Swift vs. Objective-C: A New Programming Language

Cristian González García, Jordán Pascual Espada, B. Cristina Pelayo G-Bustelo, and Juan Manuel Cueva Lovelle,

University of Oviedo, Department of Computer Science

***Abstract* — The appearance of a new programming language gives the necessity to contrast its contribution with the existing programming languages to evaluate the novelties and improvements that the new programming language offers for developers. These kind of studies can show us the efficiency, improvements and useful or uselessness of the new programming languages. Also these studies can show us the good or bad properties of the existing programming languages. For these reasons, these studies allow us to know if the new programming language is offering improvements or relapses.**

In this article, we compare the new programming language of Apple, Swift, with the main programming language of Apple before Swift, Objective-C. We are going to show the differences, characteristics and novelties to verify the words of Apple about Swift. With that we want to answer the next question: Is Swift a new programming language easier, more secure and quicker to develop than Objective-C?

***Keywords* — Object oriented programming, Programming, Functional programming, Programming profession, Software**

I. Introduction

SWIFT is the new programming language created by Apple and it was presented to the public on September 9, 2014 but developers could use it since June 6, 2014. It allows for developing applications for the new version of operating systems of Apple: iOS and OS X [1]. The Apple's intention is to offer a new programming language easier, simpler, more flexible, quicker, funnier and friendly to program than Objective-C [2] to facilitate the applications development for platforms of Apple [1].

Swift was launched to offer an alternative to Objective-C because this has a syntax which barely evolved from it was created and has a great difference with other programming languages that have appeared in the latest years, because these have based on the C++ syntax. For this, Swift is inspired in new programming languages like C++11, C#, F#, Go, Haskell, Java, JavaScript, Python, Ruby, or Scala. Then his syntax is totally different than its predecessor. The Swift's syntax is more simplified because it does not use pointers and includes improvements in its data structures and in its syntax. As we will see, Swift has an easier syntax which helps to developers to have less mistakes and incorporates new functionalities and a new programming paradigm [1].

Mainly, Swift is an object-oriented and imperative programming language as Objective-C but Swift incorporates the functional programming. Some examples of this are the closures, maps and filters.

Due to this facts, it is necessary a study about Swift to check if Swift could be a programming language adapted to the new times and if it could facilitate the application development for platforms of Apple [1].

The remainder of this paper is structured as following: in section III we present the differences among versions. In section IV we discuss the different changes introduced in Swift in front of Objectice-C. Section V explains the new language characteristics. Section VII talks about the novelties that Swift incorporates. In section VI we present the methodology, results and the discussion. Finally, section VII contains the conclusions.

II. Versions

Swift have had different versions with changes in it syntax and functionality since that the first version to developers appeared on June 6, 2014 [1]. At present, Swift is in its third version, Swift 1.2.

The first public version, Swift 1.0 GM, was presented on June 06 2014. It was a Golden Master (GM) version because Apple announced that it will continue adding changes and improvements in the programming language. Swift 1.0 GM presented a lot of changes in its syntax, native libraries and the value type of some function, variable to use the new type "optional" and the syntax of some reserved words like arrays, dictionaries and open range operators.

Swift 1.1, the second version, appeared on October 22, 2014. This update added the "failures initializers", changed some "protocols" and some internal functionalities of Swift.
Swift 1.2 appeared on April 8, 2015. It was a major update. It arrived with the new version of Xcode 6.3. It introduced different improvements in the compiler: the compiler started to create incremental builds; better compilation velocity; improved the error and warnings messages; better stability to avoid the "SourceKit warning". The new language features were: a new reserved word "as!" to clear to readers and developers; the nullability in Objective-C headers; the possibility to export "enumerations" from Swift to Objective-

DOI: 10.9781/ijimai.2015.3310

C with the attribute "@objc"; changes in constants (let); a new native collection: Set.

III. CHANGES RESPECT TO OBJECTIVE-C

Swift and Objective-C use the same compiler, the Low Level Virtual Machine (LLVM). LLVM was created for a student at the University of Illinois in 2000 and it is programmed in C++ [3], [4]. LLVM transforms the Swift source code in optimise native source code for the elected hardware (Mac, iPhone, or iPad) [2].

Swift provides full compatibility with Objective-C and old projects because it allows to use the same libraries, primitive types, control flow and other functions that has Objective-C. However, Swift has various libraries translated to Swift's native code [5].

Furthermore Swift introduces new changes to search to abstract this programming language to the same level that modern programming languages, and in some cases, it obtained more abstraction than them [2], because this is one of the current ways in computer science to help developers when they programme applications and they need to improve their programming [6]–[9]. Also, it has renovated the old syntax of Objective-C. Some examples of this are the classes, protocols, control flows and variables. We are going to explain these similarities and differences in this section.

A. Pre-processor

Swift does not have pre-processor as occurs in C and Objective-C. To achieve the same functionality in Swift, users must use constants instead of the simple macros, namely, define a constant variable and in the case of complex macros, functions [2].

B. Syntax

As in other programming languages (JavaScript, Ruby), Swift allows the optional use of the semicolon character (";") at the end of the line. Besides, Swift uses as access operator the point character (".") like many programming languages instead the square brackets ("[", "]") as Objective-C. This allows more legible code because it has a syntax more similar to the most used programming languages [10], [11].

Furthermore it contains changes in the flow structures' syntax. Now, these must use braces ("{" and "}") to enclose the scope to avoid programming problems. For example, in conditional flow structure ("if"), in some programming languages, in the case that we do not use braces, the first sentence is the sentence that the flow structure will do when the condition will be true and the other sentences will execute in the other cases ("else"). With these changes, Apple wants to do a more legible and easier syntax to developers.

C. Collections

Objective-C has three collections: NSArray, NSSet, and NSDictionary. Initially, Swift only had two collections: Array and Dictionary but in Swift 1.2 Swift added the Set collection. The Swift collections are implemented using structures which differ from the implementation of Objective-C which uses classes.

This difference implicates that in Swift, when these collections are assigned to a constant, variable or they are sent as a function's parameter, Swift creates a copy of them to work with this copy instead the original collection. Meanwhile, Objective-C collections work with the original collection because Objective-C passes references to the original collections instead a copy.

D. Variables

1) Labeled statements

One of the changes in Swift with respect to Objective-C is the functionality of the reserved word "goto". Objective-C allows to use this reserved word to go to any part of the current scope. On the other hand, Swift removed this reserved word and created the "Labeled Statements" [2]. The "Labeled Statements" have a similar functionality as the "goto" but these operate in a smaller scope than the "goto" in Objective-C. Exactly, they have the same scope as other programming languages like C#, Java, and PHP: allow to go to a tag inside a "nested loop" or "switch".

2) Boolean type

Swift, Boolean types have been simplified. Now there only exist the variables "true" and "false". In Objective-C it exists "true"/"false", 0/1, and "TRUE"/"FALSE".

3) Property observers

With Swift has simplified the process to add observers to the variables. To do this, Swift provides two new reserved words: "willset" and "didset" [2]. "willset" is called before the value is assigned to the variable and "didset" is called after the value was assigned to the variable. However, these observers are never called in the first assignation of the instance.

E. Classes and Structures

In Swift, the syntax to create a class or structure is very similar as C++, C#, and Java. Besides, Swift only uses one file (".swift") to define a class, contrary to Objective-C that uses two files (".h" and ".m"). For that, in Swift, you have to define all the class or structure in the same file.

The Fig. 10 contains an example about a definition of a class in Swift. In the first part, it uses a default method, "init", to define the constructor. It keeps the same way and reserved word as Objective-C.

So, to access to the object, Swift still uses the reserved word "self" instead the operator "->". Furthermore, Swift facilitates the access to the properties of the object because it uses the dot operator (".") like C#, Java, JavaScript and Python instead the bracket operators ("[" and "]") as Objective-C. It was possible because Swift removed pointers and it allowed to facilitate the syntax and operators to work with the different properties of the object. For this reason, now, we do not have to allocate a memory block as in Objective-C. Moreover, in Swift we do not have to use the reserved word "new" as in other programming languages when we create an object.

About the inheritance in Swift, it keeps the same way as Objective-C despite now we have to specify when we override the father's method with the reserved word "override" before the child's method and with the reserved word "super" and the dot operator before the father's method when we have to call it.

```
// Class with inheritance
class Child: Person {

    override init(name:String){
        super.init(name: name)
    }

    override func description() -> String {
        return "I'm a child. My name is \(name)."
    }
}

var child = Child(name:"Cris Jr.");
child.name = "Cristian González García Jr.";
var descriptionJr = child.description();
```

Fig. 10. Class example

Apart from this, Swift still allows to avoid the overwriting of a variable, method or class if you include, in the father's part, the reserved word "final". This system uses the same way as C++ and C#.

F. Enumerations

"Enumerations" still have the same functionality. The only changes that they have had are in their syntax. Now, they have a clearer syntax and more similar to a class. Nevertheless, Swift allows to define barcodes, QR codes and raw data with a easier form than Objective-C [2].

G. Extensions

Extensions are used to add new functionalities to an existing class, structure or enumeration to which you cannot access it code. The restriction is that you cannot overwrite the existing functionalities. Extensions are similar to Objective-C categories but without a name [2]. Owing to, for instance, developers can extend the functionality of the String class or others to add new variables or methods.

IV. Language Characteristics

Swift introduced various changes in how to program and it have added new characteristics: it have added changes in variables, it have modified functions and methods to incorporate multiple return and diverse functional programming characteristics. All this will be explained with more details in this section.

A. Variables

Swift is more restrictive than Objective-C because Swift has a strong typing to avoid insecure code [2]. Swift obliges to initialise the variables before their first use. Moreover, you must too specify if the variable is a variable ("var") or a constant ("let") using these reserved word before the name. Besides, Swift checks possible arrays and integer overflow and auto-manages the memory stack using the Automatic Reference Counting (ARC) [2].

Swift allows to the developer to specify explicitly the value type or let the compiler infers the type (Fig. 11), although Swift is strongly typed, for that, when you set a variable in the first time, the compiler assigns it the type and you cannot assign a new value with different type later but you could do an explicit type conversion for changing the type of the value that you want to assign to the type of the variable.

```
// Variable
var variable:String = "I'm a variable"

// Constant
let constant:String = "I'm a constant"

// Static variable
var string:String = "Hello World Swift"

// Inflicted variable
var newString = "Hello World Swift"
```

Fig. 11. Variable examples

B. Functions

Swift allows to send functions as parameter of other functions. Known as Lambda function, Anonymous function, Function literal or Lambda abstraction in Functional Programming. This is one of the characteristics of the Functional programming that Swift has. We show an example in Fig. 12: A – this function receives a function; B it has multiple return; C - one of these return values is a function.

```
func add(array:[Int]) -> Int{
    var aux = 0;
    for i in array{
        aux += i
    }
    return aux
}

func addWithFunction(array:[Int], f:([Int]) -> Int)
    -> (a:Int, b:Int){
    return (f(array), array.count)
}

var array = [Int](0...4)
var result = addWithFunction(array, add)
println(result.a)
println(result.b)
```

A

B

C

Fig. 12. A function that receives a function and returns multiple parameters

C. Classes

Classes incorporate two changes: a new type of constructor, thr "Convenience Initializer" and changes in destructors now known as "Deinitialization". Next, we are going to explain both concepts.

1) Convenience Initializer

The "Convenience Initializer" is an optional constructor that, in case of it exists, it is always called before of the main constructor. Thus, Apple pretends to do the creation of constructors clearly and easier because developers would use different "Convenience Initializer" as alternative constructors [2]. So, normal constructors would have the generic code to the all possible cases. To create a "Convenience Initializers" it is necessary to use using the reserved word "convenience" before the constructor (Fig. 13).

2) Deinitialization

Due that Swift incorporates ARC, for that, destructors are not needed to release memory as occur in C, C++ and Objective-C but Swift incorporates the "Deinitialization" [2].

A "Deinitialization" is a method which is called immediately after an instance is released and it cannot be called explicitly. Using "Deinitialization", developers have a mechanism to do a special clean of different resources or to do actions when the object dies. For example when the object have to work with files. In this case, when the object dies, the program releases the memory that the object had used but the

program cannot close the file because ARC cannot infer this [2]. Then, you can implement a "Deinitialization" to close the file when the object dies. To create a "Deinitialization" you only need to add and implement the method "deinit" as we show in Fig. 13.

```
class Fruit {
    var name: String
    var weight: Int

    init() {
        name = "Unknown"
        weight = 0
    }

    convenience init(name:String) {
        self.init()
        self.name = name
    }

    deinit {
        println("\(name) has disappeared")
    }
}

var fruit = Food()
var fruit1 = Food(name: "Apple")
var fruit2 = Food(weight: 50)
```

Fig. 13. Class example with Convenience Initializers and a Deinit

V. NOVELTIES

In this section we are going to explain the novelties that Swift introduced in relation to Objective-C. The most notable is the "Playground", a sandbox to programming. Also, we are going to explain the novelties in the operators, the new type of variables and some characteristics of the functional programming that Swift introduced as the "Closures". Finally, we are going to talk about the generic which Apple have introduced in Swift, the novelties in the "Switch" flow structure and how to work in Objective-C with Swift and vice versa.

A. Playground

Swift incorporates a new technology in the Integrated Development Environment (IDE) Xcode 6 [12], the "Playground [1], [2]. The "Playground" is a "compiler-in-real-time" which executes the code immediately while the user develops. It provides the value of the execution in different parts of the program (assignments, operations, returns), because the compiler auto-executes itself when it detects a change in the code. Therefore, the "Playground" is a terminal which can execute Swift as other programming languages have like Bash, PHP, Python, and Ruby but with an IDE and some utilities.

This allows to be able to test algorithms in a quick, easy, efficient and isolated way because it gives the possibility to encapsulate the algorithm in a clean sandbox without any relationship to the final application to avoid collateral damages [2].

B. Operators

Swift has all the unary operators (++, --, !), binary operators (+, -, *, /), ternary operators (a:b?c), logical operators (!, &&, ||, true, false), and the assignment of C. Even so, Apple added Range Operators, Overflow Operators and Custom operators. We are going to explain these three operators in the next lines.

1) Range operators

The "Close Range Operator" uses the ellipsis (a…b) as we show in Fig. 14. It defines a range from "a" to "b" and includes both. It is used to iterate on a range where both limits are included.

```
// Closed Range Operator
var closedExample = [0...5]
for index in 1...5 {
    println("\(closedExample)")
}
```

Fig. 14. Closed Range Operator example

On the other hand, the "Half-Open Range Operator" symbol is composed of two dots and the less-than symbol (a..<b) as we can see in Fig. 15. The difference with the "Close Range Operator" is that the "Half-Open Range Operator" does not include the "b" into the range. It defines a range from "a" to "b-1.

```
// Half-Closed Range Operator
var halfClosedExample = [0..<5]
for index in 1..<5 {
    println("\(halfClosedExample)")
}
```

Fig. 15. Half-Closed Range Operator

Using this two operators, we can define in a quick way the creation of a list, array or their iteration.

2) Overflow Operators

In Swift the arithmetic operators (+, -, *, /) do not have "overflow" by default [2]. In case that we want it, we have to add the ampersand sign (&) before the arithmetic operator as we can see in Fig. 16. For example, to apply overflow to the addition, we must use the combination "&+". If we want to have underflow in the subtraction, we have to use "&-". Swift also allows to control the division by zero with the combination "&/" and "&%". For the multiplication we have to use the "&*".

```
var max = Int8.max
//var maxPlus = max + 1 // Error
var maxPlus = max &+ 1

var min = Int8.min
//var minxPlus = max - 1 // Error
var minPlus = max &- 1

//var div = 1 / 0 // Error
var div = 1 &/ 0
var div2 = 1 &% 0

var mul = 1 &* 0
```

Fig. 16. Overflow Operators examples

3) Custom operators

Swift allows to define new operators using the existing arithmetic signs [2]. This is impossible in Objective-C while C++ affords this functionality.

To create a new "Custom Operator", we must to declare its header and do his implementation. There are three ways to do a "Custom Operator": prefix, infix, and postfix (Fig. 17).

```
prefix operator ++++ {}

prefix func ++++ (inout newValue: Int) -> Int {
    newValue += 4
    return newValue
}

var five = 5
var nine = ++++five
```

Fig. 17. Custom Operator

C. Variables

Swift has new data variable types: tuples, "optionals" and "Lazy Stored Property". Some programming languages already incorporated some of these variable types. Then, with this new variable types, Swift offers new ways to work with more flexible and facilities for developers.

1) Tuples

Tuples allow group together multiple values in a unique component. Therefore, one of its uses is to allow the return of multiple values in a function. The Fig. 18 contains an example with tuples.

```
let contact = (13, "Cristian")
var office = contact.0
var name = contact.1
```

Fig. 18. Tuples in Swift

2) Optionals

The "Optional" is a new value type in Swift which neither exist in C nor Objective-C. It is used to assign a type when the value could be of different type or nil [2]. In this way, if a conversion cannot be done, the variable would take a "nil" value. To declare an "optional" variable you must write a question mark (?) after the type. In the example of the Fig. 19, the variable "convertedNumberImplicit" can never be "nil" and in the case that the assignment value would be "nil", the program will break. In the next variable, "convertedNumberExplicit", the type is an Optional Integer. In this case, the value could be "nil".

```
// Optional
let possibleNumber: String = "123"
let convertedNumberImplicit = possibleNumber.toInt()
let convertedNumberExplicit:Int? = possibleNumber.toInt()

let possibleNotNumber = "Hello Swift"
let convertedNotNumberImplicit = possibleNotNumber.toInt()
let convertedNotNumberExplicit:Int? = possibleNotNumber.toInt()
```

Fig. 19. Example that assign value to Optional Variables

This allows to use an optional value in a conditional flow because the "nil" value is the same as a "false" value. However, you can force to read the "optional" value if you write an exclamation mark (!) after the "optional" variable (Fig. 20).

```
if convertedNumberImplicit != nil {
    println("integer value: \(convertedNumberImplicit!)")
} else {
    println("Could not be converted to an integer")
}
```

Fig. 20. Example about how to print Optional Variables

3) Lazy Stored Property

The "Lazy Stored Property" is a class or structure property which value is not calculated until its first use. Until then, it has not value [2]. It uses is similar than other programming languages like C# and Python. In Fig. 21 we create a "Lazy Stored Property" using the declaration "lazy" before the variable.

```
class Student{
    var name: String = "Cristian"
    var surname: String = "González García"
    lazy var contacts = getContacts()
}
```

Fig. 21. Lazy Properties

D. Closures

Swift incorporates the "Closures" as a part of the functional paradigm. The "Closures" allows to evaluate a function in a context which contains one or more variables depending of another context. In Fig. 22, the internal function is executed by the "map function" and it depends on the parameters of the array.

```
var numbers = [20, 19, 7, 12]

numbers.map({
    (number: Int) -> Int in
        let result = 3 * number
        return result
    })
println(numbers)
```

Fig. 22. Applying a Closure using a Map function

Swift allows to create the "Closures" with other simpler ways: implicit return which implicates to not so specify the return; the possibility to use the "Shorthand Argument Names", default arguments created automatically by Swift to work with the parameters sent to the "Closure" without a previous declaration; including just the operator (<, >, +, -,…) in the case that the function only has two parameters and only need to return the result.

E. Generics

Swift adds as a novelty the "Generics", known as "Templates" in other programming languages like C++, and Java. In Objective-C, to have this functionality, you must implement this part of the code in C++.

The incorporation of "Generics" allows to do easier, more flexible, more reusable and more functional with any other data type, avoiding the redundancy of code. This is one of the most powerful functionalities of Swift [2]. As occurs in C++ and Java, the data type used to do the method is generic "<T>". We can see an example in Fig. 23.

```
func genericFunction<T: Comparable>(a: T, b: T) -> T {
    if(a > b){
        return a
    }
    return b
}

genericFunction(5, 3)
genericFunction(3.5, 3.7)
genericFunction("a", "b")
```

Fig. 23. Generic Function

F. Switch

The "Switch" in Swift has improvements in relation with Objective-C [2]. Now, the "break" sentence is optional because the compiler automatically breaks the "switch" when finishes the "case". With this way, Apple pretends to do an easier use and with less programming mistakes. Due to this, it is impossible to have empty "cases" except if you use the reserved word "fallthrough". The reserved word "fallthrough" does that the execution continue to the next "case". The same functionality is in the "Switch" of C#.

As well allows for introducing several verifications in the same "case": you can separate them with commas (,), you can use "Range Operators" or tuples. Also, the "Switch" allows the combination between "Range Operators" and tuples to allow to developers the evaluation of mathematical functions in an easier and quicker way.

Another improvement is the optional clause "where". It allows to add a new additional check in a "case".

G. Objective-C in Swift

Swift allows to incorporate and use Objective-C code in the same Swift program [13]. To insert the Objective-C code, you must insert the Objective-C files (".h" and ".m") in the project and import the Objective-C header in the "Bridging-Header" file which you need to create in the Swift project. This file contains the import all the Objective-C headers (.h) which we will use. After this step, we must connect this file with the configuration through the "Build Settings" configuration. When it is configured, you can work with the Objective-C code using Swift syntax and rules. To import C++ code, you have to create an Objective-C or C wrapper around the C++ code.

H. Swift in Objective-C

To use Swift code in an Objective-C project, the Xcode auto-generates a file when you import the Swift code. This importation allows the layer to access to all the functionalities of the Objective-C code [13]. These files keep the original name but with the additional postfix "-Swift" and the extension ".h". When the importation is done, it just needs to import the Objective-C header files. In this way, you can use the Swift code in an Objective-C project.

VI. Evaluation and Discussion

In this section we are going to explain in detail the different process of evaluation and then we show the results. Firstly, we will describe the used methodology. After that, we will show the results and discuss them.

A. Methodology

We have evaluated the implementation of the same code in each different programming language: Objective-C and Swift. We have chosen an example of an XML parser, exactly, using the Foundation XMLParser. In our case, we implemented a main method and override two methods: "didStartElement" and "foundCharacters". The first one, "didStartElement", is called when the parser needs to process a new XML node. The second one, "foundCharacters", is called when the parser find text into a node.

After that, we evaluate the code using two ways. Firstly, we analysed and compared the syntax. Secondly, we did a quantitative analysis based on source code: we counted the lines with code; the words, reserved words and the numbers of "Switch cases"; the characters that developers need to write that code.

B. Results

In this section, we are going to explain the two ways that we used to compare Objective-C with Swift. Firstly, we are going to talk about the syntax analysis where we have implement three methods in both programming languages. Later, we are going to present the quantitative analysis where we compare both in base on their lines, words, and characters using the same three methods of the syntax analysis.

1) Syntax Analysis

Firstly, we analysed the "beginParsing" method in both programming languages (Fig. 24 and Fig. 25). As we can see, the Objective-C implementation uses a pointer. Furthermore, when it initialise the parser, it needs to allocate memory.

```
-(void)beginParsing:(NSURL *)xmlURL
{
    parser = [[NSXMLParser alloc] initWithContentsOfURL:xmlURL];
    [parser setDelegate:self];

    [parser setShouldProcessNamespaces:NO];
    [parser setShouldReportNamespacePrefixes:NO];
    [parser setShouldResolveExternalEntities:NO];

    [parser parse]
}
```

Fig. 24. "BeginParsing" method in Objective-C

```
func beginParsing(xmlUrl :NSURL)
{
    parser = NSXMLParser(contentsOfURL: xmlUrl)

    parser.delegate = self
    parser.shouldProcessNamespaces = false
    parser.shouldReportNamespacePrefixes = false
    parser.shouldResolveExternalEntities = false

    parser.parse()
}
```

Fig. 25. "BeginParsing" method in Swift

Secondly, in the "didStartElement" method (Figure 17 and Figure 18) we can observe similarities with the previous method. Objective-C uses pointers and need to allocate memory. Besides, you have to create a copy to avoid the use of the original. Another difference between Objective-C and Swift is the obligatory use of the "at sign" character (@) before a "string".

```
- (void)parser:(NSXMLParser *)parser didStartElement:(NSString *)elementName namespaceURI:(NSString *)
    namespaceURI qualifiedName:(NSString *)qualifiedName attributes:(NSDictionary *)attributeDict
{
    element = [[elementName copy] stringByTrimmingCharactersInSet:[NSCharacterSet
        whitespaceAndNewlineCharacterSet]];

    if ([element isEqualToString: @"views"]) {
        elements = [[NSMutableDictionary alloc] init];
        views = [[Views alloc] init];
    }
}
```

Fig.26. “didStartElement” method in Objective-C

```
func parser(parser: NSXMLParser!,didStartElement elementName: String!, namespaceURI: String!,
    qualifiedName : String!, attributes attributeDict: NSDictionary!)
{
    element = elementName.stringByTrimmingCharactersInSet(NSCharacterSet.
        whitespaceAndNewlineCharacterSet())

    if (element as NSString).isEqualToString("views") {
        elements = NSMutableDictionary.alloc()
        elements = [:]
        views = Views()
    }
}
```

Fig. 27. "didStartElement" method in Swift

Finally, we show the implementation in Objective-C (Fig. 28) and Swift (Fig. 29) of the method “foundCharacters”. In this case, we can see the same differences again. In addition, Swift needs less characters because it omitted the use of the reserved word “break”.

```
- (void)parser:(NSXMLParser *)parser foundCharacters:(NSString *)string
{
    int number = [string intValue];

    switch(number){
        case 0:
            [button setText: cad];
            break;
        case 1:
            [input setText: cad];
            break;
        default:
            NSLog(@"Default case");
            break;
    }
}
```

Fig. 28. Implementation of the "foundCharacters" method in Objective-C

```
func parser(parser: NSXMLParser!, foundCharacters string: String!)
{
    var number: Int = string.toInt()

    switch(number){
        case 0:
            button!.setText(cad)
        case 1:
            input!.setText(cad)
    default:
        println("Default case")
    }
}
```

Fig. 29. Implementation of the "foundCharacters" method in Swift

1) Quantitative Analysis

Now, we show the quantitative analysis about the three methods which were implemented. First of all, we show the comparison between Objective-C (OC) and Swift (S) about the lines with code in the Table 1. As we can see, the difference depends on the method but it is irrelevant. For this reason, we cannot say that with Swift you could write much less code lines.

TABLE 1
COMPARISON: LINES WITH CODE

Lines	OC	S	Difference	%
beginParsing	6	6	0	0
didStartElement	6	7	-1	14,28
foundCharacters	12	9	3	-33,33

SECONDLY, IN THE

Table 2 we show the comparison about the numbers of words that we had needed to implement the methods. In this case, we used less words in the case of Swift but the difference is insignificant. Swift is a few less verbose than Objective-C as we explained in the previous sections.

TABLE 2
COMPARISON: WORDS

Words	OC	S	Difference	%
beginParsing	23	22	1	-4,54
didStartElement	34	31	3	-9,67
foundCharacters	30	28	2	-7,14

Finally, we measured the number of characters of each method in both programming languages. In this case, we did not count the whitespaces. In the Table 3 we can see that Swift needs around 11% less characters than Objective-C. This is because Swift has changed the access operators, does not use pointers and removed the obligation of the use the semicolon and some reserved words.

TABLE 3
COMPARISON: CHARACTERS WITHOUT SPACES

Characteres	OC	S	Difference	%
beginParsing	256	226	30	-13,27
didStartElement	401	360	41	-11,38
foundCharacters	211	187	24	-12,83

VII. CONCLUSION

Apple has achieved to create a modern programming language with same of the best functionalities of other programming languages like C#, Java, JavaScript, PHP,

Python, Ruby, and Scala, among others. Furthermore, Swift incorporates the functional programming as other programming languages have adopted to provide more possibilities to developers. In addition, Swift adds new possibilities for doing easier and more effective the applications development owing to the fact that: they have done change in the syntax, classes, variables, functions, operators, and data structures; they have improved the "Switch"; they have removed of the pointers.

Furthermore, we can see that using Swift, developers need less characters to program the same code because Swift has simplified the syntax but they need the same numbers of lines and words to program.

Because of these reasons, it seems a wise decision of Apple the creation of a new programming language with more abstraction level than Objective-C to facilitate the application development but with all power of other current programming languages and the elimination of the obsolete Objective-C syntax. Moreover, this abstraction allows to make less mistakes and be more comfortable to developers without a flexibility loss and maintenance the retro-compatibility with previous code.

In conclusion, Apple has created a programming language with the necessary abstraction and functionalities of this age and, in certain cases, they have improved the current functionalities as they have done with the "Switch". Besides, they have created a programming language to improve their ecosystem due to the fact that Swift is not a competitor of Objective-C nor its evolution, Swift is a programming language ready to coexist with Objective-C and to give another possibility to develop to developers.

ACKNOWLEDGMENT

This work was performed by the "Ingeniería Dirigida por Modelos MDE-RG" research group at the University of Oviedo under Contract No. FUO-EM-086-14 of the research project "Proyecto Visio". Project partially-financed by Zed Worldwide S.A.

REFERENCES

[1] Apple Inc., "Swift," https://developer.apple.com/swift/, 2015. [Online]. Available: https://developer.apple.com/swift/. [Accessed: 17-Apr-2015].
[2] Apple Inc., The Swift Programming Language. 2014.
[3] "LLVM," http://llvm.org/, 2000. [Online]. Available: http://llvm.org/.
[4] C. A. Lattner, "LLVM : An Infrastructure for Multi-Stage Optimization," University of Illinois, 2002.
[5] @adamjleonard, @thinkclay, and @cesar_devers, "Swift Toolbox," http://www.swifttoolbox.io/, 2014. [Online]. Available: http://www.swifttoolbox.io/. [Accessed: 17-Apr-2015].
[6] E. González, H. Fernández, and V. Díaz, "General purpose MDE tools," Int. J. Interact. Multimed. Artif. Intell., vol. 1, pp. 72–75, 2008.
[7] E. R. Núñez-Valdez, O. Sanjuan-Martinez, C. P. G. Bustelo, J. M. C. Lovelle, and G. Infante-Hernandez, "Gade4all: Developing Multi-platform Videogames based on Domain Specific Languages and Model Driven Engineering," Int. J. Interact. Multimed. Artif. Intell., vol. 2, no. Regular Issue, pp. 33–42, 2013.
[8] R. Gonzalez-Crespo, S. R. Aguilar, R. F. Escobar, and N. Torres, "Dynamic, ecological, accessible and 3D Virtual Worlds-based Libraries using OpenSim and Sloodle along with mobile location and NFC for checking in," Int. J. Interact. Multimed. Artif. Intell., vol. 1, no. 7, pp. 63–69, 2012.
[9] C. G. García, C. P. García-Bustelo, J. P. Espada, and G. Cueva-Fernandez, "Midgar: Generation of heterogeneous objects interconnecting applications. A Domain Specific Language proposal for Internet of Things scenarios," Comput. Networks, vol. 64, no. C, pp. 143–158, 2014.
[10] TIOBE Software BV, "TIOBE Index," http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, 2014. [Online]. Available: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html. [Accessed: 15-May-2015].
[11] C. Zapponi, "GitHub," 2014. [Online]. Available: http://githut.info/. [Accessed: 15-May-2015].
[12] Apple Inc., "Xcode." 2015.
[13] Apple Inc., Using Swift with Cocoa and Objective-C. 2014.

Cristian González García is a Technical Engineering in Computer Systems and M.S. in Web Engineering from School of Computer Engineering of Oviedo in 2011 and 2013 (University of Oviedo, Spain). Currently, he is a Ph.D. candidate in Computers Science.
His research interests are in the field of the Internet of Things, Web Engineering, Mobile Devices and Modeling Software with DSL, and MDE.

Jordán Pascual Espada is a Research scientist at Computer Science Department of the University of Oviedo. Ph.D. from the University of Oviedo in Computer Engineering.
His research interests include the Internet of Things, exploration of new applications and associated human computer interaction issues in ubiquitous computing and emerging technologies, particularly mobile and Web.

B. Cristina Pelayo G-Bustelo is a Lecturer in the Computer Science Department of the University of Oviedo. Ph.D. from the University of Oviedo in Computer Engineering.
Her research interests include Object-Oriented technology, Web Engineering, eGovernment, Modeling Software with BPM, DSL and MDA.

Juan Manuel Cueva Lovelle is a Mining Engineer from Oviedo Mining Engineers Technical School in 1983 (Oviedo University, Spain). Ph. D. from Madrid Polytechnic University, Spain (1990). From 1985 he is Professor at the Languages and Computers Systems Area in Oviedo University (Spain). ACM and IEEE voting member.
His research interests include Object-Oriented technology, Language Processors, Human-Computer Interface, Web Engineering, Modeling Software with BPM, DSL and MDA.