CHAPTER 10

An electronic wallet

In this chapter you will develop a wallet. You will start by designing tests to define the behavior of our program, then we will define the methods according. Pay attention we will not give you all the solutions and the code.

10.1 A first test

Since we want to know if the code we will develop effectively does what it should do, we will write tests. A test can be as simple as verifying if our wallet contains money. To test that a newly created wallet does not contain money we can write a test as follow:

```
| w |
w := Wallet new.
w money = 0.
```

However doing it is tedious because we would have to manually run all the tests . We will use SUnit a system that automatically runs tests once we define them.

Our process will be the following one:

- imagine what we want to define
- define a test method
- execute it and check that it is failing
- define the method and fix it until the test pass.

With SUnit, tests are defined as methods inside a class subclass from `TestCase`. So let us start to define a test class named `WalletTest` inside the package `Wallet`.

```
TestCase subclass: #WalletTest
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Wallet'
```

And now we can define a test. To define a test, we define a method starting with `test`. Here is the definition of the same test as before but using SUnit.

```
WalletTest >> testWalletAtCreationIsZero
  | w |
  w := Wallet new.
  self assert: w money = 0
```

Now executing a test can be done in different ways:

- click on the icon close to the method in class browser,
- use the TestRunner tools,
- execute `WalletTest debug: #testWalletAtCreationIsZero` or `WalletTest run: #testWalletAtCreationIsZero`

Now you should get started. Define the class `Wallet` inside the package `Wallet`.

```
Object subclass: #Wallet
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Wallet'
```

Run the test! It should be red and now define the method `money`. For now this method is plain stupid and will return `0`. In the following of course it will sum all the coins and return such sum.

```
Wallet >> money
   ^ 0
```

10.2 Adding coins

Now we should be able to add coins to a wallet. Let us first define a new test `testCoins`.

```
WalletTest >> testCoins
  | w |
  w := Wallet new.
  w add: 2 coinsOfValue: 0.50.
  w add: 3 coinsOfValue: 0.20.
  self assert: w coinNumber = 5
```

The test adds several coins of different values and verifies that we did not lose any coins.

Now we should think how we will represent our wallet. We need to count how many coins of a given values are added or removed to a wallet. If we use an array or an ordered collection, we will have to maintain a mapping between the index and its corresponding value. Using a set will not really work since we will lose the occurrence of each coins.

10.3 Looking at Bag

A good structure to represent a wallet is a bag, instance of the class `Bag`: a bag keeps elements and their respective occurrences. Let us have a look at a bag example before continuing. You can add and remove elements of a bag and iterate on them. Let us play with it.

First we create a bag and we expect it to be empty.

```
| aFruitBag |
aFruitBag := Bag new.
aFruitBag size.
>>> 0
```

Then we add 3 bananas and verify that our bag really contains the three bananas we just added.

```
aFruitBag add: #Banana withOccurrences: 3.
aFruitBag size.
>>> 3
```

Now let us add different fruits

```
aFruitBag add: #Apple withOccurrences: 10.
aFruitBag size.
>>> 13
```

Now we check that they are not mixed together.

```
aFruitBag occurrencesOf: #Apple.
>>> 10
```

We can also add a single fruit to our bag.

```
aFruitBag add: #Banana.
aFruitBag occurrencesOf: #Banana.
>>> 4
```

We can then iterate over all the contents of the bag using the message `do:`. The code snippet with print on the Transcript (open>Tools>Transcript) all the elements one by one.

```
7 timesRepeat: [aFruitBag remove: #Apple].
aFruitBag do: [ :each |  each logCr ].
```

```
#Banana
#Banana
#Banana
#Banana
#Apple
#Apple
#Apple
```

Since for an element we know its occurrence we can iterate differently as follows:

```
aFruitBag doWithOccurrences: [ :each :occurrence | ('There is ' ,
    occurrence printString , ' ', each ) logCr ]
```

We get the following trace in the transcript.

```
'There is 4 Banana'
'There is 10 Apple'
```

10.4 Using a bag for a wallet

Since we can know how many coins of a given value are in a bag, a bag is definitively a good structure for our wallet.

We will define add an instance variable `bagCoins` to the class and the methods

- `add: anInteger coinsOfValue: aCoinNumber`
- `initialize` and
- `coinsOfValue:`.

Let us start with the method `initialize`. We define the method `initialize` as follows. It is invoked automatically when an instance is created.

```
Wallet >> initialize
  bagCoins := Bag new
```

Now define the method `add: anInteger coinsOfValue: aNumber`. Browse the class `Bag` to find the messages that you can send to a bag.

```
Wallet >> add: anInteger coinsOfValue: aNumber
  "Add to the receiver, anInteger times a coin of value aNumber"

  ... Your solution ...
```

We can define the method `coinsOfValue:` that returns the number of coins of a given value (looks like the same as asking how many banana are in the fruit bag).

```
Wallet >> coinsOfValue: aNumber
```

```
  ^ ... Your solution ...
```

10.5 More tests

The previous test is limited in the sense that we cannot distinguish if the coins are not mixed. It would be bad that a wallet would convert cents into euros. So let us define a new test to verify that the added coins are not mixed.

```
WalletTest >> testCoinsAddition
  | w |
  w := Wallet new.
  w add: 2 coinsOfValue: 0.50.
  w add: 3 coinsOfValue: 0.20.
  self assert: (w coinsOfValue: 0.5) = 2
```

We should also test that when we add twice the same coins they are effectively added.

```
WalletTest >> testCoinsAdditionISWorking
  | w |
  w := Wallet new.
  w add: 2 coinsOfValue: 0.50.
  w add: 6 coinsOfValue: 0.50.
  self assert: (w coinsOfValue: 0.5) = 8
```

10.6 Testing money

Now we can test that the `money` message returns the amount of money contained in the wallet and we should change the implementation of the `money`. We define two tests.

```
WalletTest >> testMoney
  | w |
  w := Wallet new.
  w add: 2 coinsOfValue: 0.50.
  w add: 3 coinsOfValue: 0.20.
  w add: 1 coinsOfValue: 0.02.
  self assert: w money = 1.62
```

```
WalletTest >> testMoney2
  | w |
  w := Wallet new.
  w add: 2 coinsOfValue: 0.50.
  w add: 3 coinsOfValue: 0.20.
  w add: 1 coinsOfValue: 0.02.
  w add: 5 coinsOfValue: 0.05.
  self assert: w money = 1.87
```

Now we should implement the method `money`.

```
Wallet >> money

  ^ ... Your solution ...
```

10.7 Checking to pay an amount

Now we can add a new message to know whether we can pay a certain amount. But let us write some tests first.

```
WalletTest >> testCanPay
  | w |
  w := Wallet new.
  w add: 2 coinsOfValue: 0.50.
  w add: 3 coinsOfValue: 0.20.
  w add: 1 coinsOfValue: 0.02.
  w add: 5 coinsOfValue: 0.05.
  self assert: (w canPay: 2) not.
  self assert: (w canPay: 0.50).
```

Define the message `canPay:`.

```
Wallet >> canPay: amounOfMoney
  "returns true when we can pay the amount of money"

  ^ ... Your solution ...
```

10.8 Biggest coin

Now let us define a method to determine the largest coin in a wallet. We write a test.

```
WalletTest >> testBiggestCoins
  | w |
  w := Wallet new.
  w add: 10 coinsOfValue: 0.50.
  w add: 10 coinsOfValue: 0.20.
  w add: 10 coinsOfValue: 0.10.
  self assert: w biggest equals: 0.50.
```

Note that the `assert:` message can also be replaced `assert:equals:` and this is what we did: we replaced the expression `self assert: w biggest = 0.5` by `self assert: w biggest equals: 0.50`.

Now we should define the method `biggest`.

```
Wallet >> biggest
  "Returns the biggest coin with a value below anAmount. For
    example, {(3 -> 0.5) . (3 -> 0.2) . (5-> 0.1)} biggest -> 0.5"

  ^ ... Your solution ...
```

10.9 Biggest below a value

We can also define the method `biggestBelow:` that returns the first coin whose value is stricly smaller than the argument. `{(3 -> 0.5) . (3 -> 0.2) . (5-> 0.1)} biggestBelow: 0.40` returns 0.2.

```
WalletTest >> testBiggestCoinsBelow
  | w |
  w := Wallet new.
  w add: 10 coinsOfValue: 0.50.
  w add: 10 coinsOfValue: 0.20.
  w add: 10 coinsOfValue: 0.10.
  self assert: (w biggestBelow: 1) equals: 0.50.
  self assert: (w biggestBelow: 0.5) equals: 0.20.
  self assert: (w biggestBelow: 0.48) equals: 0.20.
  self assert: (w biggestBelow: 0.20) equals: 0.10.
  self assert: (w biggestBelow: 0.10) equals: 0.
```

```
Wallet >> biggestBelow: anAmount
  "Returns the biggest coin with a value below anAmount. For
    example, {(3 -> 0.5) . (3 -> 0.2) . (5-> 0.1)} biggestBelow:
    0.40 -> 0.2"

  ^ ... Your solution ...
```

10.10 Improving the API

Better string representation

Now it is time to improve the API for our objects. First we should improve the way the wallet objects are printed so that we can debug more easily. For that we add the method `printOn: aStream` as follows:

```
Wallet >> printOn: aStream
  super printOn: aStream.
  aStream nextPutAll: ' (', self money asString, ')'
```

Easier addition

We can improve the API to add coins in particular when we add only one coin. So now you start to get used to it. We define a test.

```
WalletTest >> testAddOneCoin
  | w |
  w := Wallet new.
  w addCoin: 0.50.
  self assert: (w coinsOfValue: 0.5) = 1.
  self assert: w money equals: 0.5
```

Define the method `addCoin:`.

```
Wallet >> addCoin: aNumber
  "Add to the receiver a coin of value aNumber"

  ... Your solution ...
```

Removing coins

We can now implement the removal of a coin.

```
WalletTest >> testRemove
  | w |
  w := Wallet new.
  w add: 2 coinsOfValue: 0.50.
  w add: 3 coinsOfValue: 0.20.
  w add: 1 coinsOfValue: 0.02.
  w add: 5 coinsOfValue: 0.05.
  w removeCoin: 0.5.
  self assert: w money = 1.37
```

Define the method `removeCoin:`.

```
Wallet >> removeCoin: aCoinNumber
  "Remove from the receiver a coin of value aNumber"

  ... Your solution ...
```

We can generalize this behavior with a method `remove:coinsOfValue:`. Write a test.

```
WalletTest >> testRemoveCoins
  | w |
  w := Wallet new.

  ... Your solution ...
```

```
Wallet >> remove: anInteger coinsOfValue: aCoin
  "Remove from the receiver, anInteger times a coin of value aNumber"

  bagCoins add: aCoin withOccurrences: anInteger
```

We can also define the method `biggestAndRemove` which removes the biggest coin and returns it.

```
Wallet >> biggestAndRemove
  | b |
  b := self biggest.
  self removeCoin: b.
  ^ b
```

10.11 Coins for paying: First version

Now we would like to know the coins that we can use to pay a certain amount. We can define a method `coinsFor:` that will return a new wallet containing the coins to pay a given amount.

This is a more challenging task and we will propose a first version then we will add more complex situations and propose a more complex solution. So let us define a test.

```
WalletTest >> testCoinsForPaying

  | w paid |
  w := Wallet new.
  w add: 10 coinsOfValue: 0.50.
  w add: 10 coinsOfValue: 0.20.
  w add: 10 coinsOfValue: 0.10.
  paid := (w coinsFor: 2.5).
  self assert: paid money equals: 2.5.
  self assert: (paid coinsOfValue: 0.5) equals: 5
```

```
Wallet >> coinsFor: aValue
  "Returns a wallet with the largest coins to pay a certain amount
    and an empty wallet if this is not possible"

  | res |
  res := self class new.
  ^ (self canPay: aValue)
    ifFalse: [ res ]
    ifTrue: [ self coinsFor: aValue into: res ]
```

The method `coinsFor:` creates wallet and fill with the largest coins comprising a given value.

Using the previously defined methods, define a first version of the method `coinsFor: aValue into: accuWallet`.

```
Wallet >> coinsFor: aValue into: accuWallet

  ... Your solution ...
```

Here is a possible simple solution: we remove from the wallet the largest coin and we add it to the resulting wallet. This solution is not working well as we will show it.

```
Wallet >> coinsFor: aValue into: accuWallet

  [ accuWallet money < self money ]
      whileTrue: [ accuWallet addCoin: self biggestAndRemove ].
  ^ accuWallet
```

10.12 Better heuristics

Let us try some tests to see if our previous way to get coins is working. (The previous algorithm does not work with such behavior.)

The first test checks that when there is no more coins of the biggest value, we check that the next coin is then used.

```
WalletTest >> testCoinsForPayingWithOtherCoins
  | w paid |
  w := Wallet new.
  w add: 1 coinsOfValue: 0.50.
  w add: 10 coinsOfValue: 0.20.
  w add: 10 coinsOfValue: 0.10.
  paid := (w coinsFor: 2.4).
  self assert: paid money equals: 2.4.
  self assert: (paid coinsOfValue: 0.5) equals: 1.
  self assert: (paid coinsOfValue: 0.2) equals: 9.
```

Run the tests and define the method `coinsFor:` to invoke a copy of the method `coinsFor: aValue into: accuWallet` renamed `coinsFor: aValue into2: accuWallet` to start with.

```
Wallet >> coinsFor: aValue
  "Returns a wallet with the largest coins to pay a certain amount
    and an empty wallet if this is not possible"
  | res |
  res := self class new.
  ^ (self canPay: aValue)
    ifFalse: [ res ]
    ifTrue: [ self coinsFor: aValue into2: res ]
```

The previous algorithm (implemented above in `coinsFor: aValue into:`) does not work with such behavior. So you should start to address the problem and add more and more tests. The second test checks that even if there is a coin with a largest value, the algorithm selects the next one. Here to pay 0.6, we should get 0.5 then we should not take 0.2 the next coin but 0.1 instead.

```
WalletTest >> testCoinsForPayingWithOtherThanTop
  | w paid |
  w := Wallet new.
  w add: 1 coinsOfValue: 0.50.
  w add: 10 coinsOfValue: 0.20.
  w add: 10 coinsOfValue: 0.10.
  paid := (w coinsFor: 0.6).
  self assert: paid money equals: 0.6.
  self assert: (paid coinsOfValue: 0.5) equals: 1.
  self assert: (paid coinsOfValue: 0.1) equals: 1.
```

In this version we check that the algorithm should skip multiple coins that

are available. In the example, for 0.6 it should select: 0.5 then skip the remaining 0.5, and 0.2 to get one 0.1.

```
WalletTest >> testCoinsForPayingWithOtherThanTopMoreDifficult
  | w paid |
  w := Wallet new.
  w add: 2 coinsOfValue: 0.50.
  w add: 10 coinsOfValue: 0.20.
  w add: 10 coinsOfValue: 0.10.
  paid := (w coinsFor: 0.6).
  self assert: paid money equals: 0.6.
  self assert: (paid coinsOfValue: 0.5) equals: 1.
  self assert: (paid coinsOfValue: 0.1) equals: 1.
```

The following one is a variant of the previous test where the biggest coin should be skipped.

```
WalletTest >> testCoinsForPayingWithOtherThanTopMoreDifficult2
  | w paid |
  w := Wallet new.
  w add: 1 coinsOfValue: 1.
  w add: 2 coinsOfValue: 0.50.
  w add: 10 coinsOfValue: 0.20.
  w add: 10 coinsOfValue: 0.10.
  paid := (w coinsFor: 0.6).
  self assert: paid money equals: 0.6.
  self assert: (paid coinsOfValue: 0.5) equals: 1.
  self assert: (paid coinsOfValue: 0.1) equals: 1.
```

10.13 Conclusion

What this example shows is that while a wallet is essentially a bag, having a wallet is a much more powerful solution. The wallet encapsulates an internal representation and builds a more complex API around it.