Práctica 4: Herencia, Interfaces y Excepciones

Inicio: A partir del 18 de marzo.
Duración: 3 semanas.
Entrega: En Moodle, una hora antes del comienzo de la siguiente práctica según grupos (semana del 15 de Abril)
Peso de la práctica: 30%

El objetivo de la práctica es aprender a utilizar técnicas de orientación a objetos más avanzadas que en las prácticas anteriores. En concreto, se hará énfasis en los siguientes conceptos: *interfaces, herencia, excepciones y colecciones.*

Introducción. ¿Qué es Blockchain?

Blockchain es una tecnología para la gestión de transacciones de forma distribuida y pública, cuya máxima es que las operaciones ocurran a ojos de todos. Para ello, los usuarios se comunican de forma descentralizada a través de redes P2P, donde el intercambio de mensajes se realiza por difusión (*broadcasting*) y todos los *peers* reciben toda la información que ocurre en todo momento. Esta tecnología ha tenido un gran impacto tanto en la comunidad científica como en el mundo real. Un ejemplo de ello es el alcance de las criptodivisas (Bitcoin, Ethereum, etc), así como las líneas de investigación existentes alrededor de esta tecnología.

El funcionamiento de blockchain básico tiene cuatro fases: (i) se realiza una transacción en la red; (ii) se genera (mina) un bloque en base a esa transacción; (iii) el bloque es evaluado por los pares; (iv) si el bloque es válido se une a la cadena de blockchain y se hace efectiva la transacción.

En esta práctica se requiere diseñar e implementar una aplicación para gestionar un entorno simplificado de Blockchain*, con sus principales elementos constitutivos: Wallets, Transactions, Blocks, Nodes y Messages, entre otros.

**Algunas de las operaciones y actores relativos a la tecnología Blockchain han sido suprimidos o modificados por simplicidad..*

Apartado 1. Wallet, Transaction, Node, BlockchainNetwork (2,5 puntos)

Comenzaremos el sistema creando sus componentes básicos: Wallet, Node, BlockchainNetwork y Transaction:

- *Wallet:* corresponde a una cartera de divisas, asociada a una clave. Nuestro sistema de Blockchain permitirá realizar transacciones entre *Wallets* a través de la red. Un *Wallet* tiene un nombre de usuario, una clave y el balance total de la cartera. La clave se calculará como el SHA1 de una cadena de tipo String. Para calcular el SHA1 puedes utilizar el método sha1 dentro de la clase CommonUtils, que puedes encontrar en los ficheros adicionales de la práctica en Moodle.
- *Transaction*: los *Wallets* pueden llevar a cabo transacciones entre ellos. Cada transacción tiene un identificador único e incluye la clave del emisor, la clave del receptor y el valor de la transacción.
- *Node*: es el elemento básico para el funcionamiento del Blockchain. Cada nodo tiene un identificador único en la red, un Wallet asociado y una lista de transacciones (con el objetivo de conocer cuales están confirmadas). Hay dos tipos de nodos: simples y minadores. Los nodos simples están enfocados únicamente a hacer transacciones, mientras que los nodos minadores, además de proporcionar la misma funcionalidad que los nodos simples, son capaces de minar y validar bloques (lo veremos en apartados posteriores), por lo que definen una capacidad computacional medida en MIPS.
- *BlockchainNetwork*: representa la red de comunicaciones del entorno Blockchain. Tiene un nombre y está compuesta por una lista de nodos y subredes (*Subnet*). Una subred tiene un identificador único, está compuesta de nodos, y su constructor debe soportar la creación de subredes de distintos tamaños, es decir, debe poder recibir distinto número de nodos como parámetro.

A continuación, tienes un programa de prueba con la salida esperada.

```
public class TesterMainExercise1 {
   protected Wallet wallet1, wallet2, wallet3;
   protected MiningNode miningNode, miningNode2;
   protected Node node;
   protected Subnet subnet;
   protected BlockchainNetwork network;

   public void buildNetwork() {
        //Create the wallets
        this.wallet1 = new Wallet("Bob",   CommonUtils.sha1("PK-Bob"),   10);
        this.wallet2 = new Wallet("Alice", CommonUtils.sha1("PK-Alice"), 100);
        this.wallet3 = new Wallet("Paul",  CommonUtils.sha1("PK-Pauk"),  777);

        //Create the nodes with the wallets
        node = new Node(wallet1);
        miningNode = new MiningNode(wallet2, 10000);

        //Create a subnet inside a network
        miningNode2 = new MiningNode(wallet3, 10000);
        subnet = new Subnet(miningNode2); // we could pass more nodes here

        //Create the network and connect the elements
        this.network = new BlockchainNetwork("ADSOF blockchain");
```

```
        network.connect(node)
               .connect(subnet)
               .connect(miningNode);

        //create example transaction, which transfers 10 coins from wallet1 to wallet2
        new Transaction(wallet1, wallet2, 10);
    }

    public static void main(String[] args) {
        TesterMainExercise1 tme = new TesterMainExercise1();
        tme.buildNetwork();
        System.out.println(tme.network);
        System.out.println(tme.node.fullName()); // prints the name of the node
        System.out.println("End of party!");
    }
}
```

Salida esperada:

```
ADSOF blockchain - new peer connected: u: Bob, PK:5ae5a298259a1bb276d7ea989d9f532197422d5c, balance: 10 | @Node#000
ADSOF blockchain - new peer connected: Node network of 1 nodes: [u: Paul,
PK:6635750ecb4e3ae023736285438c547c5dd22d51, balance: 777 | @MiningNode#002]
ADSOF blockchain - new peer connected: u: Alice, PK:cfa3d38439cff3447441a1e7022d70c17d831519, balance: 100 |
@MiningNode#001
ADSOF blockchain consists of 3 elements:
* u: Bob, PK:5ae5a298259a1bb276d7ea989d9f532197422d5c, balance: 10 | @Node#000
* Node network of 1 nodes: [u: Paul, PK:6635750ecb4e3ae023736285438c547c5dd22d51, balance: 777 | @MiningNode#002]
* u: Alice, PK:cfa3d38439cff3447441a1e7022d70c17d831519, balance: 100 | @MiningNode#001

Node#000
End of party!
```

Notas adicionales:

- Organiza tu código y los testers en paquetes. No olvides crear programas de prueba adicionales al del enunciado.
- Observa con detalle la creación de la *BlockchainNetwork* y la conexión de sus elementos.

Apartado 2. Mensajes y control de errores mediante excepciones (3 puntos)

En este apartado extenderemos el sistema Blockchain para tratar el paso de mensajes y realizar un control básico de errores mediante excepciones.

Los nodos y subredes de una Blockchain podrán intercambiar diferentes tipos de mensajes. Todos los mensajes deben implementar la interfaz *IMessage* que se proporciona a continuación. La interfaz tiene dos métodos: uno para obtener el texto del mensaje, y otro que implementa cierto procesamiento por un nodo. La interfaz proporciona código por defecto para el segundo método, que simplemente imprime un mensaje, pero este comportamiento puede variar dependiendo del tipo de mensaje y tipo de nodo.

```
public interface IMessage {
    public String getMessage();
    public default void process(Node n) {
        System.out.println("["+n.fullName()+"]"+
                           " - Received notification - Nex Tx: "+
                           this.getMessage());
    }
}
```

Los nodos y subredes podrán enviar y recibir diferentes tipos de mensajes. Para ello, tanto ellos como la red (*BlockchainNetwork*) deben implementar la interfaz *IConnectable*, que se encargará de enviar los mensajes a través de la red utilizando el método broadcast.

```
public interface IConnectable {
    public void broadcast(IMessage msg);
    public IConnectable getParent();
    public default IConnectable getTopParent() {
        IConnectable parent = getParent();
        while (parent!=null) {
          if (parent.getParent()==null) return parent;
          parent = parent.getParent();
        }
        return parent;
    }
}
```

El método `broadcast(IMessage)` se encargará de procesar el mensaje cuando se ejecute sobre un nodo, o de redistribuirlo cuando se ejecute sobre una red o subred. Para soportar subredes (*Subnet*) dentro de redes (*BlockchainNetwork*), el método `getParent` devolverá el objeto *IConnectable* padre si lo hay, o null en caso contrario. El método default `getTopParent` devuelve el objeto *IConnectable* de nivel superior, y puede serte útil en apartados posteriores.

Por ahora, daremos soporte al tipo de mensaje *TransactionNotification* (en el apartado 3 daremos soporte a más tipos). La finalidad de este tipo de mensaje es notificar a la red la creación de una nueva transacción, por lo que es necesario que la clase contenga un campo para guardar la transacción asociada. Estos mensajes deben ser recibidos y gestionados por los nodos. En concreto, en este apartado, únicamente será necesario que los nodos guarden la transacción del mensaje en su lista de transacciones.

Como control de errores, si se intenta añadir a la *BlockchainNetwork* un nodo que ya existe en la red, se debe lanzar la excepción `ConnectionException`, y si el nodo pertenece a otra red o subred, se debe lanzar la excepción `DuplicateConnectionException`. Estas excepciones deben reportar el identificador del nodo y un mensaje de error. Además, si se intenta crear una transacción con un balance negativo o sin balance suficiente, se emitirá la excepción `TransactionException`, que guardará las claves públicas del emisor y el receptor, el balance y el mensaje a mostrar.

El siguiente tester ilustra el uso de estas excepciones, así como la difusión de mensajes de tipo *TransactionNotification*.

```
public class TesterMainExercise2 extends TesterMainExercise1{
   public void buildFaultyNetwork() {
        super.buildNetwork();
        try {
           this.network.connect(this.node); // cannot connect: node already in the network
        } catch (ConnectionException e) {
           System.err.println(e);
        }
        try {
           this.network.connect(this.miningNode2); // cannot connect: miningNode in a subnet
        }  catch (DuplicateConnectionException e) {
           System.err.println(e);
        }
   }

   public void createTransactions() {
        try {
           Transaction tr1 = node.createTransaction(wallet2, 10);
           network.broadcast(new TransactionNotification(tr1));
           Transaction tr2 = miningNode.createTransaction(wallet1.getPublicKey(), -1);// negative fails
           network.broadcast(new TransactionNotification(tr2));
        } catch (TransactionException e) {
           System.err.println(e);
        }
   }

   public static void main(String[] args) {
        TesterMainExercise2 tme = new TesterMainExercise2();
        tme.buildFaultyNetwork();
        tme.createTransactions();
   }
}
```

Salida esperada: (errata corregida señalada en amarillo)

```
ADSOF blockchain - new peer connected: u: Bob, PK:5ae5a298259a1bb276d7ea989d9f532197422d5c, balance: 10 | @Node#000
ADSOF blockchain - new peer connected: Node network of 1 nodes: [u: Paul, PK:6635750ecb4e3ae023736285438c547c5dd22d51,
balance: 777 | @MiningNode#002]
ADSOF blockchain - new peer connected: u: Alice, PK:cfa3d38439cff3447441a1e7022d70c17d831519, balance: 100 |
@MiningNode#001
Connection exception: Node 000 is already connected to the network
Connection exception: Node 002 is connected to a different network
[Node#000] - Received notification - Nex Tx: Transaction 1| from: 5ae5a298259a1bb276d7ea989d9f532197422d5c, to:
5ae5a298259a1bb276d7ea989d9f532197422d5c, quantity: 10
[Subnet#003] Transaction 1| from: 5ae5a298259a1bb276d7ea989d9f532197422d5c, to: cfa3d38439cff3447441a1e7022d70c17d831519,
quantity: 10
Broadcasting to 1 nodes:
[MiningNode#002] - Received notification - Nex Tx: Transaction 1| from: 5ae5a298259a1bb276d7ea989d9f532197422d5c, to:
cfa3d38439cff3447441a1e7022d70c17d831519, quantity: 10
[MiningNode#001] - Received notification - Nex Tx: Transaction 1| from: 5ae5a298259a1bb276d7ea989d9f532197422d5c, to:
cfa3d38439cff3447441a1e7022d70c17d831519, quantity: 10
Negative transfer attempt: source: cfa3d38439cff3447441a1e7022d70c17d831519, receiver:
5ae5a298259a1bb276d7ea989d9f532197422d5c, amount: -1
```

Notas adicionales:

- Observa en la salida esperada los tres tipos de excepciones a tratar, las dos primeras relativas a nodos existentes, y la tercera a problemas con transacciones. Crea una jerarquía de excepciones que facilite su uso.
- Revisa con atención el broadcast de los mensajes de tipo *TransactionNotification*.

Apartado 3. Bloques, minado, validación y consolidación (4 puntos)

Hasta ahora hemos diseñado los principales elementos de Blockchain, además de implementar la primera fase de las cuatro especificadas en la introducción, es decir, *crear y enviar una transacción a la red*. En este apartado, se van a diseñar e implementar las siguientes etapas: *generación de un bloque (minado), validación de un bloque* y *consolidación de la transacción*.

3.1 Bloques

Cuando una transacción es difundida por la red, debe ser gestionada por un nodo para ser convertida en un bloque. Este proceso se llama minado y lo realizan los nodos minadores. Para dar soporte a bloques, crea la clase *Block* que define la siguiente información:

- Un identificador único de bloque.
- El número de versión. Puedes darle como valor por defecto el del campo `VERSION` de la clase *BlockConfig*, que está incluida en los ficheros adicionales proporcionados en Moodle.
- *Nonce*, que es un número aleatorio entre 0 y 1000 de un solo uso, usado para evitar ataques de replicación en las comunicaciones.
- La marca temporal (timestamp) del momento en que se mina el nodo. En este tipo de entornos, el timestamp suele ser de tipo entero. Puedes usar (int) (new Date().getTime()/1000) para un timestamp basado en los segundos transcurridos desde el año 1970.
- La dificultad de minar un bloque (que es un número). Puedes darle como valor inicial el del campo `DIFFICULTY` de la clase *BlockConfig*.
- La transacción origen del bloque actual.
- Un flag que indica si el bloque está validado.
- El hash del bloque, que es de tipo String. Este hash debe calcularlo el nodo minador, es decir, no debe calcularse en la clase *Block*.
- El bloque anterior al actual, que podrá ser nulo.

Además, la clase *MiningNode* deberá guardar los bloques que ha validado.

3.2 Minado

Para el minado de bloques se va a partir de la interfaz *IMiningMethod*, de tal modo que se puedan definir distintos métodos de minado mediante la implementación de esta interfaz.

```
public interface IMiningMethod {
	String createHash(Block block);
	Block  mineBlock(Transaction transaction, Block previousConfirmedBlock, String minerKey);
}
```

En este apartado se implementará el método simple de minado (clase *SimpleMining*), donde los métodos de la interfaz deben tener el siguiente comportamiento:

- `createHash(Block)`: Este método genera el hash del bloque recibido como parámetro. Este hash es una cadena de texto que se calcula usando el algoritmo SHA-256, el cual recibe como argumento un String que contiene la concatenación de los siguientes elementos: versión del bloque, hash del bloque anterior (o si no hay bloque anterior, el valor del campo `GENESIS_BLOCK` de la clase *BlockConfig*), timestamp del bloque, dificultad del bloque, y *nonce* de bloque. Puedes encontrar la implementación del algoritmo SHA-256 en la clase *CommonUtils* que está en Moodle.
- `mineBlock(Transaction, Block, String)`: Realiza el minado del bloque. Esta operación crea un nuevo bloque, incluyendo todos los datos especificados anteriormente, y le asigna el hash que retorna el método `createHash`.

`¿Cuándo se realiza el minado?`
El proceso de minado se inicia por un nodo minador, tras recibir un mensaje *TransactionNotification*, y comprobar que la transacción no ha sido confirmada (recuerda que cada nodo almacena una lista de transacciones confirmadas). Dependiendo del tipo de nodo que reciba este mensaje, el comportamiento es ligeramente diferente:

- Nodo simple: Al recibir un mensaje de tipo *TransactionNotification* simplemente debe ignorarlo, ya que los nodos simples no soportan el minado de bloques.
- Nodo minador: Al recibir un mensaje de tipo *TransactionNotification* debe extraer la transacción del mensaje, comprobar que la transacción no está confirmada (no está en la lista de transacciones confirmadas del nodo), y en ese caso, realizar el minado del bloque asociado a la transacción, usando el método de minado. Por el contrario, si la transacción ya está confirmada, se descarta.

El Anexo-1 al final del enunciado contiene un diagrama de secuencia con el funcionamiento esperado.

3.3 Validación

Una vez minado el bloque, debe ser validado antes de consolidar la transacción. Para la validación de bloques se debe emplear la interfaz *IValidateMethod*, del que se realizará una implementación de tipo simple (clase *SimpleValidate*).

```
public interface IValidateMethod {
	public boolean validate(IMiningMethod miningMethod, Block block);
}
```

El método `validate(IMiningMethod, Block)` tiene como objetivo comprobar la integridad y corrección del bloque, es decir, que no ha sido manipulado ni calculado de manera errónea. Para ello, calcula de nuevo el hash del bloque y lo compara con su atributo hash. El bloque será válido si el hash resultante del nuevo cálculo es igual al atributo hash del bloque proporcionado.

`¿Cuándo se realiza la validación?`
Una vez minado un bloque, debe ser validado por toda la red. Para ello, el nodo encargado de minar el bloque debe enviar a la red (a *BlockchainNetwork*) una solicitud de validación del bloque usando el método `broadcast`. Para ello es necesario crear un nuevo tipo de mensaje llamado *ValidateBlockRq*, que es el que se propagará por la red. Este mensaje debe implementar la interfaz *IMessage*, e

incluir el bloque a validar y el nodo que ha minado el bloque. La validación la realizarán los nodos de la red al recibir un mensaje de tipo *ValidateBlockRq*, y dicha validación dependerá del tipo de nodo:

- Nodo simple: Debe ignorarlo ya que los nodos simples no soportan la validación de bloques.
- Nodo minador: Si el nodo no es el que envió el mensaje, debe extraer el bloque del mensaje y utilizar el método `validate` para obtener el resultado de la validación del bloque. En el apartado 3.4 se explicará qué hacer con este resultado.

El Anexo-2 al final del enunciado contiene un diagrama de secuencia con el funcionamiento esperado.

3.4 Consolidación de la transacción

Una vez analizado un bloque, se debe informar a la red del resultado de la validación. Cuando se recibe una validación positiva de un bloque, la transacción asociada pasa a estar confirmada, y los *Wallets* origen y destino de la transacción deben actualizar sus balances.

`¿Cuándo se realiza la consolidación de la transacción?`
Para informar del resultado de la validación a la red se debe crear otro tipo de mensaje llamado *ValidateBlockRes*, que incluye el bloque analizado y el resultado de la validación. Todos los nodos minadores (salvo el responsable del minado) deben validar el bloque minado, tal como se explica en el apartado 3.3, y enviar el resultado a la red (a *BlockchainNetwork*) para su propagación. La consolidación de la transacción la realizan los nodos cuando reciben el mensaje *ValidateBlockRes* con resultado positivo. Todos los nodos deben gestionar este mensaje actualizando su lista de transacciones. Además, si la transacción involucra al *Wallet* del nodo, se deberá actualizar su balance.

Como ejemplo de uso, el siguiente programa debe dar una salida similar a la de más abajo.

```
public class TesterMainExercise3 extends TesterMainExercise2{
   public void createTransactions() {
     //create a transaction and send it to the network
     this.miningNode.setMiningMethod(new SimpleMining());
     this.miningNode.setValidationMethod(new SimpleValidate());
     this.miningNode2.setMiningMethod(new SimpleMining());
     this.miningNode2.setValidationMethod(new SimpleValidate());
     network.broadcast(new TransactionNotification(
                                  node.createTransaction(wallet2.getPublicKey(), 100)));
   }

   public static void main(String[] args) {
     TesterMainExercise3 tme = new TesterMainExercise3();
     tme.buildNetwork();
     tme.createTransactions();
     System.out.println("End of party!");
   }
}
```

Salida esperada:

```
ADSOF blockchain - new peer connected: u: Bob, PK:5ae5a298259a1bb276d7ea989d9f532197422d5c, balance: 10 | @Node#000
ADSOF blockchain - new peer connected: Node network of 1 nodes: [u: Paul, PK:6635750ecb4e3ae023736285438c547c5dd22d51,
balance: 777 | @MiningNode#002]
ADSOF blockchain - new peer connected: u: Alice, PK:cfa3d38439cff3447441a1e7022d70c17d831519, balance: 100 |
@MiningNode#001
[Node#000] - Received notification - Nex Tx: Transaction 1| from: 5ae5a298259a1bb276d7ea989d9f532197422d5c, to:
cfa3d38439cff3447441a1e7022d70c17d831519, quantity: 100
[Subnet#003] Transaction 1| from: 5ae5a298259a1bb276d7ea989d9f532197422d5c, to: cfa3d38439cff3447441a1e7022d70c17d831519,
quantity: 100
Broadcasting to 1 nodes:
[MiningNode#002] - Received notification - Nex Tx: Transaction 1| from: 5ae5a298259a1bb276d7ea989d9f532197422d5c, to:
cfa3d38439cff3447441a1e7022d70c17d831519, quantity: 100
[MiningNode#002] Mined block: id:0, v:0, nonce:20, ts:1710704308, diff:1,
hash:4b980d9c2148a3da101e829df121d38db2634395e1ab8c4c538842fb63462fc3, minerK: 6635750ecb4e3ae023736285438c547c5dd22d51
[Subnet#003] ValidateBlockRq
Broadcasting to 1 nodes:
[MiningNode#002] Received Task: ValidateBlockRq: <b:0, src:002>
[MiningNode#002] You cannot validate your own block
[MiningNode#001] Received Task: ValidateBlockRq: <b:0, src:002>
[MiningNode#001] Emitted Task: ValidateBlockRes <b:0, res:true, src:broadcast>
[MiningNode#001] Commiting transaction : Tx-1 in MiningNode#001
[MiningNode#001]  -> Tx details:Transaction 1| from: 5ae5a298259a1bb276d7ea989d9f532197422d5c, to:
cfa3d38439cff3447441a1e7022d70c17d831519, quantity: 100
[MiningNode#001] Applied Transaction: Transaction 1| from: 5ae5a298259a1bb276d7ea989d9f532197422d5c, to:
cfa3d38439cff3447441a1e7022d70c17d831519, quantity: 100
[MiningNode#001] New wallet value: u: Alice, PK:cfa3d38439cff3447441a1e7022d70c17d831519, balance: 200
[Subnet#003] ValidateBlockRes
Broadcasting to 1 nodes:
[MiningNode#002] Received Task: ValidateBlockRes: <b:0, res:true, src:001>
[MiningNode#001] Received Task: ValidateBlockRes: <b:0, res:true, src:001>
[MiningNode#001] - Received notification - Nex Tx: Transaction 1| from: 5ae5a298259a1bb276d7ea989d9f532197422d5c, to:
cfa3d38439cff3447441a1e7022d70c17d831519, quantity: 100
[MiningNode#001] Transaction already confirmed: Tx-1
End of party!
```

Notas adicionales:
Observa la salida asociada a cada tipo de nodo, y también el tipo de mensaje a difundir *ValidateBlockRq* o *ValidateBlockRes*.

Apartado 4. Diseño y Extensión (0,5 puntos)
Realiza el diagrama de clases de los apartados anteriores, explícalo brevemente, y responde a las siguientes preguntas:

1. ¿Cómo añadirías nuevos métodos de minado y validación?
2. ¿Cómo añadirías nuevos tipos de mensaje que impliquen otros procesamientos por parte de los distintos tipos de nodos?

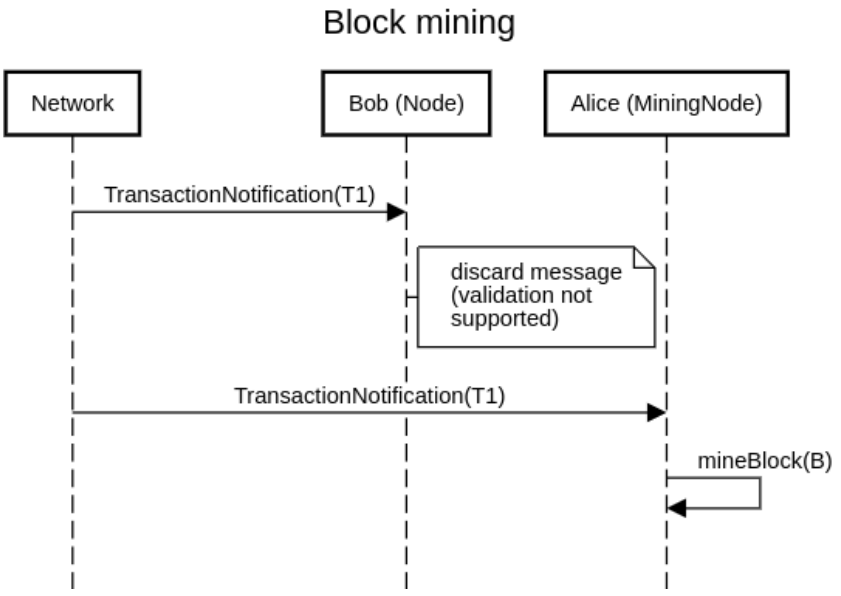
Comentarios adicionales

- No olvides que, además del correcto funcionamiento de la práctica, un aspecto fundamental en la evaluación será la **calidad del diseño**. Tu diseño debe utilizar los principios de orientación a objetos, además de ser claro, fácil de entender, extensible y flexible. Presta atención a la calidad del código, evitando redundancias.
- Organiza el código en **paquetes**.
- Crea **programas de prueba** para comprobar el correcto funcionamiento del código de cada apartado, y entrégalos.

Normas de entrega

- Se debe entregar el **código Java** de los apartados, la **documentación** generada con *javadoc*, los **programas de prueba** creados, y un **documento PDF** con la respuesta del apartado 4.
- El nombre de los alumnos debe ir en la cabecera *javadoc* de todas las clases entregadas.
- La entrega la realizará uno de los alumnos de la pareja a través de Moodle.
- Se debe entregar un único fichero ZIP / RAR con todo lo solicitado, que debe llamarse de la siguiente manera: `GR<numero_grupo>_<nombre_estudiantes>.zip`. Por ejemplo, Marisa y Pedro, del grupo 2261, entregarían el fichero: `GR2261_MarisaPedro.zip`.

Anexo-1: Minado bloque.



Anexo-2: Validación bloque y consolidación de transacción.