**Introduction**

The goal of this project is to implement ownership and renter assets using blockchain ERC20 tokens.

ERC20 tokens are tokens following a list of standards developed by the Ethereum community. In these standards we can denote mandatories functions and optional functions. Each ERC20 token must implement the 6 following functions:

* totalSupply: returns the amount of token in existence
* balanceOf: returns the amount of tokens owned by a certain account
* transfer: moves a certain amount of tokens from the message sender account to a recipient account
* transferFrom: moves a certain amount of tokens from a sender account to a recipient account
* approve: Sets a certain amount as the allowance of a spender over the caller’s tokens
* allowance: Returns the remaining number of tokens that a spender will be allowed to spend on behalf of an owner. This is zero by default.

So in each step of the project we are implementing all these function in the tokens.

An ownership asset token represents the owning of an object that we can give to someone else (by transfer) and each person that has the token must be able to show a proof of ownership.

A rental asset token represent the rent of an object. The rent is set for a certain amount of time, and during the validity of the rent each renter can give the token to someone else that has permission to rent the object. When starting a rent, the owner of the object will define a list of persons that are allowed to keep the rent and according to this list the token can be transferred.

As an example to illustrate this we can think of a car. The owner of the car can sell the car and thus give it to someone else, and he can also decide to rent the car to a group of person that can share the car: during the rent validity time, each person can give the car to someone else in the group.

The project is divided into 3 steps of implementation that are explained below. In each step of the project the contract that are implemented follow the standards of ERC20 token (meaning they are defined as being ERC20 tokens and they implement all the mandatory functions of the token).

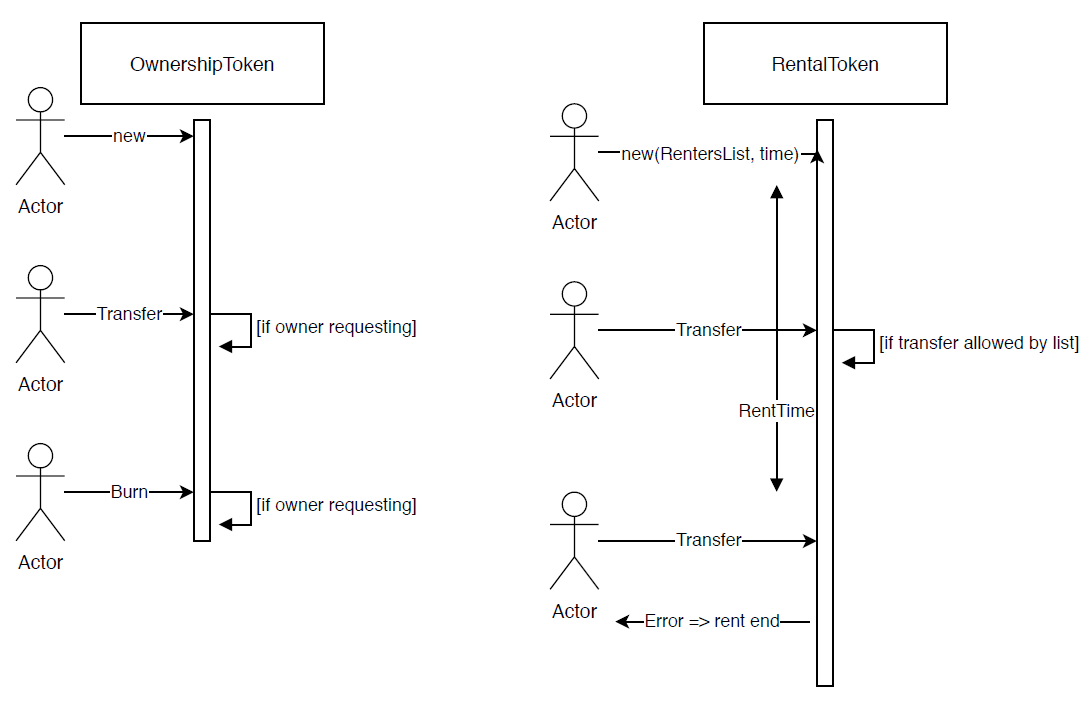
**Part 1: Modeling simple Ownership asset and Rental asset (stand-alone implementation)**

First, we implement the tokens of ownership asset and rental asset without linking them. Each owner can create an ownership token and a rental token independently. In this step the fact that someone owns an object and someone else rent it are completely independent and have no influence one on the other.

The implementation of this step can be found under the folder Step1, where the ownership asset is implemented by the contract called OwnershipToken and the rent asset is implemented by the contract called RentalToken.

To this implementation we join a script running a test on the tokens in the file TestTokens.js under the folder Test.

We can illustrate the implementation of this part with a sequence diagram explaining the different operations on the tokens:



*Figure 1. Sequence diagram illustrating part1*

**Part 2: Modeling static implementation of inter-token dependencies**

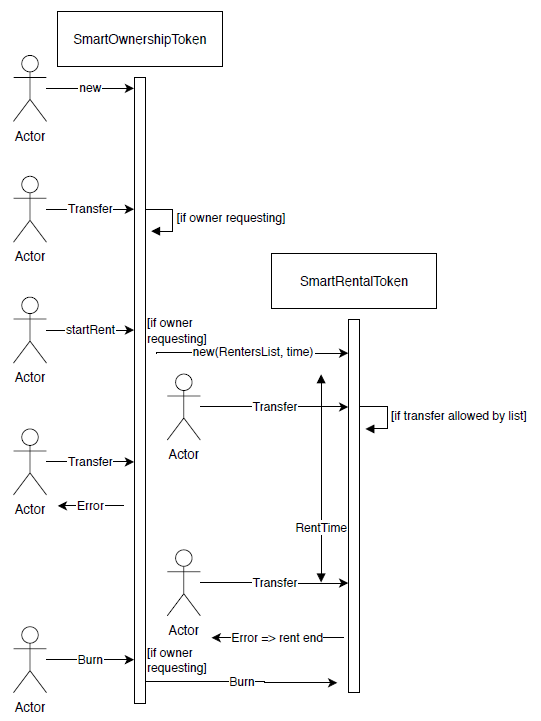
The second part of the project consists on implementing inter-token dependencies. Unlike part 1, now the tokens are ‘aware’ one of the other and they are linked together. An owner of an object will create a SmartOwnershipToken to represent this ownership asset and then if he wish to start a rent he will use the function startRent of this token. This function will create a SmartRentalToken which will be linked to the SmartOwnershipToken.

The fact that now the tokens are dependent one of the other allows us to impose some rules that a contract has to respect:

* A rent can be allowed only if it is created from an owner (meaning someone already has owning on the object) **//LIMITATION-MISSING**
* Only the owner of an object can decide to start a rent
* At starting of the rent, the owner must provide a list of addresses that are included in the rent (then each address will be able to keep the rent)
* A rent is defined within a period of time: this time is calculated in minutes and starts when the owner uses the function startRent. After this time is finished the rent cannot be transferred anymore and it will go back to the owner automatically. During the time the rent is valid, a renter can give the token to someone allowed by the list of renters.
* If an object is rented (the rent is still valid according to the time) it is forbidden to give the object (the ownership of the object)
* Even an owner is not allowed to take back the rent before the time of the rent ends (the only way for the owner to get the rent back during this time is if the actual renter transfers him the rent)

This part of the project is more realistic than the first part since it reflects the reality in a better way.

We can illustrate the implementation of this part with a sequence diagram explaining the different operations on the tokens:



*Figure 2. Sequence diagram illustrating part2*

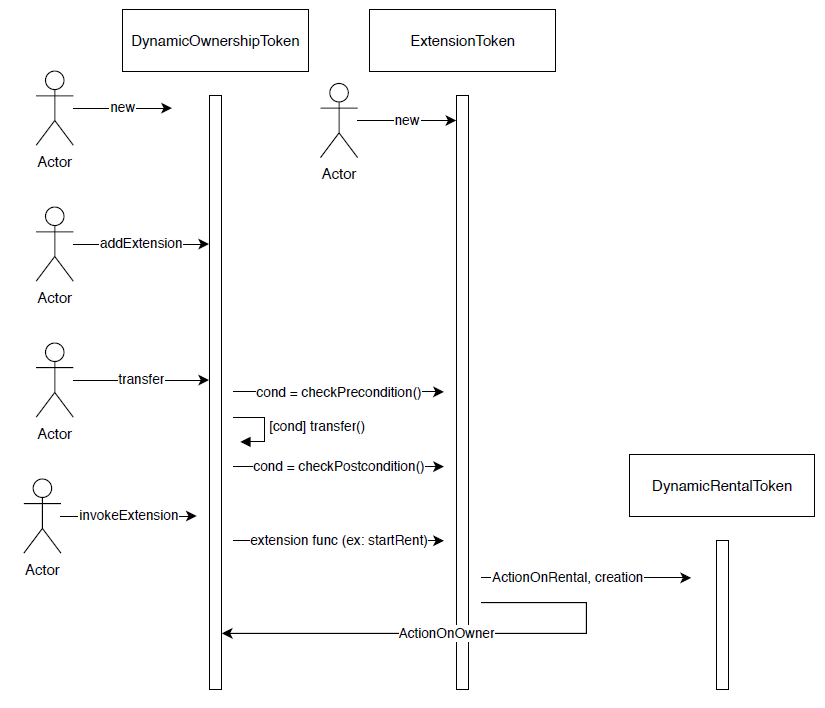
**Part 3: Modeling dynamic implementation of inter-token dependencies using extensions**

Finally, in this part we wish to implement a dynamic inter-token dependency.

We want to implement a mechanism that will allow us to add functionalities to an OwnershipToken. This mechanism includes two main behaviors:

* In the reality, we don’t always want to have rent allowed on an object, we can have an object that is not rentable (but is only sellable). In order to solve this problem, in this part we change the rent mechanism to be an extension of the ownership. At the beginning an ownership token is only sellable and there is no option of rent on him, and if we want to add rent to it we add an extension to it. An extension is a contract of type Extension that implements functions that are in charge of creating a RentalToken and handling the rent. In the same way we can define more options on an OwnershipToken.
* As we explained above, the ownership asset is implemented by an ERC20 token which follows some rules, including functions as transfer that are mandatory. What we wish to do in this part is to be able to add preconditions and postconditions to each action that is done on the token. These pre/post condition will be defined in the extension file (as function) and then when we’ll add the extension to the OwnershipToken it will automatically add the running of these functions before/after the relevant action.

Each extension in this part is created at the running time, so we can have a situation where a OwnershipToken exists and then we write the code of an extension compile it and then we add this extension to the OwnershipToken, without needing a rebuild or a recompilation of the OwnershipToken.



*Figure 3. Sequence diagram illustrating part3*

**Steps to run each part:**

* Create a new directory => mkdir step1
* In this directory run the following commands (in the command prompt):
  + truffle init
  + npm init -y
  + npm  install -E openzeppelin-solidity
  + npm install truffle-assertions
* Now the current directory has several folders: contracts, migrations, test and node\_modules
* Add in each directory the files that you can find under the same directory in the Step1 folder
* Change the content of the file truffle-config.js with the one on the git repository (this file must be located in the directory step1)
* In order to open visual studio, in the directory step1, run the following command: code .
* In order to run the test of the contracts run in the terminal the following command: truffle test --network ganache

(important: we run the test using the ganache network since we are using the accounts provided by ganache in the test. So it is important before running the test that you open ganache application an start a new Ethereum workspace)

* If you wish to test the contracts on you own you can:
  + First migrate the contracts with the command: truffle migrate --compile-all –reset
  + Then open truffle console with the command: truffle console --network ganache