Design and Implementation of a Token-NFT-Liquidity Smart Contract Suite

Roberto Di Rosa,Luca Sforza

Sapienza

3-12-2025

1. Introduction

We aim to present a suite of smart contracts designed to manage NFT auctions using a custom token called SapiCoin.

We used Solidity language for developing this suite of smart contracts.

A Token can be viewed as a class in other programming languages (e.g. Java), however to avoid binding a smart contract to a specific Token we used an abstraction standardized by the Ethereum Community.

In addition to token transfers and auctions, our system includes a liquidity pool implemented using Uniswap v3, which allows users to trade between Ether and SapiCoin.

2. Tokens

ERC-20 is the standard that represents a token on EVM compatible blockchains.

There is no central authority responsible for issuing these standards; instead, communities such as **Ethereum Magicians** provide a space to discuss improvements to the Ethereum standard through EIPs (Ethereum Improvement Proposals).

2.1.1 Polymorphism

Achieving this requires polymorphism.

The Ethereum Virtual Machine does not provide instructions for handling abstract classes, but the VM itself is polymorphic.

2.1 ERC-20

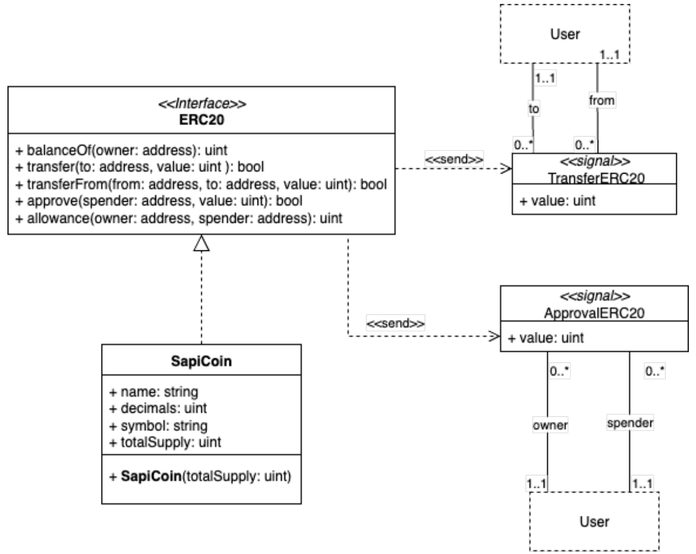
When a method of a smart contract is invoked through a transaction, the memory address of the method is accessed by computing the hash of the function signature. If the invoked method does not exist, a default fallback function is executed (which can be overridden).

```
contract SapiCoin is ERC20 {
  /* Implementation */
}
```

So Solidity exploit this behavior to defines interfaces. So ERC-20 can be viewed as an Interface.

Here we can see a code snippet which shows how to inherit the methods of an interface in Solidity.

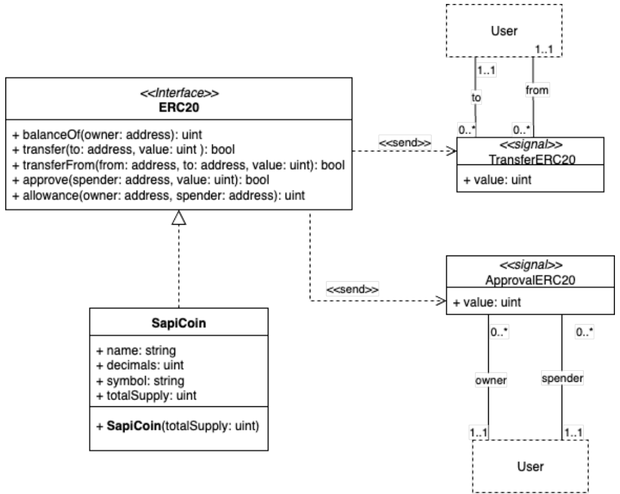
Design of the standard ERC-20



2. Tokens
— 2.1 ERC-20

1. We used a Model Driven Architecture approach, this provides a structured method for designing blockchain smart contracts. Using UML diagrams—such as Class and State Machine diagrams—developers can model both the structure and behavioral logic of smart contracts across multiple abstraction layers, improving clarity, analysis, and maintainability.
2. In this UML class diagram we have the ERC20 seen as an Interface and SapiCoin is a Class that implementes the methods. The methods are:
    1. balanceOf: returns the balance locked, in an address
    2. transfer: transfer token to another address
    3. approve: Approve fuction is need to allow a Spender, which is an address, to spend our money
    4. allowance: returns the amount of token that a spender can spend
    5. transferFrom: trasfer token from an address to another if we are approved to do so.
3. The entire approval mechanism is very useful from a

programming perspective. Users can call approve on a given smart contract for a specified amount of tokens. In this way, a smart contract can use our tokens to provide a service.



> Transfer (index_topic_1 address _from, index_topic_2 address _to, uint256 _value)

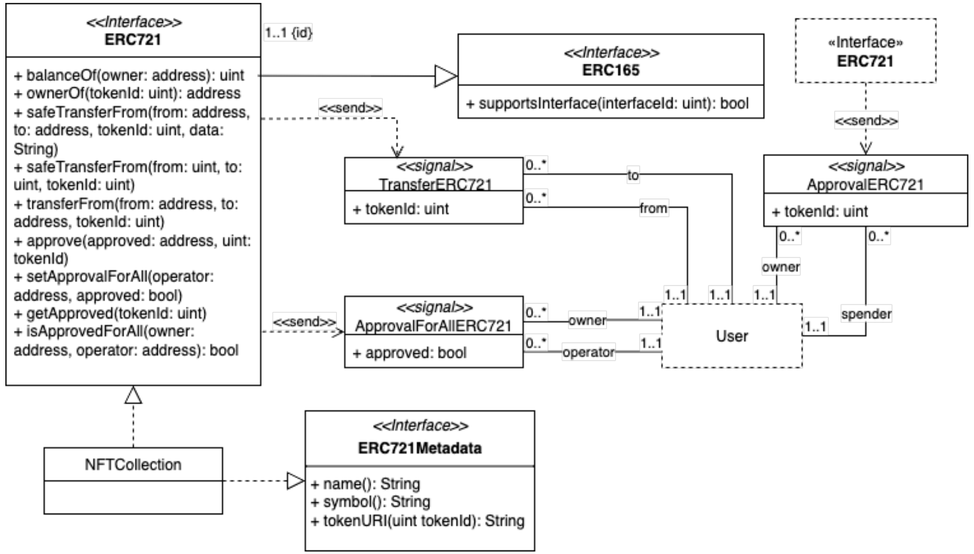
```
[topic0] 0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef
[topic1] 0x000000000000000000000000bf0099919126b15f7f7d2459dae618833c8c4f09
[topic2] 0x000000000000000000000000afd0a8018a587e8c01b5bee71165c2e72e32526f
```

Hex → 0000000000000000000000000000000000000000000000014576f770a4bd61990e9

```
  event Transfer(address indexed _from, address indexed
_to, uint256 _value);
  event Approval(address indexed _owner, address
indexed _spender, uint256 _value);
```


1. ERC-20 also defines a set of events that a token **should** emit whenever the contract's internal state changes, so that the operations executed within a smart contract can be tracked and displayed by front-end applications such as Etherscan.
2. We can also set optional vanities to the Token, such as the name of the token, the token symbol and the number of decimal.
3. To implement a specific token, in our case SapiCoin, there is no additional logic beyond being a token. If we wanted to implement a stablecoin or a wrapped coin, we would need to add methods to manage the exchange of the token with the real-world currency it represents

Design of the standard ERC-721



Now let's talk abouts NFTs defined by the ERC-721 standard.

The ERC-721 interface defines the standard for NFTs. Each token is unique and distinct, unlike ERC-20 tokens which are fungible (all identical). To contrain this non fungibility every Token must have a unique id.

Each NFT can have vanities by implementing the ERC-721-metadata interface which contains:

1. name(): that gives us the name of the collection in which the NFT is contained.
2. symbol(): This will give us the symbol of the collection.
3. tokenUri(uint tokenAdr): This will give us the URI so that we can display the NTF in a front-end application.

2.4 Token Image

The NFT that we minted is associated to this image through the ERC-721-metadata

2.5 ERC-165

The ERC-721 document imposes the implementation of the ERC-165 interface, so that any ERC-721 smart contract can be **queried** to see which interfaces are implemented by the smart contract.
Let us say that we wanted to display an NFT in our website which we did not implement. Therefore we do not know if the Token implements the tokenUri method, in this case we can just ask him with the **supportsInterface(interfaceId:uint):bool**.

According to the standard, the **supportsInterface** function must return true if and only if the provided interfaceID matches the XOR of the function selectors that define the interface.

1. According to the standard the supportsInterface function must return true iff the provided interfaceID matches the XOR of all of the interface's methods' hash signature.
2. This is not a guarantee since malicious contracts can lie and tell you they implement an interface while not actually implementing it, but ERC-165 is a way to standardize the way we ask questions to other contracts.
3. If you are doing the project of Security in Software applications you may need to implement this interface.

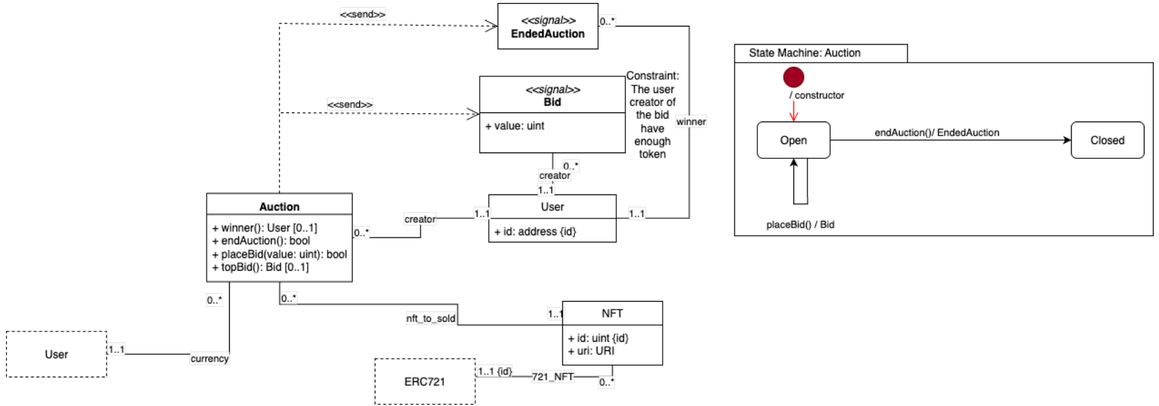
2.6 Implementing ERC-165

```
contract Taxpayer is ITaxpayer, IERC165 {
  mapping(bytes4 => bool) internal supportedInterfaces;

  constructor() {
    supportedInterfaces[type(IERC165).interfaceId] = true;
    supportedInterfaces[type(ITaxpayer).interfaceId] = true;
  }

  function supportsInterface(
      bytes4 interfaceId
  ) external view returns (bool) {
      return supportedInterfaces[interfaceId];
  }
}
```


1. This is the most gas efficient implementation of ERC-165 standard.
2. Given an Interface in solidity thanks to the type function we can access the pre-calculated-interface-id

3. Auction

3.1 Implementation details



There is no standard for NFT auctions, so creating one requires using other strategies. For example, one can use an auction system with contracts already deployed on the blockchain and create an auction through that system. This way, every user of the system can access your auction, for instance via OpenSea.

Unfortunately, the main NFT marketplaces have removed their support for testnet blockchains, so we opted to implement it from scratch.

How does an auction work? An auction is a smart-contract where the deployer of the contract is the owner of the **NFT** that he wants to sell via the auction. While the auction is not closed, users can place bids, when a user places a bid he approves the Auction to use his tokens, then the tokens are transferred from the user to the auction, this makes the auction the sole holder of all the money in the process not the deployer not the users, when you bid you are temporaly transferring your money to the Auction.
When the deployer wants to close the auction he calls the method **endAuction** which will transfer the NFT to the winner, the tokens the winner had bid will then be transferred from the Auction to the deployer, and refund all the other tokens to the losers.

4. Implementation details

4.1 Foundry

Solidity, unlike traditional programming languages, introduces additional constraints related to gas consumption, security, and transaction management.

In particular, Solidity development environments allow direct interaction with the blockchain, enabling developers to perform transactions, deploy smart contracts, and publish contract source code on services such as Etherscan.

One of the most significant constraints imposed by Solidity is that, once a contract is deployed on the blockchain, it cannot be modified. Since deploying smart contracts may incur substantial costs, ensuring the correctness of the code becomes essential. Consequently, testing or even formal verification of the code, understood as proving that the system's invariants are satisfied—is a critical step.

4. Implementation details
— 4.1 Foundry

4.2 State of Art

There are many development environments for Solidity. We chose Foundry. Another widely used option is OpenZeppelin.

Foundry provides a complete ecosystem for working with EVM-compatible blockchains. It consists of three main utilities:

1. forge
2. anvil
3. cast

1. forge, which enables compilation, testing, debugging, deployment, and verification of smart contracts.
2. anvil, which allows the instantiation of a local development node by forking the selected blockchain.
3. cast, a versatile command-line tool for interacting with on-chain applications.

4.3 How to deploy a smart contract

You can download the repo here: https://github.com/LucaSforza/token

The full tutorial is here:

Please put a star :)

This is the Homework that you will do. You have just di download our public repo and follow the tutorial.

4.3 How to deploy a smart contract

```
forge create src/sapicoin.sol:SapiCoin \
  -rpc-url https://ethereum-sepolia-rpc.  \
  -private-key $(cat path/to/pk.txt) \
  -broadcast -constructor-args $@ # constructor arguments

forge verify-contract \
  -rpc-url https://ethereum-sepolia-rpc.publicnode.com \
  -etherscan-api-key $(cat path/to/api_key.txt)
  $(echo $ADDRESS)
  src/sapicoin.sol:SapiCoin
```


By providing the private key, forge can sign transactions on the selected blockchain through the RPC endpoint, which serves as the interface enabling external tools to communicate with a network node (sending transactions, reading state, querying blocks). The output of the create command is the address of the deployed smart contract.

For interacting with the deployed smart contract one way is to publish the source code and then use it on Etherscan using the verify-contract command.

5. Liquidity Pool

Distributing the token to the masses in a "simple" way

5.1 UniSwap

After creating the token we are the sole holder of said token, of course we want people to use our token, to do that we need a scalable system from which people can exchange tokens(WETH, USDT, ...) so that they can use our token. Such a system is called a **Liquidity Pool**.

For this project, we used Uniswap v3. It allows the creation of liquidity pools between SapiCoin and Ether. By depositing an initial amount of both tokens, an initial market price for SapiCoin is established. From that point onward, anyone can buy or sell SapiCoin against Ether, and the protocol automatically updates the price.

5.1.1 Exchange Rate

Uniswap v3 uses the Constant Product Formula to determine the exchange rate:

$$x \cdot y = k$$

k is the invariant of the system, so we can freely change the values of x and y as long as k remains constant.

Therefore, the exchange rate of the token x in exchange for y is $\frac{x}{y}$.

In fiat currencies, central banks (e.g., the European Central Bank) determine the exchange rate. In DeFi currencies, this is not possible because we operate in a distributed system. Uniswap v3 uses the Constant Product Formula to determine the exchange rate:

5.1.2 Investors

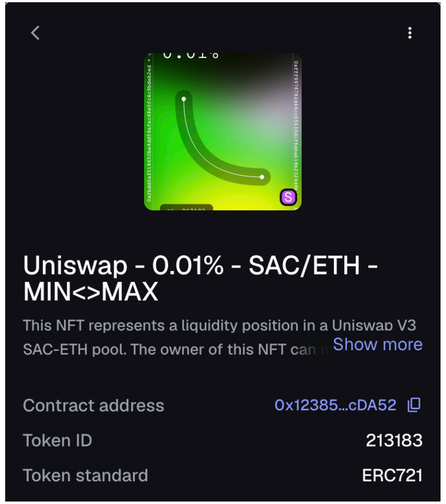
Those who provide liquidity are called investors, creating positions in the liquidity pool. Investors provide liquidity to earn a return. This return comes from fees. Depending on the amount of liquidity contributed to the pool, each investor receives a percentage of the fees collected.

The liquidity is calculated as:

$$L = \sqrt{k} = \sqrt{x \cdot y}$$

When an investor contributes, their position is represented by an NFT.

5.1.3 NFT of the position



This is a example of a position in Uniswap v3. Since this is a NFT you can auction it with our smart contract.

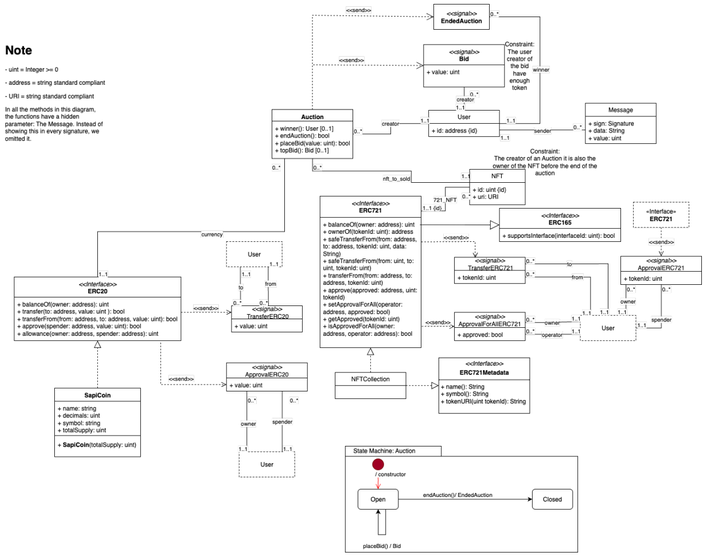
6. Conclusions

Invest money on our start-up (Not a financial Advice)

6.1 Conclusions

We have designed and implemented a complete suite of smart contracts that interact through well-established Ethereum standards. The Token contract follows the ERC- 20 specification and provides a fungible currency (SapiCoin) used as the medium of exchange within the system. The NFT contract follows the ERC-721 standard and defines unique assets that can be auctioned. Finally, the Auction and Liquidity Pool contracts coordinate token transfers, NFT ownership changes, and Ether-SapiCoin conversion.

The architecture demonstrates how interoperability on Ethereum naturally emerges from UML-style interface-based design and the polymorphic behavior of the EVM. Each component is independent yet composable, enabling reuse, modular upgrades, and clear separation of concerns.



This is the complete UML class diagram of our suite of smart contracts!

Thanks for the attention!