juego	Búsqueda de un juego	Se analiza el método de búsqueda de un juego por id.	findById save	Se valida la búsqueda de un juego en un histórico vacío y en un histórico con juegos. Se comprueba el guardado.

Jugador aleatorio - *Random player*

Modelo	Grupo de pruebas	Descripción	Métodos validados	Casos de uso
Jugador aleatorio (CPU)	Obtención de movimientos	Se analiza la realización de los movimientos aleatorios del jugador CPU.	getMovement	Se validan los movimientos en dos situaciones: con jaque y sin jaque. Se estudia con reintento y sin reintento en el valor de destino.

Tablero - *Board*

Modelo	Grupo de pruebas	Descripción	Métodos validados	Casos de uso
Tablero	Intentar un movimiento	Se analiza el intento de movimiento de una pieza dado una posición de origen, un destino y un color.	tryMove	Se validan los resultados de movimiento: mover una pieza vacía, mover una pieza negra en el turno de las blancas, mover una pieza al límite del tablero, mover una pieza atacando e intentar mover una pieza poniendo al rey en una situación de jaque.
	Situaciones de juego	Se analizan los métodos encargados de determinar el estado del juego en función del tablero.	isColorOnCheck isOnCheckMate isStalemate	Se validan tableros en diferentes estados: sin jaque, con jaque, con jaque mate y en tablas.
	Memento	Se analiza la realización de una copia del tablero en un estado determinado.	createMemento setMemento	Se valida la creación de copias del tablero y su recuperación en la instantánea del juego.
Constructor del tablero	Construcción del tablero	Se analiza la construcción de un tablero en base a un string layout determinado.	build	Se valida la construcción del tablero en base a un string layout definido previamente.

Registro - *Registry*

Modelo	Grupo de pruebas	Descripción	Métodos validados	Casos de uso
Registro	Registro	Se analiza el registro del tablero en un movimiento.	register	Se valida el primer registro del tablero.
	Undo / Redo	Se analizan los métodos de undo / redo.	undo redo	Se valida el undo y redo del movimiento cuando es posible y no es posible.
	Posibilidad de Undo / Redo	Se analiza la posibilidad de undo / redo.	isUndoable isRedoable	Se validan los estados (true / false) para la posibilidad de undo y redo.

Mensajes - *Message*

Modelo	Grupo de pruebas	Descripción	Métodos validados	Casos de uso
Mensaje	Creación de mensajes	Se analiza la creación de mensajes informativos	createMessage	Se valida la creación de un mensaje y el valor obtenido.
	Creación de mensajes de error	Se analiza la creación de mensajes de error	createErrorMessage	Se valida la creación de un mensaje de error y el valor obtenido.