Capitolo 1

Architettura delle applicazioni per Apple Watch

Gli strumenti di sviluppo messi a disposizione da Apple consentono di sviluppare tutti i componenti supportati da Apple Watch: sguardi, notifiche, complicazioni e addirittura applicazioni complete che possono prendere posto al fianco delle applicazioni native che Apple include di default nell'orologio.
Per realizzare queste applicazioni, Apple mette a disposizione una serie di strumenti di sviluppo (raccolti nell'IDE Xcode, disponibile su App Store) e una serie di framework. Nella Tabella 1.1 sono riassunti tutti i framework supportati da watchOS 2.
Il framework principale che contiene gli elementi indispensabili per sviluppare applicazioni per Apple Watch è WatchKit. Questo contiene le classi che rappresentano i componenti visuali che è possibile utilizzare nelle interfacce utente, oltre a una serie di classi di supporto per la presentazione di fogli di azione o avviso e classi base per l'estensione e i controller.

In questo capitolo

- **Struttura delle applicazioni**
- **Ciclo di vita dell'applicazione**
- **Schermate e controllori**
- **Disegno dell'interfaccia**
- **Creazione di un'applicazione per Apple Watch**
- **Esecuzione dell'applicazione**
- **Debugging**
- **Hello World**
- **Aggiungere interattività**

Tabella 1.1 Framework supportati da watchOS 2.

Framework	Descrizione
ClockKit	Supporto alla creazione di complicazioni
Contacts	Accesso al database dei contatti locali e provenienti da servizi remoti
Core Data	Gestione di un grafo di oggetti e relativa persistenza
Core Foundation	Funzionalità di base a basso livello
Core Graphics	Primitive per la grafica in due dimensioni
Core Location	Accesso ai servizi di localizzazione e geocodifica diretta e inversa
Core Motion	Accesso ai sensori di movimento, come accelerometro, giroscopio e magnetometro
EventKit	Accesso al database dei promemoria ed eventi di calendario
Foundation	Classi di base
HealthKit	Fornisce alle applicazioni una struttura per condividere informazioni su fitness e salute
HomeKit	Comunicazione e controllo di accessori di domotica compatibili
Image I/O	Lettura e scrittura file immagine
MapKit	Classi relative alle mappe e alla localizzazione geografica
Mobile Core Services	Supporto per gli UTI (Uniform Type Identifier)
PassKit	Supporto per l'aggiunta e la rimozione di pass all'applicazione Wallet
Security	Funzionalità legate alla sicurezza, come l'accesso a KeyChain, la generazione di numeri casuali, la gestione dei certificati e delle chiavi crittografiche
Watch Connectivity	Comunicazione a due vie, in modo interattivo o in background, tra Apple Watch e iPhone
WatchKit	Classi per la manipolazione dei componenti visuali dell'interfaccia utente

Struttura delle applicazioni

Le applicazioni per Apple Watch sono contenute all'interno di un'applicazione iOS ospite. Non può esistere un'applicazione watchOS senza una controparte su iPhone. In un'applicazione per Apple Watch sono presenti più elementi: l'interfaccia utente (storyboard), il codice e le risorse come immagini o altri elementi multimediali. Il codice è contenuto, insieme ad altri elementi, in un componente chiamato estensione. In Figura 1.1 è illustrata la relazione tra questi elementi. Come si può notare, l'architettura è impostata con una logica di contenitori (anche detti *bundle*) annidati: l'applicazione iOS contiene l'applicazione watchOS che a sua volta contiene l'estensione. Come si può notare dalla figura, sia l'applicazione sia l'estensione watchOS possono contenere risorse: per accedere a questi elementi si dovranno utilizzare API differenti, che descriveremo in seguito. Inoltre, dallo storyboard si devono impiegare solo risorse presenti nel file di risorse dell'applicazione.

Figura 1.1 Architettura dei componenti di un'applicazione per Apple Watch.

NOTA

Nella versione 7.0 di Xcode, quando si costruisce l'interfaccia utente con Interface Builder, è possibile selezionare anche risorse presenti nel file di risorse dell'estensione, e in fase di progettazione l'immagine viene visualizzata correttamente. In fase di esecuzione, però, se la stessa immagine non è presente anche nelle risorse dell'applicazione, non viene mostrata. La possibilità di selezionare risorse non appartenenti all'applicazione potrebbe essere un bug di questa versione dell'ambiente di sviluppo di Apple.

Quando si installa un'applicazione sull'orologio, tramite l'applicazione Watch presente su iPhone, questa viene trasferita su Apple Watch. L'applicazione watchOS e l'estensione watchOS, al momento opportuno, saranno messe in esecuzione in due contenitori separati (Figura 1.2). Il fatto che questi due elementi siano mantenuti in contenitori separati impone alcune limitazioni alle modalità di interazione tra i due componenti. Una l'abbiamo già affrontata: è l'impossibilità di utilizzare risorse presenti nell'estensione in Interface Builder per configurare lo storyboard. In modo similare, da codice è necessario utilizzare API differenti per accedere alle immagini presenti nell'applicazione e nell'estensione. Altre limitazioni saranno descritte in seguito.

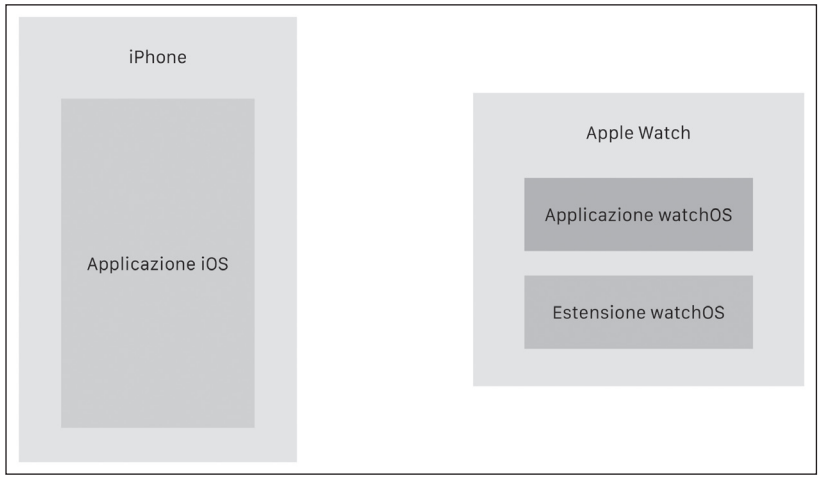

Figura 1.2 Una volta trasferita un'applicazione su Apple Watch siamo di fronte a tre processi distinti che possono interagire tra di loro: l'applicazione iOS, l'applicazione watchOS e l'estensione watchOS.

NOTA

Fino alla versione 1.x di watchOS, l'applicazione watchOS veniva trasferita su Apple Watch, mentre l'estensione girava su iPhone. A partire dalla versione 2 di watchOS, entrambi i componenti girano sull'orologio.

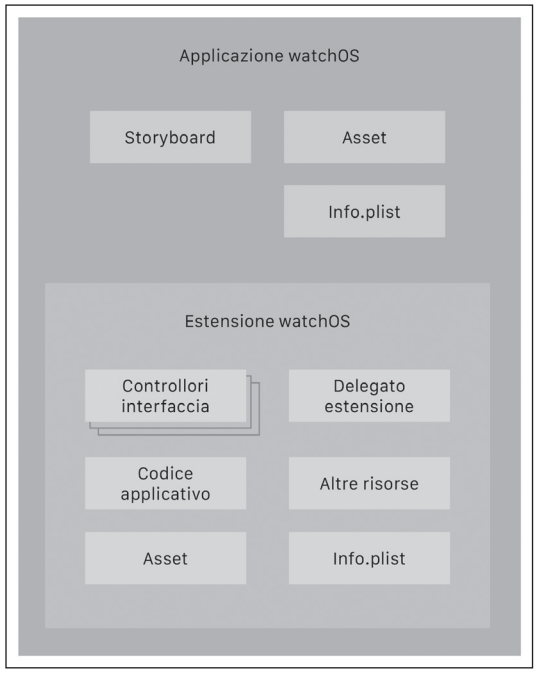

Figura 1.3 La struttura di una tipica applicazione watchOS.

In Figura 1.3 è presente in maggior dettaglio il contenuto tipico dell'applicazione e dell'estensione watchOS.
Come si può notare dall'immagine, in ciascun bundle sono presenti anche file di configurazione (`Info.plist`) che memorizzano le impostazioni per il relativo modulo.
In aggiunta a quanto già descritto, l'estensione contiene i seguenti elementi.

- *Controllori interfaccia*. Ogni schermata del progetto, sia essa relativa all'applicazione, a uno sguardo o ad una notifica, è rappresentata da una schermata nello storyboard e da una classe collegata, il controllore. Nel pattern MVC (Model-View-Controller, modello-vista-controllore), la schermata rappresenta la vista, mentre la classe a essa collegata svolge il ruolo di modello/controllore. Questa classe ospiterà i necessari outlet per accedere agli elementi dell'interfaccia e i metodi da invocare in risposta agli eventi generati dai componenti nella schermata, come per esempio il tocco di un pulsante.
- *Delegato estensione*. Questa classe implementa i metodi di callback a livello applicazione che consentono di ricevere ed eseguire codice applicativo a seguito del cambio dello stato dell'applicazione e in risposta a operazioni specifiche come le notifiche (Capitolo 6) o Handoff (vedi Capitolo 5). Il design pattern delegation prevede che un oggetto, invece che svolgere una o più attività in modo autonomo, deleghi l'attività a un altro oggetto di supporto. È la controparte watchOS del delegato `UIApplicationDelegate` che si trova nelle applicazioni iOS.
- *Codice applicativo*. Rappresenta l'insieme di classi che costituiscono l'applicazione, come per esempio classi di modello, di logica applicativa e classi di supporto.
- *Altre Risorse*. L'estensione può contenere risorse di vario tipo, come immagini da manipolare da codice, file audio, filmati, file testuali e binari (per esempio HTML, JSON e così via).

Modello MVC

Il modello MVC è un design pattern che suddivide le competenze degli elementi che concorrono a formare una interfaccia utente allo scopo di renderla più mantenibile. La vista si occupa di presentare le informazioni all'utente e gestire il feedback, mentre il modello mantiene le informazioni che sono rappresentate dalla vista. Il controllore è l'elemento che coordina vista e modello, passando le informazioni del modello alla vista per la sua rappresentazione e gestendo le interazioni dell'utente con la vista per aggiornare il modello.

Ciclo di vita dell'applicazione

Il delegato dell'estensione è in grado, tra le altre cose, di monitorare i cambiamenti di stato dell'applicazione. Questi rappresentano gli eventi principali del ciclo di vita dell'applicazione. I metodi di callback associati a questi eventi costituiscono l'occasione per eseguire attività necessarie all'avvio e alla chiusura dell'applicazione o per l'inizializzazione dell'interfaccia utente.

Il ciclo di vita dell'applicazione è illustrato in Figura 1.4 e si compone di tre stati.

- *Non in esecuzione*. L'applicazione non è stata lanciata o è stata terminata dal sistema.
- *Inattiva*. L'applicazione è attiva in primo piano, non sta ricevendo eventi ma potrebbe star eseguendo del codice. È uno stato intermedio da cui si passa appena dopo l'avvio dell'applicazione o poco prima della sua chiusura.
- *Attiva*. L'applicazione è attiva e in primo piano e sta ricevendo eventi. È la modalità normale in cui l'applicazione sta interagendo con l'utente.

Il delegato dell'estensione è creato in modo automatico dal sistema utilizzando la classe configurata nel file `Info.plist` nella chiave `WKExtensionDelegateClassName` e implementa il protocollo `WKExtensionDelegate`.

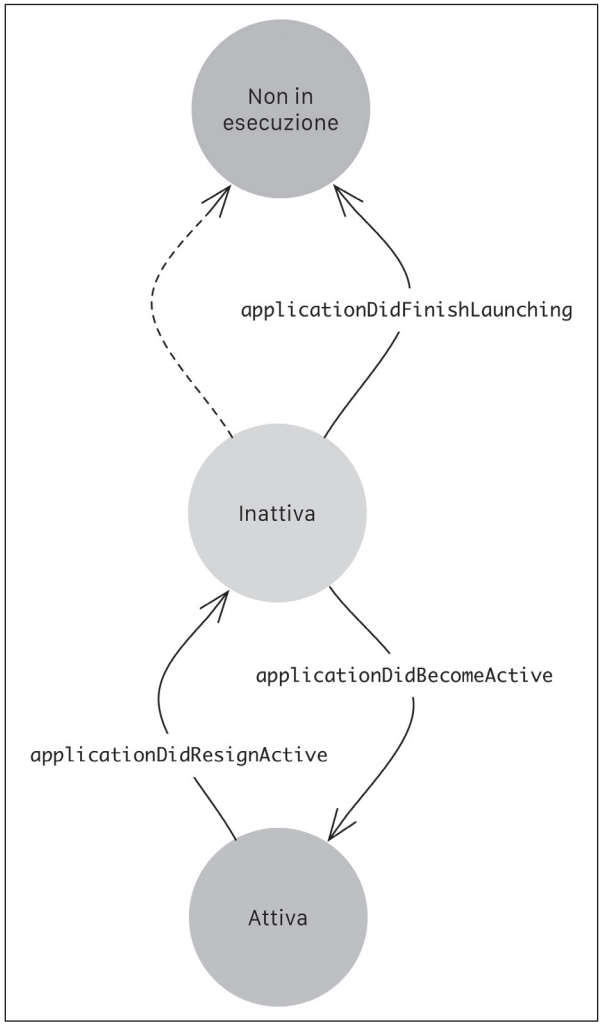

Figura 1.4 Diagramma degli stati di un'applicazione watchOS.

I metodi di callback legati ai cambi di stato sono i seguenti (non si aspettano parametri):

- `applicationDidFinishLaunching` viene chiamato una volta sola all'avvio dell'applicazione e offre l'occasione per eseguire le inizializzazioni necessarie a livello di tutta l'applicazione (non legate a una schermata in particolare). Per esempio, in questo metodo potrebbero essere inizializzati osservatori di notifiche, sessioni Watch Connectivity (Capitolo 3) o connessioni a servizi esterni. L'applicazione, in questo momento, non è ancora attiva;
- `applicationDidBecomeActive` viene invocato dopo `applicationDidFinishLaunching`. Questo metodo viene richiamato anche ai successivi accessi all'applicazione dopo il primo, ovvero quando questa viene visualizzata. In questo momento è possibile attivare eventuali timer e soprattutto inizializzare correttamente lo stato dell'applicazione.
- `applicationWillResignActive` viene invocato poco prima che l'applicazione venga messa in background. È il momento adatto per predisporsi alla successiva inattività dell'applicazione, salvando lo stato della stessa, disabilitando servizi, timer ecc. Le applicazioni in background possono essere successivamente rimosse dalla memoria per liberare risorse per altre attività.

NOTA

L'affermazione che il ciclo di vita dell'applicazione sia gestito tramite il delegato dell'estensione potrebbe confondere in merito ai termini utilizzati. Ci si aspetterebbe di avere un delegato applicativo, oppure un ciclo di vita dell'estensione. Questa apparente incongruenza terminologica è legata al fatto che per applicazione watchOS si può intendere sia il tutto, ovvero storyboard ed estensione, sia solo lo storyboard (con relativi asset). Questo è legato all'architettura di watchOS e al fatto che è necessario differenziare tra la fase di progettazione e la fase di esecuzione. Infatti, in fase di progettazione si crea un unico binario (composto comunque da più bundle separati, ovvero applicazione iOS, applicazione watchOS ed estensione watchOS), mantenendo comunque i sorgenti in cartelle separate, mentre in esecuzione questi tre elementi vengono eseguiti in contenitori differenti. Per chiarire meglio il concetto, si può pensare di approssimare il concetto di contenitore in processo.

NOTA

Il delegato dell'estensione include anche metodi di callback per supportare la tecnologia Handoff e per rispondere a notifiche locali e remote. Questi saranno discussi rispettivamente nei capitoli 5 e 6.

È possibile accedere da codice al delegato dell'estensione tramite la proprietà `delegate` della classe `WKExtension`, che rappresenta appunto l'estensione. Questo oggetto singleton si ottiene tramite il seguente metodo di `WKExtension`:

```
class func sharedExtension() -> WKExtension
```

Oltre a consentire l'accesso al delegato dell'estensione, la classe `WKExtension` consente anche di iniziare una conversazione telefonica o la composizione di un sms tramite il metodo `openSystemURL:`, che si aspetta un URL con schema `tel` o `sms`. Inoltre, la proprietà `rootInterfaceController` permette di accedere al controller principale dell'applicazione, utile particolarmente per implementare il supporto a Handoff.

Schermate e controllori

Come abbiamo già spiegato, ogni schermata dell'applicazione è implementata tramite un componente nello storyboard e una classe controllore associata alla schermata. Quando l'applicazione entra nello stato attiva, viene inizializzata e presentata la prima schermata del programma e il relativo controllore. Tutti i controllori derivano dalla classe `WKInterfaceController`. Questa classe implementa un numero significativo di metodi per supportare una miriade di funzionalità, come il ciclo di vita della schermata, la navigazione gerarchica e a pagine, l'input di testo, il supporto multimediale, la gestione dei menu, Handoff e molto altro. Avremo modo di approfondire gli aspetti più importanti di questa classe nel corso del libro. Per ora limitiamoci a introdurre gli elementi essenziali legati al ciclo di vita della schermata. In Figura 1.5 sono riassunti i metodi di callback richiamati dal sistema durante il ciclo di vita della schermata.

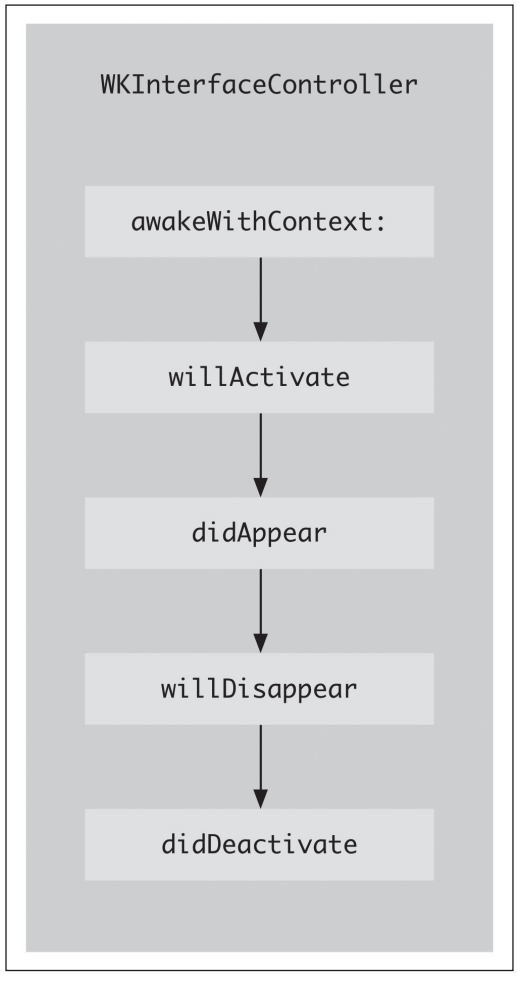

Figura 1.5 Metodi di callback del ciclo di vita di un controllore.

I metodi di callback supportati da `WKInterfaceController` sono i seguenti:

- `awakeWithContext(_ context: AnyObject?)`. Questo metodo viene invocato dal sistema in fase di inizializzazione della schermata, dopo il caricamento dell'interfaccia dallo storyboard. Alla chiamata di questo metodo gli eventuali outlet presenti nella classe sono già configurati per puntare ai rispettivi oggetti nello storyboard. Si dovrebbe sfruttare questa chiamata per completare l'inizializzazione dell'interfaccia, per esempio per caricare le celle di una tabella, caricare immagini o popolare del testo. Il parametro `context` contiene le informazioni di contesto necessarie al controllore per inizializzare correttamente la schermata ed è passato dal controllore precedente. Nel caso del controller iniziale dell'applicazione, `context` è `nil`. Questo meccanismo rappresenta un modo elegante e standard per passare informazioni di contesto da un controllore all'altro. L'oggetto di contesto consente di fare in modo che il controllore di destinazione sia a conoscenza dell'informazione che deve trattare. Si pensi per esempio a un'applicazione di gestione di note di testo composta da una prima schermata dove è presente una tabella con l'elenco delle note; toccando una riga si accede alla schermata di dettaglio che mostra la nota interamente. Il controllore di visualizzazione del dettaglio potrebbe sapere quale nota visualizzare perché, al tocco su una voce dell'elenco, il controllore di origine passa come oggetto di contesto l'oggetto di dominio che rappresenta la nota selezionata;
- `willActivate` viene invocato per segnalare che si sta predisponendo a visualizzare l'interfaccia di questa schermata. La ricezione di questa chiamata non è però garanzia del fatto che l'interfaccia sia visualizzata o che lo sia a breve. Il sistema potrebbe infatti eseguire questa chiamata con un certo anticipo, per dare modo di popolare il contenuto della stessa; per esempio, il sistema potrebbe chiamare questo metodo su un controllore di uno sguardo per minimizzare il ritardo di aggiornamento dei contenuti dello sguardo a video. Si può utilizzare questo metodo per eseguire le attività dell'ultimo minuto per assicurarsi che il contenuto della schermata sia aggiornato, ma non per la configurazione iniziale della stessa, che dovrebbe essere già stata eseguita nel metodo `awakeWithContext:`;
- `didAppear` viene invocato pochi istanti dopo la visualizzazione del controllore sullo schermo, per segnalare che questo è ora visibile. In questo metodo è possibile configurare le animazioni e altri elementi legati all'interfaccia utente;
- `willDisappear` viene richiamato pochi istanti prima della rimozione del controllore dallo schermo, per segnalare che presto non sarà più visibile. In questo metodo è necessario interrompere animazioni in corso o altre attività legate all'interfaccia utente;
- `didDeactivate` viene invocato per segnalare che il controllore di interfaccia non è più attivo. È possibile utilizzare questo metodo per invalidare timer o salvare dati applicativi non già salvati. A ogni modo, ogni attività dovrebbe completarsi nel minor tempo possibile. Un controllore inattivo potrebbe essere riattivato in seguito o rimosso dalla memoria. Non è possibile eseguire modifiche all'interfaccia da questo metodo, perché verranno ignorate; eventuali attività conclusive sull'interfaccia dovrebbero essere eseguite nel metodo `willDisappear`.

I metodi `willActivate`, `didAppear`, `willDisappear`, `didDeactivate` vengono invocati sul thread principale. L'implementazione di questi metodi e di `awakeWithContext:` della classe `WKInterfaceController` non contiene nulla.

SUGGERIMENTO

È possibile utilizzare i metodi `willActivate` e `didDeactivate` per implementare il ripristino dello stato della schermata (State Restoration) attraverso differenti utilizzi dell'applicazione.

Dato che i metodi `willActivate`, `didAppear`, `willDisappear`, `didDeactivate` e `awakeWithContext` di `WKInterfaceController` non contengono implementazione, non è indispensabile chiamare la relativa implementazione nella superclasse dalla sottoclasse, come per esempio nel codice che segue:

```
override func awakeWithContext(context: AnyObject?) {
    super.awakeWithContext(context)
}
```

Nella pratica, però, i template di Xcode includono nel codice generato proprio questo schema di codifica, che è chiamato *concatenamento delle chiamate*. È buona norma seguire l'esempio proposto da Xcode per due motivi: il primo è che in una futura versione di watchOS la classe `WKInterfaceController` potrebbe eseguire delle operazioni di qualche tipo, richiedendo l'invocazione del corrispettivo metodo nella superclasse. Includendo già questa chiamata, il codice è maggiormente a prova di modifiche future (*future proof*). In secondo luogo, dato che un controllore potrebbe non derivare direttamente da `WKInterfaceController` ma da una qualche classe intermedia che potrebbe svolgere delle operazioni proprio in questi metodi, sarebbe necessario verificare se ciò accade e inserire la relativa chiamata. Il fatto è che questa è un'operazione che può facilmente indurre in errore, in quanto per semplice distrazione si potrebbe non notare che un determinato controllore non deriva direttamente da `WKInterfaceController` ma da un'altra classe. Inoltre, bisogna considerare che il codice sorgente o la documentazione esatta della classe intermedia potrebbe non essere disponibile, per esempio perché parte di un framework di terze parti. Per questi motivi è meglio utilizzare come approccio comune il fatto di richiamare sempre, all'interno di questi metodi di callback, la relativa implementazione nella superclasse, in modo da non spezzare l'eventuale catena di chiamate.

NOTA

Nel simulatore, i metodi `willActivate` e `didDeactivate` vengono invocati quando si blocca il simulatore con la funzione *Hardware > Lock*. Utilizzando questa caratteristica è possibile testare il funzionamento del codice di attivazione e disattivazione delle proprie schermate.

Disegno dell'interfaccia

Una sostanziale differenza tra lo sviluppo per iOS e per Apple Watch è la maggior semplicità delle interfacce e la conseguente maggior semplicità di sviluppo. watchOS non supporta Auto Layout, la tecnologia Apple che su iOS consente di definire in modo dichiarativo le logiche di riposizionamento e ridimensionamento.
In WatchKit le interfacce sono disegnate all'interno di uno storyboard (Figura 1.6): non è infatti possibile instanziare oggetti visuali da codice. Gli oggetti di interfaccia vengono poi collegati al codice come avviene per lo sviluppo iOS. In questo modo, dal codice è possibile impostare proprietà sull'interfaccia, come per esempio il contenuto di una etichetta, e ricevere eventi dal sistema, come per esempio il tocco di un pulsante.

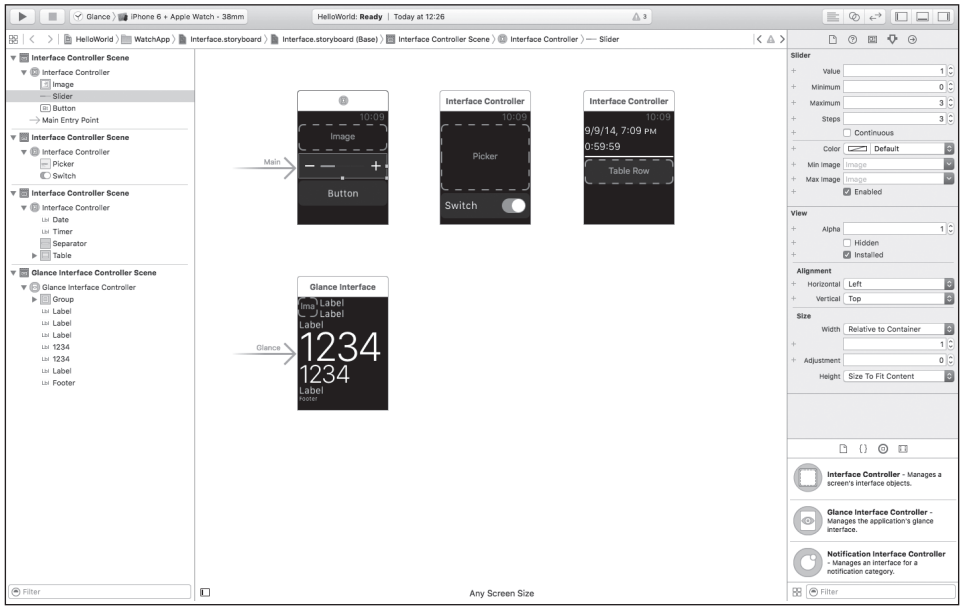

Figura 1.6 Interface Builder è lo strumento di Xcode per progettare le interfacce utente.

Come in iOS, l'interazione dell'interfaccia utente con il codice avviene tramite due meccanismi distinti: ogni componente visuale che deve essere manipolato da codice deve essere rappresentato nel codice del controller da una variabile del tipo corretto (per esempio, la variabile che punterà a un oggetto immagine sarà di tipo `WKInterfaceImage`) e marcata dall'annotazione `@IBOutlet`. Questa dovrà essere collegata in Interface Builder al componente relativo in due modi: trascinando dal codice il pulsante +, presente a fianco della riga di codice che dichiara la variabile, verso il componente da collegare o all'inverso (in questo secondo caso, oltre a trascinare il componente sarà necessario tenere premuto il pulsante Control, si veda la Figura 1.7).

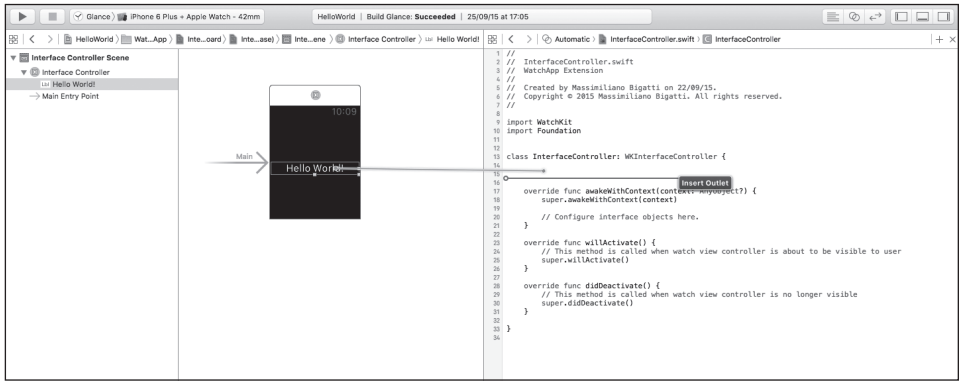

Figura 1.7 Per creare un outlet nel codice è sufficiente trascinare dal componente visuale da collegare verso il controller della schermata, tenendo premuto il tasto Control.

In questo secondo caso Interface Builder presenta un pannello che consente di specificare il nome dell'outlet. Il tipo dato è determinato automaticamente (Figura 1.8).

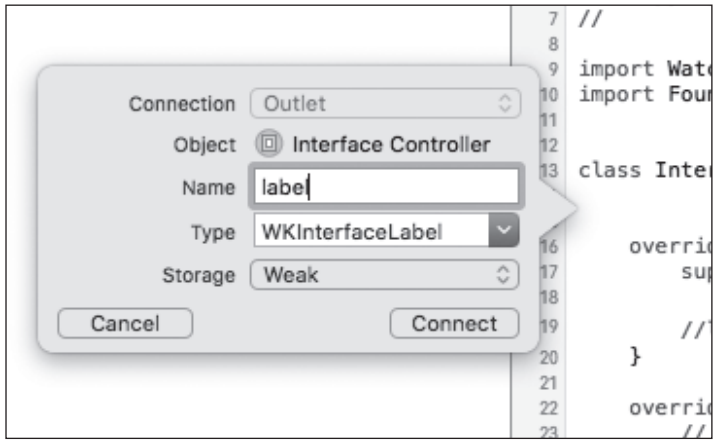

Figura 1.8 Pannello per la creazione di un outlet di collegamento tra interfaccia e codice.

L'altro meccanismo che consente di realizzare l'interazione tra interfaccia utente e codice è il meccanismo *target/action*, dove l'interazione con un componente visuale (per esempio il tocco di un pulsante) si traduce con un'azione, o messaggio, inviata verso uno specifico oggetto (target). I framework di Apple fanno ampio uso di questo pattern, non solo in relazione all'interfaccia utente.
Per fare in modo che l'interazione con un componente produca l'invocazione di un metodo del controller è necessario creare un metodo nel controller marcato con l'annotazione `@IBAction`. La firma esatta del metodo dipende dall'azione alla quale sarà collegato, che varia in funzione del tipo di oggetto collegato. Nel Capitolo 2 sono indicate le firme necessarie per tutti i componenti visuali supportati dal sistema.

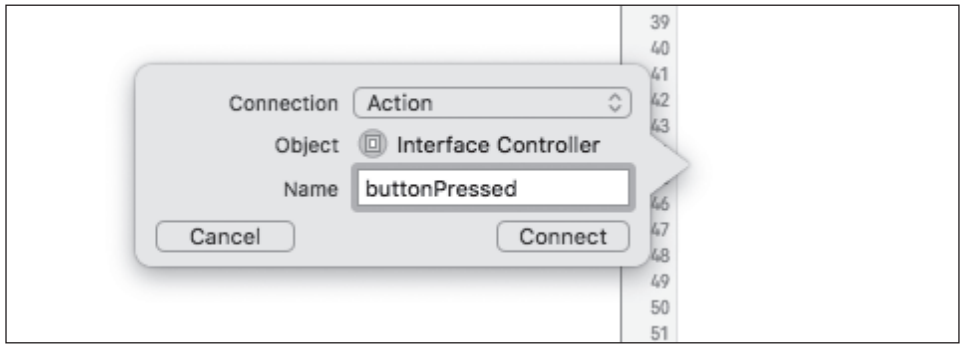

Figura 1.9 La creazione di un'azione collegata a un componente visuale si ottiene tramite lo stesso pannello impiegato con gli outlet, dove si dovrà selezionare la voce Action nel campo Connection.

È inoltre possibile generare il metodo con Interface Builder. Per fare questo si procede come per la creazione di un outlet, con la differenza che nel pannello delle opzioni si dovrà selezionare *Action* nel campo *Connection*. Per completare l'operazione sarà poi necessario specificare il nome del metodo da creare.
In realtà, gli oggetti visuali messi a disposizione da WatchKit non sono i veri e propri oggetti che il sistema usa per il rendering dell'oggetto e per l'intercettazione di possibili eventi: in realtà sono solo *proxy* dei reali oggetti visuali. Con WatchKit, infatti, non è possibile accedere direttamente agli oggetti responsabili della visualizzazione dei componenti visuali. In altre parole, le classi di WatchKit non sono inserite nel ciclo di rendering del sistema. Questo limita fortemente ciò che è possibile sviluppare in termini di personalizzazione: in primo luogo, fra proprietà/metodi, gli unici accessibili sono quelli esposti dall'oggetto proxy, dunque non sarà possibile accedere a ulteriori elementi, anche usando API private; in secondo luogo, non è possibile derivare questi oggetti per ridefinirne l'aspetto o il comportamento, in quanto non dispongono dei metodi di disegno (come invece succede per gli oggetti di UIKit derivati da UIView). Infine, i proxy sono *di sola lettura*: una volta impostati i valori delle proprietà da codice, non è possibile accedere in lettura. In caso si renda necessario conoscere il valore di una di queste proprietà, come per esempio il testo di una etichetta, si dovrà mantenere questa informazione in un'altra parte dell'applicazione, in una variabile apposita.
Tutti gli oggetti di interfaccia derivano dalla classe `WKInterfaceObject`.

NOTA

È interessante notare inoltre che i parametri di configurazione dei singoli componenti, manipolabili nello storyboard tramite Interface Builder, non trovano precisa corrispondenza con i metodi esposti dagli oggetti proxy. Infatti, le proprietà manipolabili da codice e quindi modificabili a runtime sono solo una frazione degli attributi presenti nei singoli controlli. I dettagli di cosa è possibile impostare in fase di sviluppo e cosa è modificabile a runtime sono descritti nel Capitolo 4.

In watchOS il layout dei componenti visuali è affidato all'oggetto Group (`WKInterfaceGroup`), che rappresenta un contenitore di componenti e si occupa di organizzarli in verticale o in orizzontale. Lo spazio tra i componenti e i margini interni di un gruppo possono essere personalizzati, come il colore di sfondo, il raggio, la trasparenza e altro. Inoltre, è interessante notare che ogni componente visuale, gruppi compresi, può essere allineato orizzontalmente (sinistra, centro, destra) e verticalmente (in alto, al centro, in basso). Infine, le dimensioni orizzontali e verticali di ogni componente visuale possono essere impostate come percentuale dello spazio disponibile nel contenitore, come valore assoluto o in modo dipendente dal contenuto.
Un gruppo è un componente visuale, quindi può essere contenuto all'interno di altri gruppi. Selezionando diverse opzioni per i diversi gruppi annidati è possibile ottenere layout anche complessi, per esempio alternando il layout verticale a quello orizzontale. Questo ricco insieme di caratteristiche offerto dal componente Group consente la realizzazione di layout anche discretamente avanzati, rendendo possibile ottenere quasi tutti i layout desiderati.

Creazione di un'applicazione per Apple Watch

Un'applicazione per Apple Watch è implementata come un target specifico all'interno dell'applicazione iOS ospitante. Per aggiungere questo target procedere come segue.

1. Selezionare il progetto facendo clic sull'elemento radice in *Project Navigator.*
2. Fare clic sul pulsante + (*Add a target*, Figura 1.10) oppure selezionare *Editor* > *Add Target*.

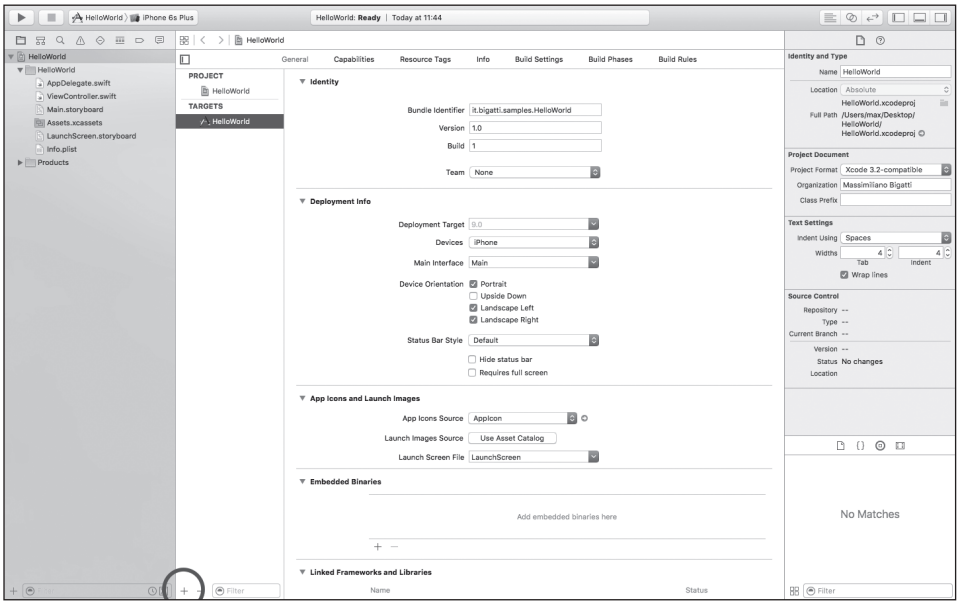

Figura 1.10 Per aggiungere l'applicazione per Apple Watch a un progetto esistente fare clic sul pulsante di aggiunta indicato in figura.

3. Selezionare *WatchKit App* dal gruppo *watchOS/Application* (Figura 1.11).

Figura 1.11 Nel foglio di selezione del template scegliere la voce WatchKit App nel gruppo Application nella sezione watchOS.

4. Fare clic sul pulsante *Next*.
5. Impostare il nome del prodotto e le altre opzioni, come il linguaggio di programmazione e se il progetto dovrà includere schermi di notifica, sguardi e complicazioni (Figura 1.12).

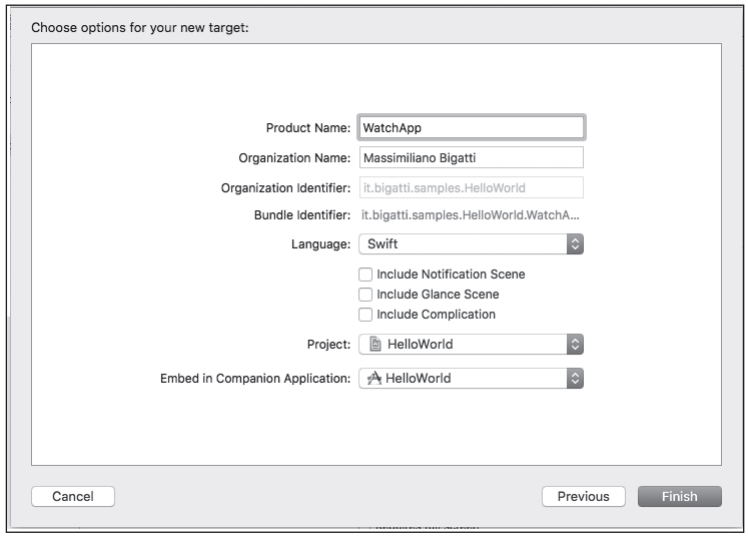

Figura 1.12 Nelle opzioni di creazione dell'applicazione è possibile specificare se creare schermate per notifiche e sguardi e la sorgente dati per una complicazione.

6. Fare clic su *Finish*. Il sistema chiederà all'utente la conferma per l'attivazione dello schema di compilazione e debug del nuovo target (Figura 1.13).

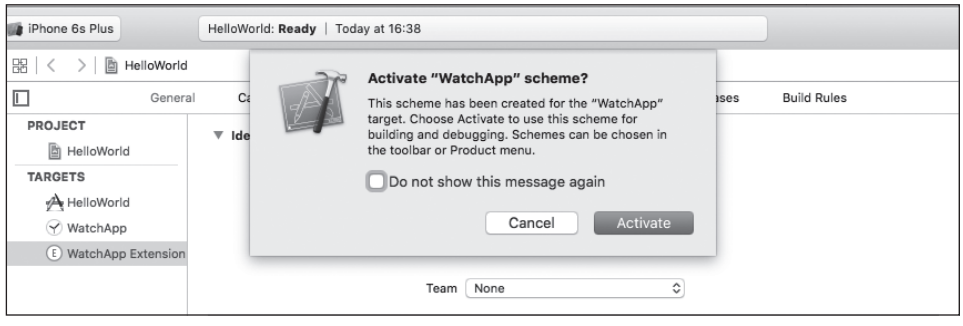

Figura 1.13 L'aggiunta di un target per le applicazioni WatchKit richiede l'attivazione di uno schema specifico di compilazione e debug, che Xcode può configurare in modo autonomo.

Quando si aggiunge un'estensione WatchKit App a un progetto esistente vengono create due cartelle distinte, una per l'applicazione e l'altra per l'estensione (Figura 1.14).

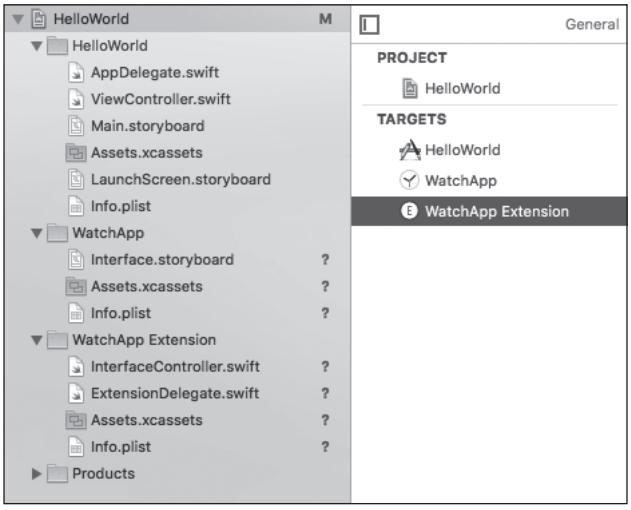

Figura 1.14 Struttura delle cartelle creata da Xcode per un'applicazione watchOS.

La prima cartella contiene lo storyboard che dovrà contenere tutte le interfacce supportate (applicazione, sguardo, notifiche), mentre nell'estensione dovrà essere inserito il codice sorgente che implementa la logica dell'applicazione. In entrambe le cartelle è presente anche un file di asset e di proprietà (plist).
Il modello di default che Xcode utilizza quando si chiede di aggiungere un'applicazione per Apple Watch a un'applicazione iOS crea nella cartella dell'estensione due file: `InterfaceController.swift` e `ExtensionController.swift`. Questi file contengono, rispettivamente, il controllore della prima schermata dell'applicazione e l'implementazione del delegato dell'estensione (`WKExtensionDelegate`). Lo storyboard presenta una sola schermata, più eventualmente una schermata per lo sguardo e per le notifiche, nel caso sia stata richiesta la loro creazione in fase di aggiunta dell'applicazione per Apple Watch.

Esecuzione dell'applicazione

È possibile eseguire l'applicazione sul dispositivo o sul simulatore. Nel primo caso è necessario che l'iPhone abbinato all'orologio sia connesso al computer tramite cavo USB/Lightning. Nel secondo caso è sufficiente lanciare l'applicazione perché il simulatore venga avviato automaticamente. Per scegliere tra dispositivo e simulatore si utilizza il menu in alto a sinistra nell'interfaccia di Xcode (Figura 1.15), dove è possibile selezionare anche il target da eseguire (nel nostro caso applicazione iOS o watchOS).
Il simulatore è un'applicazione che mostra l'applicazione all'interno di uno schermo Apple Watch simulato. Per quanto possibile, il simulatore cerca di mettere a disposizione dello sviluppatore tutti gli strumenti necessari a provare le applicazioni nel contesto più vicino possibile alla realtà. Il simulatore di Apple Watch consente di simulare lo scorrimento della corona digitale tramite il trackpad o il mouse, il click della corona digitale e il Force Touch.

Figura 1.15 Selezione della modalità di esecuzione dell'applicazione.

Purtroppo, il simulatore non è in grado di sostituire completamente i test su un dispositivo reale, per alcuni buoni motivi. Utilizzare lo schermo del computer è diverso da usare uno smartphone con due mani o un orologio con una sola mano. Alzare le braccia per interagire con Apple Watch costa fatica, che non si prova utilizzando il mouse e il computer. Le modalità di interazione del simulatore sono limitate: diversi componenti hardware non sono presenti sul computer e non sono neanche implementati in modo simulato. L'interazione con la corona digitale e con il Force Touch sono differenti sul dispositivo reale: sul computer sono solo simulati tramite un'interazione con caratteristiche differenti. Per tutti questi motivi è essenziale verificare il progetto dell'applicazione su un dispositivo reale e non limitarsi a convalidarlo nel simulatore.

Debugging

All'interno di Xcode sono presenti anche funzionalità per il debugging dell'applicazione. È possibile inserire breakpoint, eseguire il codice passo-passo, ispezionare il contenuto delle variabili e addirittura utilizzare la potente console per operazioni avanzate (Figura 1.16). Per utilizzare il debugger è sufficiente inserire un breakpoint nel codice, facendo clic sulla riga in corrispondenza della colonna che solitamente contiene il numero di riga. Al successivo avvio, sia esso nel simulatore o su un dispositivo reale, l'esecuzione si interromperà in corrispondenza del breakpoint, consentendo allo sviluppatore l'analisi dello stato della memoria, dei thread e così via. Tutte le funzionalità relative al debugger sono raccolte nel menu *Debug*, incluse quelle per l'esecuzione del codice, per la gestione di breakpoint e funzioni accessorie, come la simulazione di una determinata posizione geografica.

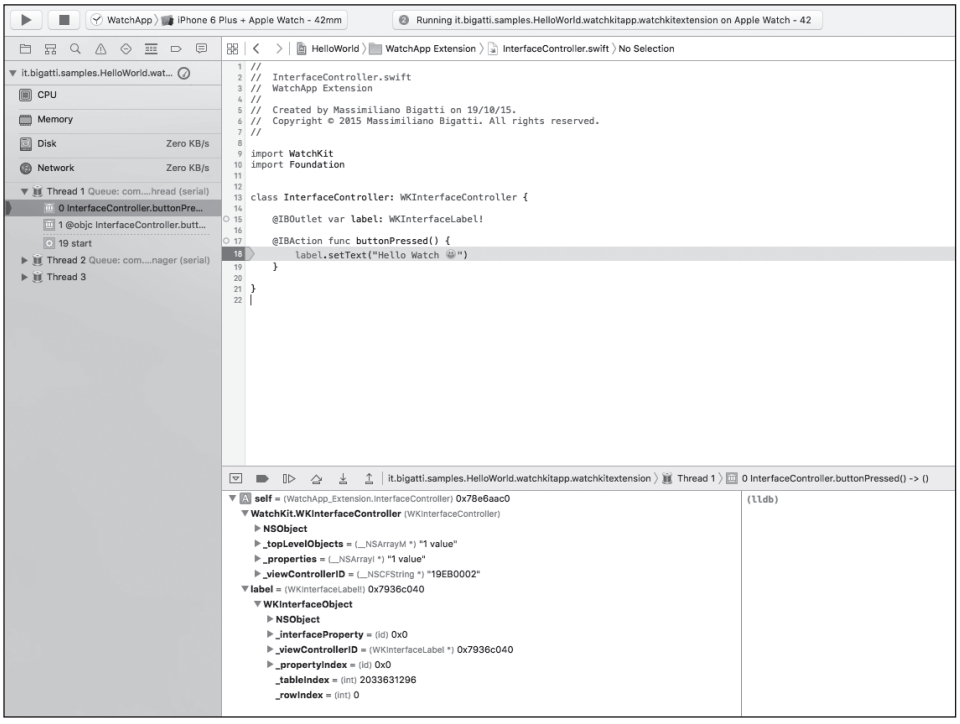

Figura 1.16 Gli strumenti di debugging di Xcode consentono di indagare il funzionamento del codice al fine di individuare eventuali errori.

ATTENZIONE

Gli strumenti di sviluppo messi a disposizione da Apple purtroppo non sono privi di bug. Alcune volte potrebbero non funzionare a dovere e potrebbe nascere il dubbio se un determinato comportamento della propria applicazione sia da imputare a un errore di programmazione o a fattori esterni. Nel dubbio è possibile eseguire qualche operazione di pulizia quale: ricompilare l'applicazione da zero (*Product* > *Clean* o Shift+Command+K), rimuovere l'applicazione dal simulatore e reinstallarla, cancellare i percorsi temporanei di Xcode (si trovano sotto *Xcode* > *Preferences* > *Locations*, campo *Derived Data*) o anche resettare il contenuto di un simulatore, accedendo alla voce di menù *Simulator* > *Reset Content and Settings* nello specifico simulatore da reimpostare. Infine, è possibile e talvolta indispensabile verificare il funzionamento su dispositivi reali (iPhone + Apple Watch) che tendenzialmente, in quanto prodotti destinati all'utente finale, diversamente da Xcode, dovrebbero presentare meno anomalie.

Hello World

Per concludere il capitolo, vediamo brevemente la realizzazione di una semplice applicazione per Apple Watch che si limita a presentare una scritta sullo schermo. Il risultato che vogliamo ottenere è illustrato in Figura 1.17. Per raggiungerlo procedere come segue.

1. Avviare Xcode.
2. Selezionare *File* > *New* > *Project* o premere Shift+Command+N per creare un nuovo progetto.
3. Selezionare il modello *Single View Application* nel gruppo *Application* nella sezione *iOS*.
4. Fare clic su *Next*.
5. Digitare il nome del prodotto nel campo *Product Name*, in questo caso **HelloWorld** e impostare le altre opzioni come in Figura 1.18.

Figura 1.17 L'output prodotto dalla nostra prima semplice applicazione di prova.

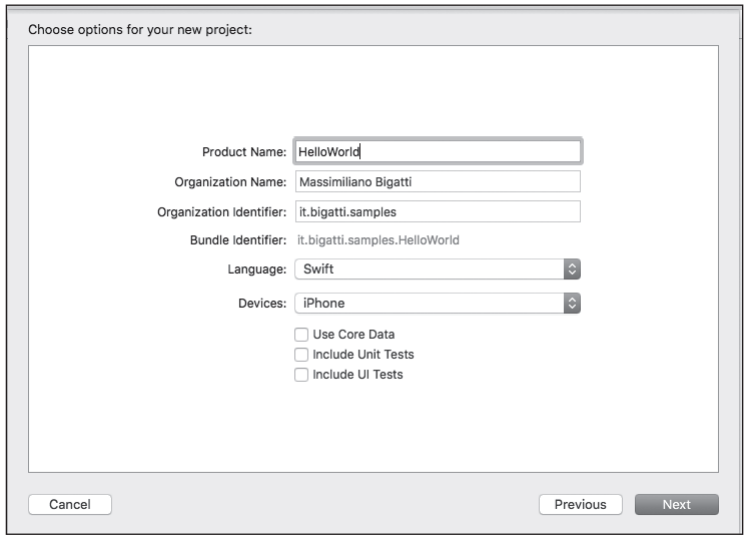

Figura 1.18 Creazione del progetto iOS necessario per ospitare l'applicazione per Apple Watch.

6. Selezionare un percorso dove memorizzare i file di progetto e fare clic su *Create*. In questa fase è possibile anche indicare se il progetto deve essere gestito tramite sistema di versionamento git.
7. Aggiungere un nuovo target come indicato in precedenza (pulsante + nel pannello con l'elenco dei target o *Editor* > *Add Target*).
8. Selezionare il modello *WatchKit App* nel gruppo *Application* nella sezione *watchOS*.
9. Fare clic su *Next*.
10. Specificare il nome del prodotto e le opzioni come indicato in Figura 1.12.
11. Fare clic su *Finish*.
12. In *Project Navigator* accedere allo storyboard, aggiungere un'etichetta all'interfaccia e configurare le opzioni come in Figura 1.19.

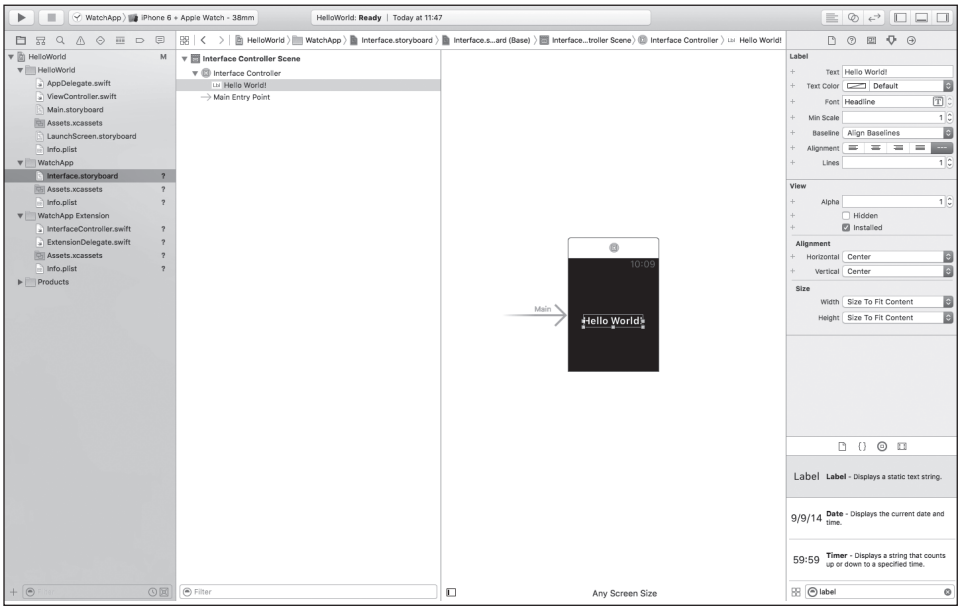

Figura 1.19 Creazione dell'interfaccia utente dell'applicazione Hello World.

13. Fare clic sul pulsante di avvio in alto a sinistra oppure selezionare *Product* > *Run* o premere Command+R. L'applicazione verrà avviata nel simulatore e mostrerà quanto presente in Figura 1.17.

Aggiungere interattività

Ora modificheremo l'applicazione per includere un pulsante che una volta premuto cambia il testo presente nell'etichetta. Per ottenere ciò si proceda come segue.

1. Aggiungere nell'interfaccia utente un pulsante come mostrato in Figura 1.20.
2. Collegare l'etichetta al codice creando un outlet.
3. Collegare l'azione del pulsante al codice creando un'azione.

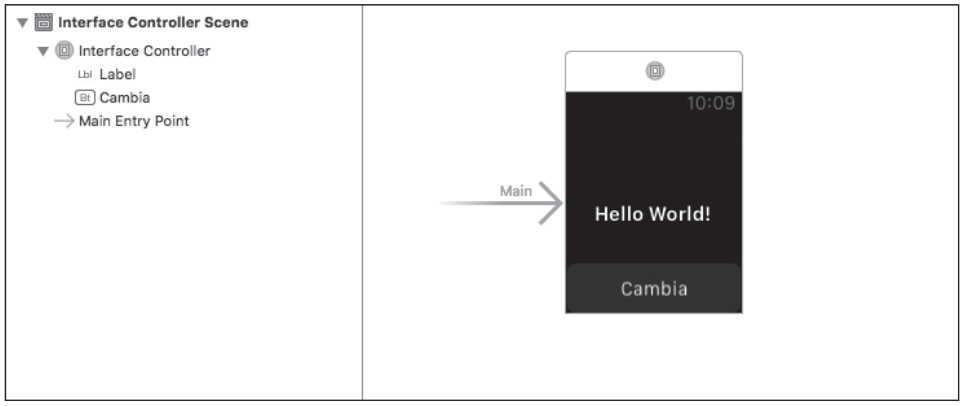

Figura 1.20 Aggiunta di un pulsante per la modifica del testo visualizzato.

All'interno del metodo inserire il codice seguente (il listato completo del controller dell'interfaccia è riportato nel Listato 1.1):

```
label.setText("Hello Watch 😃")
```

A questo punto è possibile eseguire l'applicazione. Si vedrà che al clic o tocco del pulsante, il testo dell'etichetta cambia per includere l'emoticon specificata da codice.

Listato 1.1 InterfaceController.swift.

```
import WatchKit
import Foundation


class InterfaceController: WKInterfaceController {

    @IBOutlet var label: WKInterfaceLabel!

    @IBAction func buttonPressed() {
        label.setText("Hello Watch  ")
    }


}
```