Module Code: CS3BC20

Assignment report Title: Blockchain Coursework Assignment

Student Number (e.g. 25098635): 27022879

Date (when the work completed): 19/03/2021

Actual hrs spent for the assignment: 25

Assignment Evaluation

1. Well taught module, video tutorials were very helpful
2. Fun and achievable tasks
3. Poor timetabling in relation to other modules resulted in highly stressful time constraints which led to lower quality of work

Part 1 - Project Setup

Firstly, to set up our offline block chain it is essential that the program was able to be engaged with by a user so that they can input transactions, wallets and blocks later in development. To do this we simply modified the pre-existing “BlockchainApp.cs” Windows form. Windows form is a graphical user Interface class library that we can use to construct a user interface for all the functions of our blockchain. This was done by adding a variety of buttons by using the toolbox located on the left side of the screen (Figure 1). This allows us to drag buttons and other input fields within the toolbox onto the UI shown in BlockchainApp.cs [Design] after this we can change each buttons properties to indicate the corresponding function for example Changing the text from button one to generate new block. After this, in order to handle the user inputs these buttons imply we can delve into the Blockchainapp.cs code by double clicking each button to create a function within Blockchainapp.cs. within each of these functions we can then program the appropriate code to suit that input fields purpose (Figure 2).

Part 2 - Blocks and the Blockchain

Next to fully take advantage of the C# language we implemented an object orientated system. This should occur naturally as a blockchain itself is object orientated in nature. to do this we first created two more classes block.cs and blockchain.cs, (Figure 3) the block chain should contain a chain of blocks and as such the class blockchain will contain a list variable that holds blocks. After the blockchain has been initialised we can add a number of variables to describe a block. To begin with we added a timestamp to record when the block was generated, an index to record the position of that block within the block chain, and the specific hash of the current block and the previous block. Each of these variables are set to a justified data type. for example, timestamp uses datetime as this is the perfect data type in which to describe the date and time of an occurrence. As well as this the index variable was set to an int data type as the index follows the list of blocks and should increment in whole numbers there should be no block indexed with a decimal number as this makes little logical sense and would make the output of our blockchain more difficult to understand. Finally, both hashes are stored as strings as this is the easiest way to hold a variable that will contain both letters and numbers. After we have defined all objects and variables, we can then hash and create our genesis block. A Genesis block Is defined as the first block of a block chain and holds a number of special properties. For example, the values of a genesis block are normally fixed, and the rest of the chain then builds upon this block. Any chain that starts with a block that is not identical to the original genesis block is invalid. The hash however is a combination of all the information within the block. In this case this means hashing all of the variables we defined above using the SHA256 function within a hash creation method, this method will then return the created hash variable. This hashing capability is extremely useful as it allows us to map any size of data within each block to a fixed size and to develop a trust less design as each block can now be verified using its individual hash. Once our block information can be hashed, we can modify our block constructor and produce a method to actually create a block. However, this was be based on our genesis block which was made within a separate constructor which takes no arguments (Figure 4). This constructor simply assigns an empty string to the current hash and begins the block index at 0. After this we then created a constructor for blockchain.cs, this constructor then calls the genesis block constructor adding the genesis block to the blockchains list of blocks and thereafter starting our blockchain.

Part 3 - Transactions and Digital Signatures

Although blockchains can store any type of data within each block our block chain will store a ledger of transactions, to do this we must generate wallets to perform transactions of crypto currency between. Wallets in the blockchain sense are made up of two components, a private key and a public key, this works similarly if not exactly like asymmetric encryption the two keys are mathematically related to each other and while the public key can be seen by all entities with access to the blockchain the private key Is held secretly by the individual user and cannot be decrypted. This allows a user to control their wallet funds securely. In our case we used the elliptic curve digital signature algorithm as it has proven to be cryptographically strong by other cryptocurrencies such as Bitcoin. asymmetric key validation is vital to blockchain as It maintains security of the currency much like the security features on physical bank notes. To implement this the methods found in wallet.cs were used, after which the UI was adapted to fit this purpose (Figure 1). After implementing our method of transactions, it is essential that our transactions can be authenticated to ensure security before they are added to the blockchain. This was done by providing a digital signature with each transaction. To do this each transaction hash was signed with the sender’s private key. This was accomplished by passing the sender's address their private key and the hash of the transaction as arguments into the create signature method inside wallet.cs (Figure 5)however the private key will be removed from memory as saving it as a variable would become a massive security risk. With this infrastructure generating and processing transactions was implemented. In a usual setting for cryptocurrency transactions that are currently being processed and that have not yet been added to the blockchain, they are placed into a transaction pool. This transaction pool allots time for a number of confirmations by entities on a network to reduce the chance of attacks on the block chain. Even though in our case the blockchain is offline we generated a list within blockchain.cs to hold our pending transactions.

Part 4 - Consensus Algorithms

By this point our application was able to generate transactions and a genesis block. To add more blocks onto the blockchain we called the block constructor with each user input to add these new blocks onto the chain it is essential to create a relationship between those blocks pending and those already within the chain. This is done by passing the previous blocks hash to the current block (Figure 6). This method provides a security bonus as if someone attempts to change data within the latest clock on the chain, all previous blocks will have to be changed making the blockchain more secure the longer it becomes, encouraging the generation of new blocks. Next to add transactions to our now fully implemented blocks, a list for holding transactions was added onto the block.cs class. After which the function that retrieves a number of transactions from the transaction pool and adds them to a block was introduced. However due to the nature of block composition they can only handle a finite amount of transactions. The block class constructor was then modified to accept this new list of transactions, after which these transactions are the removed from the pool.

Then to further increase the reliability of the blockchain a consensus algorithm was introduced. A consensus algorithm is used to achieve an agreement among distributed processes on a single value. In this case we used Proof-Of-Work similarly to Bitcoin. The rules of PoW are such that for a block to be added to the blockchain it must contain a hash that is equal to or higher than a given “difficulty” threshold (Figure 7). Using PoW has some advantages being that the entire block is hashed and that each block refers to its predecessor these to features make the blockchain incredibly secure as this chaining technique makes changes immutable and irreversible. However, PoW may be weak in that it is energy intensive. It's costly and requires a lot of computing to perform especially on a larger scale. As well as this for the PoW to work as intended it is reliant on a “nonce” field within each block as otherwise the hash is created it would be the same every time. Including the “nonce” or “number only used once” means that if the algorithm produces a hash that is not “difficult” enough a brand-new hash can be produced in its place. In this implementation a difficulty of 4 was set this is because it would produce a consistent set of results while not being too time costly.

Also, to incentivise users to block mine, mining a block produces a reward. This reward is given to the corresponding minor in the form of a transaction to their wallet. Cryptocurrencies such as Bitcoin greatly incentivise block mining by changing the amount of rewards based on the number of every block mined. In our implementation the reward is currently a fixed value per block mined.

Part 5 - Validation

A Blockchain is normally hosted by many nodes in a peer-2-peer network. Being a trustless network, it is impossible to trust any node in the system, as they could be malicious. Therefore, instead of trusting the other nodes, you must trust the system. To validate the structure of our blockchain we can now utilise the connection between each block to do so. As each block holds the hash for the previous block within the chain we can check the entire chains integrity by iterating though each block making sure that each blocks “PreviousHash” is equal to the “CurrentHash” of the previous block in the chain. By doing this, continuity in the blockchain can be confirmed and shows the user that none of the blocks have been modified in a malicious way, achieving greater trustibility of the system as a whole.

Next to validate our wallet balances another button was implemented into the UI that allows the user to see the current wallet balance and includes further checks within the transaction code to ensure that a user cannot spend more coins than they own (Figure 8). Thus, preventing double spends. In addition, to validate each block a function was implemented to rehash the current block. This rehash is then compared with the original. If both are the same, then the block will be marked as valid. As well as this to verify transactions within a block the Merkle root algorithm was implemented (Figure 9). Merkle root works by iteratively combining the hashes of multiple transactions after the last block has been added to the chain. By calculating the Merkle root a second time after the validate button has been pressed, we can see if any data has been tampered with. an exact match means that the transactions within the block are valid. The Merkle root method provides several benefits as it provides a way to prove both the integrity and validity of data. It significantly reduces the amount of memory needed to do the above and the required proof and management only needs small amounts of information to be transmitted across the P2P networks. Lastly to validate the transactions that take place logic has been implemented to check for the digital signature provided in each transaction. This is done by passing in the public key to see if it matches pair private key signed in the transaction. If the pair of keys works and is valid so is the transaction.

Part 6 - Assignment Tasks

Due to time constraints only task 3 of the 4 assignment tasks was attempted however was never fully implemented.

Appendix

Figure 1 – finished UI and toolbox

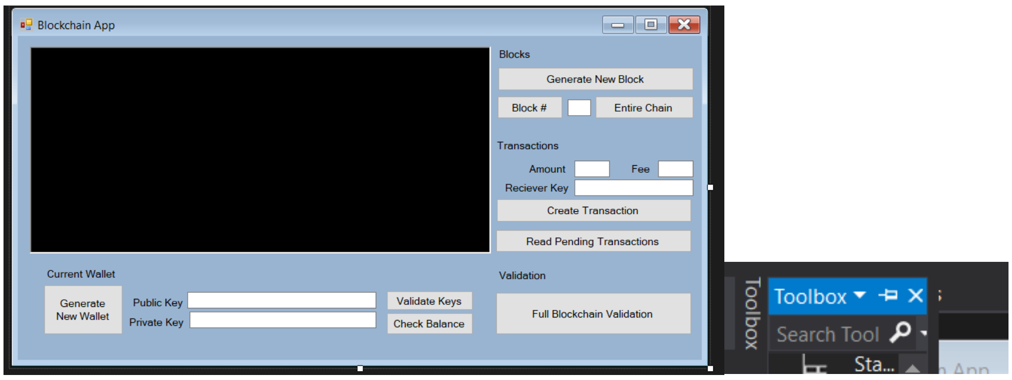
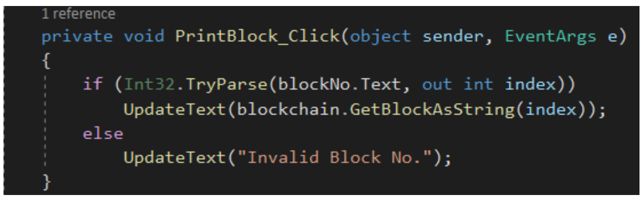
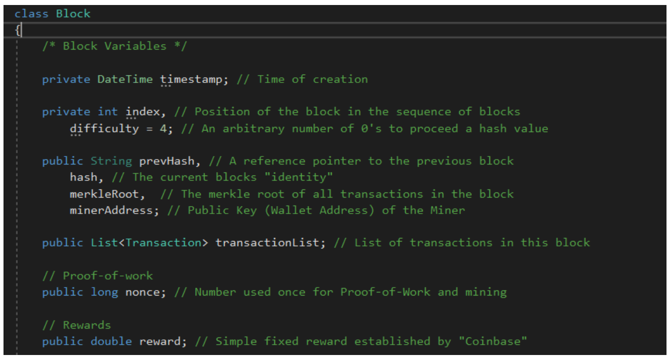


Figure 2 – example of button handling user input

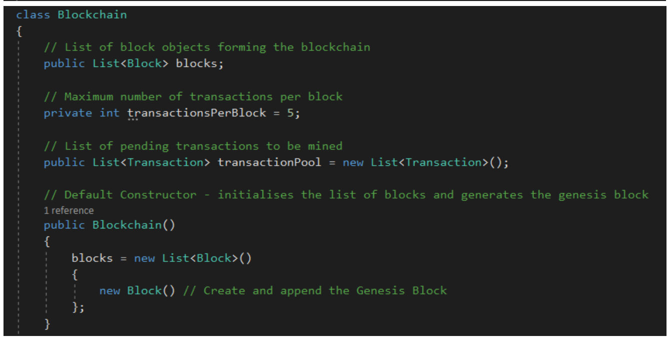


```
1 reference
private void PrintBlock_Click(object sender, EventArgs e)
{
    if (Int32.TryParse(blockNo.Text, out int index))
        UpdateText(blockchain.GetBlockAsString(index));
    else
        UpdateText("Invalid Block No.");
}
```

Figure 3 – blockchain and block complete classes

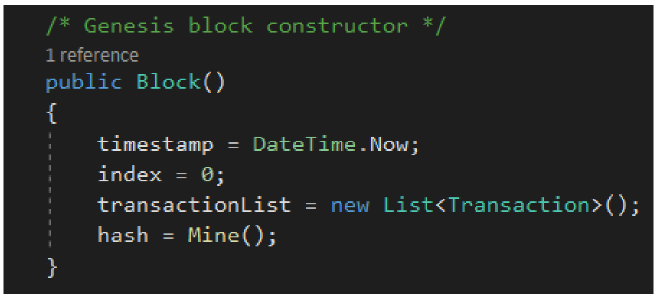


```
class Block
{
    /* Block Variables */

    private DateTime timestamp; // Time of creation

    private int index, // Position of the block in the sequence of blocks
        difficulty = 4; // An arbitrary number of 0's to proceed a hash value

    public String prevHash, // A reference pointer to the previous block
        hash, // The current blocks "identity"
        merkleRoot,  // The merkle root of all transactions in the block
        minerAddress; // Public Key (Wallet Address) of the Miner

    public List<Transaction> transactionList; // List of transactions in this block

    // Proof-of-work
    public long nonce; // Number used once for Proof-of-Work and mining

    // Rewards
    public double reward; // Simple fixed reward established by "Coinbase"
```



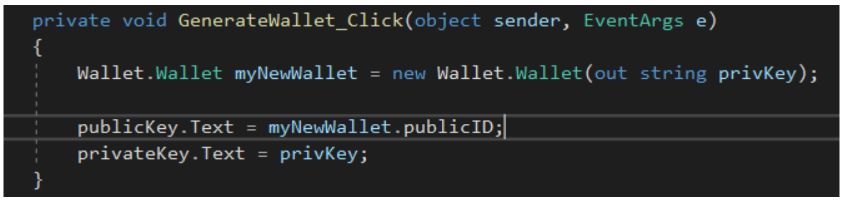
```
class Blockchain
{
    // List of block objects forming the blockchain
    public List<Block> blocks;

    // Maximum number of transactions per block
    private int transactionsPerBlock = 5;

    // List of pending transactions to be mined
    public List<Transaction> transactionPool = new List<Transaction>();

    // Default Constructor - initialises the list of blocks and generates the genesis block
    1 reference
    public Blockchain()
    {
        blocks = new List<Block>()
        {
            new Block() // Create and append the Genesis Block
        };
    }
```

Figure 4 – constructor for the genesis block



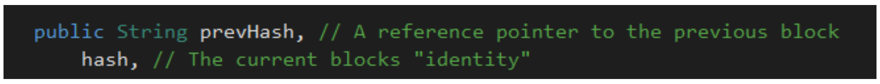
```
/* Genesis block constructor */
1 reference
public Block()
{
    timestamp = DateTime.Now;
    index = 0;
    transactionList = new List<Transaction>();
    hash = Mine();
}
```

Figure 5 – button to generate wallet passing in private key



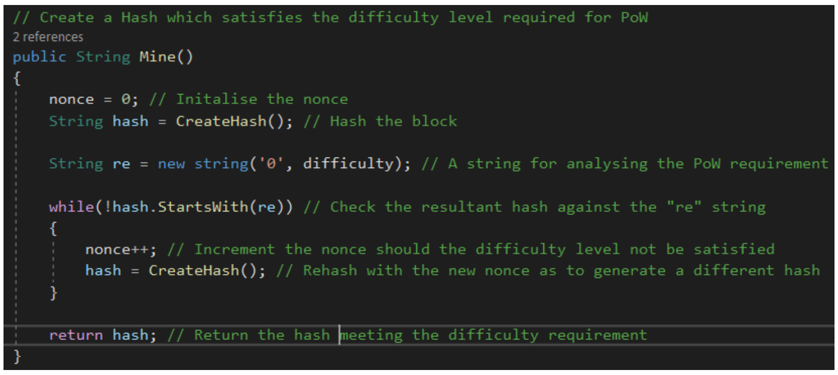
```
private void GenerateWallet_Click(object sender, EventArgs e)
{
    Wallet.Wallet myNewWallet = new Wallet.Wallet(out string privKey);

    publicKey.Text = myNewWallet.publicID;
    privateKey.Text = privKey;
}
```

Figure 6 – definition of previous and current hash with the block class



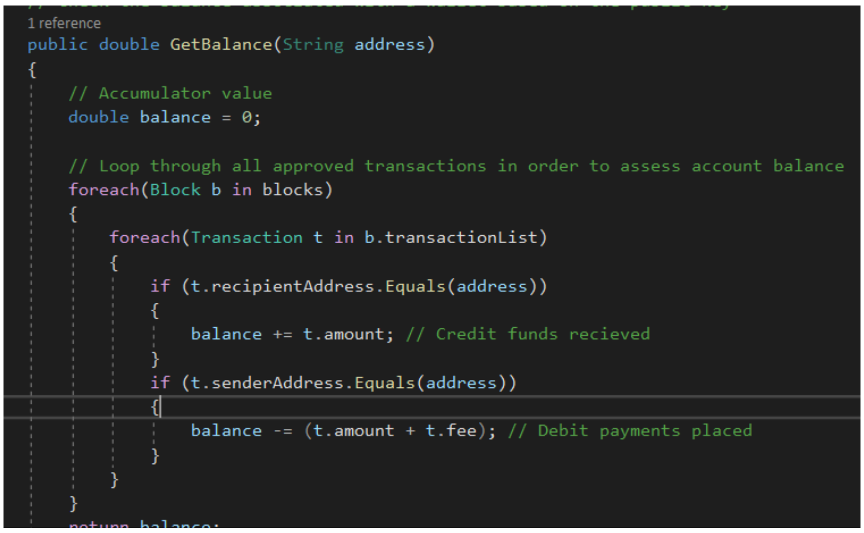
```
public String prevHash, // A reference pointer to the previous block
    hash, // The current blocks "identity"
```

Figure 7 – handles difficulty threshold set for PoW



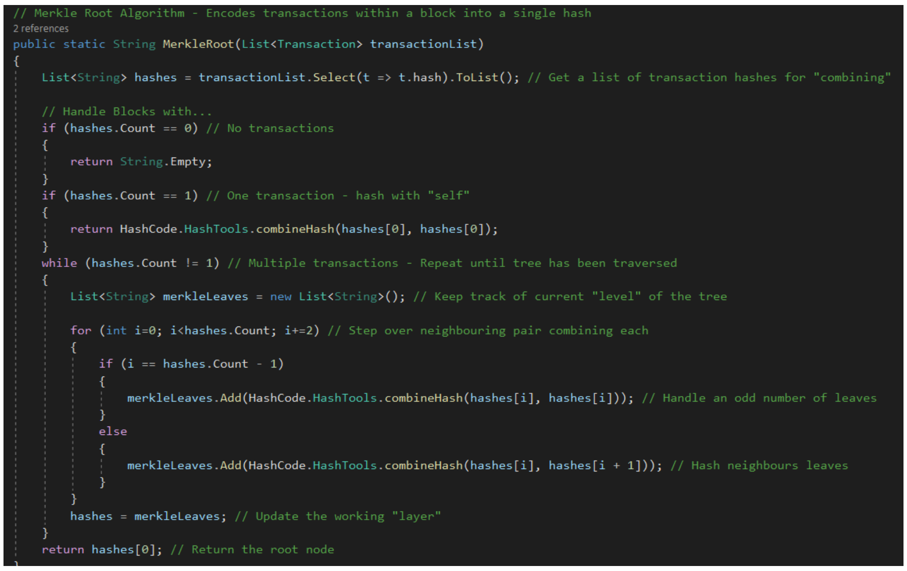
```
// Create a Hash which satisfies the difficulty level required for PoW
2 references
public String Mine()
{
    nonce = 0; // Initalise the nonce
    String hash = CreateHash(); // Hash the block

    String re = new string('0', difficulty); // A string for analysing the PoW requirement

    while(!hash.StartsWith(re)) // Check the resultant hash against the "re" string
    {
        nonce++; // Increment the nonce should the difficulty level not be satisfied
        hash = CreateHash(); // Rehash with the new nonce as to generate a different hash
    }

    return hash; // Return the hash meeting the difficulty requirement
}
```

Figure 8 – used to validate/ check wallet balance for transactions



```
1 reference
public double GetBalance(String address)
{
    // Accumulator value
    double balance = 0;

    // Loop through all approved transactions in order to assess account balance
    foreach(Block b in blocks)
    {
        foreach(Transaction t in b.transactionList)
        {
            if (t.recipientAddress.Equals(address))
            {
                balance += t.amount; // Credit funds recieved
            }
            if (t.senderAddress.Equals(address))
            {
                balance -= (t.amount + t.fee); // Debit payments placed
            }
        }
    }
    return balance;
```

Figure 9 – Merkle root algorithm



```
// Merkle Root Algorithm - Encodes transactions within a block into a single hash
2 references
public static String MerkleRoot(List<Transaction> transactionList)
{
    List<String> hashes = transactionList.Select(t => t.hash).ToList(); // Get a list of transaction hashes for "combining"

    // Handle Blocks with...
    if (hashes.Count == 0) // No transactions
    {
        return String.Empty;
    }
    if (hashes.Count == 1) // One transaction - hash with "self"
    {
        return HashCode.HashTools.combineHash(hashes[0], hashes[0]);
    }
    while (hashes.Count != 1) // Multiple transactions - Repeat until tree has been traversed
    {
        List<String> merkleLeaves = new List<String>(); // Keep track of current "level" of the tree

        for (int i=0; i<hashes.Count; i+=2) // Step over neighbouring pair combining each
        {
            if (i == hashes.Count - 1)
            {
                merkleLeaves.Add(HashCode.HashTools.combineHash(hashes[i], hashes[i])); // Handle an odd number of leaves
            }
            else
            {
                merkleLeaves.Add(HashCode.HashTools.combineHash(hashes[i], hashes[i + 1])); // Hash neighbours leaves
            }
        }
        hashes = merkleLeaves; // Update the working "layer"
    }
    return hashes[0]; // Return the root node
```