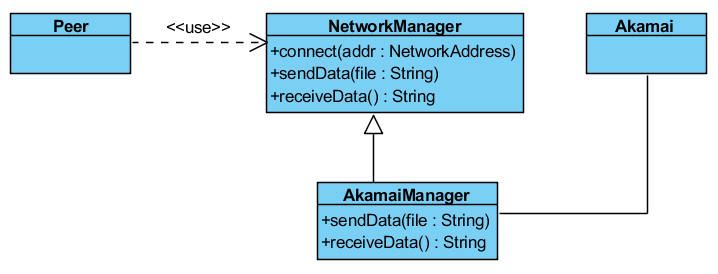
**Pattern Applications**

* **Façade Design Pattern**

For encapsulating individual subsystems, Façade pattern is applicable. By providing the subsystem interface by only one class, other classes can be abstracted. For Blockchain subsystem, BlockchainManager class will provide the interface by managing Blockchain and Peer objects. LocalDataStorage subsystem will have a class called DatabaseManager to encapsulate the peer and block databases.

* **Adapter Design Pattern**

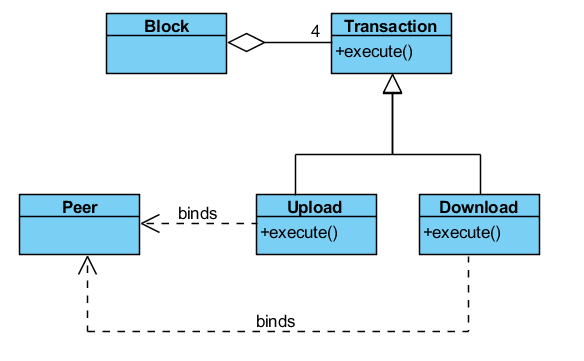
For data distribution, CrypDist will use the services provided by Akamai technologies. Therefore, it should adapt the legacy system and use it without modification. For that purpose, an adapter pattern is applicable. The following is its model.



NetworkManager class is responsinle for managing the network between the peer and Akamai. By inheritance its methods are overridden by AkamaiManager which adapts those methods for the API of Akamai.

* **Command Design Pattern**

Since this is a transaction system, encapsulating transactions with command pattern is useful. By that way, the transactions can be stored and executed without knowing their types and the system becomes extensible. In the future, new transaction types can be added by that way. The following model describes the pattern.



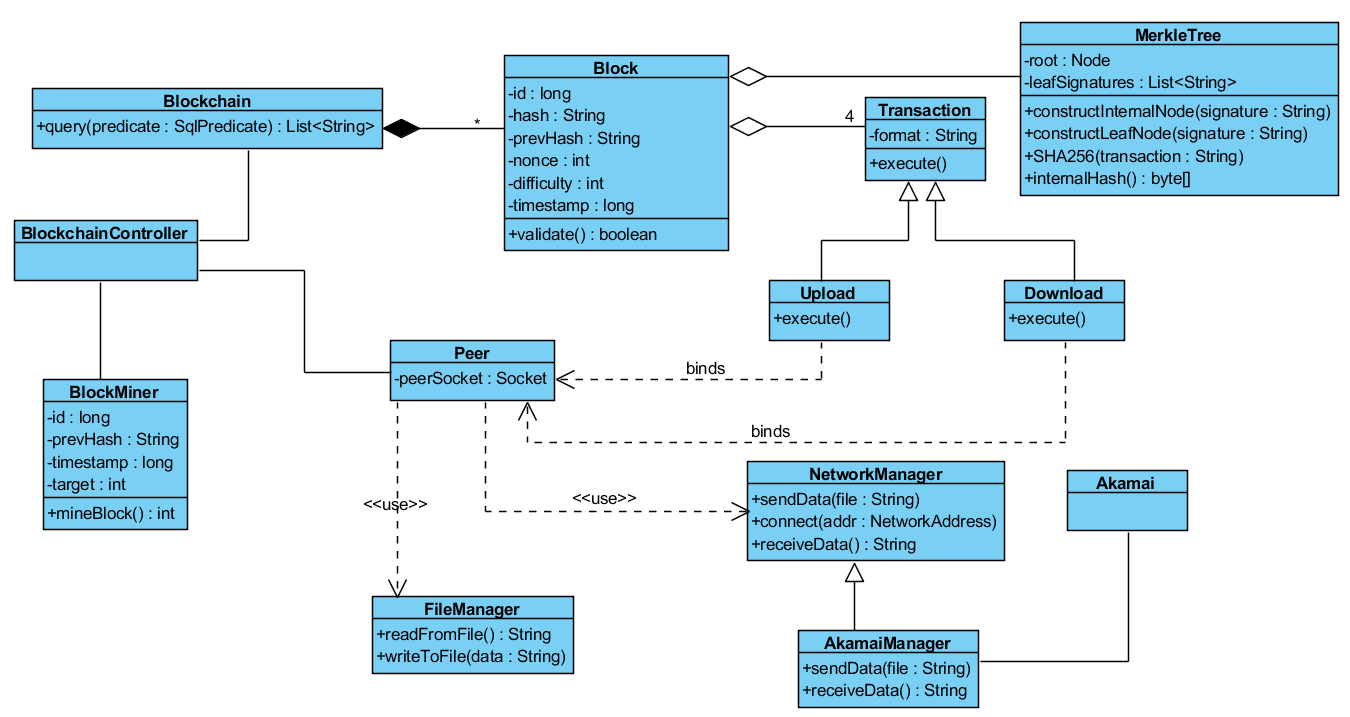
Block contains exactly four transactions. There are two types of transactions: Upload and Download. Both of them overrides the execute method of Transaction class. They belong to a single Peer object.

* **Singleton Design Pattern**

Blockchain object will be replicated among the peers. So, each local program will contain exactly one Blockchain object. Containing multiple Blockchain objects is not valid and may cause unexpected situations. So by a singleton pattern, it will be ensured that there is exactly one instance of the Blockchain class by separating its instantiation from the constructor of the BlockchainManager class.

**Blockchain Subsystem**

Following is the class diagram of blockchain subsystem.



1. **Block Class**

**Attributes:**

* **private long id:** Id of the block
* **private String hash:** Unique hash key of the block
* **private String prevHash:** Hash key of the previous block
* **private int nonce:** The value which is used in the block mining process to generate the distinguishable hash key
* **private long timestamp:** Creation time of the block
* **private int difficulty:** Difficulty of finding the hash key
* **private MerkleTree data:** The data structure which holds the transaction signatures
* **private ArrayList<Transaction> transactions:** Transactions in the block

**Operations:**

* **public boolean validate():** Checks if the block structure is valid for adding to the blockchain.

1. **Blockchain Class**

**Attributes:**

* **private ArrayList<ArrayList<String>> chains:** Holds the chains in the structure. Each chain contains block hashes. Since there are forks, there are multiple chains.
* **private HashMap<String, Block> blockMap:** Maps hash keys to blocks.

**Operations:**

* **public boolean addBlock(Block block):** Adds a new block to the chain if it is valid and returns the receipt.
* **public List<String> query(SqlPredicate predicate):** Executes a query for the chain.

1. **BlockchainController Class**

**Attributes:**

* **private Blockchain blockchain**
* **private Peer peer**

**Operations:**

* **public void addBlockToChain(Block block): Adds a block to the blockchain**

1. **BlockMiner Class**

**Attributes:**

* **int target:** Target difficulty of finding a hash key
* **int maxNonce:** Maximum value of the nonce value
* The others are the same as block except hash key

**Operations:**

* **public int mineBlock():** Finds a hash key for the block by trying different nonce values and returns the minimum appropriate one

1. **MerkleTree Class**

**Attributes:**

* **Node root:** Root node of the tree
* **List<String> leafSigs:** Signatures of the leaves

**Constructor:**

* **public MerkleTree(ArrayList<Transaction> transactions)**

**Operations:**

* **public void constructInternalNode(String signature)**
* **public void constructLeafNode(String signature)**
* **public String SHA256(String transaction):** Computes the signature of the transaction by the SHA256 algorithm.
* **public byte[] internalHash(String leftSignature, String rightSignature):** Computes the signature of the internal node from the child nodes

1. **Peer Class**

**Attributes:**

* **private Socket peerSocket:** End-point of the peer which contains its IP address and port.
* **private FileManager fileManager:** Reads and writes the data into local files.
* **private NetworkManager networkManager:** Establishes a connection between the peer and the Akamai servers.

1. **FileManager Class**

**Operations:**

* **public String readFromFile(String fileName):** Reads the data from the file.
* **public void writeToFile(String data):** Writes the data to the file.

1. **NetworkManager Class**

**Operations:**

* **public void sendData(String fileName, String link):** Send data in the file to the appropriate link.
* **public String receiveData(String link):** Gets the data from the appropriate link.
* **public void connect(NetworkAddress addr):** Connects to the network address

1. **AkamaiManager Class**

**Base Class:** NetworkManager

**Operations:**

* **public void sendData(String fileName):** Send data to the file adapted to Akamai API.
* **public String receiveData(String link):** Gets the data from the appropriate link adapted to Akamai API.

1. **Transaction Class (Abstract Class)**

**Attributes:**

* **private Peer peer:** Peer associated with the transaction
* **private String format:** String representation of transaction for producing its signature

**Operations:**

* **public void execute():** Executes the transaction.

1. **Upload Class**

**Operations:**

* **public void execute():** Executes the transaction.

1. **Download Class**

**Operations:**

* **public void execute():** Executes the transaction.