# Modeling tokens on blockchains
# Developer manual

Tutors: Dany Moshkovich and Artem Barger

Students: David Valensi and Naomie Abecassis

# I. Step 1

As you can see in the github repository the contracts implemented in this part are OwnershipToken and RentalToken: for the moment they are implemented such that they are independent one of each other.

OwnershipToken: it is implemented as an ERC20 token and contains the same fields as the ERC20 token, nothing was added. Moreover the contract contains the mandatory methods of ERC20 token (more details about them in the Introduction of the summary booklet).

RentalToken: it is implemented as an ERC20 token and contains the following fields (in addition to ERC20 tokens):

```
mapping (address => bool) public _renters;
uint public _createdTimestamp;
uint public _rentTime; //expressed in minutes
```

- _renters represents a mapping of the allowed renters in the rent
- _createdTimestamp defines the time when the contract is created
- _rentTime defines the time allowed for the rent (calculated according to the created time of the contract). This time is expressed in minutes.

Moreover the contract contains the mandatory methods of ERC20 token: these methods are overrided in the code since we have to check some pre-conditions.

# II. Step 2

As you can see in the github repository the contracts implemented in this part are SmartOwnershipToken and SmartRentalToken: they are implemented such that one can create the other and they are dependant one of the other.

SmartOwnershipToken: it is implemented as an ERC20 token and contains the following fields (in addition to ERC20 tokens):

```
Address private _owner;
SmartRentalToken private _renterToken;
Bool private _renterSet;
```

- _owner keeps the address of the owner of the contract
- _renterToken represents the SmartRentalToken that corresponds to this ownership

- _renterSet is a variable that defines if there is a rent already set on this ownership or not (we want to avoid creating several rents for the same ownership)

Moreover the contract contains all the ERC20 methods (the methods are overrided since we add some pre-condition as explained in the summary booklet) and other new methods:

- getOwner() returns the owner of the contract
- getRentalToken() returns the address of the rental token associated with this ownership token
- hasRenter() returns if the current ownership contract is associated with a rent
- getThis() returns the address of the current contract (used in the rental token)
- startRent(address[] memory rentersList, uint rentTime) is the method used in order to start a new rent. The arguments provided to this method are a list of all the renters allowed in the rent and the time defined for the rent (time in minutes)

SmartRentalToken: it is implemented as an ERC20 token and contains the following fields (in addition to ERC20 tokens):

```
address private _owner;
SmartOwnershipToken private _ownerToken;
mapping (address => bool) public _renters;
address _mainRenter;
uint public createdTimestamp;
uint public _rentTime;
```

- _owner keeps the address of the owner of the ownership contract associated with this rent
- _ownerToken represents the SmartOwnershipToken that corresponds to this rent
- _renters is a list where we keep the addresses allowed to keep the rent when the rent is valid. This list is provided when the rent is started and is not modifiable once the rent has started.
- _mainRenter represent the address that has the rent “in the hand” at every moment
- createdTimestamp defines the time when the contract is created

- _rentTime defines the time allowed for the rent (calculated according to the created time of the contract). This time is expressed in minutes.

Moreover the contract contains all the ERC20 methods (the methods are overrided since we add some pre-condition as explained in the summary booklet) and other new methods:

- setRent() is a method used in the ownership token code in order to set the different fields of the rental contract. This method should not be used by external user and this condition is checked in the method.
- rentIsValid() returns whether the rent is still valid (according to the time)
- remainingTime() returns the time remaining for the rent
- burn() is the method that you should use if you wish to destroy a rent contract. This method will burn the contract only if the rent is over and can be used only by the main renter or by the owner of the ownership token

# III. Step 3

## 1. ExtensionInfo

ExtensionInfo is a contract that we created in order to facilitate the process of creating an new extension. In fact this is a general code that is meant to be used by inheritance when you want to create a new extension.

As explained in the summary booklet in order to make extensions work we implemented a way to share data and we used the delegate call method, and as mentioned this method has one limit that is the need of the same variable environment as the caller contract. So in order to be able to use it the contract ExtensionInfo must include the same variables as an ERC20 token. In addition to these variables it also contains a mapping of the data of the extension, named extensionsData. As we explained in the summary booklet, extensionsData is a mapping from string to bytes that stores variables names and their value in form of bytes. 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”]. Moreover in this file we define the struct of an Extension:

```
struct Extension {
    string  methodSignature; // The method which is extended in Origin token
    ExtType extensionType;
    string  extensionSignature; // The method which is called on delegatecall
}
```

Where ExtType is either Precondition or Postcondition or Invokation, in order to  make a difference between the different types of extension.
Thus when defining a new extension, each function is linked to an Extension struct and each function will be added to the list of extensions in the constructor of the implemented extension.
The next fields of this contract are:

- ExtendedFunctions, an array of Extension struct, where we store all the functions of the extension. As explained above each function is represented by an Extension struct, and we store them in an array.
- numExtensions, where we store the number of functions that are defined in the current extension. This variable is used when we need to iterate on all the functions of the extension (for example in order to check all the different preconditions checked).

It is important to recall that the contract ExtensionInfo will be used in the future (see 2) in inheritance such that each new extension that you wish to create will contain all these fields.
The methods implemented in this contract are:

- getnumExtensions returns the value of the variable numExtensions
- getData gets a string as parameter and returns the data stored for that string in the extensionData mapping (this string corresponds to a variable stored in the shared memory). This method returns a bytes type and the data must be decoded in order to use it.
- initialData gets a name of variable and a value (bytes) and stored this value in the shared memory. This function is responsible of entering a new value in the shared memory.

This contract must be used when you wish to create a new extension: each new extension created must inherit from ExtensionInfo and must be implemented as explained in part 2.

## 2. Extension

The contract Extension is an example of the implementation of a new extension. In the scope of this project we implemented an extension that contains functions to start a new rent and function that are pre and post conditions to actions made on ownership token and rental token.

This contract contains:

- The functions of the extension such as transferPreCond(bytes memory params) or startRent(bytes memory params) and more. Each function like this receives as parameter a bytes memory that contains the different parameters that the developer needs in the function, and these parameters are encoded. The responsibility of the developer that writes the code is to decode this parameter in the right way (ie according to the parameters that are encoded) and it is also the responsibility of the user to pass the parameters (as the developer defined) in the right way (ie, same types and same order and encoded). In these function you can access the shared memory from the mapping extensionsData.
- The constructor where all the extensions functions must be added to the array ExtendedFunctions. As explained above, each function will be associated with a struct Extension and then this struct must be added to the array ExtendedFunctions. If a function is implemented in the file but is not added in the constructor, then it is like this function doesn't exist in the eyes of the extension.

It is important to recall that your extension must inherit from the contract ExtensionInfo.

## 3. DynamicOwnership

This contract represent an ownership contract that can use extensions. The fields added to this contract are:

- extensionsData, the shared memory with the extensions
- extensions, a mapping that contains the extensions (according to name) that will be add dynamically by the user
- extensionNames, an array that contains the names of the extensions that are dynamically added

The main differences between this contract and the one of step 2 are:

- you can add extension dynamically: this is done using the function addExtension that receives the name of the extension and the extension itself (its address)

- you can remove extension dynamically: this is done using the function removeExtension that receives the name of the extension you wish to remove
- you can invoke an extension: this is done using the function invokeExtension that receives the name of the extension, the signature of the function you wish to invoke and the parameters passed to this function
- the methods of ERC20 tokens (such as transfer, burn…) are implemented such that before making the action, preconditions are checked (invoking all the preconditions defined in the different extensions added). Only if these preconditions are valid the action will be made. In the same way postconditions functions are checked once the action is done.

In order to iterate on all the extensions that have been added the invokeHelper method checks for a certain type of extension (we recall precondition postcondition or Invokation) all the extensions contained in the array extensionNames.

## 4. DynamicRental

Finally, DynamicRental is implemented such as in part 2 since the difference in this part is implemented in the code of the DynamicOwnership. Indeed DynamicRental will be an extension for the code of the ownership and thus it must implemented in a regular way, with no dependency of the existing contracts.