Concordia Institute for Information Systems Engineering (CIISE)

Concordia University

INSE 6615 - Blockchain Technology

Final Project Report

Submitted to – Prof. Jeremy Clark

Submitted by

Name	Student ID
Abdussamad Syed	40238284
Devina Shah	40238009
Md. Saiduzzaman	40256249
Nikunj Pathak	40203713

Hands-On Solidity: Developing and Testing Smart Contracts

Introduction:

This report is based on our efforts to learn and utilize solidity in order to create smart contracts which can be deployed on the blockchain. As most of us are new to solidity and programming in general, we focused on creating multiple smaller contracts instead of larger complex ones. All these contracts can be viewed in the GitHub repository whose link is in the references. In this report we will go over the various contracts we have developed.

Smart Contracts:

1. Contracts A & B:

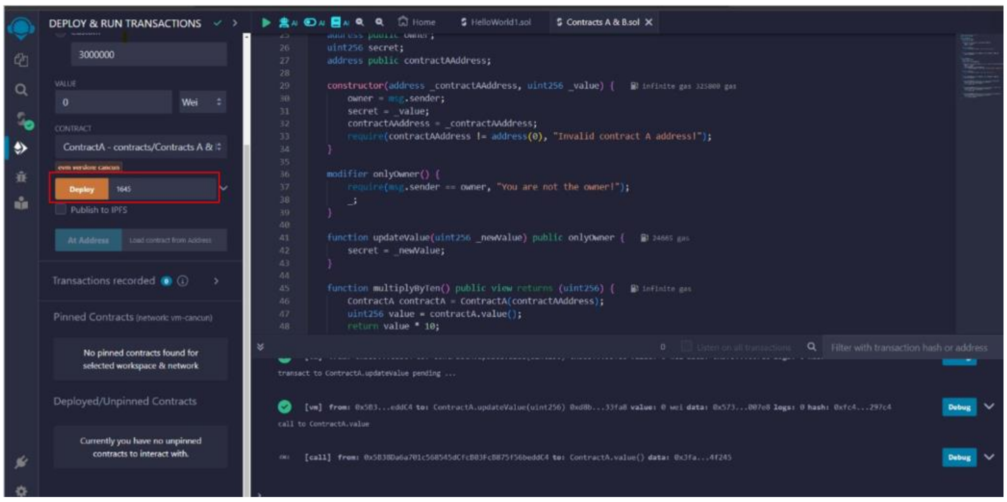
These are a set of contracts, simply title Contract A and Contract B. Here are the details of Contract A:

- Contract A is a very simple contract which has public integer value in its state, set during the contract creation.
- The address of the person deploying the contract is also stored in the state as the owner address.
- A modifier is present which checks whether the person making the transaction is the owner or not. This is used whenever anyone tries to modify the integer value.
- The integer value is made use of by Contract B in its various functions.

Contract B has a similar state, but a few extra functions:

- Contract B stores a private integer value in its state, set during contract creation. It also stores owner address.
- The private integer value can also be updated only by the owner of the contract. This is enforced by the modifier.
- Contract B also stores the address of the deployed Contract A, this value is set during creation of Contract B, with appropriate checks performed to see if the address is valid.
- This contract has 3 functions: multiplyByTen, multiplyByFifty and multiplyBySecret. These essentially pull the integer value from Contract A and multiply by the 10, 50 and the secret value respectively.

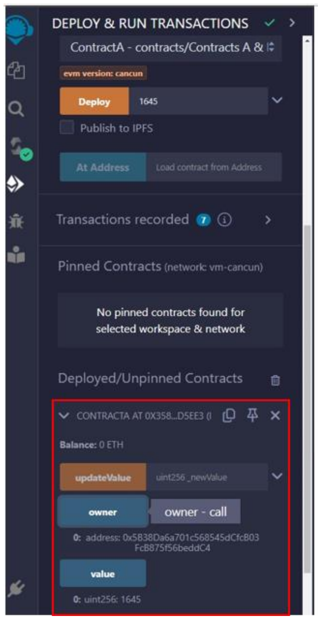
These are one of the first contracts we had come up with. We wanted to get our hands dirty with solidity and also try to see how a function in one contract can pull values from another contract. We tested this set of contracts in the Remix IDE testnets, where we found them to work perfectly as intended.


DEPLOY & RUN TRANSACTIONS
Home
HelloWorld1.sol
Contracts A & B.sol
VALUE
Wei
CONTRACT
ContractA - contracts/Contracts A &
evm version: cancun
Deploy
Publish to IPFS
At Address
Load contract from Address
Transactions recorded
Pinned Contracts (network: vm-cancun)
No pinned contracts found for selected workspace & network
Deployed/Unpinned Contracts
Currently you have no unpinned contracts to interact with.
Listen on all transactions
Filter with transaction hash or address
Debug

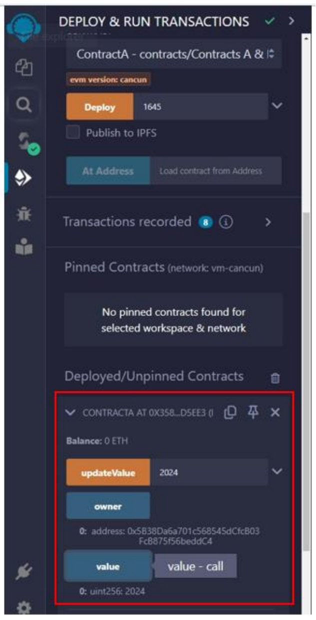
GAS LIMIT: 3000000
VALUE: 0 Wei
CONTRACT: ContractA - contracts/Contracts A &
Deploy: 1645

```
    uint256 secret;
    address public contractAAddress;

    constructor(address _contractAAddress, uint256 _value) {    infinite gas 325800 gas
        owner = msg.sender;
        secret = _value;
        contractAAddress = _contractAAddress;
        require(contractAAddress != address(0), "Invalid contract A address!");
    }

    modifier onlyOwner() {
        require(msg.sender == owner, "You are not the owner!");
        _;
    }

    function updateValue(uint256 _newValue) public onlyOwner {    24665 gas
        secret = _newValue;
    }

    function multiplyByTen() public view returns (uint256) {    infinite gas
        ContractA contractA = ContractA(contractAAddress);
        uint256 value = contractA.value();
        return value * 10;
```

```
transact to ContractA.updateValue pending ...

[vm] from: 0x5B3...eddC4 to: ContractA.updateValue(uint256) 0xd8b...33fa8 value: 0 wei data: 0x573...007e8 logs: 0 hash: 0xfc4...297c4
call to ContractA.value

[call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: ContractA.value() data: 0x3fa...4f245
```


DEPLOY & RUN TRANSACTIONS
ContractA - contracts/Contracts A &
evm version: cancun
Deploy
Publish to IPFS
At Address
Load contract from Address
Transactions recorded
Pinned Contracts (network: vm-cancun)
No pinned contracts found for selected workspace & network
Deployed/Unpinned Contracts
CONTRACTA AT 0X358...D5EE3 (
updateValue
owner
owner - call
value

Deploy: 1645
Transactions recorded: 7
Balance: 0 ETH
updateValue: uint256 _newValue
owner — 0: address: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
value — 0: uint256: 1645


DEPLOY & RUN TRANSACTIONS
ContractA - contracts/Contracts A &
evm version: cancun
Deploy
Publish to IPFS
At Address
Load contract from Address
Transactions recorded
Pinned Contracts (network: vm-cancun)
No pinned contracts found for selected workspace & network
Deployed/Unpinned Contracts
CONTRACTA AT 0X358...D5EE3 (
updateValue
owner
value
value - call

Deploy: 1645
Transactions recorded: 8
Balance: 0 ETH
updateValue: 2024
owner — 0: address: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
value — 0: uint256: 2024

2. Bank Contract:

This is a larger contract, which is arguably more complex than the previous set. As the name suggests, it is meant to mock the operation of a bank. Here is a quick description of the composition of the contract:

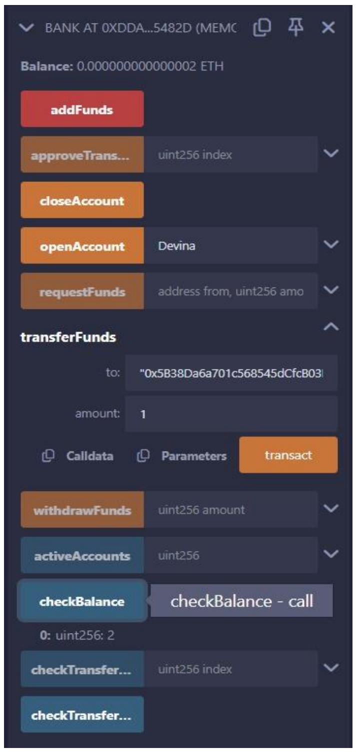
- The contract has a data structure called 'Account', which stores the name of the person who owns the account, their balance, the wallet address they used to open the account and an array of transfer requests that have been made to them.
- There is another data structure called 'TransferRequest', which has the address of the person requesting transfer of funds and the amount that they wish to be transferred.
- The contract also has the address of the person deploying this contract, a mapping of addresses to accounts and an address array of people with active accounts.
- There are a lot of modifiers present to check if an account is present or not, if sufficient balance is present in the users account to make the transaction, if transfer request is valid, etc.

The person who deploys this contract will have their address saved as the bank manager in the state of the contract. As this contract is meant to mock the functionality of the bank there are various functions present which mimic the functionality of a real bank, all the way from opening your account to closing it. All these functions are public, and can be called by anyone with a valid address. Here are the details of the various functions in the smart contract:

- openAccount – allows user to open an account with the bank. Checks are performed to ensure user does not already have an active account.
- addFunds – payable function which allows users to add funds to their account.
- withdrawFunds – function which allows user to withdraw their balance in the form of ether. Checks are performed to make sure user cannot withdraw more funds than what he has in his account.
- checkBalance – allows user to check the balance in their account.
- transferFunds – transfer funds from your account to another. Checks are performed to make sure user cannot transfer more funds than what he has in his account.
- requestFunds – make a request of funds from another account.
- checkTransferReqNum – check the number of transfer request that have been made to your account.
- checkTransferReqDetails – check the details of a particular transfer request made to your account.
- approveTransferRequest – approve a transfer request made to you by another user. Checks are performed to make sure user cannot transfer more funds than what he has in his account.
- closeAccount – close your account and withdraw all your money in the form of ether.

Note that addFunds is the only function where the user sends ether to the contract, while withdrawFunds and closeAccount send ether back to the user. The contract makes use of a fixed conversion rate when funds are added and removed to it, i.e. there is no fluctuation in the amount of ether sent and received (of course gas is spent when making these transactions). Let us consider a scenario where Alice and Bob have valid address and are just looking for somewhere to park their ether, of course there are much better ways to do this however let us assume both of them have decided to use our above contract. They both open accounts by specifying their name and then add some funds. Bob has added a lot more funds than Alice and decides to transfer some to Alice and then withdraws most of the remaining at a later date. After a while Bob is short on cash so he can make a transfer request to Alice. Alice can check who has made transfer requests to her and decide whether or not to approve them. Once she approves the transfer, the amount requested is directly deducted from her balance and then transferred to Bob. This request is then removed from Alice's list of transfer requests. Bob can then decide to cash out and close his account, transferring the funds stored in his account back to him in the form of ether.

The scenario described above is what we have used to verify the functionality of the smart contract. The contract was developed in Remix IDE and tested using the testnets that it provides, with 2 of the various accounts provided serving as Alice and Bob. Additionally, we tested this contract in a local blockchain which we created ourselves using docker, where it worked flawlessly as well.



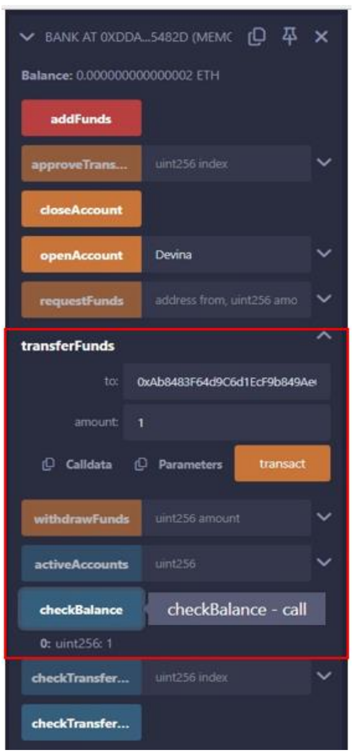
Balance: 0.000000000000002 ETH

openAccount: Devina

to: "0x5B38Da6a701c568545dCfcB03

amount: 1

0: uint256: 2



Balance: 0.000000000000002 ETH

openAccount: Devina

to: 0xAb8483F64d9C6d1EcF9b849Ae

amount: 1

0: uint256: 1

3. Message Board Contract:

This contract was developed in an effort to replicate a simple message board, where you have posts published by users, to which other users can add their comments. You can think of this as being similar to the back end of an application. Here is a quick description of the composition of the contract:

- Post data structure – consists of the title of the post, the body of the post, the address of the author, and the number of comments.
- Comment data structure – consists of the id of the post to which comment is linked, the body/content of the comment, and the address of the comment author.
- User data structure – consists of the name of the user and an array of the ids of the posts belonging to him.

- The contract state has a mapping of address to User data structures, along with an integer representing the number of users.
- In addition, the contract also has an array of posts and comments.
- Various modifiers are present to verify post and comment existence & ownership, as well as modifiers to check length of post title & body, etc.

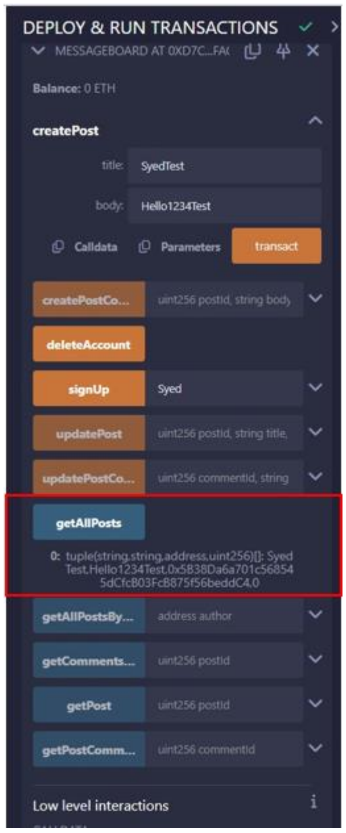
Any user with a valid address can create posts and add comments to said posts. The authors can later update their posts and comments if they wish to do so. To allow users to do so, there are plenty of functions present. Here are the details of the functions:

- signUp – this is meant to mimic how a user would sign up to a website in order to make create posts and add comments. The user has to provide a name when they sign up.
- createPost – this allows active users to create posts. Checks are performed so that the title and body are within the acceptable limits.
- getPost – get a post by specifying its id.
- getAllPostsByAddress – get all posts made by a particular address.
- getAllPosts – get all the posts ever made.
- updatePost – update a post that you own. Checks are performed for ownership and parameter lengths.
- createPostComment – add a comment to a post. Checks are performed for post existence and parameter lengths.
- getPostComment – get comment by specifying its id.
- getCommentsByPost – get all comments associated with a particular post. Checks are performed for post existence.
- updatePostComment – update a comment that you own. Checks are performed for ownership and parameter lengths.

We tried to implement a functionality to delete a post and a comment if the author wishes, but we found that it is quite tricky to implement as this breaks a lot of our existing functionality. Therefore, we were forced to put the development of that functionality on hold.

Consider a scenario where 2 users, Alice and Bob sign up to use this contract. Both these users will be able to make as many posts as they desire, but let use assume Alice made 2 and Bob made only 1. We can query for each post by specifying the id or we can fetch all the posts instead, using getPost and getAllPosts respectively. The former should give us all 3 posts made. We should also be able to query for all the posts made by a specific user/address by making use of getAllPostsByAddress. This function should return us 2 posts when we pass in Alice's address and only 1 when we pass in Bob's. If they chose to do so, Alice and Bob should be able to update their post. Now that we have these posts, the users can add comments to them. Let us say Bob has added comments to each of Alice's posts. Alice (or any other user for that matter) can check the comments she has received on her post by using getCommentsByPost, and can fetch a specific comment by specifying the id. Similar to the post, Bob can update his comment if he chooses to do so.

The above scenario is what was used to verify our smart contract functionality. Similar to the previous contracts, it was developed and tested on the Remix IDE. All the functions and logic work as expected and there are no issues with the code.


DEPLOY & RUN TRANSACTIONS

MESSAGEBOARD AT 0XD7C...FA(

Balance: 0 ETH

createPost

title: SyedTest

body: Hello1234Test

signUp: Syed

getAllPosts

```
0: tuple(string,string,address,uint256)[]: Syed
   Test,Hello1234Test,0x5B38Da6a701c56854
   5dCfcB03FcB875f56beddC4,0
```

4. SyedCoin Contract:

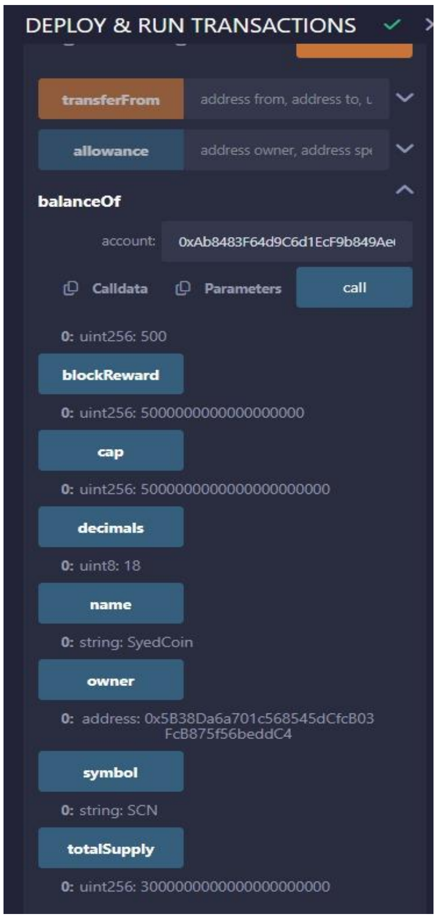
This is a smart contract which extends the functionality of a standard ERC20 implementation offered by OpenZepplin. Therefore, we are able to gain all the functionality that is required by any ERC20 token, on top of which we can build. We followed a standard guide in order to understand how to implement our own custom token known. Here are the details of our token:

- Token name – SyedCoin, Token symbol – SCN
- Cap – The maximum number of tokens that will be in circulation. Passed in as a parameter during contract creation.
- Block Reward – Given to the miner who adds any token transfer transaction to a valid block. Passed in as a parameter during contract creation.
- Owner share – Amount of tokens that will be reserved for the owner of the contract. It is 60% of the cap.
- Decimals – 18 places

We have added additional functionality on top of what we get out of the box. We have added functionality such that whichever miner adds our token transfer transaction to a valid block gets a block reward of our tokens. This block reward can be modified by the owner if required in order to increase/decrease the block reward. We also have checks present so that tokens cannot be minted if the cap has been reached.

Consider a scenario where Alice deploys this contract and specifies the cap as 10000 and 5 respectively. Once the contract is deployed, Alice will have 6000, which is the total supply of tokens that has been minted and are in circulation. Alice can then decide to send 50 tokens to Bob, and the transaction is added to a valid block by Carl. As Carl is the one who added the transaction, he will receive 5 tokens, which increases the supply to 6005. The balances of Alice, Bob and Carl will be 5950, 50 and 5 respectively. Carl can then decide to approve Dora to transfer a set number of tokens on his behalf, which Dora can do any time unless Carl decides to decrease the approved amount back down to 0. Alice may decide that the block reward being provided is too high after a set number of transaction and then halve it to 2.5 tokens.

The above scenario was used to verify the working of the contract. The contract was developed in Remix IDE, however the testing was performed on the Sepolia testnet in order to verify the block reward is provided to the miner.



```
DEPLOY & RUN TRANSACTIONS

transferFrom   address from, address to, u
allowance      address owner, address spe
balanceOf
    account: 0xAb8483F64d9C6d1EcF9b849Ae
    Calldata   Parameters   call
    0: uint256: 500
blockReward
    0: uint256: 5000000000000000000
cap
    0: uint256: 5000000000000000000000
decimals
    0: uint8: 18
name
    0: string: SyedCoin
owner
    0: address: 0x5B38Da6a701c568545dCfcB03
                FcB875f56beddC4
symbol
    0: string: SCN
totalSupply
    0: uint256: 3000000000000000000000
```

Conclusion

In conclusion, our exploration of Solidity programming through the development of various smart contracts has provided us with hands-on experience in blockchain development. From basic contract interactions to more complex simulations of banking operations, message boards, and custom token implementations, we have gained valuable insights into Solidity's capabilities and best practices.

Testing our contracts in different environments has validated their functionality and robustness, while also highlighting areas for improvement. Moving forward, we are committed to refining our skills and exploring advanced use cases to further our expertise in blockchain technology.

Overall, this journey has been quite insightful, and we learnt a lot about smart contracts and their potential applications, setting the stage for continued growth and innovation in the blockchain space.

References

1. Project Code - https://github.com/AbSamad99/blockchain-project
2. Remix IDE: http://remix.ethereum.org
3. Metamask extension (https://metamask.io/)
4. Writing Smart Contract With Remix: https://www.youtube.com/watch?v=eEQ41gD0iC4