Huy Quang Le

iOS Application Testing with the Social Application Buddify

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor Thesis

7/11/2016

Author(s) Title Number of Pages Date	Huy Quang Le iOS Application Testing with the Social Application Buddify 40 pages 7 November 2016
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor(s)	Kari Salo, Principal Lecturer

Testing is a critical and inevitable part of any application development process. This project aims to provide testing methodologies and testing implementation on an iPhone operating system (iOS) application. As a result, an iOS application called Buddify was developed by Real Application Company in Singapore aiming to create a social network for people to interact with others. Testing materials and a testing implementation process on the Buddify application was included in the project. The project used XCTest, Crashlytics, and the UserVoice framework to debug and test the application. Different testing methods were also involved in the testing process. Information from Apple developer documentation, eBooks, and online material were collected and applied in the project.

After the final testing phase, the application was submitted and approved to be in the Apple App Store for global users. The Buddify application is fully functional, responsive and smooth with no bugs and crashes. Consequently, this thesis provides a guideline for anyone who wants to start doing iOS application testing.

Keywords	iOS, Mobile Application, Testing, Crashlytics, XCTest

Contents

List of Abbreviations:	
iOS	iPhone Operating System: A mobile operating system created and developed by Apple Inc.
App	Application
Buddify	Name of the application used in the project
SDLC	Software Development Life Cycle
UI	User Interface
IDE	Integrated Development Environment
Mac	Macintosh Computer which is manufactured by Apple Inc.
OS	Operating System
OS X	Mac OS X is the current series of Unix-based graphical operating system developed by Apple Inc.
Instruments	A testing tool integrated in Xcode
CPU	Central Processing Unit
API	Application Programing Interface
SDK	Software Development Kit

1 Introduction

Software is usually designed and developed by humans, so mistakes are something inevitable during the development process. Therefore, testing is a critical part to ensure software is functional and efficient. Recently, the mobile application (app) market has been growing fast because people are using smart phones more and more. Additionally, mobile apps are extremely convenient and they can be used to do many things on the go from reading news to chatting, checking and replying emails or just playing games. The rise of mobile apps demands a diligent and precise testing process.

The purpose of this project was to study testing methods and their implementation in the iOS mobile application called Buddify. Buddify is a social network iOS application which allows users to look for people based on geographic location and country tag. The goal of this project was to test the native social network iOS application Buddify in order to improve the quality of the application.

The project aimed to test a native iOS mobile application. Also, the project applied different frameworks like XCTest, Crashlytics, and UserVoice for testing purposes. Testing was carried along the development phase and before releasing the app in the Apple App Store.

2 Importance and Benefits of Mobile Application Testing

2.1 Part of Software Development Life Cycle (SDLC)

There are many software development process models for developers to embrace for developing applications. The software development models which are the most popular are the waterfall model, the V model, the incremental model, the RAD model, the Agile model, the iterative model, and the spiral model [1]. Despite which model is applied, a software development life cycle basically should contain these phases: requirement analysis, design, implementation, testing, and maintenance (see figure 1).

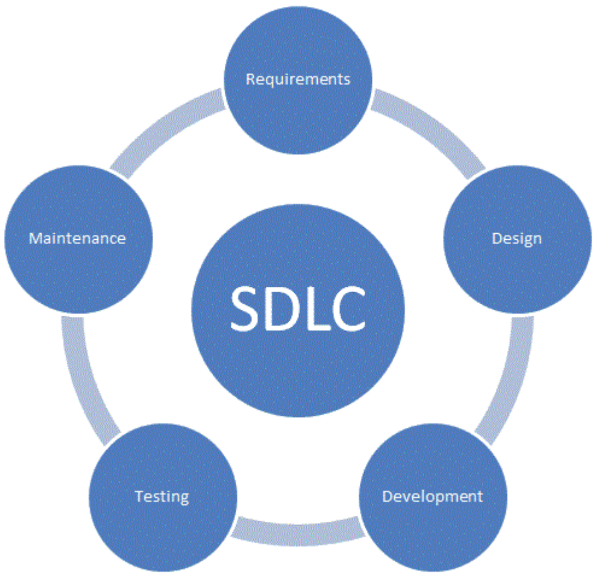


Figure 1 Software development life cycle. Reprinted from Handschuh (2011) [2].

The testing process is carried out after software development to detect bugs and potential defects of the program in question. However, a testing plan should be done at the beginning of the development phase to have specific objectives and major activities in the testing phase. During the testing, the list of bugs and defects found should be sent back to the developer team for reviewing and fixing. When a new update is ready, developers send it back to testing until the product passes all of the tests; then the product is ready for deployment. [2.]

As mobile devices, particularly smart phones, are getting popular, the mobile app market is growing rapidly. The app market shown in figure 2 points out that Google Play is currently leading the market with 2.2 million apps, followed by the Apple App Store with 2 million apps (see figure 3) [3; 4]. Obviously, the need of testing of mobile apps has increased tremendously within the past eight years.

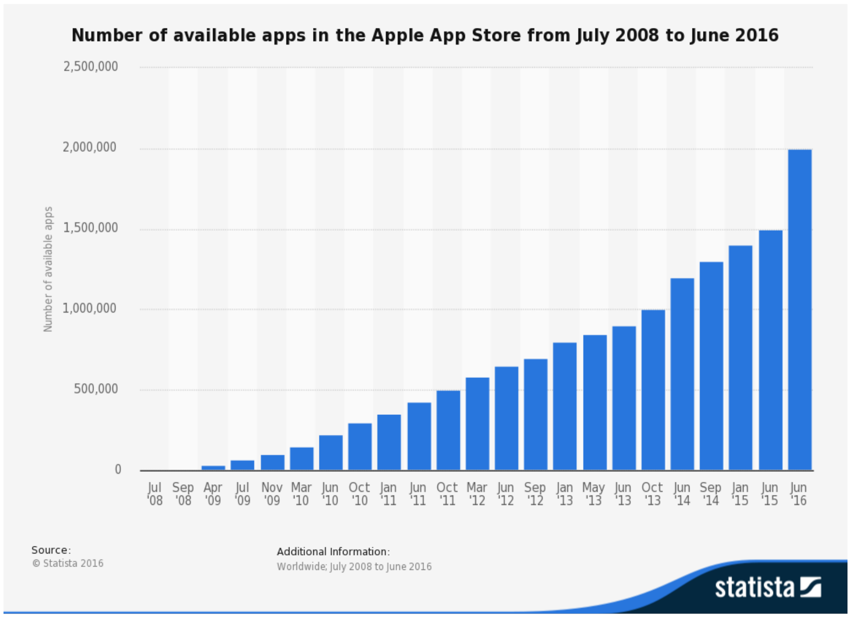


Figure 2 The amount of iOS apps in the Apple App Store 2008-2016. Reprinted from Statista (2016) [3].

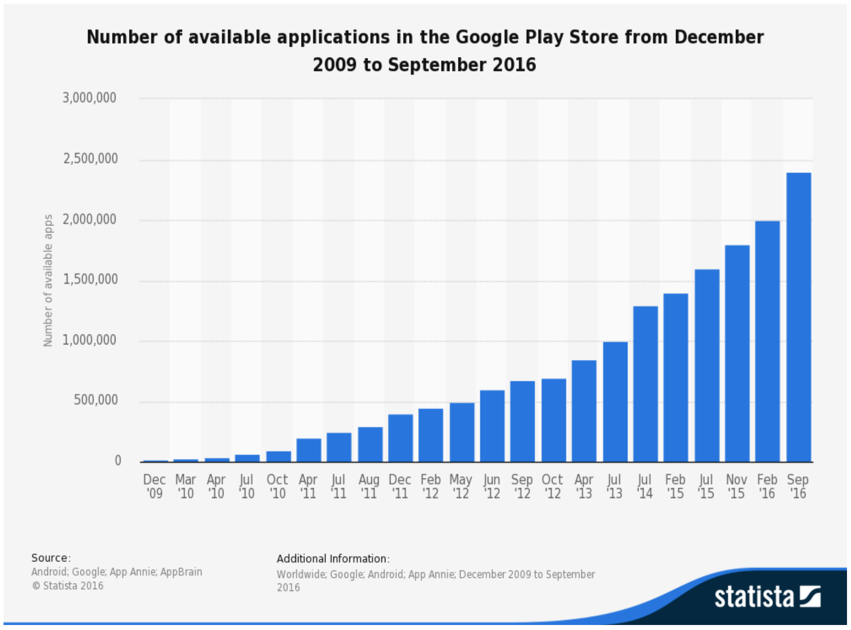


Figure 3 The amount of Android apps in the Google Play Store from 2009-2016. Reprinted from Statista (2016) [4].

2.2 Benefits of Mobile App Testing

Here are some benefits of software testing in the whole application development process:

- Testing is part of SDLC to find out bugs and defects that were made during the development process. A good testing plan at the beginning of a project implementation will make sure that testing will be done effectively and productively.
- Testing helps saving time and resources.
- Testing makes sure that users will be satisfied and excited to enjoy apps. An error-free user interface (UI) and functional app is key to making users keep the application.
- Testing guarantees the performance of the application with performance testing. No user wants to experience a slow running and lagging app. Keeping the app responsive and smooth is the testers' responsibility to motivate the user to be active in the app.

  [5.]

3 Software Testing Fundamentals

3.1 Definition of Testing

Utting and Legeard give the following general definition to testing:

> *Testing is an activity performed for evaluating product quality, and for improving it, by identifying defects and problems [6,3].*

Indeed, testing is a process of evaluating a product's quality to see if it meets the customer requirements or just the manufacturer's expectations. Finding and fixing problems can improve product quality, which increases customers' trust in the product. To be more specific, Myers et al. define software testing in the following way:

> *Testing is the process of executing a program with the intent of finding errors [7,6].*

Testing is definitely considered a process rather than a single activity. The process should contain planning, analysis, implementation and reporting. The purpose of testing is finding errors. Software is designed and developed by a programmer. Software errors are something that can happen during programming. Applications can also have bugs in their algorithm that could lead to unwanted outputs or slow running functions. Therefore, testing aims to find the errors and defects unwanted in the software development. Testers can also give feedback about the UI if there is a readability issue or inconsistent design or UI element arrangement. Software testing is the final step for a software program to be released; therefore, testing should be performed with many aspects of the program to meet customer or user expectations. When users start to use the product, there should not be any problems or complaints. Good testing coverage should cover functionality of the application, user interface, performance testing, compatibility of the application to specific OS or hardware configuration, loading testing. There are two main testing strategies that would cover these testing methods: black box testing and white box testing.

3.2 Testing Strategies

3.2.1 Black Box Testing

Black box testing, also known as data-driven or input/output-driven testing, is testing the application behavior according to its requirement. A tester will view a program as a box with input and output value. For this testing strategy, testers do not need to know how the system is structured or how functions are programmed. They just need to focus on how the program runs and what to expect from the outcome. [7,8-9.]

Black box testing usually refers to user testing because users are those who will install and execute the program. Users do not have much knowledge of the program but what they can see is application functionalities, ease of use, data validation, and a well-designed user interface. The testing methods used in the final year project, which follow the black box testing strategy, are exploratory testing and user testing. In the project, exploratory testing is performed by testers, which have in-depth knowledge of programming and debugging skills while user testing is mostly done by normal users.

3.2.2 White Box Testing

In contrast to black box testing, white box testing (also known as logic driven testing) allows testers to see and investigate the internal structure of the program. To understand how the software has been designed and developed, testers need to have knowledge of programming and structure of the program. Besides, testers need to access the source code for the testing. [7,10.]

White box testing generally includes control flow, code efficiency and flow of program as designed. In order to perform white box testing, testers need to understand the source code of the system, create test cases and execute them. White box testing is one of the best methods to detect errors in the software application in initial stage of software development. [8.]

The testing methods used in the final year project, which follow the white box testing strategy, are automated UI testing and performance testing. They are performed using the Instruments tool in Xcode, the XCTest framework and the Crashlytics framework.

4 Testing Methodologies

Automated testing and manual testing cover two huge categories of testing. A variety of methods within each category can be applied such as UI testing, performance testing and user testing. Some methods should be carried manually and some are better done with automation.

4.1 Automated Testing

Automated testing is an automated test process carried out by a computer. Since manual testing is time consuming and less efficient, automated testing is performed as a more accurate and convenient method.

4.1.1 Automated UI Testing

UI testing is a process to detect the app's bugs and errors using the UI. Automated UI testing will include UI recording, which means generating code that manipulates the app exactly the same way as users do. Furthermore, the code can be expanded to produce other UI test cases.

UI testing concept with XCTest:

- XCUIApplication: XCUIApplication is a proxy to interact with the app and it is the root of the app.
- XCUIElement: This XCUIElement represents the actual UI elements in the app. Elements nest in a tree in which the root is XCUIApplication. The XCUIElement is handled with type as an enumeration of XCUIElementType. They can be, for example, buttons, check boxes, tables, or pickers.
- XCUIElementQuery: The query class is used to locate the exact UI element to interact with. XCUIElementQuery has a matching property for accessing exactly one UI element. If there are no matches or multiple matches, the test will fail. These references can be kept and reused in a later test.
- Simulated events: Interaction with a specific UI element is triggered with functions like tap(), doubleTap(), press(forDuration:) and typeText(text:).

- Assertions: Assertions are written by testers using functions like XCAssert(), XCAssertEqual() and XCAssertNotNil() to validate UI properties and states.
  [9.]

The process of UI recording with XCTest follows the steps as documented in The Apple Developer Documentation:

1. Using the test navigator, create an UI testing target.
2. In the template file that is created, place the cursor into the test function.
3. Start UI recording. The app launches and runs. Exercise the app doing a sequence of UI actions. Xcode captures the actions into source in the body of the function.
4. When done with the actions you want to test, stop UI recording.
5. Add XCTest assertions to the source.
   [10.]

4.1.2 Performance Testing

Performance is a key factor for any software program to determine the quality of the app. Performance testing is testing to see how efficient the system runs under particular workload. The Instruments tool integrated in Xcode will allow performing this test. Generally, it can be used in the following test scenarios:

- Evaluating the app performance in specific work load
- Verifying central processing unit (CPU) use over time period
- Validating app performance with intermittent phases of connectivity
- Evaluating network usage of the app
- Evaluating memory usage of the app
  [11.]

The benefits of performance testing are as follows**:**

- Providing a great user experience to users
- Guaranteeing the app is energy-efficient
- Making the app responsive to users' interaction
  [12.]

4.1.3 Testing Using Third Party Frameworks

There are many different third party frameworks that allow a tester to use automation testing in an iOS application. The most common testing frameworks are KIF, FBSnapshotTestCase, Frank and Appium. These frameworks need to be integrated into the Xcode project in order to use test automation. However, it would be challenging to learn a new framework and probably a new language for a cross platform support. In this final year project, automated testing was performed by a native framework called XCTest. Using XCTest does not require developers or testers to learn much. Besides, it also supports native iOS languages. In addition to XCTest, the Crashlytics framework was also used to collect crash information on user devices and bugs related to the crash in order to optimize the app performance.

4.2 Manual Testing

4.2.1 Exploratory Testing

Exploratory testing is testing without a formal testing plan. Testing can be done using a real device or a simulator. Exploratory testing is considered as a black box testing technique. Based on use cases of the app, testers will define test cases.

The benefits of exploratory testing are:

- With less preparation needed, it can detect serious problems quickly.
- A testing plan is not required, so testers are given more time on bug finding.
- Most bugs from both front-end and back-end service are found in this kind of testing.
  [13.]

The drawbacks of exploratory testing are:

- Testers need to have good knowledge of testing.
- There is no guarantee to fulfil testing requirements.
- Lack of testing documentation.
  [13.]

4.2.2 User Testing

User testing is another kind of manual testing as users will test the app themselves and give feedback to testers and developers. Two types of user testing were used in the final year project.

- Concept testing: The idea of concept testing is to get user’s feedback on the app’s concept, innovative services, and personal opinions about the application.
- Beta testing: Beta testing is another testing method to get final feedback from users before releasing the app. Beta testing for iOS should follow these steps: finding testers, distributing the app via TestFlight Beta Testing product of Apple Inc, inviting testers to test the app with real data and web service, collecting feedback and possibly generating updates.
  [14.]

5 Testing Tools and Materials

5.1 Xcode Integrated Development Environment (IDE)

Xcode is Apple’s IDE for developers to build apps for Apple products involving the iPad, iPhone, Apple Watch and Macintosh computers. It also provides tools supporting the whole development process such as project management, code development, debugging, UI design, revision management, unit test, performance monitoring, and app distribution. Additionally, Xcode supports programming languages including Objective C, and Swift. Cocoa and Cocoa Touch framework are served as foundation frameworks. While Cocoa contains Foundation and AppKit frameworks, Cocoa Touch includes Foundation and UIKit frameworks.

5.2 XCTest Framework

XCTest framework allows developers to create and run unit tests, performance tests and UI tests for Xcode projects. XCTest has been integrated into XCode 5, which is the fifth version of Xcode launched in 2013. For UI testing with XCTest, new operating system (OS) features are required: iOS 9 for iOS devices’ deployment target and OS X 10.11. XCTest supports Swift and objective-C programming language.

5.3 Crashlytics Framework

Crashlytics is a testing tool that provides crash-reporting solutions by generating a full detail report on crashes and fatal errors that could lead to crashes. It will help saving time by collecting information about the amount of crashes and details about them. Popular mobile apps currently using Crashlytics are Twitter, Vine, Yelp, Kayak, TaskRabbit, Walmart, Groupon and Waze.

5.4 UserVoice Framework

The UserVoice framework allows testers and developers to collect users’ answers, ideas and feedback directly from the iOS application. The framework provides an easy integration process into the app which is shown later in testing implementation part and a useful platform for interactions between developers and users. The framework created a bridge between users and developers by ideas and feedback.

5.5 Testing Equipment

5.5.1 Real Devices

Real devices give a real feeling about the app on their hardware and all factors that can affect the application performance, for example network disconnection or call interrupt. For this project, iPhones are used as real devices to test the Buddify application.

Advantages of testing on real devices are:

- They allow testers to experience the app like a user.
- The app performs faster in real devices compared to simulators.
- UI element arrangement in different screen resolutions can be judged and tested better.
- They allow testers to test the app push notifications, geo-location and connection.
- Testing when incoming interrupts happen like calls and messages can only be done on real devices.
  [15.]

Disadvantages of using real devices are:

- iPhone devices are expensive.
- Device maintenance is also challenging.

5.5.2 Simulator

The Simulator tool in Xcode enables developers to simulate iOS devices on the Mac. However, the device's hardware cannot be simulated. Developers can just test the user interface and the functionalities of the app instantly by the Simulator as they are developing the app. Nevertheless, trouble from hardware can trigger errors or faults in the app.

The advantages of using Simulator are:

- It has a wide range of simulated iPhones for testing with different screen solutions.
- It is easy to set up since it is part of Xcode.
- It is quick to find major problems in the development phase.

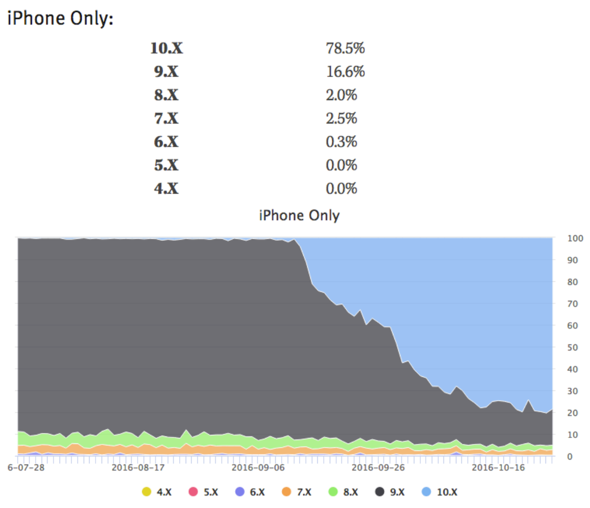
The disadvantages of using Simulator are:

- Hardware is not involved.
- An app can behave in a different way in Simulator so testing is not complete with only Simulator.

6 Buddify Application Concept

6.1 Category and Deployment Target

iOS is a mobile operating system developed by the Apple Inc. Only Apple mobile devices such as iPhones, iPads, iPods, and iWatches are licenced to run the iOS. Buddify is a native iOS application for iPhone users and its deployment target for iOS 8.0 or newer iOS versions. Therefore, Buddify supports iPhone 4S or newer iPhone generation such as iPhone 4S, iPhone 5, iPhone 5C, iPhone 5S, iPhone 6, iPhone 6 Plus, iPhone 6S, iPhone 6S Plus, iPhone SE, iPhone 7, and iPhone 7 Plus. Figure 4 shows that most iOS users (97.1%) are eligible for downloading and installing Buddify.


iPhone Only:

10.X	78.5%
9.X	16.6%
8.X	2.0%
7.X	2.5%
6.X	0.3%
5.X	0.0%
4.X	0.0%

Figure 4 iOS version usage in iPhone user in 2016. Reprinted from Smith (2016) [16].

As a matter of fact, testing on iOS involves less work compared to testing on Android, which is operated on many different devices. However, purchasing Apple devices is fairly expensive. Testing on all compatible real iPhone devices is still a challenging task for testers.

6.2 User Interface and Main Features

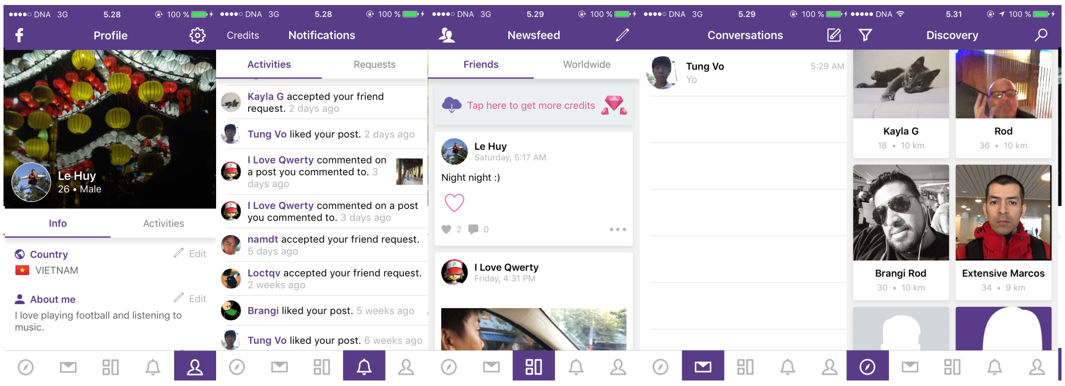
There are five main views in the application as shown in figure 5. Buddify creates a community where users can join and set up their own profile in profile view. After registration, users are asked to input name, age, gender and a profile image to start using Buddify. In the info sub-view, other information about users such as country, about me, looking for, profession, interests, music, movie, and languages spoken will be optional. In the activity sub-view, there are past activities of users. Buddify users are able to make friends in the community. By sending and accepting friend requests, users are able to be friends in the community.

In the notification view, there are two sub-views for friend requests and activity notification. Activities can be navigated by tapping the notifications, while friend requests will navigate to requested user profiles.

In the newsfeed view, users can see update activities of their friends or users with a specific country tag. Posts with text and a photo can be shared in the network with like and comment interaction.

Buddify users are able to communicate with each other by sending messages in the conversation view. Users can send photos in the message channel.

In the discovery view, Buddify users can discover other users in the network filtered by a country or location tag, gender, and age range. Users can also search other users from this view.



I love playing football and listening to music.

Kayla G accepted your friend request. 2 days ago

Tung Vo liked your post. 2 days ago

I Love Qwerty commented on a post you commented to. 3 days ago

I Love Qwerty commented on a post you commented to. 3 days ago

namdt accepted your friend request. 5 days ago

Loctqv accepted your friend request. 2 weeks ago

Brangi liked your post. 5 weeks ago

Tung Vo liked your post. 6 weeks ago

Le Huy
Saturday, 5:17 AM
Night night :)
2 0

I Love Qwerty
Friday, 4:31 PM

Tung Vo 5:29 AM
Yo

Figure 5 Main views in Buddify. Screenshots from an iPhone device.

6.3 Test Case Scenarios

Figure 6 shows all use cases listed and sorted to get an idea of user interaction in the system. The application will start with animated introduction views for showing what Buddify is all about. Then users will enter to the sign in/sign up view where unregistered users proceed with signing up and registered users continue with signing in. Unregistered users should use an unused email by the system and a six-character password in order to be verified and approved to become a new user. Registered users then can start to explore and interact with the app.

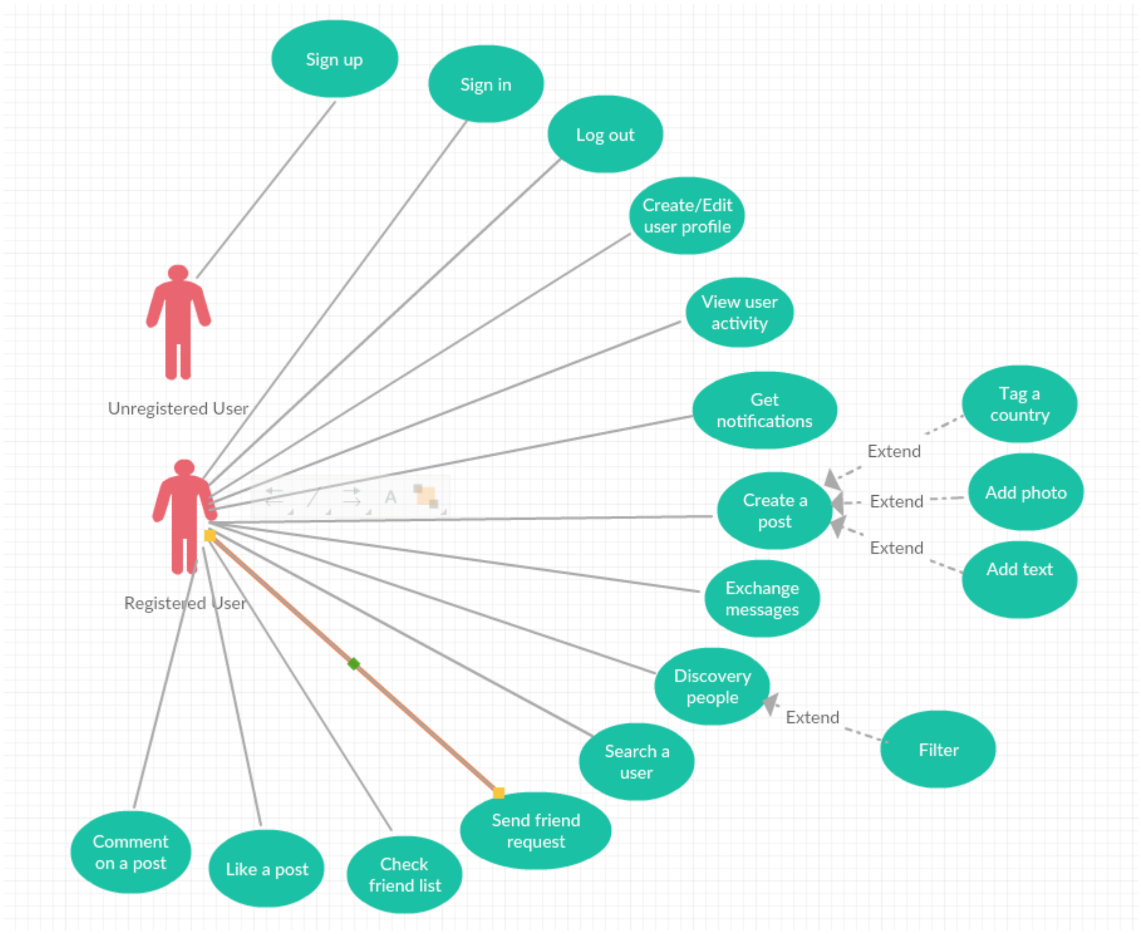


Figure 6 Use Case Diagram of Buddify

Based on the use cases, test case scenarios applicable for the app are described below:

- Sign up: users can sign up by providing a name, a valid email and a six-character password. Text fields need to be validated.
- Sign in: users can sign in using an email and password or a Facebook account. Text fields need to be validated.
- Log out: users can log out of the system.
- Create/Edit user profile: users can edit their basic information like profile picture, gender, birthday in the beginning view. Then, in the profile page, users can edit other detailed information like country, about me, looking for, profession, interests, music, movies and languages. When users tap to edit languages or countries, there will be views of lists of countries or languages and proficiency levels for them to choose. Users can re-edit their information later without any problem.
- View a user: users can see their own activities or other users' activities in activity view. In the activity view, there will be posts in chronological order. Users can also see their own bio or other users' bio in the info view.
- Get notifications: users will get and view notifications in the notification view about friend requests and activities related to their posts like comments or likes.
- Create posts: users are able to update a post with text, picture and country tag. Text input is limited to 256 characters and a/the country tag can be chosen from a list of countries. There are two options for adding a photo, which are from camera and gallery.
- Exchange messages: users can send and receive messages or even images.
- Discover people with filter: users can filter their search by gender, age range, location and nationality. Gender options are women, men, and both. Age range will be in a slider from 18 to 100. The location can be a nearby location or picked by user input with distance radius, which can be chosen within 200 km. If users do not want to have a location tag, they can choose a country tag instead. All users from a chosen country will be displayed in the discover view.
- Search a user: users are able to search other Buddify users in a text field.
- Send friend requests: users can be friends in Buddify by sending and accepting friend requests. Users should get notifications when having a request pending.
- Check friend list: users can see lists of their friends in the friend list view.

- Like posts: users can like a post. Number of likes for a post will be shown in that post via a button. By clicking on that button, users can see who liked the post.
- Comment on posts: users can comment on a post. The number of comments for a post will be shown in that post via a button. By clicking on that button, users can see who commented on the post.
- Change cover photo: users are able to change their cover photo by tapping the photo.
- Change profile photo: users are able to change their profile photo by tapping the photo.

6.4 Target Audiences

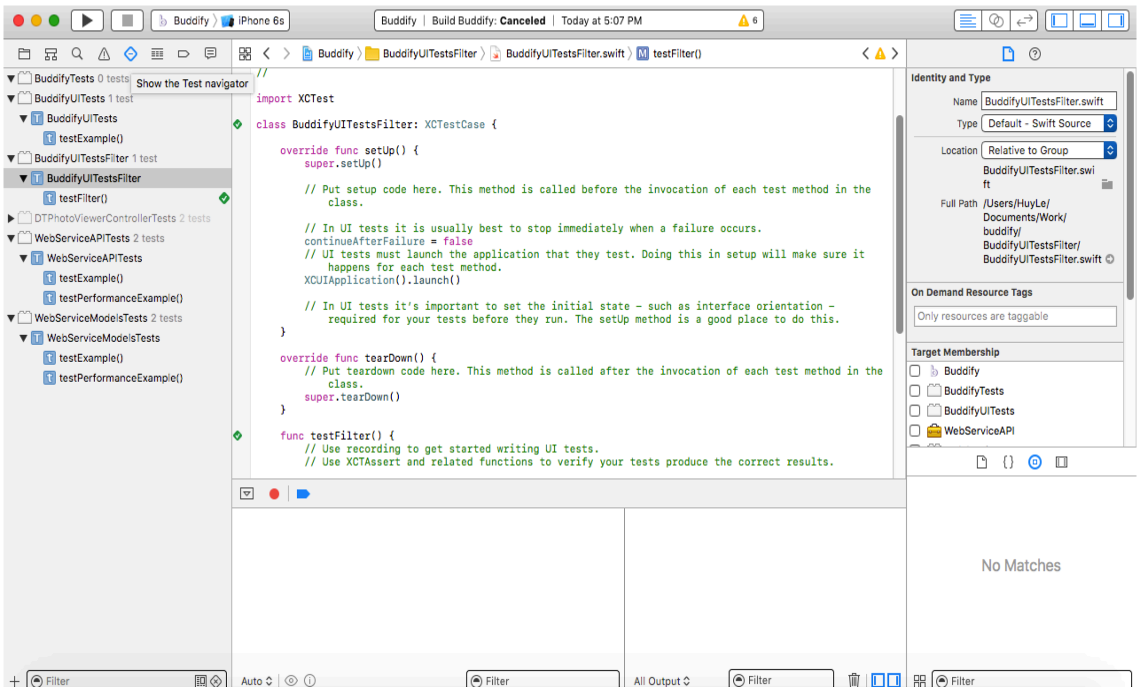
Buddify will focus on people from all over the world who are older than 18. Users can join Buddify to find international friends for language exchange, exchanging messages, friendship or potentially relationship. Besides, travellers may find Buddify to be useful to contact local people for travelling recommendation or tour guides.

7 Testing Implementation with Buddify

7.1 Setting up Testing Environment

Setting up a testing environment follows these steps:

- Join Apple developer program: it is essential to have an Apple developer account to deploy applications on iOS devices. With this account connected to iTunes Connect, developers can use TestFlight Beta Testing for beta testing apps before releasing the application.
- Install Xcode and iOS Software Development Kit (SDK): in this project, Xcode 8 version was used to test the Buddify app.
- Create a team-provisioning account and embed it into the app bundle in the Xcode project to launch apps on devices.



Buddify | Build Buddify: **Canceled** | Today at 5:07 PM

Buddify 〉 BuddifyUITestsFilter 〉 BuddifyUITestsFilter.swift 〉 testFilter()

```
▼ BuddifyTests 0 tests
▼ BuddifyUITests 1 test
  ▼ BuddifyUITests
      testExample()
▼ BuddifyUITestsFilter 1 test
  ▼ BuddifyUITestsFilter
      testFilter()
▶ DTPhotoViewerControllerTests 2 tests
▼ WebServiceAPITests 2 tests
  ▼ WebServiceAPITests
      testExample()
      testPerformanceExample()
▼ WebServiceModelsTests 2 tests
  ▼ WebServiceModelsTests
      testExample()
      testPerformanceExample()
```

```swift
//

import XCTest

class BuddifyUITestsFilter: XCTestCase {

    override func setUp() {
        super.setUp()

        // Put setup code here. This method is called before the invocation of each test method in the
            class.

        // In UI tests it is usually best to stop immediately when a failure occurs.
        continueAfterFailure = false
        // UI tests must launch the application that they test. Doing this in setup will make sure it
            happens for each test method.
        XCUIApplication().launch()

        // In UI tests it's important to set the initial state - such as interface orientation -
            required for your tests before they run. The setUp method is a good place to do this.
    }

    override func tearDown() {
        // Put teardown code here. This method is called after the invocation of each test method in the
            class.
        super.tearDown()
    }

    func testFilter() {
        // Use recording to get started writing UI tests.
        // Use XCTAssert and related functions to verify your tests produce the correct results.
```

Identity and Type

Name: BuddifyUITestsFilter.swift

Type: Default - Swift Source

Location: Relative to Group

BuddifyUITestsFilter.swift

Full Path: /Users/HuyLe/Documents/Work/buddify/BuddifyUITestsFilter/BuddifyUITestsFilter.swift

On Demand Resource Tags

Only resources are taggable

Target Membership

☐ Buddify
☐ BuddifyTests
☐ BuddifyUITests
☐ WebServiceAPI

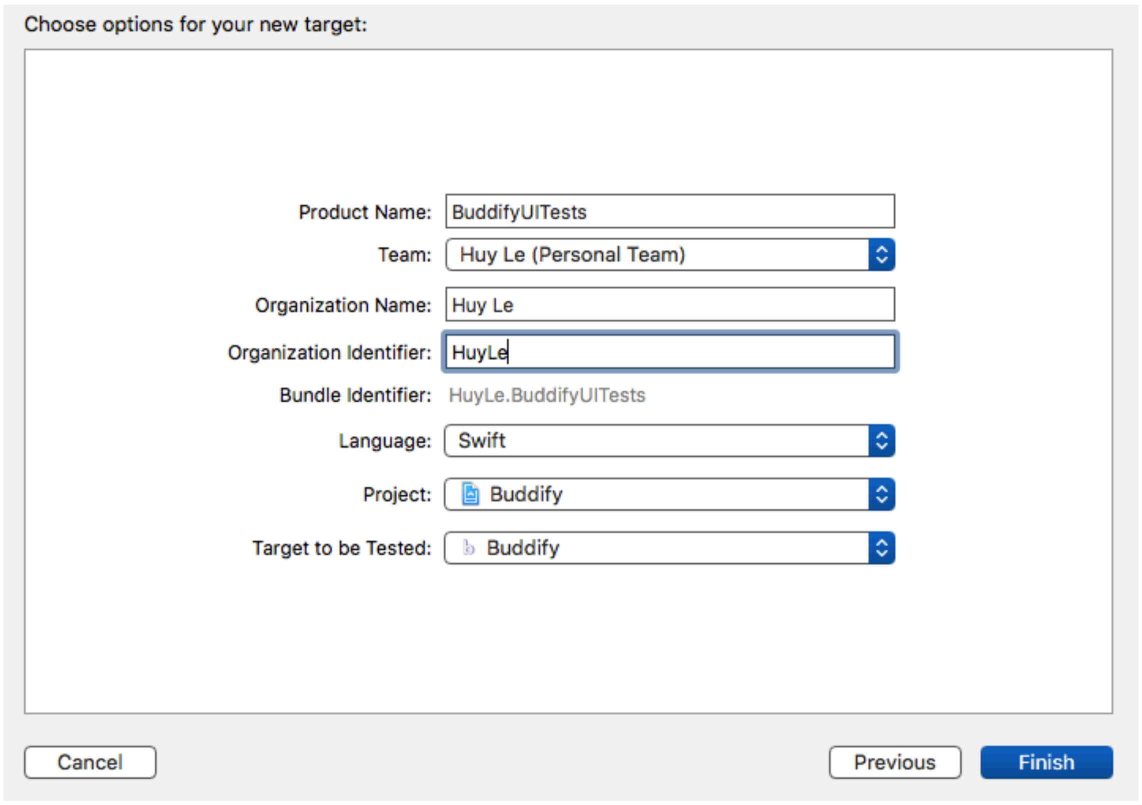
No Matches

Figure 7 Test navigator in Xcode. Screenshot from Xcode project [17].

After everything is all set (see figure 7), the testing process can start from the Xcode project.

7.2 Automated UI Testing

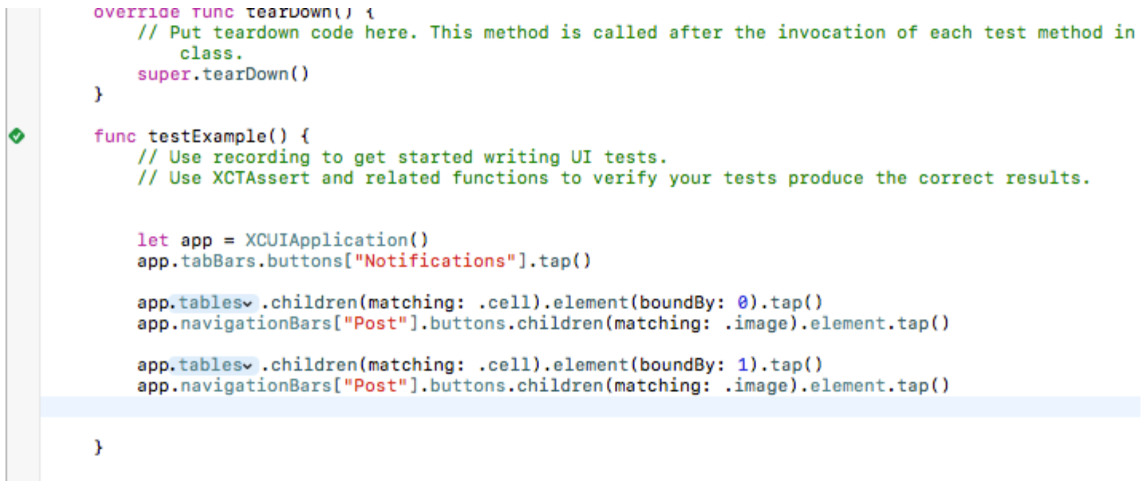
From the process of automated UI Testing described in Chapter 4, a UI testing implementation in which a user checks their notifications is demonstrated in this section. First, a test target was created called BuddifyUITests. Then the target to be tested was the application Buddify and the language for testing was Swift as in figure 8.


Choose options for your new target:

Product Name:	BuddifyUITests
Team:	Huy Le (Personal Team)
Organization Name:	Huy Le
Organization Identifier:	HuyLe
Bundle Identifier:	HuyLe.BuddifyUITests
Language:	Swift
Project:	Buddify
Target to be Tested:	Buddify

Figure 8 Creating a UI test target for Buddify. Screenshot from Xcode project [17].

Then, the keyboard curse is placed inside the testExample (see figure 7) method preparing for the UI recording. After pressing the record button in the right corner of the test class in figure 7, testing is performed in Simulator by checking a notification in the notification view. XCTest will automatically generate code in correspondence to actions that have happened in the simulator. After that, pressing the same record button can stop the recording process.



```
    override func tearDown() {
        // Put teardown code here. This method is called after the invocation of each test method in
           class.
        super.tearDown()
    }

    func testExample() {
        // Use recording to get started writing UI tests.
        // Use XCTAssert and related functions to verify your tests produce the correct results.

        let app = XCUIApplication()
        app.tabBars.buttons["Notifications"].tap()

        app.tables.children(matching: .cell).element(boundBy: 0).tap()
        app.navigationBars["Post"].buttons.children(matching: .image).element.tap()

        app.tables.children(matching: .cell).element(boundBy: 1).tap()
        app.navigationBars["Post"].buttons.children(matching: .image).element.tap()


    }
```

Figure 9 UI recording generated code in testExample(). Screenshot from Xcode project [17].

Figure 9 shows the UI recording code when testing the table of notification of Buddify. Based on the generated code, testers can add code to test when the user is checking other cells in the table by replacing the boundBy index. The testing script can be executed again to verify if the test is successful. The green check mark in figure 9 indicates that the test was successful.

Automated UI testing is a great way to test and understand the complexity of the UI element tree in the app. Automated UI testing also helps to detect bugs such as the one described below:

- When there are new notifications, the notification view does not get reloaded. So, when running the test, the newest notifications can be seen.
- When changing country in the profile view, the country does not get updated correctly when being changed many times.
- Images from posts in the newsfeed view occasionally cannot be tapped for viewing images due to a third party library.

After finishing the automated UI testing, there were few issues that cannot be solved. They are described below:

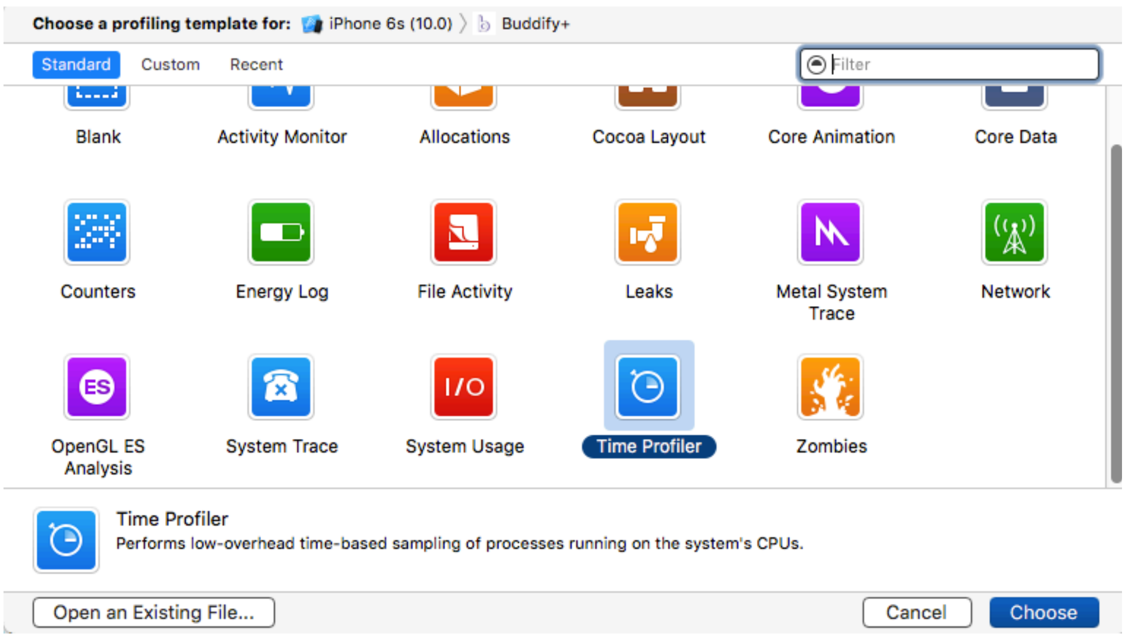
- It was hard to handle complex actions like map view zooming or pulling to refresh.
- It was impossible to interact with custom UI components.
- It took a lot of time to investigate and execute test cases.

7.3 Performance Testing

7.3.1 Using Instruments in Xcode

In this testing method, two main views in the app were tested as a demonstration. They are discovery view and newsfeed view. The discovery view shows all users available by a filter, and newsfeed view displays all posts from friends and users around the world. These views need to be tested for performance because a huge amount of work of UI rendering. This test helps to ensure that users will have smooth and responsive interaction with the app.

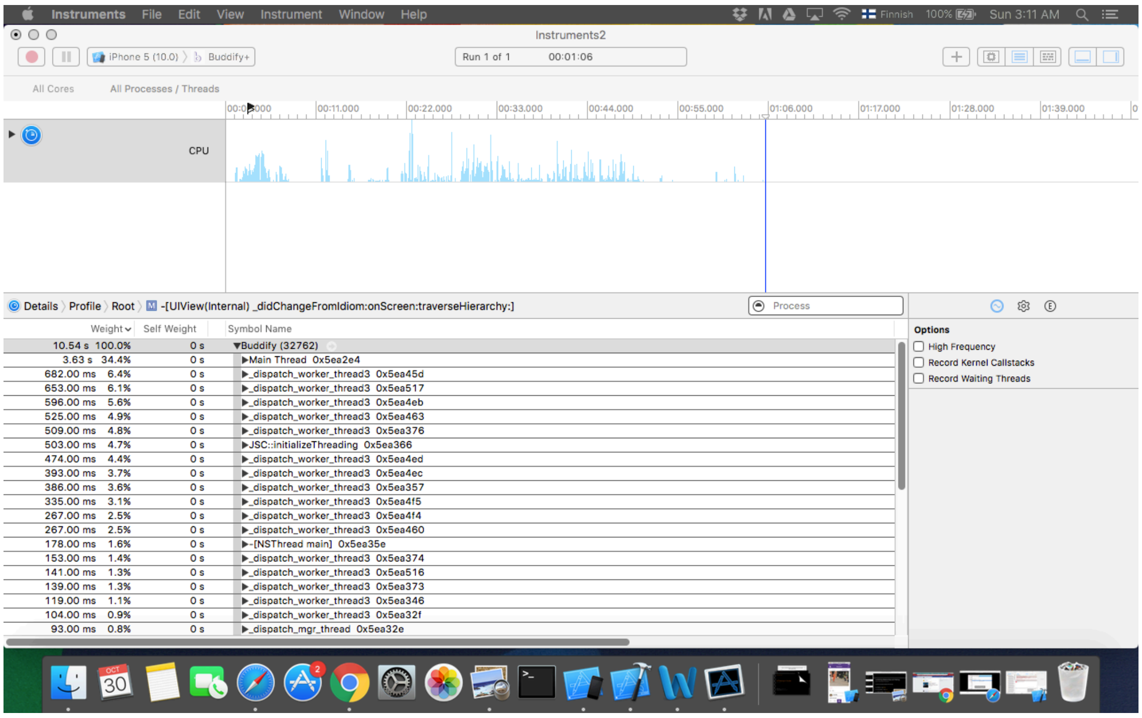
Performance testing was performed using the Instruments tool, which is integrated in Xcode. By launching Instruments from Xcode in the Xcode menu, there will be a template list of instruments to choose. The time profiler was selected for the performance test.



Time Profiler
Performs low-overhead time-based sampling of processes running on the system's CPUs.

Figure 10 Profiling templates in Instruments. Screenshot from the Instruments tool of Xcode [17].

The interface of Time Profiler Instrument appeared as in figure 10. There are two main areas in this interface. The top area is the time line view giving graphical information about CPU usage. The bottom area is the detail view, which represents collected samples aggregated by weight. Testing with Time Profiler started by pressing the record button in the left corner. The application launched and the time profiler started to collect data. After actions were performed on a test device, the stop button was pressed to stop and the investigation of the data started. In general, testing should be performed on real devices to give more accurate data about the app.

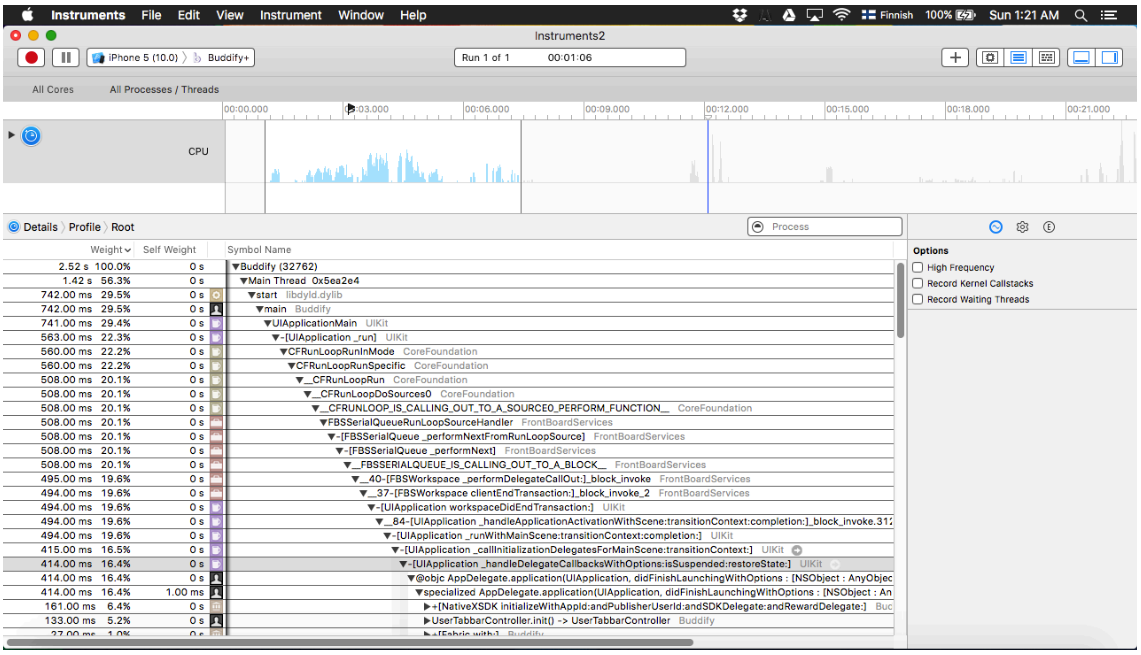


Details 〉 Profile 〉 Root 〉 -[UIView(Internal) _didChangeFromIdiom:onScreen:traverseHierarchy:]

Weight		Self Weight	Symbol Name
10.54 s	100.0%	0 s	▼Buddify (32762)
3.63 s	34.4%	0 s	▶Main Thread 0x5ea2e4
682.00 ms	6.4%	0 s	▶_dispatch_worker_thread3 0x5ea45d
653.00 ms	6.1%	0 s	▶_dispatch_worker_thread3 0x5ea517
596.00 ms	5.6%	0 s	▶_dispatch_worker_thread3 0x5ea4eb
525.00 ms	4.9%	0 s	▶_dispatch_worker_thread3 0x5ea463
509.00 ms	4.8%	0 s	▶_dispatch_worker_thread3 0x5ea376
503.00 ms	4.7%	0 s	▶JSC::initializeThreading 0x5ea366
474.00 ms	4.4%	0 s	▶_dispatch_worker_thread3 0x5ea4ed
393.00 ms	3.7%	0 s	▶_dispatch_worker_thread3 0x5ea4ec
386.00 ms	3.6%	0 s	▶_dispatch_worker_thread3 0x5ea357
335.00 ms	3.1%	0 s	▶_dispatch_worker_thread3 0x5ea4f5
267.00 ms	2.5%	0 s	▶_dispatch_worker_thread3 0x5ea4f4
267.00 ms	2.5%	0 s	▶_dispatch_worker_thread3 0x5ea460
178.00 ms	1.6%	0 s	▶-[NSThread main] 0x5ea35e
153.00 ms	1.4%	0 s	▶_dispatch_worker_thread3 0x5ea374
141.00 ms	1.3%	0 s	▶_dispatch_worker_thread3 0x5ea516
139.00 ms	1.3%	0 s	▶_dispatch_worker_thread3 0x5ea373
119.00 ms	1.1%	0 s	▶_dispatch_worker_thread3 0x5ea346
104.00 ms	0.9%	0 s	▶_dispatch_worker_thread3 0x5ea32f
93.00 ms	0.8%	0 s	▶_dispatch_mgr_thread 0x5ea32e

Figure 11 Performance test results. Screenshot from the Instruments tool of Xcode [17].

As shown in figure 11, the CPU usage was displayed in a blue graph in the time line view. In the call tree view, data about weight, self-weight and symbol name were shown. Weight means the number of samples of a particular portion of the call tree appeared in, and self-weight means the number of samples multiplied by the time between each sample [12]. Figure 11 illustrates that 34% of the CPU usage was from the main thread and the rest was from background threads. To get more details of what exactly the app has been doing, the area with high CPU usage was focused and examined.

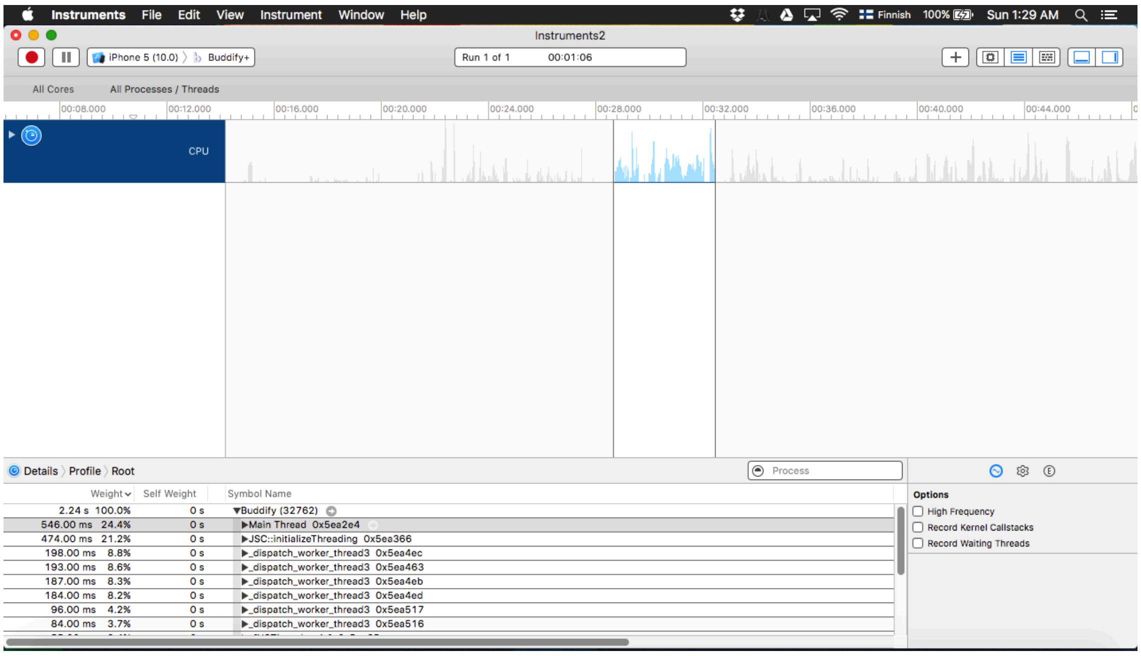


```
Weight      Self Weight   Symbol Name
   2.52 s  100.0%    0 s  ▼Buddify (32762)
   1.42 s   56.3%    0 s   ▼Main Thread  0x5ea2e4
742.00 ms   29.5%    0 s    ▼start  libdyld.dylib
742.00 ms   29.5%    0 s     ▼main  Buddify
741.00 ms   29.4%    0 s      ▼UIApplicationMain  UIKit
563.00 ms   22.3%    0 s       ▼-[UIApplication _run]  UIKit
560.00 ms   22.2%    0 s        ▼CFRunLoopRunInMode  CoreFoundation
560.00 ms   22.2%    0 s         ▼CFRunLoopRunSpecific  CoreFoundation
508.00 ms   20.1%    0 s          ▼__CFRunLoopRun  CoreFoundation
508.00 ms   20.1%    0 s           ▼__CFRunLoopDoSources0  CoreFoundation
508.00 ms   20.1%    0 s            ▼__CFRUNLOOP_IS_CALLING_OUT_TO_A_SOURCE0_PERFORM_FUNCTION__  CoreFoundation
508.00 ms   20.1%    0 s             ▼FBSSerialQueueRunLoopSourceHandler  FrontBoardServices
508.00 ms   20.1%    0 s              ▼-[FBSSerialQueue _performNextFromRunLoopSource]  FrontBoardServices
508.00 ms   20.1%    0 s               ▼-[FBSSerialQueue _performNext]  FrontBoardServices
508.00 ms   20.1%    0 s                ▼__FBSSERIALQUEUE_IS_CALLING_OUT_TO_A_BLOCK__  FrontBoardServices
495.00 ms   19.6%    0 s                 ▼__40-[FBSWorkspace _performDelegateCallOut:]_block_invoke  FrontBoardServices
494.00 ms   19.6%    0 s                  ▼__37-[FBSWorkspace clientEndTransaction:]_block_invoke_2  FrontBoardServices
494.00 ms   19.6%    0 s                   ▼-[UIApplication workspaceDidEndTransaction:]  UIKit
494.00 ms   19.6%    0 s                    ▼__84-[UIApplication _handleApplicationActivationWithScene:transitionContext:completion:]_block_invoke.312
494.00 ms   19.6%    0 s                     ▼-[UIApplication _runWithMainScene:transitionContext:completion:]  UIKit
415.00 ms   16.5%    0 s                      ▼-[UIApplication _callInitializationDelegatesForMainScene:transitionContext:]  UIKit
414.00 ms   16.4%    0 s                       ▼-[UIApplication _handleDelegateCallbacksWithOptions:isSuspended:restoreState:]  UIKit
414.00 ms   16.4%    0 s                        ▼@objc AppDelegate.application(UIApplication, didFinishLaunchingWithOptions : [NSObject : AnyObjec
414.00 ms   16.4% 1.00 ms                        ▼specialized AppDelegate.application(UIApplication, didFinishLaunchingWithOptions : [NSObject : An
161.00 ms    6.4%    0 s                          ▶+[NativeXSDK initializeWithAppId:andPublisherUserId:andSDKDelegate:andRewardDelegate:]  Buc
133.00 ms    5.2%    0 s                          ▶UserTabbarController.init() -> UserTabbarController  Buddify
 27.00 ms    1.0%    0 s                          ▶+[Fabric with:]  Buddify
```

Options
- [ ] High Frequency
- [ ] Record Kernel Callstacks
- [ ] Record Waiting Threads

Figure 12 CPU usage peaks at the beginning of the app. Screenshot from the Instruments tool of Xcode [17].

By dragging from one end to another end in the graph, all the calls executed in a specific area were filtered and highlighted blue on the top area as in figure 12. From the detail view, the tester can navigate to the call tree and check high CPU calls. These calls can be tracked to the development code by a double click. As the call tree in figure 12 shows, the high usage CPU, i.e. 56.3%, came from UI initialization and rendering, which is understandable since it was the beginning time when the app needed to process much data.



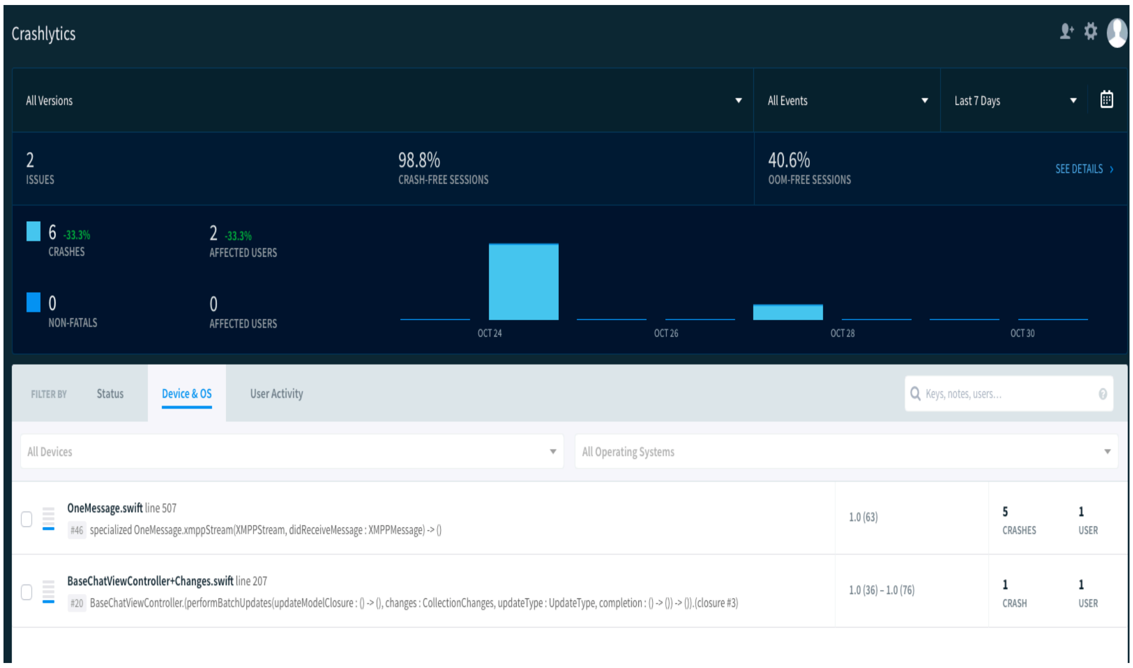
Weight		Self Weight	Symbol Name
2.24 s	100.0%	0 s	▼Buddify (32762)
546.00 ms	24.4%	0 s	▶Main Thread 0x5ea2e4
474.00 ms	21.2%	0 s	▶JSC::initializeThreading 0x5ea366
198.00 ms	8.8%	0 s	▶_dispatch_worker_thread3 0x5ea4ec
193.00 ms	8.6%	0 s	▶_dispatch_worker_thread3 0x5ea463
187.00 ms	8.3%	0 s	▶_dispatch_worker_thread3 0x5ea4eb
184.00 ms	8.2%	0 s	▶_dispatch_worker_thread3 0x5ea4ed
96.00 ms	4.2%	0 s	▶_dispatch_worker_thread3 0x5ea517
84.00 ms	3.7%	0 s	▶_dispatch_worker_thread3 0x5ea516

Figure 13 CPU high usage in the discovery view. Screenshot from the Instruments tool of Xcode [17].

Taking another peak data sample from the graph, the main thread has used 24.4% CPU (see figure 13) since it needs to load UI elements and data from the backend service. However, the CPU usage 24% was an acceptable amount of CPU work. Thanks to using the ASyncDisplayKit framework to optimize UI rendering in the background, the main thread was not doing so much work and giving more space for user interaction to happen. If there are bugs or optimization work, then testers can run the test again to verify the changes.

7.3.2 Using Crashlytics

One of the most common factors that affect the app performance is crash. The tester can collect crashes from users by integrating and implementing the Crashlytics framework in the Buddify project. After creating an account on the website fabric.io and integrating the framework in the project, the app was ready to collect crash data.



2 ISSUES
98.8% CRASH-FREE SESSIONS
40.6% OOM-FREE SESSIONS

6 -33.3% CRASHES
2 -33.3% AFFECTED USERS
0 NON-FATALS
0 AFFECTED USERS

OneMessage.swift line 507 #46 specialized OneMessage.xmppStream(XMPPStream, didReceiveMessage : XMPPMessage) -> ()	1.0 (63)	5 CRASHES	1 USER
BaseChatViewController+Changes.swift line 207 #20 BaseChatViewController.(performBatchUpdates(updateModelClosure : () -> (), changes : CollectionChanges, updateType : UpdateType, completion : () -> ()) -> ()).(closure #3)	1.0 (36) – 1.0 (76)	1 CRASH	1 USER

Figure 14 Crashlytics Dashboard with crash data. Screenshot from Fabric dashboard [18].

Crashlytics provides an application programming interface (API) for testers to collect crash data from users, and crash reports will be displayed on the website fabric.io when developers sign in with their Crashlytics account. Developers can also log user information when the app crashes with the Crashlytics SDK. By doing so, crash reports and user information will be showed in the dashboard as in figure 14. In the dashboard, it is easy to see that there are two bugs that caused six crashes and these bugs were related to message function. The dashboard clearly has a user-friendly interface for testers and developers. Crash report has been useful to find bugs or memory leaks after releasing the app for beta testing.

7.4 Exploratory Testing

Exploratory testing includes testing with all use cases described in the chapter 4. Basically, the tester will register as a user and execute all functionalities available in the app. This method has found the most bugs in the app from both UI and back-end data. For example, some bugs found during this test are:

- Text view failed to update its frame when text changes
- Test size in chat was relatively small
- Push notifications did not show up in real devices

- The app crashed when pulling to refresh the view friend list
- The number of like label did not show the right number of like when tapping many times

Testing with Internet connection type was carried out by switching between Wi-Fi and 3G connections of test device. The app appeared to work well in both cases. Crashing also did not happen when the app entered plane mode. Also, users were always informed when there was a connection problem. Testing in case in-call and SMS interruption was done as well. No problems occurred during the interruption. Buddify kept working fine after the interruption.

7.5 User Testing

7.5.1 Beta Testing

Beta testing utilized an Apple product called TestFlight Beta Testing. TestFlight Beta Testing allows developers to distribute prerelease builds of the app to test users for collecting feedback and prepare to launch the app to the Apple App Store [19].

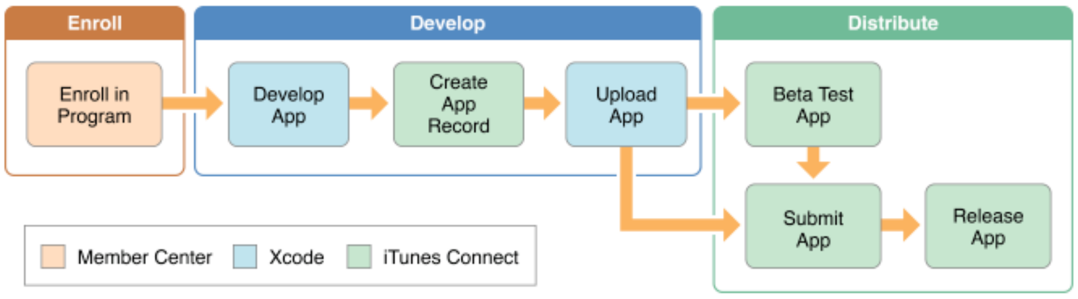


Figure 15 iOS App Distribution Process. Reprinted from About iTunes Connect [20]

At first, a record of the app was created on the iTunes Connect. iTunes Connect is a web application for developers to submit and manage their app for releasing in the App Store or the Mac Store [20]. Figure 15 above shows the role of the iTunes Connect in the whole iOS app distribution process. Then the app was submitted and uploaded to iTunes Connect by using Xcode. The beta app's distribution was configured in the iTunes Connect site in which there are two kinds of testers: internal and external tester.

- Internal tester: the beta app allows inviting up to 25 internal testers per app by emails. These testers should be in same team with a different role assigned in the iTunes Connect. These roles can be admin, technical, app manager, developer, or marketer. Internal testers can start testing the app right away after testing invitation.
- External tester: Up to 2000 users can be invited as external testers by getting them invitations by emails. These users do not have access in the iTunes Connect but can download and install the app. Apple will review the beta apps before external testers can start to test.

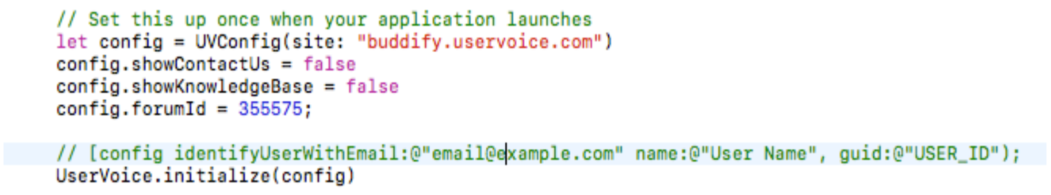
After adding tester emails in two channels and selecting the app build version, testers should receive invitation emails to download TestFlight app. In the TestFlight, testers can start downloading and installing the app to their iPhones. After getting feedback from users, developers can try to improve the app and submit a better build. Testers will get notified about new updates of the app and they can install the updates to test the app. Even though the beta testing is not compulsory for distributing the app, it is a really significant step to get the app tested so that the app can bring a great experience to its users. Developers can keep better builds of the app before it is ready to be released. Buddify had 76 builds before the app is submitted and approved by Apple.

Unfortunately, the app did not get any feedback from the TestFlight but the team has been inviting a number of users to test the application. With the use of real users and data, it would be easier for developers and testers to detect bugs. Overall, there was not any serious problem that could cause testers to send feedback. Small bugs were detected and fixed quickly by the Buddify team. The app kept updated better builds for testers.

7.5.2 Concept Testing

User testing was performed using the UserVoice framework. The framework provides APIs for testers to collect user feedback and to display the feedback in a forum. Developers should have their UserVoice accounts in order to integrate the UserVoice framework into the Xcode project. Users can access the forum integrated into the application and give feedback about the app. At the same time, they are allowed to see others' feedback and rate it by liking the feedback post. By logging into the website Uservoice.com, testers can collect ideas and feedback from users to improve and upgrade better versions of the app.

After integrating the UserVoice framework into Buddify Xcode project using CocoaPods, and importing UserVoice.h to the bridging header, configuration is created and initialized as in figure 16 below:

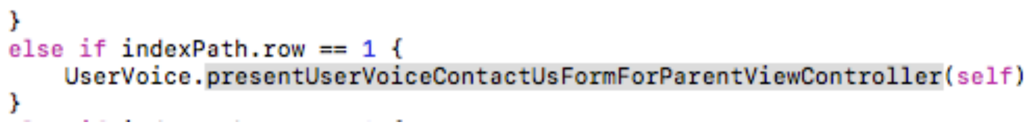


```
// Set this up once when your application launches
let config = UVConfig(site: "buddify.uservoice.com")
config.showContactUs = false
config.showKnowledgeBase = false
config.forumId = 355575;

// [config identifyUserWithEmail:@"email@example.com" name:@"User Name", guid:@"USER_ID");
UserVoice.initialize(config)
```

Figure 16 UserVoice configuration. Screenshot fromXcode project [17].

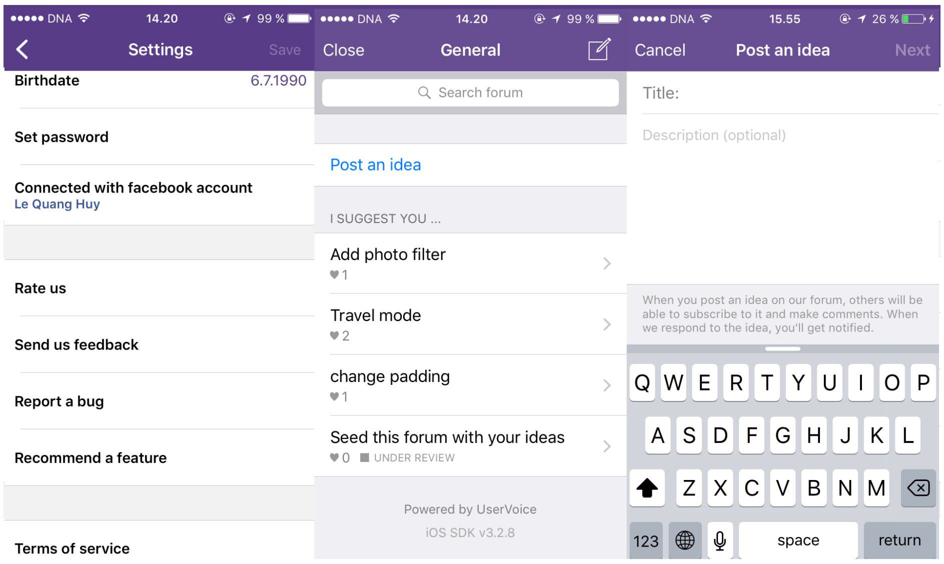
Then in the view Settings, the forum view was implemented in the Send us Feedback cell of the Settings table view as can be seen below:



```
}
else if indexPath.row == 1 {
    UserVoice.presentUserVoiceContactUsFormForParentViewController(self)
}
```

Figure 17 UserVoice presents the forum view. Screenshot from Xcode project [17].

As figure 18 shows, the view of the feedback forum appeared with a list of feedback in the table view and Post an idea button for users to send new features or simply feedback. The number of likes was shown under feedback as well. The data in the forum will be valuable for developers and testers to improve and upgrade better versions of the app.



When you post an idea on our forum, others will be able to subscribe to it and make comments. When we respond to the idea, you'll get notified.

Figure 18 Accessing the feedback forum. Screenshots from an iPhone device.

As shown in figure 18, there is some feedback about the app from test users. First, there was a suggestion about adding a photo filter. This is a good suggestion for a later feature in the app when users upload their photos to the app. The next one is about adding a travel calendar in Buddify. It is also a good idea for travellers to share their trips and stories. Since Buddify also targets travellers, this is also a promising feature in the future. The last one is feedback about changing the padding of a list, which is not clear because there was no explanation on this issue. Overall, the feedback forum is quite helpful to collect feedback from users in the long run. Users can keep giving feedback and recommendations as the app is developed.

8 Conclusion

This final year project aimed to debug a native social network iOS application called Buddify to improve the application’s quality. The project included studies about testing, testing methodologies and their implementation to test Buddify. Applying different testing methods, many aspects of the app Buddify were tested. Many bugs and crashes were found in the project when carrying out both automated testing and manual testing. While automated UI testing allows testers to understand the complexity of the UI element tree in the app, app flow and basic functionalities, performance testing enables testers to keep and maintain decent performance of the app. Manual testing was useful to detect bugs quickly as well as to collect user feedback especially before the releasing phase of the app. Thanks to the testing process, users are able to enjoy a fully functional, responsive and smooth app. The project also provides guidelines for testers or developers to start testing on iOS applications.

Generally, the app was tested well and intensively in the past two months to be released for the first time. Many important testing methods were applied and implemented during the testing process. There are also other third-party testing frameworks and more testing methods like endurance testing or unit testing that could have been applied for the test. As the app is released, the testing process will be re-applied and expanded to maintain and improve the app. Testing should be done continuously and wisely for better future versions as bugs and errors which are found early will save lot of time and effort for developers in the development process.

References

1. ISTQB Exam Certification. What Are the Software Development Models [online].
URL: http://istqbexamcertification.com/what-are-the-software-development-models/.
Accessed October 7, 2016.

2. Handschuh S. Software Development Life Cycle Overview [online]. 2013.
URL:
http://www.gamasutra.com/blogs/StanleyHandschuh/20130929/201241/Software_Development_Life_Cycle_Overview_Part_1_of_6.php.
Accessed October 7, 2016.

3. Statista. Number of Available Apps in the Apple App Store from July 2008 to June 2016 [online]. 2016.
URL: https://www.statista.com/statistics/263795/number-of-available-apps-in-the-apple-app-store/.
Accessed October 7, 2016.

4. Statista. Number of Available Applications in the Google Play Store from December 2009 to September 2016 [online]. 2016.
URL: https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/.
Accessed October 7, 2016.

5. ISTQB Exam Certification. Why Is Software Testing Necessary [online].
URL: http://istqbexamcertification.com/why-is-testing-necessary/.
Accessed October 7, 2016.

6. Utting M, Legeard B. Practical Model-Based Testing: A Tools Approach [online]. Morgan Kaufmann, 2010.
URL:
http://site.ebrary.com.ezproxy.metropolia.fi/lib/metropolia/detail.action?docID=10155918.
Accessed November 6, 2016.

7. Myers G, Sandler C, Badgett T. Art of Software Testing [online]. 3rd ed. Wiley, 2011.
URL:
http://site.ebrary.com.ezproxy.metropolia.fi/lib/metropolia/detail.action?docID=10500924.
Accessed November 6, 2016.

8. Software Testing Class. What Is a White Box Testing [online].
URL: http://www.softwaretestingclass.com/white-box-testing/.
Accessed November 16, 2016.

9. Apple Inc. UI Testing in Xcode [online].
URL: https://developer.apple.com/videos/play/wwdc2015/406/.
Accessed October 10, 2016.

10. Apple Inc. Testing with Xcode [online].
URL:
https://developer.apple.com/library/content/documentation/DeveloperTools/Conceptual/testing_with_xcode/chapters/03-testing_basics.html#//apple_ref/doc/uid/TP40014132-CH3-SW7.
Accessed October 15, 2016.

11. Apple Inc. About Instruments [online].
URL:
https://developer.apple.com/library/content/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/index.html#//apple_ref/doc/uid/TP40004652-CH3-SW1.
Accessed October 15, 2016.

12. Apple Inc. Using Time Profiler in Instruments [online].
URL: https://developer.apple.com/videos/play/wwdc2016/418/.
Accessed October 20, 2016.

13. Tutorialspoint. Exploratory Testing [online].
URL:
https://www.tutorialspoint.com/software_testing_dictionary/exploratory_testing.htm.
Accessed November 8,2016.

14. Guru99. Getting Started with iOS Testing [online].
URL: http://www.guru99.com/getting-started-with-ios-testing.html
Accessed November 8,2016.

15. Hechtel E. Mobile Device Emulator/Simulator Vs Real Device [online]. 2016.
URL: https://testobject.com/blog/2016/05/mobile-device-emulator-vs-real-device.html.
Accessed 2016 October 7, 2016.

16. Smith D. iOS Version Stats [online]. 2016.
URL: https://david-smith.org/iosversionstats/.
Accessed October 8, 2016.

17. Apple Inc. Xcode [computer program]. Version 8.0. California, United States. September 13, 2016.

18. Fabric. https://fabric.io/ [computer program]. Boston, Massachusetts. October 22, 2016.

19. Apple Inc. TestingFlight Beta Testing [online].
URL:
https://developer.apple.com/library/content/documentation/LanguagesUtilities/Conceptual/iTunesConnect_Guide/Chapters/BetaTestingTheApp.html#//apple_ref/doc/uid/TP40011225-CH35-SW3.
Accessed October 27, 2016.

20. Apple Inc. About iTunes Connect [online].
URL:
https://developer.apple.com/library/content/documentation/LanguagesUtilities/Conceptu-
al/iTunesConnect_Guide/Chapters/About.html#//apple_ref/doc/uid/TP40011225-CH1-SW1.
Accessed October 27, 2016.