Trabajo Final

Técnicas y Herramientas
UNLP
Magister en Ingeniería de Software

Profesores:
Gardey, Juan Cruz
Grigera, Julian
Urbieta, Matias

Integrantes:
Jurado, Abraham
Murdolo, Juan

Enunciado **1**
Introducción **6**
Resolución del ejercicio presentado **6**
Patrones **11**
Composite 11
Strategy 11
Factory Method 12
Testing **12**
jest.node.test 12
jest.block.test 13
jest.transactionComposite.test 14
jest.transactionHash.test 15
Conclusiones **16**
Referencias **18**

Enunciado

BlockChain

Una blockchain es un registro digital descentralizado y seguro que utiliza criptografía para proteger la información y garantizar su integridad. La información a intercambiar estará representada por tokens. Un token es simplemente un String con el patrón “TKN-<valor>” donde <valor> es un identificador con formato UUID. Un ejemplo de token puede ser TKN-6ec0bd7f-11c0-43da-975e-2a8ad9ebae0b.

Cada token es transferido de una dirección a otra mediante una transacción. Existe una transacción inicial llamada coinbase en donde un token es incorporado, y transacciones normales que representan el cambio de titular del token. En las transacciones normales se hace referencia a la última transacción del mismo token, y se guarda la dirección del nuevo titular. Cada transacción tiene un identificador con formato “Tx-<valor>”, y dos parámetros entre tres posibles:
IN: contiene el identificador de la última transacción que involucró al token. Solo las transacciones normales lo contienen
TKN: el id de token. Solo para las transacciones coinbase.
OUT: la dirección del nuevo dueño del token representada como “A-<valor>” donde <valor> es un identificador generado por el dueño de la transacción. Ambos tipos de transacción incluyen el parámetro OUT.
Además incluyen su hash (más detalles sobre esto luego). Por ejemplo:

Para asegurar la integridad y seguridad de la información registrada en una blockchain se utilizan funciones de hash. Una función hash es un algoritmo matemático que convierte datos de entrada de cualquier longitud en una cadena de caracteres de longitud fija. Esta cadena, conocida como hash, es única para los mismos datos de entrada y cualquier cambio en ellos resultará en un hash completamente diferente. En esta blockchain se deberá permitir intercambiar el mecanismo de generación y verificación de hash. Se deben poder utilizar al menos MD5 y SHA256 que son provista por la librería JSHashes.

Las transacciones se guardan en bloques, que almacenan hasta 10 transacciones. Cuando un bloque efectivamente llega a las 10 transacciones, éste se cierra, y ocurren 4 cosas:

1. Deja de admitir nuevas transacciones
    a. No permite cargar nuevas transacciones, requiriendo la creación de un nuevo bloque para ello.
2. Se completa su información y hash.
    a. Se guarda un timestamp (formato epoch) del momento de cierre, y se calcula y almacena su hash. El hash de un bloque se obtiene a partir de todos sus atributos, es decir timestamp, lista de todas sus transacciones y hash del bloque anterior:
    b. Hash(timestamp, prev_block_hash, Hash(tx1), ... , Hash(tx10))
    c. Además, cada transacción también computa y almacena su hash, lo que hace que sea muy difícil de manipular o falsificar la información en la cadena. Cualquier alteración de los datos de la transacción cambiará el hash, lo que a su vez invalidará el bloque y la cadena en la que está registrado.

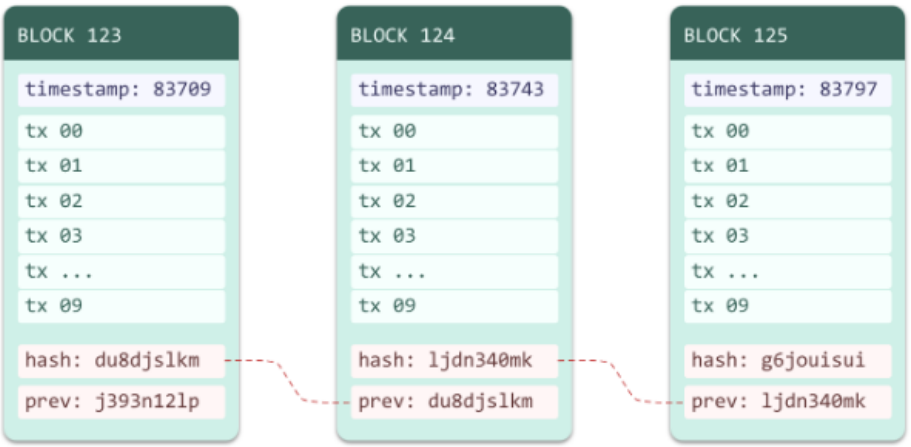


Fig 2. Ejemplo de cadena de 3 bloques.

Fig 2. Ejemplo de cadena de 3 bloques.

3. Se agrega el hash del bloque anterior
    a. Esto forma efectivamente la cadena (de ahí el nombre blockchain)
4. Se realiza el broadcast
    a. El nuevo bloque es replicado a través de diferentes nodos (ver siguiente párrafo).

Los bloques son administrados por nodos. Un nodo está en contacto con otros nodos, y todos los nodos guardan una copia de la misma blockchain (y guardan una única blockchain). Desde cualquiera de estos nodos se pueden registrar nuevas transacciones, que son registradas en un bloque solo si es correcta (es decir, si su hash es correcto).

Tener en cuenta que la blockchain solo se compone de bloques cerrados, con lo cual las transacciones que recibe cada nodo se van guardando en un bloque abierto, que aún no forma parte de la blockchain. Cada nodo mantendrá este bloque abierto (diferente al del resto de los nodos), hasta completarlo y cerrarlo. En ese punto se realizan las 4 acciones de cierre de bloque, en particular se agrega a la blockchain local y se realiza el broadcasting, es decir, se envía el bloque recién cerrado a los nodos conocidos para que todos lo agreguen a su copia local de la blockchain - validando también su integridad.

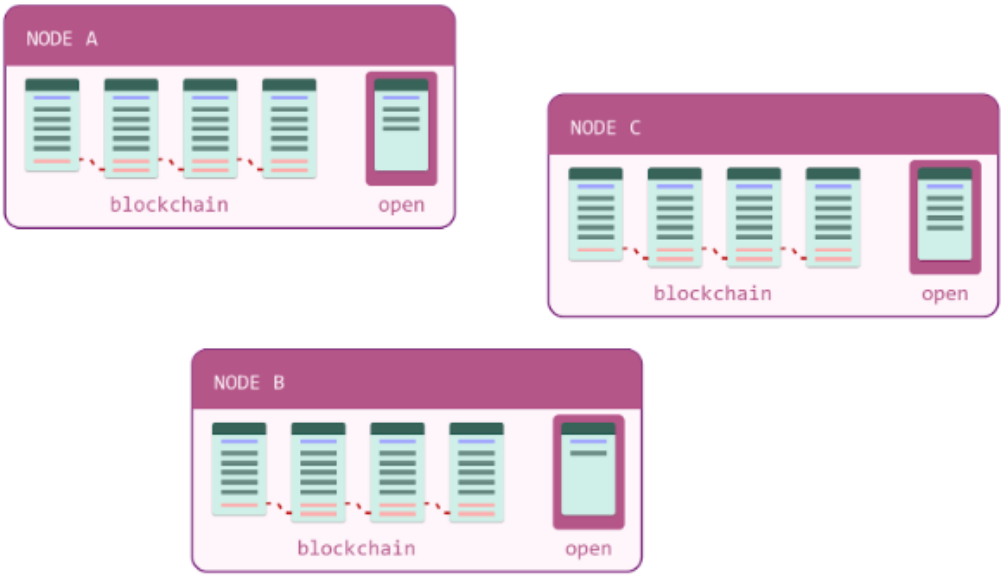


Fig 3. Ejemplo de estructura de 3 nodos, cada uno con su copia de la blockchain y un bloque abierto para agregar transacciones.

Para sortear la limitación de las 10 transacciones por bloque, se decidió permitir contar con transacciones compuestas. Este tipo de transacción permite agrupar hasta 3 transacciones compuestas, y hasta obtener una altura de 2. De esta forma, en el lugar de una transacción podemos tener hasta 9 transacciones simples (la mencionada anteriormente). Un ejemplo de transacción compuesta sería:

- Transacción Compuesta:
  - Transacción Compuesta:
    - Transacción Simple
    - Transacción Simple
    - Transacción Simple
  - Transacción Compuesta:
    - Transacción Simple
    - Transacción Simple
    - Transacción Simple
  - Transacción Simple

Aclaraciones

- Evitar el uso de expresiones lambda o funciones anónimas, salvo para la API de colecciones.
- Si se utilizan patrones de diseño para la solución, documentarlos

Objetivos

- Realizar un diagrama de clases con la solución
- Implementar la solución en NodeJS utilizando el paradigma de orientación a objetos. Se debe poder:

    - **Crear nuevos nodos**, pero no hace falta modelar la incorporación de nuevos nodos a una blockchain existente
    - **Agregar transacciones**, validando su integridad
    - **Establecer y alterar el mecanismo de hashing**
    - **Cerrar bloques** con todo lo que esto conlleva, lo que debe desencadenarse automáticamente al agregar la tx que excede el límite (11)
- Implementar tests de unidad para probar la solución con una cobertura de 80% del código.

Aclaraciones

2/6/2023

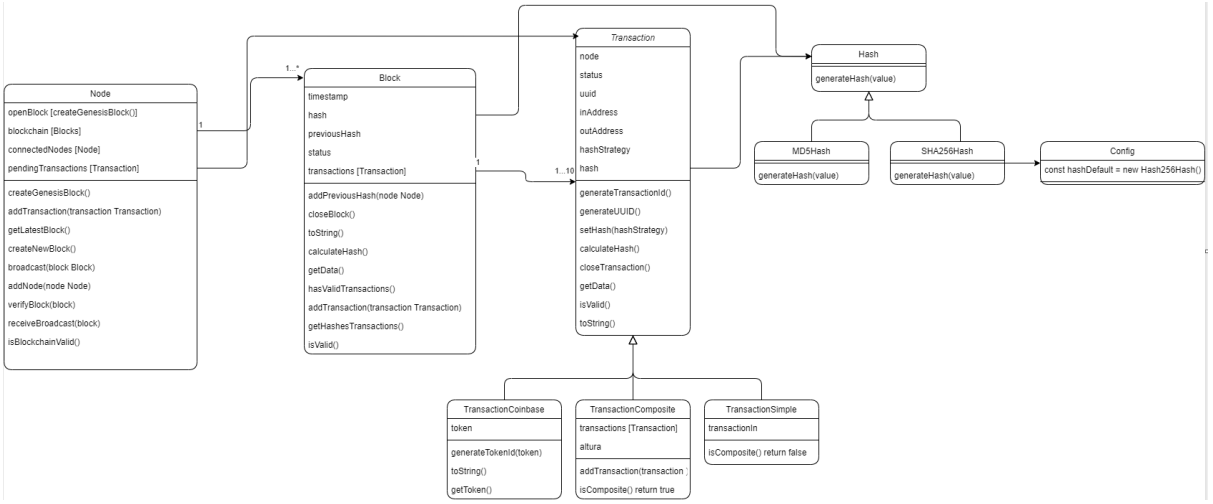
- Cuando se crean los coinbase se asignan a un dueño de forma arbitraría.
- Puede ocurrir doble gasto. Este fenómeno ocurre cuando una transacción se registra en dos nodos para un mismo token. No es necesario contrarrestarlo.
- Los primeros bloques podrían tener varios coinbase para popular la blockchain
- Para el hashing flexible, podrían optar por dos implementaciones.
    - Estático donde se indica el algoritmo cuando se inicializan los nodos y el algoritmo no se puede modificar en runtime.
    - Dinámica donde cada bloque puede ser estampado con un hash específico. Para ello, se deberá indicar en el bloque cuál algoritmo fue utilizado. Este hash se puede calcular únicamente sobre los bloques abiertos que se encuentran en construcción solamente. No se debe permitir modificar el algoritmo utilizado en bloques cerrados.
- No se requiere incorporar el token a la transacción. Para crear nuevas transacciones se deberá contar con la última transacción del token.
- Cada nodo deberá verificar cada bloque recibido confirmando que el hash del bloque es correcto.

29/6/2023

Introducción

Resolución del ejercicio presentado

En nuestro diagrama de clases se puede notar que



- **Node** representa un nodo en una red de blockchain. Cada nodo tiene una copia local de la cadena de bloques y participa en la validación y creación de nuevos bloques.

    - ***Atributos***
        - **blocks**: Es un array que contiene los bloques abiertos, que aún están en proceso de construcción y pueden recibir nuevas transacciones.
        - **blockchain**: Es un array que contiene todos los bloques cerrados que han sido añadidos a la cadena de bloques del nodo.
        - **connectedNodes**: Es un array que contiene referencias a otros nodos conectados a este nodo en la blockchain.

    - ***Métodos***
        - tiene un **constructor**(): que se ejecuta al crear una nueva instancia de Node.
        - **createGenesisBlock**(): Crea y devuelve un bloque génesis, que es el primer bloque de la cadena. El bloque génesis no contiene transacciones y su hash anterior se establece en '0'.
        - **addTransaction(transaction)**: Agrega una nueva transacción al bloque abierto actual, siempre y cuando la transacción sea válida (según su método isValid()). Si el bloque abierto ya contiene 10 transacciones, se cierra el bloque y se crea uno nuevo para agregar la nueva transacción, después hace el broadcasting enviando el bloque cerrado a los demás nodos.
        - **getLatestBlock**(): Devuelve el bloque abierto más reciente (el último en el array openBlock).

        - **createNewBlock**(): Crea un nuevo bloque y lo agrega al array openBlock. Este nuevo bloque se convierte en el bloque abierto actual.
        - **broadcast(block)**: Envia el bloque cerrado a cada nodo conectado.
        - **addNode(node):** Agrega un nodo a la lista de nodos conectados a este nodo.
        - **receiveBroadcast(block):** Recibe un bloque enviado por otro nodo y lo agrega a la cadena de bloques local si pasa las verificaciones de validez (utilizando el método verifyBlock()).
        - **verifyBlock(block):** Verifica si un bloque es válido, comprobando su integridad.
        - **addBlock(block)**: Agrega un bloque a la lista de bloques en la cadena de bloques local.
        - **isBlockchainValid()**: Verifica la validez de la blockchain local del nodo. Comprueba que todos los bloques tengan transacciones válidas, que los hashes de los bloques sean correctos y que los bloques estén vinculados correctamente a través de sus hashes previos.

- La clase **Block** representa un bloque en una cadena de bloques (blockchain) de una criptomoneda o una aplicación descentralizada que utiliza tecnología blockchain

    - ***Atributos:***
        - **timestamp**: Representa la marca de tiempo de cuando se creo el bloque.
        - **transactions**: Es un array que contiene todas las transacciones incluidas en el bloque.
        - **hash**: Representa el hash único del bloque actual. El hash se calcula en función de los datos del bloque, como su marca de tiempo, las transacciones que contiene y el hash del bloque anterior.
        - **previousHash**: Es el hash del bloque anterior en la cadena de bloques. Se utiliza para establecer la conexión entre los bloques y mantener la integridad de la cadena.
        - **status**: Representa el estado del bloque. Puede ser "open" si el bloque está en proceso de construcción o "closed" si ya se ha terminado de construir y no se permite agregar más transacciones.
        -
    - ***Métodos***:
        - un **constructor**(timestamp, transactions = [], previousHash = ""): Es el constructor de la clase, utilizado para inicializar un nuevo bloque. Recibe una marca de tiempo, un array de transacciones (que es opcional) y el hash del bloque anterior (también opcional). Al construir un bloque, también crea y agrega una transacción Coinbase a las transacciones del bloque.
        - **addPreviousHash(node):** Es un método que agrega el hash del bloque anterior. Recibe un objeto node, que se supone que tiene una propiedad blockchain que contiene los bloques anteriores. Busca el

último bloque cerrado en la cadena y agrega su hash como el previousHash del bloque actual.
        - **closeBlock():** Este método cierra el bloque actual. Marca todas las transacciones pendientes en el bloque como cerradas, calcula el hash del bloque y cambia el estado del bloque a "closed".
        - **toString():** Devuelve una representación JSON del bloque.
        - **calculateHash():** Calcula el hash del bloque actual utilizando una función de generación de hash proporcionada en la configuración (config.hashDefault.generateHash).
        - **getData()**: Devuelve una cadena que combina el hash del bloque anterior, la marca de tiempo del bloque actual y los hashes de todas las transacciones incluidas en el bloque.
        - **isValid():** Verifica la validez del bloque. Calcula el hash del bloque y lo compara con el hash almacenado en la propiedad hash.
        - **hasValidTransactions():** Verifica la validez de todas las transacciones incluidas en el bloque. Itera a través de las transacciones y verifica que cada una sea válida utilizando su método isValid().
        - **addTransaction(transaction):** Agrega una nueva transacción al bloque, siempre y cuando la transacción sea válida (según su método isValid()). Si la transacción no es válida, lanza un error.
        - **getHashesTransactions():** Devuelve una cadena que contiene todos los hashes de las transacciones incluidas en el bloque, concatenados uno tras otro.

- **Transaction** representa una transacción en una blockchain. Una transacción es una operación que involucra la transferencia de activos o datos entre dos direcciones (cuentas). En el contexto de una blockchain, una transacción se agrega a un bloque y luego se valida y agrega a la cadena de bloques.

    - ***Atributos***
        - inAddress: Dirección de la cuenta del remitente (origen) de la transacción.
        - outAddress: Dirección de la cuenta del destinatario (destino) de la transacción.
        - hashStrategy: Objeto que define la estrategia de hash que se utilizará para calcular el hash de la transacción. Puede ser un objeto de las clases MD5Hash o SHA256, dependiendo de lo que se haya requerido en el módulo Config.
        - node: Representa el nodo que emitió la transacción.

    - ***Métodos***
        - Un constructor
        - **generateTransactionId(), generateUUID():** ambos sirven para generar un identificador único para la transacción. La transacción tiene un identificador único compuesto por el prefijo "Tx-" y un UUID (identificador único universal) generado.

        - **setHash(hashStrategy):** Este método se utiliza para establecer una nueva estrategia de hash para la transacción. Puede cambiar la estrategia de hash después de la creación de la transacción.
        - **calculateHash():** Este método se utiliza para calcular el hash de la transacción. Utiliza la estrategia de hash configurada previamente (por defecto, SHA256 o MD5Hash) y aplica el algoritmo de hash a los datos de la transacción.
        - **closeTransaction():** Este método se utiliza para marcar la transacción como "cerrada". Cambia el estado de la transacción a "closed".
        - **isComposite():** Un método para verificar si la transacción es de tipo compuesto.
        - **getData():** Este método devuelve una cadena que representa los datos relevantes de la transacción que se utilizarán para calcular el hash. Combina la información del nodo, la dirección de origen (inAddress) y la dirección de destino (outAddress).
        - **isValid():** Este método se utiliza para verificar si la transacción es válida. Calcula el hash actual de los datos de la transacción y compara si coincide con el hash almacenado en la transacción (this.hash).
        - **toString():** Este método convierte la transacción en una cadena JSON para facilitar su visualización y depuración.

- **TransactionCoinbase** es una subclase de Transaction. Representa una transacción especial. En las blockchains basadas en Proof of Work (PoW), como Bitcoin, la transacción coinbase es la primera transacción en un bloque y es utilizada para recompensar al minero que ha resuelto el bloque. En otras palabras, es la transacción que crea nuevos tokens (monedas) y las envía al minero que agregó el bloque a la cadena.
-
    - ***Atributos***
        - los mismos que su superclase
        - Más el atributo token

    - ***Métodos***
        - un **Constructor** que utiliza el super para los atributos de su clase padre y token que se autogenera para términos prácticos.
        - **generateTokenId(token):** Este método se utiliza para generar un identificador único para la transacción coinbase. Concatena el token con un UUID generado utilizando el método generateUUID().
        - **getToken():** Este método devuelve el token (moneda) utilizado en la transacción coinbase. En el código, devuelve el valor 'TKN', lo que representa un token ficticio o símbolo específico para esta blockchain.

- **TransactionComposite** también es una subclase de la clase Transaction, y representa una transacción compuesta o transacción jerárquica en una blockchain. Una transacción compuesta es aquella que contiene otras transacciones como sus elementos. Esto permite la creación de estructuras jerárquicas en la blockchain,

donde una transacción compuesta puede incluir otras transacciones, y estas a su vez pueden contener más transacciones, formando una estructura en árbol.

- ***Atributos***
  - Los mismos de su clase padre
  - **transactions:** Un array que almacenará las transacciones que forman parte de esta transacción compuesta. Estas transacciones pueden ser instancias de Transaction o TransactionComposite.
  - **altura:** Un valor numérico que indica el nivel de jerarquía de la transacción compuesta. El nivel 0 indica que la transacción no es compuesta, mientras que un valor mayor a 0 indica que contiene otras transacciones.

- ***Métodos***
  - **addTransaction(transaction):** Este método se utiliza para agregar una transacción a la transacción compuesta. Si la transacción compuesta ya contiene 3 transacciones, se lanzará un error indicando que no se pueden agregar más. Además, se verifica si la transacción que se va a agregar es una transacción compuesta y si su altura más 1 excede el valor 2. Si es así, se lanza un error indicando que la altura máxima de la jerarquía no puede superar 2. Luego, se agrega la transacción al array transactions y se actualiza el valor de altura si la transacción es compuesta y su altura más 1 es mayor que la actual. Estos errores pueden ser sobrepasados si en lugar de agregar una transaction normal se agrega otra composite.
  - **isComposite():** Este método indica que la transacción es compuesta, ya que siempre devuelve true. Es necesario para distinguir entre una transacción normal y una transacción compuesta.

- **TransactionSimple** también es una subclase de Transaction, y representa una transacción simple en una blockchain. Una transacción simple es una transacción individual que no contiene otras transacciones dentro de ella. Es el tipo más básico de transacción en una blockchain.
-
  - ***Atributos***
    - Los mismo que su clase padre
    - transactionIn: La clase TransactionSimple tiene una propiedad adicional llamada transactionIn, que representa la transacción de entrada (la transacción anterior) a la cual esta transacción simple está relacionada.

  - ***Métodos***
    - Un constructor(txIn, inAddress, outAddress, hashStrategy, node)
    - isComposite(): Este método indica que la transacción no es compuesta, ya que siempre devuelve false. Es necesario para distinguir entre una transacción normal y una transacción compuesta.

- **Hash** es una clase abstracta que sirve como base para definir la interfaz del cálculo de hashes en la blockchain.
    - ***Métodos***
        - generateHash(value), el cual está sin implementar lanzando un error indicando que se debe usar una clase que herede de Hash para implementar este método.

- **MD5Hash** clase hija de Hash teniendo implementado el método generateHash(value)
    - ***Métodos***
        - generateHash(value). Utiliza la biblioteca externa jshashes para calcular el hash MD5 del valor pasado como parámetro y devuelve el resultado en formato hexadecimal.

- **SHA256Hash** Esta clase también hereda de la clase Hash y proporciona una implementación concreta del método generateHash(value)
    - ***Métodos***
        - Utiliza la biblioteca externa jshashes, para calcular el hash SHA-256 del valor pasado como parámetro y devuelve el resultado en formato hexadecimal.
- **Config** funciona como un módulo que instancia la clase SHA265Hash y la utiliza como método de hash predeterminado.

Patrones

Composite

Composite es un patrón de diseño estructural que te permite componer objetos en estructuras de árbol y trabajar con esas estructuras como si fueran objetos individuales.

El patrón Composite es fundamental para la organización y manejo de las transacciones en una blockchain. Utilizando este patrón, hemos creado una jerarquía de clases de transacciones, donde cada tipo de transacción (Simple, Coinbase, Composite) puede ser tratado como un todo o como una parte de una transacción compuesta. Esto nos permite manejar transacciones individuales y transacciones complejas de manera uniforme, simplificando la lógica y mejorando la reutilización del código.

Strategy

El patrón de diseño Strategy se utiliza para definir una familia de algoritmos, encapsular cada uno de ellos y hacerlos intercambiables. En este caso, la clase Hash representa el contexto del patrón, mientras que las clases MD5Hash y SHA256Hash representan las estrategias concretas que implementan diferentes algoritmos de hash (MD5 y SHA-256, respectivamente). El uso de este patrón permite que las estrategias (algoritmos de hash)

sean intercambiables sin que el cliente (en este caso, la clase Block o la clase Transaction) necesite preocuparse por los detalles de implementación de cada algoritmo.

Factory Method

Usamos un enfoque parcial del patrón Factory Method para crear objetos de tipo Hash en la clase Config. Al proporcionar una estrategia de hashing predeterminada (SHA-256) y permitir que el usuario configure su propia estrategia, logramos una mayor modularidad y extensibilidad del sistema.

Testing

jest.node.test

Los tests definidos en este bloque prueban diferentes aspectos y funcionalidades de la clase Node. Cada test tiene una función específica y demuestra cómo se comporta la clase en diferentes situaciones. A continuación, se presenta un resumen de cada uno de los tests:

Test: Debe crear un bloque génesis al crear un nodo:
Verifica que al crear un nuevo nodo, se cree automáticamente un bloque génesis (el primer bloque de la blockchain) con las propiedades y valores adecuados.

Test: Debe agregar una transacción a un bloque abierto:
Verifica que al agregar una transacción al nodo, esta se agregue correctamente al bloque abierto actual. Además, verifica si el tamaño de transacciones del bloque aumenta en uno después de agregar una nueva transacción.

Test: Debe crear un nuevo bloque tras alcanzar 10 transacciones en el bloque abierto:
Verifica que cuando el bloque abierto tiene 10 transacciones, se crea automáticamente un nuevo bloque en la blockchain.

Test: Debe cerrar el último bloque y crear uno nuevo si ha alcanzado el número máximo de transacciones:
Verifica que cuando se alcanza el número máximo de transacciones en el bloque abierto (10), este se cierra automáticamente y se crea uno nuevo.

Test: Debe conectar un nuevo nodo a otros nodos:
Verifica que se puede conectar un nuevo nodo a otros nodos existentes y que el nodo conectado se agrega correctamente a la lista de nodos conectados del nodo original.

Test: Debe devolver true si el bloque tiene todas sus transacciones válidas:
Verifica que el método hasValidTransactions() de la clase Block devuelve true cuando todas las transacciones del bloque son válidas.

Test: Debe agregar una transacción válida al último bloque (versión 1 y 2):

Verifica que se puede agregar una transacción válida al último bloque abierto.

Test: Debe lanzar una excepción al agregar una transacción inválida:
Asegura que se lance una excepción si se intenta agregar una transacción inválida al bloque abierto.

Test: Debe devolver false si algún bloque tiene transacciones no válidas:
Verifica que el método isBlockchainValid() devuelve false si algún bloque en la blockchain tiene transacciones no válidas.

Test: Debe devolver false si algún bloque tiene un hash incorrecto:
Verifica que el método isBlockchainValid() devuelve false si algún bloque en la blockchain tiene un hash incorrecto.

Test: Debería devolver true si la blockchain es válida:
Verifica que el método isBlockchainValid() devuelve true si toda la blockchain es válida, es decir, todos los bloques tienen transacciones válidas y los hashes son correctos.

Test: Debe devolver falso si un bloque tiene transacciones no válidas:
Verifica que el método isBlockchainValid() devuelve false si algún bloque en la blockchain tiene transacciones no válidas.

Test: Debe devolver falso si un bloque tiene un previousHash incorrecto:
Verifica que el método isBlockchainValid() devuelve false si algún bloque en la blockchain tiene un previousHash incorrecto.

Test: Debe lanzar una excepción al recibir un bloque no válido:
Verifica que se lance una excepción si se intenta recibir un bloque no válido en el nodo.

Test: Los tres nodos deben tener los mismos bloques cerrados en su blockchain:
Verifica si tres nodos conectados tienen la misma cadena de bloques cerrados después de agregar transacciones aleatorias a cada nodo y sincronizarlos mediante la función de transmisión.

jest.block.test

Los tests definidos en este bloque prueban diferentes aspectos y funcionalidades de la clase Block. Cada test tiene una función específica y demuestra cómo se comporta la clase en diferentes situaciones. A continuación, se presenta un resumen de cada uno de los tests:

Test: Debe devolver la cadena JSON del bloque:
Verifica que el método toString() de la clase Block devuelve una cadena JSON que representa correctamente el bloque.

Test: Debe verificar si todas las transacciones en el bloque son válidas:
Verifica que el método hasValidTransactions() de la clase Block verifica si todas las transacciones en el bloque son válidas llamando al método isValid() de cada una de ellas.

En este test, se utilizan transacciones simuladas, una válida y otra no válida, para verificar el resultado de la validación.

Test: Debe agregar una transacción al bloque:
Verifica que el método addTransaction(transaction) de la clase Block agrega una transacción al bloque siempre que esta sea válida. Se utiliza una transacción simulada válida para realizar la prueba.

Test: Debe lazar una excepción al agregar una transacción inválida:
Verifica que la función addTransaction() lance una excepción cuando se intenta agregar una transacción inválida al bloque.

Test: Debe devolver el hash concatenado de todas las transacciones en el bloque:
Verifica que el método getHashesTransactions() de la clase Block devuelve el hash concatenado de todas las transacciones en el bloque. Se utilizan transacciones simuladas con hashes específicos para probar la concatenación.

Test: Debe devolver el valor hash correcto:
Verifica que el método calculateHash() de la clase Block calcula el hash correcto del bloque utilizando el hash de transacciones y el hash anterior. Se utiliza un objeto de ejemplo de Block y se mockea la función de generación de hash para verificar que se llame con los datos correctos.

Test: Debe cerrar el bloque correctamente:
Verifica que el método closeBlock() de la clase Block cierra el bloque correctamente. Se utilizan transacciones simuladas con diferentes estados (pendiente y cerrada) y se verifica que la función closeTransaction() se llame solo para la transacción pendiente y que el estado del bloque y el hash se actualicen correctamente.

Test: Debe devolver el hash anterior del último bloque cerrado:
Verifica que el método addPreviousHash(node) de la clase Block devuelve el hash del último bloque cerrado de la blockchain del nodo proporcionado. Se crea un objeto de ejemplo de la blockchain y se mockea la función de generación de hash para verificar que se llame con los datos correctos.

Test: Debe devolver una cadena vacía cuando el hash anterior no está definido o es nulo:
Verifica que el método addPreviousHash(node) de la clase Block devuelve una cadena vacía cuando el hash anterior del último bloque cerrado no está definido o es nulo. Se crea un objeto de ejemplo de la blockchain para simular estas condiciones.

jest.transactionComposite.test

Estos tests prueban diferentes aspectos y funcionalidades relacionadas con las clases de transacción y su uso en el patrón Composite. A continuación, se presenta un resumen de cada uno de los tests:

Test: Transaction debe generar un identificador de transacción:
Verifica que se genere correctamente un identificador de transacción que comience con el prefijo "Tx-".

Test: Transaction debe cerrar la transacción:
Verifica que al llamar al método closeTransaction() de la clase Transaction, el estado de la transacción se actualice correctamente a 'closed'.

Test: TransactionComposite debe agregar una transacción:
Verifica que se pueda agregar una transacción simple a la transacción compuesta y que esta quede almacenada correctamente en el array de transacciones.

Test: TransactionComposite debe arrojar un error si se intentan añadir más de 3 transacciones:
Verifica que al intentar agregar más de 3 transacciones a la transacción compuesta, devuelve un error con el mensaje 'No se pueden agregar más de 3 transactions'.

Test: TransactionComposite debe arrojar un error si se intenta añadir una transacción compuesta con una altura superior a 2:
Verifica que al intentar agregar una transacción compuesta a otra transacción compuesta con una altura superior a 2, se lance un error con el mensaje 'La altura máxima de la jerarquía no puede superar 2'.

Test: TransactionComposite debe actualizar la altura al añadir transacciones compuestas:
Verifica que la altura de la transacción compuesta se actualice correctamente al agregar transacciones compuestas a ella.

Test: TransactionComposite debe devolver true para una transacción compuesta:
Verifica que el método isComposite() de la clase TransactionComposite devuelva true, lo que indica que es una transacción compuesta.

Test: TransactionSimple debe establecer la propiedad txIn:
Verifica que al crear una instancia de la clase TransactionSimple, la propiedad transactionIn quede establecida correctamente.

jest.transactionHash.test

Test: Hash genera un error si se llama a generateHash directamente:
Este test verifica que al llamar directamente al método generateHash en la clase Hash, se genere un error con el mensaje 'Se debe usar una clase que herede de Hash'. Esto asegura que generateHash solo se utilice a través de sus clases derivadas (MD5Hash y SHA256Hash).

Test: debe establecer la estrategia hash:

En este test, se crea una instancia de TransactionSimple con la estrategia de hashing MD5Hash, y luego se cambia la estrategia de hashing a SHA256Hash utilizando el método setHash. Finalmente, se verifica que la estrategia se haya establecido correctamente, es decir, que la propiedad hashStrategy de TransactionSimple ahora apunte a la estrategia SHA256Hash.

Test: debe dar error si no se establece una estrategia hash:
En esta prueba, se crea una instancia de TransactionSimple con la estrategia de hashing MD5Hash. Luego, se intenta cambiar la estrategia a null utilizando el método setHash, y se verifica que se lance un error con el mensaje 'No se ha configurado una estrategia de hash'.

Test: MD5Hash debe devolver el hash MD5 correcto:
Este test verifica que el método generateHash() de MD5Hash genere el hash MD5 correcto para un dato dado. Se espera que el hash generado para el dato 'data' coincida con el valor esperado '8d777f385d3dfec8815d20f7496026dc'.

Test: SHA256Hash debe devolver el hash SHA256 correcto:
Este test verifica que el método generateHash() de SHA256Hash genere el hash SHA-256 correcto para un dato dado. Se espera que el hash generado para el dato 'data' coincida con el valor esperado '3a6eb0790f39ac87c94f3856b2dd2c5d110e6811602261a9a923d3bb23adc8b7'.

Test: TransactionSimple debe devolver el hash correcto utilizando la estrategia especificada:
En esta prueba, se crea una instancia de TransactionSimple con la estrategia de hashing MD5Hash, y luego se calcula el hash utilizando el método calculateHash(). Se verifica que el hash generado para el objeto de transacción coincida con el valor esperado 'a545200e658fda1cf78313b178ce59e9'.

Test: el método setHash() debe cambiar la estrategia hash:
En este test, se crea una instancia de TransactionSimple con la estrategia de hashing MD5Hash, y se calcula el hash utilizando el método calculateHash(). Se verifica que el hash generado con la estrategia MD5 coincida con el valor esperado. Luego, se cambia la estrategia de hashing a SHA256Hash utilizando el método setHash, y se calcula nuevamente el hash. Se espera que el nuevo hash generado con la estrategia SHA-256 coincida con el valor esperado 'c0144abe25b3b8d802c6ea3936f16120f5f508eced4ca1ff12b64a3e1f8b42ed'. Esto asegura que el método setHash() funcione correctamente y cambie la estrategia de hashing utilizada por TransactionSimple.

Conclusiones

El trabajo fue una forma didáctica de entender a grandes rasgos como funciona las transacciones en una blockchain. A lo largo de todo el proceso de desarrollo fuimos corrigiendo sobre la marcha la falta de conocimiento o certezas del contexto que teníamos

sobre el tema (algo que no nos sorprendió porque es el día a día de cualquier desarrollador).

Cuando empezamos a plantear el ejercicio teníamos en mente que existía “una sola” Blockchain y que esta era la única que tenía bloques cerrados, por eso la implementamos como un Singleton, pero luego esto fue cambiando tomando la forma más cercana a lo que tenemos hoy, cambiando algunos métodos en el camino para que tanto las transacciones, como los bloques y nodos puedan hacer las validaciones para evitar actividades maliciosas en la blockchain. Después de esta línea de pensamiento y comenzando a hacer crecer el ejercicio desde ahí fuimos por la implementación de los patrones de diseño **Composite** y **Strategy**.

Como conclusión principal del trabajo podemos decir que comprendimos que un patrón de diseño es una **solución general** que se puede repetir a un problema común en el diseño de software. Aunque también es importante entender que un patrón de diseño no es un diseño terminado que se puede transformar directamente en código, sino que es como una plantilla sobre cómo resolver un problema.

Un patrón de diseño **nombra, motiva y explica** sistemáticamente un diseño general que aborda un problema de diseño recurrente en los sistemas de software. Describe el problema, la solución, cuándo aplicar la solución y sus consecuencias. También proporciona sugerencias y ejemplos de implementación [1].

Además, permiten acelerar el proceso de desarrollo al proporcionar paradigmas de desarrollo probados y comprobados. La reutilización de patrones de diseño ayuda a evitar problemas sutiles que pueden causar problemas importantes posteriormente y mejora la legibilidad del código para desarrolladores y arquitectos familiarizados con los patrones. Además, los patrones permiten a los desarrolladores comunicarse utilizando nombres bien conocidos y comprensibles para las interacciones de software [1].

Por último, si bien el proyecto de ejercicio no era muy grande, a lo largo del desarrollo del ejercicio trabajamos teniendo en cuenta los Code smells, pero aun así, al final del trabajo encontramos algunos indicios de codificación deficiente (código duplicado, métodos largos, etc.), por lo que recurrimos a la refactorización.

Los Code smells no son errores ni errores; no significa que el software no funcionará, pero puede ralentizar el procesamiento, **aumentar el riesgo de fallas y errores** y hacer que el programa sea vulnerable a errores en el futuro. El código maloliente contribuye a la mala calidad del código y, por lo tanto, aumenta la deuda técnica [2].

Los Code smells pueden estar presentes incluso en el código escrito por desarrolladores experimentados. Puede reducir la vida útil del software y dificultar su mantenimiento. La expansión de las funcionalidades del software también se vuelve difícil cuando hay códigos malolientes. Los olores de código pueden pasar desapercibidos muchas veces. Es por eso que tratamos de evitarlos, pero siempre se puede encontrar algo por ahí.

Referencias

[1] Fowler, M., Rice, D., Foemmel, M., Hieatt, E., Mee, R.,, Stafford, R. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional.

[2] Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley. ISBN: 0-201-48567-2