HW1-BlockChain

2301212317-LeiCheng-Fintech

TxHandler Class

The `TxHandler` class is a Java class for handling transactions. It implements functionality to validate the validity of transactions and update the Unspent Transaction Outputs (UTXO) pool.

1. Member Variables

- `private UTXOPool utxoPool`: Stores the pool of unspent transaction outputs.

2. Constructor

`public TxHandler(UTXOPool utxoPool)`

The constructor takes a UTXO pool as a parameter and creates a public ledger with the current UTXO pool copied from the given `utxoPool`.

3. Methods

`public UTXOPool getHandledUtxoPool()`

Returns the current UTXO pool being handled.

`public boolean isValidTx(Transaction tx)`

Validates whether the given transaction `tx` is valid. The validation conditions include:

1. All declared outputs are in the current UTXO pool.
2. Each input signature is valid.
3. No duplicate declaration of the same UTXO.
4. All output values are non-negative.
5. The total input value is greater than or equal to the total output value.

`public Transaction[] handleTxs(Transaction[] possibleTxs)`

Processes a set of possible transactions and returns a set of mutually valid accepted transactions. The method performs the following steps:

1. Iterates through the array of possible transactions.
2. Validates the transactions for validity and places valid transactions into a set named `acceptedTxs` and invalid ones into a set named `invalidTx`.
3. Continues processing by checking for valid transactions within invalid transactions until no new valid transactions are found.
4. Returns an array of valid transactions.

`private void HandleValidTx(Set<Transaction> acceptedTxs, Set<Transaction> newlyacceptedTxs, Transaction tx)`

A private method for handling valid transactions. It performs the following steps:

1. Iterates through the inputs of the transaction and removes spent transactions from the UTXO pool.
2. Adds the outputs of the transaction to the UTXO pool.
3. Adds the transaction to the set of accepted transactions.

These methods collectively implement the validation and processing of transactions, ensuring transaction correctness and security.

TransactionBuilder Class

The `TransactionBuilder` class is responsible for constructing various types of transactions. It provides methods to create initial transactions, subdivided transactions, and various payment transactions.

1. Member Variables

- `private KeyPairGenerator keyPairGenerator`: An instance of KeyPairGenerator for generating key pairs.

2. Constructor

`public TransactionBuilder()`

- Initializes the `keyPairGenerator` using the RSA algorithm.

3. Methods

`public Transaction createInitialTransaction(int amount, PublicKey recipientPublicKey)`

- Creates an initial transaction with the specified amount and recipient's public key.
- Generates a key pair for the sender.
- Adds an output to the transaction with the specified amount and recipient's public key.
- Adds an input to the transaction with a null hash and index 0.
- Signs the transaction using the sender's private key.

`public Transaction createSubdividedTransaction(Transaction prevTx, PrivateKey senderPrivateKey, List<Integer> amounts, List<PublicKey> recipientPublicKeys)`

- Creates a subdivided transaction based on a previous transaction.
- Adds an input to the transaction using the hash of the previous transaction and index 0.
- Adds multiple outputs to the transaction with specified amounts and recipient public keys.
- Signs the transaction using the sender's private key.

`public Transaction one_to_onePaymentTransaction(Transaction prevTx, PrivateKey senderPrivateKey, int amount, PublicKey recipientPublicKey)`

- Creates a one-to-one payment transaction based on a previous transaction.
- Adds an input to the transaction using the hash of the previous transaction and index 0.
- Adds an output to the transaction with the specified amount and recipient public key.
- Signs the transaction using the sender's private key.

`public Transaction one_to_morePaymentTransaction(Transaction prevTx, PrivateKey senderPrivateKey, List<Integer> amounts, List<PublicKey> recipientPublicKeys)`

- Creates a one-to-more payment transaction based on a previous transaction.

`public Transaction more_to_onePaymentTransaction(List<Transaction> prevTx, List<Integer> index, PrivateKey senderPrivateKey, int amount, PublicKey recipientPublicKey)`

- Creates a more-to-one payment transaction based on a list of previous transactions and indices.

These methods allow for the creation of different types of transactions, facilitating secure and efficient transfer of funds within the system.

TestMain Class

1. Test valid transactions

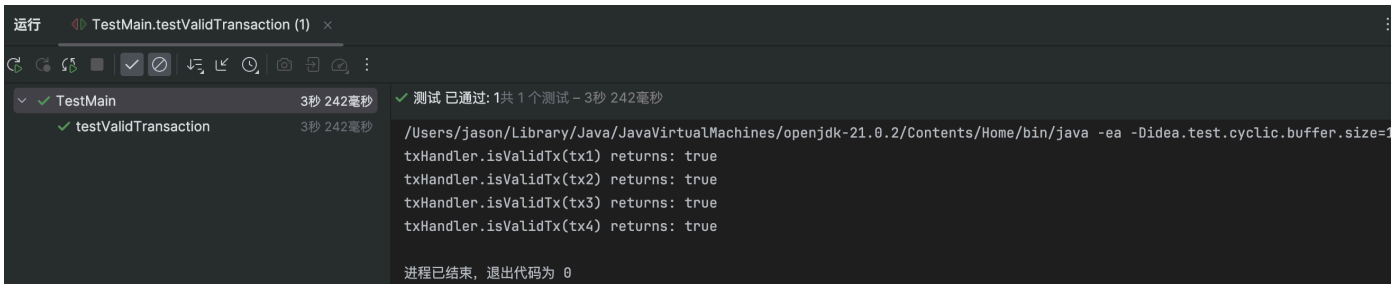
- **Function**: `public void testValidTransaction()`
- **test situation 1**: valid transactions, show basic functions

```
tx0: Jason    --> Jason[0] 12coins [Create Coins]
tx1: Jason[0] --> Jason[0] 4coins  [Divide Coins]
              --> Jason[1] 8coins
tx2: Jason[0] --> Alice[0] 4coins  [Pay separately]
     Jason[1] --> Alice[1] 8coins
tx3: Alice[0] --> Alice[2] 3coins  [Divide Coins]
              --> Alice[3] 1coins
tx4: Alice[1],Alice[2]  --> Bob[0]   8+3coins  [Pay jointly]
```

- **Assertions:**

```
assertEquals("tx1:One valid transaction", 1, txHandler.handleTxs(new Transaction[]{tx1}).length);
assertEquals("tx1:Two UTXOs are created", 2, utxoPool.getAllUTXO().size());

assertEquals("tx2:One valid transaction", 1, txHandler.handleTxs(new Transaction[]{tx2}).length);
assertEquals("tx2:Two UTXOs are used and two UTXOs are created", 2, utxoPool.getAllUTXO().size());

assertEquals("tx3:One valid transaction", 1, txHandler.handleTxs(new Transaction[]{tx3}).length);
assertEquals("tx3:One UTXOs is used and Two UTXOs are created", 3, utxoPool.getAllUTXO().size());

assertEquals("tx4:One valid transaction", 1, txHandler.handleTxs(new Transaction[]{tx4}).length);
assertEquals("tx4:Two UTXOs are used and one UTXOs is created", 2, utxoPool.getAllUTXO().size());
```

- **Running Results:**



2. Test all transactions in one time

- **Function:** `public void TestAllTx_OneTime()`
- **Test Situation:** use the txs example above in testing the vaild txs
- **Assertions:**

```
assertEquals("tx1,2,3,4: four valid transaction", 4, txHandler.handleTxs(new Transaction[]{tx1, tx2, tx3, tx4}).length);
assertEquals("tx1,2,3,4:Two UTXOs are left", 2, utxoPool.getAllUTXO().size());
```

- **Running Results:**

3. Test invalid transactions

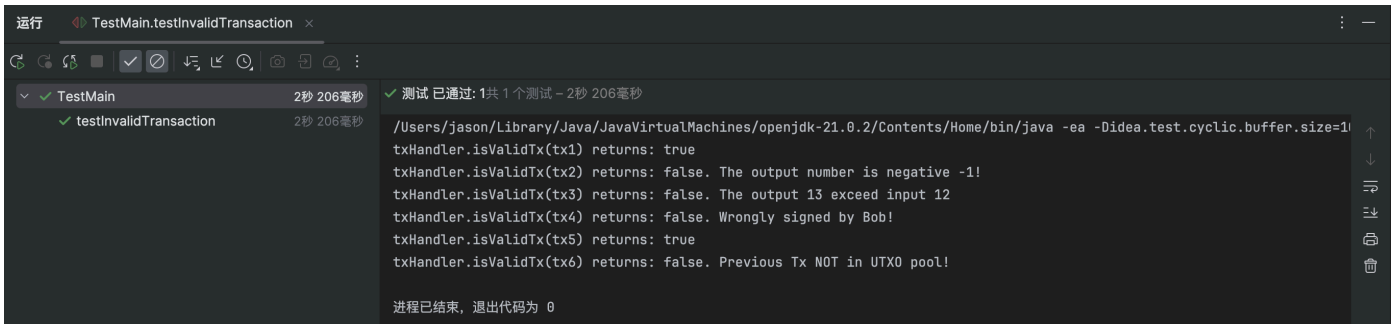
- **Function:** `public void testInvalidTransaction()`
- **Test Situation:** include the following invalid cases

```
tx0: Jason --> Jason[0] 12coins [Create Coins]
tx1: Jason[0] --> Alice[0] 12coins [Pay Coins]
tx2: Alice[0] --> Bob[0]   -1coins [*negative output number*]
tx3: Alice[0] --> Bob[0]   13coins [*output number exceed input number*]
tx4: Alice[0] --> Bob[0]   12coins [*Signed by Bob*]
tx5: Alice[0] --> Bob[0]   12coins [Pay Coins *NOT added to UTXO pool* by removing]
tx6: Bob[0] --> Jason[0]   12coins [Pay Coins *Previous Tx NOT in UTXO pool*]
```

- **Assertions:**

```
assertEquals("tx1:add one valid transaction", 1, txHandler.handleTxs(new Transaction[]{tx1}).length);
assertEquals("tx1:one UTXO's created.", 1, utxoPool.getAllUTXO().size());

assertEquals("tx2:add no invalid transaction", 0, txHandler.handleTxs(new Transaction[]{tx2}).length);
assertEquals("tx2:one UTXO's remained", 1, utxoPool.getAllUTXO().size());

assertEquals("tx3:add no invalid transaction", 0, txHandler.handleTxs(new Transaction[]{tx3}).length);
assertEquals("tx3:one UTXO's remained", 1, utxoPool.getAllUTXO().size());

assertEquals("tx4:add no invalid transaction", 0, txHandler.handleTxs(new Transaction[]{tx4}).length);
assertEquals("tx4:one UTXO's remained", 1, utxoPool.getAllUTXO().size());

assertEquals("tx5:add one valid transaction", 1, txHandler.handleTxs(new Transaction[]{tx5}).length);
assertEquals("tx5: UTXO has been removed.", 0, utxoPool.getAllUTXO().size());

assertEquals("tx2:no valid transaction", 0, txHandler.handleTxs(new Transaction[]{tx6}).length);
assertEquals("tx2:no UTXOs created.", 0, utxoPool.getAllUTXO().size());
```

- **Running Results:**



4. Test double spending

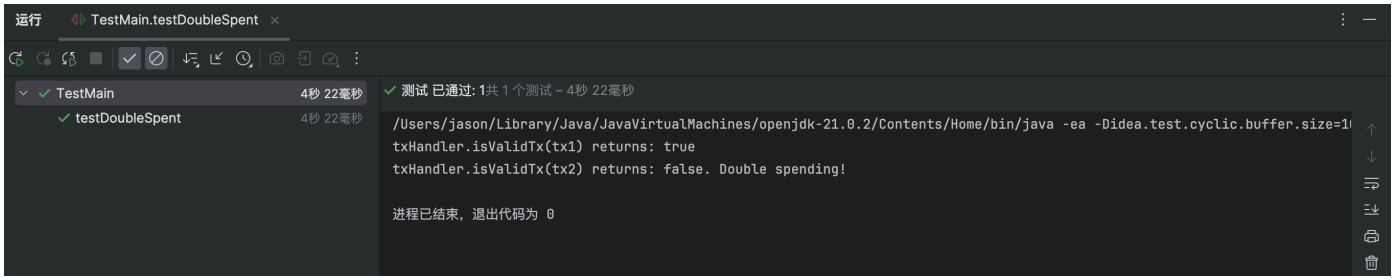
- **Function:** `public void testDoubleSpent()`
- **Test Situation:** double spending

```
tx0: Jason    --> Jason[0] 12coins [Create Coins]
tx1: Jason[0] --> Alice[0] 12coins [Pay Coins]
tx2: Jason[0] --> Bob[0]   12coins [*Double-spending*]
```

- **Assertions:**

```
assertEquals("tx1:add one valid transaction", 1, txHandler.handleTxs(new Transaction[]{tx1}).length);
assertEquals("tx1:one UTXO's created.", 1, utxoPool.getAllUTXO().size());

assertEquals("tx2:add no invalid transaction", 0, txHandler.handleTxs(new Transaction[]{tx2}).length);
assertEquals("tx2:no new UTXOs created.", 1, utxoPool.getAllUTXO().size());
```

- **Running Results:**



5. All test results: