Advanced Programming for Mobile Devices - Software Report

James Donohue - james.donohue@bbc.co.uk

Contents

1 Abstract

2 Introduction

This software report accompanies the completed iOS app *FindMyBike*. It continues some of the discussion begun in the proposal already submitted.

The Xcode source code for *FindMyBike* can be found in the zip file provided with this document.

2.1 Specifications

The development environment used for the app adheres to the requirements of the assignment brief:

- Xcode 8
- Swift 3.1
- iOS 10.3 SDK
- Platforms: iPhone 6 and later (iOS simulator or physical device)

The following Apple frameworks were used in building the application:

- CoreLocation
- CoreBluetooth
- UserNotifications
- UIKit
- Foundation

No third-party libraries or other frameworks were used.

2.2 User guide

A user guide for the app can be found on the project website and is included as a separate file `userguide.pdf` supplied along with this report.

2.3 Usability evaluation

A key goal in the design of the application was simplicity and clarity. The Apple Human Interface Guidelines (2017d) state that 'clarity' is one of the themes that differentiates iOS from other platforms, defining it in terms of legibility of text and 'a sharpened focus on functionality'.

2.3.1 Clarity

One example of applying this theme in practice is the content and layout of the cells in the main table view. Rather than specifying a particular size and weight for the text in the `UILabel` components, predefined iOS text styles of 'Headline' and 'Subhead' were used. This enables the platform's Dynamic Type mechanism to respond to user preferences and accessibility settings (Apple, 2016b) and thereby ensure legibility.

The principle of clarity also applies to the choice of what level of detail is appropriate for each level of navigation. For example, the table view cells could have included the iBeacon settings for each bike (UUID, major and minor) but this would have created a more cluttered user interface without adding much value. Instead it was decided to only show these values when the user actually edits their bike details.

Conversley, some testers have reported that the text size used for labels and text fields on the *Edit Bike* scene is too small. This screen uses an explicit font size of 14 points in order to allow the beacon UUID to be displayed in its entirety without truncating the text on a 4.7" screen such as the iPhone 6 or later while maintaining consistency with the other inputs. This was important during the early phases of app development where the UUID was entered manually, a difficult an error-prone process. Even so, on smaller screens (such as the 4" iPhone SE) the UUID is still truncated. As in the final app the UUID is no longer editable, the easiest solution to this problem would be to remove the application UUID from the Edit Bike screen completely.

2.3.2 Discoverability

Another usability factor that was considered was discoverability, meaning the level to which a new user can determine what actions can be performed and how to perform them. The right chevron displayed alongside the 'My Bike' cell is used as a visual hint that the user can tap the cell in order to view more information. The chevron (known as a 'disclosure indicator' in iOS parlance) signals what Norman (Norman, 2013) terms an 'affordance', namely the ability to access a more detailed view. Note that while other ways of signalling this affordance are possible, by using the convention of the standard table view disclosure indicator, the user can apply the knowledge they have learned from using other iOS apps. This relates to the stated Apple design principle of 'consistency', specificaly that an app should implement "familiar standards and paradigms by using system-provided interface elements" (Apple, 2017d).

2.3.2.1 Inputs

The 'Make' and 'Model' text inputs are currently free text fields. As there is a finite set of common motorcycle makes, it would be more convenient to allow the user to choose a make from a pre-populated list. The iOS equivalent of typical 'drop-down' selection boxes found on web pages are 'pickers', which present a scrollable list of values, usually at the bottom of the screen. This may be a suitable way of implementing the selection.

2.3.2.2 Photos and colours

Early in the development of the *Edit Bike* scene a method for adding a custom photo to illustrate the user's bike was added through the 'Choose photo' button. This uses the system `UIImagePickerController` to allows the user to choose a photo from their Photo Library, which is then displayed in the `RangingTableViewController`. Although this is a useful feature for an individual user, it creates challenges if these photos need to be shared with other users. These include:

- **Storage** - a network-available location for uploading and downloading user photos needs to be created, with the necessary access from the API
- **Security** - access control must be enforced to prevent users from modifying each others' photos. In addition some form of automated virus/malware scanning should be implemented

- **Privacy** - the uploaded photos should only be accessible to registered users when the associated bike is detected
- **Content safety** - uploaded photos should be checked to ensure they do not contain adult content or other imagery that violates App Store policies.

The last requirement is emphasised in section 1.2 of the App Store Review Guidelines ("User Generated Content") (Apple, 2014a), which describes the requirement for filtering and also states that apps featuring user-generated content must include "mechanism to report offensive content and timely responses to concerns". In the light of these additional concerns, it was decided to only store photos locally on the device and not share them with other users. In place of photos, the ability for users to input their bike's colour was added, so that missing bikes can be easily identified without the risks associated with allowing photos.

2.3.3 Evaluation of usability issues

2.3.3.1 Keyboards

When inputting the beacon minor on the 'Edit Bike' screen, a standard QWERTY keyboard is displayed. iOS supports displaying specialised keyboards for different types of text field using the `UIKeyboardType` enumeration. Using a numeric keyboard may help the user by preventing them inputting non-numeric characters, however the supplied numeric keyboards do not offer a built-in 'Done' key to allow the user to submit their input, as shown in Figure 1. Some options would be to implement this separately using a `UIToolbar` placed above the number pad, or a custom input view for numeric input as used by the Numbers app (Apple, 2017c).

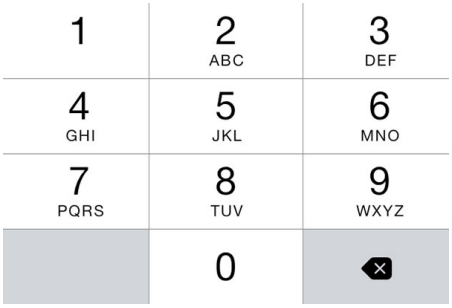

Figure 1: iOS built-in `UIKeyboardType.NumberPad` keyboard without Done key

2.3.3.2 Beacon identifiers

In the Core Location framework, iBeacons are grouped into regions identified by a *proximity UUID* - a 128-bit value that is intended to identify a certain beacon type or organisation (Apple, 2016a), optionally combined with a *major* and *minor* value (16-bit unsigned integers). The framework makes a distinction between 'monitoring' for a beacon regions (which may include many beacons) and 'ranging' for individual beacons within a region once entered, further recommending that 'ranging' is only performed when the app is in the foreground (Apple, 2014b) for reasons of accuracy.

One usability problem with the current app design arises from Apple's decision to restrict the Core Location framework to monitor 20 regions at one time. According to Apple this is because regions are a limited "shared system resource" (Apple, 2016a), although this contrasts with the Android platform

which has no OS-imposed limit on the number of beacon identifiers to be monitored. The 20-region limit restriction is described by one becacon retailer as "a great example of unnecessary complexity invented by great minds" (BeaconZone, 2017).

In a conventional geofencing-type application where the regions are defined based on geographical coordinates, the 20-region limitation can be mitigated by regularly altering the list of monitored regions to include only those already-known locations closest to the user's current location (Drobnik, 2015), however in a beacon-based application like *FindMyBike* the geographical location of each beacon is not known in advance, making this impossible.

Because of the relative complexity of configuring an iBeacon's identifier, for the greatest user convenience *FindMyBike* would be able to monitor for any identifier, so that the default values set by beacon manufacturers could be used without the need for reconfiguration. However, Core Location's restriction on the number of regions that can be monitored at once would limit the number of individual missing bikes that could be detected to 20.

Consequently, the current implementation of the app defines a single proximity UUID and major pair that is monitored by the application in the background, defined by the `Constants` struct as follows:

```
static let applicationUUID = UUID(uuidString: "21EECF71-D5C7-4A00-9B90-27C94B5146EA")!
static let applicationMajor = UInt16(1)
```

The following outlines the approach used:

- The user's iPhone detects that it has entered the application-defined `CLBeaconRegion` (defined by the identifiers above and the label `io.github.jamesdonoh.FindMyBike`)
- iOS sends the `didEnterRegion` notification to the app's instance of `CLLocationManagerDelegate`, which is provided by `ProximityMonitor`
- `ProximityMonitor` begins ranging for all beacons within the region
- When individual beacons are ranged, `ProximityMonitor` passes the minor value of each beacon to `BikeRegistry` to determine if any of them represent missing bikes
- If any minors match, a notification is sent to the user

Note that this implementation disregards Apple's advice to only perform ranging when the app is in the foreground, but given the limited number of regions that can be monitored at one time it is the only practical way to achieve the desired functionality for more than 20 missing bikes.

2.4 Application design

Figure 2 shows an overview of the main classes in *FindMyBike* and their relationships. Classes provided by the underlying UIKit framework are shown in *italics*.

2.4.0.1 View Controllers

The app implements the following custom subclasses of `UIViewController`:

- `AppEventViewController`
- `MainViewController`
- `EditBikeViewController`
- `ColourTableViewController`
- `RangingTableViewController`

Certain design features of these controllers are discussed below.

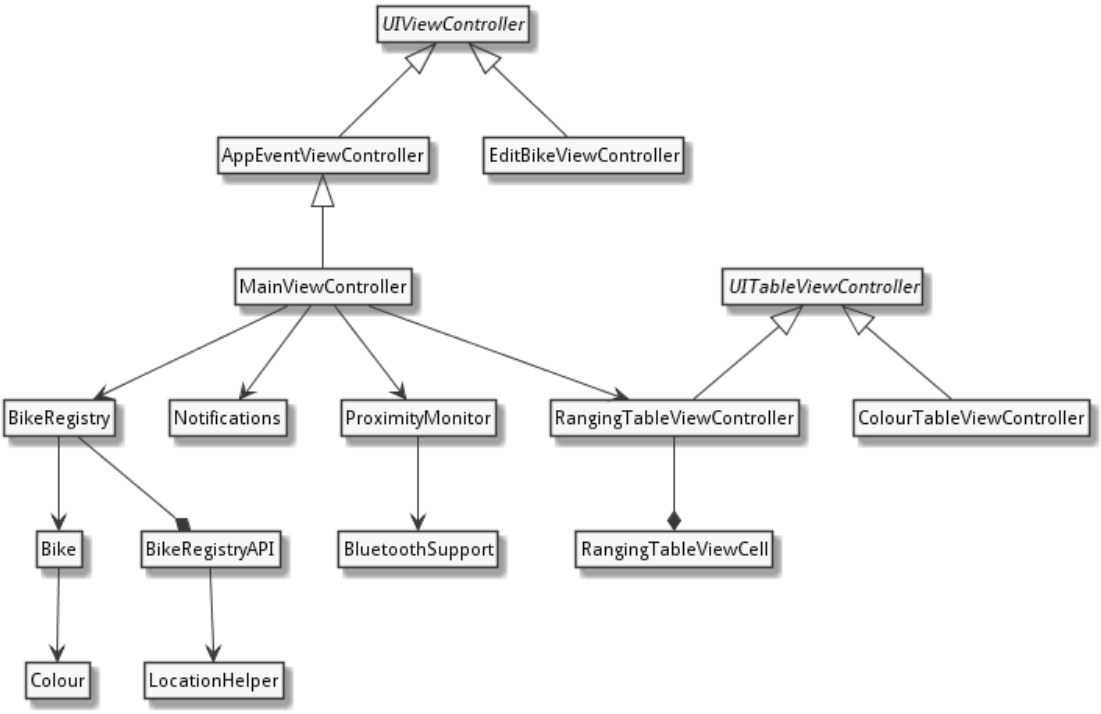

Figure 2: Partial class architecture showing key classes used in app

2.4.1 `AppEventViewController`

This `UIViewController` subclass is not instantiated directly. Instead it provides a generic base class for view controllers that wish to receive notifications about app lifecycle events that are normally only received by the `AppDelegate` object such as `applicationDidBecomeActive`. It registers to receive these events from a `NotificationCenter` instance, a form of the observer pattern (Gamma et al., 1994). This allows subclasses to provide controller-specific behaviour in response to them without requiring any direct coupling between them and the `AppDelegate` instance. As an example of this, `MainViewController` uses the capability to activate and deactivate its `ProximityMonitor` instance as the application moves between foreground and background states.

2.4.2 `MainViewController`

This controller acts as the primary view controller for the application, acting as the central point of coordination where state is stored (through a reference to an instance of `BikeRegistry`) and notifications from application services (such as `ProximityMonitor`) are handled. The storyboard scene for the `MainViewController` contains a vertical stack view that contains a `RangingTableViewController` within a Container View. The original thinking behind this approach was to allow the `MainViewController` to "combine the content from multiple view controllers into a single user interface" (Apple, 2017e). In this approach there is a parent-child relationship between each controller and the containing controller.

During development it became apparent that most of the app's interface could be provided via a single controller, `RangingViewController`. However, even with a single child controller, there are advantages to maintaining this separation, as the `UITableViewController` implementation methods (e.g. `numberOfSections`) are kept separate from other concerns such as data management and proximity detection. Therefore the arrangement was retained to illustrate this approach.

2.4.3 'ColourTableViewController"

This is a simple sublclass of `UITableViewController` that displays a list of colours from a predefined list and allow users to select one. This provides a view controller, a data source and a delegate in a single class (Keur and Hillegass, 2015). The controller is presented modally from the *Edit Bike* scene by the user tapping on a text label.

The original UIKit-provided solution for allowing the user to select items from a list (iOS 2.0 and later) is `UIPickerView`, which "uses a spinning-wheel or slot-machine metaphor to show one or more sets of values." (Apple, 2017f). However it seems that the 'house style' of many Apple apps is increasingly to use a `UITableView` in place of a `UIPickerView` where only a single value is being selected (see for example, the 'Repeat' option in the *New Event* scene in the Calendar app, or the 'Auto-Lock' option under Display & Brigtness in the Settings app, Figure 3).

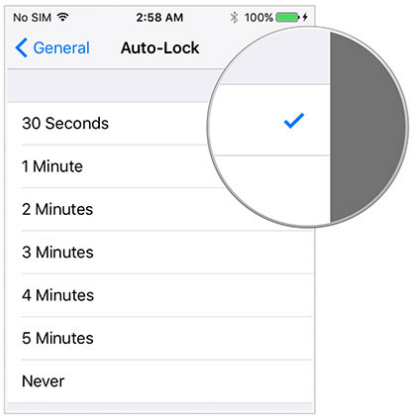

Figure 3: Auto-Lock option selection in Settings app on iOS 9

The current controller contains logic for handling item selection (using a 'checkmark' cell accessory) and passing data back to where it was presented from. As an improvement to this approach, controller could be refactored into a general-purpose 'single item picker' controller which could be used as a drop-in replacement for `UIPickerView` with only the data source needing to be customised. This would make it easy to create consistent selection interfaces and promote code reuse, following the object-oriented design goal of finding a design that is "specific to the problem at hand but also general enough to address future problems and requirements" (Gamma et al., 1994).

2.5 Development issues

2.5.1 Delegation

The classes provided by the Cocoa frameworks make extensive use of the *delegation* pattern "in which one object in a program acts on behalf of, or in coordination with, another object" (Apple, 2015). This is in contrast to a model where behaviour is inherited (and may be overridden) from class to subclass. Inheritance in object-oriented programming creates a tight coupling between a parent class and its subclass and breaks encapsulation, whereas composition enforces a 'black box' approach based on well-defined interfaces, and is therefore usually preferred (Gamma et al., 1994).

Although delegation allows complex behaviours to be composed dynamically at run time, one price (as Gamma et. al highlight, p.21) is that it make code harder to understand. This is evidenced in the app in the relationship between `MainViewController`, `RangingTableViewController` and `EditBikeController`. Since these classes only maintain a weak reference to an instance of the user's bike, for it to be persisted it must be passed "up" through two levels of composition to the `BikeRegistry` class, via the `BikeChangedDelegate` protocol. If the app evolved to include a large number of additional controllers this approach could become excessively complex.

2.5.2 Data updates

Currently *FindMyBike* requests data about missing bikes from the server only when it starts. As well as meaning that if the user is not online when they first run the app no data will be downloaded, the problem is that if a new bike is reported missing after the app has been started, other users will not detect it as missing till they restart their app. A better solution would be to download data updates periodically while the app is running, either using a `Timer` instance or scheduled background data refresh through the `performFetchWithCompletionHandler` method.

2.5.3 Prototypes

The proposal described that the app development process would use the concept of a 'production prototype' (Wysocki, 2014) both to learn the characteristics of the iOS environment and to test out specific areas of functionality. In fact a series of prototypes were developed:

2.5.3.1 BeaconRanger

The first production prototype app developed was called *BeaconRanger* and was completed around week 5 of the development timeline given in the proposal. It had no bike-related functionality or network capabilities but was focused on ranging iBeacons when the app was in the foreground (see Figure 4). Parts of its purpose was to iron out the relatively complex state transitions that occur while requesting location access permission, and to safely handle the user later rescinding that permission and re-granting it. The advantage of this approach was that it helped to be able to test and debug this discrete aspect of functionality behaviour in isolation.

Development on *BeaconRanger* ceased at this point, as it was never intended for release. Much of the code that was developed for this prototype was reused as the `ProxmityManager` and `BluetoothSupport` classes in *FindMyBike*.

Figure 4: Storyboard scene from BeaconRanger production prototype

2.5.3.2 AboutMyBike

The next production prototype that was built was called `AboutMyBike`. It did not include any iBeacon monitoring capabilities or server interactions but focused on the user interface, providing a simple form to allow a user to store useful information about their bike including make, model and engine number, along with a photo, as seen in Figure 5.

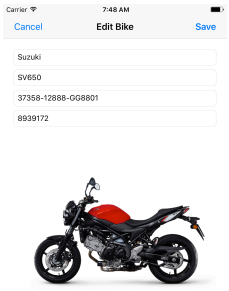

Figure 5: Edit Bike scene from AboutMyBike production prototype

In developing this prototype, one of the objectives was to submit a fully-functional app that complied with App Store guidelines and enable the developer to gain early experience of the App Store review process which would streamline the eventual submission for *FindMyBike*.

AboutMyBike was submitted for App Store review but was rejected after approximately 48h with the following message:

> Guideline 4.2 - Design - Minimum Functionality We found that the usefulness of your app is limited by the minimal amount of content or features it includes. Specifically, your app only includes a singular bike entry.

2.5.3.3 App Store description

[INSERT] + privacy?

2.6 Distribution

The app was submitted to the App Store with the following description:

[add description]

It was rejected.

2.6.1 Marketing

2.6.1.1 Network effects

There are two, slightly different features offered by *FindMyBike*:

- it allows a user to detect their proximity to their own bike, and therefore may allow them to locate it (for example, if they have forgotten where they parked it).
- it allows users to know if their missing bike has been detected by other users of the app, and therefore possibly recover it in the event of theft

The first feature above does not depend on any other app users. However, the effectiveness of the second feature is dependent on the number of users with the app installed that frequent the area of the missing bike. This feature is likely to be more useful in densely-populated areas (if a missing bike is in a desert, it is unlikely that many app users will enter its beacon region). But even if a large city, the usefulness of this feature is in proportion to the number of active app users. Each new user that installs the app in a densely-populated area increases the value of the app for other users, a phenomenon known as a 'positive network externality'(Shapiro and Varian, 1999), or network effect, which was first observed in communication technologies.

Aquiring enough users for this network effect to apply can be seen as a 'chicken-and-egg' problem. One way of resolving it is to ensure that the 'single-user' benefits are compelling enough if there are no other users of the network (Choudary, 2014). By marketing the app's 'single-user' benefits first, it is hoped that enough users will start to use it that the second use becomes viable and eventually becomes more compelling than the 'single-user' mode.

With this in mind, a plan for developing the app further should include building additional features that support the 'single-user' mode, for example:

- Allowing the user to store other useful details about their bike, such as engine number or vehicle identification number (VIN), which may be useful in the event of reporting a theft to the police
- Store details of the user's breakdown recovery service contact number and membership number (AA, RAC, etc.)
- Allow the user to record important dates relating to their bike, such as MOT date or insurance expiry date, which could trigger a local notification when the date approaches.

2.6.1.2 Promotion

Along with conventional types of promotion (such as purchasing advertising such as Google AdWords, or obtaining a paid-for promotional App Store placement, word-of-mouth is likely to be the most effective way of increasing the user base for *FindMyBike*. One way that apps achieve this is through authentic success stories. Tinder used this strategy effectively by staging stunts on college campuses where students claimed to have 'matched' with their professors [CITE] in order to drive downloads. An equivalent for *FindMyBike* might be to create an artificial scenario where someone locates a missing bike using the app, and then use social media to promote the story.

Although the app itself is free, there is a (small) cost associated with buying an iBeacon (and replacing its battery from time to time). In order to remove barriers to adoption, another approach would be to give away a number of free preconfigured iBeacons. Assuming 1,000 beacons could be bought, configured and distributed to users in central London at a total cost of £10,000, this investment should be sufficient to produce a viable initial user base for network effects to begin to apply. This giveaway could be funded either via direct investment or in partnership with a motorcyling brand, shop or publication.

One targeted way to increase uptake would be to encourage owners of venues which receive a large number of riders as customers (such as biker cafes, garages and petrol stations) to have *FindMyBike* installed on an iPhone at the venue. This would mean that if a stolen bike is taken to the garage/cafe in question it could be easily detected and the owner notified. Although undoubtedly an effective way of increasing bike recovery rates, the challenge lies in ensuring that the venue staff regularly see notifications delivered to their iPhone and are sufficiently incentivised to report the sightings.

3 Reflections and suggestions for further work

3.1 App Store review

opinion

3.2 Choice of development language

The Swift language (Apple, 2014c) was chosen to implement *FindMyBike* instead of Objecive-C. There were a number of factors in this decision:

- the app was built by a single developer with no existing experience with either Objective-C or Swift and therefore using either one would entail learning a completely new language
- the app was being created from scratch and therefore there was no need to maintain interoperability or consistency with existing legacy code as in some corporate environments (such as the BBC)
- Swift's type safety and
- the app idea did not entail using any third-party libraries which are only usable within Objective-C projects
- usage of Swift is increasing and is expected to equal levels of Objective-C development in the next four years (Hillyer, 2016)

Despite this, it was noticeable that in some respects the developer experience for Swift lags behind Objective-C. For example, some powerful Xcode features such as automatic refactoring of code are still only available for Objective-C projects. Also the documentation for some application services (such

as the new 'unified logging system' in Foundation (Apple, 2017a)) is more oriented towards Objective-C and does not provide any detailed Swift examples. This situation is likely to improve over time as Apple's focus moves increasingly from Objective-C to Swift.

3.3 Privacy and ethical implications

Although in its current form *FindMyBike* does not collect any information that identifies an individual, some users may be reluctant to use an app that requires 'always' access to Location Services on the basis that it appears to constantly track their position, which has privacy implications. Apple's decision to implement iBeacon support as part of Location Services means that the developer of such an app needs to educate users that although it constantly monitors beacon events, it only requests and transmits the user's absolute geographical location when strictly required (to report a missing bike's location).

3.4 Wearable implementation

One way of improving the usefulness of the app would be to implement watchOS support. Given the intended users of the app are motorcycle and scooter riders, who may not notice when their phone receives a notitication, a wearable extension of the app would increase the chance of them being aware of the notification and therefore reporting missing bikes. The simplest way to achieve this would be use the Watch Connectivity Framework to send missing bike detection notifications from the iPhone to the watch.

4 Appendix A: back-end server

As mentioned in the proposal, it is essential for the app to deliver its intended functionality for there to be a server component which centrally stores data about users' bikes, identifies those which are missing and records the geographical location where any missing bikes are detected. Another important role of the server is to send push notifications to the owner's device when their bike is found.

4.1 API implementation

A simple API was implementedbased on deployd, an open source Node.js framework for creating RESTful APIs [cite]. Two significant changes were made to the basic create/retrieve/update/delete (CRUD) functionality provided by deployd:

- a check was added before all requests to ensure that a valid API key was supplied, to protect the API against unauthorised use
- a custom event handler script was added to send a push notification to the correct APNs server when a new sighting is recorded, using the apn module.

The full source code of the API is attached with this report for reference.

4.2 Data storage

The data for the app is stored in a simple MongoDB instance with two collections, `bikes` and `sightings`. MongoDB is a schema-less database [cite].

[add schema]

4.3 Hosting

The API is hosted using Heroku and the MongoDB instances using MLab, both of which offer free hosting for low-volume applications.

4.4 Environments

Apple Push Notifications (APNs) supports two environments, development (aka sandbox) and production. The device token obtained by an app when it registers for notifications is specific to which APNs environment is being used by the app, which is determined by the `aps-environment` value in the entitlements file. Processes wishing to deliver push notifications determine which APNs server to use dependent on the environment the device token was obtained for (Apple, 2017b).

For this reason it is necessary to provide multiple back-end multiple environments, each with a corresponding API server and database.

[add diagram]

References

Apple (2017a) *About the Logging System*, Available at: https://developer.apple.com/documentation/os/logging (Accessed: 7 October 2017).

Apple (2014a) *App Store Review Guidelines*, Available at: https://developer.apple.com/app-store/review/guidelines/ (Accessed: 5 October 2017).

Apple (2015) *Cocoa Core Competencies: Delegation*, Available at: https://developer.apple.com/library/content/documentation/General/Conceptual/DevPedia-CocoaCore/Delegation.html (Accessed: 22 September 2017).

Apple (2017b) *Communicating with APNs*, Available at: https://developer.apple.com/library/content/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/CommunicatingwithAPNs.html (Accessed: 7 October 2017).

Apple (2014b) *Getting Started with iBeacon*, Available at: https://developer.apple.com/ibeacon/Getting-Started-with-iBeacon.pdf (Accessed: 29 July 2017).

Apple (2017c) *Human Interface Guidelines: Custom Keyboards*, Available at: https://developer.apple.com/ios/human-interface-guidelines/extensions/custom-keyboards/ (Accessed: 3 October 2017).

Apple (2017d) *Human Interface Guidelines: iOS Design Themes*, Available at: https://developer.apple.com/ios/human-interface-guidelines/overview/themes/ (Accessed: 22 September 2017).

Apple (2017e) *Implementing a Container View Controller*, Available at: https://developer.apple.com/

library/content/featuredarticles/ViewControllerPGforiPhoneOS/ImplementingaContainerViewController.html (Accessed: 3 October 2017).

Apple (2016a) *Location and Maps Programming Guide: Region Monitoring and iBeacon*, Available at: https://developer.apple.com/library/content/documentation/UserExperience/Conceptual/LocationAwarenessPG/RegionMonitoring/RegionMonitoring.html (Accessed: 3 October 2017).

Apple (2016b) *Text Programming Guide for iOS: Using Text Kit to Draw and Manage Text*, Available at: https://developer.apple.com/library/content/documentation/StringsTextFonts/Conceptual/TextAndWebiPhoneOS/CustomTextProcessing/CustomTextProcessing.html#//apple_ref/doc/uid/TP40009542-CH4-SW65 (Accessed: 22 September 2017).

Apple (2014c) *The Swift Programming Language*, Available at: https://itunes.apple.com/in/book/the-swift-programming-language-swift-3-1/id881256329 (Accessed: 7 October 2017).

Apple (2017f) *UIPickerView*, Available at: https://developer.apple.com/documentation/uikit/uipickerview (Accessed: 6 October 2017).

BeaconZone (2017) *Choosing UUID, Major, Minor and Eddystone-UID For Beacons*, Available at: https://www.beaconzone.co.uk/choosinguuidmajorminor (Accessed: 3 October 2017).

Choudary, S. P. (2014) *Building the Next WhatsApp or Instagram: The Network Effect Playbook*, Available at: https://www.wired.com/insights/2014/03/building-next-whatsapp-instagram-network-effect-playbook/ (Accessed: 22 September 2017).

Drobnik, O. (2015) *Barcodes with iOS: Bringing together the digital and physical worlds*, Shelter Island, NY, Manning Publications.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1994) *Design Patterns: Elements of Reusable Object-Oriented Software*, Upper Saddle River, NJ, Addison-Wesley.

Hillyer, N. (2016) *The Swift Programming Language*, Available at: https://savvyapps.com/blog/ultimate-guide-choosing-objective-c-or-swift (Accessed: 7 October 2017).

Keur, C. and Hillegass, A. (2015) *iOS Programming: The Big Nerd Ranch Guide*, 5th Edition, Atlanta, GA, Big Nerd Ranch.

Norman, D. (2013) *The Design of Everyday Things*, Revised and expanded edition, New York, NY, Basic Books.

Shapiro, C. and Varian, H. R. (1999) *Information Rules: A Strategic Guide to the Network Economy*, Boston, MA, Harvard Business School Press.

Wysocki, R. K. (2014) *Effective Project Management: Traditional, Agile, Extreme*, Seventh edition, Indianaopolis, IN, John Wiley & Sons.