SOFTWARE DEVELOPMENT

Computer Science

uc3m | Universidad Carlos III de Madrid

Guided exercise 3

Course 2022/2023

Group 15 - Class Group 89

31-03-2023

María Isabel Fernández Barrio

100472315

100472315@alumnos.uc3m.es

Víctor Carnicero Príncipe

100472280

100472280@alumnos.uc3m.es

INDEX

FUNCTION 1

1. EQUIVALENCE CLASSES AND BOUNDARY VALUES

These tests are better explained with the excel file as it identifies each individual test, its type and the inputs and outputs.

The inputs this function receives are: product id (string), order type (string), address of client (string), client phone number (string), client zip code (string). We perform 28 test according to the equivalence classes and boundary values method to check its correct functionality:

Tests 1-5: (product_id)
These tests check that the product id is a string compound made up of exactly 13 numbers, for example "8421691423220". Therefore, the boundary values are lower than 13, 13, and greater than 13; and the equivalence class test is checking that the id is a natural number, so we try to check if other characters as letters are accepted, as they should not.

If it is **TRUE**, the result must be an MD5 hexadecimal string.

Otherwise, different **EXCEPTIONS** are raised:
o **Product ID wrong format**: if the product id contains some non-numeric character.
o **Product ID not valid**: if the product id contains more or less numbers than needed, or does not correspond to the EAN13 format.

Tests 6-9: (order_type)
Check the order type: it must be "premium" or "regular", whether in uppercase or lowercase, but no other value is accepted.

Equivalence classes and Boundary values:
We check that the order_type is Premium or regular, no matter the case.
We also check that it is not a number, specially we compare it to the decimal representation of the number in binary that the correct value would have using ASCII.

If it is not correct different **EXCEPTIONS** are raised:
o **Order Type wrong format**: if the order type is not premium or regular.
o **Order Type not valid**: if the order type contains is only made up of numbers.

Tests 10-15: (address)
Check that the address of the client is a string containing from 20 to 100 characters (boundary values will depend on the length of the string), including at least one space to separate the street, the house number, the country, etc (matching the equivalence class).

- Boundary values include the length of the address: less than 20, 20, 100 and more than 100 characters.

In these tests we also include one to check not only that there are spaces, but also that it separates two words, as it would be possible to have only one space at the start or the end, which should not count as valid.

If it is not correct different **EXCEPTIONS** are raised:
o **Address not valid**: if the address does not match with the specifications.

Tests 16-19: (phone_number)
Check that the phone number is a string of length 9 containing only numeric characters. If the input does not contain exactly 9 characters or any of its characters is not a number, an exception is raised.

- Boundary values tests performed: lower than 9 characters string, length 9 string, and greater than 9 characters string.
- The equivalence class tests will check that the input is a string of numbers, not any other type of characters.

The two possible **EXCEPTIONS** are:
o **Phone Number wrong format**: if the phone number is not made up of only numbers.
o **Phone Number not valid**: if the phone number length is not correct.

Tests 20-27: (zip_code)
Check that the zip code is a number of 5 digits between 01000 and 52999 (the boundary values tests are defined following this rule). If this is not true an exception is raised. If the input contains any character that is not a number, equivalence class tests will reject it.

- We define two boundary values tests: for the length (6 digits, 5 digits and 4 digits) and for the range (00999, 01000, 01000, 52999 and 53000)
- For the equivalence classes tests we check that the zip_code is a string made up of only numbers

The two possible **EXCEPTIONS** are:

o **Zip Code wrong format**: if the zip_code is not made up of only numbers.

o **Zip Code not valid**: if the zip_code length is not correct or the number is not in the range specified.

Tests 28-35: (output)

These tests correspond to the output. We check that the input is valid anytime we execute a valid test.

FUNCTION 2

1. GRAMMAR

```
<File>::= <Start><Middle><End>
<Start>::= {
<End>::= }
<Middle>::= <Elem1><Separator1><Elem2>
<Separator1>::= ,
<Elem1>::= <Identifier1><Separator2><Value1>
<Separator2>::= :
<Identifier1>::= <Quote><OrderID><Quote>
<Quote>::= "
<OrderID>::= OrderID
<Value1>::= <Quote><OrderID-value><Quote>
<OrderID-value>::= [a-f0-9]{13}
<Elem2>::=<Identifier2><Separator2><Value2>
<Identifier2>::= <Quote><ContactEmail><Quote>
<ContactEmail>::= ContactEmail
<Value2>::= <Quote><ContactEmail-value><Quote>
<ContactEmail-value>::=<Before @><@><After @><After final dot>
<Before @>::= [a-zA-Z0-9-\._]+
<@>::= @
<After @>::= ([a-zA-z0-9_]+\.)+
<After final dot>::= [a-zA-Z0-9]{2,4}
```

The regular representing the possible input of function 2, which is a JSON file containing the order id and the contact email of an order, is the following:

```
{“OrderID”:“[a-z0-9]{13}”,“ContactEmail”: “[a-zA-z0-9-\._]+@[a-zA-z0-9_]+\.)+
[a-zA-Z0-9]{2,4}”}
```

It is important to mention that any amount of spaces, tabs, next line characters, etc. could be inserted between any of the main fields and the format would still be valid, but we do not include them in the expression to improve readability and understanding.

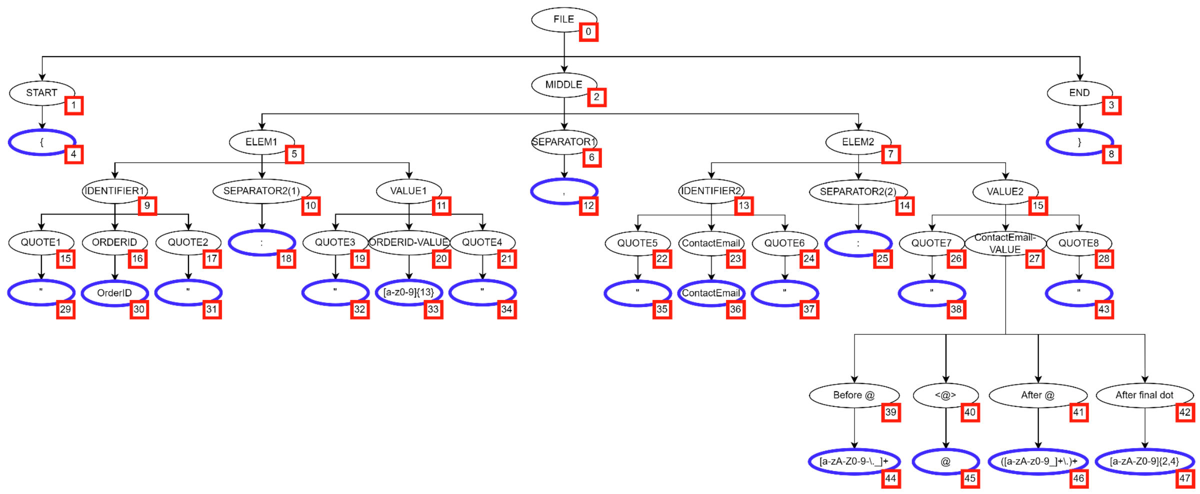
In this way, the file will follow the format below, taking into account that a valid email starts with some combination of lower and upper case letters, numbers, "_", "." and "-", then an @, then a mail format (gmail, hotmail, or any alphabetical string, followed by a "."(repeated any number of times)), and finally a DLD containing from 2 to 4 characters (com,es,fr,…).

This is the look the document would have:

```
{
"OrderID":"<String having 32 hexadecimal characters>",
"ContactEmail":"<Valid email>"
}
```

2. DERIVATION TREE

The following derivation tree represents the grammar specified above:



As you can see the terminal nodes are marked in blue, and the number of each node is specified next to it, the numbering order is from top to bottom and left to right.

3. TEST CASES

These tests are better explained with the excel file as it identifies each individual test, its type and the inputs and outputs.

We decided to test a lot of the possible of duplications, deletions and modifications that could happen in our file, to make sure that none of them are

There are 83 total tests and they are separated into five groups:

Valid: [1, 56, 58]

We can see that we get three valid cases even though there is only one test without any duplications, deletions or modifications; this is expected as some duplications of parts of the email do not make it lose the necessary format.

56: {"OrderID":"93ad8ecd0fc177ae373e3bbd3212b5c5",
"ContactEmail":"**juanjuan@correo.com**"}
58: {"OrderID":"93ad8ecd0fc177ae373e3bbd3212b5c5",
"ContactEmail":"**juan.com@correo.correo.com**"}

This proves the necessity for extensive testing, since we can make sure that possible modifications to the file do not raise an error, this we would have to test in a different way.

Delivery Email wrong format: [26, 27, 28, 29, 30, 55, 57, 59, 79, 80, 81, 82, 83]

- Deletions: [26, 27, 28, 29, 30]
- Duplications: [55, 57, 59]
- Modifications: [79, 80, 81, 82, 83]

These tests include all the ones in which the json file has a correct format but the email does not. For example:

27: {"OrderID":"93ad8ecd0fc177ae373e3bbd3212b5c5",
"ContactEmail":"**@correo.com**"}
57: {"OrderID":"93ad8ecd0fc177ae373e3bbd3212b5c5",
"ContactEmail":"juan.com**@@**correo.com"}
81: {"OrderID":"93ad8ecd0fc177ae373e3bbd3212b5c5",
"ContactEmail":"juan.com@correo.**cusat**"}

The modifications include Boundary Values testing for the fields with fixed length (the part after the dot must have 2-4 characters) and Equivalence Class tests for the other ones, for example introducing invalid characters.

Order id wrong format: [24, 53, 75, 76, 77]

- Deletions: [24]
- Duplications: [53]
- Modifications: [75, 76, 77]

These tests include all the ones in which the json file has a correct format but the order id does not. For example:

24: {"OrderID":**""**, "ContactEmail":"juan.com@correo.com"}
53: {"OrderID":"93ad8ecd0fc177ae373e3bbd3212b5c5**93ad8ecd0fc177ae373e3bbd3212b5c5**",
"ContactEmail":"juan.com@correo.com"}
77: {"OrderID":"93ad8ecd0fc**.**7**A**ae373e**_**bbd3212b5c5c",
"ContactEmail":"juan.com@correo.com"}

The modifications include Boundary Values testing for length (must be equal to 13) and Equivalence Class tests, for example introducing invalid characters.

Input file incorrect format: [4, 23, 25, 52, 54, 74, 78]

- Deletions: [4, 23, 25]
- Duplications: [52, 54]
- Modifications: [74, 78]

This exception refers to the files that do not raise an exception when reading the json file, but have some incorrect keys. For example "OrderID" missing, etc.

4: **{}**
23:{**""**:"93ad8ecd0fc177ae373e3bbd3212b5c5","ContactEmail":"juan.com@correo.com"}
52:{"**OrderIDOrderID**":"93ad8ecd0fc177ae373e3bbd3212b5c5",
"ContactEmail":"juan.com@correo.com"}

File provided not valid format: [2, 3, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73]

Includes all of the tests that cause the file to not have a .json format.

FUNCTION 2 - send_product()

In this section are included all of the tests that remove, duplicate or modify an essential part of the .json format, like separators, opening marks, quotation marks, etc.

For example:

5: {,"ContactEmail":"juan.com@correo.com"}

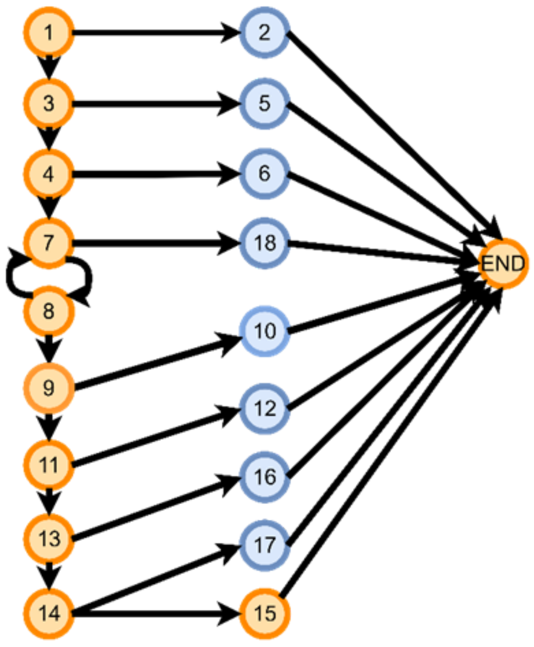
42: {"OrderID":"93ad8ecd0fc177ae373e3bbd3212b5c5",
"ContactEmail"::"juan.com@correo.com"}

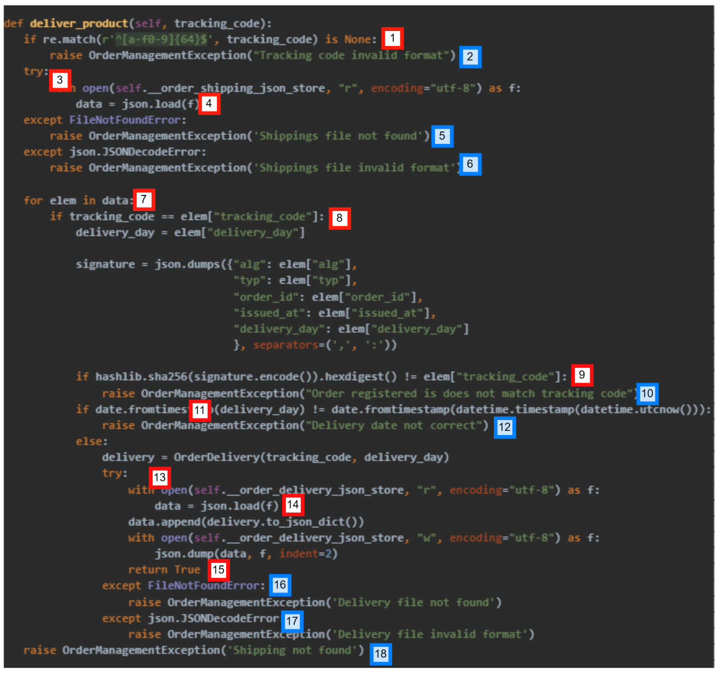
64: {"OrderID"#"93ad8ecd0fc177ae373e3bbd3212b5c5","ContactEmail":"juan.com@correo.com"}

FUNCTION 3

1. CONTROL FLOW GRAPH

Here we can observe the flow graph that Function 2 follows. The nodes in blue are exceptions, while the orange nodes represent the most common path of execution, and on the right we can find which node corresponds to each piece of code.





```
def deliver_product(self, tracking_code):
  if re.match(r'^[a-f0-9]{64}$', tracking_code) is None: 1
     raise OrderManagementException("Tracking code invalid format") 2
  try: 3
     h open(self.__order_shipping_json_store, "r", encoding="utf-8") as f:
        data = json.load(f) 4
  except FileNotFoundError:
     raise OrderManagementException('Shippings file not found') 5
  except json.JSONDecodeError:
     raise OrderManagementException('Shippings file invalid format') 6

  for elem in data: 7
     if tracking_code == elem["tracking_code"]: 8
        delivery_day = elem["delivery_day"]

        signature = json.dumps({"alg": elem["alg"],
                                "typ": elem["typ"],
                                "order_id": elem["order_id"],
                                "issued_at": elem["issued_at"],
                                "delivery_day": elem["delivery_day"]
                                }, separators=(',', ':'))

        if hashlib.sha256(signature.encode()).hexdigest() != elem["tracking_code"]: 9
           raise OrderManagementException("Order registered is does not match tracking code") 10
        if date.fromtimes 11 (delivery_day) != date.fromtimestamp(datetime.timestamp(datetime.utcnow())):
           raise OrderManagementException("Delivery date not correct") 12
        else:
           delivery = OrderDelivery(tracking_code, delivery_day)
           try:   13
              with open(self.__order_delivery_json_store, "r", encoding="utf-8") as f:
                 data = json.load(f) 14
              data.append(delivery.to_json_dict())
              with open(self.__order_delivery_json_store, "w", encoding="utf-8") as f:
                 json.dump(data, f, indent=2)
              return True 15
           except FileNotFoundError: 16
              raise OrderManagementException('Delivery file not found')
           except json.JSONDecodeError 17
              raise OrderManagementException('Delivery file invalid format')
  raise OrderManagementException('Shipping not found') 18
```

Our graph has 19 nodes and 26 edges, so the cyclomatic complexity is C = 26-19+2=9. This is equal to the number of paths we can take, regardless of the times you could loop through nodes 7 and 8. We added two more tests taking the execution of 1 or 2 loops (test 6 and 7) to illustrate the presence of the loop, but if we discard those 2 extra paths, we indeed get 9 paths.

On the other hand, the most common path of execution is the one in which no exception is raised, that is:

1-3-4-7-8-9-11-13-14-15-E.

2. TESTS

To prepare our tests we decide to create the files needed using the previous functions, as we know they are a reliable and easy way of creating the files that we need. But sometimes we do create the content manually writing what we want.

We defined 11 tests, each of them checking one of the possible paths in our graph, and some of them (6 and 7) iterating through the loop one or two times just to keep it in mind. Each different path leads to a different exception or to returning True. We illustrate this in the third page of our excel file.

It is also important to note that when doing the tests we verify that the file has changed (if it has to) and that it has not (if it was the case).

This is how we set up the tests to follow the path of execution that we want:

1- Path 1-3-4-7-8-9-11-13-14-15-E (All correct - For loop executed 1 time - code found):
- To set it up we add to the shippings file one shipment that corresponds to the tracking code to check. The test is executed 7 days after the shipping creation.

2- Path 1-2-3-E (Tracking code invalid format):
- To set it up we use a tracking code longer than it should be.

3- Path 1-3-5-E (Shippings file not found):
- To set it up we delete the shippings file before executing the function.

4- Path 1-3-4-6-E (Shippings file invalid format):
- To set it up we create a shippings file containing ***`sfjhd]`***

5- Path 1-3-4-7-18-E (For loop executed 0 times):
- To set it up we create an empty shippings file.

6- Path 1-3-4-7-8-7-18 (For loop executed 1 time - tracking code not found):
- To set it up we introduce to the function a tracking code different from the one in the input file, the input file only contains one shipment.

7- Path 1-3-4-7-8-7-8-9-11-13-14-15-E (For loop executed 2 times):
- To set it up we add a shipment with a different tracking code and then the correct one.

8- Path 1-3-4-7-8-9-10-E (Order register does not match tracking code):
- To set it up we change the tracking code inside the file, but the other attributes are the same.

9- Path 1-3-4-7-8-9-11-12-E (Delivery date not correct):

- To set it up we execute the test the same day the shipment is created.

10- Path 1-3-4-7-8-9-11-13-16-E (Delivery file not found):

- To set it up we delete the delivery file before executing the function.

11- Path 1-3-4-7-8-11-13-14-17-E (Delivery file has invalid format):

- To set it up we create the delivery file containing ***`sfjhd]`***

COMMENTS ON OUR PARTNERS' EXCEL

The tests of the excel for function 1 of our partners from group T19 are very complete. We noticed that they included None values as inputs for some of the invalid tests, which we found to be a great idea.

They did the necessary boundary values and equivalence classes tests including some more than necessary to make sure that there were not any differences between the outputs of tests that were doing the same thing.

After taking a look at all the 42 tests, we did not find any error or mistake in the logic of it, so we consider they did a great job.