Università
della
Svizzera
italiana

Faculty
of Informatics

Bachelor Thesis

June 21, 2017

Parsing & Modeling Swift Systems

Alessio Buratti

Abstract

Swift is an emerging programming language, mainly used to develop applications for Apple devices. It is widely used and its popularity is still increasing.

We developed PMSS, a toolkit, written in Swift, that parses Swift applications and exports a model of their source code according to the FAMIX meta-modeling standard. These models can be analyzed using our iOS/macOS application or Moose, a language-independent platform for software and data analysis.

Advisor
Prof. Michele Lanza
Assistant
Roberto Minelli

Advisor's approval (Prof. Michele Lanza): Date:

Contents

1 Introduction

1.1 Goal

Software systems are constantly changing, old modules and components are being deprecated and replaced by more modern ones while new features are implemented and the existing ones are optimized.[7] Code maintenance is a complicated task, to be done, a deep knowledge and understanding of the system is necessary.[6]

Reverse engineering is the process of analyze existing software systems and understand them.[3] Different tools can help the engineers to achieve this result. The Parsing & Modeling Swift Systems (PMSS) toolkit provides to Swift and Cocoa developers a way to easily get a language independent representation of a system using the FAMIX metamodel structure.

1.2 Motivation

Swift is an open source language[1] that can be used to develop any kind of project, starting from simple scripts and command line tools to more complicated apps and web-app, in both the Apple and the Linux[2] environment. The main reason that brought us to work on this project is to give to developers the possibility to easily model their system using either a stand-alone and native command line tool or a web service.

2 State Of The Art

There are several modeling tools already used by software engineerings.

An example is `jdt2famix`[3], an open source project that offers the mechanism for producing `MSE` files out of Java code. It is based on JDT Core and Fame for Java, and it requires Java 8. jdt2famix is written in Java and, like the tool that we developed, it is implemented as a standalone project.

Moose Brewer and Penumbra are two Java's modeling tools. Brewer is an Eclipse plugin to generate MSE file from an Eclipse project. Penumbra is a Visualworks application that makes the Eclipse application steerable within Smalltalk.[2]

Another example is `Parsing & Modeling C# System (PMCS)`[3], developed by Ermira Daka as Master Project. PMCS is written in C# and, as well jdt2famix, it is a standalone tool that generates `MSE` files representing a given software system.

2.1 Related work

This work was inspired by CodeCity, an integrated environment for software analysis.[7] In CodeCity software systems are visualized using the metaphor of the city, classes are represented as buildings and packages are represented as districts. The buildings reside in their specific district and, by the size, the height, and the color of the building, it is possible to understand the number of attributes, methods and line of code of each class.

[1]see: https://developer.apple.com/swift/blog/?id=34
[2]see: https://swift.org/blog/swiftlinuxport/
[3]see: https://github.com/girba/jdt2famix

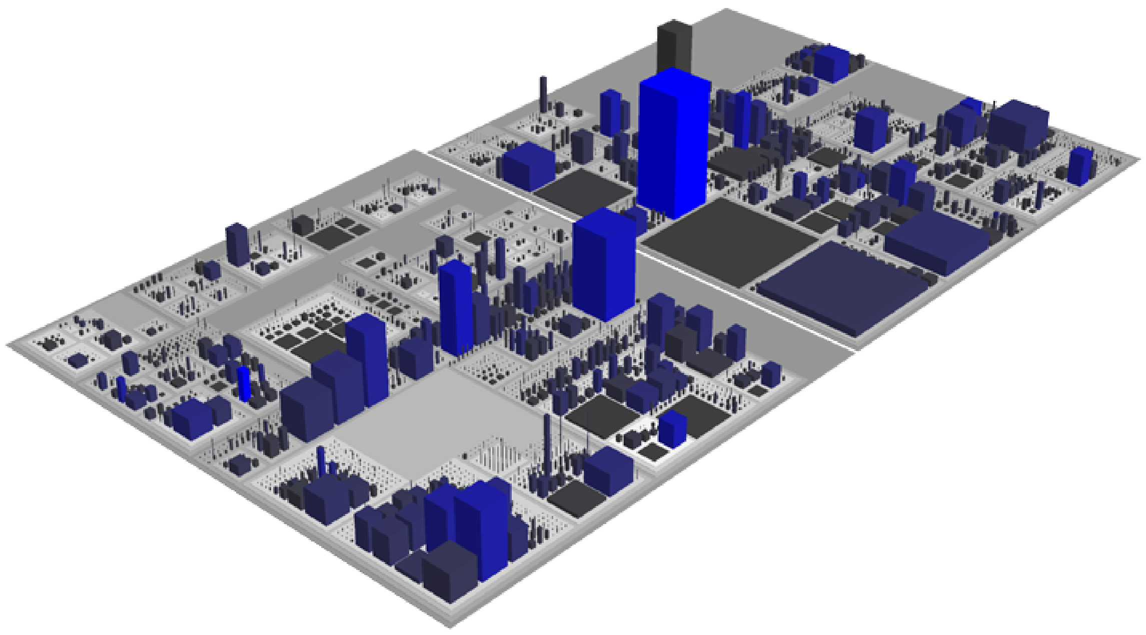


Figure 1. CodeCity visualization of JDK v1.5

3 Swift and the meta-modeling tools in a nutshell

In this section we provide an overview of the concepts needed to better understand the contents of this document, what we developed, and how.

3.1 The Swift Programming Language

Swift is a general-purpose programming language built using a modern approach to safety, performance, and software design patterns.[4] Designed by Chris Lattner and Apple Inc,[5] it was first announced at the 2014 Worldwide Developer Conference (WWDC14).[6]

Swift supports multiple paradigms, like imperative, object-oriented, functional, generic, event-driven and concurrent programming.

3.1.1 Some "Swifty" code examples

Swift supports all the standard features of an object-oriented programming language. It is possible to assign a value to a constant or a to variable as follows

```swift
var someVariable = "Hello"
let someConstant = 10
```

And a function can be defined as follows

```swift
func sayHello(to person: Person) {
    print("Hello \(person.name)")
}
```

In Swift you can decide to describe an object as a class or as a struct depending on your needs. Swift classes support all the classic object-oriented features like inheritance, polymorphism and encapsulation. As it happens for most of the object-oriented languages, class' instances are allocated in the dynamic memory.

[4]see: https://swift.org/about/
[5]see: http://nondot.org/sabre/
[6]see: https://developer.apple.com/swift/blog/?id=34

Swift structs, on the other hand, are value types, therefore they are allocated on the stack and do not support inheritance, but differently from most languages, they do have methods and they do support polymorphism if they conform to some protocols.

Swift protocols can be seen like Java's interfaces but differently from interfaces a method can have a default implementation. To be more specific, in Swift, a protocol defines a blueprint of methods, properties, and other requirements that suit a particular task or piece of functionality. The protocol can then be adopted by a class, structure, or enumeration to provide an actual implementation of those requirements. Any type that satisfies the requirements of a protocol is said to conform to that protocol. [1]

```swift
// A simple class in Swift
class Dog: Animal {
    let name: String

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

Listing 1

```swift
// A protocol that describes which methods
// a class, struct or enum should have in
// order to be seen as a Drivable entity.
protocol Drivable {
    func drive(to location: Location)
}

// Two structs, both conforming to Drivable
struct Car: Drivable {
    let engine: Engine

    // Required by Drivable
    func drive(to location: Location) {

    }
}

struct Horse: Drivable {
    let name: String

    // Required by Drivable
    func drive(to location: Location) {

    }
}

let drivables: [Drivable] = [Car(engine: Engine()), Horse(name: "Spirit")]
for drivable in drivables {
    drivable.drive(to: (latitude: 0.00000, longitude: 0.00000))
}
```

Listing 2

Like most of the modern languages, Swift supports the functional programming paradigm. Using `enums` and `indirect enums` you can describe data types like you would to in Haskell or Scala.

```swift
indirect enum Tree<Element> {
    case empty
    case node(Tree, Element, Tree)
}

enum SomeValue {
    case intValue(Int)
    case boolValue(Bool)
}
```

Swift supports extension to existing types and has a powerful pattern matching.

```swift
extension Tree {
    var elements: [Element] {
        switch self {
        case .empty:
            return []
        case let .node(leftChild, element, rightChild):
            return leftChild.elements + [element] + rightChild.elements
        }
    }
}

extension SomeValue: CustomStringConvertible {
    var description: String {
        switch self {
        case let .intValue(value): return "\(value)"
        case let .boolValue(value): return "\(value)"
        }
    }
}

let someValue = 42

switch someValue {
case 0..<10: print("someValue is between 0 and 10")
case 10...: print("someValue is biggern then 10")
case ..<0: print("someValue is negative")
default: break
}
```

After this introduction to the language we will see how Swift can be modeled and the tools that we used.

3.2 FAMIX: a language independent meta-model

FAMIX is a language independent meta-model that can represent in a uniform way multiple object-oriented and procedural languages. More generally, FAMIX describes a set of rules that can be used to represent a software system independently from the programming language in which the system is written. FAMIX stands for FAMOOS Information Exchange Model. The FAMIX core, synthesized in Figure 2, describes the main entities that take part in a object-oriented software system and the relations between these entities. The main entities are Package, Class, Method, Attribute, and the relationships between them, are Inheritance, Access and Invocation.[4]

A `Package` is an entity that can encapsulated other packages and classes. A `Class` is a set of methods and attributes and the `Inheritance` relation is used to indicate which class subclass another one. A `Method` is defined as an action that can be executed by a `Class` and the `Invocation` relation indicates how methods invoke other ones. An `Attribute` is defined as a `Class`' property and the `Access` relation indicates which methods access each attribute.

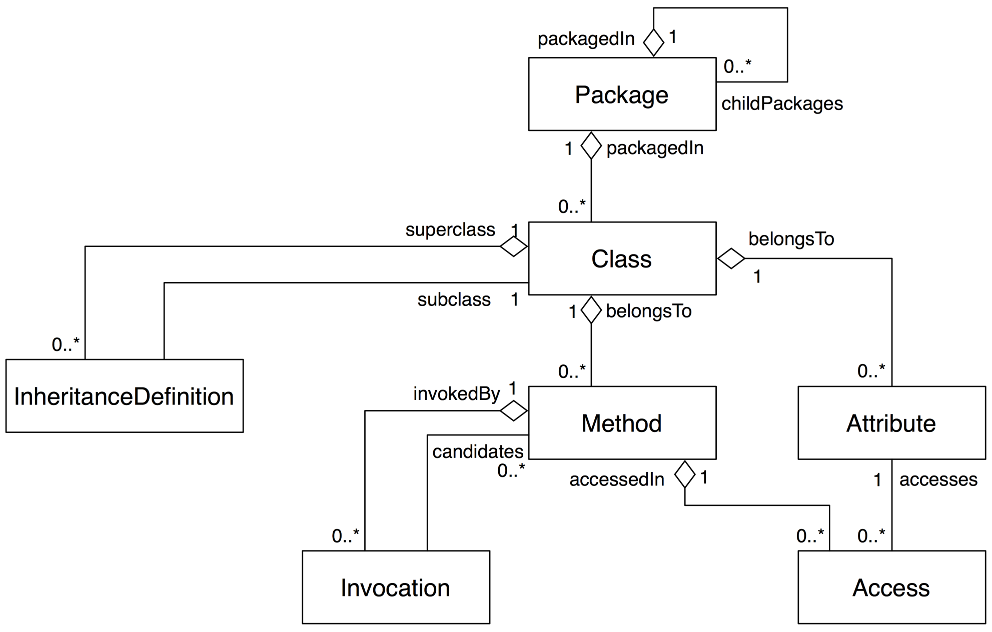
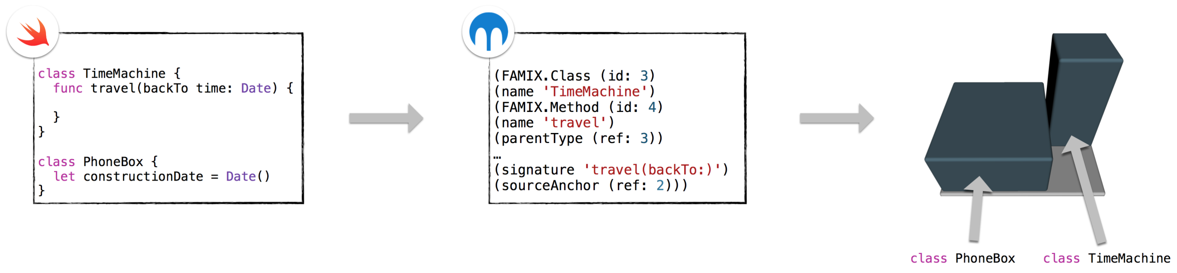


Figure 2. The core of the FAMIX meta-model

3.3 MOOSE: a platform for software and data analysis

MOOSE is an extensible, language-independent, environment for reengineering object-oriented systems [5] developed at the University of Bern.



```
class TimeMachine {
  func travel(backTo time: Date) {

  }
}

class PhoneBox {
  let constructionDate = Date()
}
```

```
(FAMIX.Class (id: 3)
(name 'TimeMachine')
(FAMIX.Method (id: 4)
(name 'travel')
(parentType (ref: 3))
…
(signature 'travel(backTo:)')
(sourceAnchor (ref: 2)))
```

Figure 3. On the left, an example of Swift source code, in the middle a piece of the FAMIX meta-model describing that source code, on the right the corresponging of rendered city

As you can see in Figure 3, MOOSE stands between a specific programming language and a set of different reverse engineering tools. MOOSE is a platform used to analyze system complexity and class hierarchies, classes and their internal complexity, package usage and system distribution in abstract space. At the same time MOOSE is also a visualization platform and offers the possibility to visualize software systems using powerful tools like Mondrian and CodeCity. [3]

`MSE` is the default file format supported by Moose. It is a generic file format and can describe any model. It is similar to XML, the main difference being that instead of using verbose tags, it makes use of parentheses to denote the beginning and ending of an element. [7]

The snippet 3 provides an example of a small model:

[7]see: http://themoosebook.org/book/index.html

```
((FAMIX.Namespace (id: 1)
  (name 'aNamespace'))
 (FAMIX.Package (id: 201)
  (name 'aPackage'))
 (FAMIX.Package (id: 202)
  (name 'anotherPackage')
  (parentPackage (ref: 201)))
 (FAMIX.Class (id: 2)
  (name 'ClassA')
  (container (ref: 1))
  (parentPackage (ref: 201)))
 (FAMIX.Method
  (name 'methodA1')
  (signature 'methodA1()')
  (parentType (ref: 2))
  (LOC 2))
 (FAMIX.Attribute
  (name 'attributeA1')
  (parentType (ref: 2)))
 (FAMIX.Class (id: 3)
  (name 'ClassB')
  (container (ref: 1))
  (parentPackage (ref: 202)))
 (FAMIX.Inheritance
  (subclass (ref: 3))
  (superclass (ref: 2))))
```

Listing 3. Example of a model according to the FAMIX standard.

The main goal of our project is to develop a tool that, given a specific Swift project developed using Xcode, is able to generate a `MSE` file containing a model that describes the project structure, according to the FAMIX standards.

4 PMSS: Parsing & Modeling Swift Systems

During the project we developed PMSS, a set of instruments to generate the model of a Swift system and analyze it. The main module the we developed is called `PMSSCore`, it is written in Swift and it is built on top of SourceKit, a framework for supporting IDE features. The PMSSCore module is then used in three different Swift applications that we built. The first one is a command line tool, called `pmss`. `pmss` can be built on any macOS and Linux system where SourceKit is installed. It allows developers to generate a `MSE` file directly from the command line and it is as easy as `./pmss /path/to/xcodeproj`.

The second application that we made is a macOS Cocoa app. The PMSSCore module is directly imported and built inside the application and this allows the user to not just generate the `MSE` file but also to graphically visualize the model in a 3D rendered view, directly from the application. The generated 3D model describes the user's software system using the metaphor of the city.[7]

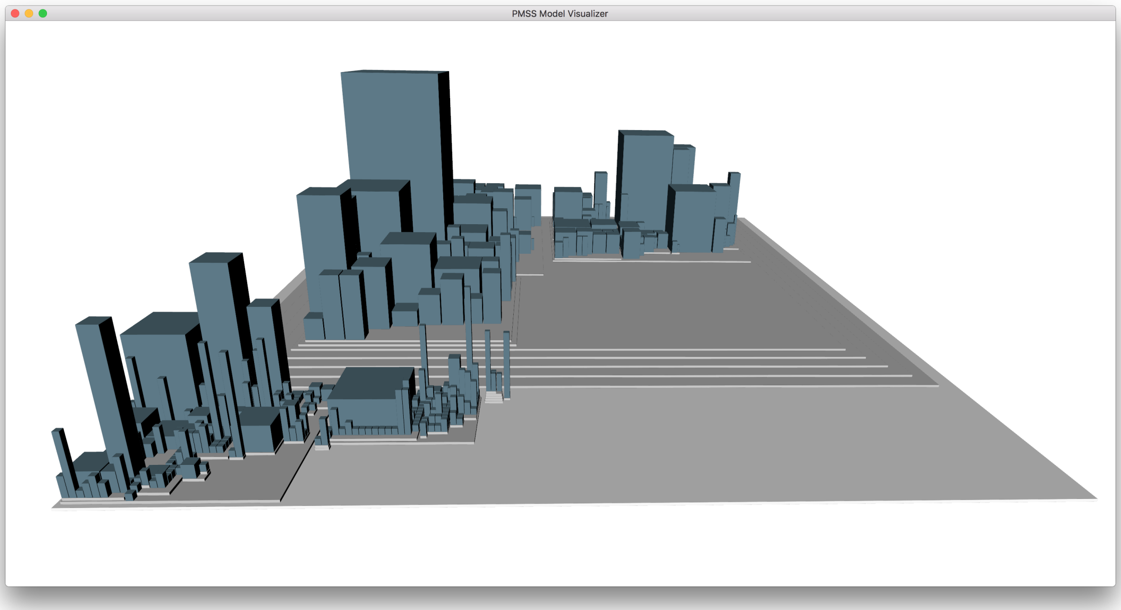


Figure 4. The representation of a model using our macOS app

The last application that we made using the PMSSCore module is a simple web-app written in Swift, using the web framework Vapor.[8] Our web-app offers some APIs to generate and download a `MSE` file representing a Swift system that is contained in a remote git repository. It is possible to choose the repository branch that we want to model and, if the Xcode project is composed by multiple targets, the one that we prefer.

Using our web API we were able to migrate the "PMSS macOS Model Visualizer" to iOS, delegating the modeling task to the server, since it is not possible to interact with SourceKit inside the iOS environment.

The Figure 5 shows how the PMSS toolkit is built.

[8]see: https://github.com/vapor/vapor

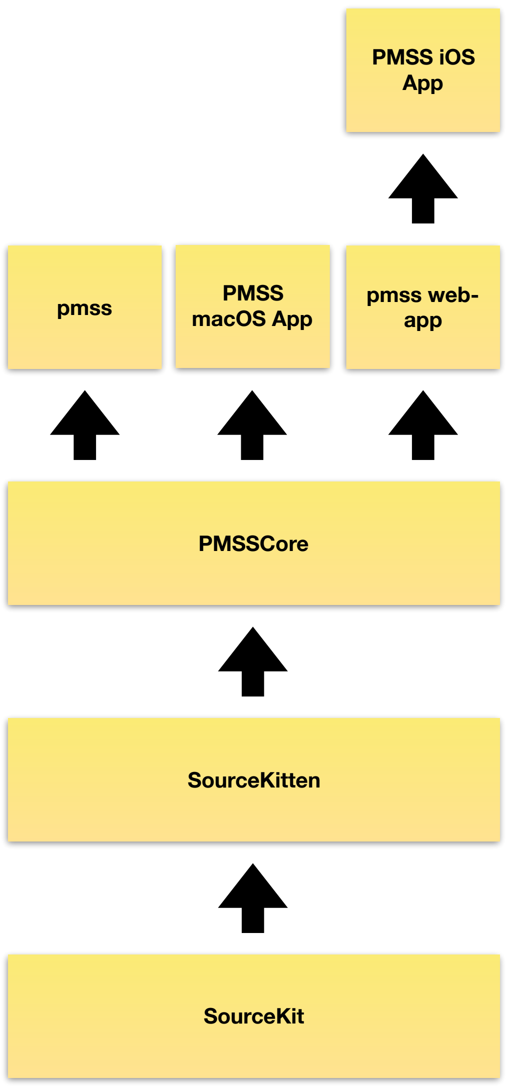


Figure 5. The PMSS toolkit hierarchy

4.1 SourceKit: the core of Swift's source code manipulation

SourceKit is a framework designed by Apple for supporting IDE features like indexing, syntax-coloring and code-completion.

The stable C API for SourceKit is provided via the `sourcekitd.framework` which uses an XPC service for process isolation and the `libsourcekitdInProc.dylib` library which is in-process.

SourceKit is capable of "indexing" source code, responding with which ranges of text contain what kinds of source code. For example, SourceKit is capable of telling you that "the source code on line 2, column 9, is a reference to a struct". [9]

4.2 SourceKitten: high-level SourceKit interaction

SourceKitten is a high level, community developed, binding framework used to interact with SourceKit. It is available both as a command line tool and as framework. When we developed PMSSCore, we imported the SourceKittenFramework directly in our project and we used it to handle the interaction with SourceKit.[10]

SourceKitten wraps several SourceKit features and expose them hiding the complexity of the SourceKit APIs. In the PMSS toolkit we used the SourceKittenFramework to index each source file of an Xcode project. From the result of this operation we took the substructure tree and the entity tree of each class, struct and protocol of the file. The substructure tree represents the structural information of the given Swift entity, like the accessibility level, the length (in characters) of the entity, its name and type and the structure of the entities declared inside of the entity's scope.

[9]see: https://github.com/apple/swift/tree/master/tools/SourceKit
[10]see: https://github.com/jpsim/SourceKitten

The entity tree, on the other side, indicates the `usr` (unique symbol resolution identier) of each entity and the lines and offsets where the symbol, is declared and used. In the entity tree, similarly to the substructure tree, it is possible to recursively retrive the entity information of each entity declared and used inside the given entity's tree.

Lets see an example of how a very simple Swift class is represented once that SourceKit indexes it. Consider the code Listing 4 as the content of the file Mario.swift. Mario is a simple class with one instance method and one instance variable.

```swift
enum Mushroom {
    case redMushroom
    case greenMushroom
}

class Mario {
    var collectedStars = 0

    func eatMushroom(_ mushroom: Mushroom) {

    }
}
```

Listing 4. An example of a Swift source file

The code Listing 5 provides the structure of the class Mario, showed in the code 4

```
{
  "key.diagnostic_stage" : "source.diagnostic.stage.swift.parse",
  "key.substructure" : [
    {
      "key.bodylength" : 93,
      "key.nameoffset" : 6,
      "key.accessibility" : "source.lang.swift.accessibility.internal",
      "key.length" : 107,
      "key.substructure" : [
        {
          "key.nameoffset" : 22,
          "key.accessibility" : "source.lang.swift.accessibility.internal",
          "key.length" : 22,
          "key.name" : "collectedStars",
          "key.kind" : "source.lang.swift.decl.var.instance",
          "key.offset" : 18,
          "key.namelength" : 14,
          "key.setter_accessibility" : "source.lang.swift.accessibility.internal"
        },
        {
          "key.bodylength" : 14,
          "key.nameoffset" : 55,
          "key.accessibility" : "source.lang.swift.accessibility.internal",
          "key.length" : 55,
          "key.substructure" : [
            {
              "key.nameoffset" : 0,
              "key.typename" : "Mushroom",
              "key.length" : 20,
              "key.name" : "mushroom",
              "key.kind" : "source.lang.swift.decl.var.parameter",
              "key.offset" : 67,
              "key.namelength" : 0
            }
          ],
          "key.name" : "eatMushroom(_:)",
          "key.kind" : "source.lang.swift.decl.function.method.instance",
          "key.namelength" : 33,
          "key.offset" : 50,
          "key.bodyoffset" : 90
        }
      ],
      "key.runtime_name" : "_TtC8__main__5Mario",
      "key.name" : "Mario",
      "key.kind" : "source.lang.swift.decl.class",
      "key.namelength" : 5,
      "key.offset" : 0,
      "key.bodyoffset" : 13
    }
  ],
  "key.offset" : 0,
  "key.length" : 108
```

Listing 5. The substructure tree generated by SourceKit

The code of the Listing 6 shows the entity tree of the class Mario.

```json
[
    {
      "key.line" : 1,
      "key.name" : "Mario",
      "key.usr" : "s:5Mario",
      "key.column" : 7,
      "key.kind" : "source.lang.swift.decl.class",
      "key.entities" : [
        {
          "key.line" : 2,
          "key.name" : "collectedStars",
          "key.usr" : "s:5Mario14collectedStarsSi",
          "key.column" : 9,
          "key.kind" : "source.lang.swift.decl.var.instance"
        },
        {
          "key.line" : 4,
          "key.name" : "eatMushroom(_:)",
          "key.usr" : "s:5Mario11eatMushroomFVS_8MushroomT_",
          "key.column" : 10,
          "key.kind" : "source.lang.swift.decl.function.method.instance",
          "key.entities" : [
            {
              "key.line" : 4,
              "key.name" : "Mushroom",
              "key.usr" : "s:8Mushroom",
              "key.column" : 34,
              "key.kind" : "source.lang.swift.ref.struct"
            }
          ]
        }
      ]
    }
  ]
```

Listing 6. The entity tree generated by SourceKit.

Using the substructure tree we can see that the structure `Mario` is the declaration of a class, its accessibility type is `internal` and the total length of its body is 107 characters. At the same time, we can see the information of the Swift structures declared inside `Mario`, for example, `starCollected` is a class' property of type `Int` and its accessibility is internal again while `eatMushroom(_:)` is a method. Since the substructure tree is a recursive model, we can see that the method `eatMushroom(_:)` has a substructure as well and this goes on until the whole structure has been explored.

In the entity tree we can see the line and the column where the specific identifier is written in the source code, for example we can see that the declaration of the method `eatMushroom(_:)` starts at character 10 of line 4. Informations like the usr are extremely useful to model the `Invocation` and `Access` FAMIX core entity.

At the same time, thanks to the recursively way that SourceKit uses to model these trees, it is possible to easily walk through nested types and model Swift edge cases like the one showed in Listing 7

```swift
enum SomeEnum {
    case someCase
    case someOtherCase

    static var someStaticComputedVariable: Int {
        func aLocalMethod() -> Int {
            class ALocalLocalClass {
                lazy var someLazyVariable: Int = {
                    return 1
                }()
            }
            return ALocalLocalClass().someLazyVariable
        }
        return aLocalMethod()
    }
}
```

Listing 7. Example of a model according to the FAMIX standard.

5 PMSS Model Visualizer: Our iOS and macOS Cocoa applications

Swift is mainly used in the Cocoa development environment so it makes sense to provide a native application for iOS and macOS.

5.1 iOS

Since it is not yet possible to compile SourceKit inside an iOS application the app delegates to our `pmss web-app` the generation of the `MSE` file. The user can specify a remote git repository that contains an Xcode project, the branch name he wants to model and, if the Xcode project is composed by multiple targets, he can choose the one he prefers. Once that the iOS application receives the MSE of the requested project, it saves it locally in order to let the user see it offline and then it renders the 3D computed model of the city using SceneKit.

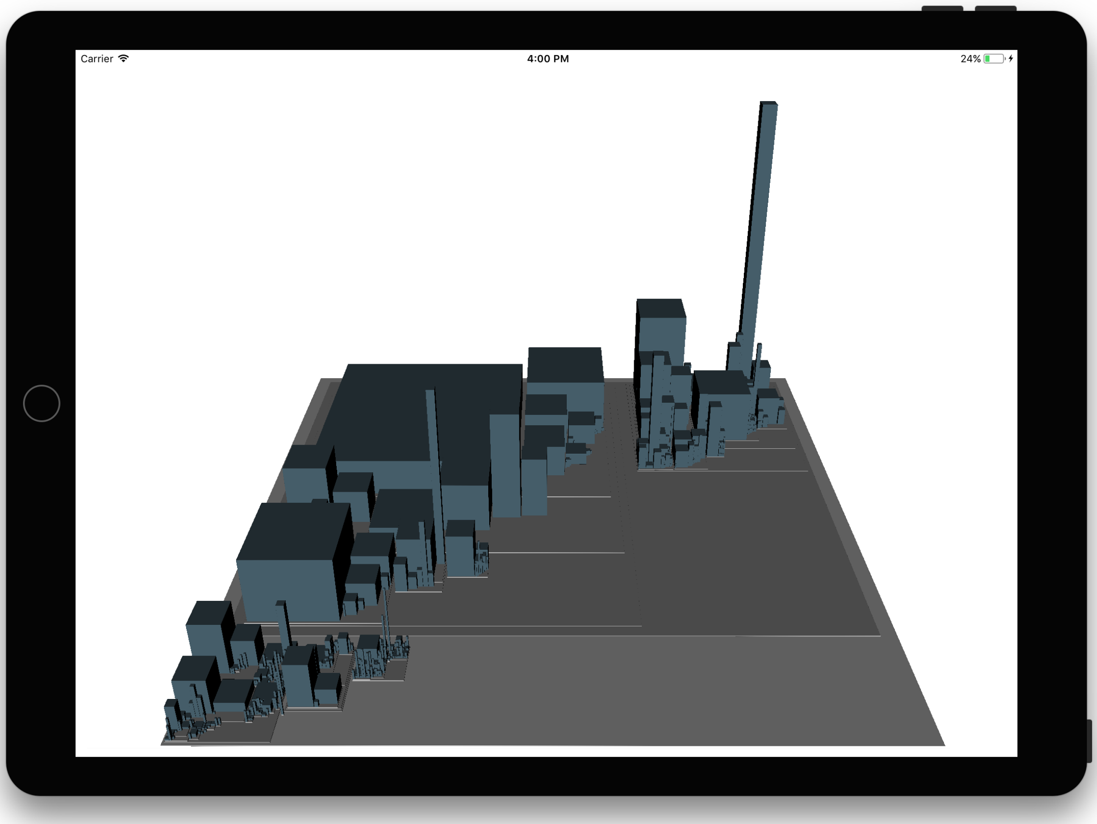
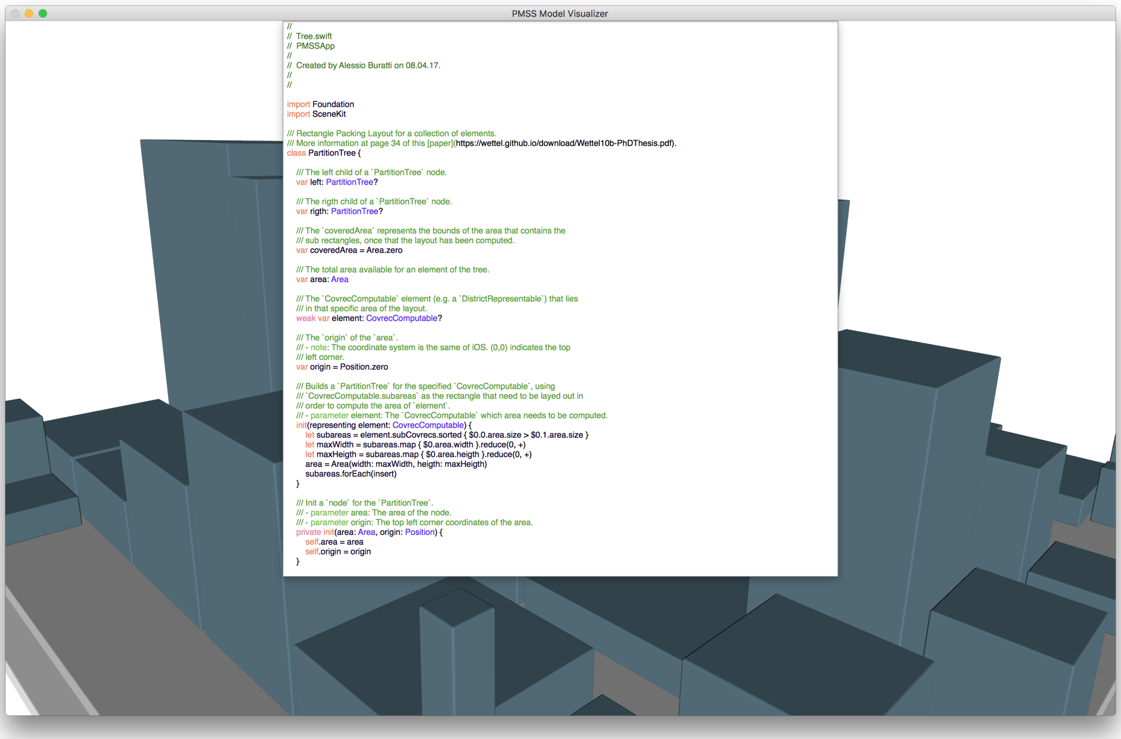


Figure 6. A system rendered in our iOS app

5.2 macOS

On macOS there is no need to use a server to generate the `MSE` file. SourceKit is already installed in the system. The macOS version of the app allows the user to generate a `MSE` file starting directly from an .xcodeproj. The application then works using the same logic of the iOS one, in fact, both the app are part of the same Xcode project and share most of their code. Since the city is rendered starting from a `MSE` file, it is possible to load and analyze the model of any software system, independently from the language, as long as it is formatted using the FAMIX meta-model. If the `MSE` is generated directly by our application, starting from an Xcode project, it is also possible to visualize and edit the source files.



```
//
//  Tree.swift
//  PMSSApp
//
//  Created by Alessio Buratti on 08.04.17.
//
//

import Foundation
import SceneKit

/// Rectangle Packing Layout for a collection of elements.
/// More information at page 34 of this [paper](https://wettel.github.io/download/Wettel10b-PhDThesis.pdf).
class PartitionTree {

    /// The left child of a `PartitionTree` node.
    var left: PartitionTree?

    /// The rigth child of a `PartitionTree` node.
    var rigth: PartitionTree?

    /// The `coveredArea` represents the bounds of the area that contains the
    /// sub rectangles, once that the layout has been computed.
    var coveredArea = Area.zero

    /// The total area available for an element of the tree.
    var area: Area

    /// The `CovrecComputable` element (e.g. a `DistrictRepresentable`) that lies
    /// in that specific area of the layout.
    weak var element: CovrecComputable?

    /// The `origin` of the `area`.
    /// - note: The coordinate system is the same of iOS. (0,0) indicates the top
    /// left corner.
    var origin = Position.zero

    /// Builds a `PartitionTree` for the specified `CovrecComputable`, using
    /// `CovrecComputable.subareas` as the rectangle that need to be layed out in
    /// order to compute the area of `element`.
    /// - parameter element: The `CovrecComputable` which area needs to be computed.
    init(representing element: CovrecComputable) {
        let subareas = element.subCovrecs.sorted { $0.0.area.size > $0.1.area.size }
        let maxWidth = subareas.map { $0.area.width }.reduce(0, +)
        let maxHeigth = subareas.map { $0.area.heigth }.reduce(0, +)
        area = Area(width: maxWidth, heigth: maxHeigth)
        subareas.forEach(insert)
    }

    /// Init a `node` for the `PartitionTree`.
    /// - parameter area: The area of the node.
    /// - parameter origin: The top left corner coordinates of the area.
    private init(area: Area, origin: Position) {
        self.area = area
        self.origin = origin
    }
```

Figure 7. A class of the PMSS Model Visualizer.

5.3 SceneKit: a high-level 3D graphics framework

SceneKit is an Apple framework to work with 3D scenes, it was first released for macOS in 2012, successively for iOS in 2014, for tvOS in 2015 and finally for watchOS in 2016. It Is is written in Objective-C and it is built on top of OpenGL[11] and Metal[12]. It offers high-level APIs to describe scenes and animations [13]. SceneKit incorporates a physics engine and a particle generator and allows developer to easily work with scene components like geometry, materials, lights, and cameras. [14]

Since the goal of our application is to be as clear as possible the whole graphic scene is composed by SceneKit primitives, in particular boxes.

The code 8 shows a simple example of how it is possible to render a 3D cube using SceneKit.

[11]see: https://www.opengl.org
[12]see: https://developer.apple.com/metal/
[13]see: https://developer.apple.com/documentation/scenekit
[14]https://developer.apple.com/scenekit

```swift
let viewSize = CGSize(width: 300.0, height: 300.0)

let view = SCNView(frame: NSRect(origin: CGPoint.zero, size: viewSize))
let scene = SCNScene()

// Creates a node that can be used to render a cube on the scene
let box = SCNBox(width: 10.0, height: 10.0, length: 10.0, chamferRadius: 0.5)
let node = SCNNode(geometry: box)

// Adds the node that contains the box to the scene
scene.rootNode.addChildNode(node)

view.scene = scene
view.autoenablesDefaultLighting = true
view.allowsCameraControl = true
```

Listing 8. Code to add a simple cube in a SCNScene.

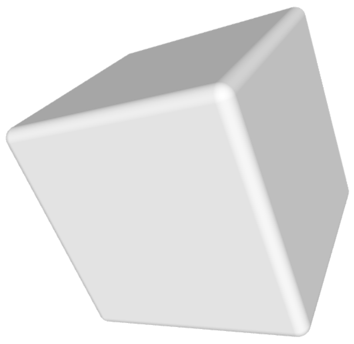


Figure 8. The view generated by the code 8.

5.4 ARKit: augmented reality experience in any iOS application

ARKit is an Apple framework that provides APIs to add augmented reality experiences in an iOS app or game. [15] It was announced during the 2017's Worldwide Developer Conference. At the time of writing this document ARKit is in beta.

The augmented reality feature in the iOS application was added as extra and offers no significant details to the software visualization but since ARKit can interact with SceneKit through a SCNView subclass called ARSCNView, it is easy to render a 3D model in an augmented reality experience once that the model is created.

[15] see: https://developer.apple.com/arkit/

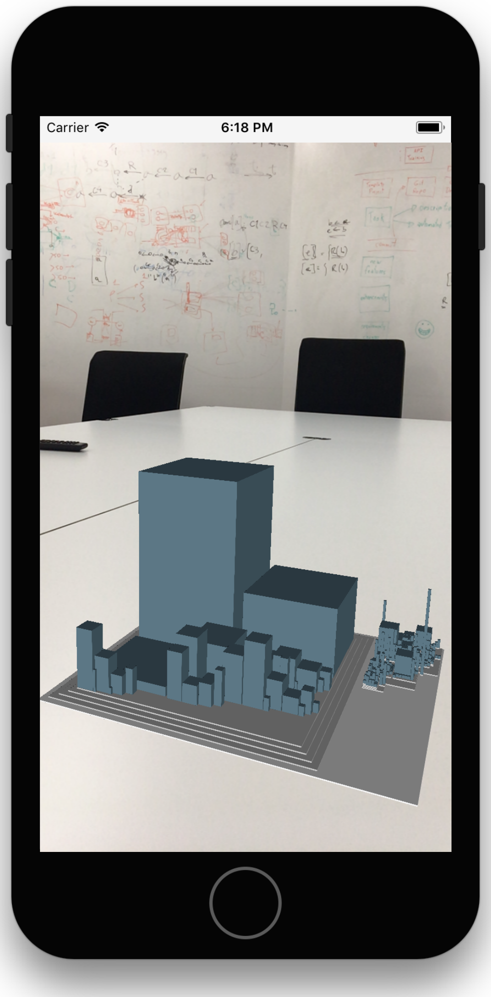


Figure 9. The iOS application, using ARKit, to render a city on a real table.

6 Conclusion

Various research works indicate that the maintenance stage consumes most of the resources needed for a software project [6]. Software visualization tool, like CodeCity, can help developers finding solutions to reverse engineering tasks.[16]

Swift is a young language, but thanks to its modern features, its performance, its portability, and its simplicity we do not doubt that it will be a main pillar in the future of programming languages and software engineering.

PMSS is a toolkit that can be very helpful in the reverse engineering of Cocoa applications and Swift systems in general. Using powerful technologies like SourceKit, it is possible to generate a detailed model of a system, even when Swift nested types would be very difficult to parse.

We believe PMSS is extremely useful for developers and engineers who are already familiar with the MOOSE environment but, thanks to the native model visualizers developed for iOS and macOS it is also possible to analyze Swift systems in a completely independent way.

PMSSCore APIs allow developers to directly include a modeling tool in their application without the need of using a meta-model language like FAMIX.

Given the possibility to visualize and edit the source code using the PMSS Model Visualizer on macOS, our goal for the future is to improve the application and add more advanced IDE features.

[16]see: https://wettel.github.io/codecity-experiment.html

References

[1] Apple Inc. *The Swift Programming Language (Swift 3.1)*.

[2] A. Bergel. Java4Moose.

[3] E. Daka. Parsing and modeling c# systems. 2009.

[4] S. Ducasse, N. Anquetil, M. U. Bhatti, A. C. Hora, J. Laval, and T. Girba. Mse and famix 3.0: an interexchange format and source code model family. 2011.

[5] S. Ducasse, M. Lanza, and S. Tichelaar. The moose reengineering environment. *Smalltalk Chronicles*, 3(2), 2001.

[6] J. C. Granja-Alvarez and M. J. Barranco-García. A method for estimating maintenance cost in a software project: a case study. *Journal of Software Maintenance*, 9(3):161–175, 1997.

[7] R. Wattel and M. Lanza. Software systems as cities. 2010.