Politecnico di Milano
Scuola di Ingegneria Industriale e dell'Informazione

Tesi di Laurea Magistrale in
Computer Science and Engineering

A.A. 2016/17

Un approccio model-driven allo sviluppo di interfacce grafiche adattative per applicazioni mobili multipiattaforma

Candidato:
Massimo Beccari
Matr. 849879

Relatore:
Prof. Luciano Baresi

Ringraziamenti

Desidero innanzitutto ringraziare il prof. Luciano Baresi, relatore di questa tesi, per la disponibilità, la cortesia e l'aiuto fornitomi durante lo svolgimento di tutto il lavoro.

Un ringraziamento ad Attilio, Lorenzo, Davide, Adrian e a tutti i compagni con cui ho condiviso il percorso di studi e in particolare i lavori a progetto, per aver reso più piacevole il tempo speso in università e i momenti di lavoro.

Ringrazio i miei amici Nicola, Lucia e Simone, per i bei momenti passati insieme a Milano.

Infine desidero ringraziare i miei genitori, per il sostegno sia morale che economico, che mi hanno permesso di raggiungere questo traguardo e Silvia, per aver reso più belli e leggeri questi mesi di lavoro.

Indice

1 Introduzione **11**

2 Contesto Generale **15**
2.1 Stato dell'arte 15
2.1.1 Software di prototipizzazione 15
2.1.2 Software di sviluppo di applicazioni mobili 18
2.1.2.1 Strumenti di sviluppo di applicazioni cross-platform 19
2.1.2.2 Ambienti di sviluppo per una singola piattaforma 20
2.2 Obiettivo della tesi 22
2.3 Soluzione proposta 23

3 Fase di prototipizzazione **26**
3.1 Introduzione 26
3.2 Framework e librerie software utilizzate 27
3.2.1 Ember.js 27
3.2.2 Yeoman.io 28
3.2.3 Bootstrap 29
3.2.4 Altre librerie 29
3.3 Applicazione di partenza: Protocode 3.0 30
3.3.1 Architettura software 30
3.3.1.1 Routing 31
3.3.1.2 Mixin 33
3.3.1.3 Controller 33
3.3.1.4 View 34
3.3.1.5 Model 34
3.3.2 Interfaccia grafica 34
3.3.2.1 View Editor 35
3.3.2.2 Model Editor 36
3.4 Aggiornamento a Protocode 4.0 37
3.4.1 Vincoli di posizione 37
3.4.2 Vincoli di dimensione 38
3.4.3 Control Chain 40
3.4.4 Componente ScreenCanvas e menù 43
3.4.5 Nuovi dispositivi 45
3.4.6 Scene 46
3.4.7 Report View 49
3.4.8 Risultato finale 51

3.4.8.1 Interfaccia grafica . . . 51
3.4.8.2 Model e Routing . . . 53
3.4.8.3 Generazione del modello astratto . . . 55

4 Fase di generazione del codice 56
4.1 Introduzione . . . 56
4.2 Strumenti e framework utilizzati . . . 56
4.2.1 Eclipse Modeling Framework . . . 57
4.2.2 openArchitectureWare . . . 57
4.3 MobileCodeGenerator 3.0 . . . 59
4.3.1 Architettura di MobileCodeGenerator . . . 59
4.3.2 Model e Metamodel . . . 60
4.3.3 Model Checks e App extensions . . . 61
4.3.4 Package di Android . . . 62
4.3.5 Package di iOS . . . 62
4.3.6 Workflow . . . 63
4.4 Aggiornamento a MobileCodeGenerator 4.0 . . . 64
4.4.1 Metamodel . . . 64
4.4.2 Model Check e Estensioni . . . 65
4.4.3 Generazione del codice per Android . . . 65
4.4.3.1 Vincoli di posizione, dimensione e catene di controlli 65
4.4.3.2 View controller, scene e menù: codice XML . . . 67
4.4.3.3 View controller, scene e menù: codice Java . . . 69
4.4.3.4 Aggiornamento componenti di AndroidStudio . . . 71
4.4.4 Generazione del codice per iOS . . . 71
4.4.4.1 Vincoli di posizione e dimensione . . . 72
4.4.4.2 Catene di controlli . . . 73
4.4.4.3 Menù . . . 75
4.4.4.4 View controller e scene . . . 77
4.4.4.5 Aggiornamento di XCode . . . 79

5 Valutazione 80
5.1 Presentazione del caso di test . . . 80
5.2 Sviluppo . . . 81
5.2.1 Prototipizzazione e generazione del codice . . . 81
5.2.2 Sviluppo dell'applicazione finale . . . 84
5.3 Analisi quantitativa . . . 89
5.4 Analisi qualitativa . . . 90

6 Conclusioni 92
6.1 Sviluppi futuri . . . 93

Bibliografia 96

Acronimi 100

Elenco delle figure

1.1 Utilizzatori di smartphone per anno (in bilioni). Fonte: Statista.com 11
1.2 Vendite di dispositivi mobili raggruppate per SO. Fonte: Statista.com 12

2.1 Interfaccia dell'applicazione desktop di Balsamiq Mockups 16
2.2 Editor di azioni di Marvel 17
2.3 Schermate del prototipo in InVision 17
2.4 Interfaccia grafica di Proto.io 18

3.1 Struttura di Protocode 31
3.2 Route di Protocode 3.0 32
3.3 Interfaccia del View Editor di Protocode 3.0 35
3.4 Interfaccia del Model Editor di Protocode 3.0 36
3.5 Interfaccia per la creazione e modifica di vincoli di posizione . . . 38
3.6 Interfaccia per la creazione e modifica di vincoli di dimensione . . 40
3.7 Tipi di Control Chain 42
3.8 Interfaccia per la creazione e modifica di Control Chain 43
3.9 Alcuni dei diversi indicatori dei vincoli in Protocode 44
3.10 Aspetto del nuovo menù in Android (sinistra) e iOS (destra) . . . 45
3.11 Nexus 9 in Protocode 4.0 46
3.12 Template di modifica di una scena in Protocode 4.0 48
3.13 Esempio di una scena di tipo multiVC in Protocode 4.0 49
3.14 Esempio di Report View in Protocode 4.0 50
3.15 Interfaccia del View Editor di Protocode 4.0 52
3.16 Pannello Application del View Editor di Protocode 4.0 52
3.17 Class Diagram parziale del Model di Protocode 4.0 53
3.18 Route di Protocode 4.0 54
3.19 Struttura di un file di modello XMI di Protocode 4.0 55

4.1 Processo di generazione del codice di MobileCodeGenerator 58
4.2 Porzione di meta model nel Sample Ecore Model Editor di Eclipse 61
4.3 Traduzione di PositionConstraint in Android 66
4.4 Traduzione di DimensionConstraint in Android 66
4.5 File di layout di un'Activity per Android generati da MCG 4.0 . . 69
4.6 Interazione che genera un evento di navigazione in Android 71
4.7 Traduzione dei constraint in XML per la storyboard di iOS 72
4.8 Confronto di constraint tradotto in XML e in Swift per iOS . . . 73
4.9 Traduzione di una catena di tipo spread in iOS 75
4.10 Traduzione di una catena di tipo packed in iOS 75

4.11 Interazione con il menù in iOS 77

5.1 View controller e scene di MathKit 84
5.2 Activity Algebra su smartphone e tablet (Android) 86
5.3 Activity Equations su smartphone e tablet (Android) 87
5.4 Activity Guide su smartphone e tablet (Android) 88
5.5 Rapporto quantità di codice sorgente generato 89

Sommario

Il successo e la diffusione dei dispositivi mobili come smartphone, tablet e smartwatch a cui si è potuto assistere negli ultimi anni ha causato un cambiamento nel mondo degli sviluppatori software, che hanno guardato sempre più con interesse le possibilità economiche offerte dalle applicazioni mobile. Se il mercato degli smart-devices risulta moderatamente frammentato, lo stesso non può dirsi per la situazione dei sistemi operativi, dominata da Android e iOS. La necessità di supportare entrambi questi sistemi per raggiungere la quasi totalità del mercato delle app ha fatto sì che nascessero una moltitudine di strumenti volti allo sviluppo di applicazioni in grado di funzionare su entrambe le piattaforme: sono nate così le app ibride, il cui sviluppo permette grandi risparmi in termini economici e di tempo, pur andando a produrre applicazioni con caratteristiche non all'altezza di quelle ottenute dallo sviluppo nativo. Questa tesi si inserisce nel contesto appena descritto: saranno presentati due strumenti sviluppati in questo lavoro che permettono, a partire dal design grafico dell'applicazione, di generare automaticamente codice nativo per entrambi i principali sistemi operativi mobile; questo consente un notevole risparmio di tempo nella fase iniziale dello sviluppo, avvicinando i costi di quello nativo e ibrido. In particolare, il presente lavoro si è concentrato sul problema dell'adattabilità dell'interfaccia: esso è un problema di importanza critica dal momento che gli odierni dispositivi presentano display di dimensione e rapporto sempre più variegati; lo sviluppo di interfacce dinamiche e componibili infatti può essere determinante nel successo di un'applicazione, consentendole di raggiungere una grossa fetta dei dispositivi presenti sul mercato. La soluzione presentata in questa tesi è partita dallo studio approfondito dei meccanismi di composizione dell'interfaccia messi a disposizione da ciascun sistema operativo, che sono stati rielaborati in un singolo sistema per permettere l'applicazione dell'approccio *model-driven*, secondo il quale un unico modello comune viene utilizzato per la generazione del codice per svariate piattaforme.

Abstract

The success and spread of mobile devices like smartphones, tablets and smartwatches that we have been able to see in the last years has caused a shift in the world of software developers, who have watched with increasing interest the economic possibilities offered by mobile applications. While the smart-devices market turns out to be quite fragmented, the same cannot be said for the situation about the operating systems running on these devices, leaded by Android and iOS. The necessity of supporting both these systems to achieve almost the entire app market gave rise to the birth of a crowd of tools for the development of applications which could work on both the platforms: the hybrid apps were born in this way and their development has allowed great savings of money and time, while producing applications with features not up to those obtained with native development. This thesis is inserted in the context just described: two tools developed in this work will be presented, which allow the automatic generation of native code for both the main mobile operating systems, starting from the graphical design of the application; this operation leads to considerable time savings in the initial phase of the development, bringing the costs of the native and hybrid ones closer. In particular, the attention of this thesis has been focused on the adaptability of the interface: this problem is a fact of critical importance since today's devices present displays with increasingly diversified size and ratio; the development of dynamic and modular interfaces in fact can be crucial for the success of an application, allowing it to reach a large portion of the market. The solution presented in this thesis began from the in-depth study of the mechanisms for the composition of the interface provided by each operating system, that were re-elaborated in a single system to allow the application of the *model-driven* approach, according to which a single common model is used to generate code for different platforms.

Capitolo 1

Introduzione

Nel Gennaio 2014, per la prima volta negli U.S.A., veniva registrato che il tempo speso navigando su internet da dispositivi mobili superava quello speso navigando da PC [1]. Questo sorpasso, già predetto da tempo, evidenziava il cambiamento che era in atto nel mondo degli sviluppatori software, che sempre più volgevano la propria attenzione verso lo sviluppo di applicazioni per dispositivi mobili piuttosto che di programmi per PC. In questi ultimi anni si è dunque visto un progressivo aumento degli utenti di dispositivi mobili come smartphone (fig. 1.1), tablet e smartwatch, con un conseguente aumento del numero delle applicazioni sviluppate per essi e degli introiti derivati da queste (stimati in quasi 200 bilioni di dollari per il 2020 [2]).
Il mercato degli smart-devices, seppur comandato dalle storiche *Apple* [3] e *Samsung* [4], sta vedendo crescere le giovani *Huawei* [5], *Xiaomi* [6], *Oppo* [7] e *Vivo* [8] (queste due ultime appartenenti alla *BBK Electronics Corporation*), rimanendo così piuttosto frammentato [9].

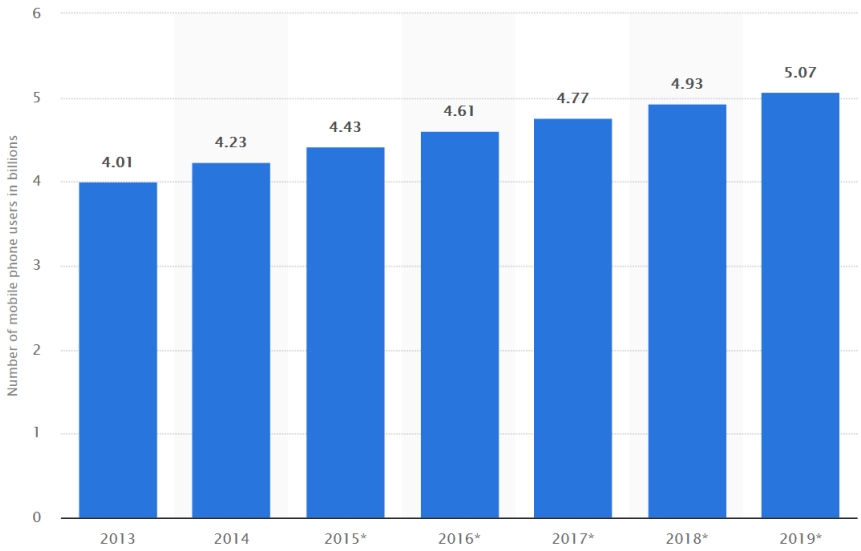

Figura 1.1: Utilizzatori di smartphone per anno (in bilioni). Fonte: Statista.com

La situazione è invece diversa per quanto riguarda i principali sistemi operativi mobile; infatti il mondo dei dispositivi mobili è dominato da *Android* [10] di *Google* [11] e da *iOS* [12] di Apple, con una percentuale di dispositivi venduti nel secondo trimestre del 2017 rispettivamente del 87.7% e 12.1%, per un totale del 99.8% (fig. 1.2). Appare quindi chiaro che un'applicazione mobile abbia maggiori possibilità di mercato se sviluppata per i sistemi operativi di Google e Apple; la vera scelta progettuale che rimane da fare è quella del tipo di applicazione: nativa, web o ibrida. Le applicazioni native sono applicazioni sviluppate specificatamente per un particolare sistema operativo; questo consente ad esse di avere prestazioni, stabilità e usabilità ottimali e di poter sfruttare al meglio tutte le funzionalità del dispositivo su cui sono installate. Le web app invece sono applicazioni che non risiedono nel dispositivo che le utilizza: sono implementate in server, ai quali i device accedono tramite un browser. Questo fa sì che non occupino spazio sul dispositivo e che il loro sviluppo sia indipendente dal sistema operativo sul quale verranno utilizzate; d'altro canto esse hanno scarso supporto per le funzionalità del device, prestazioni che dipendono dai server remoti e oltretutto per utilizzarle è necessaria una connessione ad internet (la cui qualità incide sulle prestazione della web app). Le applicazioni ibride infine sono una sorta di "via di mezzo" tra web app e applicazioni native. Sono sviluppate con linguaggi cross-platform (generalmente HTML+JS) e spesso incapsulate in applicazioni native tramite Web Views (una sorta di contenitori per codice scritto per il web). Questo offre lo stesso vantaggio delle web app per quanto riguarda il fatto che il codice è sostanzialmente unico e indipendente dalla piattaforma su cui viene eseguito, ma a differenza delle web app non per forza necessitano di connessione internet e hanno un discreto accesso alle funzionalità del dispositivo cui sono installate. Tuttavia, le prestazioni e la stabilità delle app ibride rimangono inferiori a quelle delle applicazioni native. Per questo motivo, parlando di qualità,

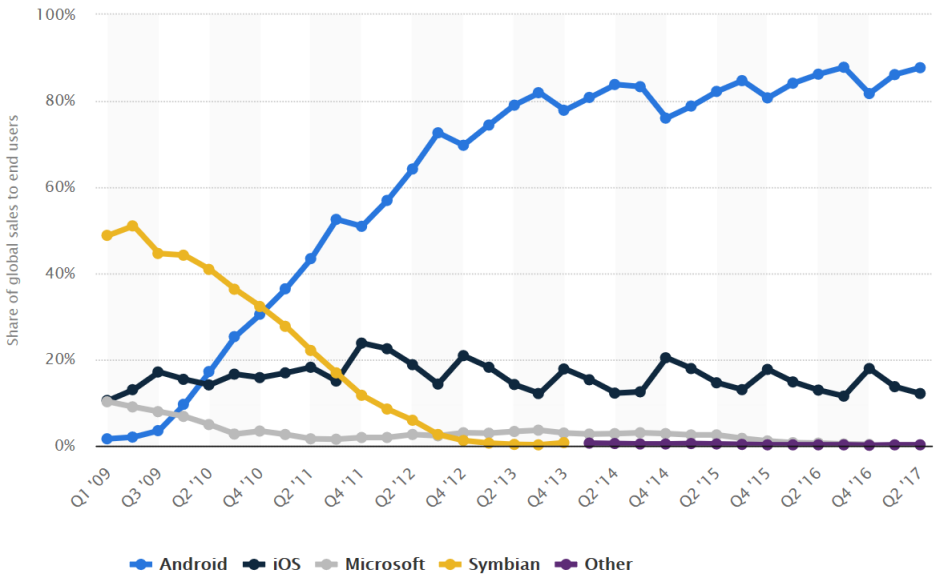

Figura 1.2: Vendite di dispositivi mobili raggruppate per SO. Fonte: Statista.com

le applicazioni native rimangono migliori, anche se hanno costi di sviluppo più elevati. Infatti il problema, in termini di tempo e budget, è che generalmente necessitano di diversi team che lavorino ciascuno distintamente su ogni sistema operativo supportato (quindi nel caso della scelta di Android e iOS come sistemi target per l'applicazione, servirebbero due diversi team di programmatori, con conseguenti costi e tempi di sviluppo elevati rispetto a una scelta di tipo ibrido o web).

In questo lavoro di tesi viene presentata una soluzione che permette di ottenere un'applicazione scritta in codice nativo per ciascuno dei due sistemi operativi di interesse (Android e iOS) a partire da un unico modello composto utilizzando uno strumento grafico. Questo permette di ottenere in pochi minuti una buona base dell'applicazione da sviluppare (principalmente la parte di codice di interfaccia grafica (GUI) e di modello dei dati), riducendo così tempi e costi di sviluppo. Gli strumenti software che vanno a comporre tale soluzione sono due: Protocode, originariamente sviluppato da Mattia Natali nel 2013 [13], e MobileCodeGenerator, la cui prima versione è stata sviluppata da Gregorio Perego e Stefania Pezzetti nel 2012 [14]. Protocode è una web application che permette di comporre l'interfaccia grafica dell'applicazione utilizzando controlli e strumenti preconfigurati: è così possibile vedere un'anteprima dell'aspetto grafico che avrà l'applicazione generata, su ciascuno dei due sistemi operativi supportati e sui diversi dispositivi disponibili. Al termine della fase di design, Protocode produce un file di modello dell'applicazione che verrà poi tradotto in codice sorgente da MobileCodeGenerator, realizzato utilizzando alcuni plugin di *Eclipse* [17], un noto IDE multipiattaforma. Le funzionalità di entrambi i software sono state poi estese e aggiornate da successivi lavori di tesi: Aldo Pintus nel 2016 ha realizzato la versione 2.0 [15], aggiungendo la possibilità di creare applicazioni per smartwatch; Alessio Rossotti nel 2017 ha realizzato la versione 3.0 [16], aggiungendo la possibilità di definire e generare automaticamente la componente di dati (Model) dell'applicazione. Con questo lavoro di tesi i due tool hanno raggiunto la versione 4.0: essa, oltre ad aggiornare il supporto per le ultime tecnologie disponibili, si focalizza sul concetto di adaptivity (adattabilità) dell'interfaccia grafica dell'applicazione. Questo concetto è diventato sempre più importante dal momento che i dispositivi mobili prodotti negli ultimi anni si sono ampiamente diversificati, tra le altre caratteristiche, nelle dimensioni dello schermo e nel loro rapporto: a titolo di esempio, si pensi soltanto al cambiamento dello schermo di iPhone, che dalla versione 4s alla versione X ha visto variare le dimensioni da 3.5" a 4" (iPhone 5, 5c e 5s), a 4.7" (iPhone 6, 7 e 8), a 5.5" (iPhone 6 Plus, 7 Plus e 8 Plus) e a 5.8" (iPhone X) [18]. La progettazione di un'interfaccia grafica che sappia adattare il proprio contenuto a display di diverse dimensioni è diventata quindi indispensabile al giorno d'oggi; un'applicazione senza questa caratteristica risulterebbe inutile o comunque difficilmente utilizzabile su un dispositivo diverso da quello su/per cui è stata progettata. A partire da queste considerazioni, il mio lavoro è stato inizialmente volto a riprogettare il sistema di posizionamento dei controlli della GUI utilizzando un meccanismo basato su constraints (vincoli); tale meccanismo è alla base del posizionamento dei controlli in iOS, mentre per quanto riguarda Android non è mai stato diffusamente utilizzato fino a tempi

molto recenti (la libreria di supporto ConstrainLayout di Android esisteva già da Android 2.3, ma è rimasta ad una versione beta fino a Febbraio 2017 in cui è stata rilasciata la versione 1.0 [19]; successivamente il suo sviluppo è proseguito fino all'ultima versione disponibile, la 1.1 beta 6, rilasciata in Marzo 2018 [20]). Il mio lavoro è poi proseguito andando ad estendere ulteriormente le capacità di adattamento dell'interfaccia grafica delle applicazioni prodotte con la toolchain Protocode-MobileCodeGenerator ai tablet, dispositivi con schermi di dimensioni molto superiori a quelle degli smartphone (da 7" a oltre 12"): è ora infatti possibile progettare applicazioni che non solo adattano posizione e dimensione dei controlli alla dimensione e al rapporto del display, ma possono addirittura mostrare schermate diverse a seconda dello spazio disponibile (quello che su uno smartphone verrebbe visualizzato in due schermate diverse, ad esempio, può essere visualizzato contemporaneamente in una schermata "composita" su un tablet). Quest'ultima estensione ha permesso, tra l'altro, la definizione di parti di GUI riutilizzabili in diverse schermate dell'applicazione.

Nei successivi capitoli di questo documento saranno descritte nel dettaglio le diverse fasi di questo lavoro; di seguito è riportata una sintesi di ciascun capitolo:

- il *secondo capitolo* presenta il contesto generale nel quale questo lavoro si inserisce, analizzando lo stato dell'arte dello sviluppo di applicazioni mobili multipiattaforma e valutando quindi le soluzioni già esistenti al problema dello sviluppo cross-platform;

- il *terzo capitolo* illustra la fase di prototipizzazione, dando una presentazione generale della struttura di Protocode e delle tecnologie utilizzate per svilupparlo, a cui segue una dettagliata descrizione delle modifiche apportate in questo lavoro di tesi;

- il *quarto capitolo* illustra, in modo simile al capitolo precedente, la fase di generazione del codice, presentando MobileCodeGenerator e descrivendo gli interventi eseguiti al fine di renderlo in grado di generare tutto il codice relativo alle nuove funzionalità introdotte in Protocode;

- il *quinto capitolo* presenta un'analisi qualitativa e quantitativa della nuova versione dei due software, mettendo in evidenza la quantità di codice che è stato possibile generare automaticamente e la qualità di esso;

- il *sesto capitolo* infine contiene le considerazioni conclusive del lavoro svolto e alcuni spunti di riflessione sui possibili sviluppi futuri.

Capitolo 2

Contesto Generale

Questo capitolo presenta il contesto generale in cui il lavoro di questa tesi si inserisce: di seguito verrà analizzato lo stato dell'arte degli strumenti per la prototipizzazione e lo sviluppo di applicazioni mobili, prendendo in esame i principali software già presenti sul mercato; successivamente verrà descritto l'obiettivo della tesi, ponendo l'attenzione sulle motivazioni e le scelte fatte per la sua realizzazione; infine verrà presentata la soluzione proposta, che verrà in seguito approfondita nei successivi capitoli.

2.1 Stato dell'arte

2.1.1 Software di prototipizzazione

Le applicazioni di prototipizzazione sono strumenti che permettono, attraverso editor grafici (non c'è bisogno di scrivere codice), di disegnare l'interfaccia utente dell'applicazione che si vuole sviluppare; essi sono molto utili al fine di avere un'anteprima visiva di quella che sarà la GUI e valutarne le possibili alternative, prima di iniziare la fase di sviluppo in codice vera e propria. Sono riportate di seguito alcune delle principali alternative presenti sul mercato, insieme alla descrizione delle loro caratteristiche e funzionalità.

Balsamiq Mockups

Balsamiq Mockups [21] è uno strumento di prototipizzazione, disponibile sia come applicazione desktop sia come web application, che permette di creare i mockup dell'applicazione, vale a dire le schermate, come se si stesse disegnando su una lavagna. Essa mette a disposizione una moltitudine di elementi grafici (come bottoni, barre, oggetti testuali ecc.) personalizzabili ed è inoltre possibile importare elementi creati da altri utenti appartenenti alla sua vasta community, consentendo di velocizzare il processo di creazione; un'altra caratteristica di Balsamiq Mockups è la possibilità di creare versioni alternative dello stesso mockup, raggruppate insieme, per modellare ad esempio la stessa schermata su dispositivi diversi. Questo software è gratis per un periodo di prova di 30 giorni, terminato il quale è necessario sottoscrivere un piano a pagamento per poter continuare ad utilizzarlo; il piano più economico costa 9 $ al mese.

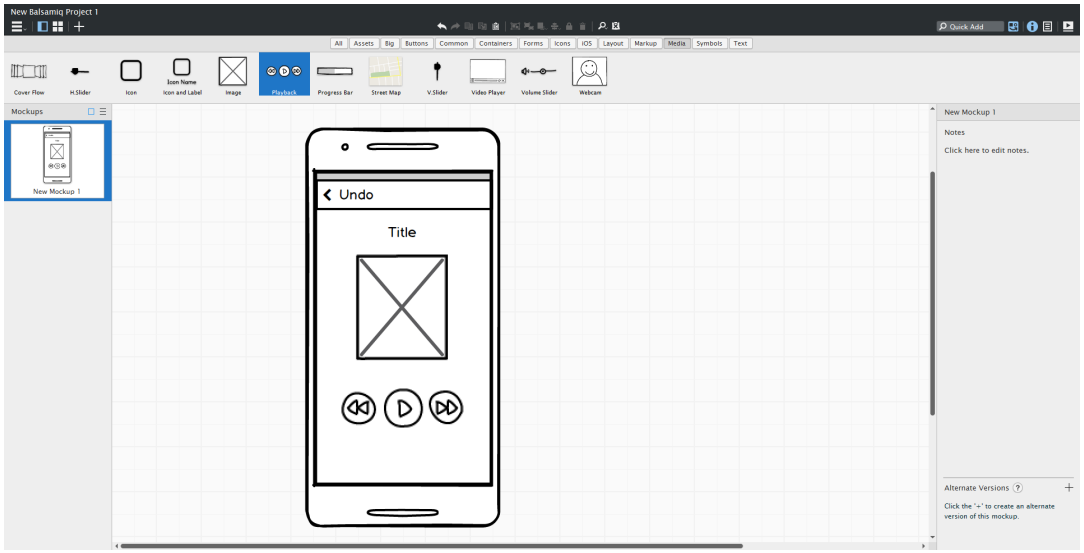

Figura 2.1: Interfaccia dell'applicazione desktop di Balsamiq Mockups

Marvel

Marvel [22] è una web application che permette la prototipizzazione di applicazioni su diversi dispositivi iOS (iPhone, iPad e Apple Watch), su un generico smartphone Android o su un dispositivo di risoluzione variabile. Essa ha un editor integrato che consente di disegnare le diverse schermate dell'applicazione utilizzando alcuni widget preimpostati o strumenti di disegno basilari come linee o forme geometriche semplici, ma permette anche di caricare file immagine dal proprio computer, in modo da poter utilizzare schermate disegnate con altri software o fotografate da disegni a mano libera. Una volta create o caricate le schermate, è possibile definire il comportamento dell'interfaccia attraverso un editor di azioni (figura 2.2): per prima cosa si seleziona un'area della schermata (ad esempio quella occupata da un bottone); poi si sceglie la schermata a cui si vuole navigare; infine si sceglie il tipo di interazione dell'utente con l'area selezionata (click, tap e altre) e il tipo di transizione da mostrare nel passaggio da una schermata all'altra (ad esempio scomparsa a sinistra, dissolvenza ecc.). Dopo aver definito tutte le azioni desiderate, è possibile testare il comportamento dell'app cliccando sul pulsante "Play", che aprirà una nuova finestra del browser contenente l'anteprima del dispositivo scelto alla creazione del progetto che visualizza la prima schermata: agendo con il mouse sulle aree per cui è stata definita un'azione sarà possibile osservare il comportamento dell'applicazione. Marvel è completamente gratuito in un piano che consente un massimo di due progetti, non scaricabili; sono presenti altri due piani a pagamento, rispettivamente da 12 e 42 $ al mese, che offrono maggiori funzionalità, come la creazione di un illimitato numero di progetti, il loro download e la collaborazione tra diversi utenti allo stesso progetto.

InVision

Un'altra applicazione web di prototipizzazione è *InVision* [23] che, come la precedente, permette il design dell'interfaccia di applicazioni mobili per diversi di-

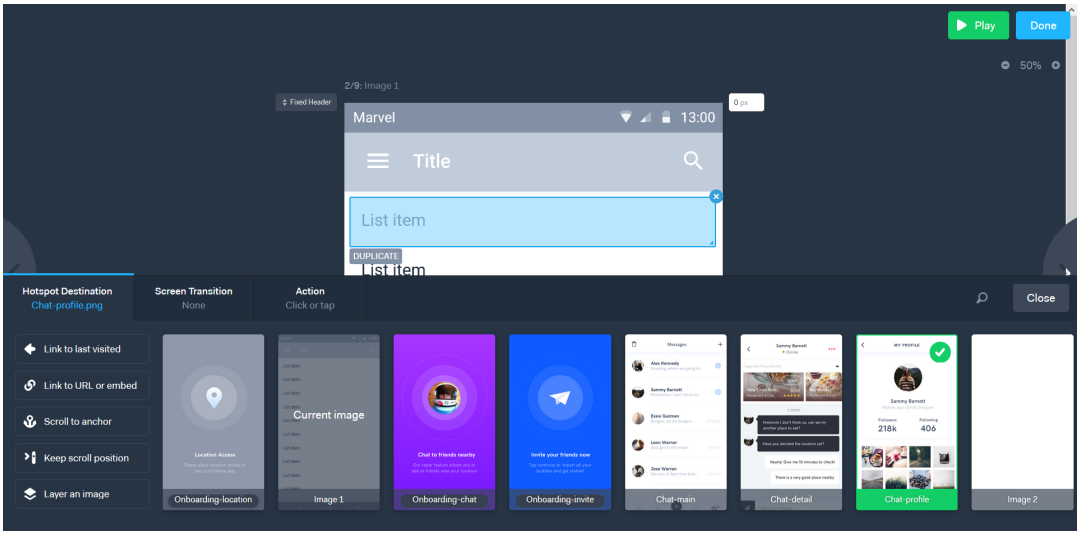

Figura 2.2: Editor di azioni di Marvel

spositivi, Android e iOS, di diversi tipi (telefoni, tablet, smartwatch). Il suo funzionamento assomiglia molto a quello di Marvel, con la differenza che in InVision è solo possibile caricare le schermate da propri file, creati con altri strumenti; dall'altro lato, InVision permette una molto più vasta tipologia di azioni e transizioni e la simulazione del comportamento dell'applicazione, oltre che su web player, è testabile anche su un dispositivo fisico attraverso l'app dedicata. InVision è gratis per sempre per un singolo prototipo, ma sono presenti altri piani che permettono più prototipi a partire da 13 $ al mese.

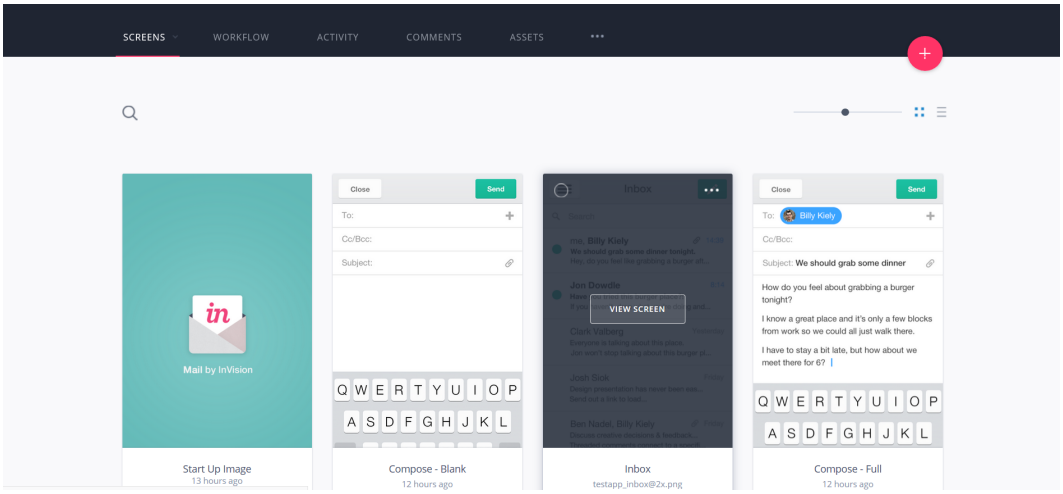

Figura 2.3: Schermate del prototipo in InVision

Proto.io

Proto.io [24] è anch'essa un'applicazione web che, come Balsamiq Mockups, mette a disposizione un editor grafico ricco di funzionalità per la composizione dell'interfaccia dell'applicazione a partire da una moltitudine di widget ed icone; esso è

inoltre simile a Marvel e InVision dal momento che, come questi ultimi, permette la definizione del comportamento dell'applicazione attraverso eventi di navigazione associati ad azioni e il testing dello stesso tramite l'app mobile dedicata. Proto.io è forse il più completo strumento di prototipizzazione tra quelli qui presi in considerazione, avendo un ricco editor di azioni simile a quello di InVision e un editor grafico più potente di quello di Marvel e paragonabile a quello di Balsamiq. Proto.io è gratuito solo per un set limitato di funzionalità; dopo il periodo di prova di 15 giorni (durante il quale è possibile sfruttarne tutte le caratteristiche), l'account viene limitato a meno che non sia effettuata l'iscrizione ad un piano a pagamento, a partire da 24 $ al mese.

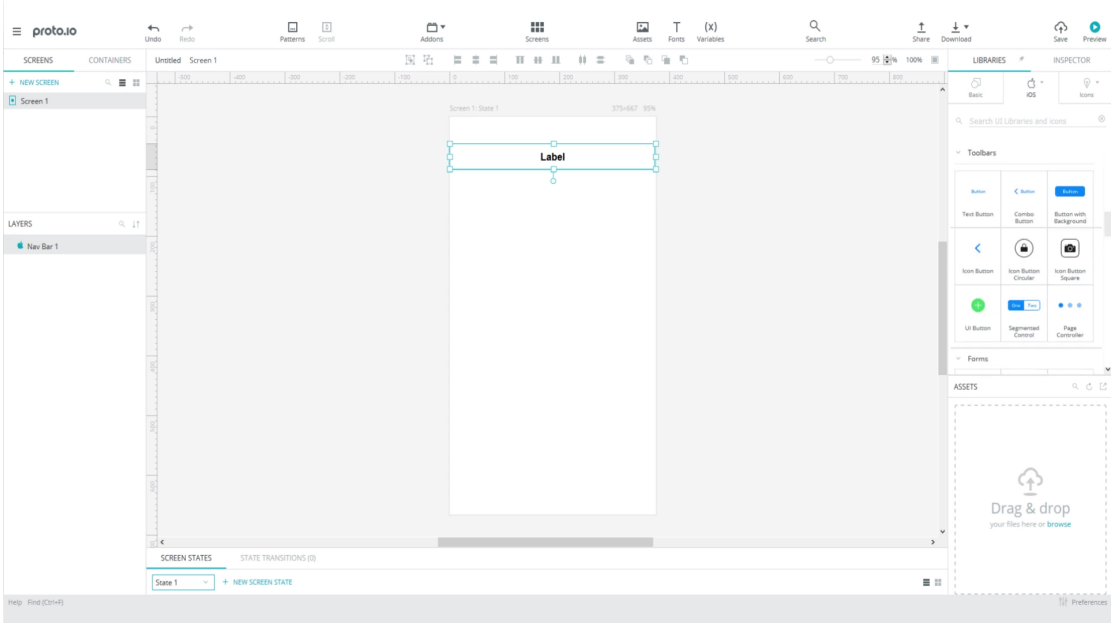

Figura 2.4: Interfaccia grafica di Proto.io

2.1.2 Software di sviluppo di applicazioni mobili

In questa sezione sono presentati i principali strumenti presenti sul mercato che supportano il programmatore nella fase di sviluppo vero e proprio di applicazioni mobili; essi sono generalmente degli IDE (Integrated Development Environment), in quanto forniscono un ecosistema di strumenti che assistono lo sviluppatore in ogni operazione, dalla stesura del codice alla compilazione, alla fase di build e al debugging. Come già anticipato nell'introduzione, quando si parla di applicazioni mobili la scelta da fare prima di iniziarne lo sviluppo riguarda le tre tipologie di applicazione possibili, vale a dire web, ibrida o nativa; tuttavia, per applicazioni di una certa complessità, le scelte più appetibili sono le ultime due, dal momento che le web application permettono un limitato utilizzo delle funzionalità del dispositivo essendo eseguite su server remoti acceduti tramite browser. Per questo motivo, gli strumenti considerati e descritti nelle sezioni successive sono quelli per lo sviluppo di applicazioni ibride o native, suddivisi a seconda che permettano lo sviluppo su diverse piattaforme (cross-platfrom) o su una sola.

2.1.2.1 Strumenti di sviluppo di applicazioni cross-platform

Xamarin

Xamarin [25] è forse il più famoso ambiente di sviluppo cross-platform attualmente disponibile; esso permette lo sviluppo di un'applicazione a partire da un unico linguaggio, C#, che poi viene automaticamente tradotto in codice nativo su diverse piattaforme, come Android, iOS e anche sistemi desktop come Windows e macOS. Xamarin è nato da ingegneri provenienti dal progetto Mono [26] ma è stato successivamente acquisito da Microsoft nel 2016; in seguito a ciò, il framework Xamarin è stato reso utilizzabile, oltre che dall'IDE dedicato *Xamarin Studio* disponibile solo su macOS, anche all'interno di *Visual Studio*, l'IDE per eccellenza di Microsoft. I principali vantaggi offerti da Xamarin sono le integrazioni con le API native di ciascun specifico sistema, la possibilità di disegnare l'interfaccia utente utilizzando i componenti nativi di ogni piattaforma e la possibilità di fare il build e il test dell'applicazione direttamente nel cloud tramite il servizio App Center Test, che mette a disposizione centinaia di dispositivi reali. Xamarin è inoltre gratuito per piccoli team, mentre per le imprese ha un costo variabile concordato direttamente con l'azienda.

Titanium

Titanium [27] è un altro framework per lo sviluppo cross-platform molto famoso e basato su JavaScript: attraverso l'IDE dedicato *Appcelerator Studio* è possibile sviluppare completamente la logica e l'interfaccia di un'applicazione utilizzando codice JavaScript, che verrà compilato in applicazioni native per Android e iOS. Le applicazioni prodotte con Titanium tuttavia non sono vere e proprie app native: il codice JavaScript infatti è incorporato all'applicazione attraverso particolari componenti e viene interpretato a runtime dal motore JavaScript del sistema su cui l'app è in esecuzione; si può parlare quindi di applicazioni ibride, le cui prestazioni rimangano inferiori a quelle di vere applicazioni native. E' anche vero però che, attraverso un altro strumento fornito insieme ad Appcelerator Studio e chiamato *Hyperloop*, è possibile utilizzare le API native di ciascun sistema e incorporare librerie e codice nativo scritti per specifiche piattaforme: in questo modo, una parte dell'applicazione viene formulata in codice JavaScript, che offre minori prestazioni ma permette di essere riutilizzato su diverse piattaforme, mentre un'altra parte verrà sviluppata attraverso codice nativo, che offre migliori prestazioni ma richiede di essere scritto indipendentemente per ogni sistema su cui l'applicazione verrà eseguita; il risultato è quindi un'app "a metà strada" tra ibrido e nativo, con prestazioni che si avvicinano molto a quelle di applicazioni completamente native. Appcelerator Studio e Hyperloop sono gratuiti, insieme ad un mese di dati per l'analisi del prodotto, in un piano chiamato "Indie Seat"; attraverso un piano da 99 $ al mese è possibile ottenere invece il "Pro Seat", che offre 3 mesi di dati per l'analisi del prodotto e altri strumenti, come ad esempio App Designer (un editor grafico dell'interfaccia dell'applicazione) e App Preview (che automatizza il test dell'applicazione prodotta).

PhoneGap

Un altro framework per lo sviluppo di applicazioni ibride è *PhoneGap* [28]: sviluppato da Adobe e basato sul progetto *Apache Cordova* (a cui contribuiscono la stessa Adobe e diverse altre grandi aziende come Google, IBM, Microsoft e Intel), esso permette di scrivere il codice di un'applicazione attraverso gli stessi linguaggi utilizzati nello sviluppo di web application, vale a dire HTML, CSS e JavaScript. Come in Titanium, anche in PhoneGap è possibile utilizzare le API native di ciascun sistema e compilare, insieme al resto del codice HTML, CSS e JavaScript, anche parti di codice scritto nei linguaggi nativi di ciascuna piattaforma. PhoneGap è gratuito e fornisce un'interfaccia a linea di comando (CLI), un'app mobile per il test live dell'applicazione prodotta e il servizio *PhoneGap Build*, che automatizza appunto la fase di build dell'applicazione effettuandola direttamente nel cloud. Recentemente, è stata rilasciata anche un'applicazione desktop (anche se ancora in versione beta), che può essere utilizzata in alternativa all'interfaccia a linea di comando, rendendo lo sviluppo con PhoneGap ancora più semplice e comodo.

Ionic Framework

L'ultimo framework preso in considerazione per lo sviluppo di applicazioni cross-platform è *Ionic* [29]; esso segue lo stesso principio di funzionamento di PhoneGap, essendo anch'esso basato su Apache Cordova. Ionic si concentra sul supporto alla creazione delle interfacce grafiche ed è definito dai suoi stessi autori "il Bootstrap per le applicazioni ibride": infatti, come Bootstrap è diventato un punto di riferimento per la creazione di siti e applicazioni web con interfacce responsive, così Ionic si pone come riferimento nella definizione di interfacce grafiche di applicazioni mobili ibride che, pur non utilizzando le API native di ciascun sistema, siano il più possibile simili a quelle che si otterrebbero sfruttandole (alcune API native sono comunque utilizzabili in Ionic, come in Titanium e PhoneGap); questo framework si basa inoltre su *AngularJS*, il framework web open source sviluppato da Google e altri sviluppatori individuali. Ionic fornisce un insieme di tool di supporto alle operazioni di sviluppo come ad esempio *Creator*, uno strumento di prototipizzazione, oppure *View App*, un'app mobile che permette il test dell'applicazione prodotta (simile a quella fornita con PhoneGap); in più, esso fornisce un servizio di monitoraggio chiamato *Monitor* che permette di individuare errori di funzionamento a runtime e un servizio di deployment, *Deploy*, che permette di spedire in tempo reale gli aggiornamenti di codice e correzioni di bug a tutti i dispositivi su cui l'applicazione è installata. Ionic ha un piano gratuito che fornisce un insieme limitato di strumenti, mentre ne esistono altri tre, rispettivamente da 29, 49 e 199 $ al mese, che completano progressivamente il set di funzionalità a disposizione del programmatore.

2.1.2.2 Ambienti di sviluppo per una singola piattaforma

Come si è potuto osservare, esiste una moltitudine di strumenti e framework che permettono lo sviluppo di codice multi-piattaforma, molti dei quali sono totalmente gratuiti; essi sono oggi ampiamente utilizzati, grazie al fatto che si sono

evoluti nel corso degli anni migliorando le prestazioni del codice prodotto e avvicinando il *look and feel* delle interfacce generate attraverso essi a quelle che si possono ottenere nativamente su ciascun sistema. Nonostante questo, l'integrazione con le funzionalità offerte da ciascun sistema e le prestazioni delle applicazioni ibride rimangono ancora distanti da quelle delle app native, così come l'aspetto delle interfacce grafiche create con questi framework rimane almeno in parte differente da quello ottenuto attraverso le librerie di sistema di ciascuna piattaforma. Per questo motivo, sopratutto se si parla di grandi applicazioni o comunque di app che offrono complesse funzionalità, la scelta migliore e in qualche caso obbligata rimane quella dello sviluppo nativo, anche se necessita di un team di sviluppo dedicato per ogni piattaforma supportata. Di seguito vengono presentati gli ambienti di sviluppo di applicazioni native di interesse per questa tesi, vale a dire i due IDE ufficiali di Android e iOS.

AndroidStudio

AndroidStudio [30] è l'IDE ufficiale di Android, sviluppato da Google: la sua prima versione stabile è stata rilasciata nel 2014 ed esso è divenuto l'IDE principale per lo sviluppo nativo su Android, sostituendo gli Android Development Tools (ADT) di Eclipse. AndroidStudio contiene diversi strumenti per lo sviluppo, il test e il debug di un'applicazione: un editor testuale di codice con svariate funzionalità (suggerimenti, autocompletamento, documentazione a portata di click), un editor grafico per il design dell'interfaccia utente (chiamato *Layout Editor*), uno strumento di emulazione, che permette il test dell'app su diversi dispositivi, un debugger e un sistema per il monitoraggio in tempo reale delle prestazioni del device, virtuale o reale, su cui l'applicazione è in esecuzione. Esso è inoltre integrato con diversi VCS (ad esempio Git) e dalla versione 3.0 permette la compilazione di codice scritto, oltre che in Java, anche nel nuovo linguaggio di programmazione *Kotlin* [31], sviluppato da JetBrains (che ha anche sviluppato IntelliJ Platform, su cui AndroidStudio è basato).

XCode

XCode [32] è l'IDE ufficiale per lo sviluppo di applicazioni per tutti i dispositivi Apple, supportandone ogni sistema operativo (macOS, iOS, watchOS e tvOS); esso è, come ci si può immaginare, sviluppato da Apple stessa ed è disponibile solo per macOS. Come AndroidStudio, XCode mette a disposizione del programmatore una moltitudine di strumenti, che facilitano ed accelerano la produzione, il debug e il test delle applicazioni; in particolare, esso contiene un editor grafico molto potente, l'*InterfaceBuilder*, che permette la creazione dell'interfaccia dell'applicazione: è possibile comporre ciascuna schermata dell'applicazione trascinandovi dentro controlli preconfigurati di cui si può ampiamente personalizzare l'aspetto; attraverso alcuni pulsanti dell'InterfaceBuilder è poi possibile posizionare i controlli secondo vincoli specifici definiti dallo sviluppatore oppure utilizzare *Auto Layout*, un sistema di posizionamento che gestisce in automatico la posizione dei controlli in base allo spazio disponibile. Tramite questo editor grafico è inoltre possibile definire parte del comportamento dell'applicazione, ad esempio

collegando un pulsante al codice che deve essere eseguito quando l'utente lo preme oppure definendo quale schermata il sistema debba mostrare dopo che l'utente ha premuto su un elemento di una lista. Sia gli elementi grafici che i comportamenti definibili con l'InterfaceBuilder sono codificati in un file specifico con estensione *.storyboard*; tuttavia, esistono comportamenti e personalizzazioni non definibili con questo editor e che richiedono l'implementazione diretta in codice Swift.

2.2 Obiettivo della tesi

Posto che non sia possibile eguagliare, attraverso lo sviluppo cross-platform ibrido, la qualità di un'applicazione sviluppata nativamente per uno specifico sistema, l'obiettivo di questa tesi è quello di realizzare una soluzione che permetta allo sviluppatore di creare, utilizzando un semplice editor grafico, l'interfaccia dell'applicazione e ottenere, a partire dal prototipo disegnato, una buona parte del codice di questa, in modo automatico: questo processo permetterebbe un notevole risparmio di tempo e, conseguentemente, ridurrebbe i costi di sviluppo di applicazioni native, i quali come già evidenziato sono notevolmente più elevati rispetto a quelli di soluzioni cross-platform (ibride o cross-compilate) o web.

L'approccio della soluzione in esame prende il nome di *model-driven development* e, sebbene sia stato ideato in un contesto generale, è stato successivamente approfondito da diverse ricerche [33] [34] [35] nel caso dello sviluppo di applicazioni per dispositivi mobili. L'idea è quella di dare una descrizione, di alto livello e secondo un modello di dati strutturato, dell'applicazione e poi utilizzare questa descrizione per generare il codice nativo per ciascun sistema; questa fase di generazione prende il nome di *model2text* e può essere preceduta da una fase di trasformazione intermedia del modello, detta *model2model*, dal momento che il modello ottenuto nella prima fase potrebbe non essere adeguato ad una traduzione diretta in codice. L'approccio descritto permette di collegare la fase di prototipizzazione con la fase di sviluppo in codice vero e proprio: anziché realizzare un prototipo al solo scopo di avere un'anteprima grafica dell'applicazione, si sfrutta lo strumento di prototipizzazione per crearne il modello astratto; successivamente si utilizza il file di modello precedentemente creato per generare il codice nativo per ciascun sistema che si vuole supportare attraverso un secondo strumento.

Come si è potuto osservare nella precedente sezione, quasi tutti gli strumenti presi in esame si concentrano solo su una delle due fasi, quella di prototipizzazione oppure quella di sviluppo; Titanium e Ionic offrono un editor per disegnare l'interfaccia dell'applicazione, ma la generazione di codice conseguente è comunque volta ad applicazioni ibride e non native. Esiste un altro strumento, *NativeStudio* [36], che permette la generazione automatica di codice nativo per Android e iOS a partire dal design grafico dell'interfaccia; tuttavia esso è disponibile solo per macOS, costa 199 $ al mese e il codice generato è Java e Objective-C (Swift non è supportato). Un'altra soluzione, già anticipata nel capitolo precedente, è quella offerta dalla toolchain formata da Protocode e MobileCodeGenerator, due

strumenti creati e perfezionati nel corso di lavori di tesi precedenti; essi, grazie a questi lavori, hanno raggiunto ottimi risultati permettendo il design e la generazione automatica del codice di:

- interfacce smartphone realizzate a partire da moltissimi controlli personalizzabili;
- interfacce dedicate alla controparte smartwatch dell'applicazione;
- diversi tipi di modelli utilizzati per memorizzare i dati persistenti dell'applicazione.

Oltretutto, in riferimento a NativeStudio, MobileCodeGenerator è stato aggiornato per la generazione del codice in linguaggio Swift per il sistema iOS, nonostante la sua prima versione fosse antecedente a questo linguaggio e andava quindi a generare codice Objective-C (aggiornamento eseguita dal lavoro di tesi di Aldo Pintus [15]).
La novità introdotta in questa tesi è la possibilità di disegnare interfacce adattative, che rispondano alle esigenze del mercato odierno, in cui i dispositivi mobili si sono enormemente diversificati in termini di dimensioni assolute del display e rapporto tra esse: i vecchi sistemi di posizionamento e dimensionamento, per la maggior parte assoluti o comunque con limitate capacità di porre vincoli relativi, non sono più efficaci nel formulare interfacce utente che mostrino il loro contenuto in modo ottimale su ogni dispositivo. Inoltre, la grande diffusione di device di grandi dimensioni come i tablet ha reso ulteriormente necessaria un'evoluzione dei meccanismi di presentazione dei diversi elementi che compongono la GUI: le librerie di supporto di ciascun sistema operativo devono infatti fornire allo sviluppatore strumenti che permettano non solo il riposizionamento e ridimensionamento dei controlli su dispositivi diversi, ma anche una vera e propria ricombinazione di questi, per far fronte al fatto che un semplice adattamento di posizione e dimensioni non basta ad ottimizzare lo spazio disponibile in display che per i tablet arrivano a dimensioni di oltre 12 pollici mentre per gli smartphone scendono anche sotto i 4. La soluzione qui proposta è partita da uno studio approfondito di questi meccanismi in ciascuna delle due piattaforme di interesse (Android e iOS), arrivando a sviluppare degli strumenti che consentano un design nativo all'avanguardia, sfruttando il più possibile le potenzialità offerte da ciascun sistema operativo; essa sarà approfondita nella successiva sezione.

2.3 Soluzione proposta

Come anticipato nella precedente sezione, la soluzione proposta in questo elaborato di tesi consiste nella modifica ed estensione delle funzionalità già presenti in Protocode e MobileCodeGenerator, per rendere la toolchain formata da questi due software un valido strumento di generazione automatica del codice conforme agli ultimi standard tecnologici. Oltre ad aggiornare i due tool alle ultime versioni di linguaggi di programmazione e librerie software, sono state aggiunte nuove funzionalità per il design di interfacce utente adattative.

Per quanto riguarda Protocode, la prima parte del lavoro si è concentrata nella rielaborazione del sistema di posizionamento e dimensionamento dei controlli nel View Editor: sono stati definiti dei vincoli di posizione, che consentono l'allineamento di un lato o del centro di un controllo a un lato/centro di un altro controllo o della view che lo contiene, e dei vincoli di dimensione, che consentono di fissare larghezza o altezza di un controllo ad un valore preciso oppure ad una frazione della larghezza/altezza della view che lo contiene; è ora inoltre possibile fissare il rapporto larghezza/altezza. Successivamente sono state introdotte le Control Chains (catene di controlli): esse permettono di formare appunto catene di controlli che adattano la visualizzazione di questi ultimi in base allo spazio disponibile. Esse possono essere orizzontali o verticali e vincolano il controllo solo sull'asse della catena (ad esempio un controllo inserito in una catena orizzontale ha la posizione Y libera; essa può essere normalmente fissata dai vincoli di posizione sopra citati); inoltre esse sono di diversi tipi, quattro in particolare: a seconda del tipo di catena lo spazio destinato ai controlli e lo spazio vuoto che li separa sono distribuiti in modi differenti. Il lavoro è poi proseguito andando a ridisegnare il menù dell'applicazione: il vecchio menù a schede (tabs) è stato sostituito dal più moderno Navigation Drawer. Esso è normalmente invisibile (lasciando quindi più spazio ai controlli) e viene richiamato tramite un pulsante che lo mostra come una tendina laterale che compare dal bordo sinistro della finestra dell'applicazione. Dopo questa più piccola modifica, il lavoro è stato volto alla ridefinizione della struttura dell'applicazione modellata: è stato introdotto il concetto di Scena, che rappresenta una schermata dell'applicazione, e che può visualizzare uno o più View Controllers. Il tipo di visualizzazione può essere di tre tipi: a singolo VC, a singolo VC con Tab Bar oppure composito (il modo con cui i View Controllers vengono presentati in scene di ciascuno dei tre tipi verrà approfondito nel capitolo successivo); oltretutto, è possibile differenziare il tipo di visualizzazione su smartphone e su tablet, ad esempio creando una Scena che su smartphone ha una visualizzazione a singolo VC mentre su tablet ha una visualizzazione composita. Questa nuova caratteristica dà all'applicazione prodotta un'ulteriore capacità di adattamento, permettendole di cambiare il suo aspetto a seconda del tipo di dispositivo su cui viene utilizzata; parallelamente a questo, la composizione di View Controllers all'interno di Scene ha permesso di definire porzioni di GUI (appunto i View Controllers) che possono essere riutilizzate in diverse schermate (Scene) dell'applicazione, rendendo più veloce e semplice il design di interfacce contenenti componenti comuni. Come ultima cosa, seguendo l'esempio dei celebri editor di XCode e Android Studio, è stato aggiunto un riquadro di verifica che segnala all'utente se sono riscontrati problemi nell'interfaccia disegnata, come ad esempio se ci sono controlli a cui manca un vincolo di posizionamento su un asse oppure se sono stati definiti dei vincoli non validi. Relativamente a MobileCodeGenerator, poco si aggiunge a quanto detto per Protocode: il tool è stato ampliato per renderlo in grado di generare il codice per tutto ciò che in Protocode è stato aggiunto o modificato; oltre a questo, sono stati aggiornati i riferimenti alle librerie software utilizzate e i file di progetto di ciascuno dei due IDE di sviluppo (Android Studio 3.0.1 e XCode 9.2).

Il risultato così ottenuto è una toolchain di strumenti aggiornati alle ultime tecnologie disponibili che permette la creazione e generazione automatica del codice di interfacce utente complesse ed in grado di adattarsi a dispositivi con display di ogni dimensione, visualizzando i proprio contenuti in modo ottimale su ciascuno di essi. Nei successivi capitoli si andranno a spiegare nel dettaglio le architetture software e le scelte implementative di Protocode e MobileCodeGenerator e ne si farà una valutazione approfondita.

Capitolo 3

Fase di prototipizzazione

3.1 Introduzione

In questo capitolo viene approfondita la fase di prototipizzazione e generazione del modello astratto dell'applicazione. Essa viene effettuata attraverso **Protocode**, seguendo un approccio *model-driven* che consiste nella generazione di un modello indipendente dagli ambienti software per i quali esso verrà poi tradotto in codice nativo (il modello è definito appunto astratto). Protocode è un'applicazione web creata nel 2013 dal lavoro di tesi di Mattia Natali e successivamente estesa e aggiornata da Aldo Pintus e Alessio Rossotti [13] [15] [16]. Essa permette all'utente di comporre l'interfaccia grafica dell'applicazione utilizzando numerosi widgets preconfigurati e consente di modificare diversi parametri di questi, dando all'utente un ampio spazio di personalizzazione. Questo è possibile tramite il View Editor, uno strumento semplice ed intuitivo sviluppato seguendo il paradigma WYSIWYG (What You See Is What You Get), cioè quello che vedi è quello che ottieni. Infatti, a differenza di un normale strumento di prototipizzazione, Protocode andrà a generare un modello XMI dell'applicazione creata, il quale verrà poi utilizzato da MobileCodeGenerator per generare il codice nativo dell'app, il cui risultato grafico sarà esattamente quello mostrato in anteprima da Protocode per ciascuno dei due sistemi operativi Android e iOS. Dalla versione 3.0 Protocode permette inoltre di definire il modello dei dati dell'applicazione attraverso il Model Editor; esattamente come per il View Editor, il Model Editor permette di esportare nel modello XMI le informazioni riguardanti il modello dei dati, successivamente tradotto in codice nativo da MobileCodeGenerator.
Di seguito verranno descritti il framework e le librerie utilizzate per l'implementazione di Protocode, l'architettura iniziale del software da cui si ha cominciato (la versione 3.0) e le modifiche effettuate in questo lavoro di tesi per implementare le nuove funzionalità e aggiornare quelle già esistenti.

3.2 Framework e librerie software utilizzate

3.2.1 Ember.js

Protocode è un'applicazione web basata principalmente su *Ember.js* [37], un framework JavaScript open-source sviluppato seguendo il design-pattern MVVM. Ember permette di creare applicazioni web con funzionalità avanzate, ben oltre quelle che si potrebbero ottenere con pagine HTML decorate da semplici script JavaScript, lasciando l'attenzione dello sviluppatore sui singoli componenti: il codice necessario a connettere tali componenti (*boilerplate code*) è infatti generato automaticamente da Ember sulla base di *best practices* e convenzioni basate sui nomi. Il framework è corredato da *Ember Data*, una libreria che permette la definizione del modello dei dati e il salvataggio di essi direttamente sul client, utilizzando il *local storage* del browser.
I componenti principali che compongono un'applicazione Ember sono:

- **Template**: descrive una porzione di interfaccia grafica dell'applicazione utilizzando il linguaggio HTML decorato con espressioni JavaScript. Ogni Route (per rappresentare un Model) e View hanno un file di Template, memorizzato con l'estensione .hbs (*handlebars*); tipicamente i Template delle Route definiscono la maggior parte dell'interfaccia in una determinata schermata dell'applicazione, mentre i Template delle View vanno a completare quelli delle Route, aggiungendo elementi più specifici. La totalità di questi file costituisce l'intera interfaccia grafica dell'applicazione.

- **View**: è un componente che serve a definire una parte di interfaccia che andrà a interagire con l'utente in modo più specifico e complesso rispetto al resto della GUI. Le View sono costituite da un file di Template (.hbs), che ne descrive l'aspetto grafico, e/o un file di script (.js) che ne specifica il comportamento in base all'interazione con l'utente.

- **Router**: è il componente centrale dell'applicazione e serve a coordinare la presentazione dei Template. Il Router mappa l'indirizzo web richiesto dal browser (URI) alla corrispondente Route, simulando la navigazione: non avviene infatti un vero caricamento di pagina, ma i Template sono sostituiti nell'unica pagina dell'applicazione (Single Page Application) per mostrarne le diverse schermate.

- **Model**: serve a definire, attraverso un file di script (.js), un modello di dato che sarà usato dall'applicazione. Esso contiene proprietà *stored*, che saranno memorizzate in modo persistente, e proprietà *computed*, che vengono calcolate dinamicamente a runtime. L'insieme di tutti i file di Model costituisce il layer di dati dell'applicazione.

- **Route**: è il componente che serve a collegare i Template ai modelli che devono mostrare e può mantenere in memoria dati relativi allo stato corrente dell'applicazione (questi dati non saranno salvati in modo persistente). E' definito da un file di script (.js) che, se non viene specificato dal programmatore, viene automaticamente generato da Ember per ogni coppia Model/Template i cui nomi siano definiti secondo una specifica convenzione.

- **Controller**: è uno script (file .js) che serve ad implementare alcuni comportamenti specifici dell'applicazione, in particolare quando avviene una modifica da parte dell'utente dei dati del Model. Come gli script di Route può mantenere in memoria dati transitori, che non saranno salvati in memoria.

3.2.2 Yeoman.io

Yeoman.io [38] è uno strumento che serve a facilitare lo sviluppo di applicazioni web. Esso è molto utile per iniziare un progetto da zero, in quanto fornisce un ecosistema di tools che permettono al programmatore di risparmiare tempo nell'installazione delle librerie e framework necessari e nella configurazione della fase di *build* dell'applicazione; inoltre supporta un'infinità di framework, per cui è possibile utilizzarlo in quasi qualsiasi tipo di progetto. Yeoman è composto principalmente da tre componenti:

- uno strumento di *scaffolding*, **Yo**, che genera automaticamente lo scheletro dell'applicazione, la configurazione di build e i task per risolvere le dipendenze delle librerie necessarie.

- Un tool di *build*, che serve a gestire automaticamente la compilazione dell'applicazione, la creazione ed il lancio di un server con essa installata e la creazione di una versione compressa dell'applicazione pronta alla distribuzione; per Protocode è stato scelto il popolare build system **Grunt** [39], ma altre scelte sono possibili.

- Un *package manager* per la gestione delle dipendenze, che esegue il download automatico dei pacchetti e librerie utilizzate per lo sviluppo dell'applicazione. Attraverso un file di configurazione, è possibile specificare la versione delle librerie che si vuole utilizzare in modo che il package manager ne scarichi la giusta versione; è anche possibile controllare e scaricare gli ultimi aggiornamenti disponibili di ciascuna libreria utilizzata. Come per il build system, anche per il package manager esistono diverse scelte; per Protocode si è optato per **Bower** [40].

In aggiunta ai tre tool appena descritti, è stato integrato **Compass** [41], un framework CSS per la gestione dei fogli di stile. Compass utilizza l'estensione Sass [42] del CSS3 (file .scss), che permette l'utilizzo di variabili e la definizione di classi di stile nidificate.

3.2.3 Bootstrap

Bootstrap [43] è il più famoso framework per lo sviluppo dell'interfaccia grafica di siti e applicazioni web; è basato su HTML, CSS e JavaScript e permette la creazione di interfacce *responsive*, cioè che adattano il proprio contenuto alle dimensioni della finestra in cui si trovano (per questo è anche ampiamente utilizzato in progetti web orientati ai dispositivi mobili). Bootstrap permette al programmatore di risparmiare moltissimo tempo nella definizione dell'interfaccia, fornendo una moltitudine di componenti (predefiniti e pronti all'uso) comunemente presenti nella GUI di applicazioni web come bottoni, pannelli, form e così via. Esso è stato aggiunto in Protocode tramite Bower.

3.2.4 Altre librerie

Oltre alle principali librerie e framework appena descritti, per la realizzazione di Protocode sono stati utilizzati i seguenti componenti software (aggiunti al progetto tramite Bower):

- **Ember Data Local Storage Adapter**, una libreria che permette il salvataggio dei dati di Model di Ember nel *local storage* del browser [44];
- **JQuery**, un framework JavaScript che permette la gestione di aspetti grafici, eventi, animazioni e manipolazioni dell'interfaccia [45]; è alla base dell'implementazione di Bootstrap.
- **Font Awesome**, un toolkit contenente una moltitudine di icone pensate per il web;
- **Blob e File Saver**, due librerie JavaScript che consentono il salvataggio di dati lato client. Sono utilizzate in Protocode per il salvataggio del file di modello (.xmi) dell'applicazione [47] [48];
- **vkBeautify**, una libreria che permette di indentare automaticamente il codice di file scritti in XML, JSON, CSS e SQL. In Protocode è utilizzata per formattare correttamente il file XMI di modello.

3.3 Applicazione di partenza: Protocode 3.0

In questa sezione viene presentato il software di partenza della prima parte di questo lavoro di tesi, Protocode nella versione 3.0. Ne saranno di seguito descritti l'architettura, i componenti e l'interfaccia grafica.

3.3.1 Architettura software

La struttura gerarchica delle directory del progetto è la seguente:

- **root**: cartella principale dell'applicazione, contiene tutte le altre directory e i file di configurazione di Bower e Grunt. I file di Bower contengono la lista delle dipendenze, ovvero le librerie software (con specificata la versione) che Bower andrà a scaricare; i file di Grunt contengono i task che saranno eseguiti per il deployment dell'applicazione sulla macchina locale (*localhost*) e i task per la distribuzione.

- **node_modules**: in questa cartella sono contenute le librerie utilizzate da tutti gli applicativi basati su *Node.js* [50], come Bower e Grunt.

- **test**: questa directory contiene i file per il testing dell'applicazione.

- **dist**: in questa cartella è contenuta la versione dell'applicazione destinata alla distribuzione; essa è generata automaticamente da Grunt con il comando `grunt build`. In questa versione della web application i file di script e i fogli di stile sono compattati, rimuovendo tutto il codice commentato, unificati e codificati: questo garantisce maggiori prestazioni e impedisce che il codice possa essere visualizzato tramite la console web del browser.

- **app**: questa cartella contiene l'applicazione vera e propria; essa è suddivisa in ulteriori 5 directory: *bower_components* contiene le librerie utilizzate dall'app e scaricate da Bower; *img* contiene tutte le immagini; *scripts* contiene tutti i file di script, suddivisi in components, controllers, mixins, models, routes e views; *styles* contiene tutti i fogli di stile; *templates* infine contiene tutti i file di template (.hbs).

L'intera struttura appena descritta è riassunta dall'immagine 3.1; di seguito si andranno a descrivere nel dettaglio tutti i componenti dell'applicazione vera e propria che, come appena mostrato, sono contenuti nella cartella *app*.

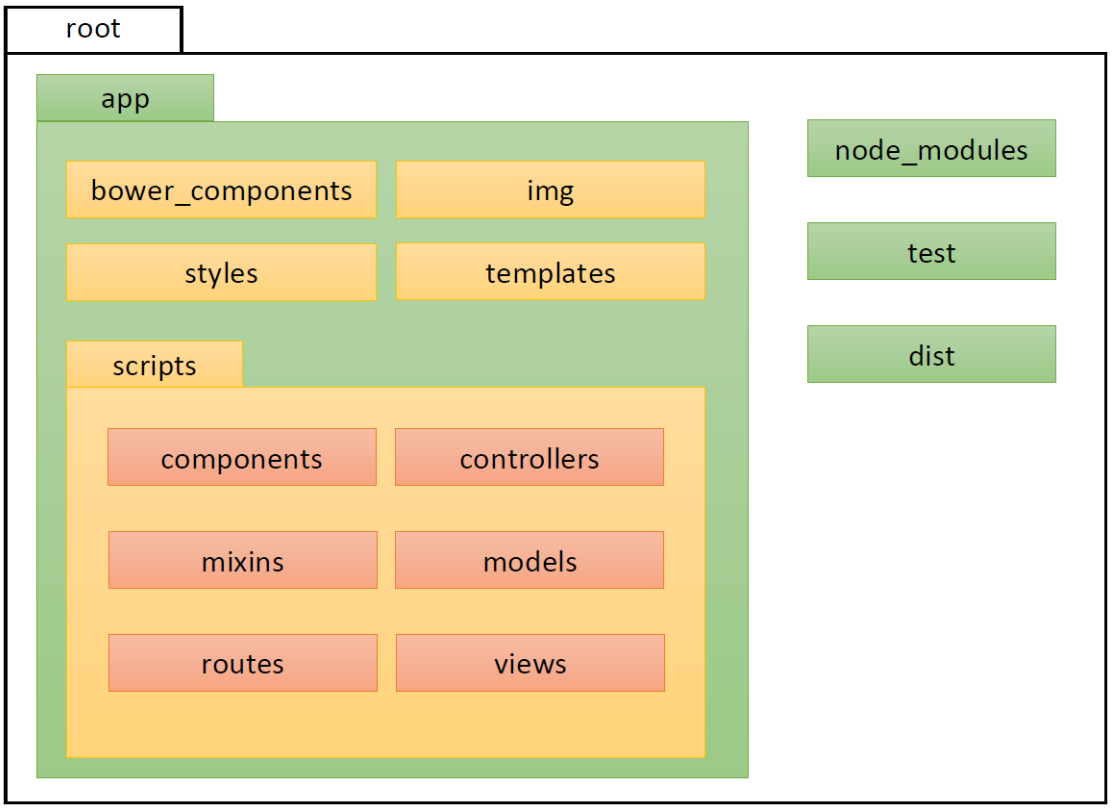

Figura 3.1: Struttura di Protocode

3.3.1.1 Routing

Il Routing consiste nella mappatura dell'URI che viene richiesto al browser dall'utente (cliccando i link presenti nell'interfaccia dell'applicazione) nell'effettiva pagina che l'app visualizza; questa relazione è definita nel *Router* (*app/scripts/router.js*), il componente tramite il quale il programmatore specifica quale *route* mostrare ad ogni indirizzo (i file di script delle route si trovano nella directory *app/scripts/routes* mentre i rispettivi template si trovano in *app/template*). Ogni route va a visualizzare un file di model, che può essere specificato dal programmatore o implicito: in quest'ultimo caso Ember seleziona automaticamente il model con lo stesso nome della route corrente. Ad ogni route è inoltre associato un controller, che ancora una volta può essere definito dal programmatore (nel caso debba svolgere compiti specifici) oppure generato automaticamente da Ember. In figura 3.2 viene riportato uno schema delle route di Protocode 3.0; per brevità non sono specificate tutte le route figlie della route ViewController.

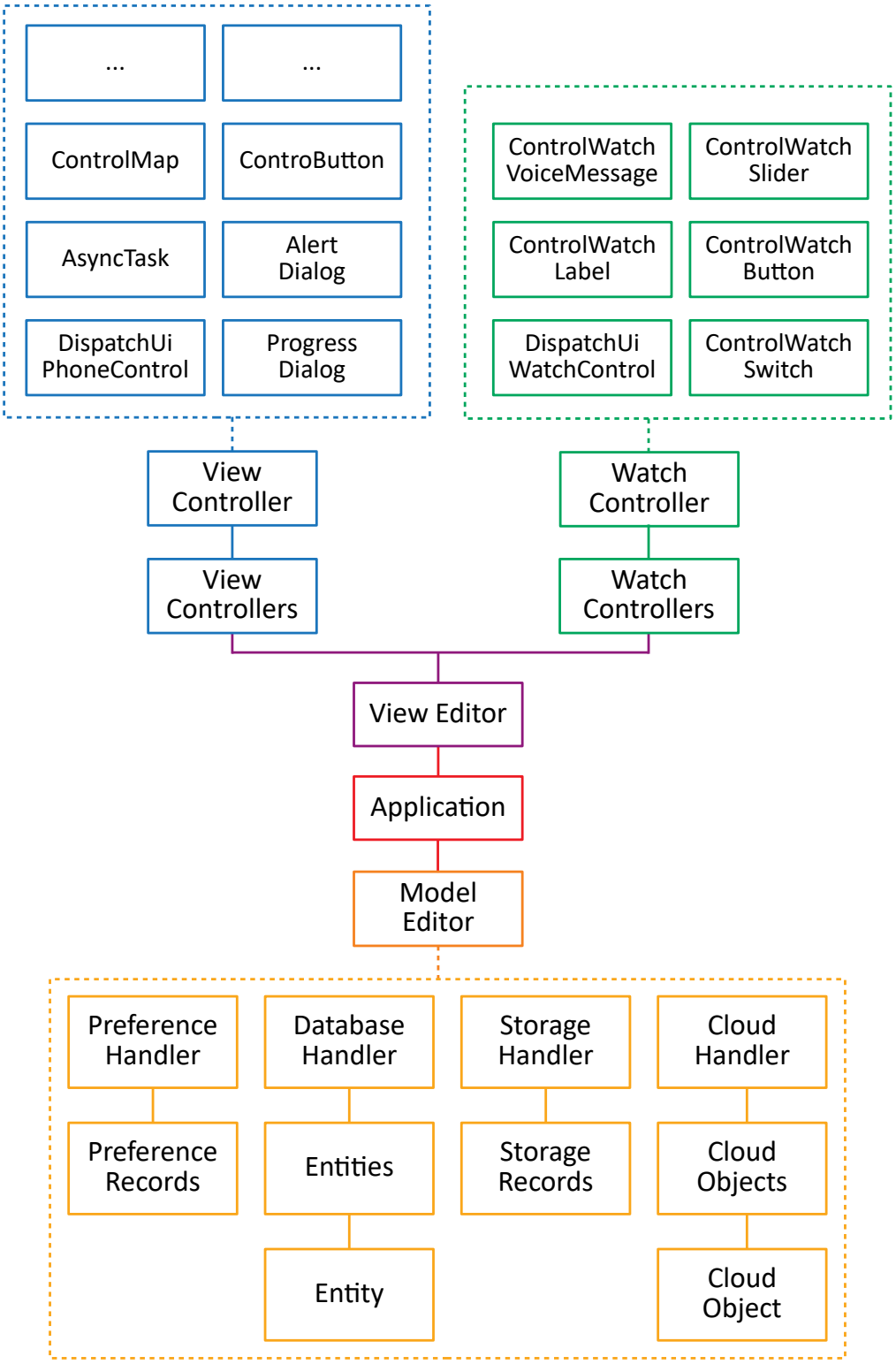

Figura 3.2: Route di Protocode 3.0

3.3.1.2 Mixin

I *mixin* sono componenti di Ember che permettono di definire proprietà e metodi riutilizzabili in diversi altri componenti. In Protocode sono stati creati e utilizzati i seguenti mixin:

- **saveable**: serve a gestire le comuni operazioni di salvataggio delle modifiche apportate a un'istanza di model da parte dell'utente;
- **deletable**: serve a gestire la cancellazione di un'istanza di model;
- **navigable**: è utilizzato per definire i metodi necessari ad impostare o resettare la destinazione per quei controlli che sono associati ad un oggetto Navigation;
- **uiDroppable**: serve a definire il comportamento di *drag & drop* di un UiPhoneControl (un widget definito per l'interfaccia di un'app su smartphone, ad esempio un bottone o un'etichetta);
- **uiMoveable**: è utilizzato per rendere spostabile un UiPhoneControl dell'interfaccia dell'applicazione che si sta disegnando;
- **uiWatchDroppable**: come uiDroppable ma definito per i controlli degli smartwatch (UiWatchControl);
- **uiWatchMoveable**: come uiMoveable ma definito per gli UiWatchControl;
- **textAlignable**: è utilizzato per gestire l'allineamento del testo in quei controlli che ne contengono;
- **textDecorable**: simile a textAlignable, ma serve a modificare l'aspetto del testo (normale, grassetto, corsivo);
- **withSourceType**: serve a definire il tipo di sorgente per i controlli che fanno un qualche tipo di rendering dei dati (esempio ImageView, che fa il rendering di un'immagine);
- **watchClickListenable**: è utilizzato per definire il comportamento di un UiWatchControl su cui è possibile cliccare.

3.3.1.3 Controller

Come già spiegato nella sezione 3.2.1, i controller sono componenti che si occupano di gestire dati temporanei e interazioni con l'utente; in Protocode si trovano nella cartella *app/scripts/controllers* e sono stati utilizzati per gestire quasi tutti gli eventi generati dall'interazione con l'utente, ad eccezione di quelli relativi allo spostamento dei widget, che sono gestiti dalle view (vedere sezione successiva). Per svolgere la loro funzione, alcuni controller utilizzano i mixin presentati nella sezione precedente: ad esempio, i controller definiti per le route di UiPhoneControl e UiWatchControl utilizzano i mixin *saveable* e *deletable* per gestire il salvataggio delle proprietà dei controlli in caso di modifica e la loro cancellazione.

I controller però svolgono moltissime altre funzioni oltre a quelle definite all'interno dei mixin, come il controllo del posizionamento dei widget, l'aggiornamento di alcune parti dei template e la generazione del file di model (.xmi).

3.3.1.4 View

Le view, in Protocode, sono state utilizzate per definire l'aspetto e il comportamento dei widget utilizzati per comporre l'interfaccia grafica dell'applicazione. Esse, com'è già stato detto, constano di un file di script (.js) che ne definisce il comportamento e un file di template (.hbs) che ne specifica l'aspetto. I file JavaScript delle view di Protocode utilizzano i mixin *uiDroppable* e *uiMoveable*, che consentono rispettivamente di aggiungere un widget all'interfaccia e di spostarlo; questi file sono localizzati nella directory *app/scripts/views*. I file *handlebars*, vale a dire i template, si trovano invece nella cartella *app/templates/views*.
In Ember esistono anche alcuni componenti chiamati proprio *components* che assomigliano alle view ma sono più sofisticati (sono definiti *Glorified Views* dagli sviluppatori di Ember stessi) e permettono un'interazione ancor più complessa; in Protocode ne sono stati realizzati alcuni (ad esempio il ColorPicker, un componente che mostra una finestra dove è possibile selezionare un colore avendone un'anteprima) e si trovano nelle cartelle *app/scripts/components* e *app/templates/components* (come le view sono anch'essi composti da un file di script e uno di template).

3.3.1.5 Model

I file di model sono gli script che modellano i dati di Protocode che saranno salvati persistentemente; alcuni di essi saranno utilizzati al momento della generazione del modello dell'applicazione, come ad esempio i model relativi a widget e view controller, mentre altri servono solamente al funzionamento di Protocode, come ad esempio i model che rappresentano i diversi dispositivi su cui è possibile fare il design dell'applicazione. A livello di codice, i model necessari alla generazione del file XMI si differiscono dagli altri per la presenza di un metodo `toXml()` che va proprio a mappare le proprietà dell'istanza di model in un elemento XML. I file di model si trovano nella directory *app/scripts/models*.

3.3.2 Interfaccia grafica

In questa sezione viene presentata l'interfaccia grafica della versione di Protocode da cui questo lavoro di tesi è cominciato. Essa contiene inizialmente solo due pulsanti: il pulsante *About*, che visualizza una schermata contenente le informazioni e i contatti degli sviluppatori, e il pulsante *Create App* che va a creare un'istanza vuota di una nuova applicazione e preparare Protocode per il suo sviluppo. Dopo la creazione dell'app (o alla riapertura del browser successiva alla creazione) l'interfaccia mostra due ulteriori pulsanti, *View Editor* e *Model Editor*: il primo porta alla schermata per il design dell'interfaccia dell'applicazione, il secondo ne mostra l'editor della componente Model (dati).

3.3.2.1 View Editor

Il View Editor (fig. 3.3) si presenta sostanzialmente immutato dalla versione 2.0 ed è composto dai seguenti componenti:

- una barra (1) contenente il nome dell'applicazione, l'identificativo del team di sviluppo e il pulsante per la creazione del file di modello (.xmi) dell'applicazione (presente anche nel Model Editor);
- una barra (2) che permette la selezione del modello di dispositivo su cui fare il design della GUI dell'app e visualizzarne l'anteprima (la selezione avviene tramite due menù a opzioni, uno per i dispositivi di tipo smartphone e l'altro per quelli di tipo smartwatch);
- un pannello (3) che permette, scegliendo attraverso due pulsanti, di visualizzare l'elenco di tutti i view controller e i watch controller creati;
- un pannello (4) che visualizza l'anteprima dell'applicazione sul dispositivo scelto tramite la barra (2);
- un pannello (5) che mostra e consente la modifica delle proprietà dell'oggetto attualmente selezionato;
- una palette (6) di widget che possono essere aggiunti all'interfaccia dell'app mediante trascinamento e rilascio (*drag & drop*).

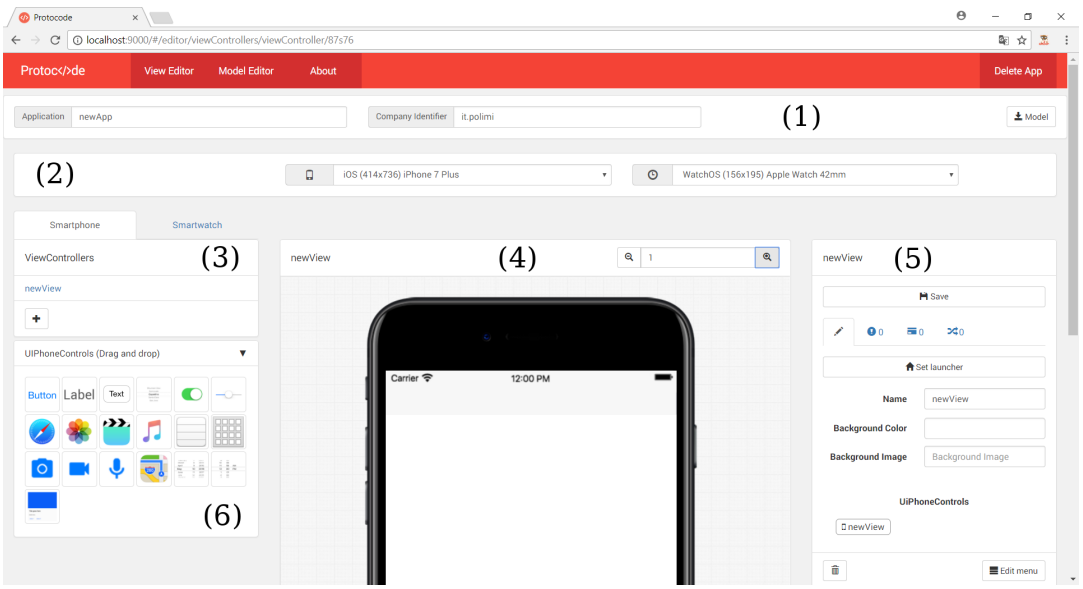

Figura 3.3: Interfaccia del View Editor di Protocode 3.0

3.3.2.2 Model Editor

Il Model Editor (fig. 3.4) è stato introdotto in Protocode 3.0 ed è composto dalla stessa barra (1) presente nel View Editor, da un pannello laterale (2) che presenta i quattro possibili tipi di dato modellabili e un pannello (3) che, una volta selezionato un tipo di dato dal (2), permette il design dei dati di quel tipo.

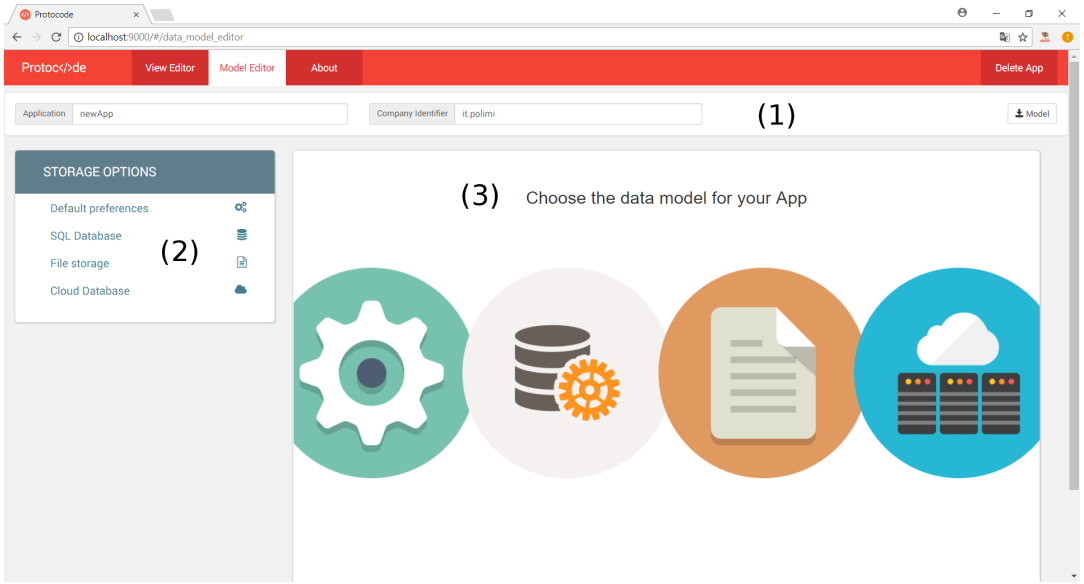

Figura 3.4: Interfaccia del Model Editor di Protocode 3.0

3.4 Aggiornamento a Protocode 4.0

Dopo aver descritto Protocode così come si presentava all'inizio di questo lavoro di tesi (versione 3.0), saranno di seguito approfondite tutte le funzionalità aggiunte e/o modificate al fine di risolvere il problema della *adaptivity* dell'interfaccia grafica a diversi dispositivi, che hanno portato all'implementazione di Protocode 4.0.

3.4.1 Vincoli di posizione

Il primo intervento su Protocode è consistito nel potenziamento del sistema di posizionamento dei controlli: esso prendeva spunto dai vincoli presenti nel RelativeLayout di Android che permettono un posizionamento, relativo appunto, di un controllo rispetto ad un altro o alla finestra che lo contiene. Vincoli di questo tipo sono facilmente tradotti in codice per Android, essendo basati sui concetti di RelativeLayout, e anche in iOS, essendo in quest'ultimo presenti vincoli molto più potenti e complessi; tuttavia questo tipo di vincoli risulta troppo semplice per permettere all'interfaccia creata con essi di adattarsi a display di ogni dimensione visualizzando sempre al meglio il proprio contenuto.
Per questo motivo è stata introdotta la classe di model **Constraint**, che permette di sfruttare le maggiori potenzialità del ConstraintLayout di Android [51]; essa è relativa ad uno specifico controllo (UiPhoneControl) e permette di specificare quanto segue:

- *layoutEdge*, il lato da vincolare del controllo (è una stringa che può assumere i valori di `top`, `bottom`, `start`, `end`, `centerX` e `centerY`);
- *withParent*, un booleano che specifica se il riferimento del vincolo è la finestra padre del controllo oppure un altro UiPhoneControl;
- *referenceElement*, specificabile solo se `withParent` è falso, rappresenta il controllo che funge da riferimento posizionale al primo;
- *referenceLayoutEdge*, il lato da vincolare del controllo di riferimento; può assumere un valore appartenente ad un sottoinsieme dei valori assumibili da `layoutEdge`, a seconda del valore dato ai precedenti parametri: ad esempio, se si va a vincolare il `top` (lato superiore) di un controllo, `referenceLayoutEdge` potrà in generale assumere solo i valori `top` e `bottom`, dal momento che non avrebbe senso vincolare un lato "verticale" ad uno "orizzontale" (è escluso dai valori assumibili da `referenceLayoutEdge` anche `centerY` dal momento che nel ConstraintLayout di Android è consentito vincolare il centro di un controllo solo al centro di un altro controllo o della finestra padre).

Dopo la definizione di questa classe di model, sono stati effettuati tutti i cambiamenti per permetterne l'utilizzo: per prima cosa sono stati rimossi dai model dei controlli i vecchi vincoli; poi sono stati aggiornati i template degli UiPhoneControl per permettere la creazione e la modifica dei vincoli nuovi; sono stati quindi aggiornati i metodi che calcolano la posizione di un controllo, basando questo calcolo

sui nuovi vincoli introdotti; infine è stato implementato un algoritmo per il controllo della validità dei vincoli creati dall'utente. In figura 3.5 è possibile osservare i template di Protocode 4.0 per la creazione e modifica dei vincoli di posizione.

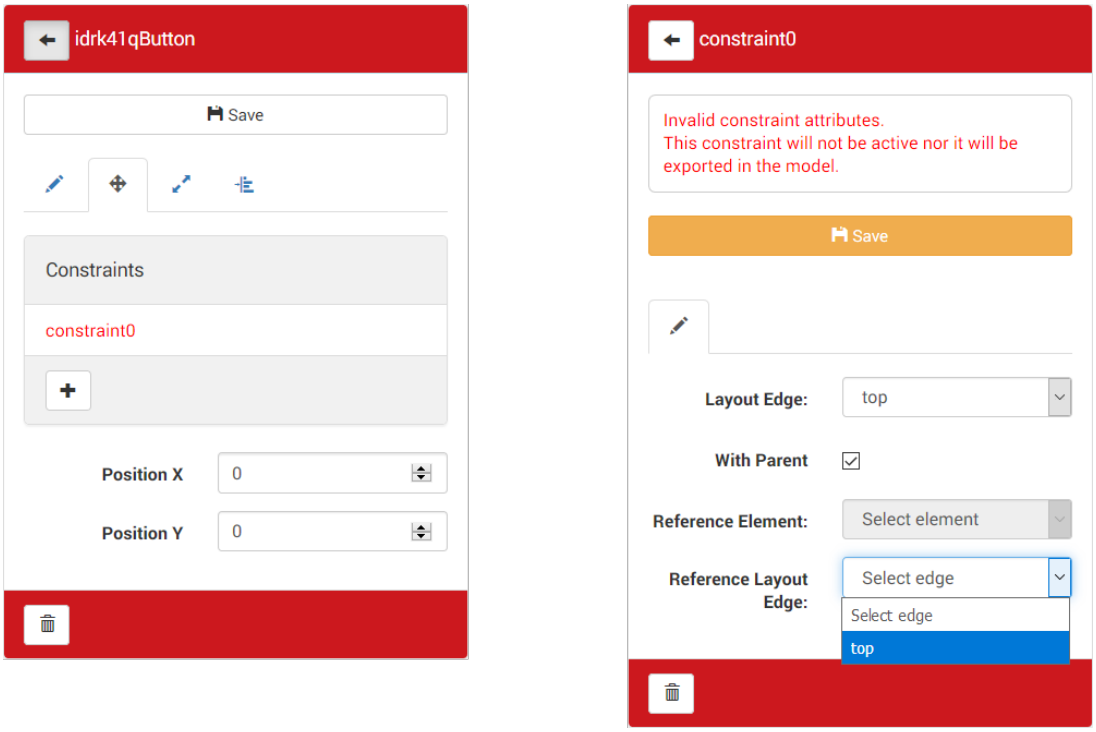

Figura 3.5: Interfaccia per la creazione e modifica di vincoli di posizione

3.4.2 Vincoli di dimensione

Oltre al posizionamento dei controlli, sono state successivamente potenziate le possibilità offerte da Protocode nella specifica della dimensione di questi. Inizialmente, Protocode consentiva solo di specificare una dimensione fissa, espressa in *dp* (density-independent pixels); questo tipo di dimensionamento tuttavia, nonostante sia talvolta necessario, non consente nessuna capacità di adattamento alle dimensioni dello schermo. Al fine di rendere *adaptive* anche le dimensioni dei controlli sono quindi stati implementati dei nuovi vincoli orientati al dimensionamento; a differenza di quanto fatto per quelli di posizione, per i quali è stata creata una nuova classe di model, si è qui preferito aggiungere alcuni attributi direttamente alla superclasse UiPhoneControl (che viene estesa poi da ciascuna classe che rappresenta uno specifico controllo, come per esempio la classe Button). Di seguito sono riportati i vincoli imponibili e per ciascuno di essi sono descritte tutte le proprietà aggiunte e la relativa funzione:

- vincolo **fisso**: le dimensioni del controllo sono impostate a un valore preciso, come era possibile fare in Protocode 3.0; le proprietà aggiunte sono:
  - *isWidthConstrained* e *isHeightConstrained*, booleani che specificano se sono attivi vincoli fissi di dimensione rispettivamente su larghezza e altezza;

  - *widthFixed* e *heightFixed*, numeri a virgola mobile che specificano le dimensioni da imporre rispettivamente a larghezza e altezza (sono effettivamente imposti solo se i due precedenti attributi booleani sono veri).

- Vincolo **percentuale**: le dimensioni del controllo sono calcolate sulla base delle dimensioni della finestra padre che lo contiene; le proprietà che lo modellano sono:

  - *isWidthPercentConstrained* e *isHeightPercentConstrained*, booleani che specificano se sono attivi vincoli percentuali di dimensione rispettivamente su larghezza e altezza (simili a quelli definiti per i vincoli fissi);
  - *widthPercent* e *heightPercent*, numeri a virgola mobile, compresi tra 0 e 1, che specificano le dimensioni da imporre rispettivamente a larghezza e altezza. La dimensione imposta è il prodotto tra dimensione della finestra padre e valore percentuale: ad esempio, specificando `widthPercent = 0.5` viene imposta al controllo una larghezza uguale alla metà di quella della finestra (a patto che, come vale per i vincoli fissi, `isWidthPercentConstrained` sia vero).

- Vincolo di **rapporto**: è qui imposta una specifica proporzione tra la larghezza e l'altezza del controllo, attraverso i seguenti attributi:

  - *isRatioConstrained*, booleano che ha la stessa funzione di `isWidthConstrained` e gli altri;
  - *ratioWidth*, numero che specifica il "peso" dato alla larghezza;
  - *ratioHeight*, numero che specifica il "peso" dato all'altezza.

  Come già visto per i vincoli fissi e percentuali, un vincolo di rapporto è imposto solo se `isRatioConstrained` è vero; il valore dato a ciascuna dimensione dipende da `ratioWidth`, `ratioHeight` e da altri vincoli: se ad esempio la larghezza fosse già vincolata in altro modo, la dimensione dell'altezza verrebbe adeguata secondo il rapporto specificato.

- Dimensione **predefinita**: sono stati aggiunti due ulteriori attributi alla classe UiPhoneControl, *defaultWidth* e *defaultHeight*. Essi sono utilizzati per il dimensionamento del controllo qualora nessun altro vincolo sia stato definito.

Allo stesso modo di quanto fatto per i vincoli di posizione, una volta modificata la classe di model dei controlli sono stati eseguite tutte le modifiche necessarie al loro utilizzo. Ciò ha consistito nell'aggiornamento dei template dei controlli e nell'implementazione di un metodo per il calcolo della dimensione a partire dai vincoli definiti. Sono state anche definite, nei componenti controller di Ember, una serie di proprietà che attivano e disattivano la possibilità da parte dell'utente di impostare e modificare i vincoli: ad esempio, se è stato impostato un vincolo fisso sulla larghezza non è possibile imporre un vincolo percentuale sulla stessa;

un altro esempio: se sono già state vincolate sia larghezza che altezza con vincoli fissi o percentuali, non è possibile impostare un vincolo di rapporto; un ultimo esempio: se i lati `start` e `end` di un controllo sono stati vincolati con constraint di posizione, non è possibile imporre un vincolo di dimensione, fisso, percentuale o di rapporto, alla larghezza del controllo (allo stesso modo, se fossero stati vincolati `top` e `bottom` non sarebbe stato possibile imporre l'altezza). Il template per la modifica dei vincoli di dimensione è riportato in figura 3.6.

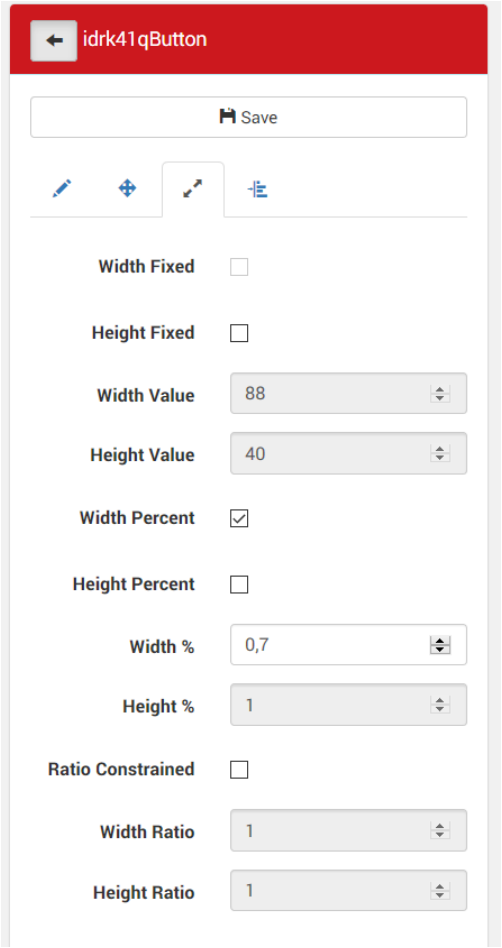

Figura 3.6: Interfaccia per la creazione e modifica di vincoli di dimensione

3.4.3 Control Chain

I vincoli di posizione introdotti nella sezione 3.4.1 hanno aumentato le possibilità offerte da Protocode per il posizionamento dei controlli; essi sono stati sviluppati seguendo i vincoli presenti nel ConstraintLayout di Android [51], meno potenti e semanticamente meno ricchi di quelli presenti in iOS. Questa scelta ovviamente deriva dal fatto che è possibile tradurre questi vincoli in entrambi i sistemi, mentre se si fosse tenuto a modello il tipo di vincoli presenti in iOS sarebbe poi stato impossibile tradurre questi in codice per Android. ConstraintLayout tuttavia offre un altro meccanismo di posizionamento, non presente in iOS, che va a completare le possibilità offerte dai normali vincoli ottenendo, sostanzialmente, gli stessi

risultati che si possono ottenere utilizzando i più potenti vincoli presenti in iOS. Il meccanismo in questione prende il nome di **Control Chain**, letteralmente catene di controlli: esso permette infatti di raggruppare i controlli in catene che, a seconda dell'asse su cui sono definite e dal tipo di catena scelto, distribuiscono i controlli nella spazio in modi differenti. Dallo studio fatto in questo lavoro di tesi è emerso che è possibile tradurre il posizionamento ottenuto con le Control Chain di Android in un'insieme di vincoli di iOS ottenendo il medesimo risultato; si è quindi deciso di implementare questo meccanismo in Protocode, al fine di aumentare ulteriormente le possibilità offerte dal sistema di posizionamento dei controlli.
Come già accennato, i due parametri principali di una Control Chain sono l'asse e il tipo. L'*asse* della catena può essere orizzontale o verticale e la catena vincola i controlli che ne fanno parte solo sull'asse sul quale è definita: un controllo appartenente ad una catena orizzontale, ad esempio, ha la propria coordinata Y libera; essa può essere imposta normalmente, utilizzando i vincoli di posizione già presentati. Il *tipo* di catena va a specificare in che modo lo spazio a disposizione va assegnato ai diversi controlli che fanno parte della catena e allo spazio che li separa. In ConstraintLayout sono presenti quattro tipi di catene; essi sono riportati di seguito, con la rispettiva spiegazione:

- tipo **spread**: una catena di questo tipo assegna a ciascun controllo lo spazio che gli è stato specificato attraverso vincoli di dimensione e distribuisce equamente lo spazio restante per separare i controlli; ad esempio, un catena *spread* orizzontale contenente due bottoni distribuisce lo spazio libero restante in tre parti, una che separa il primo bottone dal bordo sinistro della finestra, uno che separa il primo bottone dal secondo e uno che separa il secondo bottone dal bordo destro della finestra.

- Tipo **spread inside**: una catena di questo tipo si comporta come una di tipo *spread*, a differenza del fatto che distribuisce lo spazio libero (non occupato dai controlli) solamente in mezzo ai controlli; nell'esempio riportato per il tipo spread, una catena *spread inside* allineerebbe il primo bottone al margine sinistro della finestra, il secondo bottone al margine destro e utilizzerebbe quindi tutto lo spazio libero per separare i due bottoni.

- Tipo **packed**: una catena *packed* assegna a ciascun controllo lo stesso spazio che gli verrebbe assegnato da catene *spread* e *spread inside*; a differenza di queste però allinea i controlli al centro della finestra, utilizzando lo spazio libero come margine sinistro e destro (per catene orizzontali) o superiore e inferiore (per catene verticali) dalla finestra. E' possibile personalizzare la posizione dei controlli in questo tipo di catena con un attributo *bias*, un numero con la virgola compreso tra 0 e 1 che indica come distribuire lo spazio precedente ai controlli e quello successivo: ad esempio, un valore di 0.3 farebbe sì che lo spazio precedente ai controlli sia il 30% dello spazio libero totale, con un conseguente 70% di spazio libero riservato allo spazio successivo ai controlli; il valore predefinito di bias è 0.5, cioè uguale spazio prima e dopo ai controlli.

- Tipo **weighted**: questo tipo di catena, a differenza dei precedenti, assegna tutto lo spazio disponibile ai controlli (ad eccezione di eventuali margini definiti dagli stessi). A ciascun controllo viene assegnato un peso (*weight* appunto) e lo spazio disponibile viene distribuito ai controlli sulla base di esso: ad esempio, se un controllo dovesse avere un peso pari a 2 mentre un secondo controllo avesse un peso pari a 3, la dimensione del primo sarebbe uguale ai due terzi della dimensione del secondo.

La figura 3.7 mostra un riassunto di tutti i tipi di catena.

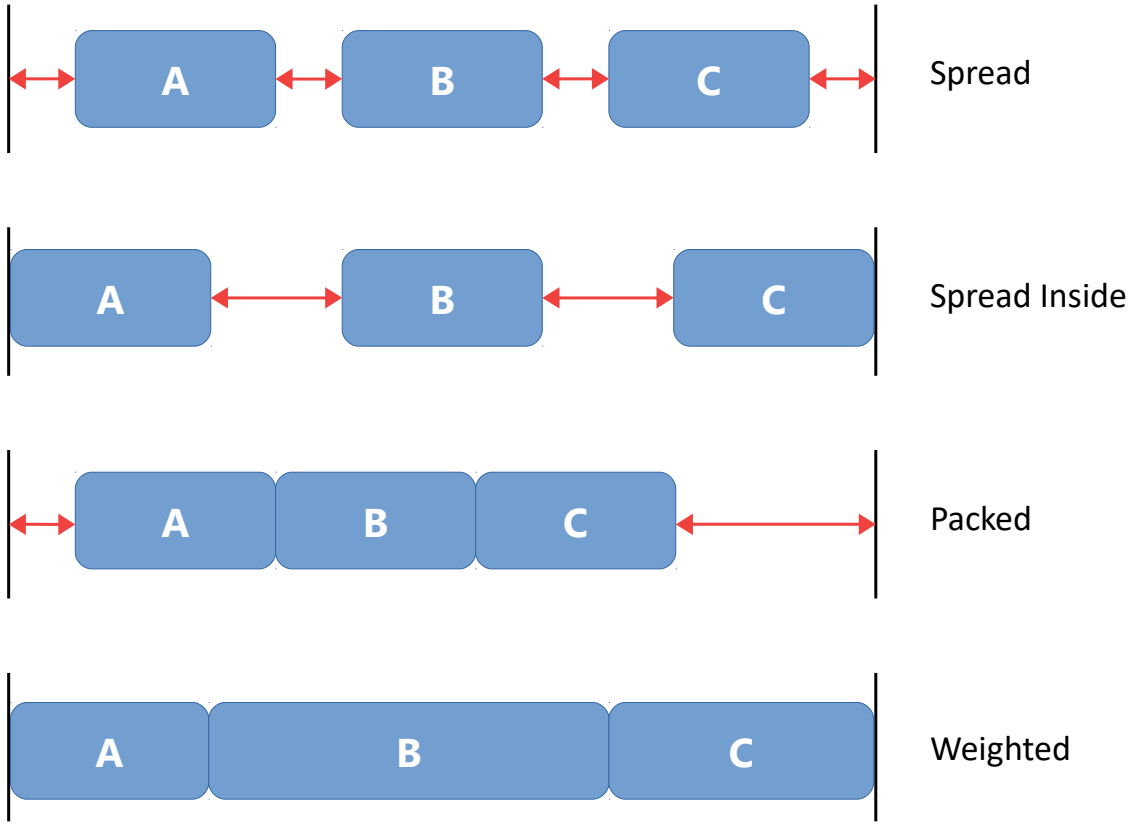

Figura 3.7: Tipi di Control Chain

Tutti questi tipi di catena sono stati implementati in Protocode attraverso la definizione di una classe di model, chiamata ControlChain; gli attributi principali di questa classe sono *axis*, *type*, *bias* e *spacing*. Riguardo ai primi tre attributi, non c'è nulla da aggiungere a quanto appena descritto (bias è definito solo per le catene di tipo *packed*); per quanto riguarda invece l'attributo *spacing* (spaziatura), esso può essere definito per catene di tipo *packed* e *weighted* e serve ad indicare la dimensione (in dp) dello spazio che separa i controlli (il valore predefinito è zero).
Dopo aver definito la classe di model ControlChain, sono stati creati e aggiornati i template che ne permettono la creazione e modifica; successivamente è stato aggiornato l'algoritmo (implementato nella superclasse UiPhoneControl) che calcola la posizione del controllo, per tenere in considerazione il caso in cui quest'ultimo appartenga ad una catena. In figura 3.8 sono riportati i template che consentono

la modifica di una Control Chain; la creazione è invece permessa attraverso un pulsante aggiunto al template del view controller.

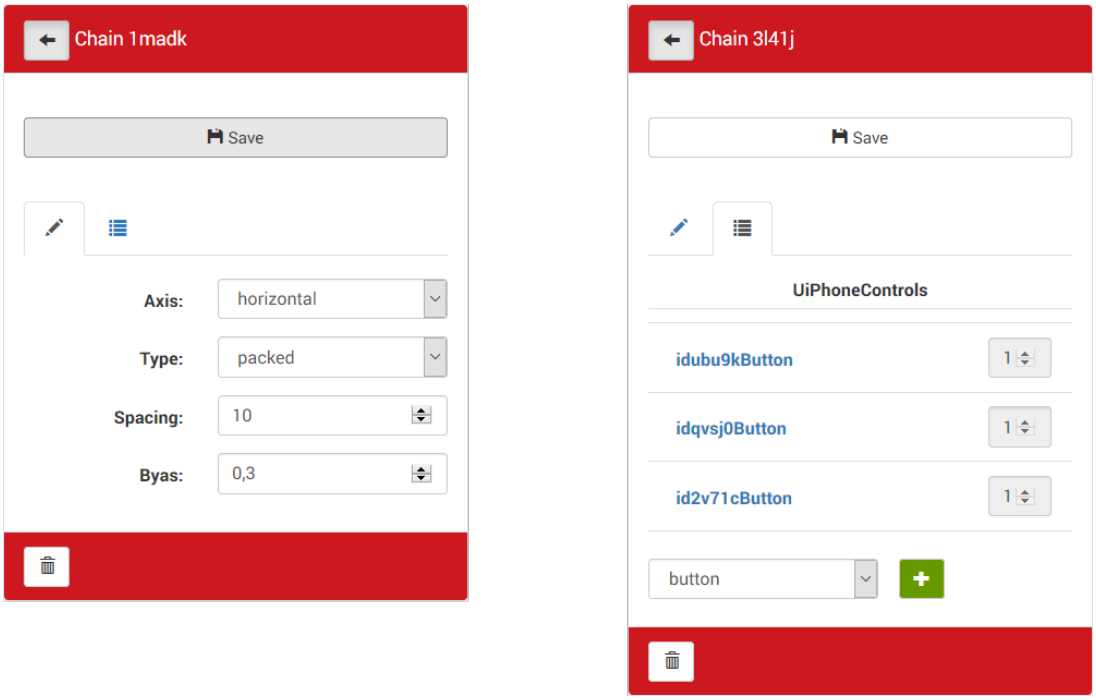

Figura 3.8: Interfaccia per la creazione e modifica di Control Chain

3.4.4 Componente ScreenCanvas e menù

Seguendo l'esempio dell'Interface Builder di XCode e del Layout Editor di Android-Studio, al fine di dare all'utente un supporto visivo per tenere sotto controllo tutti i vincoli e le catene di controlli create, è stato implementato un componente, di tipo Ember Component, chiamato **ScreenCanvas**. Esso consta di un Canvas HTML, con background trasparente, posizionato in corrispondenza dello schermo del dispositivo virtuale che mostra l'anteprima dell'applicazione; la logica di funzionamento è specificata invece in un file JavaScript, *screen_canvas.js*, situato come di consueto nella cartella destinata ai componenti Ember, *app/scripts/components*. Questo componente reagisce alla selezione di un UiPhoneControl da parte dell'utente andando a mostrarne i vincoli attraverso indicatori colorati. Gli indicatori sono sostanzialmente di cinque tipi e si differenziano in base al tipo di vincolo che vanno a evidenziare; di seguito si riportano i diversi vincoli con il rispettivo indicatore:

- vincoli di posizione con la finestra padre (*parent*): i loro indicatori sono linee rette, orizzontali o verticali, che connettono il lato vincolato del controllo al lato della finestra a cui è esso è vincolato. Assumono colore verde se sono vincoli su `top`, `bottom`, `start` o `end`; sono invece di colore giallo per i constraint sui centri, `centerX` e `centerY`.
- Vincoli di posizione con un altro controllo: a differenza di quelli con la finestra, questi vincoli sono composti da coppie di cerchi del medesimo

colore (il colore viene determinato in modo casuale) uno posizionato sul lato vincolato del controllo a cui il constraint appartiene, l'altro posizionato sul lato del controllo che funge da riferimento (fig. 3.9 di sinistra).

- Vincoli di dimensione di larghezza e altezza: i loro indicatori sono coppie di linee rosse rette; per i vincoli sulla larghezza le linee sono orizzontali e posizionate sui bordi superiore (`top`) e inferiore (`bottom`) del controllo mentre per i vincoli sull'altezza le linee sono verticali e posizionate sui bordi sinistro (`start`) e destro (`end`) del controllo (fig. 3.9 di sinistra).

- Vincoli di dimensione di rapporto: gli indicatori di questi vincoli sono piccole linee situate gli angoli del controllo e sono di colore rosa.

- Catene di controlli: quando viene selezionato un controllo che appartiene ad una catena, vengono visualizzati gli indicatori per l'intera control chain. Essi sono di colore grigio e consistono in linee che collegano a due a due i controlli che fanno parte della catena, come mostrato nella figura di destra della 3.9.

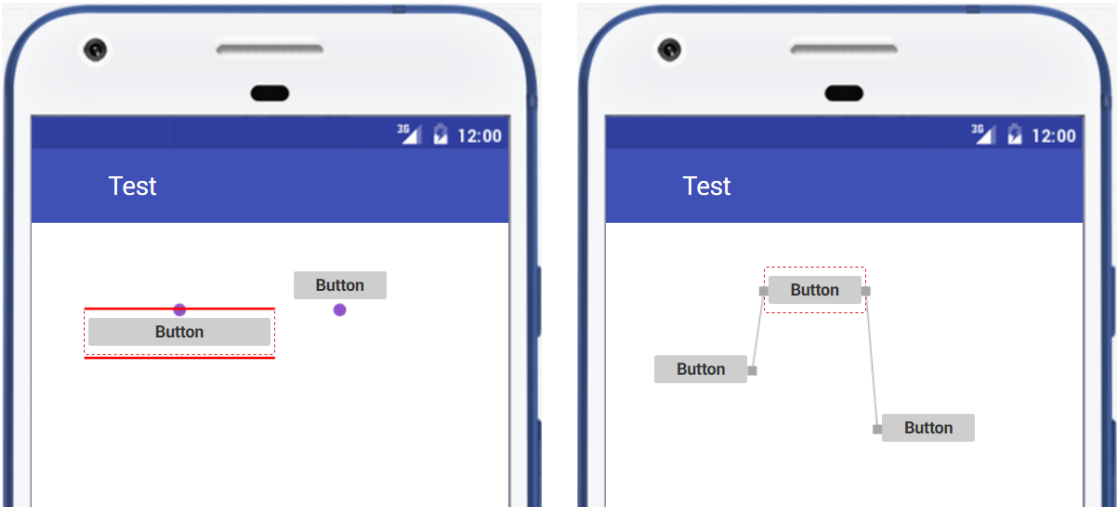

Figura 3.9: Alcuni dei diversi indicatori dei vincoli in Protocode

Un altro intervento eseguito in questa fase del lavoro è stata la modifica del menù principale dell'applicazione: il vecchio menù a TabBar è stato sostituito con un Navigation Drawer menù. Esso proviene dal mondo Android (è definito della libreria di supporto DrawerLayout) ma è ormai ampiamente usato anche all'interno di applicazioni per iOS. La figura 3.10 ne mostra un esempio, ottenuto con Protocode e visualizzato su un Nexus 6P (Android) e su un iPhone 7 Plus (iOS).

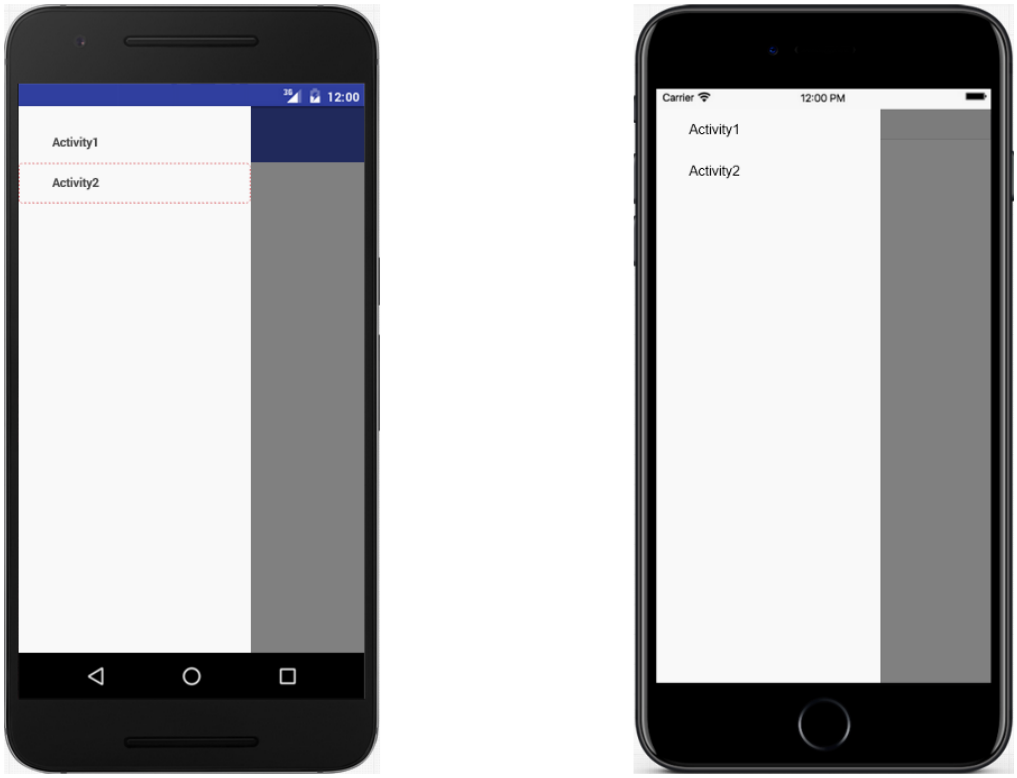

Figura 3.10: Aspetto del nuovo menù in Android (sinistra) e iOS (destra)

3.4.5 Nuovi dispositivi

A questo punto del lavoro, Protocode aveva tutte le funzionalità necessarie a prototipizzare interfacce grafiche adattative: un'applicazione disegnata con Protocode era in grado infatti di sfruttare lo spazio disponibile sul display, qualsiasi fossero le sue dimensioni, e distribuire e adattare le dimensioni dei controlli in base ad esso. L'obiettivo della tesi però si spinge oltre, volendo creare uno strumento di prototipizzazione tramite il quale sia possibile definire un'interfaccia che, al bisogno, sia in grado di presentare contenuti diversi a seconda del dispositivo su cui l'applicazione viene eseguita: ad esempio, potrebbe essere ottimale visualizzare contemporaneamente su un tablet elementi grafici che su uno smartphone sarebbero visualizzati in due diverse schermate. A partire da queste considerazioni, prima di procedere con l'estensione delle funzionalità di Protocode verso questo obiettivo, sono stati aggiunti ad esso nuovi virtual devices di tipo tablet: i Nexus 7, 9 e 10 di casa Google e gli iPad Pro da 9,7" e da 12,9" di Apple. Avendo un display di dimensioni diverse da quello di tutti gli altri iPhone precedenti, è stato ritenuto utile aggiungere anche iPhone X, l'ultimo smartphone commercializzato da Apple.

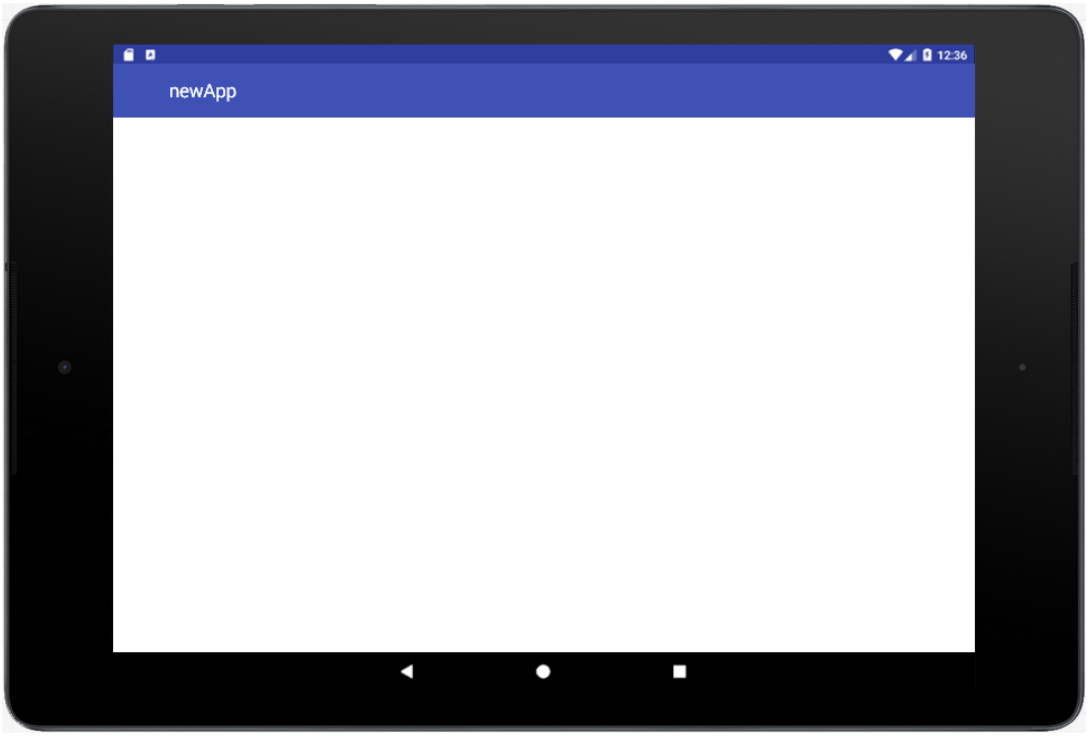

Figura 3.11: Nexus 9 in Protocode 4.0

3.4.6 Scene

Introdotta la nuova categoria di dispositivi tablet, si è quindi proceduto con lo studio e l'implementazione di una funzionalità che permettesse all'utente di differenziare ciò che l'applicazione andava a visualizzare sullo schermo a seconda del tipo di dispositivo su cui essa era in esecuzione. Per quanto riguarda il sistema Android, AndroidStudio permette la creazione di *Fragment*, porzioni di interfaccia utente definite da un file di layout che ne specifica l'aspetto e una classe Java che ne implementa il comportamento, riutilizzabili in diverse schermate dell'applicazione, le cosiddette *Activity*. Relativamente al sistema iOS, la classe *UiViewController* è componibile in una gerarchia a più livelli: un view controller cioè può avere altri view controller come figli, visualizzando contemporaneamente sullo schermo elementi grafici propri ed elementi grafici che fanno parte dell'interfaccia di ciascun figlio. A partire da queste caratteristiche di ciascun ambiente, è stato elaborato un nuovo modo di strutturare l'applicazione, basato sulla classe di model ViewController già esistente e su una nuova classe di model chiamata *Scene*. Il vecchio model ViewController è stato sfruttato per rappresentare le porzioni di interfaccia riutilizzabili; un'istanza di questa classe rappresenta cioè una sorta di Fragment per Android o equivalentemente un UiViewController figlio (cioè inserito nel contesto di un altro UiViewController che ne è il padre) per iOS. Il nuovo model Scene invece serve a rappresentare una vera e propria schermata dell'applicazione, formata da uno o più view controller; un'istanza della classe Scene quindi rappresenta di fatto un'Activity di Android o, allo stesso modo, un UiViewController padre (cioè contenente altri view con-

troller) di iOS. In questo lavoro si è scelto di preconfigurare tre diversi tipi di scena, che vengono di seguito descritti:

- tipo *singleVC*: una scena di questo tipo va a visualizzare i diversi view controller in essa contenuti uno alla volta sullo schermo. All'inizio della scena, il sistema andrà quindi a visualizzare il primo view controller, il quale conterrà alcuni controlli che permetteranno la navigazione a tutti gli altri: ad esempio, attraverso il click su un bottone contenuto nel primo view controller può essere possibile visualizzare il secondo.

- Tipo *singleVCTab*: come per il tipo precedente, una scena di questo tipo visualizza un view controller alla volta; a differenza di una scena singleVC però, una scena di tipo singleVCTab contiene anche una TabBar attraverso la quale è possibile passare da un view controller ad un altro (la navigazione attraverso i controlli, come nel precedente esempio con il bottone, è comunque possibile anche in questo tipo di scena).

- Tipo *multiVC*: una scena di questo tipo va a visualizzare tutti i view controller nella stessa schermata. Essi sono inclusi nel layout dell'interfaccia attraverso un oggetto *Container* che estende la classe UiPhoneControl; esso quindi, come ogni altro controllo, può essere posizionato e dimensionato attraverso i vincoli precedentemente introdotti.

L'implementazione dei primi due tipi di scena non è stata particolarmente complicata dal momento che esse visualizzano un solo view controller alla volta; lo sviluppo di scene `multiVC` invece ha richiesto l'implementazione di un'ulteriore classe di model (Container) e la modifica della classe esistente ViewController. Il motivo è il seguente: per velocizzare il processo di creazione dell'applicazione da parte dell'utente e favorire lo riutilizzo di codice, si è scelto di rendere i view controller riutilizzabili in diverse scene; è cioè possibile definire attraverso un view controller una porzione di interfaccia, comune a più schermate, che viene poi utilizzata in diverse scene. Questo comporta il fatto che, ad esempio, un view controller **A** può essere presente in una scena **X** di tipo `singleVC` e anche in un'altra scena **Y** di tipo `singleVCTab`; di conseguenza, il view controller **A** avrà delle dimensioni diverse nelle due scene, dal momento che la scena **X** sfrutta tutta la schermata mentre la scena **Y** contiene una TabBar che occupa un certo spazio. La questione si diversifica ulteriormente se il view controller **A** viene inserito in una terza scena **Z** di tipo `multiVC`, all'interno di un container di dimensioni definite dall'utente. Questo esempio mostra la necessità di definire delle dimensioni anche per il view controller (la classe di model ViewController originale non le aveva) e di far sì che queste dimensioni si ricalcolino ogni volta che l'utente visualizza una scena diversa in cui il view controller è presente. Oltre a questo, è inoltre necessario che, ogni volta che un view controller aggiorna le proprie dimensioni, tutti i controlli in esso contenuti ricalcolino di conseguenza la propria posizione e dimensione. Tutti questi comportamenti sono stati implementati all'interno delle classi di model citate e nei controller relativi alla route che gestisce l'anteprima dell'applicazione sul dispositivo virtuale mostrato; sono stati poi quindi creati i template che permettono la creazione e la modifica della scene.

In figura 3.12 ne viene mostrato un esempio: come si può osservare, è possibile modificare il nome della scena, impostarla come scena iniziale (`launcher`), sceglierne il tipo e aggiungere o rimuovere i view controller che la compongono.

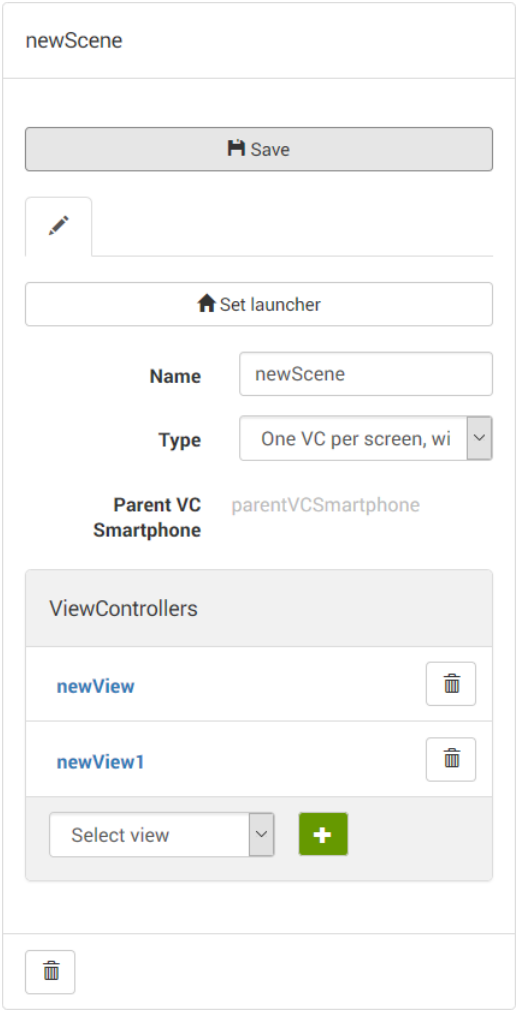

Figura 3.12: Template di modifica di una scena in Protocode 4.0

Nel caso in cui la scena sia di tipo `singleVC` o `singleVCTab`, cliccando sul nome di un view controller della scena è possibile vederne l'anteprima, che per le scene del primo tipo sarà esattamente uguale all'anteprima visualizzata durante la creazione del view controller mentre per le scene del secondo tipo avrà, in più, la TabBar; nel caso in cui la scena sia invece di tipo `multiVC`, sarà possibile cliccare sul parent view controller (`parentVCSmartphone` nel caso si abbia scelto un virtual device di tipo smartphone, `parentVCTablet` nel caso se ne abbia scelto uno di tipo tablet) per visualizzare l'anteprima della scena contenente tutti i view controller nei relativi contenitori (il parent view controller è utilizzato esclusivamente per la specifica della posizione e dimensione dei Container contenenti i vari view controller). Un esempio di quest'ultimo tipo di scena di può osservare in figura 3.13, dove viene visualizzato il `parentVCTablet` della scena e i due view controller che la compongono, inseriti ciascuno in un contenitore del quale l'utente può specificare dimensione e posizione. Si ricorda inoltre che, attraverso

questa nuova funzionalità di Protocode, il tipo di scena può essere diverso per smartphone e tablet: visualizzando una scena ad esempio su iPhone X, il tipo mostrato dal template della scena sarà il tipo scelto per quella scena su smartphone; cambiando il dispositivo e scegliendo un tablet, come ad esempio Nexus 7, il tipo visualizzato nel template della scena sarà quello scelto per la scena su tablet, potenzialmente diverso dal precedente.

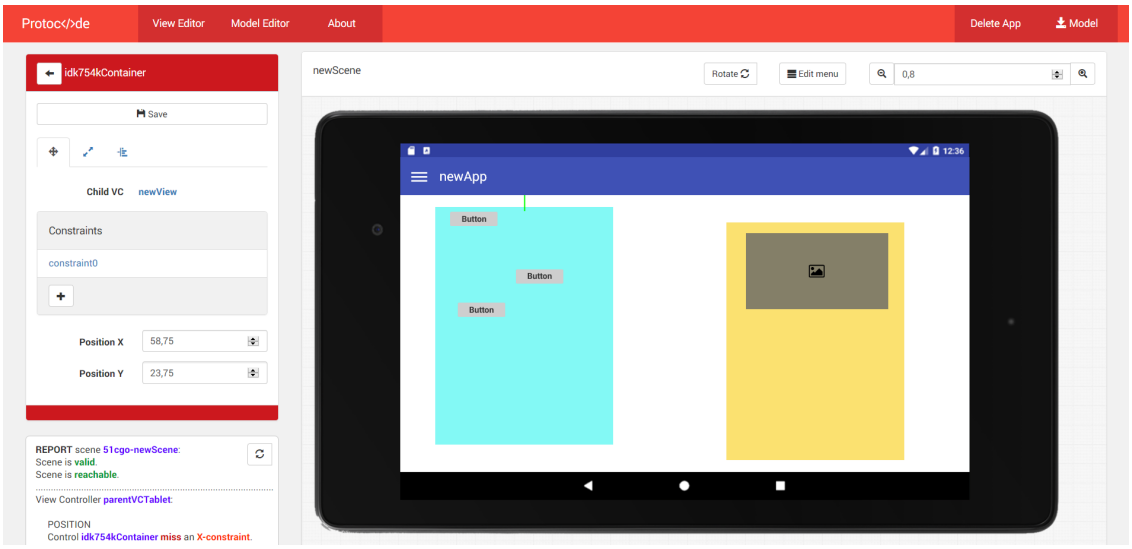

Figura 3.13: Esempio di una scena di tipo multiVC in Protocode 4.0

Al termine di tutti gli interventi sopra descritti se ne è reso necessario uno ulteriore, che riguarda il fatto che un view controller possa essere inserito in diverse scene. Il motivo è il seguente: alcuni controlli, in particolare bottoni, liste e griglie, sono legati ad oggetti Navigation, attraverso cui è possibile definire quale view controller (o, a questo punto del lavoro, scena) l'applicazione debba visualizzare dopo aver premuto su uno di essi. Il problema sorge dal fatto che, essendo un view controller potenzialmente presente in diverse scene, si potrebbe voler raggiungere una destinazione diversa dopo aver premuto sullo stesso controllo ma nel contesto di due scene diverse in cui il view controller contenente quel controllo è inserito. Sono quindi state eseguite tutte le modifiche al model Navigation e ai controller delle diverse route coinvolte nella creazione e modifica di oggetti di navigazione; in particolare, a differenza della versione precedente di Protocode in cui ogni controllo era legato al più ad un solo oggetto di navigazione, ora un bottone, lista o griglia contenuta in un view controller presente in diverse scene sarà legato ad un oggetto Navigation per ognuna di queste scene (il model Navigation contiene ora un attributo che specifica la scena per la quale è definito).

3.4.7 Report View

L'ultimo intervento su Protocode è stata la realizzazione del componente **ReportView** la cui ideazione, come per il componente ScreenCanvas, prende spunto dai celebri IDE di Android e iOS, vale a dire AndroidStudio e XCode. ReportView consiste in un piccolo pannello, visualizzato nel template delle scene, che riporta

l'analisi di alcune caratteristiche della scena che si sta creando o modificando; se ne può osservare un esempio in figura 3.14 mentre di seguito se ne riportano le diverse sezioni e la relativa funzionalità:

- la prima riga sotto il titolo (1) riporta se la scena è valida (contiene almeno un view controller) oppure no;
- la riga successiva (2) riporta il risultato di un'analisi della raggiungibilità della scena; una scena è raggiungibile se è vera almeno una delle seguenti asserzioni:
  - la scena corrente è la prima scena dell'applicazione (`launcher = true`);
  - la scena corrente è presente nel menù dell'applicazione (esiste un *menuItem* che ha essa come destinazione);
  - la scena corrente è la destinazione di un controllo legato ad un oggetto *Navigation* (un *Button*, *ListView* o *GridView*);
- la sezione (3) riporta un'analisi di ciascun singolo view controller, pertanto ne viene creata una per ogni view controller della scena; le analisi effettuate sono riportate nelle sottosezioni (4) e (5), spiegate di seguito;
- la sottosezione (4) riporta l'analisi del posizionamento dei controlli: per ciascun controllo viene verificato se esiste almeno un vincolo su ciascun asse; in caso contrario viene riportato un avviso;
- la sottosezione (5), infine, riporta l'analisi degli oggetti non validi presenti nel view controller (i quali non saranno esportati nel file di modello XMI): le Control Chain con meno di due controlli e i Constraint (vincoli di posizione) che non hanno passato il test dell'algoritmo di controllo dei vincoli.

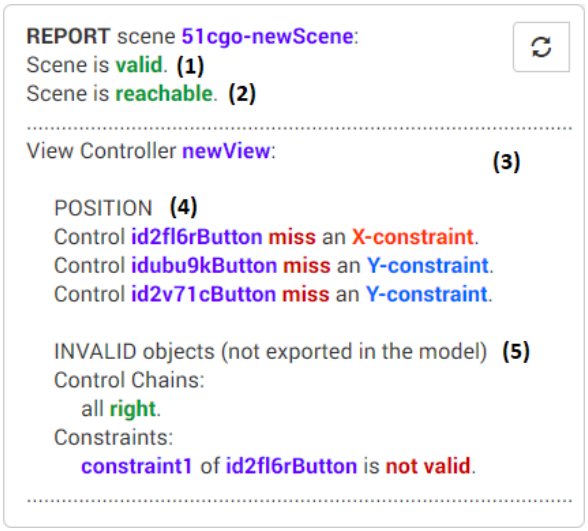

Figura 3.14: Esempio di Report View in Protocode 4.0

3.4.8 Risultato finale

In questa ultima sezione del capitolo viene riportato il risultato finale del lavoro di aggiornamento ed estensione di Protocode, in termini di interfaccia grafica, routing e model. Al termine viene infine descritto l'aggiornamento della funzionalità di generazione del file di modello a fronte di tutte le modifiche fatte.

3.4.8.1 Interfaccia grafica

Oltre a tutto il lavoro fatto sui template durante lo sviluppo delle nuove funzionalità, descritto nelle sezioni precedenti, è stato eseguito un restyling dell'interfaccia generale del View Editor di Protocode. L'obiettivo principale di questa modifica è stato quello di fornire all'utente un'interfaccia il più possibile comoda per il design sui dispositivi di tipo tablet che, date le dimensioni, necessitavano di un pannello di anteprima più grande; al contempo si è cercato di minimizzare il più possibile il numero di scroll della pagina effettuati dall'utente durante l'utilizzo di Protocode, sfruttando al meglio lo spazio a disposizione. Il risultato di queste modifiche è osservabile in figura 3.15; di seguito sono riportati i principali cambiamenti effettuati:

- la vecchia barra contenente nome dell'app, identificativo del team di sviluppo e pulsante per la generazione del file di model è stata eliminata dal View Editor (nel Model Editor è ancora presente): nome e identificativo sono stati spostati nel nuovo pannello Application (2), mentre il pulsante per la generazione del modello è stato spostato nella barra principale (la prima barra in alto, di colore rosso);

- la vecchia barra che permetteva la selezione del dispositivo smartphone e smartwatch su cui visualizzare l'anteprima dell'applicazione creata è stata eliminata: è stato mantenuto un singolo menù di selezione (1), più compatto, che cambia il suo contenuto a seconda della scelta effettuata tramite i pulsanti sottostanti;

- il pannello di anteprima (5) è stato ingrandito per permettere una migliore visualizzazione dei tablet: il vecchio pannello contenente l'elenco dei view controller è stato integrato nel nuovo pannello Application (2), consentendo di spostare a sinistra il pannello di modifica delle proprietà dell'oggetto correntemente selezionato (3), che si trovava a destra;

- la palette dei widget con cui comporre l'interfaccia dell'applicazione (4) è infine stata spostata in alto.

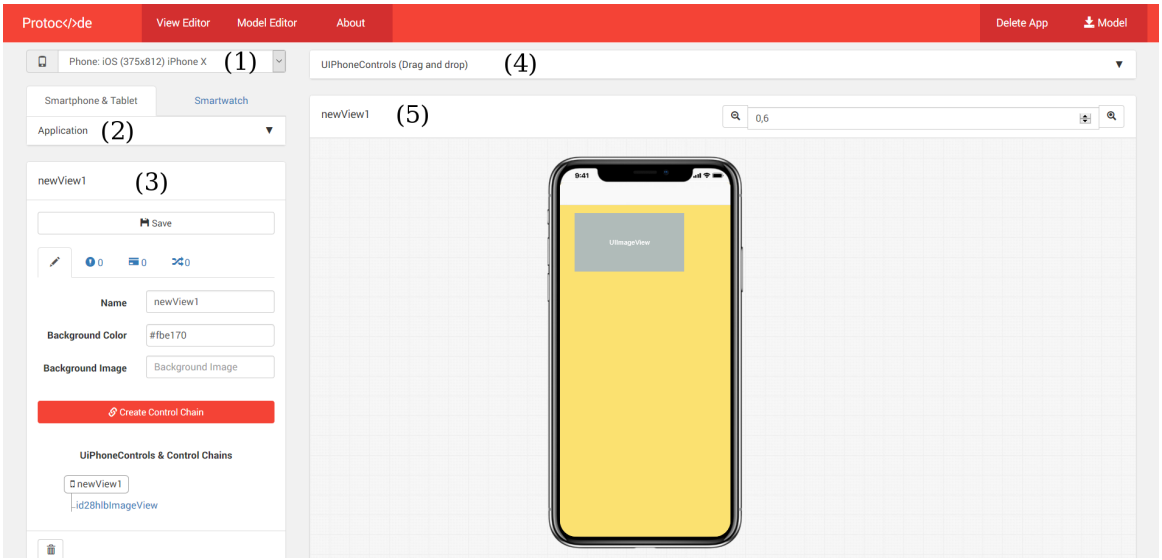

Figura 3.15: Interfaccia del View Editor di Protocode 4.0

Come si può osservare dalla figura 3.15, il pannello Application non contiene nulla di quanto appena descritto. Il motivo è che questo pannello è stato reso dinamico: normalmente è compresso, lasciando lo spazio agli altri elementi della GUI; quando il mouse si sposta su di esso, si espande automaticamente mostrando il nome dell'app, l'identificativo del team di sviluppo, l'elenco dei view controller e l'elenco delle scene (consentendo la modifica di nome e identificativo e la creazione di nuovi view controller e scene). In figura 3.16 viene riportato un esempio del pannello Application espanso.

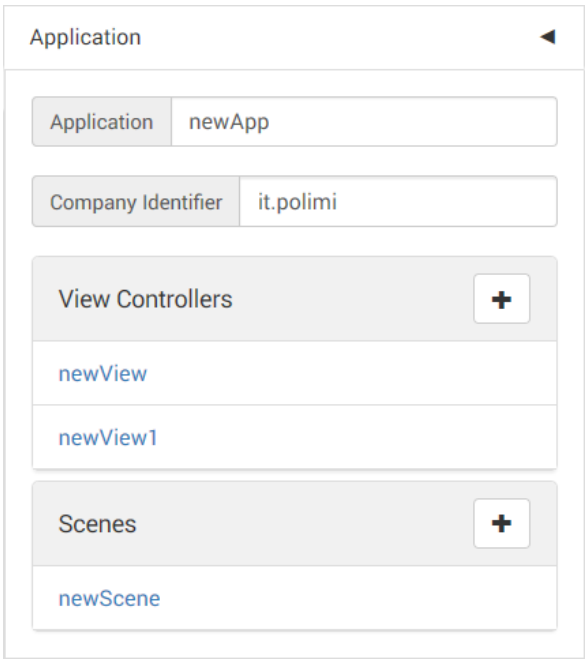

Figura 3.16: Pannello Application del View Editor di Protocode 4.0

3.4.8.2 Model e Routing

Il lavoro descritto nelle sezioni da 3.4.1 a 3.4.7 ha portato alla creazione di nuovi file di model e alla modifica di quasi tutti quelli già esistenti (ad eccezione di quelli relativi alla parte smartwatch e model dell'applicazione). Si riporta di seguito, in figura 3.17, un Class Diagram parziale dove, per questioni di chiarezza e spazio, sono stati tralasciati i file di model non modificati e gli attributi meno significativi.

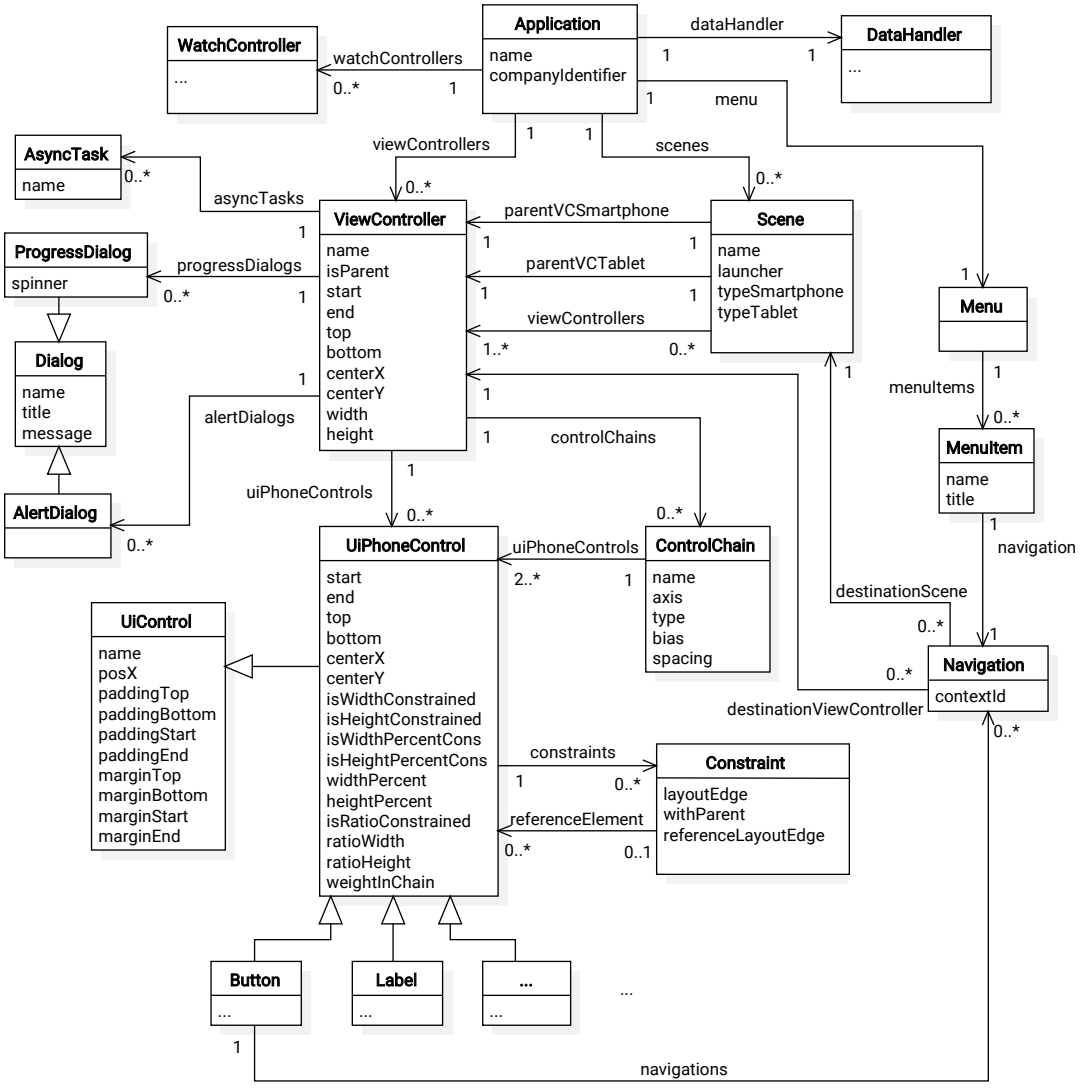

Figura 3.17: Class Diagram parziale del Model di Protocode 4.0

Oltre al model, anche il routing di Protocode è stato ampiamente modificato durante questo lavoro di tesi: non solo sono state create nuove route per Constraint, ControlChain, Scene e Container, ma sono anche state duplicate quelle di tutti gli UiPhoneControl, dal momento che è possibile visualizzare i dettagli di ciascuno di essi sia all'interno della parent route ViewController sia in quella Scene (queste due route sono ad un livello superiore nella gerarchia; in altre parole esse contengono le route dedicate ai controlli, che sono quindi presenti sia

come "figlie" di ViewController sia come "figlie" di Scene). La figura 3.18 mostra uno schema delle route come si presentano al termine del lavoro, omettendo per brevità e questioni di spazio le route relative al Model Editor e alla controparte smartwatch dell'applicazione.

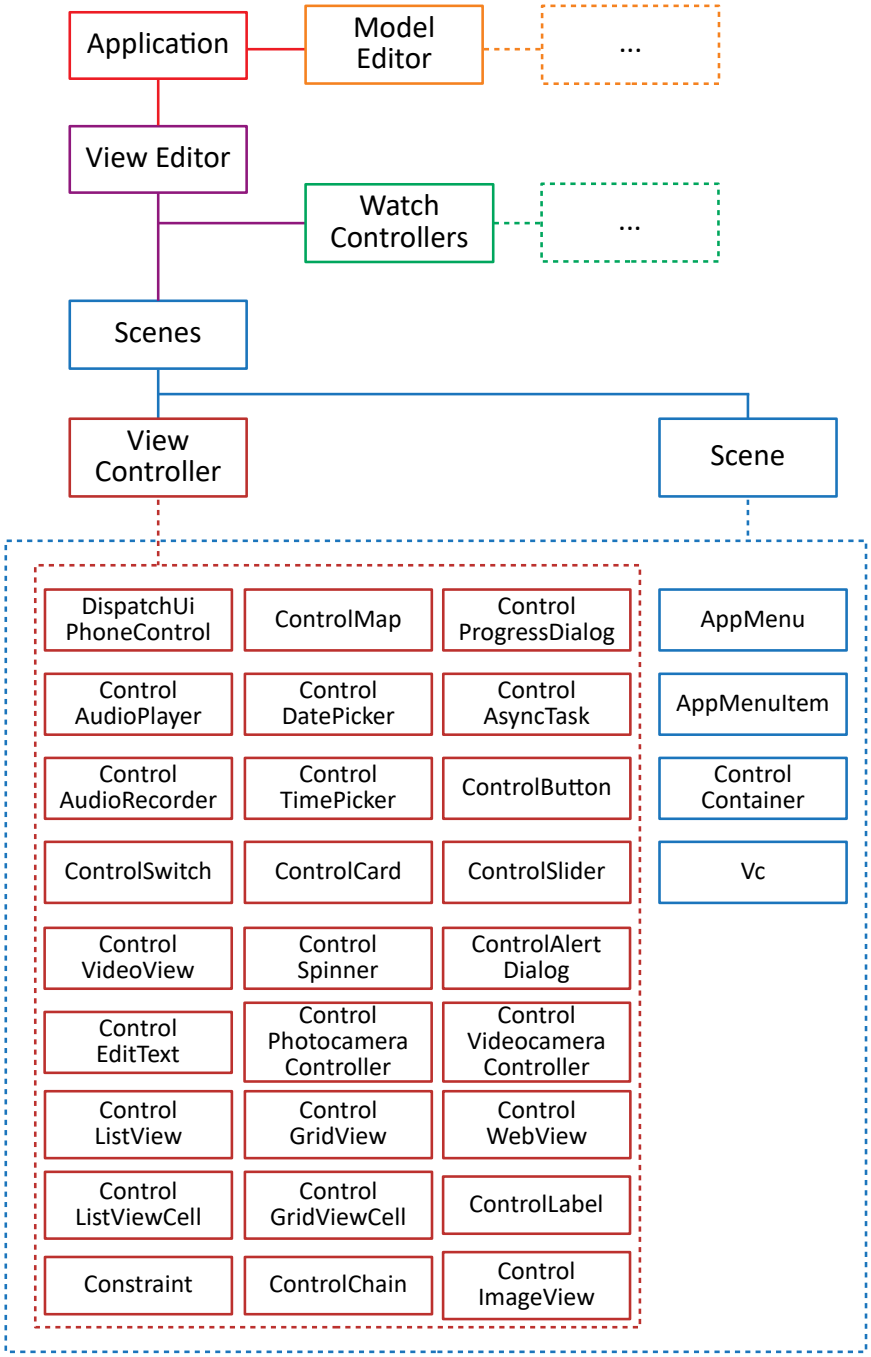

Figura 3.18: Route di Protocode 4.0

3.4.8.3 Generazione del modello astratto

La funzionalità di generazione del modello XMI dell'applicazione creata è rimasta essenzialmente la stessa delle versioni precedenti di Protocode: per ciascuna classe di model è stato definito un metodo *toXml()* che genera la parte di modello riguardante quella classe; la generazione ha inizio chiamando il metodo della classe *Application* che ricorsivamente chiama i metodi toXml() di ciascun *ViewController*, *Scene*, *WatchController*, *Menu* e *DataHandler*. Questi, a loro volta, richiamano i metodi toXml() dei model ad essi collegati e così via ricorrendo fino alla generazione del file di modello completo. Il lavoro di questa tesi, avendo modificato alcune classi di model e createne delle altre, ha conseguentemente modificato e implementato i relativi metodi toXml(); la figura 3.19 di seguito riporta uno schema sintetico del file di modello, con in evidenza solo le parti modificate o create ex-novo in questo lavoro.

```
<?xml version="1.0" encoding="UTF-8"?>
<metamodel:Application xmi:version="2.0"
   xmlns:xmi="http://www.omg.org/XMI"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:metamodel="http://metamodel/1.0"
   xsi:schemaLocation="http://metamodel/1.0 ../metamodel/metamodel.ecore"
   name="APPNAME" companyIdentifier="IDENTIFIER">
   <dataHandler>
   ...
   </dataHandler>
   <viewControllers id="VC_ID" name="VC_NAME" backgroundColor="COLOR" ...>
      <controlChains id="CHAIN_ID" viewController="VC_ID" axis="vertical"
         type="CHAIN_TYPE" nControls="INTEGER" bias="FLOAT" spacing="INTEGER"/>
      ...
      <buttons id="UPC_ID" ... >
         <dimensionConstraint uiPhoneControl="UPC_ID" widthFixed="FLOAT"
            heightPercent=''FLOAT'' ... />
         <positionConstraints id="C_ID" uiPhoneControl="UPC_ID"
            layoutEdge="EDGE" withParent="TRUE|FALSE"
            referenceLayoutEdge="EDGE"/>
         ...
         <navigations id="NAV_ID" contextScene="SCENE_ID"
            destinationScene="SCENE_ID"/>
         ...
      </buttons>
      ...
      </viewControllers>
   ...
   <scenes id="SCENE_ID" name="SCENE_NAME" launcher="TRUE|FALSE"
      typeSmartphone="SCENE_TYPE" typeTablet="SCENE_TYPE">
      <parentViewControllers id="VC_ID" name="VC_NAME" ...>
         <containers childViewController="VC_ID" id="CONT_ID" ...>
         ...
         </containers>
         ...
      </parentViewControllers>
      ...
      <childViewControllers viewController="VC_ID"/>
      ...
   </scenes>
   ...
   <watchControllers ...>
   ...
   </watchControllers>
   ...
   <menu>
   ...
   </menu>
</metamodel:Application>
```

Chains
Dimension Constraints
Position Constraints
Navigations
Parent VCs
Child VCs
Scenes

Figura 3.19: Struttura di un file di modello XMI di Protocode 4.0

Capitolo 4

Fase di generazione del codice

4.1 Introduzione

In questo capitolo viene presentata la fase di generazione del codice nativo a partire dal modello astratto generato con Protocode. Essa viene effettuata attraverso **MobileCodeGenerator**, un software che, come Protocode, è nato da lavori di tesi precedenti ([14] [15] [16]) ed è stato qui aggiornato ed esteso per generare il codice relativo alle funzionalità introdotte in Protocode 4.0 e per supportare gli ultimi aggiornamenti degli IDE AndroidStudio 3.0.1 e XCode 9.2. MobileCodeGenerator è un tool realizzato sotto forma di plugin per l'IDE Eclipse e basato su *Epsilon*, una famiglia di linguaggi e strumenti per la generazione di codice e la trasformazione *model-to-model* [52]; esso permette di generare, a partire da un file di modello creato manualmente (seguendo una struttura precisa) o generato con Protocode, due progetti dell'applicazione prototipizzata, uno per AndroidStudio (per sistemi Android) e l'altro per XCode (per sistemi iOS). Nelle sezioni che seguono verranno descritti Epsilon e gli altri strumenti e framework utilizzati per l'implementazione di MobileCodeGenerator e successivamente, come per Protocode, verranno descritti la struttura iniziale di MobileCodeGenerator (versione 3.0) e il lavoro eseguito per aggiornarlo alla versione 4.0.

4.2 Strumenti e framework utilizzati

Per la realizzazione di MobileCodeGenerator è stata utilizzata una distribuzione di Eclipse, la Neon versione 4.6, pronta all'uso e contenente una versione stabile (1.4) di Epsilon. All'interno di questa, è già presente *EMF* (Eclipse Modeling Framework), il primo dei due framework utilizzati per l'implementazione di MobileCodeGenerator; il secondo framework utilizzato invece è *oAW* (openArchitectureWare), installabile come plugin a parte.

4.2.1 Eclipse Modeling Framework

EMF [53] è un framework per la modellazione che fornisce anche dei servizi per la generazione di codice basata su dati strutturati; in MobileCodeGenerator è stato utilizzato per la definizione del *meta model*, mentre per la generazione del codice si è scelto di utilizzare oAW. Il meta model è una sorta di rappresentazione della struttura che deve avere il file di modello; esso ha diverse funzioni:

- attraverso il meta model è possibile **creare** un file di modello conforme ad esso. Questa operazione, nella toolchain formata da Protocode e MobileCodeGenerator, non viene eseguita dal momento che il file di model XMI è generato da Protocode; tuttavia, se si volesse, sarebbe possibile creare un file di modello direttamente in Eclipse, senza utilizzare Protocode, e generarne il codice corrispondente.

- Dato un file di model, è possibile farne la **verifica** della correttezza strutturale attraverso il confronto con il meta model; questa operazione è eseguita in MobileCodeGenerator all'inizio del processo di generazione dei dati.

- Attraverso il meta model, infine, viene data un'**interpretazione** dei dati contenuti nel file di modello: mentre viene verificato che il model in ingresso sia conforme al meta model, vengono anche identificate la struttura e le componenti del modello, che servono al motore di traduzione per generare il codice.

4.2.2 openArchitectureWare

openArchitectureWare è nato come progetto open source su SourceForge.net e conteneva gli strumenti *Xtext*, *Xpand* e *MWE*; successivamente, è stato spostato su Eclipse.org [54] e lo sviluppo di ciascun progetto è poi proseguito in modo indipendente. I due progetti utilizzati per la realizzazione di MobileCodeGenerator sono Xpand [55] e MWE [56]: il primo è composto da una serie di strumenti per la definizione delle regole che traducono le diverse parti del file di modello nel codice desiderato; il secondo permette la definizione di un file di workflow che specifica l'elenco ordinato delle operazioni che il motore di traduzione deve eseguire per generare il codice. Per quanto riguarda MWE, c'è poco da aggiungere a quanto già detto, se non che permette di riutilizzare più volte le parti di codice comuni a diversi processi di traduzione, rendendo il progetto modulare; inoltre esso consente di definire, oltre alle operazioni di generazione di codice, operazioni di post-processing come ad esempio l'aggiustamento automatico delle tabulazioni e dei caratteri di accapo del codice secondo le più comuni convenzioni. Xpand invece, come già specificato, è composto da svariati tool che hanno diverse funzioni:

- **Xpand**: è lo strumento principale, serve a definire le regole da applicare per eseguire la traduzione del modello in codice.

- **Xtend**: dal momento che Xpand ha un'insieme di funzionalità basilari e limitate, attraverso Xtend è possibile definire classi e metodi aggiuntivi

sfruttando tutta la potenza del linguaggio Java che possono essere utilizzati nei file di Xpand.

- **Check**: questo strumento permette di definire ulteriori controlli da eseguire sul file di modello, dopo che questo ha passato la verifica con il meta model, ma prima di essere tradotto in codice; anch'esso ha una sintassi base come Xpand ma può essere potenziato con metodi scritti in Java grazie ad Xtend.

La figura 4.1 mostra uno schema di alto livello del processo di generazione del codice.

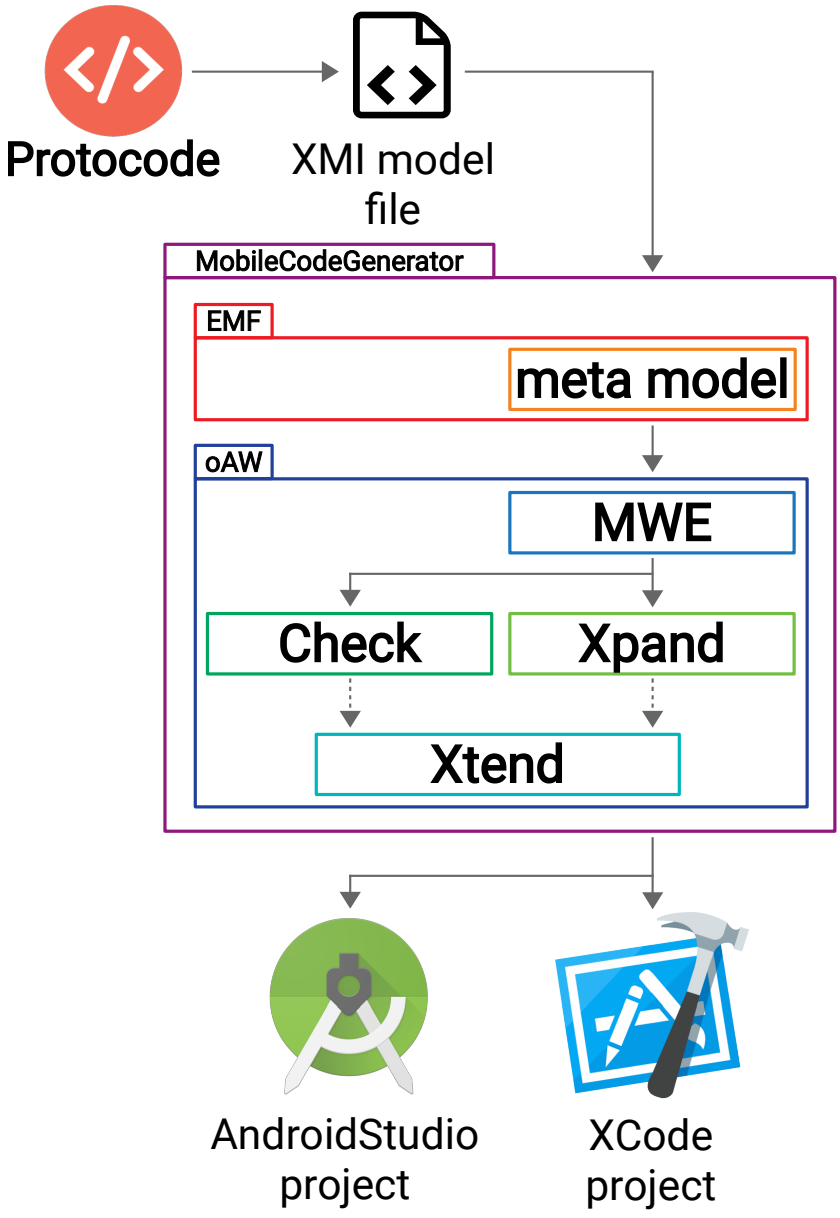

Figura 4.1: Processo di generazione del codice di MobileCodeGenerator

4.3 MobileCodeGenerator 3.0

In questa sezione verrà descritta la versione 3.0 di MobileCodeGenerator, da cui questo lavoro di tesi è iniziato: verrà inizialmente presentata la struttura generale del progetto e successivamente ciascuna parte verrà descritta più dettagliatamente. Prima di procedere con la presentazione della struttura però, è utile dare uno sguardo ai diversi tipi di file che compongono il progetto; questo aiuterà a comprendere meglio la funzione di ciascuna parte quando verrà descritta in dettaglio. Si riportano quindi di seguito i tipi di file presenti in MobileCodeGenerator e la loro rispettiva funzione:

- file `.ecore`: questo tipo di file è usato per la definizione del meta model, utilizzando una sintassi definita da EMF e basata su XML;
- file `.mwe`: questi file sono utilizzati da MWE e descrivono le operazioni che il motore di traduzione deve eseguire per generare il codice, con una sintassi basata anch'essa su XML;
- file `.xpt`: sono i file che specificano le regole di traduzione del modello in codice, la cui sintassi è definita da Xpand;
- file `.java`: questi file contengono le classi Java implementate per estendere le potenzialità di Xpand;
- file `.ext`: sono i file di Xtend e contengono le interfacce dei metodi implementati nelle classi Java e utilizzabili nei file di Xtend e Check (fungono da collegamento tra i file .xpt / .chk e le classi .java);
- file `.chk`: questo tipo di file è usato per definire i controlli da eseguire sul modello prima della traduzione.

4.3.1 Architettura di MobileCodeGenerator

MobileCodeGenerator è organizzato in tre directory principali, suddivise a loro volta in diverse sotto-cartelle; la struttura è la seguente:

- *src*: questa cartella contiene tutti i file di codice sorgente. Essa è suddivisa nelle seguenti sotto-cartelle (*package*):
  - *android_templates*;
  - *android_activities_templates*;
  - *androidwear_activities_templates*;
  - *android_extensions*;
  - *ios_templates*;
  - *ios_viewcontrollers_templates*;
  - *ios_watchcontrollers_templates*;
  - *ios_extensions*;

  - *app_extensions*;
  - *metamodel*;
  - *model*;
  - *model_checks*;
  - *workflow*.

  La descrizione dei file contenuti in ciascun package e della loro funzione nel processo di generazione del codice verranno spiegati nelle sezioni successive.

- *src-gen*: questa directory è utilizzata come destinazione del processo di generazione; essa conterrà cioè il codice generato, in sotto-cartelle chiamate ciascuna con il nome dell'applicazione scelto dall'utente nella fase di prototipizzazione. Ogni directory di ciascuna applicazione generata avrà due ulteriori sotto-cartelle:

  - *android*, contenente il progetto dell'applicazione generata per AndroidStudio (file sorgenti di codice per Android e file di progetto di Gradle, il plugin di AndroidStudio che gestisce il progetto);
  - *ios*, contenente il progetto dell'applicazione generata per XCode (file sorgenti di codice per iOS e file di progetto di XCode).

- *utils*: questa cartella contiene tutti i file, sia di codice sorgente sia di altra natura (come immagini, audio ecc.), che devono semplicemente essere copiati così come sono nelle directory dei progetti generati. I file sono divisi nelle tre sotto-cartelle seguenti:

  - *android_default_files*, che contiene tutti i file di default per il progetto di AndroidStudio;
  - *ios_default_files*, che contiene tutti i file di default per il progetto di XCode;
  - *user_files*, che contiene tutti i file che l'utente ha specificato durante la fase di prototipizzazione con Protocode (ad esempio un file immagine da inserire in una ImageView e così via).

Nelle sezioni successive, come specificato per la directory `src`, verranno presentati i file sorgente di ciascun package di MobileCodeGenerator e la rispettiva funzione.

4.3.2 Model e Metamodel

Il package `model` serve semplicemente per contenere i file di modello XMI: dopo la fase di prototipizzazione, il file di modello creato con Protocode viene spostato in questo package per poter essere utilizzato per la generazione del codice. Nel caso non si volesse, per qualche motivo, utilizzare Protocode per la creazione del file di model è anche possibile crearne uno ex-novo attraverso un editor specifico messo a disposizione in Eclipse dal framework EMF, che guida l'utente durante la creazione tenendo come riferimento il meta model.

Il package `metamodel`, come si può immaginare, contiene il file di meta model (`.ecore`); esso può essere creato e modificato sia con un semplice editor testuale sia tramite un editor specifico, ancora una volta messo a disposizione di Eclipse da EMF, che ne da una rappresentazione grafica che ne mette in evidenza la struttura gerarchica. In figura 4.2 si può osservare, a titolo di esempio, una porzione di meta model visualizzata con questo secondo editor.

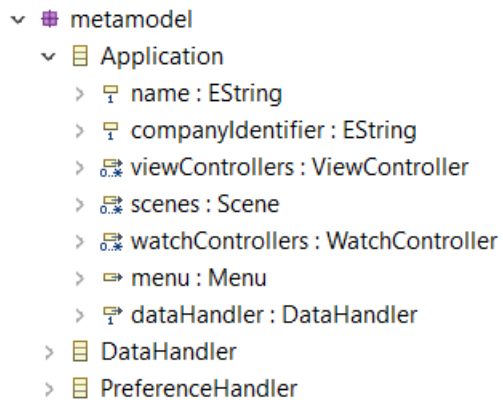

Figura 4.2: Porzione di meta model nel Sample Ecore Model Editor di Eclipse

Il package `metamodel` contiene inoltre un file, chiamato *metamodel.ecore_diagram*, che contiene come suggerisce il nome un diagramma del meta model generato automaticamente da EMF a partire dal file `metamodel.ecore`.

4.3.3 Model Checks e App extensions

Il package `model_checks` contiene tre file:

- *JavaChecks.java*, un file Java contenente l'implementazione di tutti i metodi accessori usati nei controlli del modello;
- *Extensions.ext*, un file di estensione in cui sono definite le interfacce di tutti i metodi implementati in `JavaChecks.java`;
- *Checks.chk*: questo è il file di Check che esegue i controlli sul modello prima di iniziare il processo di generazione del codice; esso sfrutta i metodi contenuti in `JavaChecks.java` attraverso il file `Extensions.ext`, per mezzo dell'istruzione "`extension model_checks::Extensions;`".

Il package `app_extension` invece contiene le classi `.java` e il file di estensione `.ext` utilizzati da file di Xpand sia di Android che di iOS (come si vedrà, saranno definite delle estensioni solo per Android o solo per iOS, mentre le classi del package `app_extension` sono utilizzati dai template contenenti le regole di traduzione per entrambi i sistemi).

4.3.4 Package di Android

I package contenenti i file per la traduzione del modello in codice nativo per Android sono `android_extensions`, `android_templates`, `android_activities_templates` e `androidwear_activities_templates`. Il primo package è simile a `app_extension`, ma contiene le classi Java e il file Xtend dei metodi utilizzati solo dai package che fanno la traduzione per Android.
Il package `android_templates` contiene i file `.xpt` per la traduzione delle componenti principali di una applicazione Android; essi sono riportati di seguito, con la descrizione della parte di applicazione che vanno a generare:

- *Activities.xpt* e *ActivitiesWatch.xpt*: contengono le regole per tradurre i view controller e i watch controller definiti nel modello XMI in Activity e Fragment dell'applicazione, per Android (smartphone) e AndroidWear (smartwatch);
- *AndroidManifest.xpt* e *AndroidManifestWatch.xpt*: contengono le regole per la generazione del file *Manifest.xml* dell'applicazione Android e AndroidWear;
- *DefaultFiles.xpt* e *DefaultFilesWatch.xpt*: anche qui suddivisi per l'applicazione Android e AndroidWear, questi file servono per la generazione dei file di progetto, come ad esempio file di Gradle o classi Java comuni;
- *DataHandler.xpt*, *CloudHandler.xpt*, *DatabaseHandler.xpt*, *DatabaseHelper.xpt*, *PreferenceHandler.xpt*, *StorageHandler.xpt* e *ModelClasses.xpt*: contengono tutte le regole per la generazione della parte dell'applicazione relativa al modello dei dati;
- *XMLFiles.xpt* e *XMLFilesWatch.xpt*: servono per la generazione di tutti file XML dell'applicazione Android e AndroidWear, relativi principalmente ai layout di Activity e Fragment.

I package `android_activities_templates` e `androidwear_activities_templates`, infine, contengono i file di Xpand per la generazione del codice relativo rispettivamente a UiPhoneControl e UiWatchControl, vale a dire il codice che gestisce il comportamento di bottoni, etichette, ImageView e tutti gli altri controlli, inseribili nei view controller e nei watch controller. Le regole contenute in questi package non sono richiamate direttamente nel file di workflow: esso infatti esegue l'espansione di quelle di `android_templates` le quali, ricorsivamente, richiamano l'espansione delle regole relative ai controlli.

4.3.5 Package di iOS

Specularmente ai package dedicati alla traduzione in codice per Android, i package `ios_extensions`, `ios_templates`, `ios_viewcontrollers_templates` e `ios_watchcontrollers_templates` contengono le regole per la traduzione del modello in codice per il sistema iOS. Il primo, allo stesso modo di `android_extensions`, contiene le estensioni utilizzate solo per la traduzione per iOS; gli ulti-

mi due, esattamente come `android_activities_templates` e `androidwear_activities_templates`, servono per la generazione del codice Swift relativa ai controlli di smartphone e smartwatch.
La funzione del package `ios_templates` è anch'essa simile al corrispondente `android_templates`, ma vale la pena descriverla in dettaglio dal momento che i file che compongono un'applicazione Android e una iOS sono diversi. Si riportano quindi di seguito i file Xpand contenuti in questo package, con una breve descrizione della loro funzione:

- *ViewControllers.xpt* e *InterfaceControllers.xpt*: contengono le regole per tradurre view controller e watch controller in UiViewController e WKInterfaceController, rispettivamente per l'applicazione iOS e watchOS;
- *Xcodeproj.xpt* e *InfoPlist.xpt*: servono a generare i file di progetto di XCode `.xcodeproj` e di playlist `Info.plist`;
- *DefaultFiles.xpt* e *DefaultFilesWatch.xpt*: come per Android, servono a generare gli altri file di progetto e le classi Swift comuni;
- *FileStorage.xpt*, *CloudHandler.xpt*, *CoreData.xpt* e *UserDefaults.xpt*: contengono le regole per la generazione del codice relativo alla parte di model dell'applicazione;
- *LaunchStoryboard.xpt*, *MainStoryboard.xpt* e *InterfaceStoryboard.xpt*: il primo serve a generare il file `.storyboard` per il lancio dell'applicazione, mentre gli altri due servono a generare le Storyboard che specificano le interfacce rispettivamente di tutti gli UiviewController e di tutti i WKInterfaceController.

4.3.6 Workflow

L'ultimo package, `workflow`, contiene i file `.mwe` utilizzati dal Modeling Workflow Engine: essi contengono le informazioni riguardo alle operazioni da eseguire nel processo di generazione del codice, come ad esempio l'esecuzione dei controlli di Check o l'espansione delle regole contenute nei file di Xpand. In questo package sono presenti tre file:

- *AndroidGenerator.mwe*, contenente tutte le operazioni da eseguire per la generazione del progetto Android;
- *iOSGenerator.mwe*, contenente le operazioni per generare il progetto iOS;
- *iOSAndAndroidGenerator.mwe*, che contiene tutte le istruzioni dei due file precedenti, permettendo la generazione automatica di entrambi i progetti.

4.4 Aggiornamento a MobileCodeGenerator 4.0

Avendo presentato MobileCodeGenerator 3.0, si può ora procedere con la descrizione di tutte le modifiche attuate in questo lavoro di tesi che lo hanno portato alla versione 4.0. Nelle sezioni che seguono è riportata la descrizione degli interventi eseguiti, a partire dalle modifiche delle parti di MobileCodeGenerator comuni alla generazione sia del codice per Android che per iOS, per poi proseguire in dettaglio sulle modifiche fatte alle parti che generano il codice specifico per ciascun sistema.

4.4.1 Metamodel

Il primo intervento eseguito su MobileCodeGenerator è stata la modifica ed estensione del meta model, al fine di renderlo conforme alla struttura dei file di modello generati con la nuova versione di Protocode; sono quindi stati creati nuovi elementi e sono state aggiornate alcune classi di quelle già esistenti. Di seguito sono riportati due elenchi: il primo mostra i nuovi elementi creati mentre il secondo mostra gli aggiornamenti fatti alle classi esistenti. Nuovi elementi:

- classe *PositionConstraint*, per i vincoli di posizione (in relazione a UiPhoneControl);
- classe *DimensionConstraint*, per i vincoli di dimensione (in relazione a UiPhoneControl);
- classe *ControlChain*, per le catene di controlli (in relazione a ViewController)
- classe *Container*, per i controlli di tipo appunto Container (in relazione a ViewController);
- classe *Scene*, per le scene (in relazione ad Application);
- classe *ChildViewController*, per rappresentare i view controller inseriti nelle scene (in relazione a Scene e a ViewController);
- enumerazione *LayoutEdge*, per i diversi lati dei controlli (in relazione a PositionConstraint);
- enumerazione *ChainType*, per i diversi tipi di catene (in relazione a ControlChain);
- enumerazione *ChainAxis*, per i diversi assi delle catene (in relazione a ControlChain);
- enumerazione *SceneType*, per i diversi tipi di scene (in relazione a Scene).

Classi aggiornate:

- classe *Application*: aggiunta relazione con Scene;

- classe *ViewController*: eliminato l'attributo `launcher` e aggiunte relazioni con ControlChain e Container;
- classe *UiPhoneControl*: aggiunti gli attributi `defaultWidth`, `defaultHeight`, `indexInChain` e `weight` e le relazioni con PositionConstraint, DimensionConstraint, ControlChain e gli UiPhoneControl della propria catena; eliminati gli attributi `width` e `height`;
- classe *Navigation*: sostituito l'attributo `destination` con le relazioni con ViewController e Scene, aggiunto l'attributo `contextScene`;
- classi *Button*, *ListView*, *GridView*: sostituita la relazione a ClickListener (classe eliminata) con relazione uno-a-molti con Navigation.

4.4.2 Model Check e Estensioni

Avendo modificato il meta model, sono poi stati aggiornati conseguentemente i controlli di Check: sono state create 30 nuove asserzioni che verificano la correttezza strutturale di vincoli di posizione e dimensione, catene di controlli e scene; sono stati anche definiti due nuovi metodi Java, utilizzati da queste asserzioni attraverso Xtend.
Oltre al file `Checks.chk` ed alla relativa estensione Xtend, sono stati anche modificati i file del package `app_extensions`: sono stati implementati tre nuovi metodi Java, utilizzati dalle regole di traduzione di Xpand per Android e iOS. I primi due metodi servono a verificare se un UiPhoneControl abbia rispettivamente la posizione o la dimensione vincolata da constraint; l'ultimo metodo serve a verificare se una scena contenga controlli di tipo `Button`, `ListView` o `GridView`, cioè quei controlli che hanno una relazione con oggetti Navigation.

4.4.3 Generazione del codice per Android

Terminate le modifiche comuni sia alla traduzione per Android sia alla traduzione per iOS, si entra ora nel dettaglio della generazione del codice per il sistema Android. Nelle successive sezioni verranno presentati gli interventi fatti per tradurre in codice tutte le nuove funzionalità introdotte in Protocode 4.0.

4.4.3.1 Vincoli di posizione, dimensione e catene di controlli

Al fine di generare il codice relativo ai vincoli di posizione, vincoli di dimensione e catene di controlli, è stato modificato il file `XMLFiles.xpt`: il layout radice di tutti i file di layout è stato cambiato da RelativeLayout a ConstraintLayout; successivamente è stata completamente riscritta la regola di traduzione della posizione e dimensione dei controlli, basandola sui nuovi elementi del model. Per quanto riguarda i vincoli di posizione, in figura 4.3 è riportato uno schema della traduzione; sono elencati gli 8 possibili vincoli del ConstraintLayout che permettono l'allineamento di tutte le coppie di lati specificabili con Protocode, eccetto `centerX` e `centerY`: la traduzione di un vincolo sul centro infatti, a differenza degli altri lati, avviene per mezzo di due vincoli sui lati dell'asse corrispondente.

In altre parole, un vincolo su `centerX` si traduce in due constraint, uno su `start` e l'altro su `end`; un vincolo su `centerY` si traduce in constraint su `top` e `bottom`. Ad esempio, un PositionConstraint del model che allinea un controllo al centro orizzontale della finestra parent viene tradotto con due vincoli, uno *Start-to-Start*, l'altro *End-to-End*, tra il controllo e la finestra, indicata in Android con la keyword `parent`.

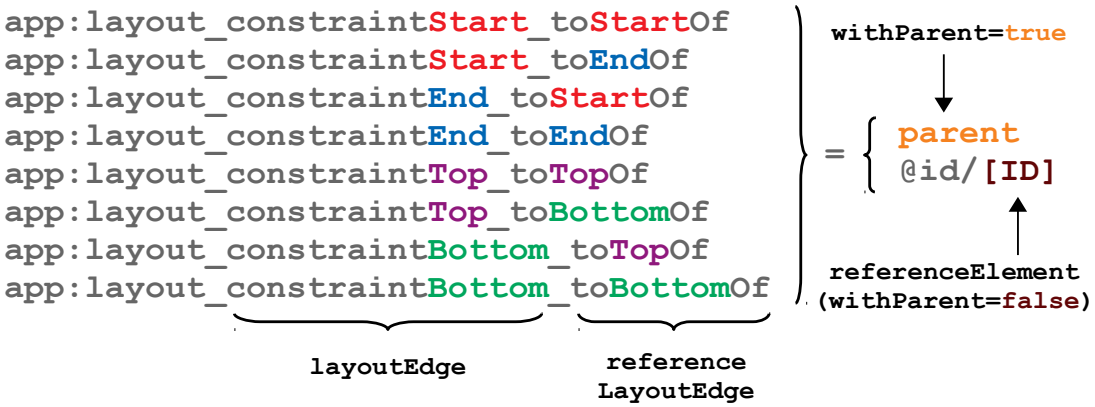

Figura 4.3: Traduzione di PositionConstraint in Android

La traduzione dei vincoli di dimensione invece segue lo schema riportato in figura 4.4: si può notare che, per quanto riguarda i vincoli fissi e la dimensione di default (utilizzata in caso di mancanza di qualsiasi tipo di vincolo su una specifica dimensione), essi sono tradotti utilizzando gli attributi standard `android:layout_width` e `android:layout_height`; la traduzione dei vincoli percentuali e di rapporto, invece, sfrutta i vincoli presenti ancora una volta nel ConstrainLayout.

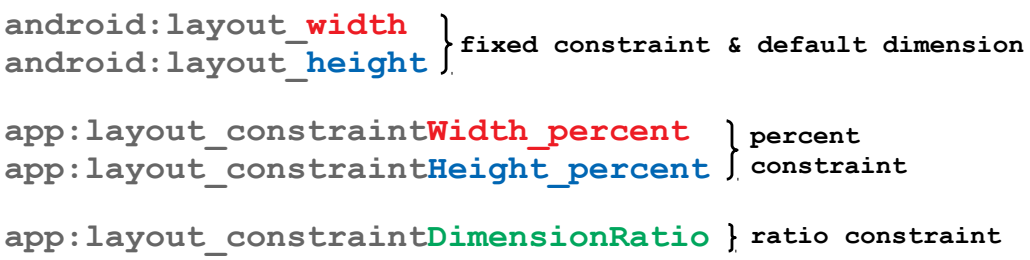

Figura 4.4: Traduzione di DimensionConstraint in Android

Per quanto riguarda le ControlChain, esse si traducono in un'insieme di coppie di vincoli bidirezionali; ad esempio, una catena orizzontale formata da tre controlli avrà:

- un vincolo *Start-to-Start* tra il primo controllo e la finestra parent;
- due vincoli, uno *End-to-Start* tra il primo controllo e il secondo, l'altro *Start-to-End* tra il secondo controllo e il primo;

- due vincoli, uno *End-to-Start* tra il secondo controllo e il terzo, l'altro *Start-to-End* tra il terzo controllo e il secondo;
- un vincolo *End-to-End* tra il terzo controllo e la finestra parent.

Oltre a questi vincoli di allineamento, si aggiungono gli attributi che caratterizzano le catene dei diversi tipi. Si riportano di seguito questi attributi, esemplificati nel caso di catene orizzontali (per quelli delle catene verticali basta sostituire banalmente *Horizontal* con *Vertical* e "sinistro" con "superiore"):

- **tutte** le catene: il primo controllo della catena ne specifica il tipo, attraverso l'attributo `app:layout_constraintHorizontal_chainStyle`.
- Catene **packed**: il primo controllo della catena ne specifica il parametro `bias`, attraverso l'attributo `app:layout_constraintHorizontal_bias`; ogni controllo inoltre specifica il parametro `spacing` della catena, attraverso il proprio margine sinistro.
- Catene **weighted**: ogni controllo specifica il proprio parametro `weight`, attraverso l'attributo `app:layout_constraintHorizontal_weight` e, come per le catene *packed*, specifica anche il parametro `spacing` della catena, attraverso il proprio margine sinistro.

Si ricorda infine che, nel caso la posizione di un controllo non sia vincolata su uno specifico asse (niente vincoli di posizione né il controllo fa parte di una catena su quell'asse), ne viene creato automaticamente uno a partire dagli attributi `posX` (asse orizzontale) o `posY` (asse verticale).

4.4.3.2 View controller, scene e menù: codice XML

Come anticipato nella sezione 3.4.6 del capitolo riguardante Protocode, si è scelto di dare il seguente significato a view controller e scene:

- i **view controller** rappresentano le porzioni di interfaccia utilizzabili per comporre una schermata (e riutilizzabili all'interno di diverse schermate); in Android sono quindi tradotti in Fragment.
- Le **scene** rappresentano le schermate dell'applicazione, ottenute dalla composizione dei view controller; in Android sono tradotte in Activity.

Sia Fragment che Activity consistono in file di layout che ne specificano l'aspetto e classi Java che ne implementano il comportamento; tuttavia, anche da questo punto di vista è stata effettuata una divisione funzionale dei due componenti:

- i **Fragment** conterranno, all'interno del proprio file di layout, tutti gli elementi grafici veri e propri dell'interfaccia dell'applicazione (vale a dire i controlli) e ne implementeranno il comportamento nella relativa classe Java;
- le **Activity** conterranno gli elementi grafici basilari di qualunque schermata (ad esempio l'ActionBar o il menù) e si occuperanno di definire come i Fragment saranno visualizzati nella schermata, adattando la visualizzazione al tipo di dispositivo corrente.

Di seguito verrà descritta la traduzione di view controller, scene e menù in file di layout XML (che avviene per mezzo di regole presenti nel file `XMLFiles.xpt`); la traduzione degli stessi nelle relative classi Java verrà spiegata nella sezione successiva (essa avviene invece tramite regole contenute nel file `activities.xpt`).
L'organizzazione dei file di layout generati è stata definita seguendo le linee guida di Android: il layout del menù viene generato in un file a parte nominato `app_menu.xml`; il layout di ciascun Fragment viene generato in un file chiamato `fragment_VC_NAME.xml`; per i layout delle Activity invece sono state seguite le seguenti regole:

- le Activity **senza menù** (generate a partire da scene che non sono raggiunte dal Navigation di un MenuItem) hanno un solo file di layout, nominato `activity_SCENE_NAME.xml`, che specifica l'aspetto della ActionBar e lascia il resto dello spazio libero per i fragment che saranno inseriti;
- le Activity **con menù** (generate a partire da scene che sono raggiunte dal Navigation di un MenuItem) hanno due file di layout:
  - `app_bar_SCENE_NAME.xml`: definisce l'aspetto della ActionBar e lascia il resto dello spazio libero per i fragment;
  - `activity_SCENE_NAME.xml`: ha come layout radice il DrawerLayout, che serve per visualizzare il menù (inserisce nella ActionBar il pulsante per richiamarlo), e include il precedente file che specifica il contenuto dell'Activity e il file `app_menu.xml` che specifica l'aspetto del menù.

A questo schema generale va inoltre aggiunto che:

- le Activity generate da scene di tipo `multiVC` hanno un ulteriore file di layout, `content_SCENE_NAME.xml`, che specifica la posizione e dimensione di ciascun fragment incluso (le Activity generate dagli altri tipi di scene non ne hanno bisogno dal momento che contengono un oggetto ViewPager, inserito nello spazio libero lasciato dal file di layout che specifica la ActionBar, che si occupa di visualizzare un Fragment alla volta);
- le Activity generate da scene che hanno un tipo diverso per smartphone e per tablet avranno alcuni dei file di layout appena descritti duplicati in due versioni, una appunto per telefono e una per tablet: Android permette infatti di definire più file di layout con lo stesso nome (quindi utilizzati per la stessa Activity) ma all'interno di cartelle differenti, una per il layout da applicare nel caso "generale" e le altre per i layout da applicare su dispositivi con caratteristiche specifiche. Le linee guida di Android definiscono tablet un dispositivo con larghezza (in *dp*) maggiore o uguale a 600 *dp* e per questo motivo MobileCodeGenerator, per applicazioni con almeno un'Activity differenziata su smartphone e tablet, genera due cartelle di layout, una generale dal nome `layout` e l'altra specifica per i tablet, dal nome `layout-sw600dp`; il nome non è arbitrario ma assegnato secondo una sintassi specifica e significa *smaller width* uguale a 600 *dp*, cioè i file di layout in questa cartella sono utilizzati per dispositivi con appunto una larghezza di almeno 600 *dp*. Ogni Activity con `typeSmartphone` diverso da

`typeTablet` avrà dunque alcuni file di layout duplicati in due versioni e posizionati nelle due directory appena definite; il sistema si occuperà automaticamente, a runtime, di scegliere il layout appropriato al dispositivo sul quale l'applicazione è in esecuzione.

La figura 4.5 mostra uno schema riassuntivo dei file di layout generati.

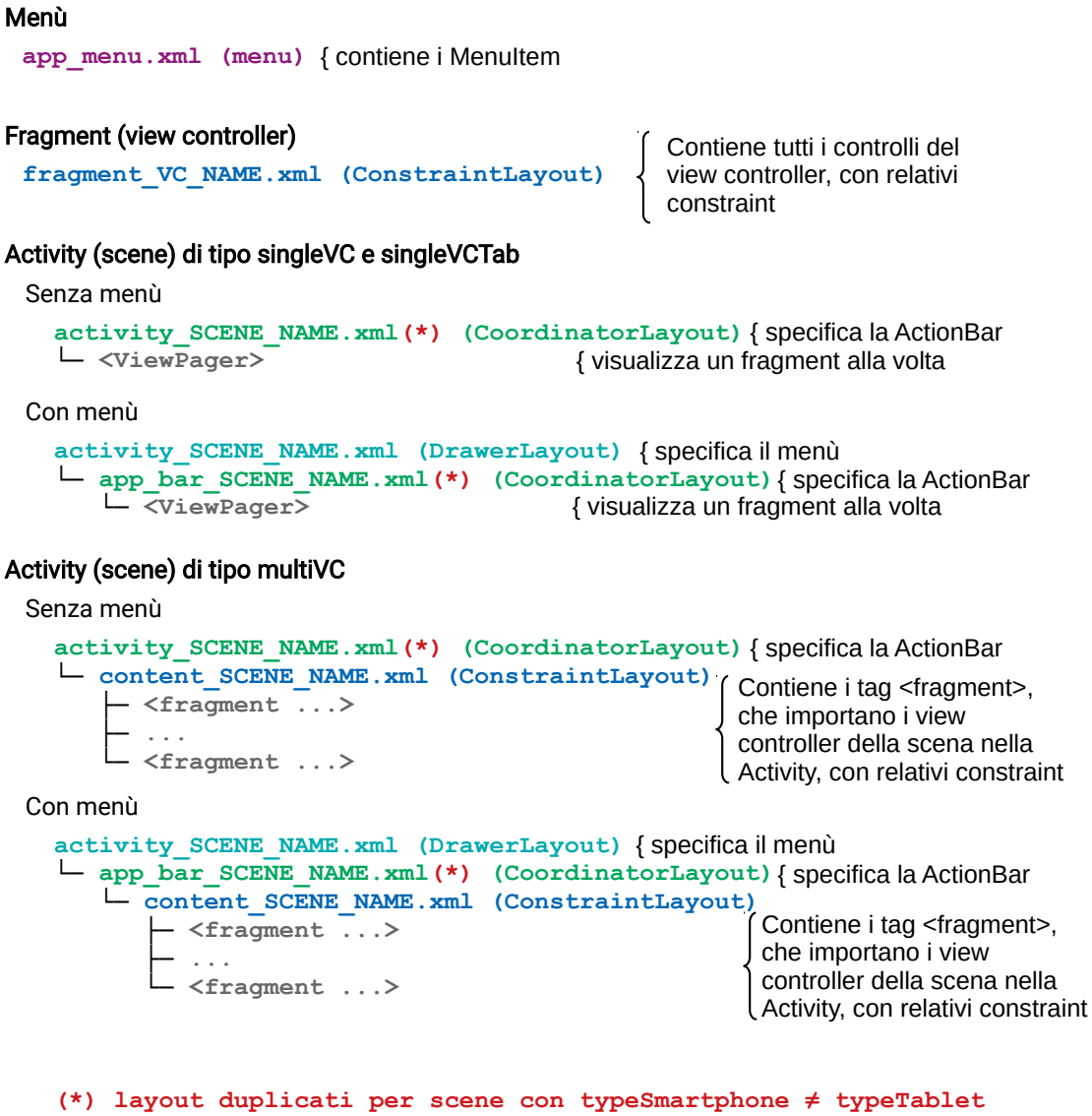

Figura 4.5: File di layout di un'Activity per Android generati da MCG 4.0

4.4.3.3 View controller, scene e menù: codice Java

Per quanto riguarda i view controller, la generazione del codice Java non è stata modificata: essi, fino alla versione 3.0 di MobileCodeGenerator, venivano tradotti in Fragment o in Activity, a seconda che fossero o no raggiungibili dal menù (il quale era definito con una TabBar); la regola di traduzione in Activity è stata semplicemente messa da parte (non è stata eliminata perchè potrebbe essere utile

in sviluppi futuri, ma non viene espansa nel workflow di MobileCodeGenerator 4.0) ed è stata utilizzata la regola di traduzione in Fragment per ogni view controller.
Per quanto riguarda invece le scene, è stata creata una nuova regola per la traduzione in Activity; di seguito sono riportati i principali metodi implementati e la relativa funzione:

- metodo *onCreate*: esegue tutte le operazioni di setup dell'interfaccia; esso gestisce il fatto che il tipo della scena da cui la Activity è stata generata può essere diverso per smartphone e per tablet nel modo seguente (si ricordi che Android sceglie in automatico quale layout applicare a seconda del dispositivo):
  - prima di tutto, viene cercato nel layout il ViewPager chiamando `findViewById` sull'id utilizzato per quest'ultimo nel file di layout.
  - Se il ViewPager non è presente significa che il tipo di scena per il dispositivo corrente è `multiVC`; in questo caso non c'è altro da fare nel setup dell'interfaccia, dal momento che nel layout sono specificati i tag `<fragment>` e il sistema provvede a caricare i relativi Fragment in automatico.
  - Se il ViewPager invece è presente, ne vengono effettuate le operazioni di setup chiamando i metodi opportuni; inoltre, ciò sta a significare che il tipo di scena per il dispositivo corrente è `singleVC` oppure `singleVCTab`: si procede quindi a cercare, sempre attraverso il metodo `findViewById`, il TabLayout che rappresenta la tab bar.
  - Se il TabLayout non è presente, il tipo di scena per il dispositivo corrente è `singleVC` e non c'è altro da fare nel setup dell'interfaccia; al contrario, se il TabLayout viene trovato, vengono chiamati i metodi per farne il setup e ciò sta anche a significare che il tipo di scena per il dispositivo corrente è `singleVCTab`.
  - In tutti i casi, il tipo di scena che viene dedotto è salvato in un attributo dell'Activity chiamato `layoutType` e implementato con delle costanti definite in un file `Constants` generato ad-hoc da MobileCodeGenerator; in questo modo, qualsiasi metodo dell'Activity può conoscere quali elementi grafici sono presenti nel layout, adattando il suo funzionamento in base a questo.

- Metodo *onNavigationItemSelected*: questo metodo serve per gestire l'interazione dell'utente col menù; esso esegue cioè le operazioni necessarie a terminare l'Activity corrente e creare e visualizzare quella selezionata dall'utente tramite il menù.

- Metodo *onFragmentNavigationInteraction*: dal momento che un Fragment può essere presente in diverse Activity, non è possibile definire direttamente all'interno di esso l'azione che deve essere eseguita quando l'utente interagisce con un controllo contenuto in quel Fragment. L'unica azione modellabile

con Protocode è quella di navigazione (con destinazione un Fragment della stessa Activity oppure un'altra Activity) e l'evento di navigazione può avvenire solo se si preme su un `Button` o su un elemento di una `ListView` o `GridView`: per implementare questa interazione è stata definita un'interfaccia Java, chiamata `OnFragmentNavigationInteractionListener`, che contiene appunto il metodo `onFragmentNavigationInteraction` e che viene implementata dalla Activity contenente il Fragment che a sua volta contiene un bottone, una lista o una griglia; questo metodo esegue l'operazione di navigazione, se definita, ed è chiamato dal Fragment che contiene il controllo con cui l'utente ha interagito (fig. 4.6).

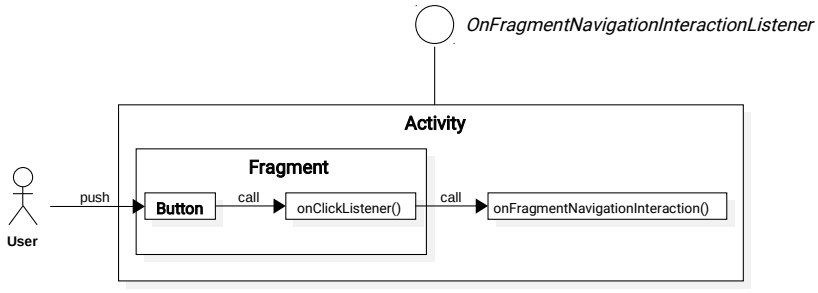

Figura 4.6: Interazione che genera un evento di navigazione in Android

Oltre a questi principali metodi, per scene che hanno almeno uno dei due tipi uguale a `singleVC` o `singleVCTab`, nella corrispondente Activity generata viene anche aggiunta la inner-class (definizione di classe Java contenuta in un'altra classe) SectionsPagerAdapter: essa estende FragmentPagerAdapter ed è utilizzata per creare le istanze dei fragment che il ViewPager andrà a visualizzare (per questo motivo nel caso di scene con `typeSmartphone = typeTablet = multiVC` non viene aggiunta).

4.4.3.4 Aggiornamento componenti di AndroidStudio

L'ultimo intervento su MobileCodeGenerator riguardante la generazione del codice Android è consistito nell'aggiornamento dei file di progetto di Gradle (ultima versione di Gradle: 3.0.1, ultima versione build tools: 26.0.2) e dei riferimenti alle librerie di supporto di Android e dei Google Play Services.

4.4.4 Generazione del codice per iOS

Terminato il lavoro svolto sulla parte relativa ad Android, si è proceduto ad estendere MobileCodeGenerator per la generazione del codice per iOS; nelle sezioni successive vengono riportati tutti gli interventi eseguiti a questo fine. E' utile però fare una premessa sulla scelta compiuta riguardo alla traduzione di view controller e scene, entrambi tradotti in UiViewController ma con alcune differenze: per quanto riguarda i primi, si è scelto di tradurne l'aspetto grafico all'interno nel file `Main.storyboard` e il comportamento nella relativa classe Swift,

come veniva fatto nelle precedenti versioni di MobileCodeGenerator; per quanto riguarda le scene invece, si è optato per una traduzione completa in codice Swift, sia della parte grafica sia della parte logica. Il motivo è il seguente: i view controller contengono la parte vera e propria dell'interfaccia dell'applicazione, per cui è utile poterli visualizzare e modificare con l'InterfaceBuilder di XCode. Le scene invece, dal punto di vista grafico, non contengono particolari elementi, se non la barra di navigazione (e le view dei view controller figli che però, essendo inserite a runtime a seconda del tipo di dispositivo su cui l'applicazione viene eseguita, non sarebbero visualizzate nell'InterfaceBuilder); per questo motivo, averne un'anteprima grafica nell'InterfaceBuilder non avrebbe comportato nessun vantaggio ma avrebbe anzi reso inutilmente più affollata la storyboard.

4.4.4.1 Vincoli di posizione e dimensione

Le regole di generazione dei vecchi vincoli grafici si trovavano nel file `ViewConstraints.xpt`: il primo intervento sulla parte di traduzione per iOS e consistita nella completa riscrittura di queste regole sulla base dei nuovi vincoli di posizione e dimensione. In figura 4.7 è riportato lo schema della traduzione, valido sia per PositionConstraint che per DimensionConstraint; in particolare, si noti che per i vincoli di posizione il valore `constant`, utilizzato dai constraint di iOS per indicare lo spazio in *dp* che separa i due lati specificati in `firstAttribute` e `secondAttribute`, è ricavato dalla somma dei margini e del padding. Si ricorda inoltre che in alcuni tipi di constraint è anche presente un campo `multiplier`, in sostituzione o aggiunta al campo `constant`: ad esempio, un DimensionConstraint di tipo percentuale sulla larghezza viene tradotto in un constraint con `multiplier` uguale a `widthPercent`.

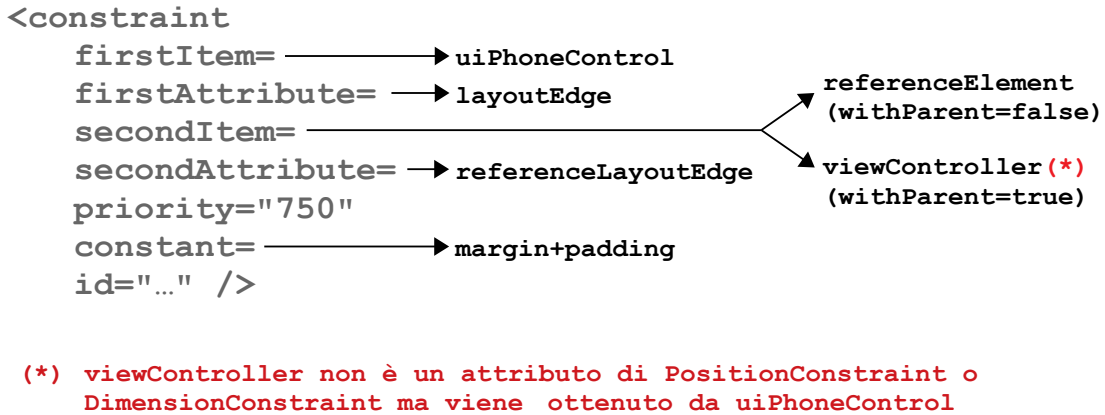

Figura 4.7: Traduzione dei constraint in XML per la storyboard di iOS

Come specificato nella premessa, si è scelto di tradurre le scene del modello XMI completamente in codice Swift: questo ha reso necessario formulare altre regole, inserite nel file `ViewControllers.xpt`, per la traduzione dei vincoli in codice Swift, dal momento che le regole contenute nel file `ViewConstraints.xpt` eseguono la traduzione in codice XML inserito poi nella storyboard (`Main.storyboard`). Un esempio di traduzione di un vincolo di posizione in Swift è illustrato nella

figura 4.8 , che mostra anche un confronto con lo stesso vincolo formulato in notazione XML per la storyboard.

```
<constraint firstItem="button" firstAttribute="bottom"
    secondItem="label" secondAttribute="top"
    priority="750" constant=20 id="…" />
```



```
button.bottomAnchor.constraint(
    equalTo: label.topAnchor, constant: 20)
```

Figura 4.8: Confronto di constraint tradotto in XML e in Swift per iOS

4.4.4.2 Catene di controlli

Dopo la traduzione dei vincoli, sono state create le regole per generare il codice relativo alle catene di controlli. Queste regole traducono le catene in un insieme di vincoli che saranno inseriti nella storyboard (visualizzabili nell'InterfaceBuilder di XCode) e sono state quindi aggiunte al file `ViewConstraints.xpt` insieme alle regole di traduzione dei vincoli; non è stato necessario formulare altre regole per la traduzione delle catene in vincoli espressi in codice Swift dal momento che i Container di Protocode, che rappresentano le view principali dei view controller inseriti in una scena, non sono inseribili in ControlChain. Si riporta di seguito la descrizione di come viene eseguita la traduzione di ciascun tipo di catena in vincoli di iOS; per comodità si esemplifica la traduzione di una catena orizzontale (la corrispondente traduzione di una catena verticale si può ottenere sostituendo il lato superiore a quello sinistro, il lato inferiore a quello destro e l'altezza alla larghezza):

- catene **spread**: le catene di questo tipo, come già illustrato nella sezione 3.4.3, distribuiscono lo spazio non occupato dai controlli equamente per separare i controlli stessi; sfortunatamente, in iOS è solo possibile definire vincoli di posizione che impongano una distanza fissa tra due oggetti e non una distanza che vari a seconda dello spazio a disposizione. Per ovviare a questa mancanza, nell'effettuare la traduzione di catene *spread* si inseriscono delle view vuote e trasparenti, qui denominate *ghost view*, che vengono utilizzate per separare i controlli: in questo modo, imponendo che tutte le ghost view abbiano la stessa larghezza, lo spazio che separa ciascuna coppia di controlli risulta uguale. La traduzione avviene dunque nel modo seguente:
  - per una catena formata da N controlli vengono create $N + 1$ ghost view, ciascuna posizionata tra una coppia di controlli oppure tra il primo/ultimo controllo della catena e il lato sinistro/destro della parent view.

- Vengono creati $N + (N + 1) + 1 = 2(N + 1)$ vincoli di posizione: i primi N allineano il lato sinistro di ciascun controllo al lato destro della ghost view che lo precede; i successivi $N + 1$ allineano il lato sinistro di ciascuna ghost view al lato destro di ciascun controllo che la precede (la prima ghost view viene allineata al lato sinistro della parent view); l'ultimo vincolo allinea il lato destro dell'ultima ghost view al lato destro della parent view.
- Vengono creati $(N+1)-1 = N$ vincoli di dimensione, che impongono che la larghezza di ciascuna ghost view eccetto la prima sia uguale alla larghezza della prima.

La figura 4.9 mostra uno schema riassuntivo di quanto appena illustrato.

- Catene **spread inside**: la traduzione di catene di questo tipo avviene esattamente come quella delle catene di tipo *spread*, con l'unica differenza che, per una catena di N controlli, le ghost view sono stavolta $N - 1$ e quindi i vincoli di posizione generati sono $N + (N - 1) + 1 = 2N$ mentre i vincoli di dimensione sono $(N - 1) - 1 = N - 2$.

- Catene **packed**: per tradurre questo tipo di catene è ancora necessario aggiungere delle ghost view, ma in questo caso ne bastano due, qualsiasi sia il numero di controlli della catena. La traduzione avviene nel seguente modo (esemplificato in figura 4.10 con *bias*=0.3):
  - la prima ghost view è posizionata a sinistra dei controlli, tra il primo e il lato sinistro della parent view; allo stesso modo, la seconda ghost view è posizionata a destra, tra l'ultimo controllo della catena e il lato destro della parent view.
  - Vengono creati $N+2+1 = N+3$ vincoli di posizione: i primi N distanziano il lato sinistro di ciascun controllo dal lato destro del controllo che lo precede di un numero di *dp* uguale a `spacing` (eccetto il primo controllo che è vincolato al lato destro della prima ghost view); i successivi due allineano rispettivamente il lato sinistro della prima ghost view al lato sinistro della parent view e il lato sinistro della seconda ghost view al lato destro dell'ultimo controllo della catena; l'ultimo vincolo allinea il lato destro della seconda ghost view al lato destro della parent view.
  - Viene creato un unico vincolo di dimensione, che lega le larghezze delle due ghost view secondo la seguente equazione ($width[0]$ rappresenta la larghezza della prima ghost view, cioè quella di sinistra, mentre $width[1]$ si riferisce all'altra): $width[1] = width[0] \cdot \frac{(1-bias)}{bias}$.

- Catene **weighted**: questo è l'unico tipo di catena per cui non è necessario introdurre delle ghost view. Per ciascun controllo inserito nella catena, viene creato un vincolo di posizione che distanzia di un numero di *dp* uguale al parametro `spacing` il suo lato sinistro dal lato destro dell'elemento che lo precede nella catena (il primo elemento è vincolato al lato sinistro della

parent view). Oltre a questi N vincoli di posizione (con N uguale al numero di controlli della catena) ne viene aggiunto un $(N+1)$-esimo che vincola il lato destro dell'ultimo controllo della catena al lato destro della parent view (sempre imponendo una distanza uguale a `spacing`). Per tradurre i pesi (`weight`) invece vengono creati $N-1$ vincoli di dimensione, che legano la larghezza di ciascun controllo eccetto il primo a quella del primo secondo la seguente equazione ($width[i]$ e $weight[i]$ significano rispettivamente larghezza e peso del controllo i-esimo): $width[i] = width[1] \cdot \frac{weight[i]}{weight[1]}$.

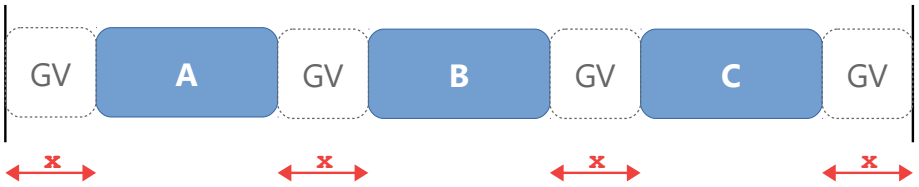

Figura 4.9: Traduzione di una catena di tipo spread in iOS

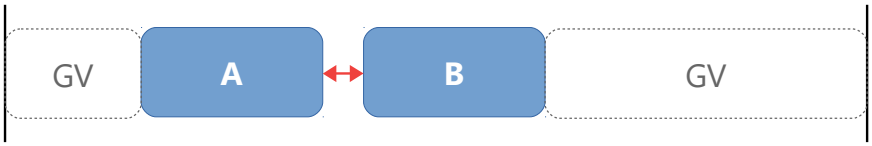

Figura 4.10: Traduzione di una catena di tipo packed in iOS

Il risultato ottenuto con queste regole di traduzione è esattamente uguale a quello ottenuto utilizzando le Control Chain del ConstrainLayout di Android.

4.4.4.3 Menù

Terminata la creazione delle regole per la traduzione delle catene di controlli, è stato rimosso il TabBarViewController che era utilizzato come menù dell'applicazione e sono state formulate nuove regole (inserite nel file `ViewControllers.xpt`) per la generazione del codice relativo a quest'ultimo; come per le catene, in iOS non esiste un menù simile al Navigation Drawer di Android, per cui è stato necessario implementarlo completamente da zero. Di seguito vengono riportati e descritti i principali file generati da MobileCodeGenerator relativi al menù:

- *MenuInteractionProtocol.swift*: la classe contenuta in questo file specifica il protocollo che ogni UiViewController in grado di visualizzare il menù deve implementare; esso definisce un unico metodo, `menuInteraction()`, che viene chiamato dall'UiViewController che rappresenta il menù quando l'utente interagisce con esso.

- *MenuViewController.swift*: questo è il file che contiene la classe che rappresenta il menù; essa estende UiTableViewController, visualizzando un elenco di tutti gli UiViewController (tradotti dalle scene) che possono essere raggiunti. Inoltre, `MenuViewController` ha un attributo `delegate` di

tipo `MenuInteractionProtocol`: ogni volta che un UiViewController visualizza il menù, imposta se stesso come `delegate`; `MenuViewController` quindi, quando l'utente seleziona un elemento dell'elenco, chiama il metodo `menuInteraction()` del `delegate` il quale, prima fa scomparire il menù chiamando `dismiss()`, poi esegue la transizione all'UiViewController selezionato dall'utente tramite l'elenco del menù. La figura 4.11 riassume questo schema di funzionamento.

- *MenuPresentationManager.swift*: la classe contenuta in questo file implementa il protocollo `UIViewControllerTransitioningDelegate`, definito nella libreria di sistema UIKit e svolge la funzione di coordinamento della transizione da un UiViewController a `MenuViewController` e viceversa. Essa funziona in questo modo:
  - un UiViewController che vuole utilizzarla la setta come proprio `transitioningDelegate` (questo attributo è predefinito per la classe UiViewController);
  - quando l'UiViewController effettua una transizione chiamando la funzione `present`, il `transitioningDelegate` gli fornisce un oggetto di tipo UiPresentationController e un oggetto di tipo UiPresentationAnimator che ne coordinano la transizione.

  Nel caso del menù, `MenuViewController` setta il proprio attributo `transitioningDelegate` ad un'istanza di `MenuPresentationManager`: in questo modo, quando il menù viene presentato o dismesso, `MenuPresentationManager` ne coordina la presentazione e dismissione fornendo i due oggetti appena spiegati, la cui implementazione è contenuta nei due successivi file che vengono presentati.

- *MenuPresentationController.swift*: questo file contiene la prima classe necessaria alla presentazione del menù, la quale estende la classe di sistema UiPresentationController. Essa definisce la posizione del menù (allineamento a sinistra nella finestra), la sua dimensione (altezza uguale all'altezza della finestra, larghezza uguale ai due terzi della larghezza della finestra) e crea una view grigio-trasparente che ombreggia la restante parte di finestra (la parte non occupata da `MenuViewController`) e che, se premuta, fa scomparire il menù e tornare all'UiViewController che era in esecuzione prima della comparsa del menù.

- *MenuPresentationAnimator.swift*: la classe contenuta in questo file è la seconda che serve al coordinamento della transizione ed estende la classe di UIKit UiPresentationAnimator. Essa definisce le animazioni che il menù esegue quando compare e scompare, ovvero la comparsa dal margine sinistro della finestra e la scomparsa dalla stessa parte.

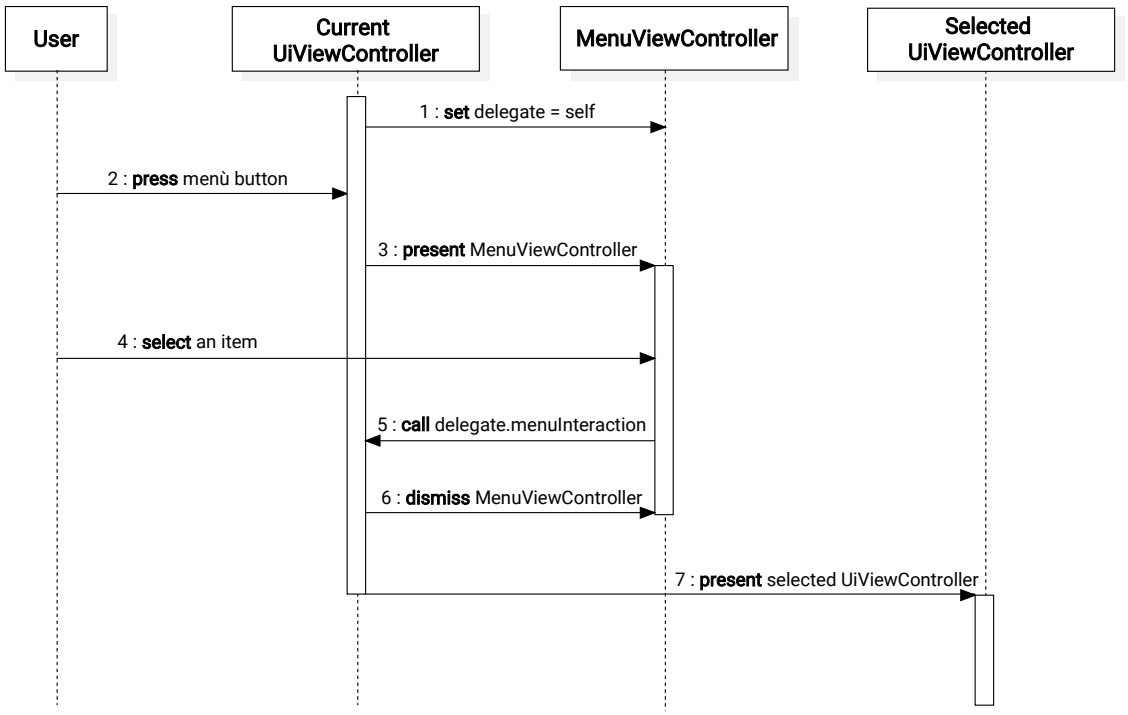

Figura 4.11: Interazione con il menù in iOS

4.4.4.4 View controller e scene

Come già anticipato nella sezione 3.4.6 del capitolo riguardante Protocode e richiamato più volte in questo stesso capitolo, sia i view controller che le scene del modello vengono tradotte in UiViewController; di seguito ci si riferirà ai primi come agli UiViewController figli, dal momento che essi saranno inseriti tra i `childViewControllers` degli UiViewController tradotti dalle scene, a cui ci si riferirà con il termine di UiViewController padri (l'attributo `childViewControllers` è presente di default nell'implementazione della classe UiViewController).
Per quanto riguarda la generazione del codice degli UiViewController figli, le regole preesistenti sono state lasciate sostanzialmente invariate, a meno di modifiche minori; per quanto riguarda invece la traduzione delle scene in UiViewController padri, sono state formulate nuove regole di traduzione, inserite insieme a quelle per la generazione degli UiViewController figli nel file `ViewControllers.xpt`. Di seguito sono riportate le principali funzioni implementate nella classe generata da ciascuna scena (le prime due sono funzioni già presenti nella classe UiViewController, di cui è stato fatto `override`, mentre le altre sono create ex-novo):

- funzione *init*: è la funzione chiamata da iOS quando viene creata un'istanza del view controller; essa va a sua volta a creare un'istanza di ciascun UiViewController figlio.

- Funzione *viewDidLoad*: questa funzione è chiamata dal sistema quando la view principale del view controller viene caricata. Essa esegue il codice di setup del pulsante del menù, nel caso sia presente, e successivamente chiama le funzioni per il setup dell'intera interfaccia: se il tipo di scena che si sta traducendo è uguale sia su smartphone che su tablet, la `viewDidLoad` chiama la funzione `setupUI` in ogni caso; se invece il tipo di scena è diverso per i due tipi di dispositivo, la `viewDidLoad` testa il tipo di device su cui l'applicazione è in esecuzione e, a seconda del tipo, chiama la funzione `setupPhoneUI` oppure la funzione `setupTabletUI` (il test viene eseguito sfruttando l'attributo `current` della classe `UIDevice` che rappresenta appunto il dispositivo corrente).

- Funzione *setupUI*, *setupPhoneUI* e *setupTabletUI*: queste tre funzioni, come appena accennato, servono ad eseguire il setup dell'interfaccia; non sono mai presenti tutte e tre, ma solo la prima nel caso di scena con `typeSmartphone` uguale a `typeTablet` oppure solo le ultime due in caso contrario. Il codice in esse contenuto va ad aggiungere ogni UiViewController figlio all'array `childViewControllers` del padre; successivamente, nel caso di scene `singleVC` o `singleVCTab`, va a visualizzare il primo view controller della scena, mentre nel caso di scene `multiVC` va a visualizzare le view principali di tutti gli UiViewController figli, impostandone anche i vincoli di posizione e dimensione (che nel modello creato con Protocode erano i vincoli definiti sui Container).

- Funzioni *show[VC_NAME]VC*: viene generata una di queste funzioni per ogni view controller di una scena che ha almeno uno dei due tipi,`typeSmartphone` o `typeTablet`, uguale a `singleVC` o `singleVCTab`. Questa funzione serve a mostrare l'UiViewController VC_NAME: chiamando `show[VC_NAME]VC` infatti, la view principale dell'UiViewController figlio correntemente visualizzato viene tolta e sostituita con la view principale dell'UiViewController VC_NAME.

A queste funzioni, definite per ogni UiViewController padre, se ne aggiungono potenzialmente altre tre:

- la funzione *menuViewController*, definita nel protocollo `MenuViewControllerDelegate`, che è presente solo se la scena può visualizzare il menù (serve a gestire la transizione dopo che l'utente ha selezionato un elemento del menù, come illustrato nella sezione precedente);

- la funzione *tabBar*, definita nel protocollo `UiTabBarDelegate`, presente solo se il tipo di scena su smartphone o su tablet è `singleVCTab` (serve a gestire l'interazione dell'utente con la TabBar);

- la funzione *childViewControllerInteraction*, definita nel protocollo `ChildViewControllerInteractionProtocol` (generato anch'esso da MobileCodeGenerator), presente solo se almeno un view controller della scena contiene un elemento che può generare eventi di navigazione, vale a dire i soliti

`Button`, `ListView` e `GridView`; lo schema di funzionamento è lo stesso usato per l'interazione tra Fragment e Activity in Android, con la differenza che in Java si parla di interfacce mentre in Swift di protocolli.

Oltre alla formulazione delle regole di traduzione delle scene in UiViewController padri contenuta nel file `ViewControllers.xpt`, è stata anche necessaria una modifica della regola di generazione del file *AppDelegate.swift* contenuta in `Default-Files.xpt`: dal momento che nella storyboard sono presenti solo gli UiViewController figli, non è possibile impostare il view controller iniziale (il primo visualizzato dall'applicazione dopo l'apertura) in quest'ultima. E' stata quindi estesa la regola di traduzione della funzione `application` della classe `AppDelegate`, che va a generare il codice per visualizzare il view controller iniziale; le istruzioni aggiunte sono:

- la creazione di un'istanza di un UiNavigationController (che visualizza la barra di navigazione presente nella parte alta della finestra dell'applicazione);
- la creazione di un'istanza dell'UiViewController di partenza dell'applicazione (quello tradotto dalla scena che ha l'attributo `launcher = true`);
- l'aggiunta dell'UiViewController di partenza all'array `viewControllers` del navigation controller;
- l'impostazione del navigation controller come view controller radice della finestra e la chiamata alla funzione che rende quest'ultima visibile (istruzioni `self.window!.rootViewController = nav` e `self.window?.makeKeyAndVisible()`).

4.4.4.5 Aggiornamento di XCode

L'ultimo intervento eseguito è stato l'aggiornamento della componente di MobileCodeGenerator relativa ai file di progetto, derivata dal fatto che, successivamente a quando la versione 3.0 è stata rilasciata, Apple ha sviluppato una nuova versione di XCode (l'ultima stabile è la 9.2); è quindi stato conseguentemente aggiornato il file di progetto di XCode, generato dalle regole contenute in `Xcodeproj.xpt`. Per quanto riguarda Swift, si è deciso di non aggiornare il codice generato dalla versione 3 alla versione 4; il motivo è che il compilatore Swift 4 è in grado di compilare codice scritto in Swift 3 e, inoltre, XCode 9 ha una funzionalità che permette di tradurre automaticamente codice Swift 3 in codice Swift 4.

Capitolo 5

Valutazione

Questo capitolo presenta una valutazione del lavoro svolto sui due tool descritti nei capitoli precedenti, la cui utilità e qualità è stata testata attraverso lo sviluppo di un'applicazione completa: per prima cosa è stata effettuata la prototipizzazione di tutti gli elementi dell'applicazione definibili con la nuova versione si Protocode; successivamente, utilizzando MobileCodeGenerator, sono stati generati automaticamente il codice e i progetti per AndroidStudio e XCode a partire dal file di modello; infine è stato effettuato il completamento delle due applicazioni per Android e iOS modificando manualmente gli elementi che necessitavano di ulteriori aggiustamenti ed implementando la logica applicativa. Nella sezione successiva verrà data una presentazione preliminare dell'applicazione implementata, motivandone la scelta; in seguito sarà descritto in dettaglio il processo di sviluppo appena accennato; infine verranno riportate le due valutazioni, quantitativa e qualitativa, del risultato del lavoro svolto sulla toolchain.

5.1 Presentazione del caso di test

Per effettuare la valutazione delle nuove funzionalità di Protocode e MobileCodeGenerator è stata realizzata un'applicazione, denominata *MathKit*, che contiene un insieme di strumenti dedicati alla risoluzione di semplici quesiti matematici. L'idea di base nasce da una moltitudine di app molto simili, già esistenti sia per Android che per iOS: è stata appositamente scelta un'idea proveniente da un caso reale, per mostrare l'effettiva utilità degli strumenti prodotti in questo lavoro di tesi alla realizzazione di un'applicazione che assomiglia molto ad alcune già esistenti; la differenza sostanziale tra MathKit e le applicazioni simili presenti su App Store e Play Store infatti sta solamente nel numero di strumenti integrati nell'app.
Oltre a questo motivo preliminare, la scelta di realizzare un'applicazione del tipo descritto deriva dal fatto che essa permette di mostrare diversi esempi di ognuna delle nuove funzionalità implementate in Protocode e MobileCodeGenerator: per ciascuna schermata dell'applicazione relativa ad un singolo strumento di risoluzione di un problema matematico sono stati utilizzati parecchi controlli; questo ha reso necessario un largo uso di vincoli di posizione e dimensione ed è stato necessario inoltre sfruttare le possibilità offerte dalle catene di controlli per utilizzare

al meglio lo spazio disponibile. In aggiunta a ciò, la struttura di un'applicazione di questo tipo, fatta di tante schermate "semplici", permette di mostrare l'evidente utilità derivata dalla possibilità di creare schermate composte da diversi frammenti di interfaccia, connessi a parti distinte della logica applicativa, e dalla possibilità di riorganizzare questi frammenti in modo diverso a seconda del tipo di dispositivo sul quale l'applicazione verrà eseguita.

Prima di procedere con la descrizione della fase di sviluppo dell'applicazione campione, si riportano di seguito le diverse funzionalità di quest'ultima; nella sezione successiva, insieme agli altri dettagli implementativi, sarà specificato come e in quali schermate esse vengono messe a disposizione dell'utente:

- conversione di un numero decimale in una frazione (nella quale numeratore e denominatore sono primi tra loro);
- semplificazione di una frazione (nella quale numeratore e denominatore hanno fattori comuni);
- calcolo delle permutazioni (l'ordine conta) di k oggetti scelti da un insieme di N elementi, con e senza ripetizione;
- calcolo delle combinazioni (l'ordine non conta) di k oggetti scelti da un insieme di N elementi, con e senza ripetizione;
- calcolo del massimo comune divisore tra due numeri;
- calcolo del minimo comune multiplo tra due numeri;
- risoluzione di un'equazione lineare;
- risoluzione di un'equazione quadratica;
- risoluzione di un sistema di due equazioni lineari in due incognite;
- visualizzazione di articoli relativi agli argomenti che stanno alla base dei problemi che l'applicazione permette di risolvere.

5.2 Sviluppo

5.2.1 Prototipizzazione e generazione del codice

Definite le funzionalità che l'applicazione deve fornire all'utente è iniziata la fase dello sviluppo, che è iniziata con la formulazione del prototipo attraverso Protocode. La prima scelta di design fatta è stata quella relativa alla ripartizione delle diverse funzionalità nei frammenti di interfaccia che le rendono accessibili, vale a dire i view controller di Protocode; si riporta di seguito l'elenco dei view controller creati, con le relative funzionalità:

- view controller *Fractions*:
  - conversione numero decimale/frazione;
  - semplificazione frazione;
- view controller *Probability*:
  - calcolo delle permutazioni;
  - calcolo delle combinazioni;
- view controller *Commons*:
  - calcolo del M.C.D.;
  - calcolo del m.c.m.;
- view controller *Linear*:
  - risoluzione di un'equazione lineare;
- view controller *Quadratic*:
  - risoluzione di un'equazione quadratica;
- view controller *System*:
  - risoluzione di un sistema di due equazioni lineari;
- view controller *Websites*:
  - visualizzazione di articoli matematici su internet.

Ciascuno di questi view controller, ad eccezione dell'ultimo, è stato composto utilizzando oggetti `Label`, `EditText`, `ImageView` e `Button`, la maggior parte dei quali sono stati inclusi in catene verticali, per una disposizione ordinata e uno sfruttamento ottimale dello spazio. Il view controller `Websites` invece è stato composto da una singola `WebView` per mostrare contenuti web relativi ai problemi matematici risolti utilizzando i precedenti view controller.
La seconda scelta di design è stata quella relativa alla composizione dei diversi view controller nelle schermate effettive dell'applicazione, ovvero le scene di Protocode; si è optato per una suddivisione di tipo funzionale: è stata creata una scena *Algebra* contenente i primi tre view controller, relativi appunto a problemi di algebra semplice; successivamente è stata creata la scena *Equations*, contenente i successivi tre view controller, tutti relativi alla risoluzione di equazioni; infine è stata creata la scena *Guide*, contenente il view controller `Websites`, che funge da guida matematica per l'utente. Per ciascuna scena è poi stato anche scelto il tipo di visualizzazione dei view controller in essa contenuti; di seguito sono riportate le scelte fatte per ognuna:

- per la scena `Algebra`, si è voluto definire un'interfaccia a singolo view controller che visualizzasse come prima cosa un elenco delle funzionalità algebriche disponibili e successivamente, attraverso la selezione di un elemento della lista, visualizzasse il view controller che implementa quella funzionalità; questo ha richiesto la definizione di un ulteriore view controller chiamato *List* contenente una `ListView` che permette la scelta delle tre funzionalità algebriche.

- Per la scena `Equations` invece, si è scelta una visualizzazione a singolo view controller con TabBar per smartphone e una visualizzazione composita (`multiVC`) su tablet: in questo modo, su uno smartphone viene mostrata una TabBar che permette di scegliere tra equazione lineare, equazione quadratica e sistema, visualizzando il rispettivo view controller; su tablet invece i tre view controller sono visualizzati uno accanto all'altro, permettendo all'utente di accedere a tutte le funzionalità relative alla risoluzione di equazioni contemporaneamente.

- Per la scena `Guide`, infine, si è scelta una visualizzazione simile alla classica SplitView di iOS: su smartphone viene visualizzata una schermata che permette di scegliere un elemento, i cui dettagli sono mostrati in una seconda schermata, visualizzata a seguito della selezione dell'utente; su tablet invece, la schermata di selezione e quella di dettaglio sono affiancate, solitamente destinando un terzo o un quarto dello spazio disponibile alla prima e il resto dello spazio alla seconda. Lo stesso è stato fatto per la scena `Guide`, creando un ulteriore view controller, chiamato *Links*, che contiene tre pulsanti, uno per ciascuno dei principali argomenti matematici con cui l'applicazione ha a che fare (algebra semplice, probabilità e risoluzione di equazioni). Questo view controller è stato aggiunto come primo view controller della scena, la quale è stata differenziata tra smarthphone e tablet: sul primo sarà visualizzato un view controller alla volta, prima `Links` e poi `Websites` (il collegamento tra i due è gestito attraverso l'oggetto `Navigation` legato alla `ListView`); sul secondo i due view controller saranno visualizzati contemporaneamente, affiancati l'uno all'altro, destinando la maggior parte dello spazio al secondo.

In figura 5.1 è riportato uno schema che riassume tutti i view controller e le scene create e le relative scelte di visualizzazione.
La fase di prototipizzazione è stata conclusa con la definizione del menù (è stato creato un `MenuItem` per ciascuna delle scene) e con l'impostazione della scena `Algebra` come prima scena da mostrare all'avvio dell'applicazione (`launcher=true`); a questo punto è stato salvato il file di modello XMI e si è proceduto con la generazione del codice: è stato sufficiente importare il file di modello nel package `model` di MobileCodeGenerator, modificare il file di workflow `iOSAndAndroid-Generator.mwe` sostituendo il nome del vecchio model di esempio con quello del model `MathKit.xmi` e lanciare la generazione.

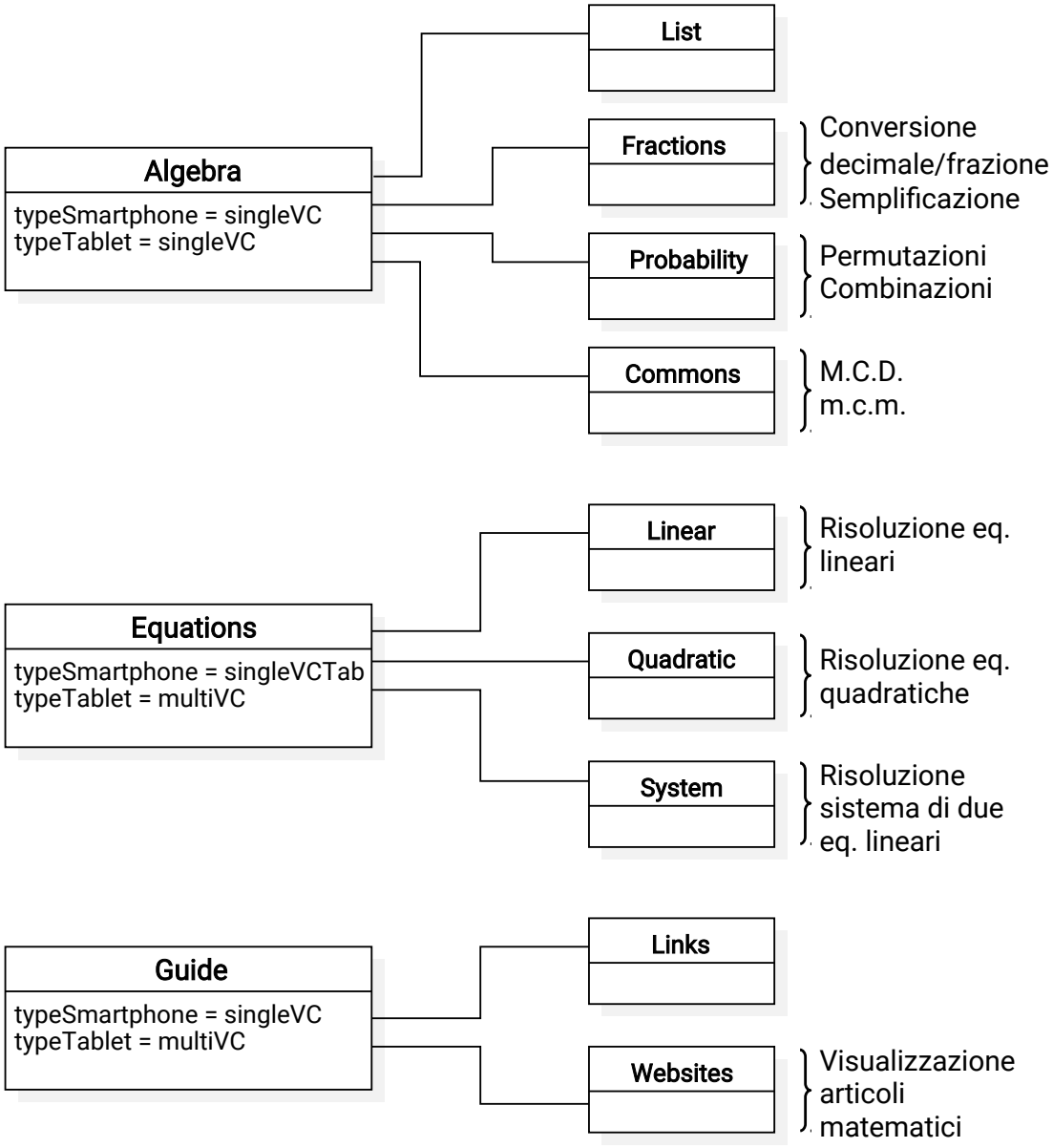

Figura 5.1: View controller e scene di MathKit

5.2.2 Sviluppo dell'applicazione finale

Una volta generati i progetti per Android e iOS, l'applicazione è stata completata manualmente, utilizzando gli IDE nativi AndroidStudio e XCode. Per quando riguarda la scena `Algebra`, sono stati innanzitutto implementati i metodi per il calcolo relativo alle frazioni, probabilità, M.C.D. e m.c.m.; è stato quindi modificato il colore del testo delle etichette che mostrano i risultati e specificato il tipo di input (che deve essere un numero) di ciascuno dei campi di inserimento dei valori. Successivamente, ogni elemento della lista del view controller `List` è stato collegato a ciascuno degli altri view controller e, per ognuno di questi, è stato scritto il codice che va a mostrare il pulsante "indietro" che permette di tornare a visualizzare la lista di selezione. Infine, in questa scena è stata introdotta

un'ulteriore personalizzazione relativa alla sola interfaccia per tablet: invece di visualizzare un solo view controller alla volta, sono stati modificati il file di layout per Android (creandone una versione specifica per tablet) e la classe Swift di iOS per visualizzare il view controller `List` sulla sinistra dello schermo (utilizzando circa un terzo della larghezza) e gli altri view controller, uno alla volta in base alla selezione dalla lista, sulla destra, in modo simile alla SplitView di cui si è parlato in precedenza. Si noti che questa personalizzazione in Protocode non è direttamente possibile: in esso infatti si possono definire scene che mostrano o solo un view controller alla volta (tipi `singleVC` e `singleVCTab`) oppure tutti i view controller insieme (tipo `multiVC`); in questa scena invece è stata implementata una "via di mezzo", dal momento che uno dei view controller è sempre visibile (`List`) mentre gli altri sono visualizzati uno alla volta. Tuttavia, anche se questo specifico caso non è modellabile direttamente in Protocode, la quantità di codice richiesto per effettuare questa modifica manualmente è molto esigua e oltretutto la maggior parte delle linee di codice da aggiungere sono sostanzialmente identiche a quelle generate automaticamente dalla toolchain per altre parti (questi aspetti saranno approfonditi nelle valutazioni quantitativa e qualitativa).

Per quanto riguarda la scena `Equations`, le prime modifiche effettuate sono state simili a quelle fatte per la scena precedente: implementazione delle funzioni per calcolare le soluzioni di equazioni lineari, quadratiche e del sistema, la modifica del colore del testo delle etichette relative ai risultati e la specifica dell'input, ancora una volta numerico, dei campi di inserimento. Per questa scena sono state poi anche inserite le immagini di esempio che raffigurano la struttura delle equazioni risolte; non sono state invece effettuate modifiche al layout.

L'ultima scena, `Guide`, ha richiesto pochissimi interventi, dal momento che il layout era già completamente definito e il codice relativo al collegamento tra i bottoni del view controller `Links` e il view controller `Websites` era già stato generato automaticamente a partire dagli oggetti Navigation di ciascun bottone, la cui destinazione era stata specificata con Protocode. E' stato quindi sufficiente scrivere le poche linee di codice che servono a modificare, in seguito alla pressione di uno dei pulsanti, l'URI della pagina web mostrata dalla WebView contenuta in `Websites`; è stata inoltre aggiunta una linea verticale per separare i due view controller, mostrati affiancati nell'interfaccia per tablet.
Le figure 5.2, 5.3 e 5.4 mostrano l'aspetto delle tre scene al termine dello sviluppo per Android, sia su smartphone che su tablet.

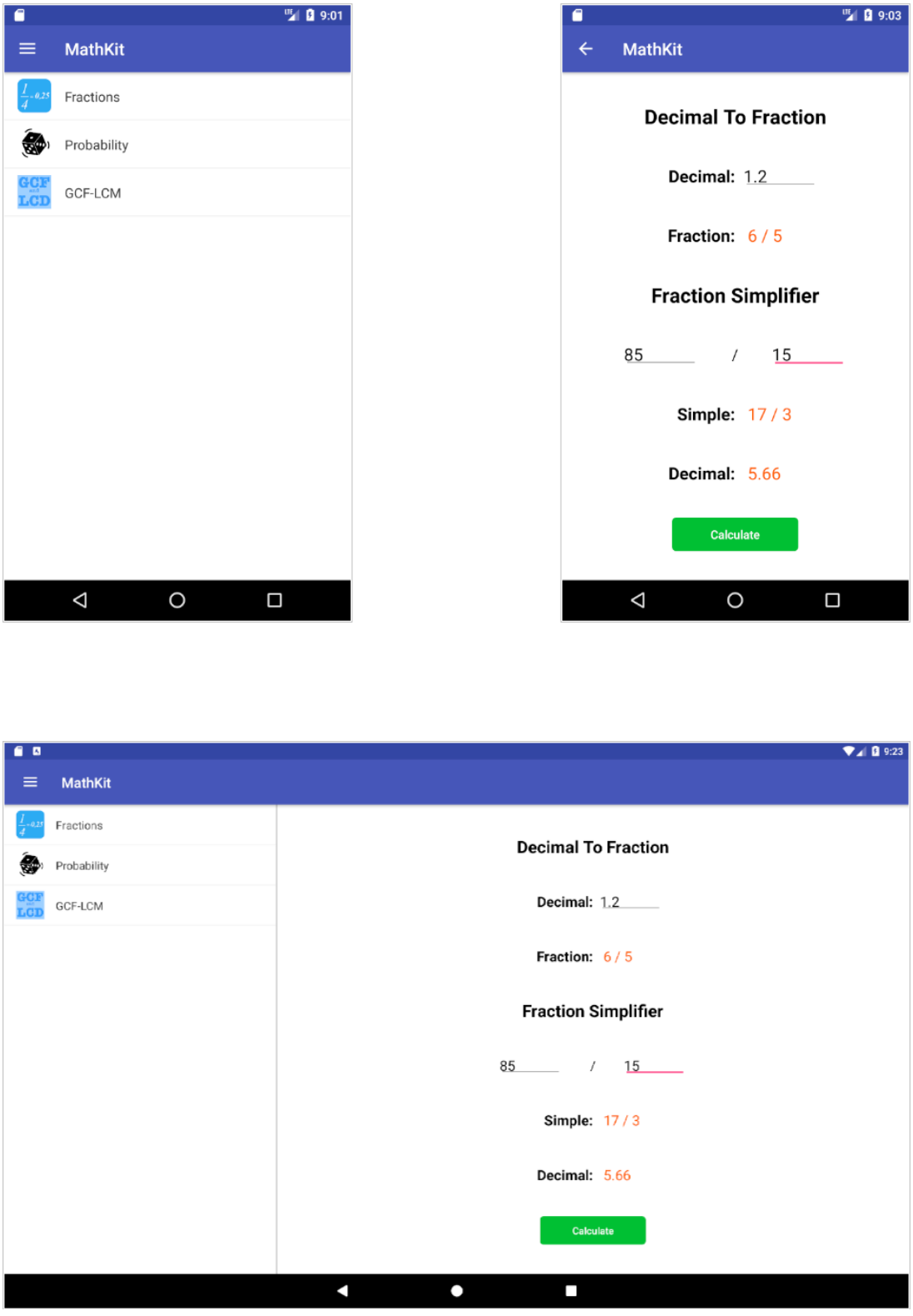

Figura 5.2: Activity Algebra su smartphone e tablet (Android)

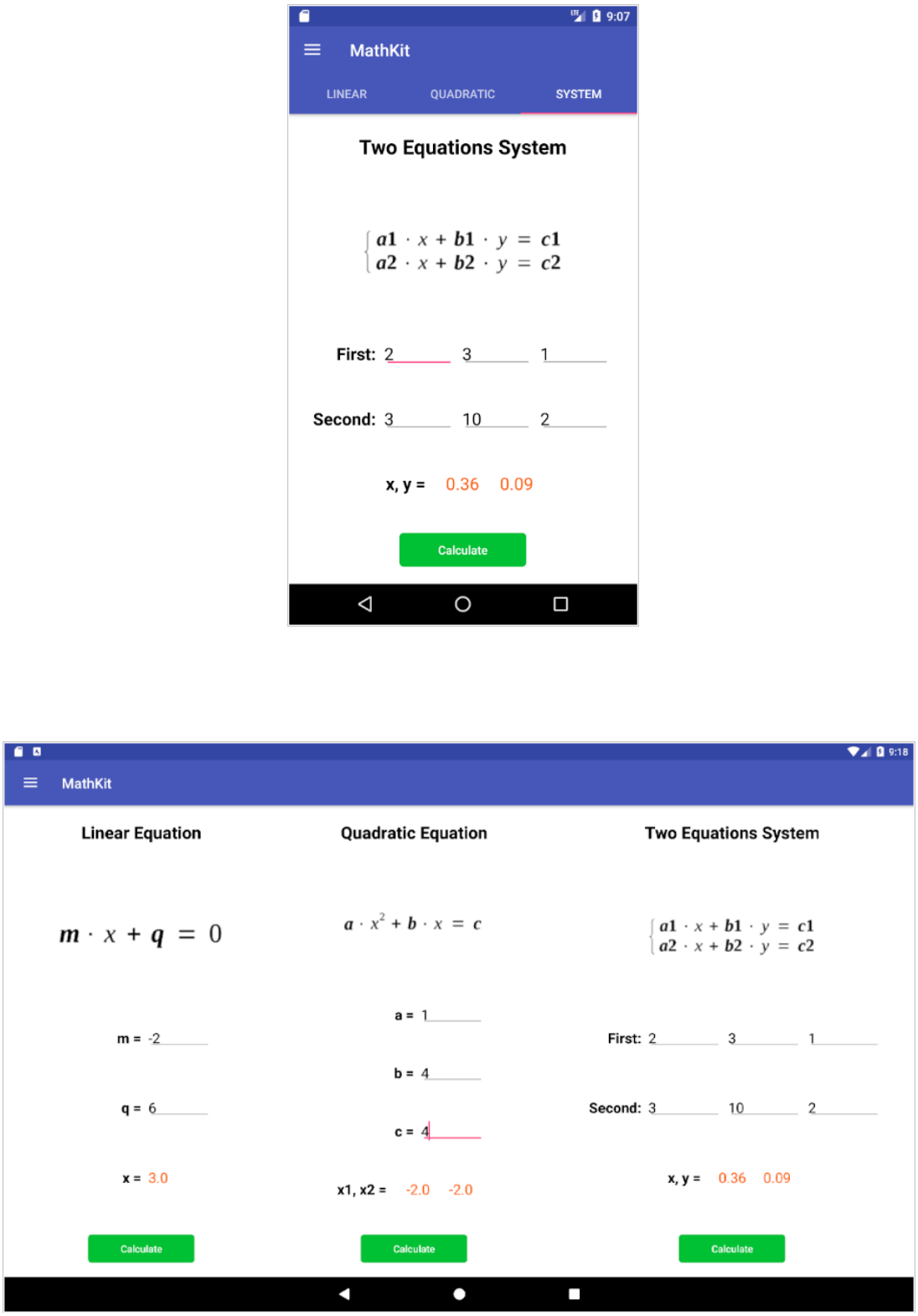

Figura 5.3: Activity Equations su smartphone e tablet (Android)

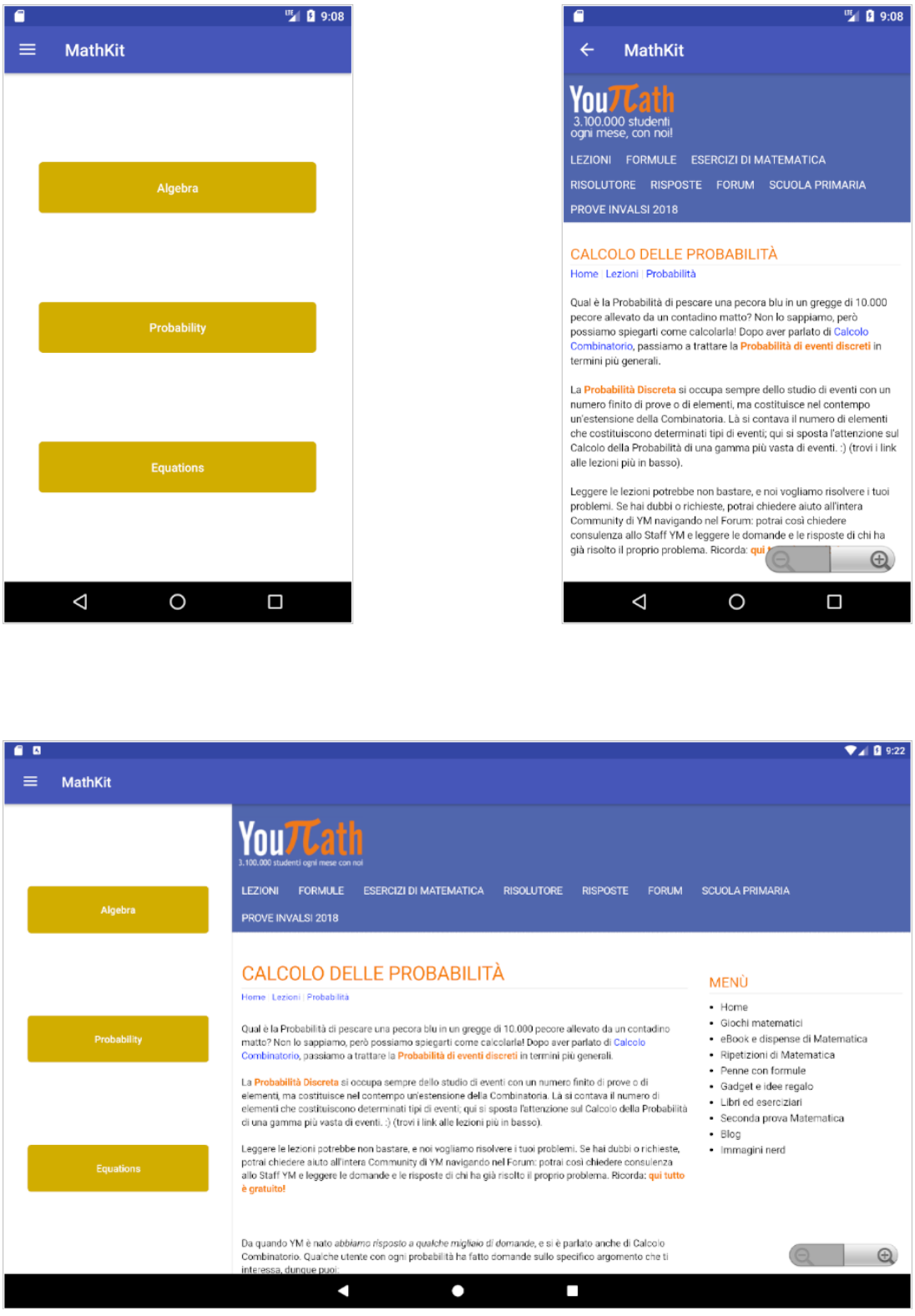

Figura 5.4: Activity Guide su smartphone e tablet (Android)

5.3 Analisi quantitativa

La tabella in figura 5.5 mostra un confronto tra la quantità di codice automaticamente generato dalla toolchain Protocode-MobileCodeGenerator e il numero totale di linee di codice che vanno a comporre l'applicazione completa per ciascuno dei due sistemi operativi; l'analisi è stata effettuata contando separatamente le SLOC (Source Lines Of Code) relative al codice Java e ai file XML per quanto riguarda Android e quelle relative al codice Swift e ai file storyboard per quanto riguarda invece iOS. Questa separazione permette di dare una prima valutazio-

Android			
	App. generata (SLOC)	App. completa (SLOC)	% SLOC generate
Java	1347	1662	81.0
XML	2135	2245	95.1
Totale	3482	3907	89.1
iOS			
	App. generata (SLOC)	App. completa (SLOC)	% SLOC generate
Swift	1170	1549	75.5
Storyboard	1741	1741	100.0
Totale	2911	3290	88.5

Figura 5.5: Rapporto quantità di codice sorgente generato

ne del lavoro svolto in questo lavoro di tesi, che si è concentrato sullo sviluppo dell'interfaccia grafica dell'applicazione: la percentuale di codice puramente dedicata ad essa infatti (file di layout XML di Android e file `.storyboard` di iOS) raggiunge il 100% per quanto riguarda iOS e supera il 95% per quanto riguarda invece Android. A questo primo risultato va specificato quanto segue:

- l'ottima percentuale di generazione di iOS nasconde il fatto che, se è vero che la maggior parte dell'interfaccia dell'applicazione è contenuta nei view controller la cui traduzione è svolta all'interno della storyboard, una parte invece (quella relativa alle scene) viene tradotta in Swift: il valore di 100% riportato in figura necessita pertanto di una rettifica, prendendo in considerazione anche il codice Swift. Specificato ciò, l'unica vera modifica riguardante l'interfaccia dell'applicazione è stata quella relativa alla scena `Algebra` la cui differenziazione su tablet, come già spiegato, è stata effettuata manualmente: questa modifica ha richiesto solo qualche decina di linee di codice facendo sì che, se si considera anche la parte di codice Swift relativo all'interfaccia, la percentuale di generazione automatica scende solo di qualche punto, rimanendo sopra al 97%.

- Anche la percentuale relativa ad Android necessita una considerazione: delle 110 linee di codice scritte manualmente, che costituiscono quel 4.9% che separa la percentuale di generazione automatica dal 100%, oltre la metà (68 SLOC) riguardano la differenziazione fatta per la scena `Algebra`; di queste, quelle veramente scritte a mano sono solo 5, dal momento che le altre sono esattamente uguali a quelle che servono per differenziare i layout delle altre scene e quindi è stato possibile copiarle, modificandole solo in minima parte. Alla luce di questo, la percentuale di codice di layout XML generato da Protocode e MobileCodeGenerator sfiora il 98%.

Una seconda valutazione quantitativa può essere fatta guardando il codice Java/Swift generato: dal momento che esso contiene tra le altre cose la logica applicativa dell'applicazione, che non può essere generata automaticamente, le relative percentuali scendono, attestandosi su un 81% per Java e un 75.5% per Swift; si noti in particolare che la percentuale di Swift è più bassa di quella di Java a causa del fatto che alcune componenti grafiche che per Android sono definite nei file di layout XML, per iOS sono invece implementate nelle classi Swift. Anche per questa valutazione occorre fare un'ulteriore considerazione, che riguarda il tipo di applicazione scelta come caso di test: un'app come MathKit non richiede una logica applicativa particolarmente complessa; sono bastate infatti poche centinaia di linee di Java/Swift per implementarne le funzionalità. Rimane però altrettanto vero che una discreta quantità di codice Java/Swift è richiesta per gestire il setup e il comportamento dei componenti grafici dell'interfaccia e per implementare la navigazione dal menù e dai controlli che la permettono: in questo, la toolchain prodotta in questo lavoro di tesi si è dimostrata molto efficace, generando praticamente tutto il codice necessario a queste funzionalità. Oltretutto, se si osservano le percentuali totali di codice automaticamente generato, si può notare che per entrambi i sistemi esse sono molto vicine al 90%: questo risultato può considerarsi ottimo, dato che in questo caso allo sviluppatore rimane praticamente un solo decimo dell'applicazione da sviluppare manualmente.

5.4 Analisi qualitativa

Oltre all'analisi quantitativa del codice generato automaticamente attraverso gli strumenti sviluppati in questa tesi, è possibile fare alcune considerazioni riguardanti la qualità. Per far questo, si può innanzitutto fare un confronto tra le interfacce che è possibile progettare con la nuova toolchain e quelle che era possibile creare con le vecchie versioni di Protocode e MobileCodeGenerator: si consideri un view controller di MathKit, per esempio `Fractions`, che è composto da una catena verticale di tipo `spread` contenente diversi controlli, centrati con vincoli di posizione e proporzionati con vincoli di dimensione basati su quelle della finestra, mostrato in fig. 5.2 su Nexus 5X (alto); sarebbe stato possibile ottenere la stessa interfaccia attraverso la vecchia versione della toolchain, utilizzando i vecchi vincoli e dimensioni assolute. Tuttavia, l'applicazione generata dalla toolchain 3.0 avrebbe avuto un'interfaccia ottimale solo su dispositivi con display delle esatte dimensioni di quello tenuto a riferimento durante la prototipizzazione: l'esecuzione su qualsiasi altro device con uno schermo diverso avrebbe, per prima cosa,

causato il decentramento di tutta l'interfaccia; oltre a questo, su uno schermo di dimensioni inferiori ci sarebbe stato il rischio che parte dei controlli finisse fuori dalla finestra; su un display di dimensioni superiori, l'interfaccia avrebbe sfruttato solo una parte dello spazio, lasciando il resto della schermata vuoto. Con la nuova toolchain invece, sfruttando i nuovi vincoli e il meccanismo delle Control Chain, è possibile ottenere layout ottimali su ogni tipo di dispositivo e oltretutto è possibile progettare schermate composte da diversi frammenti di interfaccia, cosa prima non possibile, e differenziare la visualizzazione di questi, adattando al meglio l'interfaccia a dispositivi con display di grandi dimensioni. Un ultima considerazione qualitativa riguarda il menù: anche se in iOS non esiste un componente preconfigurato che implementi un menù a tenda laterale, a differenza di Android nel quale ormai da molto tempo è proposto il Navigation Drawer, in tantissime applicazioni iOS è comunque possibile trovare un menù simile, sviluppato ad-hoc dai programmatori di ognuna di esse; il motivo deriva probabilmente dal fatto che un menù come questo permette di lasciare più spazio ai controlli di ciascuna schermata, cosa non vera per i menù a TabBar che occupano persistentemente una parte di essa. Il fatto che attraverso Protocode e MobileCodeGenerator sia generato automaticamente un menù del genere permette di seguire le più attuali *best practices* e, in particolar modo per iOS, risparmiare parecchio tempo nella sua implementazione.

Capitolo 6

Conclusioni

Il lavoro svolto in questa tesi ha portato al miglioramento della toolchain formata da Protocode e MobileCodeGenerator i quali, pur costituendo un ottimo strumento già all'inizio di questo lavoro, sono stati perfezionati e resi ancor più potenti e all'avanguardia; il risultato ottenuto dimostra una volta in più quanto l'approccio *model-driven*, che è alla base di questo progetto, risulti valido nella realizzazione di applicazioni multi-piattaforma, consentendo un notevole risparmio di tempo e allo stesso tempo supportandone lo sviluppo nativo, esente dai compromessi presenti in quello ibrido o web.

I nuovi strumenti permettono di produrre interfacce grafiche al passo coi tempi, con un aspetto fedele al design di ciascun sistema oppure personalizzato secondo le preferenze dello sviluppatore, ma sopratutto in grado di adattarsi alla continua evoluzione e differenziazione dei dispositivi mobili, che presentano display dalle caratteristiche sempre più variegate. Il processo di sviluppo di applicazioni che inizia con l'utilizzo di questi strumenti consente una rapida impostazione di tutti gli elementi grafici che compongono l'interfaccia utente, con meccanismi avanzati di disposizione, dimensionamento e ricombinazione; la possibilità di definire elementi riutilizzabili permette inoltre un notevole risparmio di tempo, dal momento che spesso si ha a che fare con porzioni di interfaccia comuni, che in assenza di una modalità di riciclo dovrebbero essere ridefinite ogni volta dal principio. Tutte queste caratteristiche, unite alla possibilità di definire la componente *Model* dell'applicazione, sviluppata nel precedente lavoro di tesi, forniscono al programmatore un enorme supporto nello sviluppo, consentendogli di concentrarsi sulla parte più specifica dell'applicazione, vale a dire la logica applicativa.

Nonostante non si possa dimostrare l'assoluta validità degli strumenti sviluppati attraverso la singola valutazione che è stata precedentemente svolta, dal momento che sarebbe necessario un test su svariate applicazioni di tipi diversi, risulta comunque evidente che almeno una parte di qualsiasi ipotetica applicazione si voglia creare è formulabile e generabile attraverso Protocode e MobileCodeGenerator, che permettono quindi un risparmio di tempo, anche se variabile, nella fase iniziale dello sviluppo in qualsiasi contesto riguardante i sistemi Android e iOS. Oltretutto, questi strumenti possono risultare molto utili, oltre a programmatori esperti, anche ad utenti alle prime armi o con competenze più incentrate

sul design dal momento che l'interfaccia di Protocode risulta molto semplice ed intuitiva e la generazione del codice, una volta generato il file di modello XMI, avviene con poche e semplici operazioni.

6.1 Sviluppi futuri

Sebbene con questo lavoro gli strumenti sviluppati abbiano raggiunto un buon livello di maturità, derivato dal fatto che essi sono il frutto di ormai cinque lavori di tesi, si possono delineare alcuni possibili sviluppi futuri, che potrebbero rendere la toolchain ancora più completa e competitiva. Il primo aspetto è di natura tecnica e riguarda specificatamente Protocode: esso è stato sviluppato nel 2013/14 utilizzando la versione 1.6.1 di Ember.js e la versione 1.0.0 beta di Ember Data, che erano allora le ultime versioni disponibili; dopo il primo sviluppo, per garantire il corretto funzionamento del software, sono state mantenute quelle versioni dei framework, nonostante i continui aggiornamenti che hanno portato alle versioni correnti, a marzo 2018, 3.0.0 per Ember e 3.0.1 per Ember Data. Sarebbe necessario un lavoro di ristrutturazione completa di Protocode a partire da queste nuove versioni delle librerie, che permetterebbero un prodotto sicuramente più potente e performante e ne assicurerebbero inoltre la stabilità. A livello invece di funzionalità, sarebbe utile un aggiornamento della parte di toolchain dedicata allo sviluppo della controparte smartwatch dell'applicazione: essa è stata sviluppata nel 2016, quando i modelli di dispositivi di questo tipo si erano appena diffusi e i rispettivi sistemi operativi erano nati da poco (la prima versione di Android Wear è stata rilasciata in *Developer Preview* il 18 marzo 2014 mentre la prima versione di watchOS è stata pubblicata il 24 aprile 2015); le possibilità offerte da Protocode e MobileCodeGenerator per questi sistemi sono sicuramente esigue e ormai datate. Un ultimo spunto per il miglioramento della toolchain riguarda ancora il design delle interfacce grafiche, essendo veramente vasto l'insieme di possibilità offerte ai programmatori da entrambi i sistemi:

- per quanto riguarda i vincoli di posizione, la toolchain 4.0 mette a disposizione il più grande sottoinsieme di funzionalità implementabili sia in Android che in iOS, a meno di:
  - bias nei constraint di centro che, similmente al parametro `bias` definibile per le catene di tipo `packed`, influenzerebbe il *centering* del controllo, spostandolo ad esempio in corrispondenza di un quarto della larghezza della finestra (bias = 0.25 su centering orizzontale) invece che nell'esatta metà;
  - controlli inseribili in due catene diverse, una per ciascun asse (non è stato implementato in Protocode 4.0 perché avrebbe richiesto una grande quantità di tempo).
- Relativamente alla combinazione di view controller nelle scene, si potrebbe migliorare il meccanismo di composizione dichiarando le scene come una sorta di view controller estesi capaci di contenere, oltre ai normali controlli (bottoni, etichette ecc.) normalmente definibili per i view controlli, anche:

  - nuovi componenti adibiti al contenimento di altri view controller (si potrebbe aggiungere ai Container già definiti in Protocode 4.0 un nuovo tipo di contenitori, simili al ViewPager di Android, con la capacità di contenere diversi view controller);
  - controlli come TabBar per permettere di coordinare quale view controller mostrare nei nuovi contenitori simili ai ViewPager oppure semplicemente implementare altre azioni di navigazione.

- Infine, dal momento che molte applicazioni presenti nel Play Store contengono controlli nella Action Bar, sarebbe utile poterne aggiungere alcuni anche in fase di prototipizzazione; questo richiederebbe tuttavia di trovare un'alternativa per quanto riguarda iOS, dato che non è altrettanto comune trovare controlli nelle Navigation Bar delle applicazioni per dispositivi Apple.

Bibliografia

[1] *Mobile apps overtake PC Internet usage in U.S.* - James O'Toole, http://money.cnn.com/2014/02/28/technology/mobile/mobile-apps-internet/index.html.

[2] *Worldwide mobile app revenues in 2015, 2016 and 2020 (in billion U.S. dollars)* - Statista.com, https://www.statista.com/statistics/269025/worldwide-mobile-app-revenue-forecast/.

[3] **Apple Inc.** - Cupertino, California, USA, https://www.apple.com/.

[4] **Samsung Group** - Seoul, Corea del Sud, http://www.samsung.com/.

[5] **Huawei** - Shenzhen, Cina, https://consumer.huawei.com/.

[6] **Xiaomi Inc.** - Distretto di Haidian, Pechino, Cina, http://www.mi.com/.

[7] **Oppo Digital** - Dongguan, Cina, https://www.oppo.com/.

[8] **Vivo** - Dongguan, Cina, http://www.vivo.com/.

[9] *Smartphone Vendor Market Share, 2017 Q1* - IDC Quarterly Mobile Phone Tracker, https://www.idc.com/promo/smartphone-market-share/vendor.

[10] **Android** - Google LLC, https://www.android.com/.

[11] **Google LLC** - Mountain View, California, USA, https://www.google.com/.

[12] **iOS** - Apple Inc., https://www.apple.com/ios.

[13] *Prototipizzazione rapida per applicazioni mobili multipiattaforma* - Mattia Natali, Tesi di Laurea Magistrale, Politecnico Di Milano, 2014.

[14] *Un approccio model-driven per lo sviluppo di applicazioni mobili native* - Gregorio Perego, Stefania Pezzetti, Tesi di Laurea Magistrale, Politecnico Di Milano, 2013.

[15] *Generazione automatica di applicazioni mobili native secondo un approccio model-driven* - Aldo Pintus, Tesi di Laurea Magistrale, Politecnico Di Milano, 2016.

[16] *Prototipizzazione e generazione automatica della componente Model per applicazioni mobili multipiattaforma* - Alessio Rossotti, Tesi di Laurea Magistrale, Politecnico Di Milano, 2017.

[17] **Eclipse** - Eclipse Foundation, `https://www.eclipse.org/`.

[18] *A quick reference for iOS devices.* - iOSRes.com, `http://iosres.com/`.

[19] *Release Updates - ConstraintLayout 1.0 is now available* - AndroidStudio.googleblog.com, `https://androidstudio.googleblog.com/2017/02/constraintlayout-10-is-now-available.html`.

[20] *Release Updates - ConstraintLayout 1.1.0 beta 6* - AndroidStudio.googleblog.com, `https://androidstudio.googleblog.com/2018/03/constraintlayout-110-beta-6.html`.

[21] **Balsamiq Mockups** - Balsamiq Studios LLC, `https://balsamiq.com/`.

[22] **Marvel** - *Marvel Team*, `https://marvelapp.com/`.

[23] **InVision** - InVision, `https://www.invisionapp.com/`.

[24] **Proto.io** - Proto.io Team, `https://proto.io/`.

[25] **Xamarin** - Xamarin Inc., `https://www.xamarin.com/`.

[26] *Announcing Xamarin* - Miguel de Icaza, `http://tirania.org/blog/archive/2011/May-16.html`.

[27] **Titanium** - Axway Software, `https://www.appcelerator.com/`.

[28] **PhoneGap** - Adobe Systems Inc., `https://phonegap.com/`.

[29] **Ionic Framework** - Ionic,
`https://ionicframework.com/`.

[30] **Android Studio** - Google LLC,
`https://developer.android.com/studio/index.html`.

[31] **Kotlin** - JetBrains s.r.o.,
`https://kotlinlang.org/`.

[32] **XCode** - Apple Inc.,
`https://developer.apple.com/xcode/`.

[33] *Domain-specific modeling and code generation for cross-platform multidevice mobile apps* - Eric Umuhoza, CoRR, abs/1509.03109, 2015.

[34] *Automatic code generation for cross-platform, multi-device mobile apps: Some reflections from an industrial experience* - Eric Umuhoza, Hamza Eddouibi, Marco Brambilla, Jordi Cabot, Aldo Bongio,
Proceedings of the 3rd International Workshop on Mobile Development Lifecycle, MobileDeLi 2015, pages 37–44, New York, NY, USA, 2015. ACM.

[35] *Model driven development approaches for mobile applications: A survey* - Eric Umuhoza, Marco Brambilla,
Mobile Web and Intelligent Information Systems - 13th International Conference, MobiWIS 2016, Vienna, Austria, August 22-24, 2016, Proceedings, pages 93–107, 2016.

[36] **Native Studio** - Neonto Ltd.,
`https://neonto.com/nativestudio`.

[37] **Ember.js** - Ember Team,
`https://emberjs.com/`.

[38] **Yeoman.io** - Yeoman Team,
`http://yeoman.io/`.

[39] **Grunt** - Grunt Team,
`https://gruntjs.com/`.

[40] **Bower** - Bower Team,
`https://bower.io/`.

[41] **Compass** - Chris Eppstein,
`http://compass-style.org/`.

[42] **Sass** - Sass Team,
`http://sass-lang.com/`.

[43] **Bootstrap** - Bootstrap Core Team `https://getbootstrap.com/`.

[44] **Ember Data Local Storage Adapter** - Ricardo Mendes,
`https://github.com/locks/ember-localstorage-adapter`.

[45] **JQuery** - JS Foundation,
https://github.com/locks/ember-localstorage-adapter.

[46] **Font Awesome** - Fonticons Inc,
https://fontawesome.com/.

[47] **Blob.js** - Eli Grey,
https://github.com/eligrey/Blob.js.

[48] **FileSaver.js** - Eli Grey,
https://github.com/eligrey/FileSaver.js.

[49] **vkBeautify** - Vadim Kiryukhin,
https://github.com/vkiryukhin/vkBeautify.

[50] **Node.js** - Node.js Foundation,
https://nodejs.org/.

[51] **ConstraintLayout** - Android Support Library,
https://developer.android.com/reference/android/support/constraint/ConstraintLayout.html.

[52] **Epsilon** - Eclipse Foundation,
https://www.eclipse.org/epsilon/.

[53] **Eclipse Modeling Framework (EMF)** - Eclipse Foundation,
http://www.eclipse.org/modeling/emf/.

[54] *openArchitectureWare has moved to the Eclipse Modeling Project* - openArchitectureWare.org,
https://web.archive.org/web/20140105071651/http://openarchitectureware.org/.

[55] **Xpand** - Eclipse Foundation,
http://wiki.eclipse.org/Xpand.

[56] **Modeling Workflow Engine (MWE)** - Eclipse Foundation,
http://wiki.eclipse.org/Modeling_Workflow_Engine_(MWE).

Acronimi

ADT, Android Development Tools, 21

CLI, Command Line Interface, 20
CSS, Cascading Style Sheets, 28

EMF, Eclipse Modeling Framework, 56

GUI, Graphical User Interface, 13

HTML, HyperText Markup Language, 12

IDE, Integrated Development Environment, 13

JS, JavaScript, 12
JSON, JavaScript Object Notation, 29

MVVM, Model View View-Model, 27

oAW, openArchitectureWare, 56

PC, Personal Computer, 11

SLOC, Source Lines Of Code, 89
SO, Sistema Operativo, 12
SQL, Structured Query Language, 29

URI, Uniform Resource Identifier, 27

VC, View Controller, 24

WYSIWYG, What You See Is What You Get, 26

XMI, XML Metadata Interchange, 26
XML, eXtensible Markup Language, 29