FINAL PRACTICE

Software Development, group 15.

MARTA BENITO SERRANO, 100429066

SEBASTIAN IONUT STOEAN, 100429139

1. Introduction

In this report, we will expose an explanation of our design choices used to implement the requirements in the statement of the final practice. Moreover, we also provide all the techniques that we have added for testing the new method revoke_key().

2. New implementations

For this project, we were asked to implement multiple new functionalities to the given code. The first two functional requirements regard the modification of existing methods to perform a new action. Concretely, the first requirement states that the value *access_code* must be stored in each access request, and that this code must be the key used for searching in the storage file. Moreover, the second requirement demands that a storage log must be kept every time that a door is opened. Finally, the last task is to create a new method called *revoke_key()* which is used to deactivate a key before it has expired.

2.1 Storing the Access Code

The first functionality we had to implement was storing the access code for each access request. In the provided code, when inserting an access request into the store json, we would not save the access code. This makes little sense, as we are not saving in the archive the output of the function and, therefore, losing this output. Moreover, this output can be then used to request an access key, and so it is useful to be able to know if that person had requested access in the first place.

In order to implement this, we didn't have to modify a lot of code. When saving the access request, the coding will save all the *self* variables of the object. We have taken advantage of this and added a variable to the object in order to store the Access Code. We have utilized the code from the previous property method to give the value to this new variable and created a new method for the property of access code, which allows the program to return this new variable.

The last step in this section was to modify the rest of the coding so that it utilizes the access code as the main variable for the access requests. The code had to search the archive using it instead of the dni.

2.2 Open door log storage

The second required functionality was to create a json file to store each time the door is opened. This storage will only contain the key number used to open the door and the timestamp of this operation. This log is useful if we want to control who accesses the building and have a record of this.

In order to implement this, we have created a new class called AccessLogStore, which inherits the JsonStore class. This new class will be in charge of creating and managing the log archive. This new class has a modified method in order to execute insertions into the log, this is because the parent's method does *item.__dict__*, which does not work when the item is a list.

Given that this is a storage class, we have implemented the singleton design pattern to make sure that there is a unique instance of this class. We also added tests in the file *test_singleton_access_manager_tests.py* to make sure that the singleton was implemented properly.

2.3 Revoke Key function

The last implementation we had to make was a new function called revoke_key. This new function will take as input a json file containing a key, the type of revocation and the reason behind it. From this input, the function will revoke the key and make it invalid or remove it from the storage, depending on the type of revocation.

For this functionality, we thought of two different approaches: the first approach was creating a new json file which would store the revoked keys, this store would be then read by the function in order to see if the key was already revoked or not; the second approach was to add to the object AccessKey a new attribute called "revoked", which would be true if the key was revoked and false if not. For both approaches, the way to manage "final" revocations was the same, deleting the key from the archive all together. After discussing these two ideas, we decided to develop the second approach, as we thought it would be easier to implement. However, this ended up not being the case, as we chose the complicated path.

As we have already said, in order to implement this functionality, we have added a boolean attribute called *revoked* to the class AccessKey. This attribute could only be accessible with a @property method, and modified with the @revoked.setter method.

In addition, we have created a new way of getting the Key in the access manager class, using a method called *create_key_for_revoke()*. This method takes a json file as input and executes the following actions.

First, it will read the file and check the validity of each input variable using attributes. Once the attributes have been checked, it will use the function revocation_find_item to search the key in the storeKeys.json. This function will raise an exception if the key has already been revoked or if it does not exist in the archive.

Once the key has been found, the function will create a new AccessKey object with the same data and will check if it expired using the *is_valid()* method. If the key is expired, it will raise an exception, otherwise it will look at what the type of revocation is.

If the type of revocation is *Final*, then the function will erase the key from the storeKeys json using the method *delete_item()*. On the other hand, if the revocation is *Temporal*, it will use the method *change_revoke()* to change the revoked attribute of the key. Finally, the function will return the AccessKey object created.

3. Testing for Revoke Key.

In order to test this functionality, it is very useful to use syntax analysis since we are receiving an input file which must meet a specific structure. This is the first technique that we have used. In the following sections we provide the grammar that we have created along with its derivation tree. Moreover we decided to reinforce the tests of this method by using equivalence classes and boundary values. Finally, to ensure a good coverage of all the branches, we used the testing method of structural analysis.

3.1 Syntax Analysis.

In this section we will analyze the syntax the input file of the function needs to have in order to be valid. To do this, we have created a grammar for this file, which represents the correct format the file must have. Moreover, from this grammar, we have drawn a derivation tree in order to further visualize the testing cases we had to implement.

3.1.1 Grammar.

File ::= Begin_object Data End_object
Begin_object ::= {
End_object ::= }

Data ::= Field1 Separator Field2 Separator Field3
Field1 ::= Label_Field1 Equals Value_Field1
Field2 ::= Label_Field2 Equals Value_Field2
Field3 ::= Label_Field3 Equals Value_Field3

Separator ::= ,
Equals ::= :
Quotation ::= “

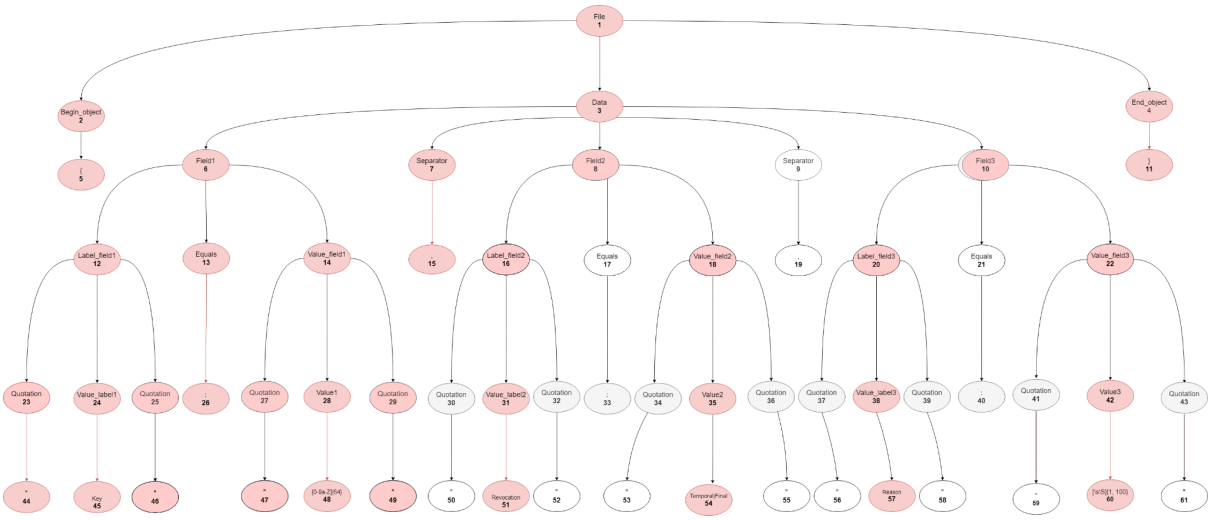
Label_Field1 ::= Quotation Value_Label1 Quotation
Value_Label1 ::= Key
Value_Field1 ::= Quotation Value1 Quotation
Value1 ::= [0-9a-z]{64}

Label_Field2 ::= Quotation Value_Label2 Quotation
Value_Label2 ::= Revocation
Value_Field2 ::= Quotation Value2 Quotation
Value2 ::= Temporal | Final

Label_Field3 ::= Quotation Value_Label3 Quotation
Value_Label3 ::= Reason
Value_Field3::= Quotation Value3 Quotation
Value3 ::= [\s\S]{1,100}

3.1.2 Derivation Tree.

Observation: The difference in colour of the nodes exists in order to show those nodes that have been implemented for testing (red) and those that were not (white). As it can be seen, we have tested the majority of the nodes in the grammar.



Note: We have included a copy of this image in the folder *docs* of the github repository.

3.2 Equivalence classes and boundary values.

The syntax analysis does not provide enough testing to check if the value for the reason contains a valid length (it should contain 100 characters max).

Equivalence classes:

CRITERIA/DATATYPE	IDENTIFIER	DESCRIPTION
Reason containing number of characters in range [1, 100]	EC_V_01	Reason containing the value "Change of residence."
Reason containing < 1 characters.	EC_NV_02	Empty reason.
Reason containing > 100 characters	EC_NV_03	Reason containing more than 100 characters

Boundary values:

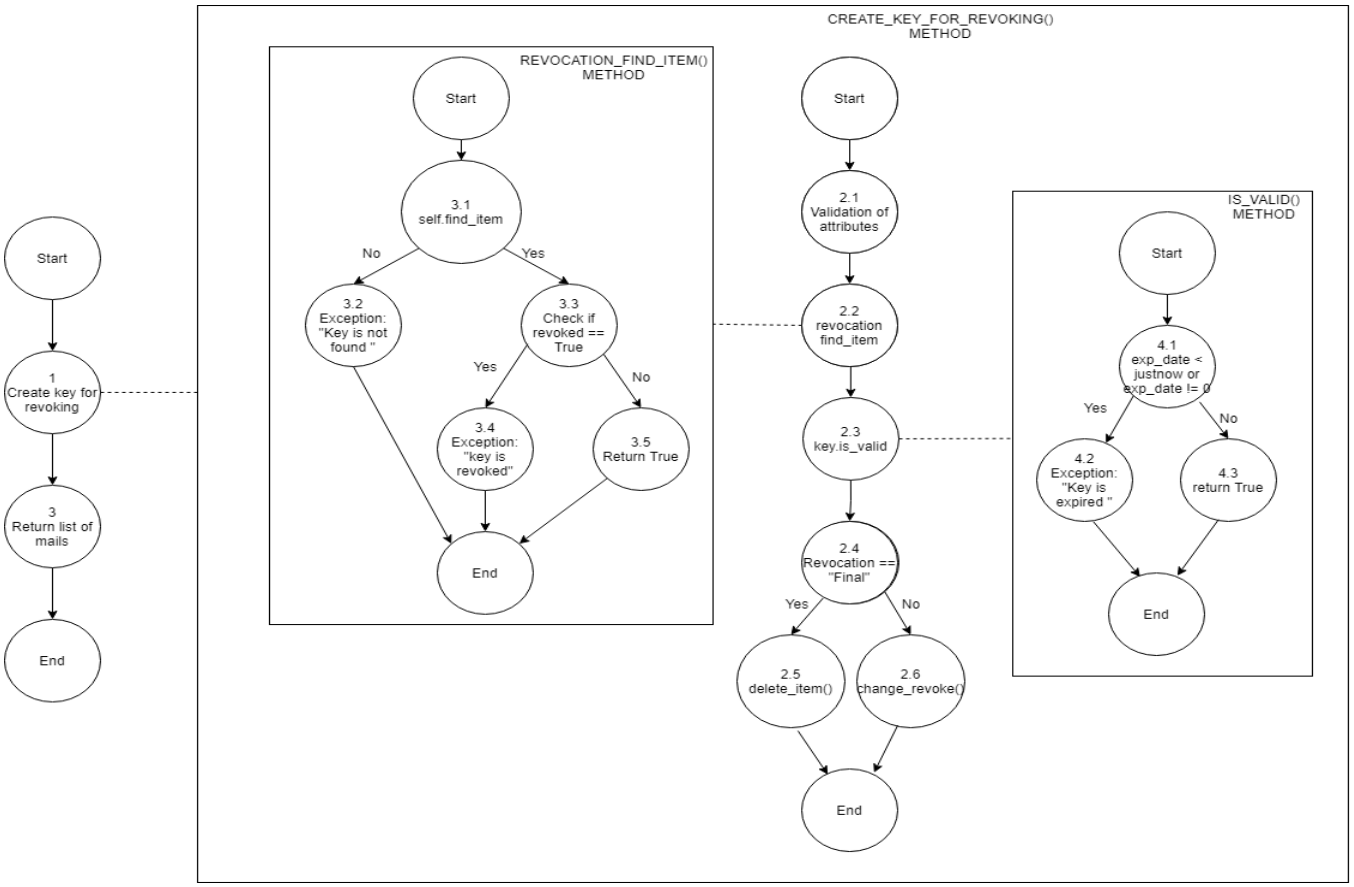
Observation: the lower bound of the equivalence class EC_V01_01 is already tested by syntax analysis (we check an empty string for the reason). This is why we will not implement this test again, even though we have it in this section, the boundary value tests for the lower bound will only be applied for the case in which the length of the string is 1. However we will observe all the possible cases of the upper bound. That is, we will test a length of the reason of 99, 100 and 101 characters.

CRITERIA/DATA TYPE	IDENTIFIER	DESCRIPTION	JSON FILE PATH	EXPECTED RESULT
BV Reason string 1	BV_V01_01	Lower bound. 0 characters (invalid)	rk_no_val_value3.json	"reason invalid" AccessManagementException.
BV Reason string 1	BV_V01_02	Lower bound. 1 character (valid)	rk_bv_lower_valid.json	List of mails
BV Reason string 99 characters.	BV_V01_03	Upper bound. 99 characters (valid)	rk_bv_upper_valid2.json	List of mails
BV Reason string 100 characters.	BV_V01_04	Upper bound. 100 characters (valid)	rk_bv_upper_valid3.json	List of mails
BV Reason string 101 characters.	BV_V01_05	Upper bound. 101 characters (invalid)	rk_bv_upper_invalid.json	"reason invalid" AccessManagementException.

3.3 Structural testing.

To ensure that the method works properly, we also decided to perform whitebox testing by using the technique of structural analysis. We have created a control flow diagram of the function and its auxiliary functions to explore all the possible branchings and paths that the code can take, and we have created tests to cover all the cases.

3.3.1 Control Flow Graph.



Note: We have included a copy of this image in the folder *docs* of the github repository.

The McCabe complexities are the following:

GRAPH	# EDGES	# NODES	McCabe complexity (E - N + 2)
Main	3	4	1
Create key for revoke	7	7	2
Revocation find item	8	7	3
Validating key	5	5	2

3.3.2 Basic paths and definition of test cases.

ID TEST	NODES COVERED	OBSERVATIONS
RF4 st1	1, 2.1, 2.2, 3.1, 3.2	Key is not found in the find_item() method. Return exception: “key is not found”.
RF4 st2	1, 2.1, 2.2, 3.1, 3.3, 3.4	Key is found in the find_item() method but is already revoked
RF4 st3	1, 2.1, 2.2, 3.1, 3.3, 3.5, 2.3, 4.1, 4.2	Key is found in the find_item() method, not revoked but is expired and we return exception “Key is expired”
RF4 st4	1, 2.1, 2.2, 3.1, 3.3, 3.5, 2.3, 4.1, 4.3, 2.4, 2.5, 3.	Key is found in the find_item() method, not revoked and it is final, so it is deleted from the storage json and return list of mails.
RF4 st5	1, 2.1, 2.2, 3.1, 3.3, 3.5, 2.3, 4.1, 4.3, 2.4, 2.6, 3.	Key is valid and can be revoked. Perform the change_revoke() to update the storage json and return list of mails.

REMARK: In practice, the tests RF4 st5 and RF4 st6 will not be implemented as a test because this path has already been explored by other previous valid tests (in syntax analysis and boundary values) and there is no need to repeat them.