Homework 1: ScroogeCoin

This homework uses **ScroogeCoin** as an example to practice the content learned in class. The core of the assignment is to complete the writing and testing of the **TxHandler** class. The related runnable project code that I have completed is open source[1], and you can also find the related codes in the attachment folder[2]. I will try to describe the process of completing the homework in as much detail as possible in this report, but I still highly **recommend that the teacher and TA review the code and documentation for more information**!

The following content of this report is organized as follows: first, a detailed analysis of the homework's **overall requirements and the provided code** is presented; next, **the logic behind writing the relevant code** is clearly described, including the implementation and testing of the class; finally, information about the runtime **environment** and the configuration of the code used in this report is provided.

1. Overview & Analysis

The detailed requirements of homework 1 are in the **docs/Homework_1.pdf**: (1) We will implement the logic used by Scrooge to **process transactions and produce the ledger**; (2) Transactions can't be validated in isolation; it is a tricky problem to choose a subset of transactions that are **together valid**. (3) We will be responsible for **creating** a file called **TxHandler.java** that implements the required API and **testing** it.

This is a cryptocurrency trading process that matches what was taught in class. We need to write the transaction handler to ensure that the transaction is correct. To complete this assignment, we must first analyze **the given class code**:

- ***Transaction.java***: Represents a ScroogeCoin transaction and has inner classes **Transaction.Output** and **Transaction.Input**.
  - Consists of a list of inputs, which have a value and a public key to which it is being paid.
  - Consists of a list of outputs, which have the hash of the transaction that contains the corresponding output, the index of this output in that transaction, and a digital signature. The raw data that is signed is obtained from the **getRawDataToSign(int index)** method.
  - Consists of a unique ID (see the **getRawTx()** method).
  - Contains methods to add and remove an input, add an output, compute digests to sign/hash, add a signature to an input, and compute and store the hash of the transaction once all inputs/outputs/signatures have been added.
- ***Crypto.java***: Verifies a signature, using the **verifySignature()** method.
- ***UTXO.java***: Represents an **unspent transaction output**.
  - A UTXO contains the **hash of the transaction** from which it originates as well as **its index** within that transaction.

[1] https://github.com/KarryRen/BC-and-DC/tree/main/Homeworks/Homework_1/H1_Project

[2] The implementation code of the **TxHandler** is in *src/main/java/TxHandler.java* and the test code is in *src/main/test/TxHandlerTest.java*.

  - Includes **equals()**, **hashCode()**, and **compareTo()** functions in UTXO that allow the testing of equality and comparison between two UTXOs based **on their indices and the contents of their txHash arrays**.
- *UTXOPool.java*: Represents the **current set of outstanding UTXOs**
  - Contains **a map** from each UTXO to its corresponding transaction output.
  - This class contains **constructors** to create a new empty UTXOPool or a **copy** of a given UTXOPool, and methods to **add and remove** UTXOs from the pool, **get the output** corresponding to a given UTXO, **check** if a UTXO is in the pool, and **get a list of all UTXOs** in the pool.

Based on these elements, we need to finish **implementing and testing** the **TxHandler** based on the given framework. Specifically:

- The implementation of **handleTxs()** should return a mutually valid transaction **set of maximal size** (one that **can't be enlarged simply by adding** more transactions). It need not compute a set of maximum size (one for which there is no larger mutually valid transaction set).
- Based on the transactions it has chosen to accept, **handleTxs()** should also **update its internal UTXOPool** to reflect the current set of unspent transaction outputs, so that future calls to **handleTxs()** and **isValidTx()** are able to correctly process or validate transactions that claim outputs from transactions that were accepted in a previous call to **handleTxs()**.

2. Implementation & Test

The runnable result code is in *TxHandler.java* and *TxHandleTest.java*. In this section, I will elaborate on the class **implementation** and **test** code writing logic. The implementation of the **TxHandler** class contains three methods: **TxHandler()**, **isValidTx()**, and **handleTxs()**. The test suite is used to verify the implementation.

2.1 Details of the implementation

Part 1. TxHandler(UTXOPool utxoPool)

This method is used to create a public ledger with the current UTXOPool being **utxoPool**. So, I defined a private attribute **UTXOPool utxoPool** first. To make a defensive copy of utxoPool, I utilize **new UTXOPool(UTXOPool uPool)** constructor.

```
public TxHandler(UTXOPool utxoPool) {
    this.utxoPool = new UTXOPool(utxoPool);
}
```

Part 2. boolean isValidTx(Transaction tx)

This method is used to verify the validity of the given transaction. It returns true only if the transaction meets the following **5 conditions**:

(1) All outputs claimed by `tx` are in the current UTXO pool. This condition makes sure that all the inputs of the transaction are generated from past transactions' outputs. To implement this, I go through all the inputs of the transaction and check if the utxoPool (created from `TxHandler()`) contains the current input's utxo.

```
// (1) verify if all outputs claimed by tx are in the current UTXO pool
if (!utxoPool.contains(curUTXO)) {
    System.out.println("ERROR (1): The output is not in the current UTXO pool !!");
    return false;
}
```

(2) The signatures on each input of tx are valid. This condition makes sure that people who paid the money is the money's owner. To implement this, I go through all the inputs of the transaction and check if all the inputs' `publickey`, `message` and `signature` can match.

```
// (2) verify the signature on each input of tx is valid
if (!Crypto.verifySignature(preOutput.address, tx.getRawDataToSign(i),
curInput.signature)) {
    System.out.println("ERROR (2): The input's signature is invalid !!");
    return false;
}
```

(3) No UTXO is claimed multiple times by tx. This condition makes sure that there is no double spending in the given transaction. To implement this, for a certain input, I go through all the inputs after it to see if there is any input using the same UTXO with it.

```
// (3) ensure no UTXO is claimed multiple times by tx
if (spentUTXOs.contains(curUTXO)) {
    System.out.println("ERROR (3): Multiple claim of one UTXO !!");
    return false;
} else {
        spentUTXOs.add(curUTXO);
}
```

(4) All of tx's output values are non-negative. This condition is easy to understand because nobody will spend the negative value of money. To implement this, I go through all the outputs and check if their output value is non-negative.

```
// (4) ensure all of tx's output values are non-negative
if (tx.getOutput(o).value < 0) {
    System.out.println("ERROR (4): The output's value is negative !!");
    return false;
}
```

(5) The sum of tx's input values is greater than or equal to the sum of its output values. This condition is also easy to understand because, except for coinbase transactions, no paycoins transactions will have output values larger than the input values. To implement this, I go through all the outputs and outputs and sum the values of inputs and outputs, respectively.

```
// (5) ensure the sum of tx's input values is greater than or equal to the sum of its output values
if (inputsTotalValues < outputsTotalValues) {
    System.out.println("ERROR (5): inputsTotalValues < outputsTotalValues !!");
    return false;
}
```

Part 3. Transaction[] handleTxs(Transaction[] possibleTxs)

Given an **unordered** array of proposed transactions, this method can check each transaction for correctness and then return a mutually valid array of accepted transactions, updating the current UTXO pool as well.

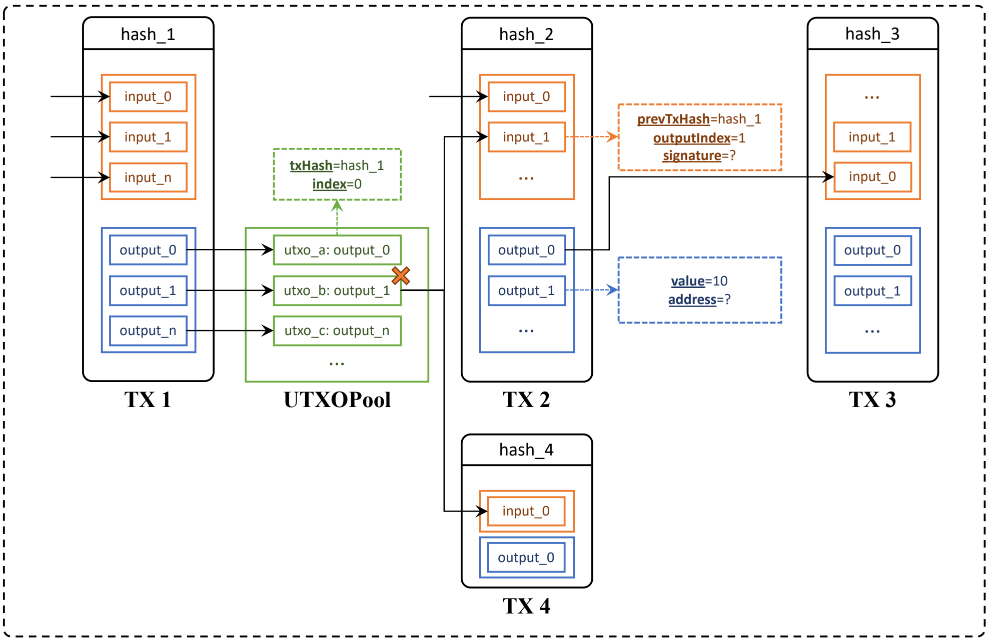


Figure 1. The example of transactions.

Considering the input array is unorder, some transactions may be valid due to other transactions' confirmation. So, I go through all the transactions over and over again until no valid transactions can be found. If there are valid transactions, update the utxopool. This could be kind of inefficient, but it can make sure there won't be any mistakes! To implement this, I divide it into three steps:

- *Step 1*. Go through all the transactions over and over again and record whether it has a new valid transaction or not.
- *Step 2*. Check the validity of each transaction using **isValidTx()**. If it is valid, add it into the acceptedTxs and update the UTXOPool: add new valid output & delete the spent utxo.
- *Step 3*. If there is no new valid transactions can be found, stop going through all the transactions. The list acceptedTxs is what we need!

This is not the end of the analysis! In the real situation, it is easy to see that it is difficult to deal with double spending without a timestamp because there is no way to tell which of the inputs using the same utxo is valid. As Figure 1. shows, there is no way to determine whether TX2 or TX4 is valid. Therefore, after talking with the TA, an additional assumption is made in this place: **for transactions with the double-spend problem, the one that is processed first is valid, and the one that is processed later is invalid**, which is equivalent to a random selection.

For example, **in Figure 1**: If the processing sequence is [TX1, TX2, TX4, TX3], then the acceptedTxs = [TX1, TX2, TX3]. If the processing sequence is [TX1, TX4, TX2, TX3], then the acceptedTxs = [TX1, TX4].

2.2 Details of the test suit

Here, a **full range of black-box tests** need to be performed on the three functions implemented above. Simulating a real transaction between Scrooge, Alice and Bob will be very helpful for testing the functionality. To make things easier, some preparatory work has been done before completing the tests in turn.

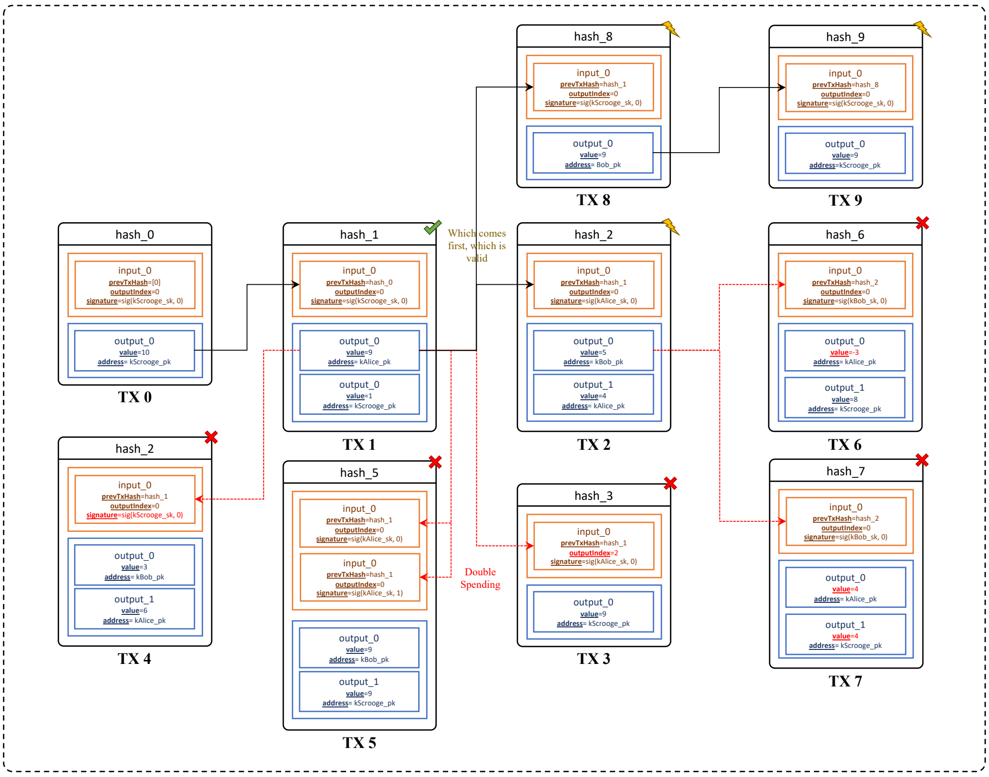


Figure 2. The transaction relationship for testing.

Some preparations for tests @Before

The original Transaction class did not have a function to sign, which means that it is a question of where the signature in Input comes from. Here, the **signTX()** function is added according to the **verifySignature()** method. I create the coinbase transaction **tx0** and initialize the **UTXOPool**. I also created nine other translations (from **tx1** to **tx9**) between Alice, Bob and Scrooge. Some of them are correct, while others are incorrect. The specific input-output relationships are shown in **Figure 2**.

Tests for the isValidTx() function

To present the testing process more clearly, I have made the test table form here. Table 1. shows the details of tests for **isValidTx()** function, including test function, purpose, data, and results.

Table 1. The test table for **isValidTx()** function.

Test Function	Test Purpose	Data for testing	Expected Result	Actual Result
test_tx_is_valid()	Test for valid transactions (which meet 5 conditions).	Valid transaction tx1.	True	True
test_tx_input_not_in _utxoPoo1()	Test for UTXO not being containing in UTXOPool because the previous transaction hasn't happened.	Transaction tx2 before and after transaction tx1 being handled.	Before False After True	Before False After True
test_tx_input_not_in _utxoPool2()	Test for UTXO not being contained in UTXOPool because of pointing the wrong previous transaction's index of outpout.	Transaction tx3 with correct previous transaction tx1 's hashValue but wrong index.	False	False
test_tx_input_with_ wrong_signature()	Test for transaction with wrong signature.	Transaction tx4 with wrong signature	False	False
test_utxo_is_claimed_ multiple_times()	Test for no UTXO is claimed multiple times.	Transaction tx5 with double spending.	False	False
test_tx_output_with_ negative_values()	Test for all of transactions' output values are non-negative.	Transaction tx6 with negative values of outputs even though the total amount of outputs less than the total inputs.	False	False
text_tx_outputValue_ larger_than_inputValue()	Test for the sum of transactions' input values is greater than or equal to the sum of its output values.	Transaction tx7 with output > input even though each output amount is less than the total inputs.	False	False

Note that all the testing data with certain deficiencies can become valid transactions only if we fix the deficiencies. This can ensure that the data for testing are suitable for the certain circumstances we want.

Tests for the handleTxs() function

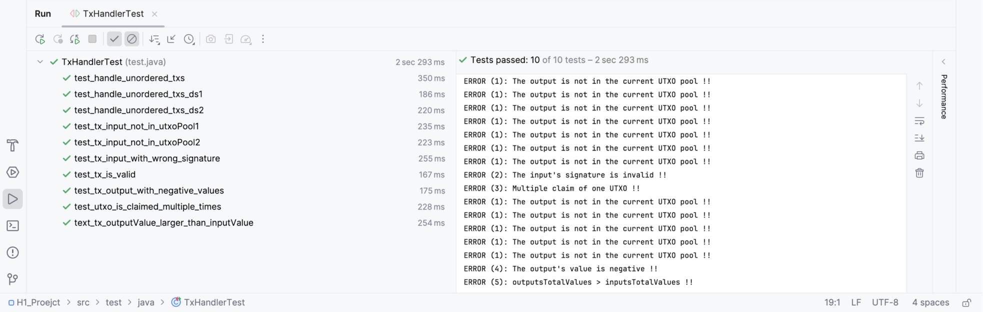
To present the testing process more clearly, I have also made Table 2. I strongly recommend that you refer to Figure 2. for the observations made here!

Table 1. The test table for **handleTxs()** function.

Test Function	Test Purpose	Data for testing	Expected Result	Actual Result
test_handle_unordered_txs()	Test for an unordered array of proposed transactions (some are valid, and some are invalid).	{tx3, tx4, tx5, tx6, tx7, tx2, tx1}	{tx1, tx2}	{tx1, tx2}
test_handle_unordered_txs_ds1()	Test the first situation of double spending in different txs.	{tx2, tx8, tx9, tx1}, where tx2 and tx8 are double-spending	{tx1, tx2}	{tx1, tx2}
test_handle_unordered_txs_ds2	Test the second situation of double spending in different txs.	{tx8, tx2, tx9, tx1}, where tx2 and tx8 are double-spending	{tx1, tx8, tx9}	{tx1, tx8, tx9}

Test results

In my environment (details of which are described below), **all of the above 10 test functions passed**, as shown in Figure 3. This implies that the above function implementation logic and validation ideas are correct.



```
TxHandlerTest (test.java)                          2 sec 293 ms
    test_handle_unordered_txs                          350 ms
    test_handle_unordered_txs_ds1                      186 ms
    test_handle_unordered_txs_ds2                      220 ms
    test_tx_input_not_in_utxoPool1                     235 ms
    test_tx_input_not_in_utxoPool2                     223 ms
    test_tx_input_with_wrong_signature                 255 ms
    test_tx_is_valid                                   167 ms
    test_tx_output_with_negative_values                175 ms
    test_utxo_is_claimed_multiple_times                228 ms
    text_tx_outputValue_larger_than_inputValue         254 ms
```

```
Tests passed: 10 of 10 tests – 2 sec 293 ms
ERROR (1): The output is not in the current UTXO pool !!
ERROR (1): The output is not in the current UTXO pool !!
ERROR (1): The output is not in the current UTXO pool !!
ERROR (1): The output is not in the current UTXO pool !!
ERROR (1): The output is not in the current UTXO pool !!
ERROR (1): The output is not in the current UTXO pool !!
ERROR (1): The output is not in the current UTXO pool !!
ERROR (2): The input's signature is invalid !!
ERROR (3): Multiple claim of one UTXO !!
ERROR (1): The output is not in the current UTXO pool !!
ERROR (1): The output is not in the current UTXO pool !!
ERROR (1): The output is not in the current UTXO pool !!
ERROR (1): The output is not in the current UTXO pool !!
ERROR (1): The output is not in the current UTXO pool !!
ERROR (4): The output's value is negative !!
ERROR (5): outputsTotalValues > inputsTotalValues !!
```

Figure 3. The test results in my environment.

3. Environment

Because JDK versions are updated so quickly, there are a lot of things in the old code that will be wrong in the new JDK version, such as the **finalize()** function. It is highly recommended to **install Java 8 instead of the latest version of the JDK**.

All of the code in this repo is run on the MacOS (M2) with **JDK1.8, junit-4.13.2 and hamcrest-1.3**. Just download the **.jar** files and use the **file => project structure** to organize them.