Dynamic Analysis Techniques to Reverse Engineer Mobile Applications

Philippe Dugerdil(✉) and Roland Sako

Geneva School of Business Adminsitration,
University of Applied Sciences Western Switzerland,
7 route de Drize, 1227 Geneva, Switzerland
philippe.dugerdil@hesge.ch, roland.sako@gmail.com

Abstract. Nowadays mobile applications have moved to mainstream. Service companies such as IBM advise us to develop on the "Mobile First". Although earlier mobile apps were simple data access front ends, today's apps are quite complex. Therefore the same problem of code maintenance and comprehension of poorly documented apps, as in the desktop world, happen to the mobile today. Hence we need techniques to reverse engineer mobile applications starting from the mere source code. In this paper we present the methodology and suite of tools we developed that helps with the reverse engineering and understanding of mobile apps. The performance of these tools is demonstrated on two case studies of iPhone applications. The contribution of the paper is to show how dynamic analysis techniques can be applied to mobile applications and the techniques we develop to make educated guesses about the role and structure of the classes that make up the app.

Keywords: Mobile application · Dynamic analysis · Reverse engineering · Program understanding

1 Introduction

According to several surveys, mobile business applications are the trend of the day, although not all surveys agree on the strength of the trend [2, 17, 30]. With the growing interest in B2B and B2E mobile apps [17], mobile development becomes mainstream [14, 16]. Then the very same problems of application maintenance and understanding arise as in desktop applications. There are no reasons to believe that mobile apps will be any easier to maintain than desktop ones. In particular the lack of documentation could even be higher, on average, than on traditional desktop platform since these applications are notoriously developed using agile approaches such as Scrum which leaves a lot of freedom to the developer as to what documentation to produce. Then we decided to develop a mobile version of our methodology for the reverse engineering of applications. This is a complete set of techniques and tools to analyze the functional structure of an application [8] to improve its understanding hence its maintenance. Indeed it is known for a long time that to "understand" a large software system, the structural aspects of the system are more important than any single algorithmic component [29]. Since there are several views of software architecture [5], each targeting

P. Lorenz et al. (Eds.): ICSOFT 2015, CCIS 586, pp. 250–268, 2016.
DOI: 10.1007/978-3-319-30142-6_14

a particular purpose, we propose new ones specifically targeted at software understanding. First we developed the functional structure view of the system [8] i.e. the classes, their relationships and the methods that implement some business relevant scenario. Second we developed the classes' time series that let us observe when a given class or set of classes are involved in the execution of some business relevant scenario. These views rests on dynamic analysis techniques i.e. the analysis of the execution trace of the program corresponding to some scenarios (use-cases) relevant to the business. One key problem in dynamic analysis is to cope with the amount of data to process. In fact, the execution trace file can contain several hundreds of thousands of events. To cope with this data volume, we developed a trace segmentation technique [6] that has showed to be very efficient at analyzing the interactions between the components of the system. In [26] we investigated the way to apply our tools and techniques to the analysis of iPhone apps. This led us to extend the interpretation of our tools' and methodology's results in the mobile environment. In this paper we first present our reverse engineering framework for software system (Sect. 2). Section 3 presents the technique we use to recover the use-cases from users of the apps. Next we show the tools we developed specifically to adapt our framework to the reverse engineering of Objective-C applications on the iPhone (Sect. 4). In Sect. 5, we present two case studies and the way the results can be interpreted to make educated guesses about the structure and roles of the classes of the app. Section 6 presents the related work and Sect. 7 concludes the paper.

2 Reverse Engineering Technique

The goal of our reverse engineering process is to recover the functional structure of the program [8] i.e. to analyze what classes or components support the business relevant function of the application. The process starts with the recovery of the use-cases of the system, if they are not readily available from the documentation of the app (which is generally the case), by watching the users interacting with the system. We simply ask a user to go through all the business-relevant scenario and we take note of all the actions he does with the app. In the case of legacy desktop applications we even video-record the actions of the user. But this is not required here because the use-cases for mobile apps are usually much simpler. Next we instrument the source code of the app to be able to generate the execution traces (i.e. the sequence of method calls in a given run of the system). Code instrumentation consists of inserting extra statements in the source code to record events when the methods are executed. An event is generated when the method is entered and exited. Next, the system is run according to the use-cases and the corresponding execution trace is recorded. Finally, an off-line analysis of the execution trace is performed to recover the functional structure of the system using many views. Figure 1 illustrates a simplified version of the reverse engineering process with only the key tasks. This process has been implemented using a set of tools that are presented in Fig. 2. To instrument the source code, many variants exist among which:

- Developing an instrumentor for the programing language of the system;
- Leveraging an AOP environment to inject the "instrumentation aspects" into the code.

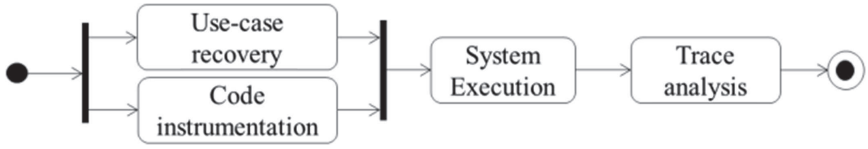

Fig. 1. Reverse engineering process.

Depending on the programming language considered, the second option may not be available. For Objective-C (iPhone) it is indeed the case and we developed our own code instrumentor that will be detailed in the next section. Once the code has been instrumented it is compiled and shipped onto the mobile phone. Then the app is run according to the use-cases and the execution trace is recorded in a file on the device. Next, the file is downloaded from the device and uploaded into a trace database using a trace loader which performs a few integrity checks. Finally, the trace is analyzed using our trace analysis tools. The latter is able to present the information from the trace using several views. There are two formats for the events to be recorded in the execution trace. The first is for method entry and the second for method exit. By recording these two kinds of events, we can reconstruct the call graph with the call hierarchy.

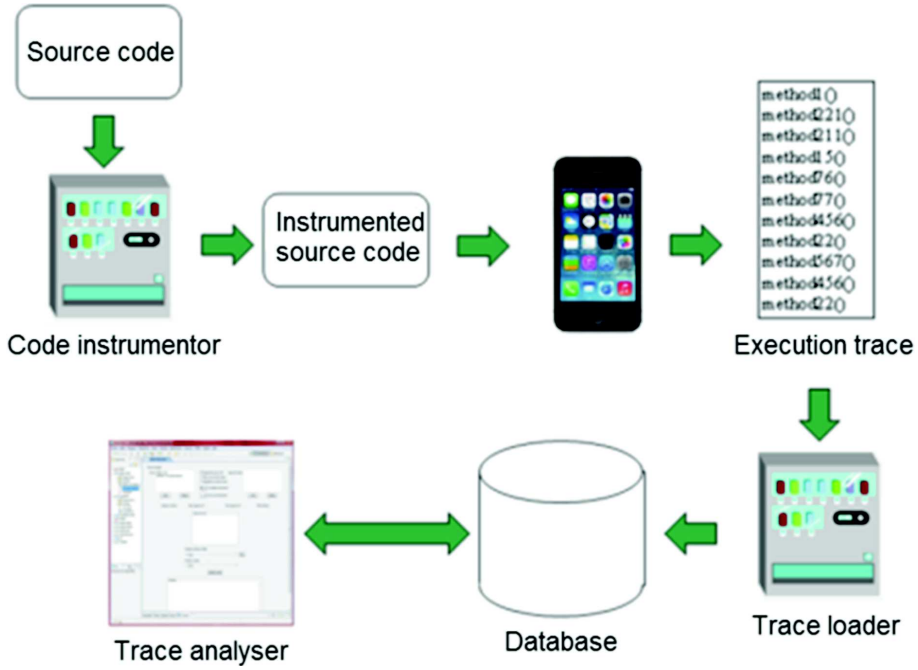

Fig. 2. Tools workflow.

The syntax of the events is the following:
[SCI] [DCI] '['[TN] ']' [Sign] 'AS' [Type] '['[TS] ']' [Param]
or
'END' [SCI] [DCI] '['[TN] ']' [Sign] 'AS' [Type] '['[TS] ']'
With:

[SCI]: Static class identifier: the package name and class name in which the executed method is implemented.

[DCI]: Dynamic class identifier: the package name and class name corresponding to the class of the instance that executed the method.
[TN]: Thread number.
[Sign]: Method signature.
[Type]: Type of the element returned by the method.
[TS]: Time stamp of the event.
[Param]: List of the comma-separated values for the primitive-typed parameters of the method. Non primitive-typed values are replaced by '_'.

The first event represents the entry into a method and the second, headed by the keyword 'END', indicates the exit from the method. The thread number allows us to gather all the events that belong to the same thread for further analysis. Since Objective-C does not have any package construct, the class identifiers of the events uses only the class name (in our tool we append the "default" keyword to replace the package name in the long name of the class).

3 Use Case Recovery

The first step of our approach is to recover the use-cases of the app. This is compulsory since the dynamic analysis rests on the execution of a relevant business scenario. If no user documentation is available, which is the common situation, the technique is first to identify all the user categories (the actors in the UML parlance). Then, we ask each representative of each user categories what he would use the app for, what is for him the expected result of using the app. This represents the "business goal" of each use-case to recover. Next, starting from each of these goals, we ask each representative to show the manipulations of the app they would perform get the expected result, with the main variants. This allows us to re-document the use-cases with the main flow and the alternative flows. In complex situations (like in the reverse engineering of large desktop application) we even video-recorded the interaction of the users with the application to make sure we remembered all the interactions. Then by slowly watching the video we can write down the sequence of action the user made to get the expected result from the app. Then we abstract the manipulations and write down the use-cases. The last step it to present the use cases to the users for validation. In particular we ask each user to manipulate the device according to the recovered use-case in order to get the expected result. Then we can immediately observe if the user feels comfortable with the scenario or if something is missing.

4 App Instrumentation

4.1 Introduction

Dynamic analysis as opposed to static analysis aims at observing the application's behavior while it is running. Although many techniques can be used [15] we decided to use code instrumentation because, on the mobile device, there are not many alternatives. Indeed one cannot install any profiling or debugging environment without deeply

impacting the behavior of the code. The least intrusive technique is simply to add lightweight tracing statements in the application source code to write the events in a flat file. Each of the recorded events must contain the signature of the method called. As for the class identifier we record the name of the class and, in case of the languages using module or package declarations, the package or module in which the class is defined. Once the trace file is generated (that could hundreds of thousands of events), it is loaded into a database for further processing. Many of the existing dynamic techniques focus on the monitoring of the low level instructions of the program, in particular when the purpose is to analyze an app for which only the compiled code is available. Since we wish to reverse engineer and understand the source code of the app, access to the source code is a must. The first step to build our own instrumentor is to be able to parse the source code. To build such a parser, several possibilities exist. Tools like JavaCC [21] YaCC [31] or ANTLR [1] are capable of generating a parser given the syntax definition of the programming language in the EBNF format. Such parser is completed by adding some extra parsing instructions in the target language. The main difference between these tools is the language in which the parser is generated. Our choice was JavaCC which generates a parser in Java. This is because JavaCC-encoded grammars are available for several programming languages, including Objective-C, and also because we had some previous successful experience with it. However we do not only need to parse the code, we also need to build an abstract syntax tree (AST) of the code in memory so that we could add the extra trace event generation code to some of the nodes in the AST. We used the Java Tree Builder [22] to produce the AST. Some Visitor [10] classes are generated by the same tool to visit each node of the AST. We use these "Visitor" classes to add the instrumentation instructions at the proper locations in the code. For the method start event we add the instrumentation instruction as the first statement of the method. For the method end event we enclose all the statements of the method in a try-finally construct and write the instrumentation instruction in the finally block. With this technique we can catch the method end whatever the way the method is ended. The output of the parser generation process is represented by two packages named "syntaxtree" and "visitor" which respectively contain the AST elements and their associated "visitors". Because every single abstract syntax tree element comes with its own "visitor" class, we focused on the ones responsible for the handling of methods. The added instructions in the source code must satisfy two conditions:

1. Do not produce any changes to the application semantics;
2. Limit as much as possible the impact on the application processing time.

The first constraint is self-evident. The second constraint aims at avoiding any impact on the scheduling of multi-threaded applications. To be able to record the events during the execution of the app, we need to build a little runtime program to write the events to a flat file. This is specific to each programming language. The instrumentation instructions we insert in the source code of the methods are simple calls to the functions of the run time. Because of the above constraint on the processing time, using a database to record the events is not an option. We must record them as fast as possible hence the choice of a flat file.

4.2 The Case of Objective-C

In the case of Objective C, the runtime program contains:

- A class with two methods to write an event at the entry and at the exit of the instrumented method.
- A class responsible for converting the primitive-typed values of the parameters into NSString, to write these values in the trace event (see the [Param] element of the trace event grammar).

Every iOS application has its own set of directories in which it can read and write files. An application's private file system is called a Sandbox [3] and it is specific to the application. Inside a sandbox, there are three predefined directories: Documents, Library and tmp. To store a trace file, the runtime program can write in either the Library or Documents directory. But we should avoid tmp, since its content may be cleared away by the system when the application stops running. Because these folders generally contain user-generated content and other resources used by the application's logic, we need to make sure the trace files we write will not interfere with the existing files. To do so, we create the trace files in a custom folder inside the Library folder:

```
<Application_Home> /Library/HEG_TRACE/trace_[timestamp].
```

This will not only ensure that our tool does not hamper the application's behavior but also allows the running of our use-cases in sequence to get several trace files all at once. Next, to upload the trace file into the desktop machine for further analysis we pull it out of the iPhone using iExplorer [18] which gives access to the part of the device's file system where the applications reside. A technique to shortcut the creation of the trace file could have been to embed a socket communication module in our runtime program to "pipe" all the data in real time to a listening socket. However this would require a permanent connection to server and this would not respect our second constraint to have as little an impact on the processing time as possible. Another alternative technique to trace file writing could have been to monitor the application execution using an embarked version of a debugger such as GDB [11]. Unlike C++ or Java, the runtime of Objective-C [24] uses a specific syntax to do message sending. A message sending is a statement like "[object1 foo:@ arg]" meaning that object1 is sent a message whose "selector" is foo: and whose argument is "arg". This syntax is converted to: "objc_msgSend(object1,foo(arg))" by the Objective-C runtime. Then, using the debugger, we would set a breakpoint on every "objc_msgSend" to monitor the execution. As the iOS devices use the ARM processor, fetching the right registers could give access to all the methods' execution context. But this technique would delay the program execution at each message sending and then would exaggeratedly slow down the whole application, therefore not respecting the second constraint. The chosen instrumentation technique using our own instrumentor has the extra advantage to be applicable to any programming language provided that a LALR-analyzable grammar is available. Hence the technique presented in this paper can be extended to the Android platform [25] since it uses Java as the programming language.

5 Case Studies

5.1 Data Access Application

We will first present the technique and tool we developed on a simple app that allows the user to search and display the acts and articles of the Swiss Law recorded in a database on the phone. With our reverse engineering technique we can quickly identify what classes are involved in the delivery of a given functionality and what are the dynamic caller-callee relationships for the use-case. As an example, here is the analysis of the classes involved in the use-case "Read a judgment of the Swiss Federal Court". The execution trace is rather short ($\sim$2000 calls) and the number of classes involved is small (6 classes). But this app is well suited to show the power of the analysis we may perform with our tools. Once we loaded the execution trace in our analysis tool, we can display the main features of the execution.

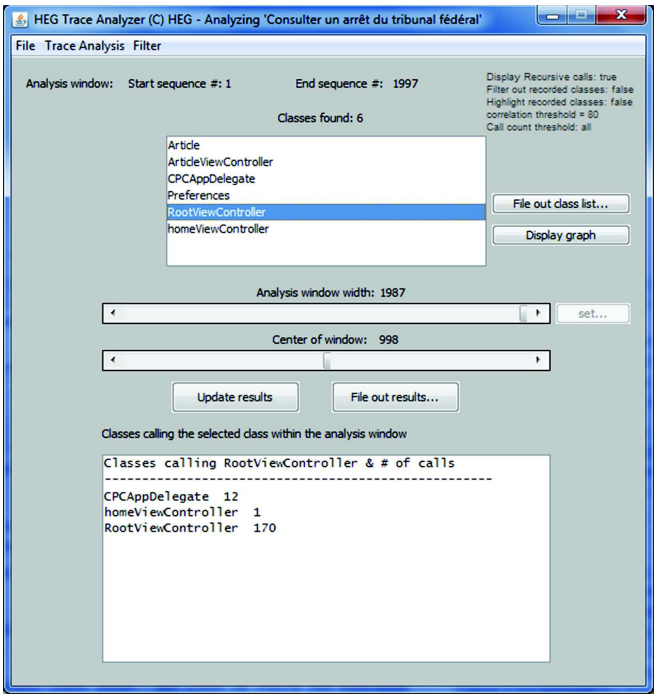

Fig. 3. Trace analyzer.

For example, in Fig. 3 the trace analyzer tool displays the classes involved in the use-case and specifically what class calls what other classes.

As we can see in the display, the class RootViewController is called by 3 other classes:

- CPCAppDelegate 12 times
- homeViewController only once
- RootViewController 170 times.

Figure 4 displays the call graph with all the involved classes. In this figure we can see that four classes are coupled bi-directionally which, on the point of view of code quality, could be something to investigate further. But this is neither the case of the ArticleViewController nor the Preferences classes. The call graph is generated by our tool using the Graphviz open source library [13].

Now we are interested to know when, in the course of the execution, the classes are involved. Then our trace analysis tool could display a "time series" graph of the classes' execution in the trace. But the problem is that the trace could be quite huge. Then the display of each and every method in the trace would lead to a very dense graph. To overcome the problem we introduced a little bit of statistical processing: we segment the trace in contiguous segments of a predefined size and, for each segment, we count the number of times a given class is called. Figure 5 presents such a time series graph for the Preference class.

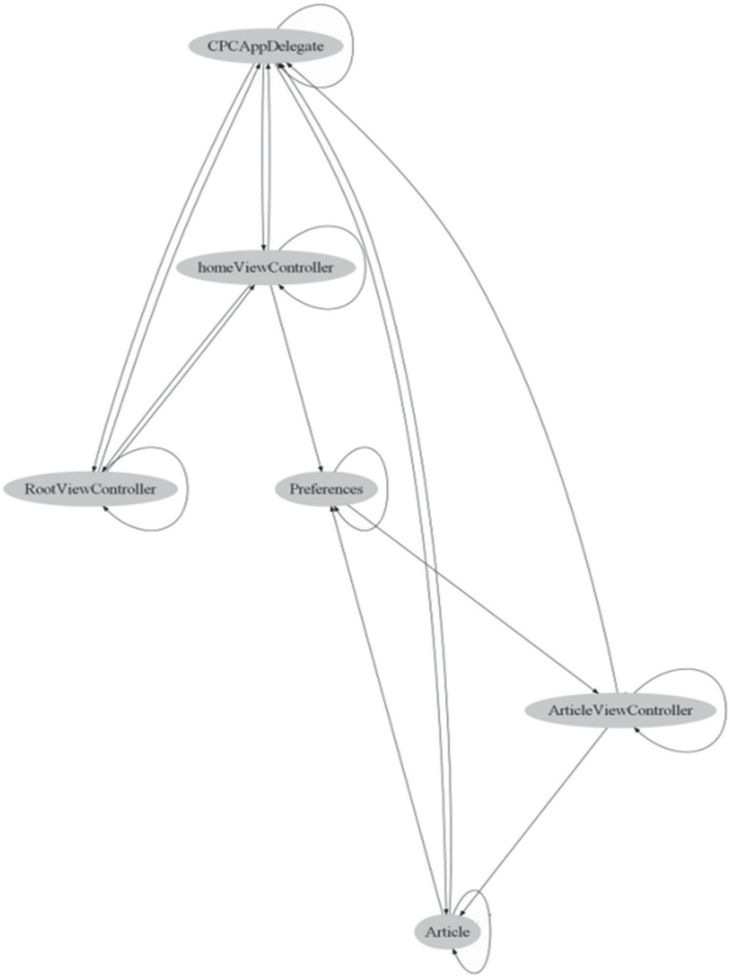

Fig. 4. Caller-callee graph.

On the top of the window we see the segment size (35 calls in this example). The horizontal axis shows the location of each of the segment. To make the time series comparable between use-cases, we normalize the horizontal axis to 100. The position of a class execution is therefore relative to the trace size.

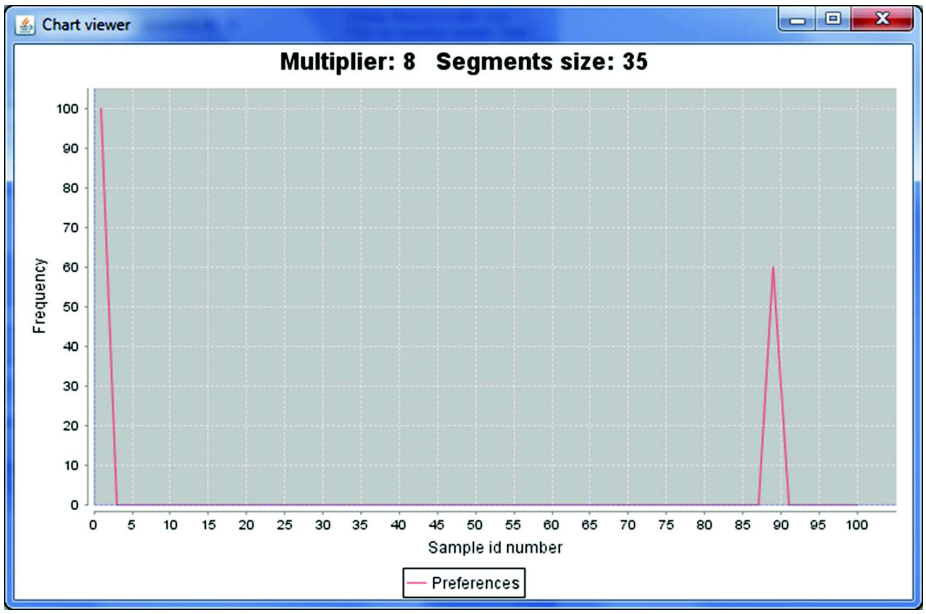

Fig. 5. Preference class time series.

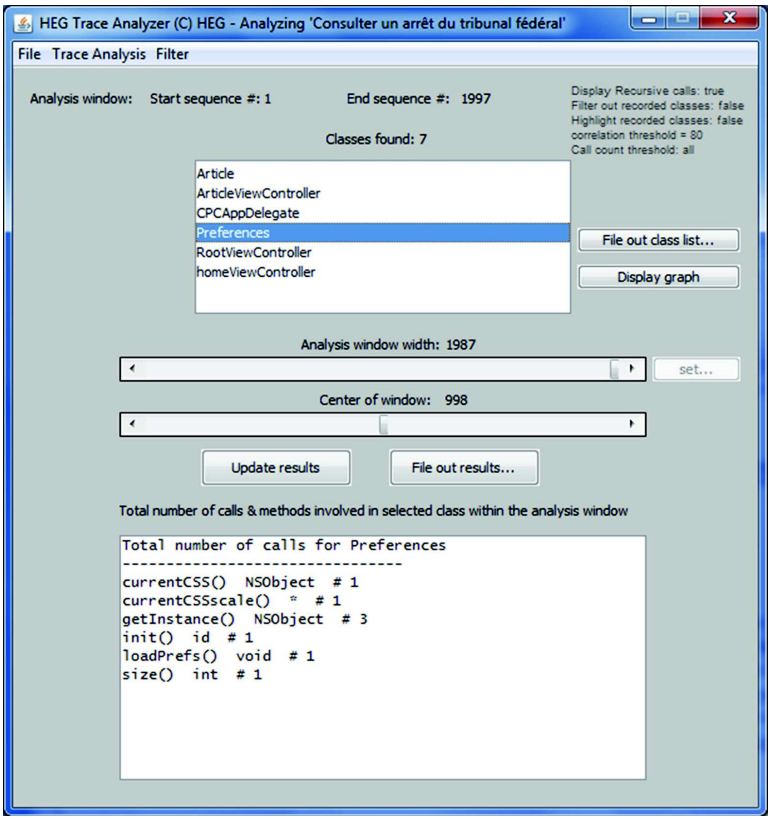

Fig. 6. Methods called in preference class.

As we can see, the Preference class is executed at the beginning of the processing and close to the end. Figure 6 displays the methods that are called in the Preferences class. We observe that very few calls are made in this class. Indeed this class holds the

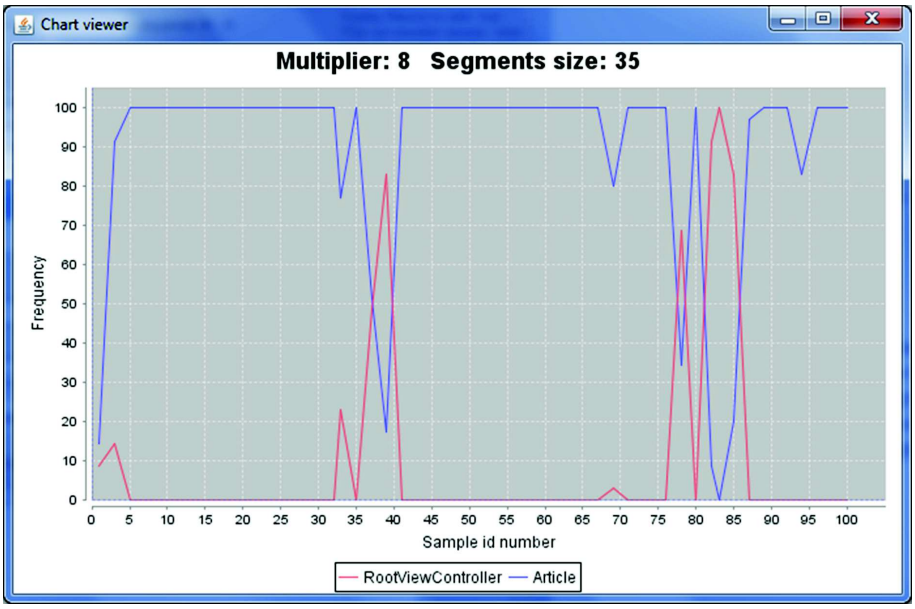

Fig. 7. Joint time series for 2 classes.

application's preferences parameters. All the behavior, showed by Figs. 5 and 6, rightfully represents what we could expect from a class which holds preferences information. Next, we could compare the time series of two classes. Figure 7 shows the joint time series for the classes RootViewController and Article.

Interestingly, the involvement of these two classes seems opposite. In the few segment where the Article class is much less involved then the RootViewController class is heavily involved. A further source code investigation revealed that the hundreds of Article instances (i.e. articles of the law) to be loaded in memory from a file are loaded all at once. Because this process is not in a dedicated thread, it blocks everything else until it is finished. The RootViewController contains a UITableView and implements its delegate and datasource protocols [4]. Because the structure of the law acts and articles is hierarchical, a RootViewController is reclusively created every time the user browses a subcategory of the law acts and articles. Then the relevant Article objects are accessed in memory, inserted into the UITableView cells and the RootViewController is quit. This explains the sudden "bursts" of activity of the RootViewController following the activity on Article objects. With this information we can now reconstruct the UML class diagram corresponding to the executed use case (Fig. 8). This diagram represents the implementation classes of the functional structure of the system in relation to the use-case. It contains the classes, methods and dynamic associations involved in the execution of the use-case. In some sense this represents a "projection" of the use-case to the whole system (in the relational database sense). Today, this UML class diagram is built by hand from the output of the tool. We intend however to integrate our tool with the software modeling environment we use (IBM's Rational Software Architect) so that this class diagram could be created automatically.

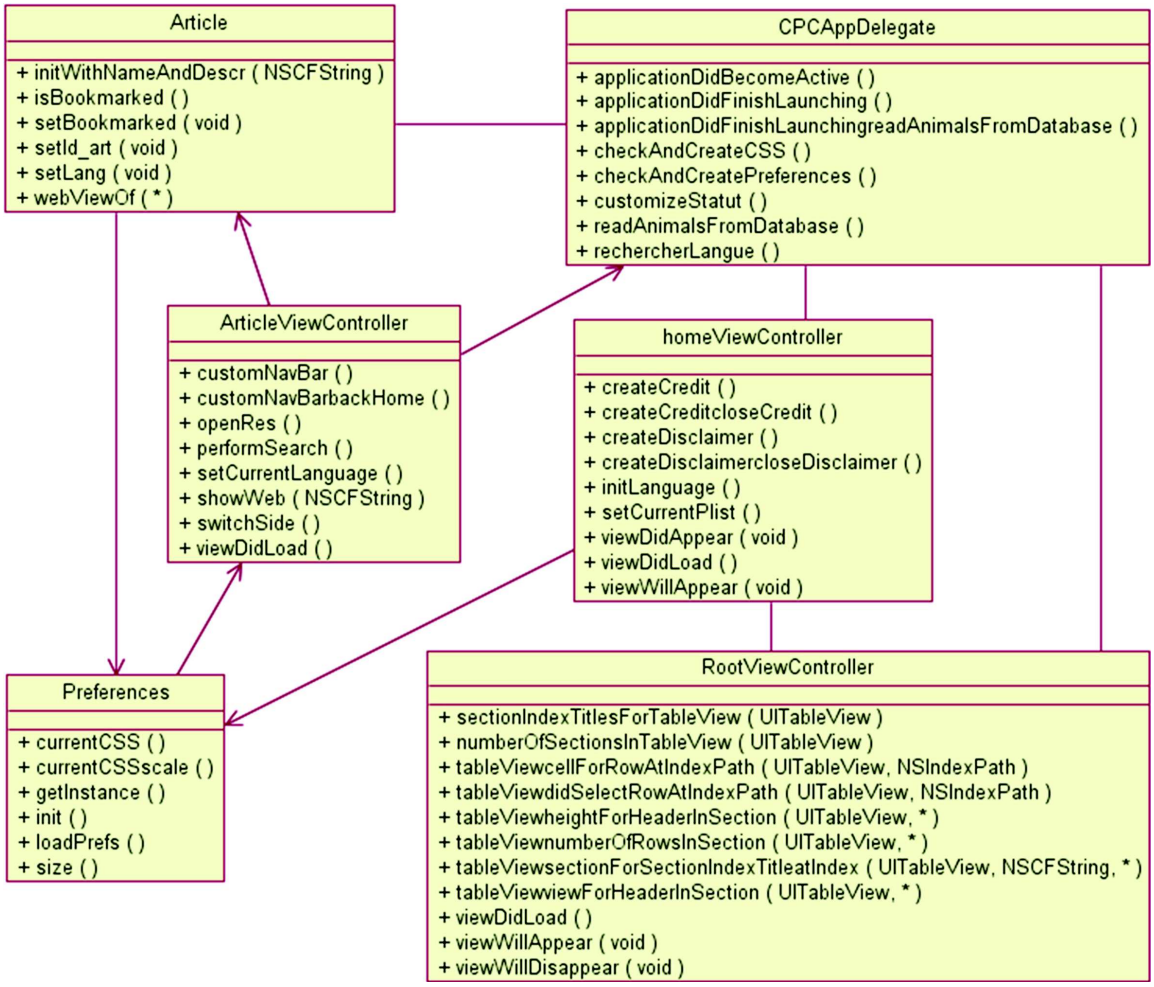

Fig. 8. Class diagram of the functional structure.

5.2 Word Press

As a second example, we ran the WordPress framework to create a blog from the iPhone. This open source framework was instrumented, compiled and then shipped to the iPhone. Then we used the app to create a blog ("create blog" use case). The execution trace is about 1 Mb long and has 4252 calls which involve 53 classes. First we can analyze the diversity of methods called per class (the number of different method called in the classes). While more than half of the classes (30) have 5 of less distinct methods called, a single class, ReaderPostsViewController, has 55 distinct methods called and CreateAccountAndBlogViewController has 29. Table 1 shows the 15 classes that get 10 or more distinct method executed.

Next we can analyze the coupling between the classes: what are the classes called by each of the classes. A class that calls many classes may play the role of a "controller" class i.e. a class that organizes the work of other classes (following the MVC

Table 1. Number of distinct method executed per class, for classes with 10 methods or more.

Class	#methods
ReaderPostsViewController	55
CreateAccountAndBlogViewController	29
WordPressAppDelegate	24
ReaderPostView	21
ReaderPostService	19
LoginViewController	18
NotificationsViewController	16
ReaderPostServiceRemote	15
WPStyleGuide	13
WPAvatarSource	12
WPAnalyticsTrackerMixpanel	11
WPNUXUtility	11
ReaderPostTableViewCell	10
WPTableImageSource	10
WPWalkthroughTextField	10

Table 2. Classes calling other classes.

Class name	# class called
ReaderPostsViewController	17
WordPressAppDelegate	17
LoginViewController	16
WPImageSource	15
CreateAccountAndBlogViewController	12
ReaderPostView	10

style). A close look at the classes that call the most of other classes show that their name include the term “controller” in 50 % of the cases (Table 2).

If one displays the location where these classes are involved one finds that ReaderPostsViewController is involved in almost every part of the execution trace while WordPressAppDelegate is much more localized (Fig. 9) although the number of classes they call is similar. ReaderPostsViewController seems to be a key class in the implementation of the use-case.

On Fig. 10 we compare ReaderPostsViewController and LoginViewController whose number of class it calls is close (16). The execution of LoginViewController is somewhat more localized than ReaderPostsViewController and seems to play less of a central role.

The sequence of screens that we get when the use-case is executed is presented in

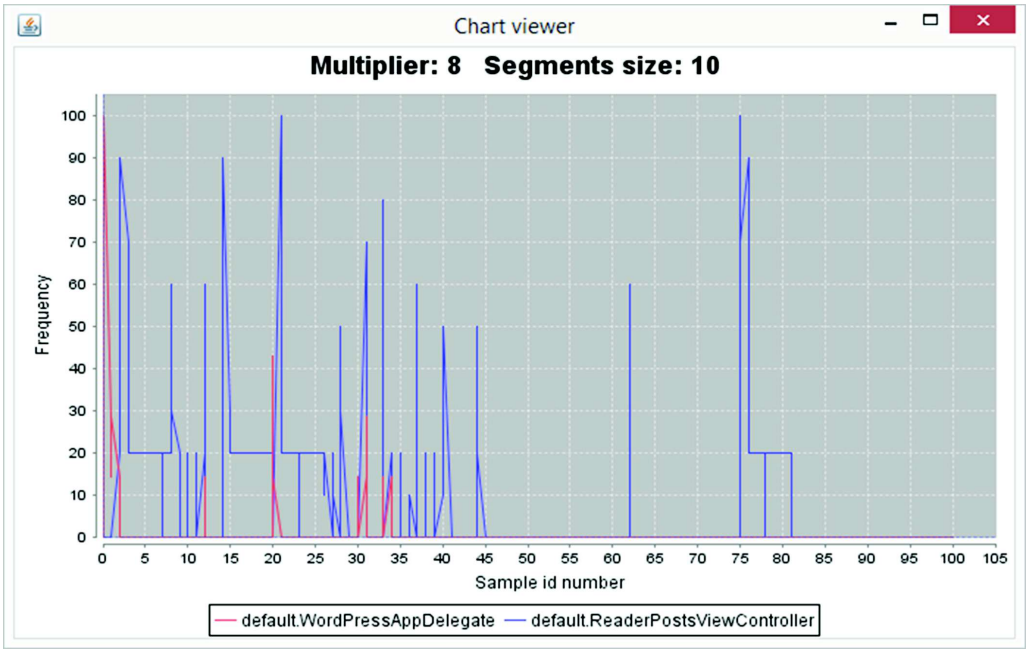

Fig. 9. Time series comparison of WordPressAppDelegate and ReaderPostViewController.

Fig. 11 (as recorded on September 20st, 2015). The screen on the left is displayed for a short period of time (1 s) after wich the second screen is displayed. Once the information have been entered into this screen and the account created the third screen opens to sign in.

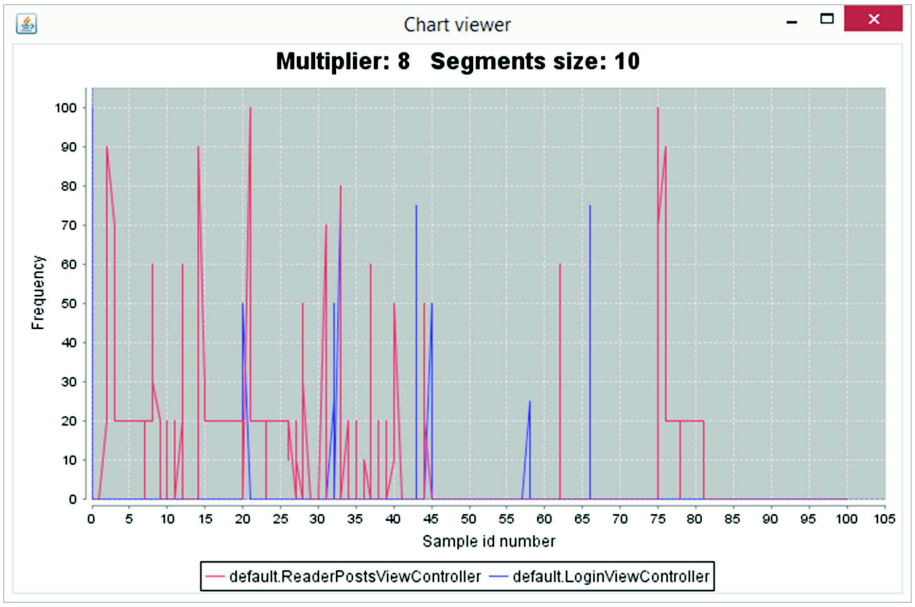

Fig. 10. Time series comparison of LoginViewController and ReaderPostViewController.

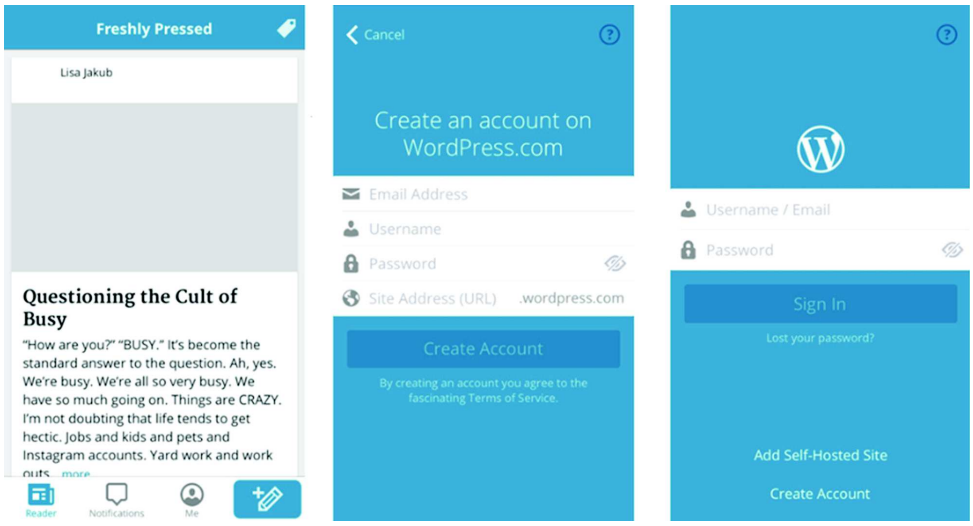

Fig. 11. Sequence of screen in the create blog use-case.

Since we suspect the classes whose name start with "UI" to be involved in the implementation of the displays we draw their time series in the same graph. These may represent screens or parts of screens. This is presented on Fig. 12.

We see that UIDevice, which represents the current device and which is used to retrieve the device parameters, is called at the beginning of the use-case as we could have guessed. On the other hand UIColor is used to manage the colors and could be involved in every screen. There is not much information to infer from the location of the execution of this class in the time series except that each time it is executed there must be some screen involved. UITableView is a class used to display a list of items as a single column that is vertically scrollable. But the first screen in Fig. 11 has exactly this kind of behavior. Therefore we can identify where in the execution trace this screen is located. It is at coordinates 13 in the time series. Finally, UILabel represents a

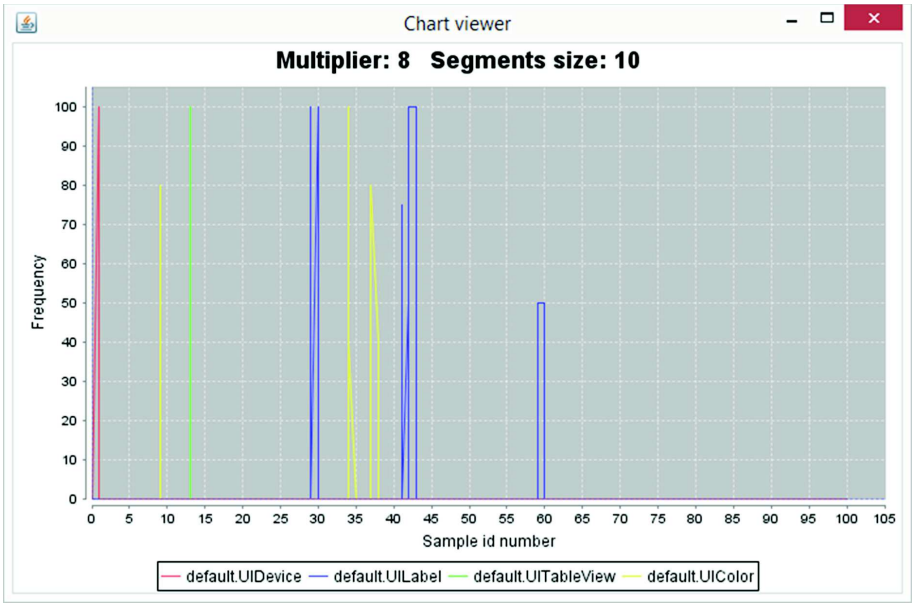

Fig. 12. Execution of UI classes.

read-only text view. Since the last two screen are full of these, the time series let us observe where in the trace these screens might be displayed.

Now we could observe the execution of the UILabel, UITableView and LoginViewController classes in the trace (Fig. 13).

We remark that LoginViewController is located close to the other classes which could mean, as its name suggests, that it performs some access control once the view is displayed. Of course all these are only educated guesses. But they gives us a good starting hypotheses from which we could explore the source code.

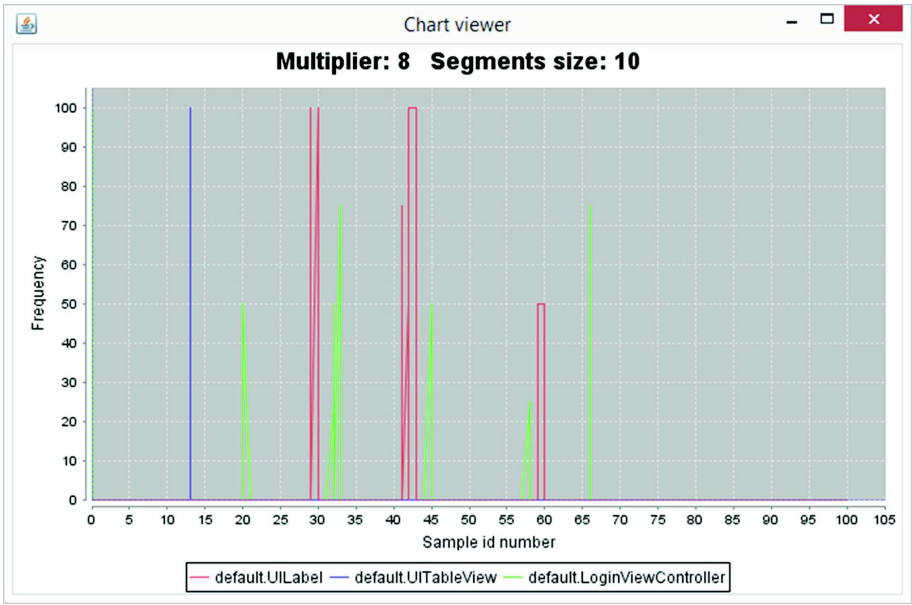

Fig. 13. UI classes and login view controller.

5.3 Interpretation Process

The above analysis shows that some simple statistical comparison among classes as well as the timing of their executions may help us formulate hypotheses when trying to understand some unknown code. Here are the steps of our interpretation process. From each execution trace we identify:

(a) The list of classes involved
(b) The methods called in each of the classes
(c) The classes called by each class (i.e. the classes implementing the methods called by some method in the other classes)
(d) The classes calling each class (i.e. the classes implementing the methods calling methods in the other classes)
(e) The places in the execution trace where the methods of each classes are executed, displayed as a time series.

From (a), (b), (c) and (d) we can build the functional structure of the system for each use-case (see Fig. 8 as an example). This gives us an overview of the classes, their methods and their associations required to implement each use case. But such a structure

may not be complete since we usually do not recover all the alternative flows of a use-case. From (c) and (d) we can see if some class interacts with a lot of other classes. This would suggest the role of a controller in the MVC paradigm. The time series of the executions let us find classes that work closely together. Indeed if the execution of some classes is repeatedly adjacent in the time series, forming a pattern of executions, we can make the hypothesis that these classes together implement some function or subfunction. Of course, this is not directly observable in the source code since it is difficult to infer when a given class will call a method in another class. In contrast, the time series let us observe when the classes are called in the context of a business relevant use-case. For example, we leveraged this dynamic technique to retrieve the functional components of a system using a clustering algorithm [7]. From (c) and (e) we could infer the scope of a controller class. If the number of class called in high but the activity of the class is spread all over the time series, then it is probably a global controller class, whereas if the activity of this class is localized it is more like a subfunction controller.

6 Related Work

Dynamic analysis of apps has been a subject of interest for a few years. For example, it has been used to check the security of the app when its source code is unavailable and specifically to do black-box penetration testing. However, when the source code of the app is available, the tester generally turns to static code review and white box testing. Gianchandani [12] uses snoop-it [27] to hook into a chosen application's process and to monitor network and file system activities. He also uses Introspy [19] which is composed of a tracer module and an analyzer module. After having selected the API to trace, the tracer will log the corresponding calls to a database.

Next, the analyzer will produce a human readable report in HTML. However the tool does not target all the custom application classes but focuses on the specific ones related, but not limited to cryptography, data storage and networking. Szydlowski et al. [28] proposed a technique to perform automatic dynamic analysis of iOS applications by hooking to the application's delegate and triggering all of the UI controls on every view. The result is a state model of the application. However, most of the dynamic analysis methods operate on the low level instructions. Hence, hooking to the running process is needed. But Apple does not include any default debugger on the device and installing one requires to jailbreak the iPhone. An alternative consists of running the application on the iOS Simulator [20] that comes with XCode then monitoring its process using GDB [11] or LLDB [23]. But the dynamic analysis of a simulated application using a debugger does not provide as much information as is available when writing the trace events to a file and analyzing the file off-line. Indeed the latter method let us perform statistical analysis which is difficult when using a debugger. Moreover, working on a simulated device, the technique does not allow analyzing apps that involve sensors such as accelerometer, compass or camera as they cannot be reproduced in the iOS Simulator.

7 Conclusion

The contribution of this paper is to present a reverse-engineering process and the associated tools to reverse-engineer and mobile applications. Indeed the technique is not limited to iPhone apps since the only specific part is the generation of the trace file. Hence the technique is applicable to whatever environment, provided that we can build a source code instrumentor for the associated programming language to generate an application trace in a standard format. In particular, since we already developed an instrumentor for Java, we are ready to analyze any Android application. The trace analyzer we developed provides a rich set of views through which the maintenance engineer can study the running of the code. In our case studies, we observed that the "time series" technique can visually present the mutual behavior of the classes in a convenient format. It provides some useful clues as to how classes interact when running the use-cases. The UML class diagram of the functional structure of the use-case summarizes all the programming elements involved in the execution of some use-case. It is important to highlight that our technique is based first and foremost on business relevant use-cases, i.e. whole scenarios that make sense for the user of the app, and not on the execution of some arbitrary function. This allows us to make hypothesis about the business semantics of the observed patterns of classes' execution. But the difficulty of the technique comes from the very same reason. Since we are not sure to recover all the relevant alternative paths in each of the scenarios, because the use-cases are drawn from the observation of the user, the analysis will be incomplete. To overcome the problem, in the case of legacy desktop applications, we investigated a semi-automated technique to recover the use cases directly from the legacy code [9] with moderate success however, due to the complexity of the task. Indeed, use-case recovery from source code is still an open problem. As future work we will integrate our tool with IBM's RSA to be able to generate the UML class diagram of the functional structure automatically. We also intend to develop new views to directly represent the dynamic business-level application semantics. Indeed we are building domain concept ontologies whose concepts will be dynamically identified in the executed code. This technique will help us to close the semantic gap between the high level business domain concepts and the code level.

References

1. ANTLR: ANother Tool for Language Recognition (2014). http://www.antlr.org/. Accessed 12 October 2014
2. Appcelerator/IDC: Mobile Developer report (2013). www.appcelerator.com.s3.amazonaws.com/pdf/developer-survey-Q2-2013.pdf. Accessed 5 March 2015
3. Apple iOS: File System Programming Guide (2014). https://developer.apple.com/library/mac/documentation/FileManagement/Conceptual/FileSystemProgrammingGuide/FileSystemOverview/FileSystemOverview.html. Accessed 12 October 2014
4. Apple UITableView: UITableView Class Reference (2014). https://developer.apple.com/library/ios/documentation/UIKit/Reference/UITableView_Class/. Accessed 12 October 2014

5. Bachmann, F.: Documenting Software Architectures: Views and Beyond, 2nd edn. Addison-Wesley, Reading (2010)
6. Dugerdil, P.: Using trace sampling techniques to identify dynamic clusters of classes. IBM CAS Software and Systems Engineering Symposium (CASCON) October 2007
7. Dugerdil, P., Jossi, S.: Computing dynamic clusters. In: 2nd Indian Conference on Software Engineering (ISEC) 2009. ACM Digital Lib (2009)
8. Dugerdil, P., Niculescu, M.: Visualizing software structure understandability. In: 23rd Australasian Software Engineering Conference (ASWEC) 2014. IEEE Digital Library, Sydney (2014)
9. Dugerdil, P., Sennhauser, D.: Dynamic decision tree for legacy use-case recovery. In: 28th ACM Symposium On Applied Computing (SAC 2013), Coimbra, Portugal, 18–22 March 2013
10. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns Elements of Reusable Object Oriented Software. Addison-Wesley, Reading (1995)
11. GDB: GNU Debugger (2014). http://www.gnu.org/software/gdb/ Accessed 12 October 2014
12. Gianchandani, P.: Damn Vulnerable iOS Application (DVIA) (2014). http://damnvulnerableiosapp.com/#learn. Accessed 12 October 2014
13. Graphviz (2015). http://www.graphviz.org/Home.php. Accessed 17 April 2015
14. Hammond, J.S.: Development Landscape: 2013, Forrester Research (2013)
15. Hamou-Lhadj, A., Lethbridge, T.C.: A Survey of Trace Exploration Tools and Techniques. In: Proceedings of the IBM Conference of the Centre for Advanced Studies on Collaborative Research (2004)
16. IBM: IBM Mobile First initiative (2014). www.03.ibm.com/press/us/en/presskit/39172.wss. Accessed 12 October 2014
17. IDC: IDC Predictions 2013 Competing on the 3rd Platform (2013). www.idc.com/getdoc.jsp?containerId=WC20121129. Accessed 5 March 2015
18. iExplorer (2014). http://www.macroplant.com/iexplorer/. Accessed 12 October 2014
19. Introspy-iOS (2014). https://github.com/iSECPartners/Introspy-iOS. Accessed 12 October 2014
20. iOS Simulator (2014). https://developer.apple.com/library/ios/documentation/IDEs/Conceptal/iOS_Simulator_Guide/GettingStartedwithiOSStimulator/GettingStartedwithiOSStimulator.html. Accessed 12 October 2014
21. JavaCC: Java Compiler Compiler – The Java Parser Generator (2014). https://www.javacc.java.net/. Accessed 12 October 2014
22. JTB: Java TreeBuilder (2014). http://compilers.cs.ucla.edu/jtb/. Accessed 12 October 2014
23. LLDB: LLDB Debugger (2014). http://lldb.llvm.org/. Accessed 12 October 2014
24. Objective, C: Runtime Reference (2014). https://developer.apple.com/library/mac/documentation/Cocoa/Reference/ObjCRuntimeRef/Reference/reference.html. Accessed 12 October 2014
25. Parada, A.G., de Brisolara, L.B.: A model driven approach for android applications development. In: Proceedings of the Brazilian Symposium on Computing System Engineering (SBESC) (2012)
26. Sako, R.: Reverse engineering d'une application mobile Apple. Bachelor Thesis (2011)
27. Snoop-it (2014). https://code.google.com/p/snoop-it/. Accessed 12 October 2014
28. Szydlowski, M., Egele, M., Kruegel, C., Vigna, G.: Challenges for dynamic analysis of iOS applications. In: Camenisch, J., Kesdogan, D. (eds.) iNetSec 2011. LNCS, vol. 7039, pp. 65–77. Springer, Heidelberg (2012)
29. Tilley, S.R., Santanu, P., Smith, D.B.: Toward a framework for program understanding. In: Proceedings of the IEEE International Workshop on Program Comprehension (1996)

30. Wasserman, A.I.: Software engineering issues for mobile application development. In: Proceedings of the 2nd Workshop on Software Engineering for Mobile Application Development MobiCase 2011 (2011)
31. YaCC: Yet Another Compiler-Compiler (2014). http://dinosaur.compilertools.net/yacc/. Accessed 12 October 2014