# Modeling tokens on blockchains
# Technion - IBM industrial project
# Summary booklet

Tutors: Dany Moshkovich and Artem Barger

Students: David Valensi and Naomie Abecassis

1) Introduction
2) First part: Modeling simple Ownership asset and Rental asset (stand-alone implementation)
3) Second part: Modeling static implementation of inter-token dependencies
    a) Rules of inter-dependencies
    b) Sequence diagram
    c) Class diagram
4) Third part: Modeling dynamic implementation of inter-token dependencies using extensions
    a) Motivation
    b) What is dynamic inter-token dependency?
    c) Developer manual for this mechanism
    d) User manual for this mechanism
    e) Summary
5) Installing the environment
6) Steps to run each part of the project
7) Explanations for developing new smart contract
8) Example how to use the ganache console

## 1) Introduction

The goal of this project is to model inter-token dynamic dependencies.

The way to succeed is to gradually improve ownership and renter tokens to get close to reality behavior using blockchain ERC20 tokens.

- ERC20 standard

  ERC20 is a popular standard for Ethereum tokens supported by community. In these standards we denote mandatories functions and optional functions. Each ERC20 token implements 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.

- Ownership & Rental

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

  A rental asset token represents 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 transfer 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 contracts 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).

The main goal of this project is to reflect the reality of the ownership of an object using smart contracts in order to make secured transactions. Thus, in this project you will be able to follow the evolution of the implementation through three main steps where each step become more realistic that the previous, getting to reflect the reality in a better way.

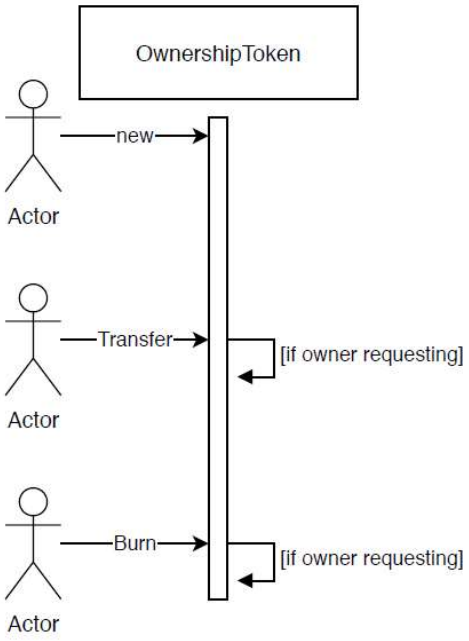
## 2) First part: Modeling simple Ownership asset and Rental asset (stand-alone implementation)

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:



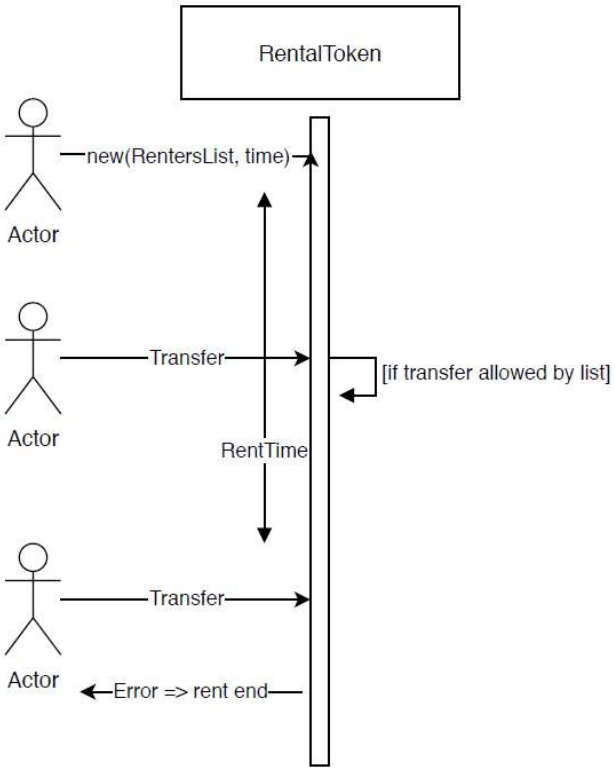



*Figure 1. Sequence diagram illustrating part1*

This implementation reflects a reality where we can rent or own an object, but the rent and the ownership are independent. This is a real limitation. Indeed in the reality two actions made on the same object can have influence one on the other and that is why in the second part we'll implement a dependency between token.

## 3) Second part: 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. The owner of an object will create a SmartOwnershipToken to represent this ownership asset and if he wishes 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.

### *a) Rules of inter-dependencies*

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

- 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 time interval: 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/transfer objet ownership.
- 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)
- An owner can decide to burn a rental token after the time allowed to the rent is over.

These rules are set before deploying the contracts to the blockchain and they allow to handle the dependencies between tokens in a legal way. The rules that we set are logical rules that we thought will reflect well the reality. And thus in this part of the project we get a more realistic implementation of ownership and rental assets than in the first part.

### *b) Sequence diagram*

In this part, someone wishing to create a rent for an object will first create a SmartOwnershipToken for this object, then assign to him a rent using the function startRent which will create a new SmartRentalToken. This function get as parameters a list of addresses that are allowed in the rent and a time defining the rent. Once the rent is set different accounts can freely interact with the SmartRentalToken following the rules defined above. We can illustrate the activity flow of this part with a sequence diagram explaining the different operations on the tokens:

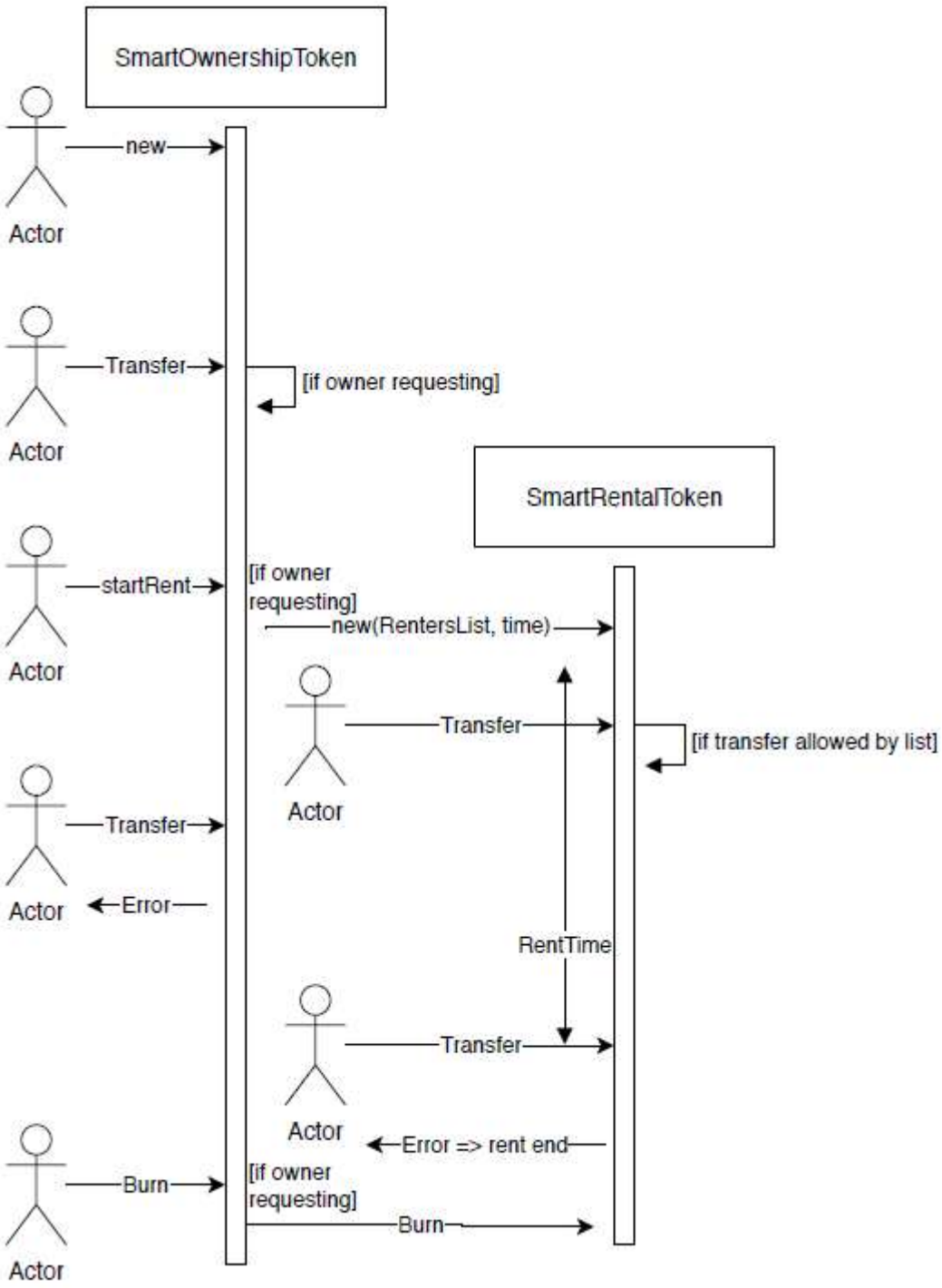



*Figure 2. Sequence diagram illustrating part2*

### c) Class diagram

As we explained above, this part of the project introduces static dependencies between the tokens. These dependencies are implemented using association: the SmartOwnershipToken stores in his parameters the address of the SmartRentalToken and in the same way the SmartRentalToken also stores the address of the SmartOwnershipToken. We can illustrate the mechanism by a class diagram:

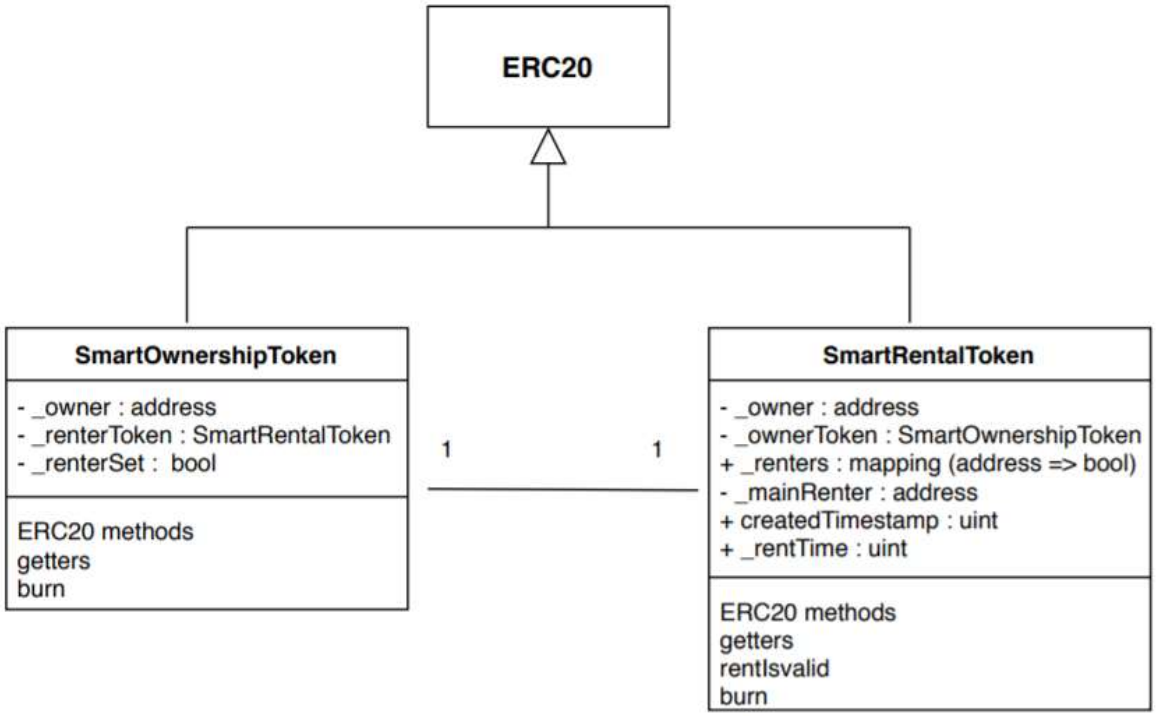



*Figure 3. Class diagram illustrating part2*

Storing a SmartRentalToken in the SmartOwnershipToken allows us to be able to check the rules before and after each transfer (or other action that is done on the ownership token or the rental token). Moreover this implementation shows well that only one rent contract is allowed for a given object, such as in the reality (you couldn't rent twice the same object).

However, this implementation has limitations since in reality we don't necessarily want to allow the rent of an object. And more than that, we wish to be able to add new functionalities to an object even after it has been deployed to the blockchain. This explains the need of part 3, in which we want to change the inter-token dependency to be dynamic and not static such as this part.

## 4) Third part: Modeling dynamic implementation of inter-token dependencies using extensions

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

### *a) Motivation*

As explained above a blockchain is a public database where data can be stored and we can make actions on this data in a secured way. The security that a blockchain provides makes the data immutable. So Blockchains have an important limitation in their ability to change. However, in real world, things can evolve and change and that is why, if we want to reflect the reality we need to implement a mechanism that will allow us to introduce new features to an object, to change rules of an object freely, without any need of changing what already exists.

### *b) What is dynamic inter-token dependency*

That's why in this part we 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 to rent it. 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 the extension will be added to the OwnershipToken it will automatically add the running of these functions before/after the relevant actions.

Each extension in this part is created at the running time, so we can deploy a OwnershipToken and later, write the code of an extension, compile and deploy it and add this extension to the OwnershipToken. All this without a recompilation or redeployment of the OwnershipToken.

## c) *Developer manual for this mechanism*

Let us explain how works this mechanism. First, we define a contract named ExtensionInfo that contains the same parameters as DynamicOwnership and data structures that store information about an extension. These information are the number of functions contained in the extension and for each function a struct Extension is added (containing the name of the function that is being extended, the signature of the function extending and the type of extension).

There are 3 types of extension functions:

- Precondition: the function is linked with a certain ERC20 method and will be called at the beginning of the method. The requirement is that this pre-condition passes in order to be able to compute the ERC20 method.
- Postcondition: the function is linked with a certain ERC20 method and will be called at the end of the method.
- Invocation: the function can be invoked by a DynamicOwnership if the corresponding extension has been added to the token.

*Every Extension has the same template:*

To create a new extension you must create a new smart contract Extension that **inherits from ExtensionInfo**. ExtensionInfo is the common skeleton of any extension of OwnershipToken: it has the same state variable layout as the OwnershipToken and has the required data structures to extend functions. In Extension contract, you will add the code of the functions you wish to add dynamically. Then for each function like that, you must add it in the constructor as an Extension struct with its parameters (defining the function extended, the signature and the type of extension) and you must increase the number of extensions by 1.

*Delegate Call on Extension functions:*

As you'll see in the code of the OwnershipToken, the extension functions are called using the delegatecall() Solidity function. This function:

- Identifies inside a given contract, the function with a given signature.
- Perform a call to it
- Doesn't switch context – state variables are from calling contract and not from called contract. It's the reason why we want to keep the memory layout from Ownership in Extension (see here).

When creating a new extension it is important that you don't add any parameters to the contract since this will change the state variable layout and the mechanism of invoking function may not work fine . If you need to store new variables you can store them in the mapping extensionsData such as explained below.

Let us illustrate this with a scheme:

*How to extend functions or dynamically add new ones ?*

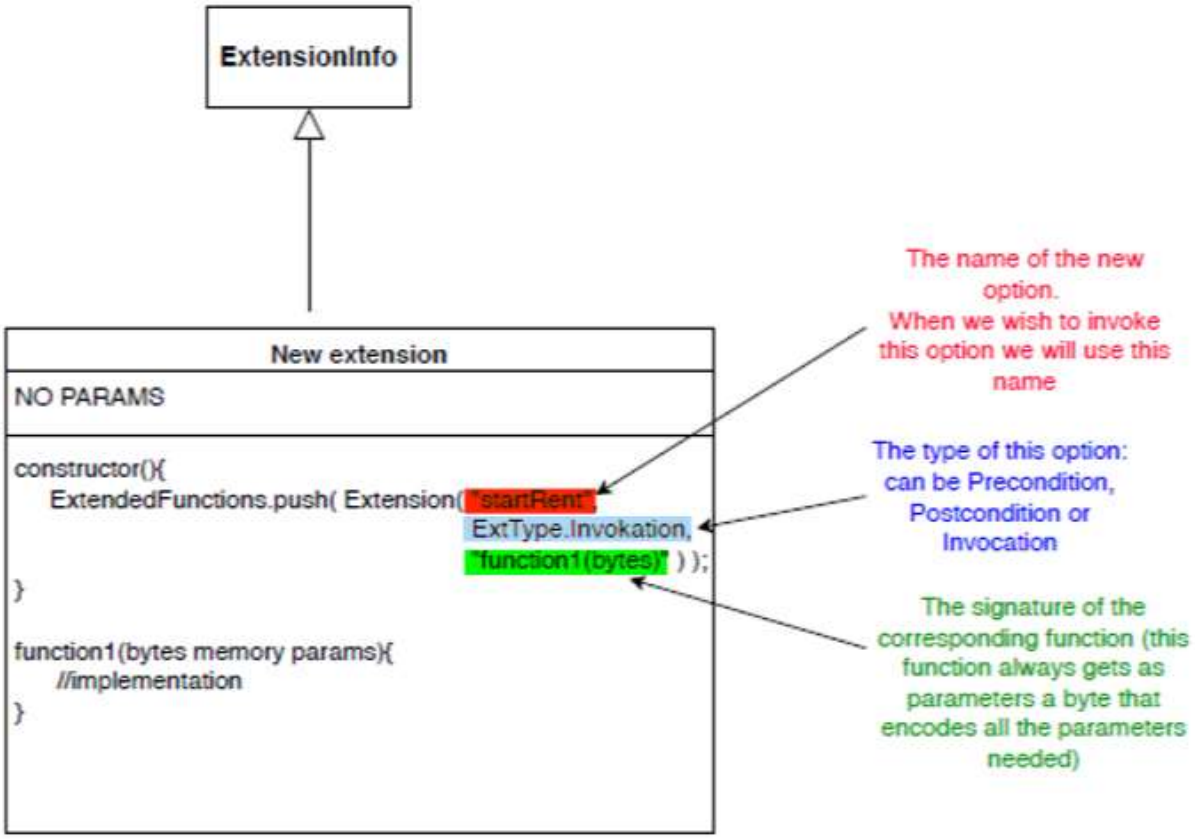



*Figure 4. Illustration for adding a new option to an extension*

For each pre-condition or post-condition we wish to run before/after an ERC20 method we define a function in the extension and we add it in the constructor in the same way as invocation. The following scheme illustrates a pre-condition to the function transfer:

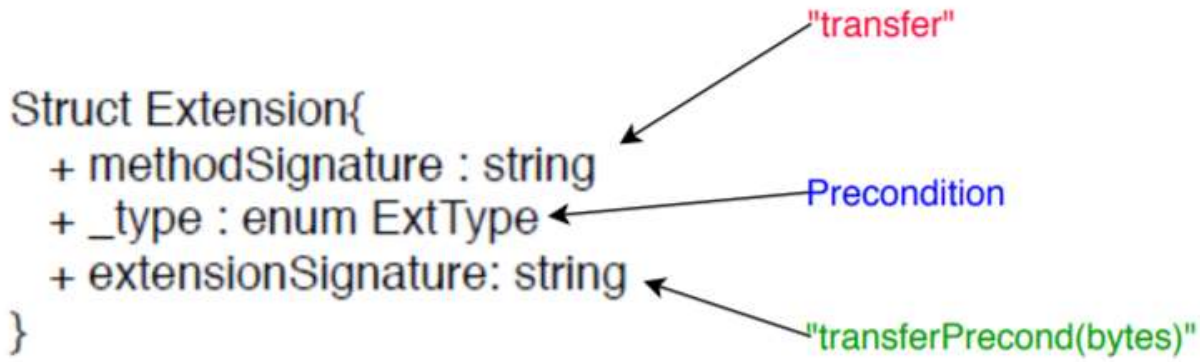



*Figure 5. Illustration for adding a new precondition to transfer function*

*How Extensions will be able to read/write data during delegatecall if the context is the caller context ?*

The implementation of the dynamic inter-token dependencies allows an extension to store data in the DynamicOwnership memory: the data is stored in the mapping extensionsData. In order to keep data coherency, we define the following convention: **data named b that is used by the extension A will be stored in extensionsData["A_b"]**. This allows any extension to use the data of a DynamicOwnership token when being invoked. For example, the data renterSet used by the extension Rent will be stored in extensionsData["Rent_renterSet"]. Every data is always stored in an encoded form of bytes.

*How called functions could have parameters ?*

Moreover, each function that is of type Invokation can receive parameters that are encoded as bytes. **When adding a function to an extension we define as a convention that the function gets only one parameter that is of type *bytes***. Thus, any function in an extension must have a signature as: *function_name*(bytes memory params). We make a difference between:

- Precondition/PostCondition functions: the byte received as a parameters will **always** contains the parameters passed to the extended method. For example if the function transferPreCond(bytes memory params) is extending the transfer ERC20 method, params will contain the encoded version of the parameters received by the transfer ERC20 method (which are (address recipient, uint amount)).
- Invocation functions: each developer defines which parameters an invocation function should get. These parameters are received in an encoded form (in the parameter *params* of type *bytes*). You can see example of how to encode params in Step3 test script in the Github repository or below in the User section. The developer must decode params in order to get the actual values of the parameters. For example, if you wish to write a function in the extensions that will receive as first parameter an *address* and as second parameter a *uint*, you must define:
  *(address param1, uint param2) = abi.decode(params, (address, uint) );*
  Then, you can use param1 and param2.
  To follow this implementation, when invoking an extension, you must pass this bytes parameter as you defined the parameters of the function. If the encoded parameter types are not corresponding to the expected parameter typed, the behavior is not defined.

We can illustrate the implementation of the mechanism by the following class diagram:

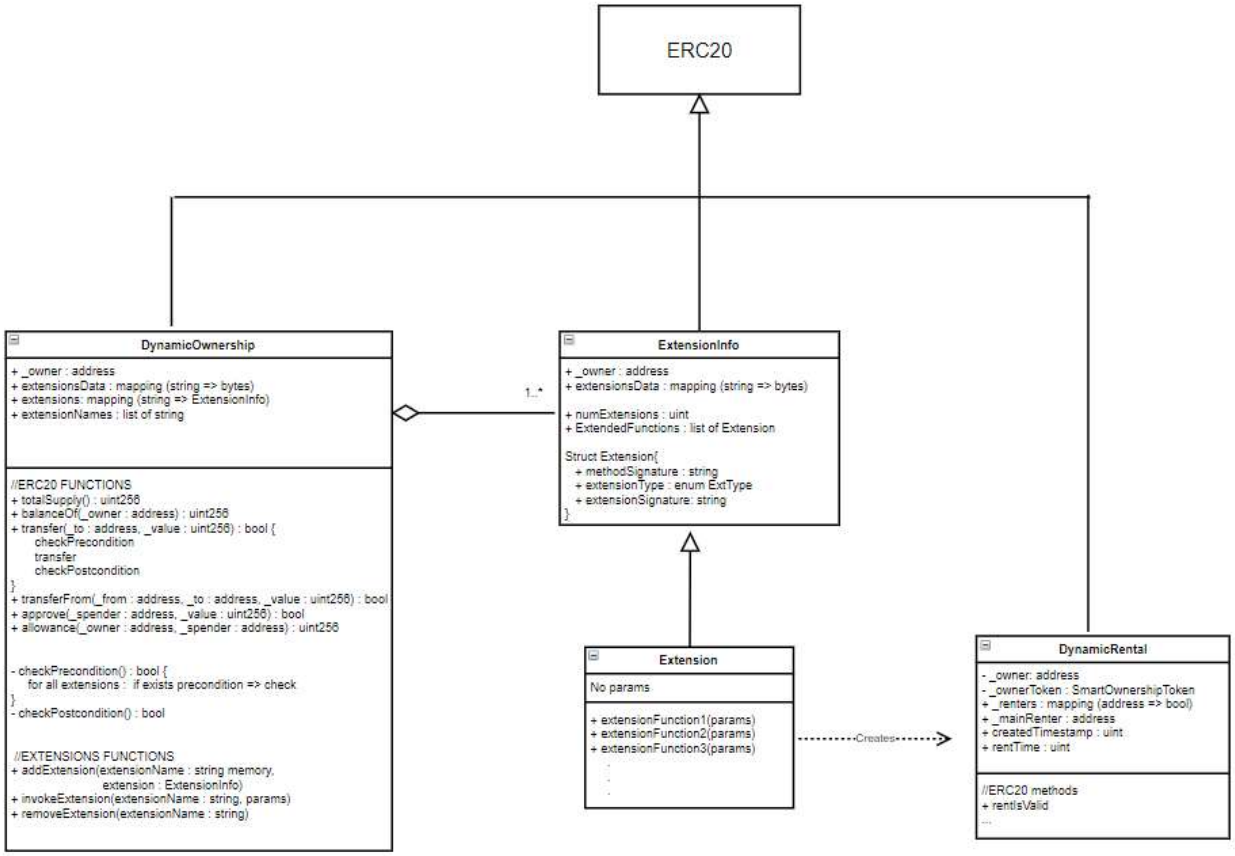



*Figure 6. Class diagram illustrating part3*

## *d) User manual for this mechanism*

Once all the code is written in a good way we can describe the general flow of the activities in order to add an extension. First you need to create an instance of a DynamicOwnership token, and an instance of the extension you prepared. Then, use the function addExtension of the DynamicOwnership token to add the current extension to the token. This function gets as parameters the name of the extension (this name will be used later to invoke options) and the address of the extension that you wish to add. Once the extension is added you can invoke functions of the extension using the function invokeExtension. This function gets as parameters the name of the extension where the function can be find, the signature of the function you wish to invoke and the bytecode of the params you wish to pass to the function. We define that if the function does not exist in any extension added to the token, the invokeExtension function will return false.

In the same way, once the extension has been added to the token, when the token will be requested to run an ERC20 method, it will first check all pre-condition functions of the extension (corresponding to this ERC20 method), then compute the core of the

method, and then run the post-condition functions of the extension (corresponding to this ERC20 method).

We can describe this flow of activity with a sequence diagram:

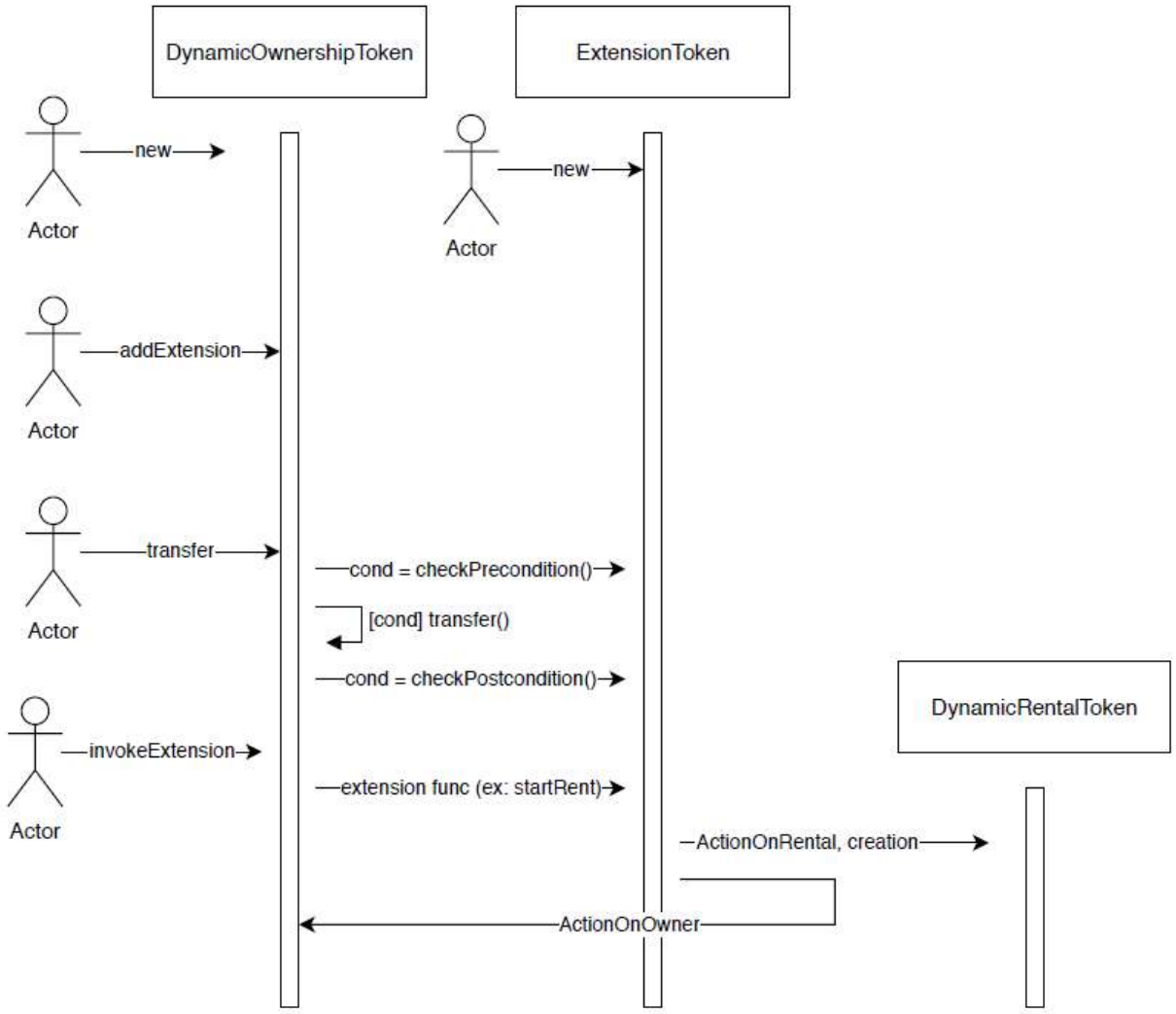



*Figure 7. Sequence diagram illustrating part3*

As explained above each function of an extension gets a parameter of type bytes that contains an encoding of the parameters of the function. For example, if the function is defined to receive as parameters an address (denoted parameter1) and a uint (denoted parameter2) in order to invoke the function we need to pass the argument:

abi.encode(parameter1, parameter2)

Then in the function a mechanism is implemented in order to unpack these arguments.

Moreover, as defined above each extension can add data to the DynamicOwnership and this is done by storing this data in the mapping extensionsData. As a convention we

define that the data named b added by an extension named A is stored in extensionsData[“A_b”]. For example, the variable named renterSet defined by Extension will be stored in extensionsData[“Extension_renterSet”]. In order to get access to this data you can use the DynamicOwnership method getMapElement which get as a parameter the name of the data as defined by the convention (as a string) and returns the encoded data stored. In order to decode the data you must use the function abi.decodeParameters.

### *e) Summary*

As explained above, the mechanism that allows us to add extension and invoke new function from an existing contract is implemented using four main functions:

- addExtension: add a new extension to the contract
- invokeExtension: invoke the call of a function defined in an extension that has been added to the current contract
- checkPrecondition: this function is called before every method of ERC20 in order to check that the precondition assigned to the actions are fulfilled. This function should not be called by a user and that is why it is define to private.
- checkPostcondition: this function is called at the end of each method of ERC20. This function should not be called by a user and that is why it is define to private.
- removeExtension : remove extension and its functionalities.

To conclude, this last part of the project reflects the reality in the best way possible since we can now add functionalities to an existing contract representing an object and remove them. The mechanism implemented here gives us the ability to handle an ownership asset in a very flexible way. Even though there are some limitations to this implementation since the extension token must contain the same state variables as the extended contract (including his parents) and when invoking an extension the errors thrown don’t bubble up and finally the invoked functions of the extension all have the same parameters (bytes memory).

## 5) Installing the environment

1) Download Nodejs (this will install npm too) with the following link https://nodejs.org/en/ and install it.

(After installing it you can check the installation version by writing the command "node -v" in the command prompt and "nmp -v")

2) Install the IDE: we choose to work with Visual Studio since it is convenient to developp smart contracts.

- You can download and install it from https://code.visualstudio.com/

- Install Visual Studio extension: go to the extensions section and install these plugins

- Solidity
- Material Icon Theme

- Enable icon theme: select File -> Preferences -> Fill icon theme

3) Install truffle : npm install -g truffle (in the regular command prompt)

4) Install ganache, an application that is used in order to test the different contracts implemented along the project. Ganache gives us the ability to fire up a personal Ethereum blockchain that is used to run tests. You can download it with the following link: https://www.trufflesuite.com/ganache

## 6) Steps to run each part of the project

- 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
  - npm install -g ganache-cli
- Now the current directory has several folders: contracts, migrations, test and node_modules:
  - contracts: directory to store the smart contracts that are created
  - migrations: directory for deploying the smart contracts into the blockchain
  - tests: directory for testing the smart contracts
  - node_modules: directory that contains the library that we need (erc20…)

- 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

## 7) Explanations for developing new smart contract

If you wish to extend the implementations that was done in this project and create new contracts (for example new extensions for the third part), here are the steps to follow in order to add a new contract to the blockchain:

1) Add the code of your smart contract under the directory contracts/ let's say for the simple example NewExtension.sol

2) Add the test for NewExtension under the test/ directory.

3) Add a migration file that will deploy the NewExtension, named 2_deploy_contracts.js and define there the deployment of the NewExtension (the index is important because it defines in which order the compiler should look at the files)

4) Run the test with the command "truffle test" (in the command prompt). This command will compile the files and run the tests that are written for the NewExtension

5) Now you can deploy the smart contracts using ganache. First install ganache with the following command:

npm install -g ganache-cli

6) Open a truffle console with the command

truffle develop

7) For convenience you can open a second tab where you can see the log of the transactions. Run the command

truffle develop --log

8) In the first tab use the command "migrate" to deploy the contracts

9) You can see the transactions cache, the contract address, the account that it came from, the balance of that account...

## 8) Examples how to use the ganache console

- Instantiate a token OwnershipToken with a balance of 1 to the msgSender (msgSender is what's mentioned in the brackets: from accounts[9]):
  let owner = await DynamicOwnership.new("Name", "Symbol", {from:accounts[9]})
- let accounts = await web3.eth.getAccounts()
- To get the balance of a certain account:

let balance = await owner.getBalance(accounts[0])

- To make a transfer of the token to accounts[8]:
  owner.transfer(accounts[8], 1)
- To instantiate a new extension:
  let extension = await Extension.new()
- To add this extension to the owner token previously created:
  owner.addExtension("Extension", Extension.address)
  let params = await web3.eth.abi.encodeParameters(['address[]', 'uint'], [accounts, '3'])
  a.invokeExtension("Extension", "startRent", params)
  rentA = await a.getMapElement("Extension_renterToken")
- Now let's say that Extension contains the method named startRent. In order to invoke the extension:
  - First prepare the parameters the method should receive:
    let params = await web3.eth.abi.encodeParameters(['address[]' , 'uint'], [accounts, '3'])
  - Then invoke this method with the owner token:
    owner.invokeExtension("Extension", "startRent", params)
  - Now you can access the Rental token that has been created and stored in the mapping ExtensionsData of the owner token:
    (CONVENTION: a variable a of an Extension named B in the mapping extensionsData is stored under the string "B_a")
    let rentA = await owner.getMapElement("Extension_renterToken")
    //If there are few params:
    let addrRentA = await web3.eth.abi.decodeParameters(['address'], rentA)
    addrRentA = addrRentA['0']
    // OR if there's only one param:
    let addrRentA = await web3.eth.abi.decodeParameter('address', rentA)

    let renterToken = await DynamicRental.at(addrRentA)
  - Now renterToken contains a DynamicRental token and you can use it to make transfers of the rent or anything else. For example if you wish to get the balance of accounts[3] you can use:
    let balance3 = await renterToken.balanceOf(accounts[3])