POLITECNICO DI MILANO

Scuola di Ingegneria Industriale e dell'Informazione
Dipartimento di Elettronica, Informazione e Bioingegneria

Corso di Laurea Magistrale in Ingegneria Informatica

Un Approccio Model-Driven per lo Sviluppo di Applicazioni Mobili Native

Relatore: Prof. Luciano BARESI

Tesi di Laurea di:

Gregorio PEREGO Matr. 778932

Stefania PEZZETTI Matr. 782867

Anno Accademico 2012-2013

*Alla mia **famiglia** e a **Irene**,*
per il sostegno e la vicinanza mostrati.

Non posso che cominciare ringraziando i miei genitori. Non sono mai riuscito a comunicarvi con le giuste parole il mio affetto nei vostri confronti, ma spero di esser riuscito a mostrarvi la mia notevole riconoscenza per tutti i sacrifici che avete fatto per me in questi anni. Senza di voi non sarei certamente la persona che sono: grazie per tutto quello che avete fatto e continuerete a fare per me!
Con voi ricordo anche tutta la famiglia, il mio fratellone Tommaso per primo, che ha sempre dimostrato di sapermi guidare nella giusta direzione e che per me continua ad essere un punto di riferimento, poi i miei nonni, i miei zii e i miei due cugini, che mi hanno sempre sostenuto e valorizzato.

Un ringraziamento speciale va alla donna che amo, Irene, con cui ho vissuto cinque anni indimenticabili della mia vita e senza la quale non avrei mai avuto la determinazione di proseguire lungo questo arduo percorso.

E poi c'è la mia seconda famiglia, i miei Amici. In particolare Nino e Yuzzo, che mi sono sempre vicini nei momenti di difficoltà e con cui mi trovo davvero bene. Ringrazio inoltre Bonfa, che nonostante i continui tour in giro per il mondo è un amico su cui è sempre possibile contare, e Zanna e Gloria, con cui ho legato molto ultimamente.

Ringrazio tutti i miei compagni dell'università, in particolare Andre, Visco e Vetto, con cui ho affrontato la fatica degli esami e trascorso momenti in compagnia indimenticabili. Non posso non ringraziare naturalmente la mia eccezionale collega di tesi, Stefania, che mi ha sopportato per tutto questo tempo e con cui ho condiviso le difficoltà ma soprattutto i momenti di gioia e successo che hanno caratterizzato gli ultimi due anni di università.

Ringrazio infine il professor Baresi, che è stato il primo a trasmettermi la passione nello sviluppo di applicazioni per dispositivi mobili e che ha dato un contributo significativo nello svolgimento della tesi.

Gregorio

*A te nonna **Anita**,*
il tuo sorriso è il mio ricordo più caro.

*Alla mia **famiglia**,*
per l'immensa bellezza della sua semplicità.

Un grazie davvero di cuore ai miei genitori per avermi dato la possibilità di frequentare un corso di studi universitario.

Un ringraziamento speciale ad Augusto per non avermi mai negato il suo aiuto, per avermi sempre consigliata, supportata e sopportata in qualsiasi momento.

Grazie a mia nonna Rosa per avermi sempre appoggiato e a mio fratello Marco, che ha rallegrato questi anni, anche nei giorni di studio profondo.

Ancora un ringraziamento per i miei amici Maria, Guido e Fabrizio, con cui ho trascorso due anni stupendi, e grazie a tutti coloro che ho incontrato durante il periodo universitario, con i quali è sorta un'amicizia vera e con i quali ho condiviso momenti di gioia, ma anche di impegno e di studio.

Infine un doveroso e sentito ringraziamento al professor Luciano Baresi che ci ha seguito nello svolgimento della tesi lasciandoci liberi nelle nostre scelte, pur indicandoci il percorso da intraprendere, al professor Sam Guinea per il sostegno nella comprensione del mondo iOS e alla signora Alessandra Viale per la sua gentilezza e la sua infinita disponibilità.

Stefania

Indice

Elenco delle figure

Elenco delle tabelle

Sommario

La sempre più ampia diffusione di smartphone e tablet apre grandi opportunità nell'ambito dello sviluppo di applicazioni per questi dispositivi. Questo lavoro di tesi presenta un approccio model-driven allo sviluppo di applicazioni mobili native, alternativo a quello cross-platform, che, per quanto interessante e diffuso, presenta delle limitazioni non trascurabili.
Con l'obiettivo di consentire agli sviluppatori di sfruttare un approccio model-driven per l'implementazione di applicazioni mobili abbiamo seguito un percorso composto da varie fasi. Innanzitutto, basandoci sul linguaggio di modellazione UML, abbiamo definito un metamodello astratto, in grado di concettualizzare ad alto livello gli elementi che strutturano le applicazioni mobili, indipendentemente dalla piattaforma per cui sono sviluppate. Successivamente abbiamo prodotto la versione implementativa del primo metamodello, le cui istanze permettono di delineare in maniera dettagliata i modelli delle applicazioni che si intende realizzare. Infine abbiamo definito le regole di traduzione in codice degli elementi che compongono il modello dell'applicazione, creando uno strumento software per la generazione dei sorgenti nativi, utilizzando diversi strumenti e un insieme di tecnologie esistenti. Il generatore di codice permette di generare i sorgenti Android e iOS corrispondenti al modello dell'applicazione definita dallo sviluppatore. La scelta di limitarsi a queste due piattaforme è legata principalmente al fatto che esse costituiscono i due sistemi operativi per dispositivi mobili attualmente più diffusi al mondo.
Oltre alle varie fasi che hanno condotto alla definizione dei due metamodelli e all'implementazione del generatore di codice, abbiamo svolto una valutazione qualitativa e quantitativa dell'utilizzo di quest'ultimo, da cui è emersa la possibilità concreta di usarlo come strumento di supporto alle fasi di design e di implementazione, in quanto efficace nel ridurre lo sforzo richiesto agli sviluppatori di applicazioni mobili.

Parole chiave: applicazioni mobili native, metamodello UML, sviluppo model-driven, generazione di codice, Android, iOS.

Abstract

The increasingly widespread production of smartphones and tablets opens up great opportunities in the development of applications for these devices. The aim of this thesis work is to provide a model-driven approach to the development of native mobile applications as alternative to the cross-platform one, which, despite its popularity, has considerable limitations.
The present work has been divided in four steps. First of all, relying on the UML modeling language, we defined a new abstract metamodel able to conceptualize, at a high-level, elements that structure mobile applications, regardless of the platform for which they are developed. Later we defined the implementative version of the first metamodel, whose instances allow to define in detail the models of applications intended to be accomplished. Finally we defined some rules in order to translate the elements that compose the application model into code, creating a software tool for the generation of the native sources, using different tools and a set of existing technologies. The mobile code generator allows to produce the Android and iOS source code corresponding to the application model defined by the developer. The choice of focusing the attention only to these two platforms is mainly related to the fact that they currently constitute the most widespread operating systems for mobile devices worldwide.
In addition to the various steps, which led to the definition of the two metamodels and the implementation of the code generator, we carried out a qualitative and quantitative evaluation of the code generator use. This step highlighted the possibility of adopting the code generator as a supporting tool to the design and the implementation phases, showing a significant reduction of the effort required to mobile application developers.

Keywords: native mobile applications, UML metamodel, model-driven development, code generation, Android, iOS.

Capitolo 1

Introduzione

Negli ultimi anni il mercato degli smartphone ha subito una rapida evoluzione che ha condotto a porre attenzione sia all'hardware che al software utilizzato da questi dispositivi, al fine di migliorare le prestazioni e far apprezzare sempre più il prodotto ai clienti di questo vasto mercato. Android, iOS e Windows Phone rappresentano le principali piattaforme smartphone esistenti nel mondo del mobile, ciascuna caratterizzata dalla propria architettura, sulla base della quale i programmatori si adattano nello sviluppo delle loro applicazioni. All'interno delle grandi software-house aumentano i team di sviluppo di applicazioni mobili native, ciascuno dei quali specializzato nella produzione di applicazioni mobili per una particolare piattaforma, con strumenti e linguaggi di programmazione specifici. Quindi, nel caso in cui si ritenesse necessaria la progettazione di una stessa applicazione per sistemi operativi differenti, in aggiunta al necessario coinvolgimento di più team di sviluppo, si corre il rischio che i diversi team producano applicazioni che discostano molto l'una dall'altra, quando invece è richiesta una corrispondenza a livello di funzionalità e di interfaccia grafica.

Per le applicazioni web e per i software sviluppati in un linguaggio di programmazione ad oggetti, lungo gli anni, sono emersi diversi strumenti, metodologie di sviluppo e linguaggi di modellazione che si sono ormai consolidati come standard. Tra i più noti si trova l'Unified Modeling Language, un linguaggio di modellazione e progettazione basato sul paradigma object-oriented, dal quale derivano altri linguaggi usati nella modellazione del software, come SysML, un linguaggio general-purpose di modellazione visuale per la rappresentazione di sistemi complessi, e WebML, il linguaggio di specifica per le applicazioni web data-intensive. Per quanto riguarda le applicazioni mobili, invece, mancano strumenti specifici di supporto alla fase di progettazione e design. Tuttavia sono nate diverse piattaforme di sviluppo cross-platform, ciascuna con i suoi strumenti e linguaggi di programmazione,

con l'obiettivo di sviluppare l'applicazione una sola volta ed eseguirla su più piattaforme. Un approccio di questo genere può portare numerosi benefici, ma non mancano le limitazioni, per esempio quelle che che riguardano le basse performance di esecuzione, l'impossibilità di sfruttare completamente l'hardware del dispositivo e il discostamento dell'interfaccia grafica da quella tipica delle applicazioni native.

Il presente lavoro di tesi si inserisce quindi in questo nuovo contesto, e mira a presentare una soluzione di modellazione che possa supportare la fase di progettazione di applicazioni mobili, indipendentemente dalla specifica piattaforma per cui vengono sviluppate. Verrà presentato in particolare uno strumento software da noi realizzato che introduce un approccio model-driven per lo sviluppo di applicazioni mobili native, alternativo a quello cross-platform e di supporto a quello classico finalizzato allo sviluppo parallelo della stessa applicazione per le diverse piattaforme mobili. Esso consente infatti la definizione di un modello dell'applicazione desiderata e, a partire da questo, permette di generare automaticamente i sorgenti Android e iOS corrispondenti.
Per supportare la fase di modellazione, siamo partiti dalla definizione di un metamodello astratto, basato sull'estensione di UML e le cui istanze costituiscono modelli ad alto livello di applicazioni mobili, per poi creare una versione più concreta, chiamata metamodello implementativo, finalizzata alla definizione di un modello dettagliato dell'applicazione. Relativamente alla generazione di codice, invece, ci siamo concentrati sulla produzione dei sorgenti per le piattaforme Android e iOS, in quanto rappresentano le piattaforme che attualmente dominano il mercato dei dispositivi mobili. Comunque, aggiungendo nuove regole di traduzione, sarebbe stato possibile offrire la possibilità di generare i sorgenti anche per le altre piattaforme.
La scelta di procedere in questo modo è stata mantenuta con l'obiettivo di definire uno strumento in grado di supportare i diversi team di sviluppo nella fase di progettazione dell'applicazione. Quest'ultimi, sfruttando il metamodello astratto, hanno infatti la possibilità di appoggiarsi ad uno strumento di facile comprensione, per collaborare e interagire con chiunque intervenga nello sviluppo dell'applicazione, compreso il cliente. Definendo un'istanza della versione implementativa del metamodello gli sviluppatori possono invece ottenere i sorgenti nativi corrispondenti per le diverse piattaforme, a partire dai quali possono completare lo sviluppo dell'applicazione implementando le parti mancanti.

Il lavoro svolto può essere suddiviso in quattro fasi: una prima fase, molto generale, riguardante la definizione del metamodello astratto; una seconda

fase in cui si è reso necessario manipolare il primo metamodello per costruirne uno nuovo e più concreto, ovvero il metamodello implementativo, finalizzato alla definizione di una sua istanza che costituisse il modello vero e proprio dell'applicazione desiderata; una terza fase, molto pratica, volta alla produzione del codice necessario alla traduzione del modello dell'applicazione nei corrispettivi sorgenti nativi; infine una quarta fase in cui viene valutato il contributo del lavoro nell'ambito dello sviluppo delle applicazioni mobili.

Di seguito è riportata una breve descrizione del contenuto di ciascun capitolo della tesi:

- Il *secondo capitolo* presenta gli approcci esistenti allo sviluppo di applicazioni mobili, in particolare quelli che appartengono alla categoria cross-platform, riportando per ciascuno di essi benefici e limitazioni. Inoltre descrive i metamodelli specifici delle applicazioni Windows Phone 7 e Android, ripresi dallo stato dell'arte.

- Il *terzo capitolo* presenta il metamodello astratto, che riproduce la struttura generale delle applicazioni mobili. In quanto basato sul linguaggio UML, verranno ripresi alcuni dei concetti UML esistenti e ne saranno introdotti dei nuovi. Infine a titolo d'esempio riporta un'istanza del metamodello.

- Il *quarto capitolo* illustra il metamodello implementativo, che si presenta come evoluzione del metamodello astratto, fornendo una descrizione dettagliata per ciascuno dei suoi componenti.

- Il *quinto capitolo* descrive gli strumenti e le tecnologie sfruttati nella fase di definizione del generatore di codice e presenta la struttura del progetto realizzato. Inoltre descrive le regole, le convenzioni e le notazioni da adottare al momento dell'utilizzo del generatore di codice e spiega quale sia l'output della fase di traduzione del modello dell'applicazione nei sorgenti corrispondenti.

- Il *sesto capitolo* fornisce una valutazione quantitativa e qualitativa del lavoro di tesi, e in particolare del generatore di codice. Al fine di ottenere dei dati per l'analisi quantitativa abbiamo sviluppato e analizzato due semplici applicazioni. Per l'analisi qualitativa abbiamo considerato invece un'applicazione di media complessità presente sul mercato. Grazie agli esempi presentati, al termine del capitolo risulterà chiaro come utilizzare lo strumento software per definire il modello dell'applicazione.

- Il *settimo capitolo* riporta le considerazioni finali sul lavoro svolto e illustra i possibili sviluppi futuri.

Capitolo 2

Stato dell'arte

Questo capitolo presenta il contesto scientifico nel quale il lavoro di tesi si inserisce. In particolare descrive le differenze basilari tra le applicazioni native e quelle sviluppate con un approccio di tipo cross-platform, il metamodello Windows Phone 7 e il metamodello Android.

2.1 Applicazioni Cross-Platform

La vastità e diversità di dispositivi e sistemi operativi presenti sul mercato porta le aziende a dover produrre e distribuire la stessa applicazione più volte, una per ciascuna delle diverse piattaforme mobili esistenti, rendendo così necessaria la ricerca di una soluzione a questo problema. In alcuni casi le aziende si affidano a più team di sviluppatori, ciascuno dei quali specializzato nello sviluppo di app per uno specifico sistema operativo, in altri casi si affidano invece a una delle strategie di sviluppo cross-platform, che verranno presentate di seguito. Attualmente esistono infatti due approcci allo sviluppo di applicazioni per dispositivi mobili: l'approccio classico, che produce applicazioni mobili native, e quello cross-platform.

Il beneficio principale, che proviene dallo sviluppo di una **applicazione nativa**, è quello di avere accesso a tutti i componenti hardware installati sul dispositivo: NFC, accelerometro, GPS, fotocamera, magnetometro, sensore di prossimità, sensore di luminosità, sensore di posizionamento e così via. Un altro aspetto riguarda la possibilità di pubblicare l'applicazione sullo store: l'abitudine degli utenti a cercare nei negozi virtuali le applicazioni che vogliono installare sul proprio smartphone rende significativo questo vantaggio. Un'applicazione nativa installata sul dispositivo, inoltre, è accessibile anche in modalità offline, a differenza di quanto accade per le applicazioni web che, come vedremo tra poco, richiedono necessariamente la connessione a Internet. Dal punto di vista dell'utilizzo, l'interfaccia grafica nativa è in

genere preferita dall'utente ed è più reattiva rispetto a quella interpretata dal browser perché le operazioni per la sua gestione sono svolte direttamente dal sistema. Considerando infine un aspetto di marketing, le aziende possono sfruttare le applicazioni native per mandare messaggi promozionali in base alla geo-localizzazione dell'utente, al contrario di quanto accade per le applicazioni multipiattaforma.
Tuttavia lo sviluppo di un'applicazione nativa implica alti costi, tempistiche lunghe e una grande quantità di risorse laddove si voglia rendere disponibile l'applicazione per diverse piattaforme, in quanto lo stesso progetto deve essere portato avanti da diversi team, con capacità e strumenti specifici [1]. Per definizione le applicazioni native sono infatti realizzate e compilate sulla base di una determinata piattaforma. Le applicazioni per iOS (iPhone, iPad e iPod), per esempio, sono scritte in Objective-C, quelle per Android sono sviluppate in Java, mentre quelle per Windows Phone sono sviluppate in ambiente .NET. Esistono applicazioni tuttora sviluppate anche per i sistemi operativi meno diffusi, come RIM per Blackberry o Symbian per i vecchi Nokia.

Una soluzione ideale, obiettivo delle piattaforme di sviluppo **cross-platform**, potrebbe essere quella di creare e mantenere un'unica applicazione che possa funzionare su piattaforme diverse. Infatti lo sviluppo cross-platform comporta una riduzione delle capacità richieste, della quantità di codice e dei tempi e costi di sviluppo e mantenimento, senza avere la necessità di conoscere specifiche API. Nonostante questi vantaggi le attuali piattaforme cross-platform non sembrano essere ancora del tutto mature e esenti da difetti al punto da sostituirsi completamente alla metodologia classica di sviluppo.

Di seguito riportiamo una descrizione dei vari approcci e dei relativi tool utilizzati nello sviluppo cross-platform [2][3][4], illustrando i principali vantaggi e svantaggi di ciascuno di essi.

2.1.1 Approccio Web

Le applicazioni sviluppate con un approccio web sono progettate per essere eseguite attraverso il web browser del dispositivo, senza che vengano installate sullo stesso: appartengono a questa categoria tutte le web application accessibili dai dispositivi mobili. La business logic dell'applicazione è implementata sul server, mentre sul client risiede unicamente l'interfaccia grafica e la logica di gestione dei dati dell'utente. Il fatto che l'applicazione si basi sull'utilizzo del browser e sia server-driven fa in modo che sia completamente indipendente dalla piattaforma. Uno dei principali vantaggi di questo ap-

proccio è il fatto di poter sfruttare gli strumenti della programmazione web, quali HTML, CSS e JavaScript. Inoltre, l'applicazione non necessita di installazione o aggiornamenti sullo specifico device, in quanto salvata sul server ed acceduta di volta in volta attraverso il web browser, rendendo così riutilizzabile l'interfaccia grafica sulle diverse piattaforme.
Un'importante limitazione di questo approccio è quella per cui le applicazioni prodotte non possono accedere al software e all'hardware del dispositivo, inoltre le performance dell'applicazione possono essere aggravate dall'indispensabile utilizzo del browser e del collegamento a Internet. Infine bisogna considerare che l'utente è abituato a cercare le applicazioni sullo store ufficiale, dove chiaramente non si trovano quelle appartenenti a questa categoria.

AppsBuilder [5] e *iBuildApp* [6] sono dei siti che offrono la possibilità di creare gratuitamente online, in maniera semplice e senza scrivere alcuna riga di codice, applicazioni di questo tipo. Tali servizi offrono un ambiente grafico per costruire le view dell'applicazione, le quali vengono automaticamente realizzate in linguaggio HTML5, supportando in questo modo un approccio di tipo cross-platform.
Il procedimento per comporre le applicazioni è simile tra i due siti: si parte dalla definizione di un tema principale a scelta fra quelli proposti per poi andare ad aggiungere, con semplici operazioni di drag and drop, le view con i relativi contenuti, quali immagini, video e testi. AppsBuilder permette addirittura l'inserimento di file .pdf, mentre iBuildApp appare più rigido nella scelta delle risorse da poter inserire. In entrambi i casi esiste la possibilità di pubblicare il prodotto finale su uno store e quindi renderlo accessibile agli utenti finali.
Nonostante sia facile e immediato ottenere un'applicazione con questi strumenti, il prodotto non è particolarmente soddisfacente per diversi motivi. Innanzitutto la struttura dell'applicazione è rigida in quanto si è costretti a scegliere tra uno dei template proposti e una volta scelto non esiste la possibilità di modificarlo, poi la grafica è essenziale ed è la stessa su tutti i dispositivi su cui l'applicazione sarà installata. Inoltre non è possibile accedere a nessuna delle API del sistema operativo nativo e non potendo scrivere codice la personalizzazione delle view è davvero riduttiva. Questo tipo di applicazioni è pertanto consigliato ad utenti inesperti che desiderano realizzare in poco tempo e senza scrivere alcuna riga di codice una piccola applicazione mobile, installabile su tutte le piattaforme.

2.1.2 Approccio Ibrido

L'approccio ibrido si colloca a metà tra le metodologie nativa e web, in quanto le applicazioni sono sviluppate utilizzando le classiche tecnologie web, ma vengono eseguite all'interno di container nativi sul device. Infatti, l'interfaccia utente è mostrata attraverso un web browser ma può essere riusata sfruttando le caratteristiche native della piattaforma, mentre le risorse hardware (fotocamera, microfono, sensori, etc.) sono rese accessibili all'applicazione ibrida attraverso un layer astratto, che rende disponibili le funzionalità del dispositivo grazie a delle API JavaScript. A differenza delle applicazioni web, le applicazioni che appartengono a questa categoria possono essere distribuite sullo store e quindi sono scaricabili e installabili sullo specifico device.
Tra gli svantaggi di questo approccio emerge il fatto che le performance sono inferiori rispetto a quelle delle applicazioni native, perché l'esecuzione avviene nel browser engine. Inoltre il codice JavaScript può essere interpretato in modo differente da ogni dispositivo e, per quanto riguarda l'interfaccia grafica, nonostante possa essere riutilizzata su diverse piattaforme, in genere non presenta lo stile grafico delle applicazioni native.
Tra i tool esistenti che si basano su questo approccio troviamo PhoneGap e MoSync, descritti di seguito.

PhoneGap [7] è un ambiente di sviluppo completamente open-source, sotto la licenza del progetto Apache Cordova, che permette di creare applicazioni mobili attraverso i linguaggi di programmazione web moderni (HTML, CSS e JavaScript) e le funzionalità dell'SDK, risparmiando così allo sviluppatore l'apprendimento di linguaggi meno noti come Objective-C, Java, etc. PhoneGap non dispone di un proprio IDE, al contrario lo sviluppatore si appoggia agli IDE nativi dei sistemi operativi mobile (Eclipse per Android, Xcode per iOS, etc). In questo modo non viene fornito al programmatore un ambiente di sviluppo centralizzato, ma grazie all'uso di IDE diversi PhoneGap può essere sfruttato su diversi sistemi operativi come Mac, Linux e Windows.
Con HTML, CSS e Javascript viene definita anche l'interfaccia grafica dell'applicazione, mostrata all'utente grazie al web browser del dispositivo.
PhoneGap possiede un'API che permette di accedere alle funzionalità native del sistema operativo, quindi la logica dell'applicazione viene scritta in JavaScript e l'API PhoneGap si occuperà di gestire la comunicazione con il sistema operativo nativo. Queste caratteristiche rendono l'approccio di sviluppo un approccio cross-platform ibrido.
Il prodotto finale è un archivio binario dell'applicazione che può essere distribuito sulle diverse piattaforme, come se l'applicazione fosse stata implementata con linguaggi nativi: per iOS l'output è un file .ipa, per Android un file .apk, per Windows Phone un file .xap, etc.

Per quanto riguarda i dati, PhoneGap non gestisce alcun database locale, quindi è necessaria l'interazione con un server che si occupi di procurare i dati all'applicazione client attraverso un suo database. In genere la comunicazione tra client e server avviene tramite richieste HTTP che utilizzano XML o JSON come formati per lo scambio dei dati.

MoSync [8] è un ambiente di sviluppo open source di applicazioni ibride simile a PhoneGap. La differenza principale tra i due risiede nel fatto che MoSync offre un proprio IDE, unico per tutte le piattaforme, installabile sui sistemi operativi Windows e Mac. Nel caso in cui non fosse possibile implementare qualche funzionalità con JavaScript, MoSync permette di implementarle direttamente in linguaggio C++, senza incorrere in problemi di compatibilità con le specifiche piattaforme. MoSync offre, inoltre, la possibilità di integrare all'interno dell'applicazione un database SQLite, una libreria software scritta in linguaggio C che implementa un Database Management System di tipo ACID e che permette di realizzare una base di dati incorporata in un unico file.

2.1.3 Approccio Interpretato

Il codice dell'applicazione viene caricato sul dispositivo e interpretato in un secondo momento, grazie ad un interprete che esegue il codice sorgente a runtime sulla specifica piattaforma, offrendo in questo modo la possibilità di riutilizzare la logica applicativa su diverse piattaforme e quindi di supportare uno sviluppo di tipo cross-platform. Le caratteristiche hardware e software della piattaforma sono accedute tramite un framework: l'applicazione interpretata interagisce con un layer astratto per accedere alle API native del dispositivo e sfrutta gli elementi dell'interfaccia grafica della specifica piattaforma, offrendo all'applicazione un'interfaccia utente che corrisponde a quella dell'applicazione nativa.
I principali problemi riguardano le performance che possono degradare drasticamente per via dell'interpretazione del codice a runtime e in generale lo sviluppo e il riutilizzo dell'interfaccia utente dipendono fortemente dal livello di astrazione del framework scelto. Tra le piattaforme di sviluppo che seguono questo approccio troviamo Rhodes e Appcelerator Titanium.

Rhodes [9] è un ambiente di sviluppo open-source basato su Ruby e creato per sviluppare applicazioni interpretate per iOS, Android, BlackBerry, Windows Phone e Windows 8. Un'applicazione scritta con Rhodes è una web application che può essere eseguita direttamente sul dispositivo mobile e che viene implementata sfruttando il pattern MVC, dove le view sono un

insieme di template HTML, file CSS e file JavaScript che possono essere eseguiti da un piccolo web server locale presente sullo smartphone, il controller è costituito da un insieme di script Ruby salvati in file con estensione .rb e che possono accedere ad alcune caratteristiche del dispositivo, mentre il model è costituito anch'esso da script Ruby salvati come file .rb. Inoltre, le applicazioni costruite con Rhodes possono usare database locali come SQLite.

Appcelerator Titanium è framework open-source per scrivere applicazioni mobili basato esclusivamente su un SDK Javascript. Può essere scaricato sia nella versione stand-alone, che comprende una versione di Eclipse già configurata per lo sviluppo di applicazioni mobili, sia come plugin. In quest'ultimo caso è necessario installare anche l'SDK della piattaforma per cui si intende sviluppare. I programmatori scrivono tutto il codice sorgente unicamente in JavaScript e tramite le API del framework possono accedere alle funzionalità del sistema e definire l'interfaccia grafica. Durante la compilazione Titanium combina i codici sorgenti con un interprete JavaScript e altri contenuti statici in un application package, mentre a runtime l'interprete processa codice Javascript. Per queste ragioni Titanium, come Rhodes, appartiene alla categoria di ambienti per lo sviluppo di applicazioni interpretate.

2.1.4 Approccio Cross-Compilato

Le applicazioni vengono scritte dallo sviluppatore in un linguaggio di programmazione comune e un cross-compiler è responsabile della traduzione del codice sorgente in file binari nativi. Questo approccio è pertanto strettamente dipendente dall'efficienza e affidabilità del cross-compiler. Il vantaggio di questo approccio è che le applicazioni prodotte hanno accesso a tutte le caratteristiche native della piattaforma e tutti i componenti dell'interfaccia grafica nativa possono essere utilizzati senza problemi.
Il problema è che l'interfaccia grafica non può essere riutilizzata, così come le caratteristiche hardware native. Queste caratteristiche, infatti, dipendono dalla specifica piattaforma e il modo di accedervi è strettamente legato ad essa. Inoltre l'approccio non è adatto per lo sviluppo di applicazioni sofisticate, dove la cross-compilazione diventa complicata. I due tool più utilizzati che sfruttano questo approccio sono Xamarin e Corona.

Xamarin [10] è un ambiente di sviluppo per applicazioni cross-compilate che permette al programmatore di utilizzare il linguaggio C# per scrivere applicazioni eseguibili su piattaforme Android e iOS. Il compilatore Xamarin offre tutte le API native di iOS e Android come normali librerie C#, permettendo così l'accesso a tutte le caratteristiche hardware dello specifico

device. Per eseguire il codice sul dispositivo, Xamarin si appoggia a Mono, un'implementazione open source del framework Microsoft .NET, che prende il posto della macchina virtuale nativa del sistema operativo, producendo così lo stesso output su piattaforme diverse.

Corona [11] è un ambiente di sviluppo che consente di creare applicazioni mobili secondo un approccio cross-compilato, appoggiandosi al linguaggio di programmazione Lua per astrarre lo sviluppo dell'applicazione da uno specifico sistema operativo. Corona possiede un proprio IDE e può essere scaricato in diverse versioni a partire da quella gratuita con funzionalità limitate sino ad arrivare alla versione enterprise ricca di funzionalità, come l'integrazione con le librerie native. Corona possiede anche un proprio SDK che viene costantemente aggiornato dalla community ed un proprio simulatore. Questo tool è considerato particolarmente utile per l'implementazione di videogiochi, ma risulta inadatto allo sviluppo di applicazioni mobili tradizionali.

2.1.5 Altri approcci

La classificazione riportata sopra, sebbene copra la maggior parte degli approcci cross-platform che concorrono allo sviluppo di applicazioni mobili, non è esaustiva per via del fatto che il mondo delle applicazioni mobili è in continua evoluzione, portando sempre nuovi strumenti e metodologie di sviluppo. Uno strumento che non segue nessuno fra gli approcci sopra citati è DragonRAD.

DragonRAD [12] è un ambiente di programmazione distribuito con licenza commerciale e dotato di un proprio IDE, che permette lo sviluppo di applicazioni per iOS, Android, BlackBerry e Windows Phone. La piattaforma è stata progettata con l'obiettivo di semplificare il design, lo sviluppo, il mantenimento e l'amministrazione di applicazioni mobili database-driven, senza la necessità di dover scrivere righe di codice, usando il tool grafico incluso nel framework, che offre un intuitivo ambiente drag and drop. Ai fini di soddisfare sofisticati requisiti applicativi DragonRAD offre anche un supporto per poter scrivere degli script in linguaggio Lua.
DragonRAD ha un'architettura host-client, che offre un pacchetto completo con tutti i prerequisiti per accogliere i server e database più noti (Tomcat, MySql, etc): è richiesto il setup del server e del database, in base alle esigenze dello sviluppatore. In questo modo DragonRAD facilita l'integrazione e la sincronizzazione del database sul server con le funzioni native delle piattaforme mobili: contatti, calendario, geolocation, etc. Una limitazione di DragonRad è il fatto di essere usato principalmente per sviluppare applica-

zioni enterprise, che richiedono la sincronizzazione con sistemi di backend, quindi non è del tutto indicato a sviluppatori di applicazioni stand-alone.

2.2 Metamodelli UML esistenti

Nella modellazione di applicazioni basate sulla programmazione ad oggetti il linguaggio maggiormente utilizzato è UML (Unified Modeling Language). Le proposte presentate finora in merito alla realizzazione di metamodelli per applicazioni mobili si basano proprio sull'estensione di questo linguaggio di natura general purpose, che, in quanto tale, da solo non potrebbe garantire una completa descrizione di una specifica applicazione mobile. Per metamodello si intende un modello di specifica per una classe di sistemi, in cui ogni sistema è rappresentato da un modello valido, espresso in un certo linguaggio di modellazione. Il metamodello UML in particolare contiene elementi che permettono di modellare in maniera semplice oggetti e relazioni tra di essi. Passiamo dunque ad analizzare i metamodelli UML presenti in letteratura per le piattaforme Windows Phone 7 e Android [13][14].

2.2.1 Metamodello Windows Phone 7

Gli autori nel loro lavoro [13] partono dall'idea di voler definire un metamodello in grado di esprimere ad alto livello le caratteristiche fondamentali delle applicazioni sviluppate per il sistema operativo Windows Phone 7. Dopo un analisi approfondita della struttura della piattaforma presentano una classificazione delle caratteristiche, che vengono suddivise in caratteristiche software e caratteristiche hardware. Gli elementi software sono usati per definire un modello che descriva la struttura di una generica applicazione, mentre gli elementi hardware sono usati per descrivere l'utilizzo delle risorse hardware da parte dell'applicazione. In questo modo gli autori riescono a modellare indipendentemente la parte software e la parte hardware e, al momento della migrazione di un'applicazione sviluppata per uno smartphone su un dispositivo appartenente ad un'altra categoria, come un tablet o uno smartbook, è richiesta unicamente la modifica delle parti relative alle risorse hardware, in quanto le parti software rimangono invariate. Entrambe le parti, descritte in dettaglio di seguito, sono state modellate appoggiandosi al linguaggio UML e ampliandolo con degli stereotipi.

Caratteristiche Software

Gli stereotipi che descrivono le caratteristiche software del sistema operativo sono stati definiti seguendo il pattern MVC (Model View Controller), dove il Model rappresenta i contenuti, la View costituisce l'interfaccia grafica

che mostra i contenuti, e il Controller contiene la logica applicativa dell'applicazione, che controlla i contenuti e l'interfaccia grafica. In Figura 2.1 è mostrato il metamodello del software, dove è possibile notare l'utilizzo del pattern MVC tra gli stereotipi che modellano l'applicazione.

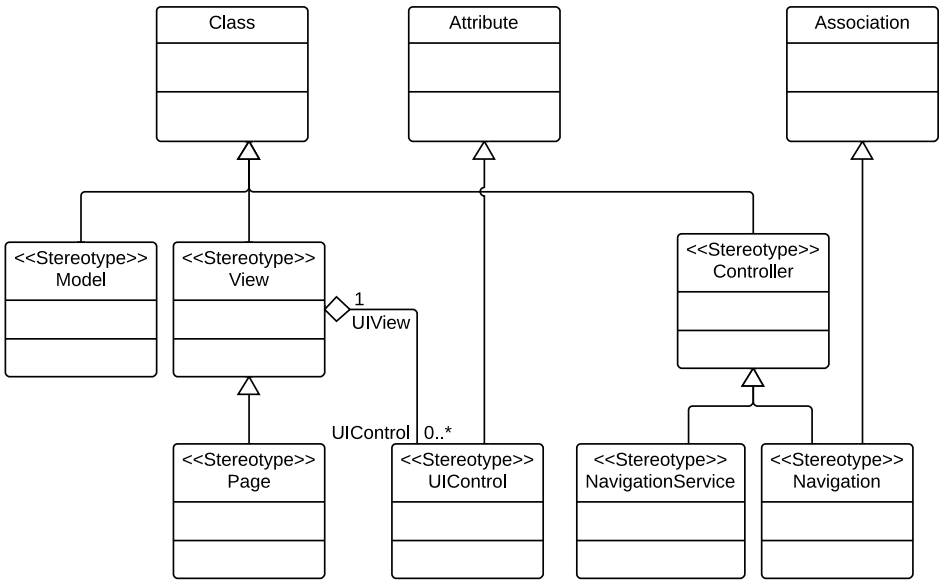

Figura 2.1: *WP7*: Metamodello del software

Il **Model** rappresenta unicamente le classi che presentano contenuti e che mantengono l'informazione durante l'esecuzione dell'applicazione. Al verificarsi di un certo evento è poi il **Controller** che si occupa di estrarre i contenuti dal Model e mostrarli sull'interfaccia utente. La classe Controller è quindi indicata nel metamodello del software con lo scopo di modellare l'identificazione e il controllo del flusso di dati tra gli elementi del sistema. La **View** è invece relazionata all'interfaccia grafica, perciò richiede di essere implementata con attenzione, ai fini di soddisfare l'utente utilizzatore dell'applicazione. Per quanto riguarda la parte grafica, a design time non è semplice determinare quale elemento verrà usato, infatti i dispositivi, smartphone o tablet che siano, possono avere diverse tipologie di componenti dell'interfaccia. Per questo motivo, a design time, gli autori hanno preferito indicare nel metamodello lo stereotipo View, che rappresenta in maniera astratta l'insieme dei componenti grafici. Il componente specifico viene poi determinato in fase di implementazione, dopo aver stabilito il target device.

Caratteristiche Hardware e Funzioni Fondamentali

Nel metamodello Windows Phone 7 per caratteristiche hardware si intende l'insieme di tutte le risorse hardware del dispositivo accessibili dall'applicazione sviluppata. Le classi che contribuiscono alla modellazione di questo genere di risorse corrispondono anche in questo caso a degli stereotipi e sono riportate in Figura 2.2. Per problemi di spazio nel modello presentato sono riportate solo alcune delle svariate risorse normalmente disponibili all'interno di un dispositivo, ovvero quelle che gli autori hanno considerato essere le più utilizzate. Tra le risorse modellate troviamo l'accelerometro, il GPS, il microfono, la fotocamera e il sensore di rete.

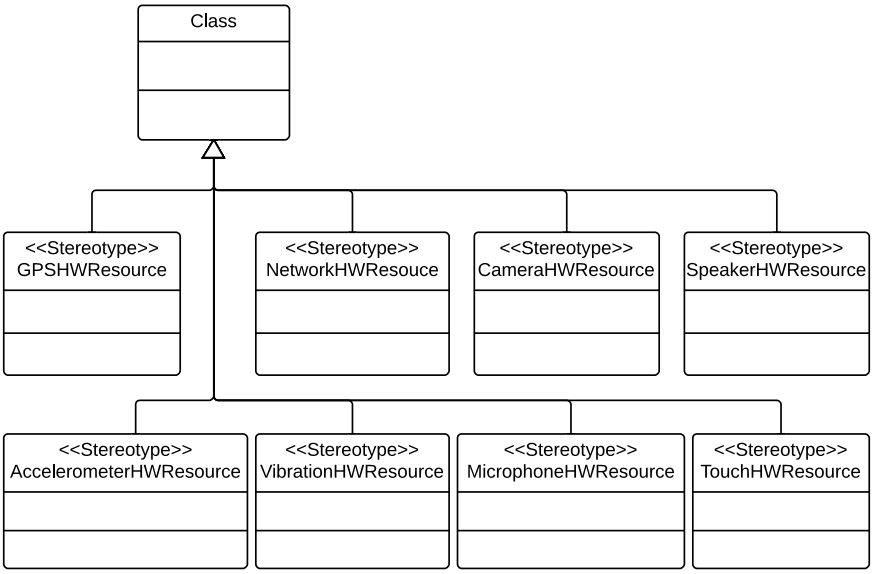

Figura 2.2: *WP7*: Metamodello delle risorse hardware

Oltre alla risorse hardware sono state modellate anche alcune delle funzioni fondamentali rese disponibili dal sistema operativo con lo scopo di migliorare l'user experience e le performance dell'applicazione. Tali funzioni sono riportate in Figura 2.3. La prima di esse è l'*Isolate Storage*, un meccanismo di archiviazione dei dati supportato da Windows Phone 7 che offre isolamento e sicurezza. Con l'Isolate Storage i dati sono infatti protetti da apposite applicazioni che dispongono dell'accesso allo spazio di memorizzazione isolato [15]. Un'altra funzione, chiamata *Security*, è un aspetto rilevante nei dispositivi mobili che sfruttano tecnologie come l'accelerazione dell'hardware, le quali possono essere utilizzate a scopi malevoli. In Silverlight, la piattaforma per lo sviluppo di applicazioni Windows Phone 7, esiste il concetto di

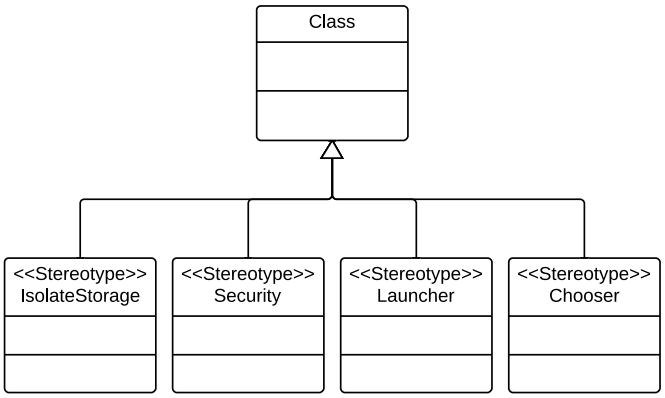

Figura 2.3: *WP7*: Metamodello delle funzioni fondamentali

sandbox, che fornisce un ambiente in cui le applicazioni, per questioni di sicurezza, hanno privilegi limitati e non hanno accesso al file system e alle altre risorse: ogni applicazione è eseguita all'interno della propria sandbox. I programmatori che nello sviluppo della loro applicazione necessitano di accedere alle caratteristiche del dispositivo possono sfruttare delle API chiamate *Launcher* e *Chooser*. Le Launcher sono utilizzate per invocare applicazioni che non ritornano alcun tipo di dato all'applicazione chiamante, mentre le Chooser invocano applicazioni finalizzate a ritornare dati all'applicazione che le ha chiamate [16].

2.2.2 Metamodello Android

Un'applicazione Android è costituita da una serie di interfacce utente, risorse e altri concetti che concorrono al suo corretto funzionamento: molti di questi elementi si prestano ad essere modellati nel momento in cui il programmatore decide di sviluppare un'applicazione. L'articolo [14] riporta una soluzione alla modellazione di questi elementi, presentati anche in questo caso come stereotipi del linguaggio UML. Attraverso lo stereotipo UML si riesce infatti a sfruttare ed estendere il linguaggio di modellazione senza dover necessariamente cambiare la sua notazione o sintassi. Il lavoro svolto dagli autori consiste, in particolare, nell'identificazione e nell'analisi delle caratteristiche della piattaforma Android e nella conseguente produzione di un metamodello UML esteso, in grado di offrire allo sviluppatore un insieme esaustivo di elementi di modellazione, utili nella fase di design delle applicazioni per questa piattaforma.

Gli autori classificano le caratteristiche della piattaforma Android in strut-

turali e dinamiche, che comprendono rispettivamente gli elementi base che formano l'applicazione e i metodi usati dagli sviluppatori per controllare il comportamento di tali elementi.

Caratteristiche strutturali

Le caratteristiche strutturali comprendono l'interfaccia utente, le risorse di sistema e i componenti fondamentali che caratterizzano una qualsiasi applicazione Android.

L'**interfaccia utente** è una parte fondamentale per tutte le applicazioni mobili e Android mette a disposizione molti strumenti per personalizzare le schermate mostrate. I componenti dell'interfaccia grafica sono i Widget (Button, TextView, WebView, ImageView, etc.) e gli UI Event Processing, ovvero gli elementi che processano gli eventi scatenati dall'utente che interagisce col dispositivo.
Tutta questa parte è stata modellata seguendo il pattern MVC: gli elementi grafici sono stati definiti aggiungendo gli stereotipi *Controller* e *Model*, come mostrato in Figura 2.4. Non è presente un vero e proprio stereotipo View per via del fatto che sono stati esplicitamente indicati concetti come *UIControlButton*, che modella gli elementi Button e ImageButton, *UIControlViewer*, che modella gli elementi ListView, WebView e MapView e *UIControlText*, che modella gli elementi EditText e TextView.

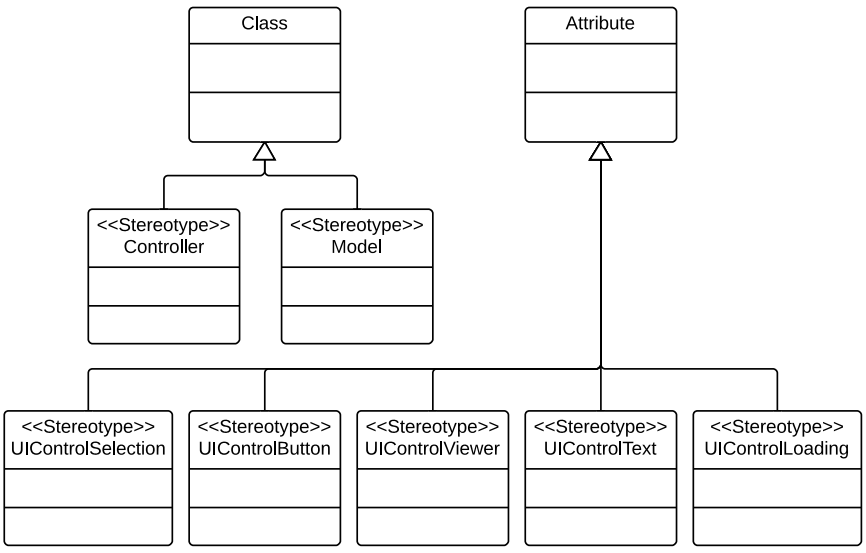

Figura 2.4: *Android*: Metamodello dell'interfaccia grafica

Le **risorse di sistema** sono invece costituite dalle librerie esterne, ovvero le librerie di sistema di cui fa uso l'applicazione, dalle content resource, intese come i file audio, i file di testo, le immagini e tutti gli altri file multimediali utilizzati dall'applicazione, e dalle risorse hardware, che rappresentano i componenti hardware e sensori del dispositivo a cui l'applicazione richiede l'accesso.
Per queste risorse sono stati definiti gli stereotipi *ResourceController* e *ExternalLibrary*, mostrati in Figura 2.5. Il primo, indicato come figlio dell'elemento Class, è stato introdotto per modellare la gestione delle content resource e delle risorse hardware, mentre il secondo, definito come figlio del concetto Subsystem del linguaggio UML, è stato introdotto per modellare le librerie di sistema esterne. Per le risorse hardware è stato poi definito un metamodello molto simile a quello presentato per modellare le caratteristiche hardware della piattaforma Windows Phone 7, che abbiamo quindi preferito non riportare.

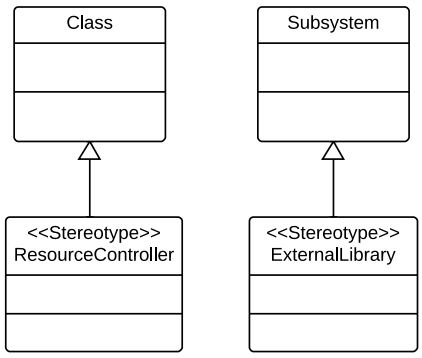

Figura 2.5: *Android*: Metamodello delle risorse di sistema

I **componenti fondamentali** che caratterizzano le applicazioni Android sono i seguenti:

- *Activity*: componente dell'applicazione che fornisce una schermata con cui l'utente può interagire per svolgere determinate operazioni. Ogni Activity agisce infatti all'interno di una window nella quale si trova l'interfaccia grafica. Un'applicazione solitamente è costituita da più Activity collegate tra loro: una fra queste viene mostrata per prima e in genere viene definita MainActivity. Quando viene attivata una nuova Activity, quella precedente viene arrestata e inserita in uno stack che funziona seguendo una logica LIFO. Nel momento in cui viene premuto il tasto "back" sul dispositivo l'Activity corrente viene rimossa dallo stack e quella precedente viene mostrata a schermo. Se una generica

Activity viene fermata riceve una notifica del cambiamento di stato attraverso i metodi di callback del ciclo di vita, che come vedremo più avanti sono utilizzati per gestire la sua creazione, ripresa e distruzione [17].

- *Service*: componente dell'applicazione che non presenta un'interfaccia utente e che viene utilizzato per eseguire operazioni di lunga durata che non richiedono l'interazione con l'utente, come la riproduzione di brani musicali o il download di immagini da Internet, oppure per rendere disponibile una certa funzionalità ad altre applicazioni installate sul dispositivo [18].

- *Content Provider*: componente che gestisce l'accesso ai dati strutturati dell'applicazione che devono essere condivisi con le altre applicazioni. I Content Provider delle applicazioni incapsulano i dati e forniscono meccanismi per definire la sicurezza dei dati. Per accedere ai dati di un certo Content Provider, nel contesto dell'applicazione che si sta sviluppando, si utilizza l'oggetto Content Resolver, che funge da client nella comunicazione col provider e il cui compito è quello di rispondere alle richieste. Il Content Provider è pertanto necessario solo nel caso si voglia costruire un'applicazione che condivide i suoi dati con altre applicazioni [19].

- *Broadcast Receiver*: componente dell'applicazione in grado di ricevere gli annunci dal sistema finalizzati a notificare determinati eventi. Il Broadcast Receiver è sempre in ascolto e i messaggi inviati e ricevuti riguardano informazioni di vario genere: notifica di batteria scarica, notifica della perdita del segnale GPS, notifica di chiamata o sms in entrata, etc.

Poiché ogni componente viene eseguito singolarmente, Android mette a disposizione un oggetto *Intent* per far comunicare i componenti a runtime.
I componenti fondamentali sono riportati in Figura 2.6, dove si può notare come siano stati tutti definiti come figli dell'elemento Class. L'elemento Intent invece è presentato come figlio di Attribute ed è associato all'elemento *Navigation*, figlio di Association, ai fini di modellare il ruolo di intermediazione rivestito dall'oggetto Intent nei confronti dei quattro componenti base di cui sono composte le applicazioni Android.

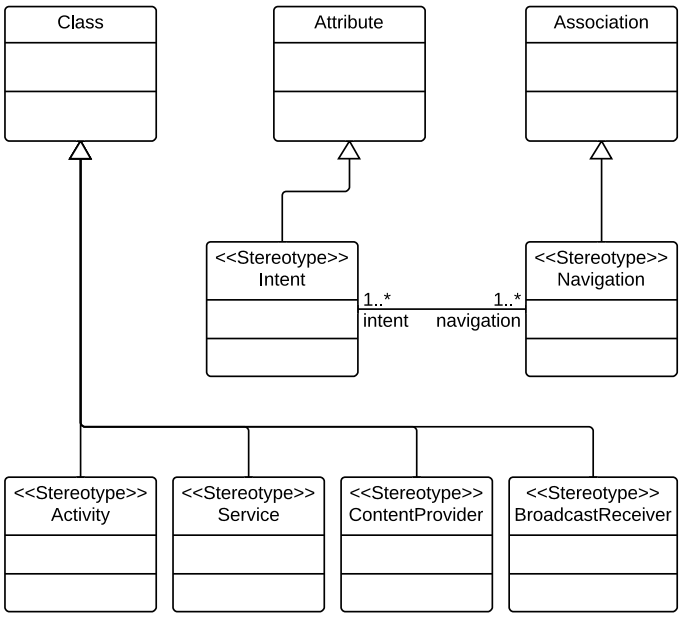

Figura 2.6: *Android*: Metamodello dei componenti fondamentali

Caratteristiche dinamiche

In un'applicazione Android ogni componente fondamentale ha un particolare ciclo di vita che inizia con la sua creazione e si conclude con la distruzione, che nel caso delle Activity avvengono rispettivamente con l'aggiunta e la rimozione delle stesse dallo stack presentato in precedenza. Le caratteristiche dinamiche definite dagli autori per la piattaforma Android cercano di modellare proprio la gestione del ciclo di vita dei componenti fondamentali Activity e Service. Per questo motivo nel metamodello è stato introdotto anche uno stereotipo *LifeCicle*, definito come figlio di Operation, in quanto in Android la gestione del ciclo di vita di questi componenti avviene sulla base dell'implementazione di questi metodi.

2.3 Conclusioni

In questo capitolo abbiamo presentato alcuni approcci allo sviluppo di applicazioni mobili, citando per ciascuno di essi principali benefici e limitazioni. Abbiamo descritto nel dettaglio la metodologia cross-platform, che prevede lo sviluppo di un'unica applicazione installabile su più piattaforme. I problemi principali di questa metodologia sono riconducibili all'impossibilità di sfruttare tutte le API native del sistema operativo, ad avere performance limitate e, nel caso delle applicazioni sviluppate con un approccio web, al-

l'essere vincolati a dover sfruttare il browser per utilizzare l'applicazione.
In ogni caso i problemi di compatibilità delle applicazioni sui diversi dispositivi non sono legati unicamente al sistema operativo, ma anche al fatto che ciascun device presenta uno schermo con risoluzione e dimensioni specifiche, processori grafici più o meno potenti ed in generale caratteristiche hardware differenti. Sorge pertanto la necessità di uno studio approfondito dei dispositivi esistenti e dei loro sistemi operativi da un punto di vista astratto, modellistico.
In seguito alla presentazione delle metodologie di sviluppo esistenti abbiamo descritto due approcci concreti alla modellazione delle applicazioni mobili, entrambi basati sull'estensione del linguaggio UML e finalizzati alla definizione di un metamodello per una specifica piattaforma: per quanto interessanti, questi approcci non offrono direttamente un contributo pratico allo sviluppo di applicazioni multipiattaforma. Da qui nasce la necessità di analizzare nel dettaglio le analogie e le differenze che risiedono nella struttura delle applicazioni mobili appartenenti alle diverse piattaforme presenti sul mercato, e astrarre la modellazione dallo specifico sistema operativo.

Nel capitolo successivo entreremo nel dettaglio del lavoro di tesi presentando il primo dei risultati ottenuti, vale a dire il metamodello astratto, da noi definito come strumento di supporto alla fase di progettazione delle applicazioni mobili, indipendentemente dalla piattaforma per cui vengono realizzate.

Capitolo 3

Metamodello astratto

In questo capitolo presentiamo il nostro metamodello per le applicazioni mobili. L'obiettivo della fase di modellazione è quello di identificare ad alto livello i principali elementi comuni e le eventuali discordanze che possono esistere all'interno della struttura di applicazioni native di piattaforme diverse. Nel nostro caso la modellazione è stata definita a seguito di uno studio approfondito dei sistemi operativi Android e iOS, ma in generale il metamodello definito può essere considerato valido per una qualsiasi piattaforma mobile.

Come punto di partenza della fase di modellazione è stato scelto il linguaggio UML che, in quanto standard di specifica dei modelli di sviluppo, permette di rendere il nostro metamodello il più possibile universale.
Così come i metamodelli Android e Windows Phone 7 esistenti, il nostro metamodello comprende una frazione esistente dei concetti UML, ben conosciuta nei modelli di sviluppo per linguaggi ad oggetti, ed una parte nuova finalizzata all'introduzione di nuovi concetti specifici, che costituiscono i componenti base delle applicazioni mobili.

3.1 Parte esistente del metamodello

I concetti e le relazioni del linguaggio UML che compongono la parte esistente del metamodello sono riportati in Figura 3.1 e descritti di seguito [20][21].

- **Classifier**: è un elemento UML che descrive un insieme di istanze con le stesse caratteristiche (Feature) strutturali e/o comportamentali. Il Classifier rappresenta una metaclasse astratta utile a generalizzare concetti che possono avere delle istanze, come Class, Interface e Subsystem.

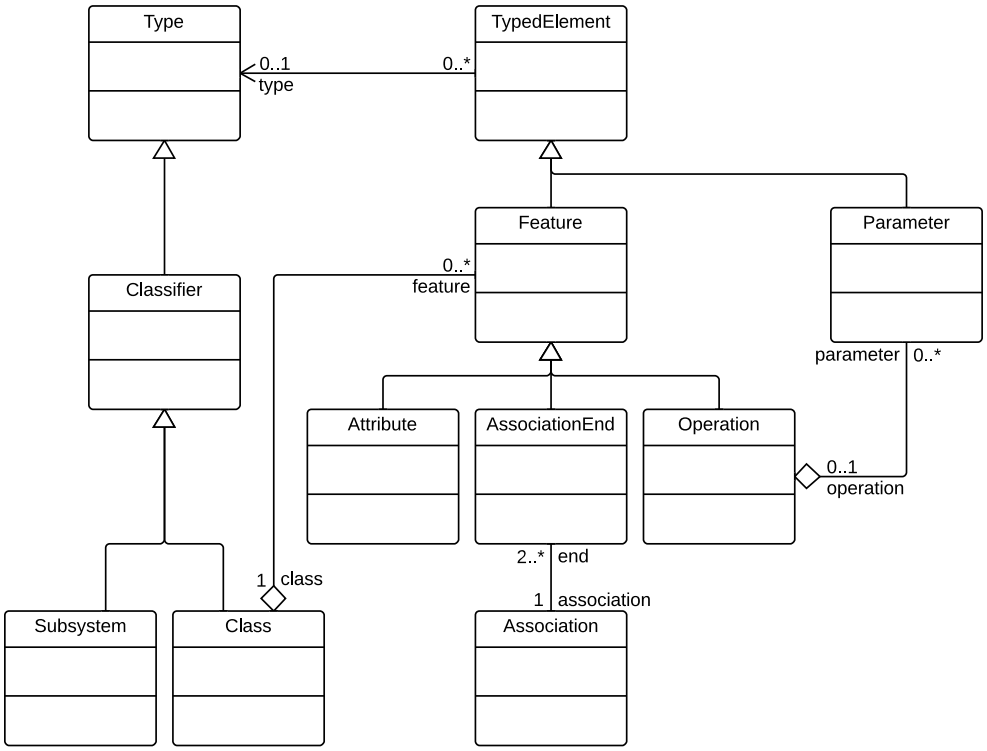

Figura 3.1: *Metamodello Astratto*: parte esistente

- **Association**: è una relazione binaria o n-aria tra Classifier utilizzata per mostrare che le istanze di uno o più Classifier possono essere collegate tra loro o combinate in aggregazioni. Tale concetto si contraddistingue da quello di *Generalization*, che rappresenta invece una relazione tassonomica tra un concetto più generale e uno più specifico. Talvolta un Classifier può essere definito attraverso una composizione di altri Classifier: questo è un tipo speciale di Association binaria denominata *Aggregation*, dove abbiamo da una parte il componente aggregato e dall'altra tutte le sue parti. Oltre ad Aggregation esiste la *Composition*, che rappresenta un tipo più forte di aggregazione: ogni sottocomponente in una Composition può appartenere ad un solo componente aggregato e quindi nel momento in cui quest'ultimo viene eliminato, anche tutte le sue parti vengono rimosse. Per questioni di spazio abbiamo preferito non riportare in Figura 3.1 questi ultimi concetti dello standard UML.

- **Subsystem**: è un componente, ovvero un Classifier, che rappresenta un'unità indipendente del sistema. Viene usato generalmente per

descrivere parti molto grandi di un sistema complesso.

- **Feature**: è un elemento che rappresenta una caratteristica strutturale o comportamentale di un Classifier o delle sue istanze. Le Feature non hanno una notazione propria, la quale è definita in maniera specifica dalle sue sottoclassi. Le possibili sottoclassi di Feature sono Attribute, che rappresentano le caratteristiche strutturali, Operation, che rappresentano le caratteristiche comportamentali, e AssociationEnd. Quando una Feature è una AssociationEnd il suo valore o i suoi valori sono legati all'istanza o alle istanze del Classifier all'altro lato dell'Association di cui fa parte.

- **Attribute**: è una proprietà, ovvero una Feature, di un Classifier che definisce una sua particolare caratteristica strutturale e quindi la struttura delle sue istanze. Il contesto dell'attributo è il Classifier in cui si trova.

- **Operation**: è una Feature di un Classifier che, invocata su una sua istanza, definisce una sua particolare caratteristica comportamentale e quindi specifica il nome, il tipo, i parametri e vincoli che caratterizzano tale caratteristica per invocare il comportamento associato. Nella programmazione ad oggetti le Operation sono spesso chiamate metodi.

- **AssociationEnd**: è considerata parte di una Association e rappresenta il tipo e la cardinalità di un'entità (Classifier) appartenente a una delle estremità dell'associazione. Ogni associazione binaria deve avere esattamente due AssociationEnd, ciascuna delle quali può avere un nome che indica il ruolo all'interno della associazione e una serie di proprietà che determinano la semantica di come il Classifier all'estremità partecipa alla relazione, tra cui in genere compare la cardinalità, indicante il numero di istanze del relativo Classifier partecipanti all'associazione.

- **Class**: è un Classifier che descrive un insieme di oggetti che condividono le stesse Feature, gli stessi vincoli e la stessa semantica, ovvero lo stesso significato. Tale elemento può avere da zero a molte Feature, ciascuna delle quali può essere statica o non statica: le Feature statiche rappresentano le caratteristiche del Classifier, mentre quelle non statiche sono da riferirsi alle singole istanze del Classifier.

- **Type**: è una metaclasse astratta che rappresenta un insieme di valori ed è usata come vincolo per l'intervallo di valori ammissibili per un TypedElement.

- **TypedElement**: è una metaclasse astratta con in genere un Type associato allo scopo di vincolare l'intervallo di valori che può assumere. Esistono casi in cui il TypedElement non ha un Type e può quindi avere valori di qualsiasi tipo.

- **Parameter**: è un TypedElement che rappresenta un parametro di una Operation. Ogni parametro di un'operazione ha un Type e una cardinalità e, al momento dell'invocazione di un'operazione su un Classifier, viene passato un argomento per ciascuno dei suoi parametri. Nel diagramma è specificata la cardinalità 0..1 sulla composizione che lega Parameter a Operation perché in UML esistono casi in cui le sottoclassi di Parameter non sono necessariamente associate a un'operazione.

Tutti questi elementi del linguaggio UML rispecchiano essenzialmente la struttura dei linguaggi Java, C# e C++, ma si adattano molto bene anche ad altri linguaggi ad oggetti, come Objective-C che sta alla base dello sviluppo di applicazioni per iOS.

3.2 Parte nuova del metamodello

In fase di studio delle piattaforme Android e iOS abbiamo riscontrato numerose differenze di base. Prima di tutto le applicazioni appartenenti alle due piattaforme sono sviluppate appoggiandosi a linguaggi diversi: nel caso di Android si utilizza prevalentemente il linguaggio Java, mentre nel caso iOS si utilizza Objective-C. Anche gli strumenti e i pattern architetturali utilizzati sono differenti, infatti le applicazioni iOS sono strutturate seguendo fortemente il pattern MVC, mentre in quelle Android la distinzione tra i ruoli Model, View e Controller non è così netta. Con questa premessa si inizia a capire che il risultato dell'analisi svolta sulle due piattaforme è frutto di un compromesso, che, in fase di modellazione, cerca di riassumere nel modo più completo e preciso possibile la struttura delle applicazioni di entrambe.

Tra gli elementi del linguaggio UML, oltre a quelli presentati in precedenza a introduzione della parte esistente del metamodello, è di fondamentale importanza l'elemento **Stereotype**, che permette di estendere i concetti esistenti nel linguaggio di modellazione per definirne dei nuovi, personalizzati. Infatti, grazie all'utilizzo di questo elemento è stato possibile delineare nuovi concetti relativi alle applicazioni mobili, introdotti di seguito e rappresentati in Figura 3.2, dove è mostrata la struttura del metamodello astratto.
Per facilitare al lettore la distinzione della parte esistente da quella nuova del metamodello, abbiamo posto in alto, nella figura, alcuni degli elementi appartenenti alla parte esistente del metamodello, che a differenza di quelli

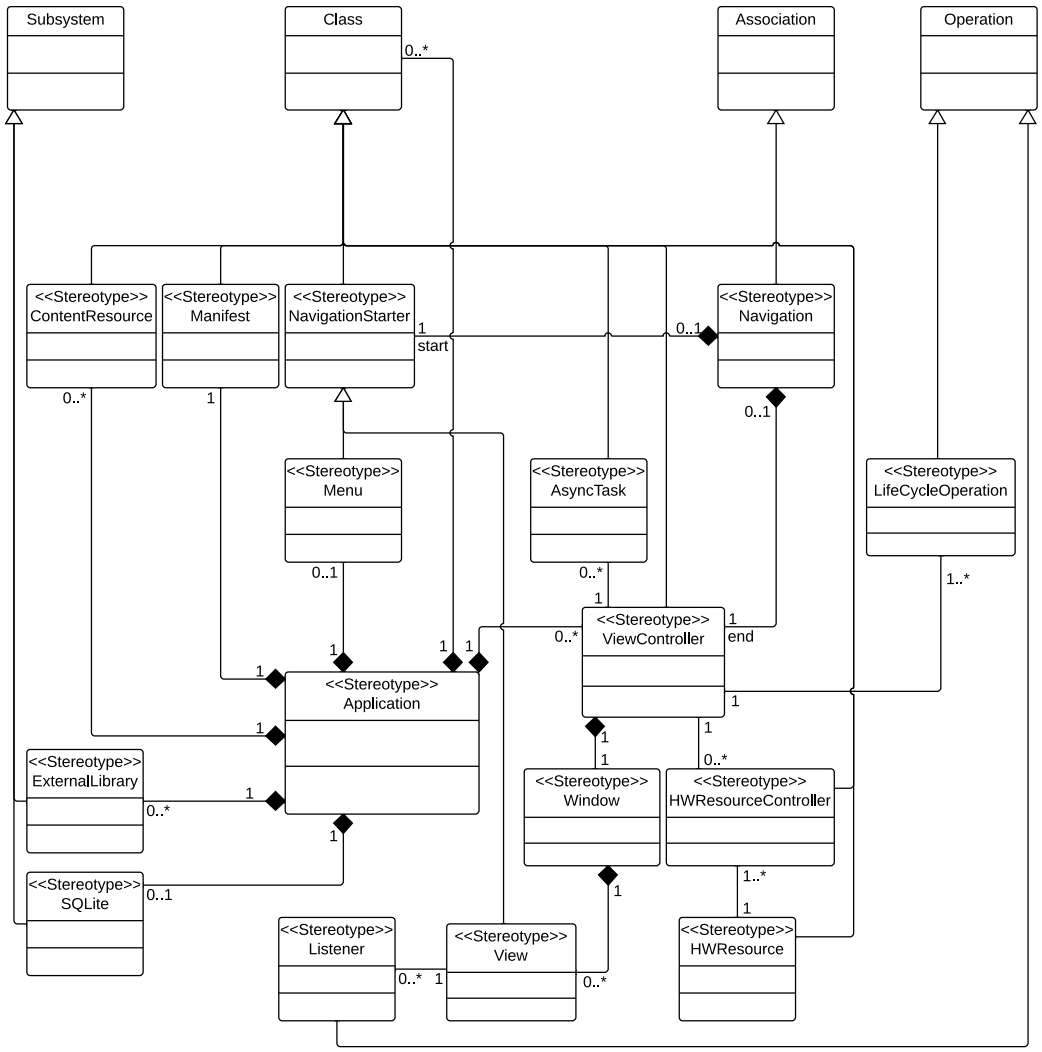

Figura 3.2: *Metamodello Astratto*: parte nuova

nuovi non presentano l'etichetta Stereotype. Il diagramma è stato definito evitando di lasciare elementi isolati, perciò ciascun elemento è relazionato ad un concetto UML esistente o ad un elemento della parte nuova. Il fulcro del metamodello è rappresentato dallo stereotipo **Application**, da cui partono diverse composizioni. Abbiamo scelto di lasciare questo elemento privo di generalizzazioni in quanto lo consideriamo l'elemento cardine del metamodello. Osservando la Figura 3.2 è immediato individuare le composizioni che partono dall'oggetto Application, che risulta essere così composto dai seguenti elementi:

- **Class**: elemento appartenente alla parte esistente del metamodello e che rappresenta per Application la possibilità di includere delle generiche classi di supporto definite dal programmatore in fase di implementazione. Rientrano in questa categoria le classi per interfacciarsi con il database, le classi di supporto contenenti costanti e metodi statici pubblici accessibili dalle altre classi dell'applicazione, etc.
  La relazione tra Application e Class è una composizione con cardinalità da zero a molti verso Class, che bisogna intendere come la possibilità per Application di avere da zero a molte classi di supporto, al contrario la cardinalità pari a uno verso Application indica che ogni istanza di Class appartiene ad un'unica applicazione.

- **Manifest**: in Android si tratta di un file che permette di specificare tutte le Activity presenti nell'applicazione, i permessi e i temi che si intendono utilizzare. In ambiente iOS questo concetto si concretizza come l'insieme dei file di configurazione autogenerati dal tool Xcode che servono a definire la Window principale, la categoria dell'applicazione e in generale a gestire la compatibilità sui device di casa Apple. Nessuna applicazione può funzionare se non dispone di queste informazioni, per questo sulla relazione di composizione che lega i due elementi la cardinalità è uno in entrambi i sensi: l'istanza dell'elemento Manifest non può che avere una sola applicazione di riferimento, mentre ogni applicazione ha un unico Manifest, rappresentato nel metamodello come concetto generico, ovvero figlio dell'elemento esistente Class.

- **ContentResource**: è un elemento le cui istanze possono essere risorse di varia natura (immagini, video, file di testo, file audio, etc.) salvate e utilizzate all'interno dell'applicazione. Anch'esso è legato con una generalizzazione all'elemento Class, mentre le cardinalità sulla relazione di composizione con Application rivelano che ogni applicazione può contenere da zero a molte risorse, mentre ogni risorsa risiede ed è accessibile da un'unica applicazione.

- **ExternalLibrary**: ci si riferisce ad una specifica libreria esterna di sistema, necessaria per il corretto funzionamento dell'applicazione. L'indipendenza dalla specifica applicazione si può intuire dal fatto che il concetto è introdotto come figlio di Subsystem, e non di Class. Le librerie esterne vengono considerate dei sottosistemi indipendenti, in quanto non sono altro che progetti a sé stanti, i cui sorgenti vengono sfruttati per soddisfare le funzionalità richieste dall'applicazione.
  Osservando quindi le cardinalità sulla composizione che lega i due concetti di ExternalLibrary e Application, si può asserire che ogni appli-

cazione può utilizzare da zero a molte librerie esterne, le quali devono essere importate nei sorgenti dell'applicazione. Per questo motivo la relazione inversa indica che ogni libreria esterna è utilizzata da un'unica applicazione.

- **SQLite**: così come ExternalLibrary anche questo elemento è rappresentato con una generalizzazione verso l'oggetto Subsystem e viene interpretato come l'insieme di tutte le librerie e gli oggetti necessari alla costruzione di un funzionale database interno al dispositivo sul quale l'applicazione verrà installata. SQLite potrebbe essere considerato figlio dell'elemento più generico ExternalLibrary, ma la scelta di collegare la generalizzazione con Subsystem ha l'obiettivo di mettere in maggior risalto il concetto generico di SQLite, inteso come strumento che permette di realizzare una base di dati incorporata in un unico file e che risulta compatibile con diverse piattaforme mobili, tra cui Android e iOS.
  La relazione di composizione ha cardinalità 0..1 verso l'elemento SQLite a significare che l'applicazione può opzionalmente incorporare un database SQLite, mentre dall'altro capo la cardinalità è uno in quanto ogni database è legato alla specifica applicazione che lo gestisce.

- **Menu**: è inteso come menu principale dell'applicazione e, in quanto tale, è definito come figlio del concetto astratto **NavigationStarter**, da intendersi come generico elemento che può costituire il punto di partenza di una navigazione all'interno dell'applicazione.

- **ViewController**: si tratta di un elemento di fondamentale importanza nelle applicazioni mobili, e in particolare rappresenta le Activity in ambiente Android e i ViewController in ambiente iOS. Nel pattern MVC riveste i ruoli di View e Controller allo stesso tempo, in quanto costituisce l'oggetto alla base della gestione dei contenuti dei componenti mostrati sull'interfaccia grafica. Nel metamodello ViewController è indicato come figlio di Class, perciò le istanze di ViewController possono comprendere attributi e metodi. Considerando le cardinalità della relazione di composizione tra Application e ViewController si evince che ogni ViewController appartiene ad un'unica applicazione, mentre ogni applicazione può avere molti ViewController o addirittura nessuno, sebbene in quel caso si tratterebbe di un'applicazione priva di interfaccia grafica e quindi di utilità pratica.

Collegati al concetto di ViewController vi sono poi diversi altri elementi, infatti esso può contenere delle istanze di LifeCycleOperation, AsyncTask e

Navigation e può avere accesso ad elementi grafici e risorse hardware.

Per **LifeCycleOperation** si intende un generico metodo per la gestione del ciclo di vita di un ViewController dell'applicazione. In quanto metodo lo stereotipo LifeCycleOperation è figlio dell'elemento Operation. In Figura 3.3 è mostrato un confronto diretto tra i metodi maggiormente utilizzati per la gestione del ciclo di vita delle Activity in Android e dei ViewController in iOS, dove emergono alcune corrispondenze.

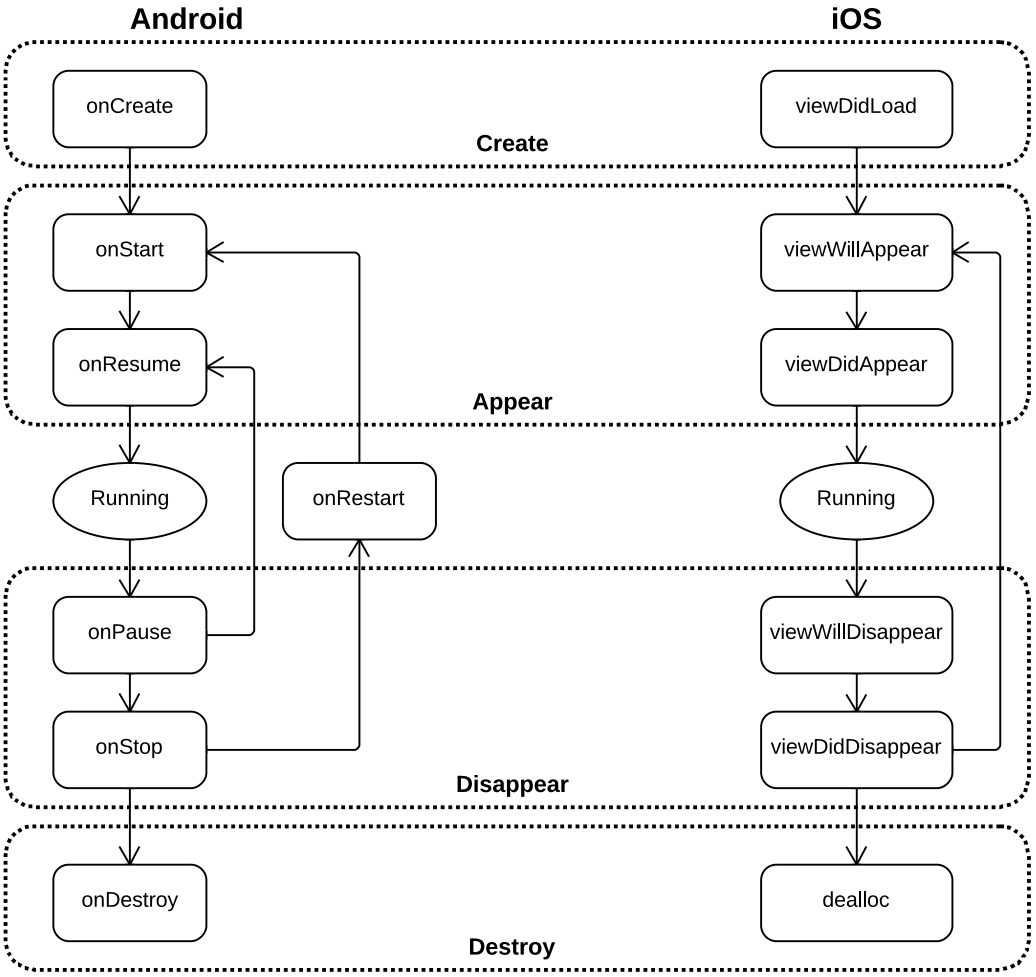

Figura 3.3: *LifeCycle*: Activity Android e ViewController iOS

Dall'analisi a monte della modellazione sono stati introdotti i seguenti concetti, presenti nel metamodello come figli concreti dell'elemento astratto LifeCycleOperation e mostrati in Figura 3.4.

- **Create**: fa riferimento alla onCreate in Android e alla viewDidLoad in iOS, ovvero al primo metodo invocato sul ViewController al momento della sua creazione.

- **Appear**: fa riferimento alla onStart e alla onResume in Android e alla viewWillAppear e alla viewDidAppear in iOS, invocati in prossimità dell'apparizione dell'interfaccia grafica del relativo ViewController sullo schermo.

- **Disappear**: fa riferimento alla onPause e alla onStop in Android e alla viewWillDisappear e alla viewDidDisappear in iOS, invocati al momento della scomparsa dell'interfaccia grafica del relativo ViewController dallo schermo.

- **Destroy**: fa riferimento alla onDestroy in Android e alla dealloc in iOS, ovvero al metodo invocato sul ViewController al momento della suo cancellazione dalla memoria del dispositivo.

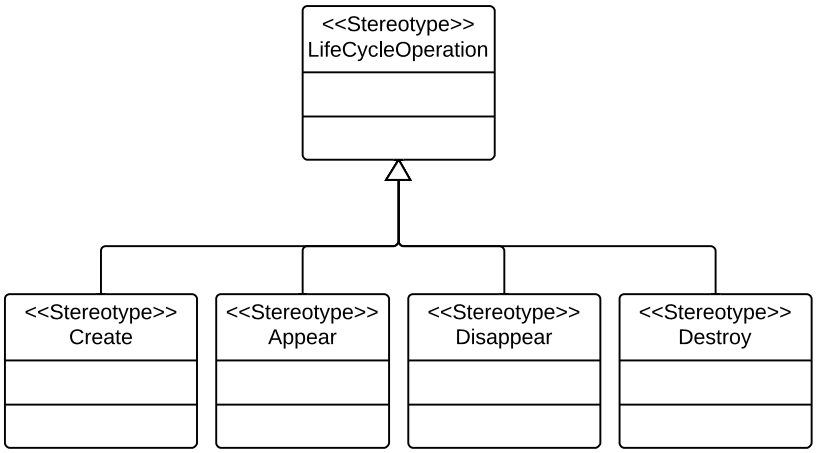

Figura 3.4: *Metamodello Astratto*: parte nuova - metodi del LifeCycle

L'associazione che lega ViewController a LifeCycleOperation ha cardinalità uno verso il ViewController, ad indicare che ogni metodo del ciclo di vita è implementato all'interno del relativo ViewController, mentre da uno a molti nell'altro senso, in quanto ogni ViewController può contenere diversi metodi di gestione del ciclo di vita, e in particolare deve necessariamente contenere l'implementazione del metodo Create.

Entrambe le piattaforme Android e iOS offrono la possibilità di eseguire

blocchi di codice in maniera asincrona rispetto al task principale di esecuzione, perciò è stato introdotto l'elemento **AsyncTask**. L'associazione che lega ViewController a AsyncTask mostra la possibilità per un ViewController di istanziare e quindi utilizzare zero o più task asincroni.

L'ultimo elemento che rimane da analizzare prima di affrontare la parte relativa all'interfaccia grafica è **Navigation**. Con questo concetto vogliamo modellare la possibilità di passare da un ViewController ad un'altro all'interno dell'applicazione. Ogni oggetto Navigation è composto quindi da un punto di partenza e un punto di arrivo e per questo motivo nel diagramma sono indicate due relazioni di composizione. La prima, contraddistinta dal nome *start*, è diretta verso l'elemento NavigationStarter, da cui la navigazione ha origine, invece la seconda, contraddistinta dall'etichetta *end*, è diretta verso un nuovo ViewController. Considerando le generalizzazioni sull'elemento NavigationStarter la navigazione ha origine nel momento in cui viene cliccata una voce del menu principale oppure quando si verificano determinati eventi su qualcuno degli elementi dell'interfaccia grafica.
Abbiamo definito Navigation come elemento figlio di Association proprio perché una navigazione all'interno dell'applicazione rappresenta un'associazione tra due componenti interne.

3.2.1 Interfaccia grafica

I ViewController dell'applicazione sono costituiti da una **Window**, a sua volta composta da zero a molte View, ciascuna delle quali rappresenta uno specifico componente grafico mostrato sullo schermo. Una Window priva di View non è altro che un semplice ViewController privo di interfaccia grafica. Come si evince dal metamodello, View, in quanto figlio di NavigationStarter, ha una relazione col concetto di Navigation, perché da una View può essere avviata una navigazione verso un nuovo ViewController al verificarsi di un determinato evento. Abitualmente gli eventi si verificano a causa dell'interazione dell'utente col dispositivo (la pressione di un bottone, l'identificazione di una particolare gesture, l'immissione di un testo, etc), ma possono essere scatenati anche dal sistema stesso. I metodi che catturano gli eventi sono chiamati **Listener** e in quanto tali sono figli di Operation. Ogni View può avere diversi Listener associati, uno per ciascuno degli eventi scatenabili che si intende monitorare.

I principali elementi che caratterizzano l'interfaccia grafica sono stati indicati esplicitamente nel metamodello come figli dell'elemento astratto View. Tali elementi, che trovano spazio all'interno delle Window dei vari ViewController,

sono descritti di seguito e rappresentati in Figura 3.5.

- **Button**: è un bottone che può essere premuto dall'utente per compiere una determinata azione. Solitamente al bottone è associato il Listener per catturare il click scatenato dall'utente.

- **ImageView**: mostra un'immagine o un'icona caricata dalle risorse, scaricata da Internet o estratta dalla memoria del dispositivo.

- **VideoView**: mostra un video caricato dalle risorse, scaricato da Internet o estratto dalla memoria del dispositivo.

- **TextView**: è un'area di testo non editabile.

- **EditText**: è un'area di testo il cui contenuto può essere modificato dall'utente.

- **ListView**: rappresenta una lista di elementi con scorrimento verticale.

- **GridView**: rappresenta una collezione di elementi disposti su una griglia con scorrimento bidimensionale.

- **WebView**: permette di visualizzare una pagina web all'interno del ViewController, per cui i suoi contenuti e il layout sono definiti attraverso file HTML e CSS estratti dalle risorse.

- **MapView**: mostra una mappa geografica con eventuali marker o percorsi indicati.

- **AlertDialog**: è un messaggio che viene mostrato in sovraimpressione sullo schermo del dispositivo e richiede tipicamente la scelta di una o più opzioni all'utente.

- **ProgressDialog**: è un messaggio simile all'AlertDialog ma con la differenza di essere utilizzato per informare l'utente che l'applicazione sta eseguendo delle operazioni che richiedono del tempo e di cui è necessario attenderne il completamento. Può essere la tipica rotellina (spinner) o una barra orizzontale che mostra la percentuale di completamento dell'operazione.

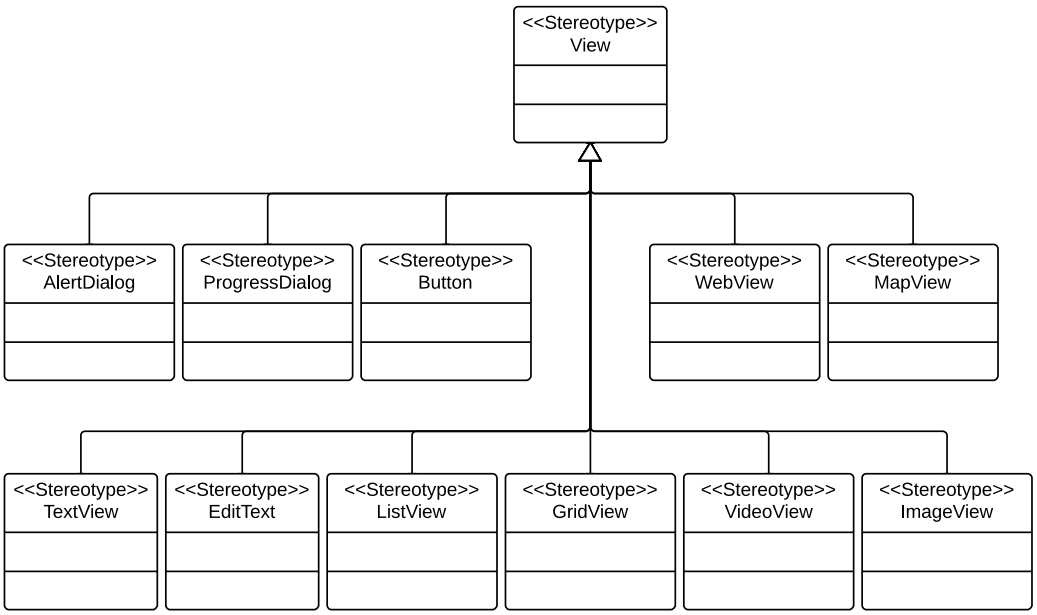

Figura 3.5: *Metamodello Astratto*: parte nuova - componenti grafici

3.2.2 Hardware

Nel capitolo precedente abbiamo visto che uno dei punti di forza delle applicazioni sviluppate in linguaggio nativo è la possibilità di utilizzare tutte le API dei componenti hardware del dispositivo sul quale vengono installate. Quindi, all'interno di un metamodello che si propone di presentare i componenti principali che strutturano le applicazioni mobili, non può mancare una parte dedicata all'accesso e alla gestione delle risorse hardware. I componenti di questa parte sono illustrati sia in Figura 3.2 che in Figura 3.6, dove sono rappresentati i figli dell'elemento astratto HWResource.

L'analisi di questa sezione può partire dall'associazione che lega ViewController e **HWResourceController**. HWResourceController, nel metamodello definito come generica classe, è da intendersi come un elemento che consente la gestione dell'interfacciamento con una particolare risorsa hardware del dispositivo, perciò assume il ruolo di mediatore tra lo specifico ViewController che richiede l'accesso al componente hardware e il componente stesso. I ViewController possono contenere zero o più istanze di HWResourceController, a seconda che richiedano o meno l'interfacciamento con specifici componenti hardware.

Le risorse hardware sono invece rappresentate dal concetto **HWResource**,

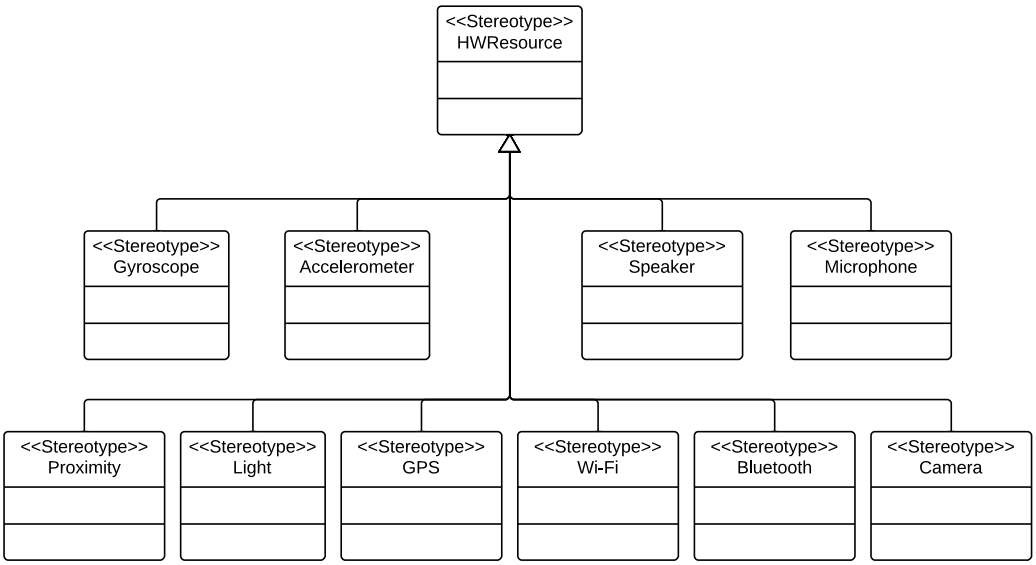

Figura 3.6: *Metamodello Astratto*: parte nuova - risorse hardware

anch'esso definito come figlio di Class. E' interessante notare le cardinalità dell'associazione che lega questo elemento con HWResourceController: una risorsa hardware deve avere almeno un'istanza di HWResourceController che ne gestisce l'interfacciamento, ma in generale ne ha anche più di una, mentre un HWResourceController fa riferimento ad un unico componente hardware. Di seguito vengono presentati in modo sintetico gli elementi figli del concetto astratto HWResource, che costituiscono le principali risorse hardware del dispositivo sfruttabili dalle applicazioni.

- **Accelerometer**: sensore in grado di misurare l'accelerazione applicata al dispositivo, inclusa quella determinata dalla forza di gravità. Risulta utile a percepire lo spostamento e in genere viene utilizzato allo scopo di cambiare automaticamente l'orientamento dell'interfaccia grafica sullo schermo da landscape a portrait e viceversa, a seconda dell'inclinazione del dispositivo.

- **Gyroscope**: sensore in grado di identificare la posizione e la velocità angolare del dispositivo rispetto a uno degli assi x, y e z del dispositivo. Insieme all'accelerometro e al GPS è usato spesso per individuare più facilmente la posizione del dispositivo.

- **GPS**: sensore del dispositivo che riceve dal satellite le informazioni relative alle coordinate geografiche. In genere è utilizzato congiuntamente alle mappe.

- **Wi-Fi**: sensore che consente al dispositivo di accedere alla rete Internet attraverso una connessione a un router Wi-Fi.
- **Bluetooth**: sensore che permette lo scambio di dati tra dispositivi dislocati nel raggio di pochi metri.
- **Light**: sensore che rileva la quantità di luce ambientale e che viene utilizzato per regolare la luminosità dello schermo.
- **Speaker**: componente finalizzato all'emissione dei suoni.
- **Camera**: componente utilizzato per scattare le foto e registrare i video. Può essere più di uno in uno stesso dispositivo.
- **Proximity**: sensore in grado di rilevare la presenza di oggetti nelle immediate vicinanze del dispositivo. Negli smartphone con schermo touchscreen il sensore viene utilizzato generalmente durante le telefonate: al momento del rilevamento di una certa vicinanza del volto al dispositivo viene disattivato il touchscreen, evitando così il possibile tocco involontario dei tasti sullo schermo. Il touchscreen viene riattivato all'avvenuto distacco del volto dal device.

3.3 Modello dell'applicazione Valtellina

Per sondare la possibilità di utilizzare il metamodello come strumento per modellare una generica applicazione per dispositivi mobili, indipendentemente dalla piattaforma per cui viene progettata, abbiamo provato a generare una sua istanza. Tale istanza costituisce il modello di un'applicazione di quelle che proveremo a generare attraverso il generatore di codice, che costituisce l'output della fase di implementazione del nostro lavoro e che verrà quindi presentato più avanti. L'applicazione che intendiamo modellare si chiama **Valtellina** e permette all'utente finale di accedere a una serie di informazioni e foto relative alle più rinomate località turistiche della zona.

Essa è composta da quattro ViewController, dove il primo ad essere mostrato all'apertura dell'applicazione corrisponde a quello che presenta una descrizione generica del territorio, con una foto che riporta il marchio del territorio e un bottone che permette di accedere a un secondo ViewController, contenente un video caratteristico. L'applicazione presenta inoltre un menu che permette di accedere al primo ViewController e a un terzo che contiene una lista delle più frequentate località turistiche. A partire da quest'ultimo l'utente ha la possibilità di selezionare una voce dall'elenco e visualizzare un nuovo ViewController contenente i dettagli di quella specifica località, e in

particolare una descrizione generale e una foto.

In Figura 3.7 è mostrato il modello dell'applicazione. Partendo dall'alto possiamo vedere come l'applicazione sia composta da un Manifest, che nella versione Android corrisponderà ad un file XML, e da un certo numero di risorse interne, che per ragioni di spazio abbiamo preferito non specificare direttamente nel modello, ma che concettualmente corrispondono a tutte le immagini, al video e ai file di testo relativi alle località turistiche.
Osservando le altre composizioni su Application è possibile notare i quattro ViewController che abbiamo presentato poco fa: *Description*, che costituisce il ViewController iniziale, *Video*, che contiene il video, *TurismLocationsList*, composto da una lista contenente alcune località turistiche della zona, e infine *TurismLocationDetail*, che mostra i dettagli di una certa località turistica. L'ultima composizione che rimane da segnalare è quella con l'elemento Menu, che costituisce un NavigationStarter, per cui può presentare una o più relazioni con l'elemento Navigation. Nel nostro caso, a partire dal Menu dell'applicazione, è evidente nel modello l'intenzione di consentire all'utente finale di raggiungere i ViewController Description e TurismLocationsList.

Per capire come sono strutturati i diversi ViewController e come sono legati tra loro partiamo con ordine, analizzando il primo che viene mostrato all'apertura dell'applicazione. Il ViewController **Description** è composto, a livello concettuale, da una Window contenente gli elementi dell'interfaccia grafica, che in questo caso sono tre. Il primo indicato in figura è una ImageView chiamata *logoImageView*, la quale è stata modellata con lo scopo di mostrare in un'immagine il marchio della Valtellina. Il secondo elemento, chiamato *descriptionWebView*, è una WebView contenente la descrizione generica del territorio: abbiamo scelto di utilizzare una WebView, in alternativa ad una TextView, per mostrare come è possibile sfruttare le tecnologie del web, nel nostro caso HTML e CSS, per definire in maniera semplice e completa, evitando di utilizzare un particolare editor grafico, il contenuto e il layout di un ViewController. L'ultimo elemento grafico, *videoButton*, è invece il bottone che permette di avviare una navigazione statica verso il ViewController **Video**, la cui Window non contiene altro che una VideoView per la riproduzione del video.

A partire dal menu è poi possibile raggiungere il ViewController **TurismLocationsList**, composto da una Window contenente una ListView, chiamata *turismLocationsListView*. La ListView ha una relazione con un Listener, ad indicare che vogliamo fare in modo che l'evento di click di un particolare elemento interno alla lista venga monitorato. Al momento del click partirà una

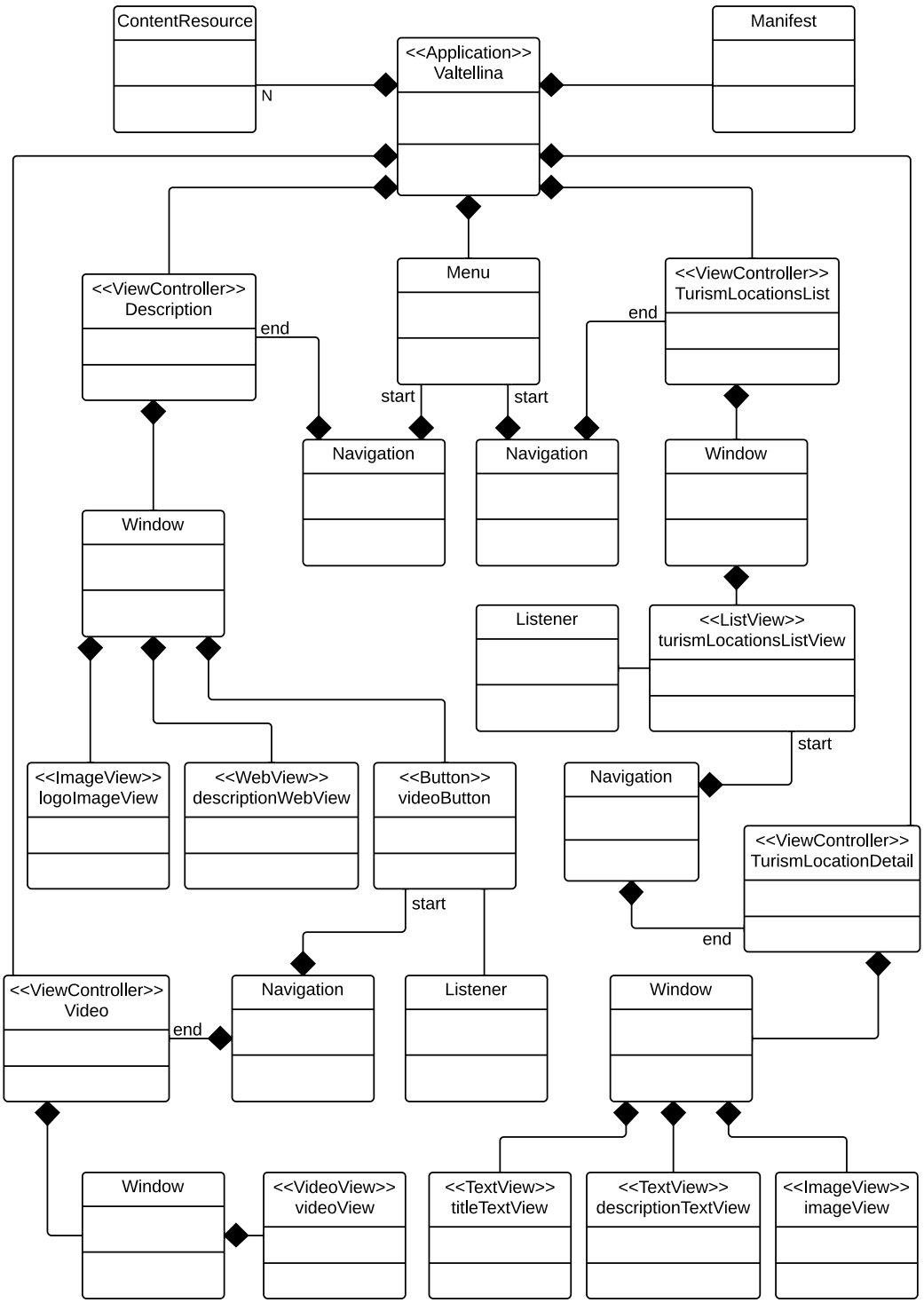

Figura 3.7: *Valtellina*: istanza del metamodello

navigazione verso il ViewController TurismLocationDetail, per questo motivo è stata indicata una relazione con l'elemento Navigation che ha come punto di arrivo il ViewController contenente i dettagli della località turistica. A differenza di quelle presentate in precedenza questa navigazione è dinamica, in quanto il contenuto del ViewController destinazione è dipendente dall'elemento che è stato cliccato nella ListView interna a TurismLocationsList. Come vedremo più avanti, le navigazioni dinamiche richiedono allo sviluppatore di implementare via codice, modificando i sorgenti dell'applicazione generata, la logica applicativa che determina il passaggio delle informazioni necessarie dal ViewController di partenza al ViewController di destinazione, sebbene nella maggior parte dei casi richiede un effort minimo. La Window dell'ultimo ViewController, **TurismLocationDetail**, è composta da tre elementi grafici. I primi due sono semplici TextView, chiamate *titleTextView* e *descriptionTextView*, finalizzate a contenere rispettivamente il nome e un testo descrittivo della località turistica che è stata selezionata dall'utente nel ViewController precedente. Il terzo e ultimo elemento grafico è una ImageView contenente una foto della località turistica.

Il modello presentato in questo paragrafo costituisce un'istanza del metamodello presentato: quello che possiamo affermare è che, per quanto chiaro possa apparire, esso non è completo in termini di informazioni richieste per determinare i contenuti dell'applicazione che si intende generare. Per citare un esempio, il modello presentato non permette di indicare esplicitamente le informazioni da inserire nelle TextView, o quali siano i file locali contenenti le immagini che devono essere mostrate nelle ImageView. Tutto ciò è conseguenza del fatto che quello che manca principalmente nel metamodello presentato è la definizione di attributi all'interno degli elementi che lo compongono.

3.4 Conclusioni

In questo capitolo abbiamo visto una prima versione, ad alto livello, del metamodello per le applicazioni mobili e abbiamo provato a definire una sua istanza. Gli oggetti riportati in questa versione sono stati descritti sinteticamente, per permettere al lettore di comprendere i concetti che stanno alla base della struttura delle applicazioni mobili, indipendentemente dalla piattaforma per la quale vengono progettate.
Come abbiamo avuto modo di constatare attraverso il modello dell'applicazione Valtellina, le istanze del metamodello astratto si presentano tuttavia come modelli non sufficientemente espressivi da essere tradotti in codice. In altre parole, per procedere con la fase di generazione del codice, ovvero

traduzione di una certa istanza del metamodello, definita dall'utente, nei corrispondenti sorgenti nativi dell'applicazione, risulta necessario partire da una versione più concreta del metamodello. Per questo motivo nel capitolo seguente presenteremo una seconda versione, in cui molti dei concetti modellati in questo capitolo verranno ripresi, analizzati nel dettaglio e riadattati laddove necessario, con lo scopo di rendere più completo e dettagliato l'output della fase di modellazione dell'applicazione desiderata, in modo da rendere fattibile la traduzione del modello generato nei sorgenti nativi corrispondenti.

Capitolo 4

Metamodello implementativo

Il metamodello astratto, descritto nel capitolo precedente, è stato progettato in modo da permettere una prima definizione, ad alto livello, del modello dell'applicazione. In questo capitolo presentiamo il metamodello implementativo, risultato di una rivisitazione del metamodello astratto. Quest'ultimo è stato manipolato e rivisto per produrre una versione del metamodello idonea a guidare lo sviluppatore nella fase di definizione del modello dell'applicazione desiderata, a rendere più dettagliate le istanze generabili e ad avere così un concreto punto di partenza per il processo di traduzione in codice del modello dell'applicazione. In particolare nel metamodello implementativo abbiamo riportato la maggior parte degli elementi concreti presenti nel metamodello astratto e ne abbiamo introdotti dei nuovi. A ciascun elemento abbiamo aggiunto gli attributi necessari per il processo di traduzione.
Così come per il metamodello astratto, la versione implementativa è stata realizzata analizzando nello specifico la struttura delle applicazioni Android e iOS, ma, anche in questo caso, l'essersi basati su due piattaforme non preclude la possibilità di considerare valido il metamodello per qualsiasi altra piattaforma. Entrambi i metamodelli sono stati definiti in modo da permettere allo sviluppatore di produrre una sola istanza per ogni applicazione indipendentemente dalle piattaforme per cui essa viene sviluppata.

4.1 Elementi modellati

In questa sezione saranno analizzati nel dettaglio tutti gli elementi del metamodello astratto che sono stati riportati nel metamodello implementativo, ovvero quelli finalizzati ad essere mappati direttamente in codice nei sorgenti dell'applicazione da generare. Partendo dall'elemento radice del metamodello, ovvero l'elemento Application, arriveremo a spiegare tutti i concetti introdotti. In Figura 4.1 è rappresentato l'intero metamodello implementativo, completo degli attributi specifici di ogni elemento. Osservando le

associazioni si nota che sono state utilizzate solo composizioni: il motivo di questa scelta non è concettuale, ma strettamente connesso agli strumenti di modellazione che abbiamo sfruttato nell'implementazione del generatore di codice, che saranno presentati nel capitolo successivo.

Application

Ogni applicazione per dispositivi mobili possiede un nome, corrispondente a quello mostrato insieme all'icona nel menu delle applicazioni di un qualsiasi sistema operativo. Tutte le applicazioni pubblicate in rete presentano inoltre un id, che corrisponde a quello mostrato sull'App Store e che permette di identificare univocamente l'applicazione. Nell'Apple Store tale codice identificativo è chiamato iOS Identifier Name e corrisponde al Bundle Identifier indicato nel tool di sviluppo Xcode, mentre nel Play Store è chiamato Android Package Name e corrisponde al Package Name indicato nel file Manifest.xml. In entrambi i casi esso è stabilito dallo sviluppatore seguendo le convenzioni dei package Java e dovrebbe essere il nome dell'applicazione concatenato al dominio dell'organizzazione riportato in ordine inverso. Per questo motivo è stato aggiunto, oltre al nome dell'applicazione, l'identificativo dell'azienda sviluppatrice tra gli attributi di Application e, in fase di generazione del codice, abbiamo definito l'identificativo dell'applicazione concatenando il nome dell'applicazione a questo id.
Gli attributi di Application sono:

- *Name*: è il nome dell'applicazione.

- *CompanyIdentifier*: é l'identificativo dell'organizzazione sviluppatrice e, insieme a Name, contribuisce alla definizione dell'identificativo dell'applicazione.

Oltre agli attributi appena descritti, Application ha due relazioni:

- *ViewControllers*: è una relazione da zero a molti, essenziale per l'inserimento di un numero indefinito di ViewController nell'applicazione.

- *Menu*: è la relazione utilizzata dall'utente per indicare la volontà di aggiungere un Menu nell'applicazione. Coerentemente con quanto affermato nella versione astratta del metamodello, la cardinalità della seconda relazione prevede di avere al massimo un menu per ogni applicazione.

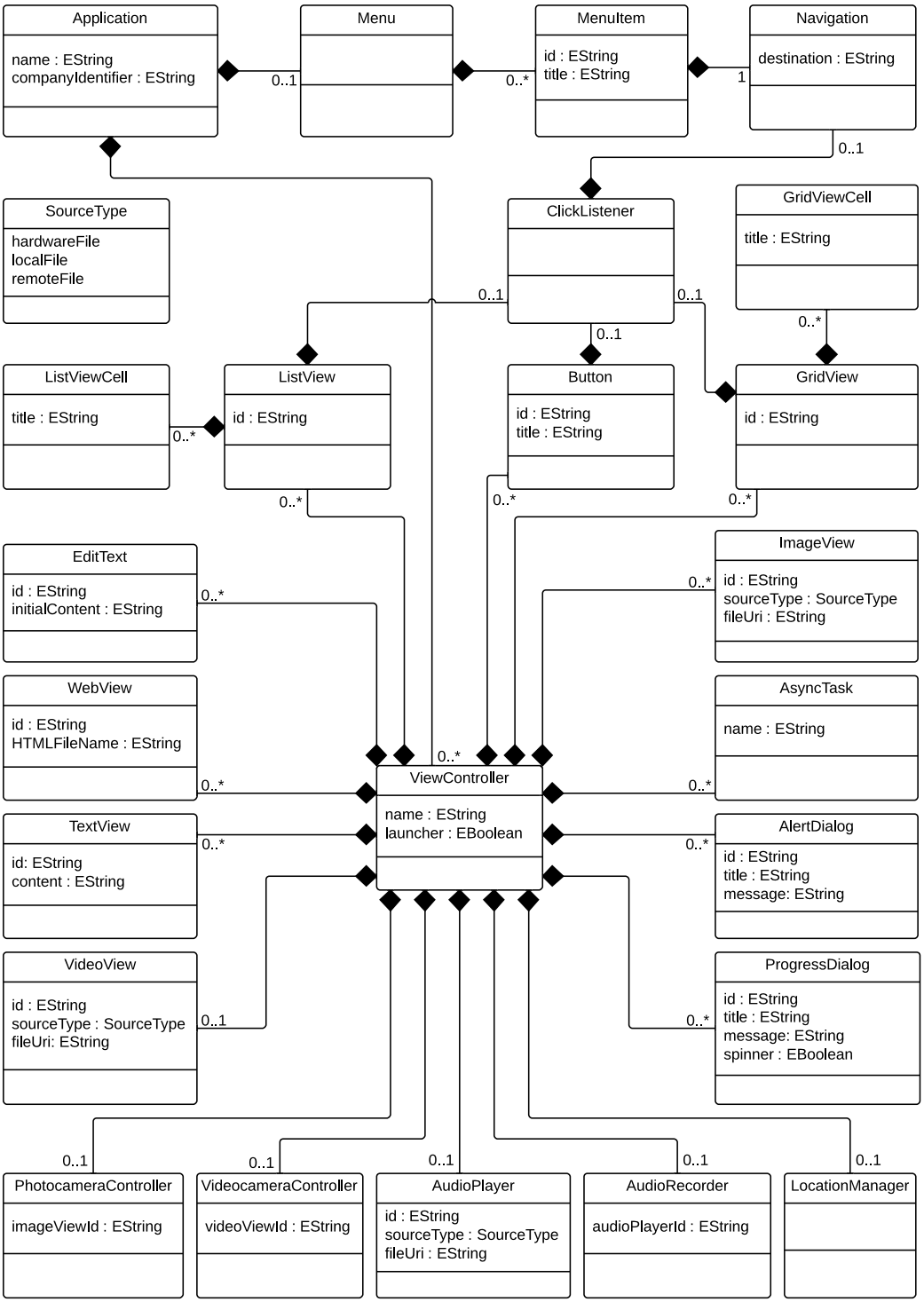

Figura 4.1: *Metamodello Implementativo*

ViewController

Nel metamodello implementativo ViewController è collegato direttamente ad Application attraverso una relazione, la quale indica che Application può contenere diversi ViewController.
L'elemento ViewController corrisponde a una classe, quindi ha un nome univoco che lo identifica nei sorgenti. Il nome viene utilizzato all'interno dell'applicazione per determinare i punti di arrivo delle navigazioni che hanno come origine un componente dell'interfaccia grafica o una voce del menu. Inoltre in ogni applicazione è necessario indicare il ViewController iniziale tra quelli presenti, ovvero quello che viene mostrato all'avvio dell'applicazione, perciò è stato aggiunto un attributo booleano tra quelli di ViewController.
Un ViewController può mostrare un'interfaccia utente composta da diversi componenti grafici e può avere accesso alle risorse hardware grazie ai rispettivi controller. Per questo motivo sono state definite diverse relazioni su ViewController che rendono immediata e intuitiva l'aggiunta di tutti questi componenti nell'istanza del metamodello. Le cardinalità delle relazioni sono state definite sulla base del numero di istanze di un particolare elemento generabili all'interno di uno stesso ViewController. Per gli elementi che possono presentare più istanze all'interno dello stesso ViewController sono state definite relazioni con cardinalità da zero a molti, mentre per quelli che presentano al più un'istanza sono state definite relazioni con cardinalità da zero a uno.
Gli attributi di ViewController sono:

- *Name*: è il nome del ViewController.
- *Launcher*: è l'attributo booleano che indica se il ViewController è il primo ad apparire all'avvio dell'applicazione.

Le relazioni di ViewController sono:

- *Application*: è la relazione inversa di viewControllers su Application.
- *Buttons*: permette l'aggiunta dei Button. La sua cardinalità va da zero a molti.
- *TextViews*: permette l'aggiunta delle TextView. La sua cardinalità va da zero a molti.
- *WebViews*: permette l'aggiunta delle WebView. La sua cardinalità va da zero a molti.
- *ImageViews*: permette l'aggiunta delle ImageView. La sua cardinalità va da zero a molti.

- *ListViews*: permette l'aggiunta delle ListView. La sua cardinalità va da zero a molti, anche se raramente si trovano più ListView all'interno di uno stesso ViewController.

- *GridViews*: permette l'aggiunta delle GridView. La sua cardinalità va da zero a molti ma, come per le ListView, difficilmente se ne trovano più di una in uno stesso ViewController.

- *EditTexts*: permette l'aggiunta delle EditText. La sua cardinalità va da zero a molti.

- *AsyncTasks*: permette l'aggiunta di AsyncTask. La sua cardinalità va da zero a molti.

- *ProgressDialogs*: permette l'aggiunta di ProgressDialog. La sua cardinalità va da zero a molti.

- *AlertDialogs*: permette l'aggiunta di AlertDialog. La sua cardinalità va da zero a molti.

- *LocationManager*: permette l'inserimento di un LocationManager, ovvero di una serie di metodi di gestione della posizione geografica dell'utente. La sua cardinalità va da zero a uno, in quanto ogni ViewController può o meno necessitare delle coordinate geografiche dell'utente.

- *VideoView*: permette l'aggiunta di una VideoView. La sua cardinalità va da zero a uno, in quanto abbiamo ritenuto estremamente raro e complicato da un punto di vista implementativo l'inserimento di più video all'interno di uno stesso ViewController.

- *PhotocameraController*: permette l'inserimento di un gestore delle risorse provenienti dalla fotocamera del dispositivo, ovvero delle fotografie. La cardinalità è quindi da zero ad uno.

- *VideocameraController*: permette l'inserimento di un gestore delle risorse provenienti dalla videocamera del dispositivo, quindi delle registrazioni video. Come nel caso del controller della fotocamera, la cardinalità va da zero a uno.

- *AudioPlayer*: permette l'inserimento di un riproduttore di file audio. La sua cardinalità va da zero a uno, in quanto abbiamo ritenuto raro l'inserimento di più AudioPlayer all'interno di uno stesso ViewController.

- *AudioRecorder*: permette l'inserimento di un registratore di file audio, vale a dire del controller del microfono del dispositivo. La cardinalità della relazione va da zero a uno in quanto abbiamo ritenuto improbabile e complicato da un punto di vista implementativo l'inserimento di più registratori audio all'interno di un ViewController.

Menu

Il menu di un'applicazione è un componente la cui veste grafica varia in base alla piattaforma per cui l'applicazione viene sviluppata. In Android il menu standard prende il nome di Option Menu ed è accessibile premendo l'apposito tasto sul dispositivo, mentre in iOS il menu prende il nome di Tab Bar e corrisponde a una barra in basso allo schermo composta da un titolo ed eventualmente un'icona per ciascuna voce del menu.
L'elemento Menu è al più uno all'interno di una stessa applicazione e non richiede particolari attributi. Dobbiamo però aggiungere una relazione con l'elemento MenuItem, che rappresenta una singola voce del menu. Tale elemento è stato introdotto nel metamodello implementativo e verrà presentato tra poco.
L'elemento Menu ha due relazioni:

- *MenuItems*: consente di inserire un numero indefinito di voci nel menu, per cui la sua cardinalità va da zero a molti. Un menu vuoto è privo di utilità pratica, ma la cardinalità inferiore è stata pensata in modo da permettere all'utente di avere già implementati nel codice generato tutti gli strumenti per la predisposizione del menu e poter ultimare rapidamente l'implementazione con l'aggiunta delle voci necessarie.

- *Application*: è la relazione opposta di menu su Application.

MenuItem

L'elemento MenuItem rappresenta una voce del menu dell'applicazione ed è composto da un titolo e da un'icona opzionale. Per permettere all'utente di individuare immediatamente la voce del menu nel codice generato aggiungiamo tra gli attributi di MenuItem il titolo e un id.
In quanto figlio del concetto astratto NavigationStarter, MenuItem corrisponde a uno tra gli elementi che azionano le navigazioni all'interno dell'applicazione, per questo nel metamodello implementativo introduciamo una relazione diretta con l'elemento Navigation che, come vedremo, consente di definire il ViewController di destinazione.
L'elemento MenuItem presenta, quindi, due attributi e due relazioni. Gli attributi sono:

- *Id*: è l'identificativo della voce del menu.
- *Title*: è il titolo della voce del menu, che dovrebbe essere indicativo della sezione dell'applicazione mostrata al momento del click da parte dell'utente dello specifico MenuItem.

Le relazioni di MenuItem sono:

- *Navigation*: collega il MenuItem al ViewController destinazione della navigazione.
- *Menu*: è la relazione inversa di menuItems su Menu.

ClickListener

L'elemento ClickListener è un particolare Listener, definito nel metamodello astratto, che rappresenta il metodo invocato al momento del click dell'elemento che sta monitorando. In particolare nel nostro metamodello implementativo, che cerca di modellare nel modo più concreto possibile la struttura delle applicazioni mobili, gli elementi che presentano una relazione con questo tipo di oggetto sono Button, ListView e GridView, i quali sono stati progettati per scatenare un'azione al momento del click da parte dell'utente.
Abbiamo scelto di modellare il ClickListener poiché corrisponde al Listener maggiormente utilizzato nello sviluppo delle applicazioni mobili. Tra gli altri più utilizzati e che abbiamo scelto di non modellare ricordiamo quello in grado di riconoscere un click prolungato e quello utilizzato per rilevare doppi click.
L'elemento ClickListener non ha attributi, ma ha un'unica relazione opzionale con l'elemento *Navigation*, che indica se il Listener deve avviare o meno una navigazione verso un nuovo ViewController. Un'istanza di ClickListener priva della relazione con un elemento Navigation implica la necessità per lo sviluppatore di definire via codice l'azione che intende scatenare al momento del click da parte dell'utente finale.

Navigation

L'elemento Navigation modella una specifica navigazione verso un nuovo ViewController. Può essere definito attraverso la composizione che parte dall'elemento ClickListener appena analizzato, oppure attraverso quella che parte dall'elemento MenuItem. Navigation possiede un unico attributo chiamato *destination*, utilizzato per identificare il ViewController destinazione della navigazione.

AsyncTask

L'elemento AsyncTask permette, all'interno dell'applicazione, l'esecuzione asincrona di pezzi di codice, effettuando in background operazioni che potrebbero risultare molto pesanti ed evitando così di bloccare l'interfaccia grafica mostrata all'utente. L'unico attributo richiesto per questo elemento è il nome, perché non è possibile definire attraverso un modello la logica applicativa dell'elemento.
AsyncTask presenta quindi un attributo e una relazione:

- *Name*: è il nome del task asincrono e deve essere univoco all'interno di uno stesso ViewController.
- *ViewController*: è la relazione inversa di asyncTasks su ViewController.

SourceType

L'elemento SourceType non corrisponde a nessun componente dell'applicazione, ma è un oggetto di tipo enumerativo introdotto esclusivamente nel metamodello implementativo per essere usato come attributo di ImageView, VideoView e AudioPlayer ai fini di definire la provenienza dei file di cui fanno uso.
I valori possibili sono tre:

- *HardwareFile*: indica che il file a cui si deve accedere proviene direttamente dal componente hardware del dispositivo che lo genera.
- *LocalFile*: indica che il file a cui si deve accedere proviene da un file locale presente in memoria.
- *RemoteFile*: indica che il file a cui si deve accedere risiede su un server remoto.

4.1.1 Interfaccia Grafica

In questa sezione riportiamo l'analisi dettagliata di tutti gli elementi del metamodello implementativo che compongono l'interfaccia grafica, definibili grazie alle relazioni sull'oggetto ViewController che sono state presentate in precedenza.

Button

Tra i diversi widget che l'utente può utilizzare in fase di sviluppo di un'applicazione i bottoni sono i più basilari e funzionali. Essi determinano lo

scatenarsi di determinati eventi e il conseguente verificarsi di certe azioni, come la navigazione verso un nuovo ViewController, attivata in genere dall'elemento ClickListener, che rileva il click del bottone da parte dell'utente. Quest'ultimo può infatti essere usato per definire una Navigation che conduce a un nuovo ViewController. Ogni Button presenta un codice identificativo univoco, così come la maggior parte dei componenti grafici, e un titolo, che sono stati quindi introdotti tra gli attributi di Button.
Per poter modellare correttamente il Listener che rileva il click sul bottone abbiamo aggiunto una relazione che lega Button all'elemento ClickListener.
Gli attributi di Button sono:

- *Id*: è l'identificativo del bottone.
- *Title*: è il titolo che compare sopra il bottone.

Le relazioni di Button sono:

- *ClickListener*: rende il bottone selezionabile e quindi in grado di scatenare un'azione al momento del click da parte dell'utente. La sua cardinalità va da zero a uno poiché è consentita la definizione di bottoni non cliccabili.
- *ViewController*: è la relazione inversa di buttons su ViewController.

ImageView e VideoView

L'elemento ImageView permette di inserire foto o immagini all'interno dei ViewController, mentre VideoView consente di caricare un video. Le immagini e i video possono corrispondere a file salvati all'interno dell'applicazione, file remoti scaricati da Internet, oppure file che provengono direttamente dal gestore della fotocamera o videocamera del dispositivo sul quale l'applicazione è installata. Si rende quindi necessario aggiungere un attributo per indicare la provenienza del file che si intende mostrare e un altro per identificarlo, utilizzato solo nel caso in cui il file non abbia come fonte la fotocamera o videocamera del dispositivo.
Gli attributi di ImageView e VideoView sono:

- *Id*: è l'identificativo dell'ImageView o della VideoView.
- *FileUri*: è l'attributo che permette al programmatore di inserire il nome del file da utilizzare nel caso in cui sourceType fosse impostato a *localFile*, oppure l'URL a cui accedere per scaricare la foto o il video nel caso in cui il valore di sourceType fosse *remoteFile*. Per i file che provengono dalla fotocamera o videocamera del dispositivo questo attributo deve essere ignorato.

Le relazioni di ImageView e VideoView sono:

- *SourceType*: è la relazione con l'attributo enumerativo sourceType, che permette di stabilire la provenienza dell'immagine o del video da mostrare a schermo.

- *ViewController*: è la relazione inversa di imageViews o videoView su ViewController.

TextView

Una TextView rappresenta un'area di testo il cui contenuto non può essere modificato da parte dell'utente finale.
Gli attributi di TextView sono:

- *Id*: è l'identificativo dell'oggetto TextView.

- *Content*: é il contenuto della TextView. Il contenuto corrisponde a una stringa e può essere molto lungo.

L'unica relazione di TextView è:

- *ViewController*: è la relazione inversa di textViews su ViewController.

EditText

L'elemento EditText rappresenta un'area di testo il cui contenuto può essere modificato dall'utente finale e corrisponde in genere ad un campo di una form. Le sue proprietà sono molto simili a quelle definite per l'elemento TextView.
Gli attributi di EditText sono:

- *Id*: è l'identificativo della EditText.

- *InitialContent*: è il contenuto iniziale mostrato in modo predefinito all'apertura del ViewController e in quanto opzionale può essere lasciato vuoto.

L'unica relazione di EditText è:

- *ViewController*: è la relazione inversa di editTexts su ViewController.

ListView e GridView

ListView e GridView sono concetti simili, utilizzati per mostrare un elenco di elementi all'interno di uno stesso ViewController. La differenza risiede nel fatto che le ListView sono delle liste scorrevoli, mentre le GridView sono tabelle con scorrimento bidimensionale. Entrambe sono costituite al loro interno da celle, ciascuna delle quali può avere contenuti di vario tipo, anche se in genere si tratta di testi e/o immagini. Solitamente la selezione di una determinata cella conduce all'apertura di un nuovo ViewController con lo scopo di mostrare all'utente maggiori dettagli riguardo all'elemento cliccato. Ciò è possibile grazie ad un ClickListener, che corrisponde al metodo invocato al momento del click di una cella. In base alla logica implementativa definita all'interno del metodo viene scatenata una determinata azione, come l'apertura del nuovo ViewController configurato sulla base dell'elemento selezionato.
L'unico attributo di ListView e GridView è:

- *Id*: è l'identificativo della ListView o GridView.

Le relazioni di ListView e GridView sono:

- *ListViewCells* o *GridViewCells*: consentono al programmatore di inserire un numero qualsiasi di celle nella ListView o GridView. La cardinalità di queste relazioni varia da zero a molti, in quanto l'utente può decidere di inserire manualmente le celle successivamente. Nel caso in cui l'utente intendesse rendere i contenuti della lista o della tabella dinamici, per esempio sulla base di dati salvati in un database locale, l'utente deve lasciare la View priva di celle e gestire successivamente via codice la logica applicativa necessaria per inserire i contenuti nelle varie celle.

- *ClickListener*: consente alla lista o alla tabella di rendere le proprie celle selezionabili e quindi in grado di scatenare un'azione al momento del click da parte dell'utente. La sua cardinalità va da zero a uno poiché la lista o la tabella può presentare celle non selezionabili.

- *ViewController*: è la relazione inversa di listViews o gridViews su ViewController.

ListViewCell e GridViewCell

ListViewCell e GridViewCell sono le celle che possono essere aggiunte a ListView e GridView rispettivamente. Questi elementi non sono presenti nel metamodello astratto proprio per la natura di tale metamodello di modellare

ad alto livello la struttura delle applicazioni. Abbiamo anticipato che il contenuto delle celle delle GridView e delle ListView può essere personalizzato e, nel caso più semplice, il contenuto è rappresentato da una stringa. Per questo abbiamo scelto di dare all'utente la possibilità di definire nel modello dell'applicazione esclusivamente un titolo per ciascuna cella: modificando poche righe di codice può sempre decidere di personalizzarla successivamente. In base a quanto detto l'unico attributo necessario per l'elemento ListViewCell o GridViewCell è pertanto *title*, che permette di definire il titolo da inserire nella specifica cella. Non si tratta di un attributo il cui valore deve essere inserito obbligatoriamente dallo sviluppatore, il quale può decidere di settare il titolo in un secondo momento via codice.

WebView

Una WebView è una View usata per mostrare una pagina Web, il cui contenuto e il layout sono definiti attraverso file HTML e CSS, che appartengono alle risorse dell'applicazione. L'elemento WebView è particolarmente indicato nello sviluppo di applicazioni mobili per chi intendesse definire in maniera semplice i contenuti e lo stile di una View, sfruttando gli strumenti della programmazione web.
Gli attributi di WebView sono:

- *Id*: è l'identificativo della WebView.
- *HTMLFileName*: è l'attributo che rappresenta il nome del file HTML che si intende utilizzare per definire il contenuto della WebView. Questo campo può essere lasciato vuoto ed in tal caso il sistema genera il codice per il corretto settaggio della WebView, senza specificare alcun riferimento al file HTML, che dovrà essere inserito manualmente dal programmatore. Nel caso in cui si intendesse definire il layout della pagina attraverso un file CSS, è necessario definire il riferimento al file nell'header del file HTML.

L'unica relazione di WebView è:

- *ViewController*: è la relazione inversa di webViews su ViewController.

AlertDialog

L'elemento AlertDialog consente di visualizzare un messaggio con titolo e contenuto in sovrimpressione sullo schermo del dispositivo. Quindi, si rende necessario avere all'interno dell'elemento due attributi per inserire titolo e contenuto del messaggio. In quanto la comparsa di un messaggio di questo

tipo blocca l'interfaccia grafica, vengono automaticamente mostrati anche due bottoni di conferma e annullamento.
Gli attributi di AlertDialog sono:

- *Id*: è l'identificativo dell'AlertDialog.
- *Title*: è il titolo dell'AlertDialog.

Le relazioni di AlertDialog sono:

- *Message*: rappresenta il contenuto del messaggio.
- *ViewController*: è la relazione inversa di alertDialogs su ViewController.

ProgressDialog

L'elemento ProgressDialog è, come l'AlertDialog, un messaggio che può avere un titolo e un contenuto, ma con due differenze sostanziali: non ha bottoni di conferma o annullamento e presenta uno spinner oppure una barra orizzontale, che indica lo stato di avanzamento dell'operazione in corso.
Gli attributi di ProgressDialog sono:

- *Id*: è l'identificativo del ProgressDialog.
- *Title*: è il titolo del ProgressDialog.
- *Message*: rappresenta il contenuto del messaggio.
- *Spinner*: è un attributo booleano utilizzato per scegliere se l'avanzare dell'operazione deve essere mostrato attraverso uno spinner oppure attraverso una semplice barra orizzontale.

L'unica relazione di ProgressDialog è:

- *ViewController*: è la relazione inversa di progressDialogs su ViewController.

4.1.2 Hardware

Di seguito riportiamo una descrizione dei principali elementi di una generica applicazione che gestiscono l'interfacciamento con l'hardware del dispositivo. Ricordiamo che lo scopo del metamodello implementativo è quello di consentire la definizione di un modello dell'applicazione sufficientemente espressivo da permettere la generazione dei sorgenti nativi corrispondenti. Per i Controller dei componenti hardware questa fase si concretizzerà nella produzione

dei pezzi di codice che implementano la logica applicativa di interfacciamento con il componente modellato.
Ogni ViewController può presentare al suo interno diversi metodi per l'interazione con l'hardware, tuttavia di tutti i metodi implementabili abbiamo deciso di modellarne solo alcuni, quelli utilizzati più di frequente nelle applicazioni. Tali elementi sono stati modellati come figli del concetto HWResourceController, presentato nel metamodello astratto, e sono presentati di seguito. E' bene tenere presente che quando è necessario interfacciarsi con un qualsiasi componente hardware dall'interno di un ViewController i concetti del metamodello astratto coinvolti sono sempre due: quello di HWResource e quello di HWResourceController.

LocationManager

L'elemento LocationManager è un HWResourceController che sfrutta il sensore GPS del dispositivo per determinare la posizione geografica dell'utente. Questo elemento non necessita di particolari attributi nel metamodello implementativo, perché quando l'utente decide di fare in modo che un certo ViewController possa recuperare la posizione dell'utente non deve settare alcun parametro, quanto piuttosto decidere come implementare la logica dell'utilizzo della posizione ottenuta, che purtroppo non può essere modellata. Perciò, quando il programmatore definisce nel modello un oggetto di tipo LocationManager, nel codice generato automaticamente verranno inseriti i metodi di interfacciamento con il sensore GPS, la cui implementazione deve essere ultimata dallo sviluppatore.
L'unica relazione che presenta l'elemento è *viewController*, ossia la relazione inversa di locationManager su ViewController.

AudioPlayer

AudioPlayer è un HWResourceController che sfrutta la risorsa hardware Speaker per la riproduzione di file musicali provenienti da file locali, da remoto o per la riproduzione di registrazioni effettuate col microfono del dispositivo. Quando lo sviluppatore decide di inserire nel modello questo elemento, al momento della generazione dei sorgenti vengono automaticamente inseriti i pezzi di codice che definiscono la logica applicativa di gestione della riproduzione musicale (play, pause, stop). Le funzionalità offerte dall'AudioPlayer riguardano in particolare il recupero di uno specifico file audio e la sua riproduzione, oltre alla messa in pausa e all'arresto della riproduzione stessa.
Gli attributi di AudioPlayer sono:

- *Id*: è l'identificativo dell'AudioPlayer.

- *FileUri*: è l'attributo che permette al programmatore di inserire il nome del file locale da utilizzare come sorgente, oppure l'url da cui scaricare il file musicale nel caso fosse remoto. Laddove il file audio da riprodurre corrisponda a una registrazione col microfono del dispositivo l'attributo fileUri deve essere ignorato.

Le relazioni di AudioPlayer sono:

- *SourceType*: è la relazione con l'attributo enumerativo sourceType, che permette di stabilire la provenienza del file musicale che si intende riprodurre.

- *ViewController*: è la relazione inversa di audioPlayer su ViewController.

PhotocameraController, VideocameraController e AudioRecorder

PhotocameraController, VideocameraController e AudioRecorder vengono utilizzati rispettivamente per scattare una foto con la fotocamera, registrare un video con la videocamera ed effettuare una registrazione audio attraverso il microfono del dispositivo.
PhotocameraController e VideocameraController, una volta implementati, si presentano graficamente con un bottone all'interno della Window del ViewController in cui vengono inseriti. Tale bottone, una volta cliccato, permette all'utilizzatore di aprire sul dispositivo l'applicazione predefinita per la cattura di foto o di video. I file generati vengono automaticamente salvati in memoria e ritornati all'applicazione chiamante, dove lo sviluppatore dovrà ultimare l'implementazione definendo via codice la logica di gestione dei file ricevuti. L'unico attributo di PhotocameraController è imageViewId, che permette di associare al PhotocameraController una ImageView con HWResource come sourceType. Tale attributo dovrà essere settato dal programmatore nel caso in cui decidesse di visualizzare direttamente nella Window del ViewController l'ultima foto scattata. Considerazioni analoghe valgono per VideocameraController, che presenta l'attributo videoViewId, da utilizzare per indicare l'id della VideoView dove riprodurre l'ultimo video registrato.
AudioRecorder si presenta anch'esso con un bottone con titolo "Rec" nella Window del ViewController in cui viene inserito. Una volta cliccato il bottone non viene mostrata nessuna schermata particolare, al contrario l'utente continuerà a visualizzare la stessa View con il titolo del bottone mutato in "Stop", ad indicare che la registrazione dal microfono è in corso e può essere interrotta cliccando nuovamente sul bottone. E' possibile riprodurre direttamente l'ultimo file audio registrato attraverso un elemento AudioPlayer con

HWResource come SourceType: come per PhotocameraController e VideocameraController, l'elemento AudioRecorder presenta un apposito attributo, chiamato audioPlayerId, che lo sviluppatore può utilizzare per indicare l'id dell'AudioPlayer dove intende riprodurre l'ultimo file audio registrato.
Gli elementi PhotocameraController, VideocameraController e AudioRecorder presentano un attributo e una relazione:

- *ImageViewId*, *VideoViewId* o *AudioPlayerId*: è l'attributo che permette di indicare l'id dell'oggetto oggetto dove mostrare o riprodurre il file generato utilizzando l'hardware del dispositivo. Nel caso in cui lo sviluppatore preferisca o necessiti di definire via codice la logica applicativa di gestione del file ricevuto deve ignorare il settaggio dell'attributo al momento della modellazione dell'applicazione.

- *ViewController*: è la relazione inversa di photocameraController, videocameraController o audioRecorder su ViewController.

4.2 Elementi esclusi dalla modellazione

Alcuni elementi tra quelli presenti nel metamodello astratto non sono stati volutamente riportati nel diagramma del metamodello implementativo.
Tra questi ci sono quelli che rappresentano concetti puramente astratti. E' il caso dell'elemento *Window*, che può essere inteso come il contenitore delle View mostrate sullo schermo, o dei concetti *View*, *HWResource* e *HWResourceController*, i cui figli rappresentano i concetti concreti, ovvero quelli istanziabili. Persino l'elemento *NavigationStarter* è da considerarsi astratto, in quanto rappresenta un generico elemento da cui può avere origine una navigazione verso un nuovo ViewController. Oltre a questi elementi, per cui risulta ovvia la mancata introduzione in un metamodello contenente unicamente concetti concreti, ci sono quelli che lo sviluppatore non è tenuto a specificare direttamente al momento della definizione del modello per l'applicazione desiderata, in quanto scontati e implementati in automatico dal generatore di codice sulla base della definizione degli altri elementi. Infine ci sono gli elementi non astratti che abbiamo dovuto escludere dall'implementazione per una serie di ragioni che vedremo tra poco. Ma prima di considerare quest'ultimi elementi entriamo nel merito di quelli esclusi dalla modellazione in quanto implementati implicitamente.

4.2.1 Implementati implicitamente

Tra gli elementi appartenenti a questa categoria troviamo *LifeCycleOperation* e i suoi figli, non riportati nel metamodello implementativo perché è consi-

derata implicita la presenza nei ViewController dei metodi per la gestione del loro ciclo di vita. Quindi nei sorgenti provenienti dalla traduzione degli elementi ViewController sono presenti le intestazioni dei metodi del ciclo di vita, nonostante essi non siano esplicitati nel metamodello implementativo.
Lo stesso vale per l'elemento *Manifest*, che costituisce una parte fondamentale per il funzionamento corretto delle applicazioni e che viene automaticamente generato al momento della produzione dei sorgenti dell'applicazione, ricavando le informazioni necessarie dagli altri elementi appartenenti al modello dell'applicazione.
Non abbiamo previsto la modellazione nemmeno delle *Content Resource*, in quanto, come vedremo nel capitolo successivo, al fine di raccogliere tutte le risorse necessarie all'applicazione abbiamo predisposto nel progetto realizzato un'apposita cartella chiamata *user_files*, che l'utente deve riempire prima di lanciare il workflow del generatore di codice, il quale si occupa di copiare i file contenuti in user_files nelle giuste directory del progetto Android o iOS dell'applicazione, in base alla loro estensione.
Per quanto riguarda, invece, l'elemento *ExternalLibrary*, non è stato indicato nel metamodello implementativo per via del fatto che il generatore incorpora automaticamente nei sorgenti Android o iOS le librerie esterne necessarie.

4.2.2 Attualmente non implementati

A partire dal metamodello implementativo, come vedremo nel dettaglio nel prossimo capitolo, siamo riusciti a fornire una prima mappatura in codice della maggior parte degli elementi concreti presenti nel metamodello astratto. Gli elementi per cui non abbiamo implementato la traduzione in codice non sono stati volutamente riportati nel metamodello implementativo. Le ragioni di questa incompletezza riguardano principalmente la forte dipendenza dell'implementazione di alcuni elementi dalla piattaforma per cui l'applicazione deve essere sviluppata e il fatto che sono caratterizzati da una logica applicativa non definibile con un approccio model-driven. Questi elementi sono in particolare SQLite, MapView e alcuni controller delle risorse hardware.

SQLite è una libreria esterna che può essere usata per la creazione e gestione di un database relazionale ed esiste sia la versione per Android che quella per iOS. Abbiamo pensato che l'implementazione dell'elemento SQLite non costituisse una priorità per una serie di ragioni. In primo luogo risulta difficile definire la struttura completa delle tabelle del database e le query richieste attraverso un modello, in aggiunta al fatto che non tutte le applicazioni necessitano di un database relazionale e per le due piattaforme si tende ad adottare soluzioni diverse. Sebbene in Android l'utilizzo di SQLite sia ab-

bastanza frequente, gli sviluppatori iOS tendono a sfruttare la libreria Core Data, che offre la possibilità di creare in maniera estremamente semplice dei database ad oggetti. Ciò non esclude comunque il fatto che l'impiego della libreria SQLite permetterebbe sicuramente di uniformare la scelta del database su un'applicazione pensata per essere sviluppata per più piattaforme.

MapView concettualizza nel metamodello astratto una View contenente una mappa geografica. In iOS attraverso l'oggetto MKMapView è possibile definire in maniera molto semplice via codice il contenuto della mappa geografica. Invece risulta essere più complicata la definizione della mappa attraverso l'oggetto MapView di Android, al punto che si tende spesso a usare piuttosto una WebView, sfruttando le API offerte da Google Maps. Nel momento in cui abbiamo iniziato la definizione del metamodello implementativo, era inoltre in corso l'aggiornamento della libreria esterna utilizzata da MapView, che dalla prima versione passava alla seconda, dove certi aspetti riguardo a come personalizzare la mappa non appaiono tuttora chiari. Per queste ragioni, in aggiunta al fatto che risulta difficile definire la mappa da mostrare a schermo con un approccio model-driven, abbiamo preferito accantonare l'idea di implementare MapView e di concentrarci maggiormente sugli altri elementi del metamodello, elencando tra gli eventuali sviluppi futuri del nostro lavoro l'implementazione di MapView.

Abbiamo escluso dall'implementazione anche i controller delle risorse hardware raramente sfruttate dalle applicazioni mobili, escludendo il caso dei videogiochi dove i sensori hardware frequentemente utilizzati sono differenti. Tra questi citiamo l'accelerometro, il giroscopio, il sensore di prossimità, il sensore di luce e il Bluetooth: il loro impiego è talmente raro che abbiamo preferito concentrarci sull'implementazione degli HWResourceController delle risorse comunemente usate, come lo speaker, il microfono, la fotocamera e la videocamera.

4.3 Conclusioni

Nella prima parte di questo capitolo abbiamo descritto gli elementi introdotti nel metamodello implementativo, molti dei quali derivano da quello astratto. La definizione degli elementi del metamodello implementativo è stata effettuata cercando di mantenere bassa la sua complessità, considerando gli attributi e le relazioni strettamente indispensabili per la traduzione dei concetti in codice. Per citare un esempio consideriamo gli elementi ListViewCell e GridViewCell, dove ci siamo limitati a definire un unico attributo testuale per il settaggio del titolo: aggiungendo altri attributi avremmo potuto defi-

nire un prototipo di cella personalizzato, magari contenente un sottotitolo e un'immagine, che avrebbe però richiesto un effort non indifferente nella fase di traduzione del modello in codice, senza aggiungere alcun valore al lavoro di tesi.
Nella seconda parte del capitolo abbiamo invece analizzato gli elementi del metamodello astratto che non sono stati riportati in quello implementativo, in quanto esclusi dalla fase di implementazione per la serie di ragioni presentate oppure tradotti direttamente in codice sulla base degli altri elementi del modello e inseriti quindi implicitamente nei sorgenti generati.

Da un punto di vista strutturale i metamodelli astratto e implementativo presentano delle evidenti differenze sostanziali. Innanzitutto non è stato possibile mantenere le stesse relazioni presentate nella versione astratta per via della presenza nel nuovo metamodello di un sottoinsieme dei concetti della prima versione, in particolare quelli concreti, che si prestano ad esse tradotti in codice. Inoltre, il metamodello implementativo è stato definito con lo scopo di essere istanziato seguendo un procedimento standard volto a guidare lo sviluppatore nella fase di modellazione, ovvero partendo dall'elemento Application e seguendo le composizioni indicate nel diagramma di Figura 4.1, presentato a inizio capitolo. Il metamodello implementativo così definito permette infatti allo sviluppatore di definire in maniera molto semplice il modello di una qualsiasi applicazione mobile.

Nel capitolo successivo presenteremo gli strumenti usati nella fase di implementazione del lavoro di tesi e descriveremo la struttura del generatore di codice prodotto, dimostrando come sia possibile tradurre in codice gli elementi definiti nel modello dell'applicazione. Attraverso il generatore di codice il programmatore avrà così la possibilità di sviluppare buona parte dell'applicazione desiderata per le piattaforme Android e iOS seguendo un approccio puramente model-driven, ovvero definendo, attraverso un editor grafico, un'istanza del metamodello implementativo.

Capitolo 5

Generatore di codice

Prima di iniziare l'implementazione abbiamo valutato, tra gli strumenti software esistenti, quelli che ci permettessero di definire il metamodello implementativo e di generarne, attraverso un editor grafico, una sua istanza che rappresenti il modello dell'applicazione desiderata. A partire dall'istanza ci siamo poi preoccupati di generare i sorgenti nativi corrispondenti per le diverse piattaforme mobili presenti sul mercato, tra cui Android e iOS, che nel nostro lavoro costituiscono i target del processo di traduzione del modello in codice.
In questo capitolo partiremo pertanto con la presentazione degli strumenti e tecnologie utilizzati in fase di implementazione, per poi descrivere dettagliatamente la struttura del progetto realizzato, ovvero il generatore di codice. Inoltre presenteremo le regole e le notazioni da seguire al momento della definizione dell'istanza del metamodello per la corretta generazione dei sorgenti corrispondenti e analizzeremo l'output della fase di traduzione del modello in codice prodotto per ciascuno degli elementi modellati.

5.1 Tecnologie utilizzate

Durante l'implementazione abbiamo utilizzato l'ambiente di sviluppo Eclipse, e in particolare la distribuzione **Juno 4.2** contenente la versione 1.0 di **Epsilon**, che costituisce una famiglia di linguaggi e di strumenti utili per la generazione del codice, per la trasformazione da un modello ad un altro e per la validazione dei modelli [22]. Abbiamo scelto questa distribuzione principalmente perché offre supporto al framework di modellazione **EMF**, che verrà descritto di seguito e che rappresenta un potente strumento per la definizione e la gestione di modelli. Oltre ad EMF, Epsilon offre numerosi altri strumenti interessanti, tra cui EuGENia, che costituisce un tool utile per la costruzione di un editor grafico GMF a partire da un metamodello definito con EMF. Come vedremo, la creazione di un editor grafico persona-

lizzato per la definizione di un'istanza del metamodello rientra comunque tra i possibili sviluppi futuri del nostro lavoro.
Attraverso questa distribuzione di Eclipse siamo riusciti a sfruttare le funzionalità offerte dal framework EMF per la definizione, attraverso un editor grafico minimale, del metamodello implementativo. Aggiungendo alcuni plugin all'ambiente di sviluppo siamo riusciti, inoltre, a sfruttare gli strumenti del framework **openArchitectureWare** che, come vedremo in questo capitolo, è finalizzato alla definizione della traduzione di un'istanza del metamodello in codice.

5.1.1 Eclipse Modeling Framework

E' un framework per la modellazione e la generazione del codice di applicazioni basate su un modello di dati strutturato [23]. In particolare EMF mette a disposizione diversi strumenti con un'interfaccia grafica semplice per la definizione di un metamodello e la generazione di una sua istanza. A partire dall'istanza è possibile poi produrre classi Java, senza utilizzare altri strumenti all'infuori di quelli offerti da EMF. L'utilizzo esclusivo di questo strumento non ci permette di generare il codice nativo dell'applicazione per le diverse piattaforme mobili, per questo abbiamo deciso di utilizzarlo unicamente per la definizione del metamodello implementativo e per la generazione di sue istanze. Il metamodello implementativo è stato salvato in un file .ecore chiamato *metamodel.ecore*, interno al progetto, mentre l'istanza dinamica generata dallo sviluppatore viene salvata in un file .xmi, del quale l'utente può scegliere il nome, che dovrebbe corrispondere a quello dell'applicazione che intende sviluppare. In Figura 5.1, a titolo d'esempio, è riportata parte della

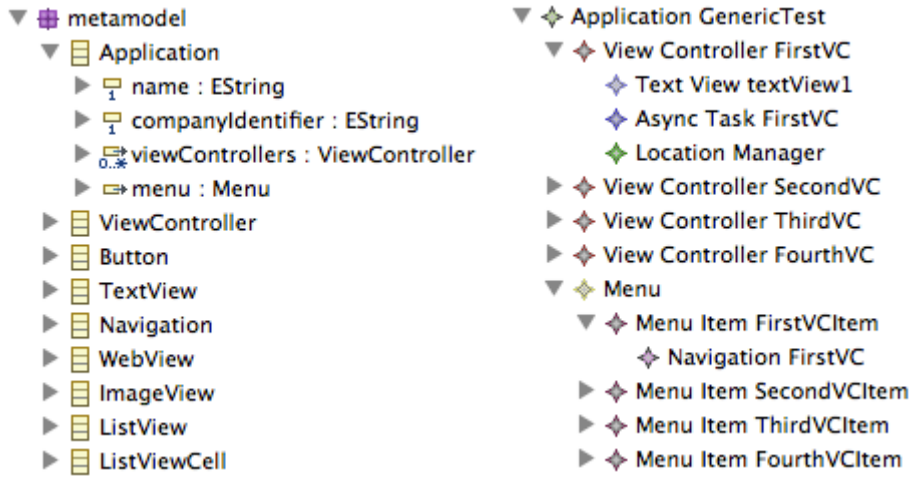

Figura 5.1: *Esempio*: parte di un file .ecore e parte di un file .xmi

struttura di un file .ecore, che costituisce il metamodello implementativo, e di un file .xmi, che rappresenta una sua istanza.
Una volta creata l'istanza del metamodello, ovvero il modello dell'applicazione, siamo interessati a generare i sorgenti corrispondenti per le piattaforme mobili Android e iOS: come anticipato in precedenza, per definire il processo di traduzione del modello in codice abbiamo deciso di appoggiarci ai componenti del framework openArchitectureWare.

5.1.2 openArchitectureWare

E' un framework sviluppato in Java utile per la validazione e la trasformazione dei modelli in codice [24]. Tale strumento fornisce un ottimo supporto per i modelli definiti con EMF, ma è in grado di parsare anche modelli UML2, XML o JavaBeans.
La piattaforma oAW offre diversi strumenti che consentono di sviluppare l'applicazione in modo modulare, generando codice in un qualsiasi linguaggio di programmazione. Il cuore di questa architettura è il Modeling Workflow Engine (MWE), il quale permette di inglobare diversi componenti, ciascuno con una propria funzionalità, che concorrono alla corretta generazione del codice sulla base del metamodello e del modello dell'applicazione definito dallo sviluppatore. Gli strumenti di oAW che abbiamo utilizzato durante l'implementazione sono elencati di seguito e descritti in dettaglio nei paragrafi successivi.

- *Xpand*: per definire i template di traduzione del modello in codice.
- *Xtend*: per aggiungere alle funzionalità offerte da Xpand le potenzialità del linguaggio Java.
- *Check*: per validare il modello dell'applicazione.
- *Modeling Workflow Engine*: per integrare i diversi componenti che devono essere processati per la generazione del codice.

Tutti questi componenti del framework sono stati recentemente incorporati nell'Eclipse Modeling Project (EMP), che costituisce un progetto mirato alla diffusione e promozione di tecnologie di sviluppo model-based attraverso la community Eclipse.

Xpand è un framework progettato con lo scopo di fornire allo sviluppatore un linguaggio per creare output testuali partendo da modelli EMF Ecore, Eclipse UML2, JavaBeans o XMLSchema [25]. In particolare Xpand permette di creare uno o più template, ovvero file con estensione .xpt che definiscono

l'effettiva traduzione del modello in codice. Tramite questo strumento è possibile produrre pezzi di codice in un qualsiasi linguaggio di programmazione, in quanto è in grado di generare tutto ciò che può essere espresso in forma testuale. Definendo opportunamente i template possiamo quindi sfruttare Xpand per generare i sorgenti nativi per una qualsiasi piattaforma mobile.
Nel nostro caso, una volta definito il metamodello in un file .ecore, possiamo creare dei template per mappare gli elementi modellati sui pezzi di codice da generare per la specifica piattaforma.
Per citare un esempio ipotizziamo che l'utente intendesse inserire un elemento Button in un ViewController all'interno dell'istanza del metamodello. Il generatore di codice dovrebbe generare il codice per la gestione del bottone all'interno del file del ViewController in cui è stato definito e, per fare in modo che questo accada, è necessario implementare dei template, ovvero dei file che definiscano il mapping richiesto. Come vedremo in dettaglio nella seconda parte del capitolo, durante l'implementazione abbiamo definito due file Xpand per ciascun elemento del metamodello implementativo, suddividendo in cartelle diverse i file finalizzati alla produzione di codice Java per Android, da quelli finalizzati alla produzione di codice Objective-C per iOS. Il framework Xpand dispone infatti di numerose potenzialità, che permettono, attraverso un linguaggio con una sintassi particolarmente semplice, di avere flessibilità nella definizione dei template e quindi nella gestione dei file del progetto.
All'inizio di qualsiasi template Xpand viene sempre eseguito l'import del metamodello sulla base del quale deve essere generato il codice. Nello stesso file vengono quindi delineate le traduzioni da compiere e, facendo uso dei costrutti "extend" e "define", è possibile includere template dislocati su altri file esterni, rendendo così modulare e comprensibile la struttura del progetto.
In Figura 5.2 è riportato un pezzo di un template Xpand, estratto dal progetto del generatore di codice che presenteremo tra poco, dove si può osservare l'importazione del metamodello implementativo e l'utilizzo di altri costrutti scritti in maiuscolo appartenenti alla sintassi Xpand. Nell'esempio si vuole

```
«IMPORT metamodel»
«DEFINE buttons FOR ViewController-»
«FOREACH buttons AS b-»
«IF b.clickListener!=null && b.clickListener.navigation==null-»
-(IBAction)«b.id»ClickAction:(UIButton *)sender
//TODO Implement the action
«ENDIF-»
«ENDFOREACH-»
«ENDDEFINE-»
```

Figura 5.2: *Esempio*: parte di un file Xpand

in particolare inserire un metodo Objective-C per la gestione dell'azione da svolgere al momento del click di un bottone che non rappresenti un NavigationStarter, la cui logica applicativa dovrà essere in seguito implementata dallo sviluppatore.
Oltre a Xpand, il framework oAW mette a disposizione il linguaggio Xtend, che permette di aggiungere nuove funzionalità a quelle offerte da Xpand.

Xtend è un linguaggio di programmazione che permette di sfruttare le potenzialità offerte dal linguaggio Java durante il processo di traduzione del modello in codice. Grazie a Xtend è infatti possibile definire dei metodi all'interno di una classe Java e chiamarli dai template Xpand, a patto di inserire il mapping all'interno di un file Xtend (con estensione .ext) e importare tale file all'interno del template facendo uso del costrutto "extension".

```
import metamodel;
Void deleteApplicationFolder():
   JAVA android_extensions.AndroidJavaUtil.deleteApplicationFolder();
Void copyDefaultAndroidFiles():
   JAVA android_extensions.AndroidJavaUtil.copyDefaultAndroidFiles();
Void copyUserFiles():
   JAVA android_extensions.AndroidJavaUtil.copyUserFiles();
```

Figura 5.3: *Esempio*: parte di un file Xtend

In Figura 5.3 è riportato a titolo di esempio un file Xtend estratto dal nostro progetto. Come per i file .xpt la prima riga di codice è finalizzata all'importazione del metamodello, mentre a seguire si trovano le dichiarazioni dei metodi del file Java che possono essere invocati dai template Xpand. Ciascuna dichiarazione presenta a sinistra il nome con il quale il metodo può essere chiamato dai file Xpand attraverso la notazione «nome_del_metodo», mentre a destra è riportato il nome del metodo all'interno del file Java. In questo modo avviene la mappatura tra il nome usato dai template Xpand al momento dell'invocazione e il nome reale presente nella classe Java. I file Java definiti dallo sviluppatore sono comuni classi Java pertanto possono contenere metodi e attributi ausiliari che non devono essere necessariamente mappati dai file Xtend.
Tuttavia i file .ext non sono utilizzati unicamente come mediatori per l'accesso a metodi di classi Java, in quanto possono contenere anche delle operazioni specifiche, definite in linguaggio Xtend. Come vedremo tra poco abbiamo sfruttato questa funzionalità di Xtend al momento della validazione del modello dell'applicazione, attraverso il linguaggio Check.

Check è un linguaggio fornito da openArchitectureWare per specificare i

vincoli che il modello dell'applicazione definito dall'utente deve adempiere per garantire la corretta generazione del codice corrispondente. Un file Check, con estensione .chk, è costituito da un insieme di espressioni, che fanno riferimento agli elementi del metamodello EMF da cui è stata generata l'istanza dell'applicazione, e che fanno uso della logica matematica e di un sottoinsieme delle funzioni Java esistenti. Per esempio, nelle espressioni scritte con Check, si possono usare i quantificatori "forall" ed "exist" della logica, e funzioni Java come "split" e "matches", utilizzate principalmente sulle stringhe.
Sfruttando questo linguaggio è possibile controllare che i testi inseriti dall'utente in fase di definizione dell'istanza non contengano caratteri proibiti, oppure eseguire controlli di altro genere, come verificare che gli id assegnati agli elementi dell'applicazione non violino una certa espressione regolare e siano univoci. Proprio ai fini di effettuare dei controlli più articolati è possibile definire un file Xtend che, come avviene per i template Xpand, affianchi i file .chk offrendo la possibilità di chiamare metodi Java o eseguire operazioni definite in linguaggio Xtend. Nel nostro caso abbiamo sfruttato questa funzionalità per effettuare dei controlli sugli id degli elementi del metamodello: per verificare l'univocità di questi attributi abbiamo infatti costruito un array all'interno di un file Xtend contenente tutti gli id degli elementi del modello, per poi richiamarlo al momento opportuno dal file di Check.

```
import metamodel;
extension model_checks::Extensions;

context Application ERROR "Application name '"+name+
"' must match the regular expression ^[a-zA-Z]+" :
      validAppName(name)==true;

context Application ERROR "Names of ViewControllers must be unique.
Duplicates: " + duplicatedIds(viewControllers.name) :
      duplicatedIds(viewControllers.name).size==0;

context Application ERROR "Ids must be unique. Duplicates: " +
duplicatedIds(ids()) :
      duplicatedIds(ids()).size==0;
```

Figura 5.4: *Esempio*: parte di un file Check

In Figura 5.4 è riportata a titolo d'esempio una parte del file Checks.chk, che abbiamo utilizzato nel progetto per svolgere tutti i controlli sul modello dell'applicazione e di cui parleremo successivamente. Come per i file Xtend, nella prima riga effettuiamo l'import del metamodello, necessario per accedere agli elementi che lo compongono. Nella seconda riga è riportata la dichiarazione di utilizzo di un file Xtend esterno, contenuto nel package

model_checks. A seguire sono riportate alcune espressioni che caratterizzano il linguaggio Check: in ogni espressione è richiesto che venga definito il contesto, ovvero l'elemento del modello da controllare e il messaggio da mostrare nel caso in cui la regola definita di seguito non venga soddisfatta dal modello analizzato. Prima di indicare il testo del messaggio è necessario definire il tipo di azione che deve essere eseguita nel caso in cui il vincolo non venga soddisfatto: il vincolo non soddisfatto deve essere classificato come errore, attraverso la keyword "ERROR", oppure come warning, utilizzando la keyword "WARNING". La differenza risiede nel fatto che nel primo caso viene bloccata l'esecuzione del workflow e quindi la generazione del codice, mentre nel secondo caso viene unicamente mostrato a titolo informativo il messaggio definito nel vincolo. I messaggi vengono riportati in fase di compilazione nella console di Eclipse solo nel caso in cui non venga soddisfatto il vincolo associato, insieme agli eventuali altri errori riscontrati dal Modeling Workflow Engine, che effettua la vera e propria generazione di codice.

Modeling Workflow Engine è un framework per l'integrazione e la gestione del workflow di processamento dei modelli [26], che consente di generare un output testuale, nel nostro caso codice nativo Android o iOS, partendo dal modello .xmi dell'applicazione, sfruttando gli strumenti definiti in precedenza. Per eseguire il generatore è necessario definire, all'interno di un file con estensione .mwe scritto con un linguaggio basato su XML, il workflow, ovvero la sequenza di componenti che devono essere processati. Ogni WorkFlowComponent rappresenta una parte del processo di generazione e in genere si tratta di un parser, di un trasformatore del modello o di un generatore di codice. All'interno di questo file si definisce solitamente quale modello .xmi e quali template Xpand occorre considerare al momento della traduzione in codice, in quale cartella inserire il codice generato, quali sono i file di Check che contengono i vincoli da verificare, quali operazioni occorre eseguire all'avvio del workflow, etc. Il file .mwe è usato in questo senso come eseguibile per avviare il Workflow Engine e definire tutti i passaggi che portano alla generazione del codice.

5.2 Progetto

In questa sezione illustreremo la struttura del progetto, e in particolare spiegheremo come abbiamo utilizzato le tecnologie descritte nel paragrafo precedente nel contesto dell'implementazione, che rappresenta il fulcro del lavoro di tesi. I sorgenti del progetto completo sono liberamente scaricabili dal seguente link, dove abbiamo aggiunto a titolo d'esempio anche alcune appli-

cazioni prodotte col generatore di codice:

`https://github.com/perego-pezzetti/MobileCodeGenerator`.

Ricordiamo che lo scopo del generatore di codice è permettere allo sviluppatore di definire un'istanza del metamodello implementativo che rappresenti il modello dell'applicazione da sviluppare e, a partire da questa, riuscire a generare automaticamente, lanciando il Workflow Engine, i sorgenti nativi corrispondenti per la piattaforma Android, per la piattaforma iOS o per entrambe. In Figura 5.5 è mostrato lo schema di funzionamento ad alto livello del processo di traduzione del modello in codice, che prende in input il file .xmi e produce in output i sorgenti nativi.

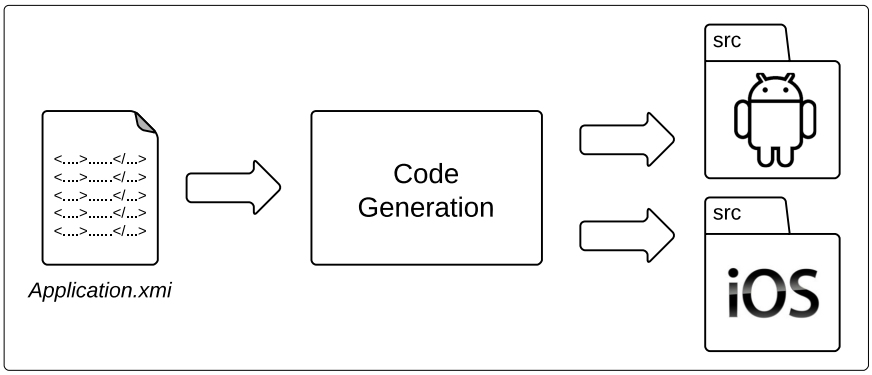

Figura 5.5: *Generatore di codice*: processo di traduzione

In Figura 5.6 è mostrata invece la struttura generale del progetto completo, dove le cartelle di maggiore interesse sono due: *src* e *utils*. La prima con-

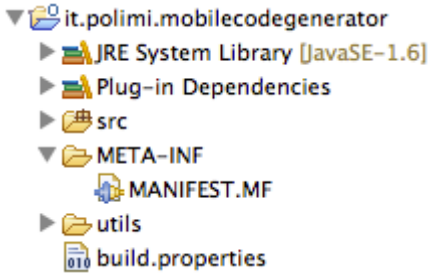

Figura 5.6: *Generatore di codice*: struttura generale dei sorgenti

tiene tutti i file utili alla definizione del metamodello e alla produzione di un'istanza di esso attraverso il framework EMF, oltre a quelli finalizzati alla

generazione del codice definiti attraverso gli strumenti di oAW. La seconda, come vedremo, contiene tutti i file di supporto per la corretta generazione del codice. Tra i sorgenti ci sono inoltre anche tutte le librerie di sistema, i plugin e i file di configurazione necessari (all'interno della cartella *META-INF*) per garantire il corretto funzionamento del generatore. Vediamo ora nel dettaglio il contenuto delle cartelle src e utils.

5.2.1 Cartella *src*

In Figura 5.7 è riportata la struttura della cartella src, dove si può notare la presenza di numerose sottocartelle, che a loro volta contengono file diversi con funzioni specifiche nel contesto della modellazione e traduzione del modello in codice. Cercheremo di analizzarle tenendo presente quello che è stato l'obiettivo della fase di implementazione: creare dei template finalizzati alla traduzione di un'istanza del metamodello in codice nativo per le piattaforme Android e iOS. Per quanto riguarda Android, i template sono stati definiti per produrre applicazioni compatibili con qualsiasi smartphone con livello API minimo pari a 8 e scegliendo come target il livello API 17, per iOS invece i template sono stati definiti in modo da produrre applicazioni compatibili con iPhone e iPod Touch, scegliendo come target la versione 6 di iOS.

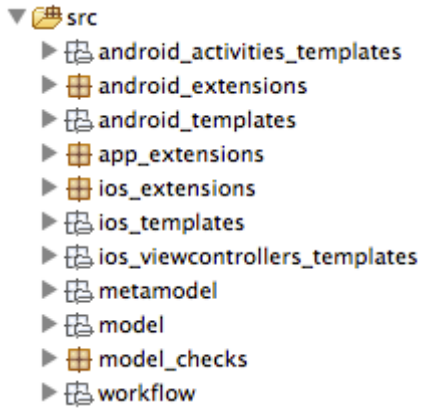

Figura 5.7: *Generatore di codice*: cartella src

Inizieremo, quindi, dall'analisi della cartella *metamodel*, in cui si trovano il file .ecore e il diagramma che definiscono, in forma testuale il primo e in forma grafica il secondo, il metamodello implementativo delle applicazioni mobili ampiamente descritto in precedenza. Di seguito a metamodel analizzeremo la cartella *model*, in cui è presente un archivio di tutti i file .xmi, generati dall'utente al momento della definizione del modello dell'applicazione desiderata, e la cartella *model_checks*, che contiene i file finalizzati alla validazione

dei modelli dell'applicazione. L'analisi si conclude con la cartella *workflow*, che contiene i file MWE per la definizione del workflow del generatore.
Le cartelle rimanenti contengono invece tutti i template Xpand, le estensioni Xtend e le classi Java necessarie per la definizione della traduzione del modello in codice. Non ci addentreremo chiaramente nel dettaglio di tutti questi file, i cui contenuti possono comunque essere visualizzati nel progetto.

Sottocartella *metamodel*

Abbiamo appena visto che metamodel costituisce la cartella contenente i file definiti attraverso il framework EMF, in cui è delineato il metamodello implementativo con tutti i suoi componenti. Come si può notare in Figura 5.8, nonostante vi siano due piattaforme target del processo di generazione del codice, è presente un solo file .ecore per il metamodello (con il relativo diagramma), a conferma che la parte di modellazione astrae dalla specifica piattaforma per cui si intende produrre il codice nativo.

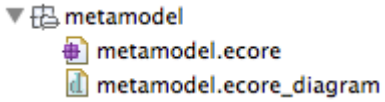

Figura 5.8: *Generatore di codice*: cartella metamodel

I file che verranno presentati di seguito faranno tutti leva sul contenuto di questa cartella e in particolare sul file .ecore, che costituisce il punto di partenza del lavoro di implementazione.

Sottocartella *model*

Questa cartella contiene esclusivamente file con estensione .xmi, proprio perché è finalizzata ad archiviare tutti i modelli delle applicazioni create dall'utente utilizzatore del nostro progetto. Perciò, nel momento in cui il programmatore crea una nuovo modello di applicazione a partire dal metamodello .ecore della cartella metamodel, è preferibile che il file .xmi generato venga salvato all'interno di questa cartella. I modelli di applicazione che abbiamo creato per testare il funzionamento del generatore sono i seguenti:

- *MethodicalTestApplication.xmi*: modello di un'applicazione priva di utilità pratica, utilizzata esclusivamente per testare la generazione dei pezzi di codice per gli elementi del metamodello implementati. Quello che abbiamo fatto è stato creare un ViewController iniziale composto da una serie di bottoni, ciascuno dei quali conduce ad un nuovo specifico ViewController creato appositamente per testare la generazione del codice per uno certo elemento del metamodello.

- *Multimedia.xmi*: è il primo dei tre esempi di applicazione che abbiamo creato per illustrare il funzionamento del generatore. Si tratta del modello di un'applicazione che mira a sfruttare l'hardware del dispositivo, in particolare la fotocamera, la videocamera e il microfono del dispositivo, per permettere all'utente di scattare delle foto, effettuare dei filmati e delle registrazioni audio che vengono poi salvate in memoria e riproposte all'interno di una gallery. Lo scopo principale di questo modello è stato quello di dimostrare la possibilità di generare un'applicazione in grado di interfacciarsi con le risorse hardware del dispositivo e che possa sfruttare come normali risorse le foto, i video e le registrazioni ricevute.

- *Promemoria.xmi*: è il modello di un'applicazione che consente all'utente di salvare in memoria delle note e cancellarle qualora non fossero più necessarie. Lo scopo di questo modello è mostrare come vengono tradotti in codice gli elementi EditText, Button, ProgressDialog e AlertDialog.

- *Valtellina.xmi*: è il modello di un'applicazione che permette di ricavare informazioni relative alle più rinomate località turistiche della Valtellina. L'applicazione corrisponde a quella che abbiamo presentato nel Capitolo 3 come esempio di istanza del metamodello astratto. Il modello è stato definito per accertare la corretta traduzione in codice di alcuni elementi dell'interfaccia grafica, tra cui VideoView, ListView, ImageView, TextView e WebView.

Come anticipato in precedenza, l'input del processo di generazione dei sorgenti per le piattaforme Android e iOS è costituito proprio dal file .xmi che contiene il modello dell'applicazione.

Sottocartella *model_checks*

Si tratta della cartella che contiene tutti i file necessari per validare il modello dell'applicazione che l'utente intende sviluppare. Il file Checks.chk, citato in precedenza come esempio concreto di utilizzo dello strumento Check, è posto all'interno di questa cartella, come è possibile vedere in Figura 5.9.

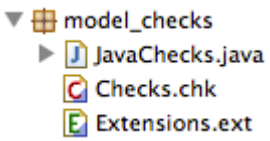

Figura 5.9: *Generatore di codice*: cartella model_checks

All'interno di un unico file abbiamo posto tutti i vincoli che devono essere necessariamente soddisfatti ai fini di garantire allo sviluppatore una generazione corretta dei sorgenti. Nel caso in cui qualcuno dei vincoli definiti non venga soddisfatto il codice non viene generato. Oltre ai vincoli che verificano la sintassi delle stringhe inserite dall'utente e che accertano l'univocità degli identificativi degli elementi del modello, abbiamo definito anche dei vincoli che controllano la coerenza nei riferimenti tra elementi dipendenti l'uno dall'altro.
Per citare un esempio, le risorse provenienti dai componenti hardware del dispositivo (foto, video, registrazioni audio, etc) vengono spesso utilizzate per configurare direttamente un elemento grafico (ImageView, VideoView, AudioPlayer) all'interno di uno specifico ViewController. Ipotizziamo che l'utente intenda associare un AudioPlayer a un AudioRecorder per riascoltare immediatamente l'ultima registrazione effettuata: quello che deve fare nel modello è indicare come audioPlayerId, che è un attributo di AudioRecorder, quello dell'AudioPlayer presente all'interno dello stesso ViewController che intende utilizzare come riproduttore di file registrati col microfono. Un controllo, rappresentato in Figura 5.10, potrebbe dunque essere quello che verifica che l'id dell'AudioPlayer indicato nell'AudioRecorder corrisponda a uno tra gli id degli elementi AudioPlayer di quel particolare ViewController. Oltre a questi controlli all'interno del file Checks.chk sono presenti controlli come quelli che accertano l'indicazione di una destinazione valida sugli elementi Navigation o che controllano le estensioni dei file che l'utente intende utilizzare come risorse per l'applicazione.

```
context AudioRecorder if(!audioPlayerId.matches("")) ERROR
"AudioPlayerId of AudioRecorder must be the id of an existing AudioPlayer
with sourceType = 'hardwareFile' in the same ViewController" :
viewController.audioPlayer!=null &&
viewController.audioPlayer.id.matches(audioPlayerId) &&
viewController.audioPlayer.sourceType.toString().matches("hardwareFile");
```

Figura 5.10: *Checks.chk*: check dell'id dell'elemento AudioRecorder

In aggiunta al file Checks.chk, nella cartella model_checks sono presenti anche un file Java, contenente i metodi di supporto invocati dal file Checks, e un file Xtend, che oltre a contenere l'array con tutti gli id dell'applicazione citato in precedenza, funge da mediatore tra il file Checks.chk e il file Java.

Sottocartelle *android_templates* e *android_activities_templates*

Queste due cartelle rappresentano il fulcro del progetto per quanto concerne la generazione del codice per la piattaforma Android, perché contengono

tutti i template finalizzati alla traduzione degli elementi del metamodello nei sorgenti corrispondenti. Alcuni tra questi template estendono dei file Xtend esterni, collocati nelle cartelle *android_extensions* e *app_extensions*, che vedremo nel dettaglio successivamente, in modo da poter invocare determinati metodi all'interno di classi Java.
In Figura 5.11 sono presentate alcune parti di codice tratte dal template Xpand per la costruzione del Manifest delle applicazioni Android, che verrà descritto dettagliatamente di seguito. Per motivi di spazio non abbiamo rappresentato tutto il contenuto del file: i puntini di sospensione che si trovano in alcune parti della figura indicano la presenza nel template originale di pezzi di codice che non sono stati volutamente riportati.

```
«IMPORT metamodel»
«DEFINE main FOR Application-»

«FILE name+"/AndroidManifest.xml"-»
<?xml version="1.0" encoding="utf-8"?>
<manifest ... >
    «IF viewControllers.locationManager.contains(true)-»
    <uses-permission android:name=
                    "android.permission.ACCESS_FINE_LOCATION"/>
    «ENDIF-»
    ...
    <application
        «FOREACH viewControllers AS vc-»
        <activity
            android:name="«vc.application.companyIdentifier».
                          «this.name.toLowerCase()».«vc.name»Activity"
            android:label="«this.name»" >
        </activity>
        ...
        «ENDFOREACH-»
    </application>
    ...
</manifest>
«ENDFILE-»
«ENDDEFINE-»
```

Figura 5.11: *AndroidManifest.xpt*: creazione del file AndroidManifest.xml

Analizzando il codice riportato in figura, partendo dall'alto, si può notare l'importazione del metamodello, la definizione di una variabile *main* per Application, a significare che siamo proprio nel contesto dell'elemento centrale del metamodello implementativo, e a seguire si trova il codice per la generazione del file AndroidManifest.xml. Attraverso l'uso dei guillemets, le speciali virgolette angolari, è possibile indicare i costrutti Xpand che si intende utilizzare, oltre a estrarre i valori degli attributi degli elementi del

modello e invocare delle funzioni esterne. Per esempio, per creare un nuovo file è stato utilizzato il costrutto "FILE" seguito dall'attributo *name* (nome dell'applicazione) concatenato alla stringa "/AndroidManifest.xml", ad indicare che il file che verrà creato si troverà nella cartella dell'applicazione e sarà nominato AndroidManifest.xml. Di seguito vi sono alcune righe di codice che saranno scritte nel file generato così come le leggiamo in figura, mentre ciò che è contenuto all'interno del costrutto "IF" successivo verrà scritto nel Manifest solo se l'applicazione conterrà almeno un elemento LocationManager, infatti si tratta del permesso di utilizzo del GPS. Questo non sarà chiaramente l'unico permesso che andremo all'occorrenza ad inserire nel file: le dichiarazioni degli altri permessi sono state omesse dal codice riportato in figura. Proseguendo si trova la dichiarazione di un ciclo "FOREACH": il codice al suo interno verrà scritto nel Manifest tante volte quante il numero di ViewController presenti nell'applicazione e riporterà l'identificativo di ogni Activity, ricavato grazie alla relazione inversa *application* e all'attributo *name* dell'elemento ViewController. In fondo al template è indicata la fine del file con il costrutto "ENDFILE", e la chiusura della definizione della variabile *main* con "ENDDEFINE", presente in tutti i template Xpand.

Dopo una breve ma concreta presentazione attraverso un esempio del funzionamento e dell'utilità dei template Xpand, passiamo ora a descrivere il contenuto delle due cartelle *android_templates* e *android_activities_templates*. All'interno della prima directory sono presenti i template per la produzione di tutti i sorgenti necessari di una qualsiasi applicazione Android, mentre nella seconda ci sono i template per la definizione dei pezzi di codice da inserire all'interno di ciascuna Activity, i quali vengono espansi e quindi importati quando necessario all'interno del template Activities.xpt, presente nella prima cartella. La ragione del fatto che abbiamo suddiviso su più file il codice che definisce la traduzione dell'elemento ViewController è legata all'elevato numero di righe di codice necessarie.
Riportiamo di seguito l'elenco completo di tutti i file presenti nella cartella android_templates.

- *Activities.xpt*: è il file creato con lo scopo di generare il codice per ciascuna Activity dell'applicazione Android, una per ogni elemento ViewController del modello dell'applicazione delineato dall'utente. Nel caso in cui l'utente aggiunga un menu all'applicazione, viene creata anche un'altra Activity chiamata MenuActivity. Come anticipato nel capitolo precedente, all'interno di ogni Activity generata abbiamo scelto di inserire automaticamente i metodi per la gestione del ciclo di vita, ovvero onCreate, onStart, onPause, onResume e onStop.

- *AndroidManifest.xpt*: è il template utilizzato per definire il Manifest.xml dell'applicazione Android, che include i permessi necessari all'applicazione e la dichiarazione di tutte le Activity presenti. Queste informazioni sono inserite dinamicamente, in base ai ViewController presenti nel modello e alla presenza o meno all'interno di queste di elementi che richiedono determinati permessi per offrire le funzionalità richieste. Abbiamo visto in precedenza il caso della dichiarazione del permesso di utilizzo del GPS, da inserire nel Manifest nel caso in cui all'interno di un certo ViewController venga definito un LocationManager.

- *DefaultFiles.xpt*: si tratta del template utilizzato per copiare all'interno dei sorgenti i file statici, che non dipendono dalla struttura del modello ma che sono necessari per fare in modo che il progetto generato possa essere importato correttamente in Eclipse, l'IDE per il quale è stato rilasciato il plugin ADT contenente tutti gli strumenti necessari per lo sviluppo di applicazioni Android. Inoltre il template è utile per copiare tutte le risorse inserite dall'utente nella cartella user_files nella directory adeguata, in base alla loro estensione. Per come è stato progettato il template, il programmatore ha infatti la possibilità di aggiungere in una cartella i file che intende usare come risorse per l'applicazione, senza preoccuparsi di spostarli all'interno del progetto dell'applicazione generato. Infine questo template produce una classe *Utils*, nella quale il programmatore ha la possibilità di inserire i metodi statici di supporto alla fase di implementazione successiva alla generazione del codice. La classe si presenta con un metodo statico già implementato che può essere utilizzato al momento della lettura dei file dalla memoria interna del dispositivo.

- *ImageDownloadingTask.xpt*: è un template utilizzato per generare una classe Java contenente la logica applicativa di un particolare AsyncTask, utilizzato per scaricare un'immagine da un server esterno attraverso la rete Internet e in modo asincrono rispetto al task principale di esecuzione. La classe viene generata ed istanziata nel codice qualora nel modello venga definita una ImageView il cui attributo sourceType sia settato a remoteFile.

- *XMLFiles.xpt*: è il template che consente la creazione di tutti i file .xml che definiscono il layout delle Activity. Per ogni elemento ViewController del modello viene prodotto, oltre alla Activity, il rispettivo file XML, contenente il codice per definire la sua interfaccia grafica. I contenuti dei file generati dipendono dagli elementi del metamodello che sono stati inseriti all'interno dei ViewController al momento della

definizione dell'istanza.
Generando una qualsiasi applicazione è possibile notare come gli elementi grafici vengano disposti sullo schermo in modo lineare, uno sotto all'altro. Questa disposizione permette allo sviluppatore di identificare immediatamente, attraverso l'editor grafico reso disponibile dal plugin ADT di Eclipse, tutti gli elementi grafici presenti nel ViewController e quindi riposizionarli senza troppe difficoltà secondo le proprie preferenze.

Sottocartelle *ios_templates* e *ios_viewcontrollers_templates*

Le directory *ios_templates* e *ios_viewcontrollers_templates* corrispondono, a livello d'importanza e tipo di contenuti, alle cartelle descritte nel paragrafo precedente, con la differenza che contengono i file utilizzati per la generazione dei sorgenti per la piattaforma iOS. Anche in questo caso la distinzione del codice per la traduzione in più file è legata esclusivamente alla volontà di avere un progetto ordinato e modulare, dove ogni elemento del ViewController dispone di un template dedicato. L'elenco che segue riporta una breve descrizione dei file contenuti nella cartella ios_templates.

- *ViewControllers.xpt*: corrisponde al file Activities.xpt nella cartella android_templates del progetto, in quanto il concetto di Activity per la piattaforma Android può essere accostato a quello di ViewController in ambiente iOS. Per ciascun elemento ViewController del modello il template produce due file Objective-C distinti, vale a dire il file .h, nel quale avvengono le dichiarazioni delle property pubbliche e dei protocolli usati, ed il file .m, che contiene l'implementazione vera e propria per il dato ViewController. Così come per le Activity Android, all'interno del file .m vengono automaticamente generati i metodi del ciclo di vita di un ViewController. I contenuti da inserire all'interno di questi due file sono definiti nei file .xpt presenti nella cartella ios_viewcontrollers_templates, i quali vengono espansi nel template ViewControllers.xpt al momento opportuno.

- *DefaultFiles.xpt*: è un template che ha la stessa funzione del suo corrispondente per Android, quindi anche in questo caso vengono copiati nella cartella dell'applicazione dei file predefiniti necessari per permettere l'importazione corretta dei sorgenti generati nell'IDE di sviluppo Xcode, in aggiunta ai file inseriti dall'utente che andranno a riempire la cartella *Supporting Files* del progetto iOS dell'applicazione.

- *MainStoryboard.xpt*: è un template di fondamentale importanza utilizzato per generare la Storyboard, un file con una struttura simile a XML

che definisce l'interfaccia grafica di tutti i ViewController dell'applicazione. In Xcode viene aperto con lo strumento Interface Builder, un editor grafico molto comodo per la definizione della GUI dell'applicazione. La Storyboard in iOS corrisponde in Android all'insieme di tutti i file XML utilizzati per definire il layout delle Activity.
Anche in questo caso il contenuto del file generato dipende dagli elementi del metamodello che sono stati inseriti all'interno del ViewController al momento della definizione dell'istanza e gli elementi grafici vengono automaticamente disposti nelle Window uno sotto l'altro, facilitando il lavoro di restyling grafico successivo alla fase di generazione di codice.

- *Xcodeproj.xpt*: è un template creato con lo scopo di generare il file *project.pbxproj*, un file contenuto nella cartella Xcodeproj presente in tutti i progetti sviluppati in Xcode. Questo file ha lo scopo principale di dichiarare tutti i file .h e .m, i framework e le risorse presenti nel progetto e la loro effettiva posizione all'interno dei sorgenti. Il file inoltre definisce la versione iOS dell'applicazione sviluppata, che nel nostro caso corrisponde alla versione 6, e i dispositivi supportati.

- *InfoPlist.xpt e PrefixPch.xpt*: sono i template che permettono di generare il file *Info.plist* per l'applicazione, che contiene informazioni come il nome dell'applicazione e l'identificativo dell'azienda sviluppatrice, e il file *Prefix.pch*, che contiene invece le informazioni relative all'SDK adottato. Questi due file, insieme a project.pbxproj e ai file predefiniti che vengono copiati dalla cartella utils del generatore, sono necessari per fare in modo che i sorgenti generati possano essere aperti senza produrre errori di compilazione in Xcode.

- *Utils.xpt*: template utilizzato per creare una classe Objective-C che definisce una View contenente un testo. Questa classe viene usata per definire il contenuto delle celle delle GridView, che in iOS corrispondono a UICollectionViewController e viene generata ed utilizzata nel codice solo se l'utente inserisce nel modello dell'applicazione almeno un elemento GridView del metamodello.

- *ImageDownloadingTask.xpt*: corrisponde al template ImageDownloadingTask.xpt presente nella cartella android_templates. Allo stesso modo, il codice per il download asincrono delle immagini viene generato solo nel caso in cui siano presenti una o più ImageView nel modello dell'applicazione.

Sottocartelle *android_extensions*, *ios_extensions* e *app_extensions*

Nelle sezioni precedenti abbiamo fatto riferimento a queste cartelle, create con lo scopo di contenere i file Java con i metodi invocabili dai template. Per questa ragione, oltre ai file Java le cartelle contengono anche i file Xtend necessari per gestire l'interfacciamento con i template Xpand. Come si può intuire dal nome, le cartelle *android_extensions* e *ios_extensions* sono state definite per contenere i file Java di supporto alla generazione dei sorgenti Android e iOS rispettivamente. La directory *app_extensions* contiene invece i file Java con i metodi invocati dai template sia al momento della generazione dei sorgenti Android che di quelli iOS.

Sottocartella *workflow*

All'interno di questa cartella ci sono i file MWE, che determinano il workflow che il generatore deve seguire per generare correttamente i sorgenti corrispondenti al modello definito dall'utente. Come è possibile vedere in Figura 5.12,

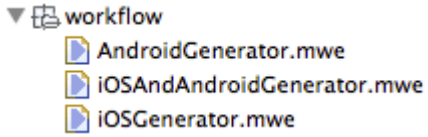

Figura 5.12: *Generatore di codice*: cartella workflow

abbiamo preferito creare tre file MWE per dare la possibilità all'utilizzatore di scegliere per quale piattaforma generare i sorgenti: eseguendo il file *AndroidGenerator.mwe* vengono generati i sorgenti per la piattaforma Android, eseguendo il file *iOSGenerator.mwe* vengono generati i sorgenti per la piattaforma iOS, con il file *iOSAndAndroidGenerator.mwe* vengono invece generati nello stesso momento i sorgenti per entrambe le piattaforme mobili. In tutti e tre i casi il generatore legge il file .xmi dell'applicazione e, dopo aver validato il modello verificando i vincoli elencati nel file Checks.chk, sfrutta i template Xpand e i file a essi relazionati per generare l'output richiesto.
In Figura 5.13 è riportato uno schema generale del generatore di codice che chiarisce il ruolo di mediazione rivestito dal Workflow Engine, il quale, come anticipato in precedenza, integra e processa i diversi componenti del progetto sulla base del file .mwe mandato in esecuzione. Al momento dell'utilizzo del generatore per lo sviluppo dell'applicazione desiderata lo sviluppatore si dovrà preoccupare unicamente di definire il modello, generando il file .xmi, e lanciare il Workflow Engine.
I sorgenti delle applicazioni generate vengono salvati all'interno di una nuova cartella *src-gen*, che viene creata in automatico nella root del progetto nel

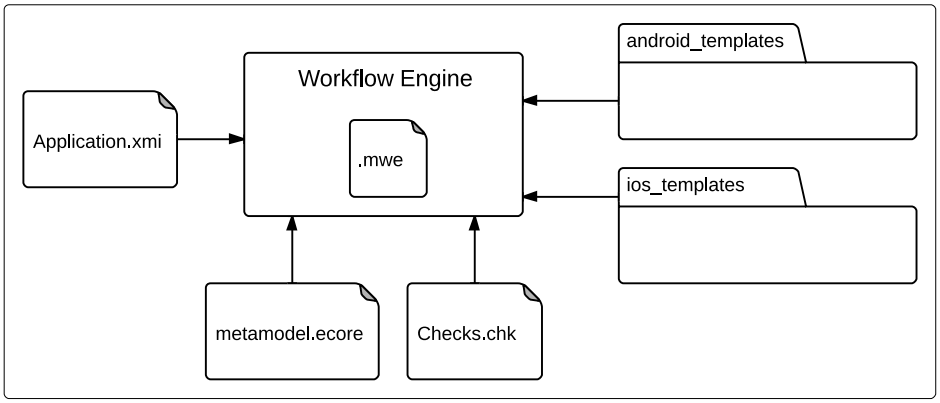

Figura 5.13: *Generatore di codice*: schema generale dei componenti interni

caso in cui non sia già presente. All'interno di questa cartella si trovano due directory *android* e *ios*, create appositamente per distinguere i sorgenti generati per la piattaforma Android da quelli generati per la piattaforma iOS. In Figura 5.14 è riportato un esempio che chiarisce la struttura della cartella src-gen, dove sono stati salvati i sorgenti Android e iOS generati per l'applicazione Valtellina descritta in precedenza.

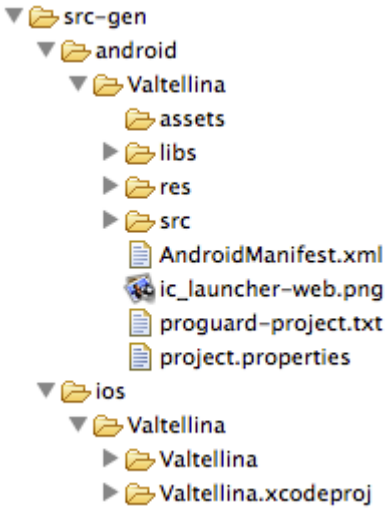

Figura 5.14: *Generatore di codice*: cartella src-gen

5.2.2 Cartella *utils*

All'interno di questa cartella si trovano due sottocartelle molto importanti nel processo di generazione del codice, chiamate *android_default_files* e

ios_default_files. Tali cartelle includono i file statici che devono essere necessariamente inseriti nei sorgenti delle applicazioni per fare in modo che il progetto generato possa essere importato, senza produrre errori di compilazione, in Eclipse nel caso di Android e Xcode nel caso di iOS. Tra questi ci sono diversi file di configurazione, le librerie di sistema e l'icona di default dell'applicazione.
Oltre a queste due sottocartelle si trova la directory *user_files*, la cui utilità è quella di permettere allo sviluppatore di inserire le risorse che devono essere copiate all'interno dell'applicazione, come anticipato in precedenza.

5.3 Corretta definizione del modello

Nel paragrafo precedente abbiamo descritto la struttura e il funzionamento del generatore di codice. In questa sezione forniamo al lettore le notazioni, le convenzioni e i vincoli che dovrebbero essere seguiti durante la definizione del modello dell'applicazione che, come spiegato in precedenza, si concretizza nella creazione di un file .xmi, input del processo di traduzione del modello in codice.

5.3.1 Notazioni e Convenzioni

Per evitare di generare sorgenti contenenti errori di sintassi al momento dell'importazione negli ambienti di sviluppo, consigliamo agli sviluppatori che vorranno usare il generatore di codice di attenersi alle notazioni e convenzioni riportate di seguito. Le notazioni stabilite sono in linea con quelle previste nell'ambito dello sviluppo di applicazioni Android e iOS.

- Gli id di tutti gli elementi devono cominciare con la lettera minuscola e terminare con il nome dell'elemento di cui sono istanza. Nel caso in cui il nome fosse composto da più parole si consiglia di adottare una notazione CamelCase, che prevede il concatenamento delle diverse parole, lasciando le loro iniziali maiuscole.

- I nomi degli elementi ViewController devono iniziare con la lettera maiuscola.

- L'attributo name dell'elemento AsyncTask deve cominciare con la lettera minuscola.

- Tutti i file locali referenziati da WebView, ImageView, VideoView e AudioPlayer vanno indicati nel modello con la classica notazione nome.estensione ed inseriti nella cartella user_files.

- I file CSS che dovranno essere usati nelle WebView o che si vorranno caricare nel progetto della propria applicazione devono essere inseriti nella cartella user_files e vanno linkati dall'interno dei file HTML che li referenziano.

- Per quanto riguarda gli elementi ImageView, VideoView e AudioPlayer, nel caso in cui lo sviluppatore non abbia ancora a disposizione il file locale o l'url del file remoto da referenziare, deve lasciare l'attributo fileUri non settato. Lo stesso discorso vale per l'attributo HTMLFileName di WebView.

- Gli attributi testuali (contenuti, titoli, etc) degli elementi del metamodello devono presentare testi privi di caratteri speciali, altrimenti il generatore può produrre dei sorgenti contenenti errori di sintassi.

5.3.2 Vincoli

In questa sezione presentiamo i vincoli che devono necessariamente essere soddisfatti dallo sviluppatore al momento della definizione del modello dell'applicazione. Tali vincoli sono verificati dal generatore di codice secondo quanto riportato nel file Checks.chk analizzato in precedenza in questo capitolo. Ricordiamo che i vincoli che non vengono rispettati determinano l'interruzione della compilazione e l'apparizione di un messaggio di errore sulla console di Eclipse.

- Il nome dell'applicazione può contenere solo lettere minuscole e maiuscole, quindi deve rispettare l'espressione regolare ^[a-zA-Z]+.

- L'attributo companyIdentifier di Application deve rispettare l'espressione regolare ^[a-z]+(.[a-z]+)*.

- L'attributo name dell'elemento ViewController deve rispettare l'espressione regolare ^[a-zA-Z][a-zA-Z0-9_]*.

- I nomi degli elementi ViewController devono essere univoci.

- Gli id devono rispettare l'espressione regolare ^[a-zA-Z][a-zA-Z0-9_]*.

- Gli id devono essere univoci.

- Ci deve essere un unico ViewController il cui attributo launcher è settato a true, in quanto può esistere un unico ViewController mostrato all'apertura dell'applicazione.

- Tutti gli elementi Navigation devono avere l'attributo destination settato correttamente, indicando il nome di un ViewController esistente.
- L'attributo imageViewId dell'elemento PhotocameraController deve essere settato con l'id di un elemento ImageView esistente con attributo sourceType=hardwareFile, altrimenti non deve essere settato.
- L'attributo videoViewId dell'elemento VideocameraController deve essere settato con l'id di un elemento VideoView esistente con attributo sourceType=hardwareFile, altrimenti non deve essere settato.
- L'attributo audioPlayerId dell'elemento AudioRecorder deve essere settato con l'id di un elemento AudioPlayer esistente con attributo sourceType=hardwareFile, altrimenti non deve essere settato.
- L'attributo fileUri degli elementi ImageView, VideoView e AudioPlayer non deve essere settato se l'attributo sourceType è impostato ad "hardwareFile".
- L'attributo fileUri dell'elemento ImageView, se settato, deve corrispondere al nome di un file con estensione png, jpg o gif.
- L'attributo fileUri dell'elemento VideoView, se settato, deve corrispondere al nome di un file con estensione mp4 o 3gp.
- L'attributo fileUri dell'elemento AudioPlayer, se settato, deve corrispondere al nome di un file con estensione mp3.
- L'attributo HTMLFileName dell'elemento WebView, se settato, deve corrispondere al nome di un file con estensione html.
- I nomi degli elementi AsyncTask devono avere almeno un carattere ed essere univoci.
- Nel caso in cui l'elemento Menu contenga degli elementi MenuItem, il primo MenuItem deve avere come destinazione il ViewController iniziale.
- Per ogni elemento MenuItem deve essere settata la relazione con l'elemento Navigation, che a sua volta deve avere impostata una destinazione valida, vale a dire il nome di un ViewController esistente.
- La cartella user_files non deve contenere altre cartelle e i nomi dei file contenuti in essa devono rispettare l'espressione regolare [a-z0-9_] e contenere l'estensione.

- Le estensioni ammesse per i file (visibili e nascosti) presenti nella cartella user_files sono html, txt, css, png, jpg, gif, mp4, 3gp e mp3.
- I file all'interno della cartella user_files non possono avere nomi uguali, nemmeno se presentano estensioni diverse.

5.4 Codice generato

Avendo considerato la struttura del generatore e le regole da seguire per il suo funzionamento, rimane da analizzare l'output del processo di generazione del codice. In questo paragrafo presentiamo, in maniera sintetica, il contenuto dei file sorgenti generati a fronte di ciascuno degli elementi del modello dell'applicazione, in modo da rendere chiara la struttura del progetto generato allo sviluppatore e permettergli di rivedere e modificare i sorgenti prodotti per un particolare elemento del modello. Vedremo in particolare gli elementi che compongono l'interfaccia grafica, per poi passare a quelli che interagiscono direttamente con i componenti hardware del dispositivo.

5.4.1 Interfaccia grafica

Le tabelle seguenti mostrano l'output della fase di traduzione del modello in codice per gli elementi del metamodello implementativo che costituiscono l'interfaccia grafica.

Elemento	**Android**	**iOS**
Button	L'elemento viene dichiarato e settato nel metodo onCreate dell'Activity di appartenenza, dove viene collegato al relativo layout XML attraverso il suo id. Nel caso in cui sia stato indicato il ClickListener viene inserito il metodo onClickListener, nel quale si trova, laddove sia definita una navigazione, l'intent per il passaggio all'Activity destinazione.	Il bottone viene creato all'interno della Storyboard. Se è stato definito un ClickListener senza navigazione nel file .m del ViewController di appartenenza viene inserito un metodo per la definizione dell'azione da compiere, al contrario se è prevista una navigazione viene inserito un segue direttamente nella Storyboard.
TextView	L'elemento è definito e settato nel layout XML dell'Activity di appartenenza.	L'elemento è definito e settato nella Storyboard.
EditText	L'elemento viene dichiarato e settato nel metodo onCreate dell'Activity di appartenenza, dove viene collegato al relativo layout XML attraverso il suo id.	L'elemento viene settato nella Storyboard e collegato al file .m del ViewController di appartenenza attraverso un outlet.

Tabella 5.1: *Interfaccia grafica*: output del generatore - parte 1

Elemento	Android	iOS
WebView	L'elemento viene dichiarato e settato nel metodo onCreate dell'Activity di appartenenza.	L'elemento viene dichiarato e settato nel metodo viewDidLoad del ViewController di appartenenza.
AlertDialog	La logica applicativa viene gestita tutta nel metodo onCreate dell'Activity di appartenenza, dove viene istanziato l'elemento, vengono settati il titolo e il messaggio, vengono inseriti i metodi per la gestione dell'evento di click dei bottoni che lo compongono e viene inserito il codice per renderlo visibile sullo schermo.	Oltre alla dichiarazione dell'elemento tra le property nel file .m del ViewController e alla LazyInstantiation, viene implementato un metodo delegate, dove è richiesta l'implementazione delle azioni da svolgere al momento della pressione di uno dei due bottoni che lo compongono.
ProgressDialog	L'elemento è gestito nel metodo onCreate dell'Activity di appartenenza, nel quale viene settato titolo, messaggio e tipologia (spinner o progress bar). Per come viene generato il codice, il ProgressDialog viene dismesso immediatamente per evitare il blocco della UI.	L'elemento è gestito all'interno del metodo viewDidLoad del ViewController di appartenenza, dove viene settato e immediatamente dismesso, come nel caso Android.
ImageView	Il settaggio dell'elemento avviene nel file XML o nell'Activity di appartenenza a seconda del valore dell'attributo sourceType. Nel caso in cui l'immagine provenga da un file locale l'elemento viene settato direttamente nel file XML, altrimenti viene settato nell'Activity. Nel caso di provenienza da remoto l'immagine viene prima scaricata attraverso un task asincrono, nel caso di provenienza dalla fotocamera la logica applicativa di settaggio dell'ImageView è definita nel PhotocameraController.	L'immagine viene settata nella Storyboard se il file da cui proviene è locale, al contrario viene settata nel metodo viewDidLoad del ViewController di appartenenza. Nel caso di provenienza da remoto l'immagine viene prima scaricata dal server in maniera asincrona, nel caso di provenienza dalla fotocamera la logica applicativa di settaggio dell'ImageView è definita nel PhotocameraController.
VideoView	L'elemento viene settato nel metodo onCreate dell'Activity di appartenenza se il file referenziato è locale o remoto. Se il video proviene dalla videocamera del dispositivo la logica applicativa di settaggio della VideoView è definita al momento della traduzione in codice dell'elemento VideocameraController.	L'elemento viene settato nel metodo viewDidAppear del ViewController di appartenenza se il file referenziato è locale o remoto. In caso contrario la logica applicativa di settaggio della VideoView è definita al momento della traduzione in codice dell'elemento VideocameraController.

Tabella 5.2: *Interfaccia grafica*: output del generatore - parte 2

Elemento	Android	iOS
GridView	L'elemento viene dichiarato e settato nel metodo onCreate dell'Activity di appartenenza, dove viene collegato al relativo layout XML attraverso il suo id. Nell'Activity vengono definiti un array e un adapter per memorizzare e mostrare a schermo i contenuti delle celle. Nel caso in cui sia stato indicato il ClickListener viene implementata l'interfaccia OnItemClickListener, facendo l'override del metodo onItemClick, nel quale si trova, laddove sia definita una navigazione, l'intent per il passaggio all'Activity destinazione.	L'elemento viene inserito nella Storyboard e nel file .h del ViewController vengono inclusi i protocolli UICollectionViewDelegate e UICollectionViewDataSource, mentre nel file .m viene dichiarato l'array per memorizzare i contenuti, il quale viene istanziato solo se sono state definite delle celle. Nel file .m vengono implementati tutti i metodi necessari per la gestione della CollectionView, tra i quali quello per la gestione dell'azione da compiere al momento della selezione di una cella.
GridViewCell	Gli elementi GridViewCell collegati ad un particolare elemento GridView contribuiscono, attraverso il loro attributo title che li contraddistingue, alla definizione dell'array dei contenuti di GridView.	Per questo elemento viene creata una classe apposita chiamata MyCollectionViewCell al cui interno si trova la dichiarazione di una stringa, perché in ambiente iOS non è previsto un prototipo di default per le celle delle GridView. Così come per Android, l'attributo testuale della singola cella contribuisce alla definizione dell'array dei contenuti di GridView.
ListView	Cosi come per l'elemento GridView, vengono definiti nell'Activity di appartenenza un adapter ed un array per memorizzare e mostrare a schermo i contenuti delle celle. Nel caso in cui sia stato indicato il ClickListener viene implementata l'interfaccia OnItemClickListener, facendo l'override del metodo onItemClick, nel quale si trova, laddove sia definita una navigazione, l'intent per il passaggio all'Activity destinazione.	L'elemento viene inserito nella Storyboard e nel file .h del ViewController vengono inclusi i protocolli UITableViewDelegate e UITableViewDataSource, mentre nel file .m viene dichiarato l'array per memorizzare i contenuti. Nel file .m vengono implementati tutti i metodi necessari per la gestione della TableView, tra i quali quello per la gestione dell'azione da compiere al momento della selezione di una cella.
ListViewCell	Così come GridViewCell, gli elementi ListViewCell collegati ad un particolare elemento ListView contribuiscono, attraverso il loro attributo title che li contraddistingue, alla definizione dell'array dei contenuti di ListView.	Così come per Android, l'attributo testuale della singola cella contribuisce alla definizione dell'array dei contenuti della ListView. A differenza di GridViewCell, per ListViewCell non è necessario creare alcuna classe Objective-C, in quanto è già previsto un prototipo di default per le celle delle ListView.

Tabella 5.3: *Interfaccia grafica*: output del generatore - parte 3

5.4.2 Hardware

Le tabelle seguenti mostrano l'output della fase di traduzione del modello in codice per gli elementi del metamodello implementativo che sfruttano l'hardware del dispositivo sulla quale l'applicazione è installata.

Elemento	Android	iOS
AudioPlayer	Nel file XML del layout dell'Activity di appartenenza vengono inseriti tre bottoni per avviare, fermare e mettere in pausa la riproduzione del file audio. Nel metodo onResume dell'Activity avviene invece il settaggio del file da riprodurre e la gestione delle azioni da compiere al momento del click dei tre bottoni.	Nella Storyboard sono definiti i tre bottoni Play, Stop e Pause, mentre le loro azioni sono definite nei metodi delegate associati ai bottoni all'interno del file .m del ViewController di appartenenza. Il settaggio del file da riprodurre avviene nel metodo viewDidAppear.
AudioRecorder	Nel file XML del layout dell'Activity di appartenenza viene inserito un bottone per avviare o fermare la registrazione. La logica applicativa di registrazione e salvataggio in memoria del file registrato viene implementata nel metodo onResume dell'Activity. Nel metodo onResume viene anche settato l'AudioPlayer nel caso in cui all'AudioRecorder nel modello sia associato un AudioPlayer con attributo sourceType=hardwareFile.	Nella Storyboard viene definito un bottone per avviare o fermare la registrazione, la cui logica applicativa è implementata nel metodo record associato al bottone all'interno del file .m del ViewController di appartenenza. In questo metodo viene gestita in particolare la registrazione e il salvataggio del file registrato nella memoria del dispositivo, oltre al settaggio dell'eventuale AudioPlayer.
LocationManager	La logica applicativa di recupero della posizione del dispositivo attraverso il GPS viene implementata nel metodo onCreate dell'Activity di appartenenza, dove viene implementata l'interfaccia LocationListener. Vengono aggiunte infatti le dichiarazioni dei metodi onLocationChanged, onProviderDisabled, onProviderEnabled e onStatusChanged, la cui implementazione viene però lasciata allo sviluppatore.	Nel file .h viene incluso il protocollo UINavigationControllerDelegate e i suoi metodi sono implementati nel file .m del ViewController di appartenenza, dove risiede tutta la logica applicativa: nel metodo viewDidLoad viene recuperata la posizione dell'utente facendo uso del GPS.

Tabella 5.4: *Hardware*: output del generatore - parte 1

Elemento	Android	iOS
PhotocameraController	Nel metodo onCreate dell'Activity di appartenenza viene creato un bottone che conduce all'apertura dell'applicazione Android sul dispositivo che consente di scattare una foto. Nell'Activity viene implementato il metodo onActivityResult, dove viene ricevuta l'immagine proveniente dalla fotocamera. In questo metodo viene settata l'eventuale ImageView associata al PhotocameraController e salvata l'immagine nella memoria interna del dispositivo.	Nella Storyboard viene creato un bottone che conduce all'apertura dell'applicazione iOS che consente di scattare una foto. Il file .h del ViewController di appartenenza include il protocollo UIImagePickerControllerDelegate, mentre nel file .m viene inserita l'implementazione dei suoi metodi che consentono di gestire la foto ricevuta, come salvarla in memoria o usarla per settare l'eventuale ImageView associata al PhotocameraController.
VideocameraController	Come per l'elemento PhotocameraController, viene creato un bottone nel layout dell'Activity di appartenenza che permette l'apertura dell'applicazione Android che consente la registrazione di un filmato. Nell'Activity viene implementato il metodo onActivityResult, per il salvataggio in memoria del video ricevuto e il settaggio dell'eventuale VideoView associata.	Nella Storyboard viene creato un bottone che conduce all'apertura dell'applicazione iOS che consente di registrare un filmato. Nel file .h viene incluso il protocollo UIImagePickerControllerDelegate, mentre nel file .m viene inserita l'implementazione dei suoi metodi che consentono di gestire il video ricevuto, come salvarlo in memoria o usarlo per settare l'eventuale VideoView associata al VideocameraController.

Tabella 5.5: *Hardware*: output del generatore - parte 2

5.4.3 Altri elementi

Per quanto riguarda gli elementi non appartenenti all'interfaccia grafica o all'hardware, vediamo qual è l'output della fase di traduzione del modello in codice per AsyncTask e Menu.

L'elemento *AsyncTask* comporta, in entrambi i casi Android e iOS, l'aggiunta nel ViewController di appartenenza dell'intestazione di un metodo con il nome indicato dallo sviluppatore nel modello. L'implementazione della logica applicativa interna al metodo viene lasciata allo sviluppatore.

L'elemento *Menu* prevede, nei sorgenti Android, la creazione di una classe MenuActivity, che verrà poi estesa dalle altre Activity dell'applicazione. A seconda dei vari elementi MenuItem collegati a Menu la classe creata comprende la definizione degli intent necessari per avviare la navigazione verso le Activity di destinazione. Per quanto concerne iOS invece la presenza dell'elemento Menu comporta l'inserimento di una UITabBarController nella Storyboard e di tante UINavigationController quanti sono i MenuItem relazionati al Menu: in questo caso la logica applicativa di gestione del Menu è quindi interamente definita nella Storyboard.

5.4.4 Problemi riscontrati

Durante le fasi di implementazione e testing del generatore di codice abbiamo utilizzato di frequente i tool di sviluppo Eclipse, per le applicazioni Android, e Xcode, per le applicazioni iOS. In particolare abbiamo riscontrato in Eclipse due problemi abbastanza rilevanti che preferiamo riportare nel caso si manifestassero dopo l'importazione dei sorgenti dell'applicazione generata nell'ambiente di sviluppo.
Il primo problema si verifica nel caso in cui per una certa Activity vengano definiti un numero eccessivo di widget: secondo la nostra logica di posizionare gli elementi in maniera lineare nel layout può succedere che alcuni elementi non siano visibili sullo schermo. Lanciare l'applicazione senza aver prima riposizionato gli elementi nel file XML può determinare il verificarsi di un arresto inatteso.
Il secondo problema, invece, può accadere nel momento in cui all'interno di una stessa Activity siano presenti sia un elemento ImageView che un elemento VideoView. Questa situazione può condurre ad un errore di compilazione al momento dell'importazione dei sorgenti in Eclipse. Il problema, già noto all'interno della community Eclipse, può essere risolto cancellando il file R ed effettuando una clean del progetto Android dell'applicazione.

5.5 Conclusioni

In questo capitolo abbiamo presentato gli strumenti principali che sono stati considerati in fase di implementazione e, dopo essere entrati nel merito della loro utilità e del loro funzionamento, abbiamo fornito una descrizione generale della struttura del progetto del generatore di codice, così da fornire una visione ad alto livello di quanto è stato prodotto. Utilizzando una distribuzione di Eclipse contenente Epsilon e installando alcuni plugin siamo riusciti ad avere un ambiente di lavoro compatto e completo in termini di strumenti

necessari per soddisfare tutte le esigenze implementative volte alla definizione del metamodello, alla generazione di una sua istanza e alla traduzione di quest'ultima in codice. Oltre alla descrizione della struttura, abbiamo fornito le regole e le notazioni da seguire per il corretto utilizzo del generatore di codice da parte degli sviluppatori e ne abbiamo presentato l'output, analizzando il codice prodotto per ogni elemento che è possibile inserire nel modello delle applicazioni.

Nel prossimo capitolo presenteremo alcuni esempi che dimostrano la possibilità di utilizzare il generatore di codice ai fini di produrre la stessa applicazione per le piattaforme Android e iOS, a partire da un modello comune. Inoltre, verrà effettuata una valutazione quantitativa e qualitativa del risparmio, in termini di effort e tempistiche richieste allo sviluppatore, per completare un'applicazione prodotta attraverso il generatore di codice.

Capitolo 6

Valutazione

Nei capitoli precedenti abbiamo presentato le due versioni, astratta e implementativa, del metamodello per le applicazioni mobili, gli strumenti software utilizzati in fase di implementazione, la struttura e il funzionamento ad alto livello del progetto realizzato. In questo capitolo vediamo come sfruttare il nostro lavoro per produrre alcune istanze del metamodello implementativo, ciascuna delle quali finalizzata a modellare una particolare applicazione. A partire dalle singole istanze vedremo come è possibile ottenere i sorgenti Android e iOS corrispondenti e come importarli negli ambienti di sviluppo Eclipse e Xcode rispettivamente. L'obiettivo principale di questa fase, comunque, non è quello di mostrare al lettore come utilizzare lo strumento, quanto piuttosto fornire una valutazione, quantitativa e qualitativa, del codice prodotto attraverso il generatore e quindi dell'efficacia di quest'ultimo nel mondo dello sviluppo di applicazioni mobili.

6.1 Valutazione quantitativa

In questa sezione daremo una breve descrizione del processo di modellazione e generazione dei sorgenti per due applicazioni pensate da noi, chiamate Valtellina e Multimedia. Per quanto riguarda la prima, è già stata fornita una descrizione generica della sua struttura nel terzo capitolo, quando abbiamo generato un'istanza del metamodello astratto. La seconda, citata solamente nel capitolo precedente, è un'applicazione incentrata sull'utilizzo dell'hardware del dispositivo per produrre e rendere accessibili all'utente finale diversi contenuti multimediali.
Al termine della presentazione delle due applicazioni forniremo una valutazione quantitativa del codice generato grazie alla metrica del conteggio del numero di righe di codice sorgente (SLOC). Questo metodo di misura è il più indicato per stabilire la complessità di un software e per stimare le risorse necessarie per la sua produzione e il suo mantenimento. Quello che faremo,

per ciascuna delle due versioni Android e iOS delle applicazioni Valtellina e Multimedia, sarà svolgere due conteggi distinti, uno inerente le righe di codice fisiche (Physical SLOC) e l'altro relativo alle righe di codice logiche (Logical SLOC). Nei conteggi di tipo Physical SLOC si considerano tutte le righe di testo del codice sorgente, includendo anche i commenti e le righe vuote, al contrario nei conteggi di tipo Logical SLOC si considerano gli statement, ovvero il numero di istruzioni effettive, indipendentemente dalla loro collocazione su una o su più righe.

6.1.1 Applicazione Valtellina

Prima di procedere con la generazione del file .xmi ci siamo preoccupati di definire una versione più concreta del modello dell'applicazione Valtellina presentato nel capitolo 3, istanziando questa volta la versione implementativa del metamodello. A partire dal nuovo modello, mostrato in Figura 6.1, è infatti possibile definire in maniera univoca il contenuto del file .xmi e quindi la struttura dell'applicazione che intendiamo realizzare.
Ai fini di modellare l'applicazione attraverso il generatore di codice nell'ambiente di sviluppo Eclipse, innanzitutto è necessario generare il file .xmi: apriamo quindi il file *metamodel.ecore* presente nel package *metamodel* del progetto e, con il tasto destro del mouse, facciamo click su *Application* e selezioniamo *Create Dynamic Istance*. Si aprirà una finestra che ci permetterà di stabilire il nome e in quale cartella del progetto intendiamo salvare il modello dell'applicazione che stiamo per definire: selezioniamo quindi la cartella *model* e indichiamo *Valtellina.xmi* come filename. Una volta creato, il file verrà automaticamente aperto, e sarà composto dall'unico elemento Application, a partire dal quale possiamo iniziare ad inserire gli elementi del metamodello per delineare la struttura dell'applicazione.
Attraverso la finestra *Properties* di Eclipse è possibile settare tutti gli attributi degli elementi che vengono man mano inseriti nel modello. Per l'elemento Application ad esempio è possibile settare *companyIdentifier* e *name* che, secondo quanto riportato in figura, assumono i valori "it.polimi" il primo e "Valtellina" il secondo. L'aggiunta degli elementi nel modello avviene sulla base delle composizioni indicate nel metamodello implementativo, a partire dall'elemento root Application, facendo click con il tasto destro del mouse sull'elemento da cui ha origine la composizione, selezionando *New Child* e quindi il nome del componente che si intende aggiungere. Per l'applicazione Valtellina, partendo da Application, è infatti possibile, attraverso la composizione *viewControllers*, definire i quattro ViewController che compongono l'applicazione, oltre ad aggiungere il menu. Sfruttando l'apposito editor grafico integrato nella distribuzione di Eclipse utilizzata, è infatti possibile

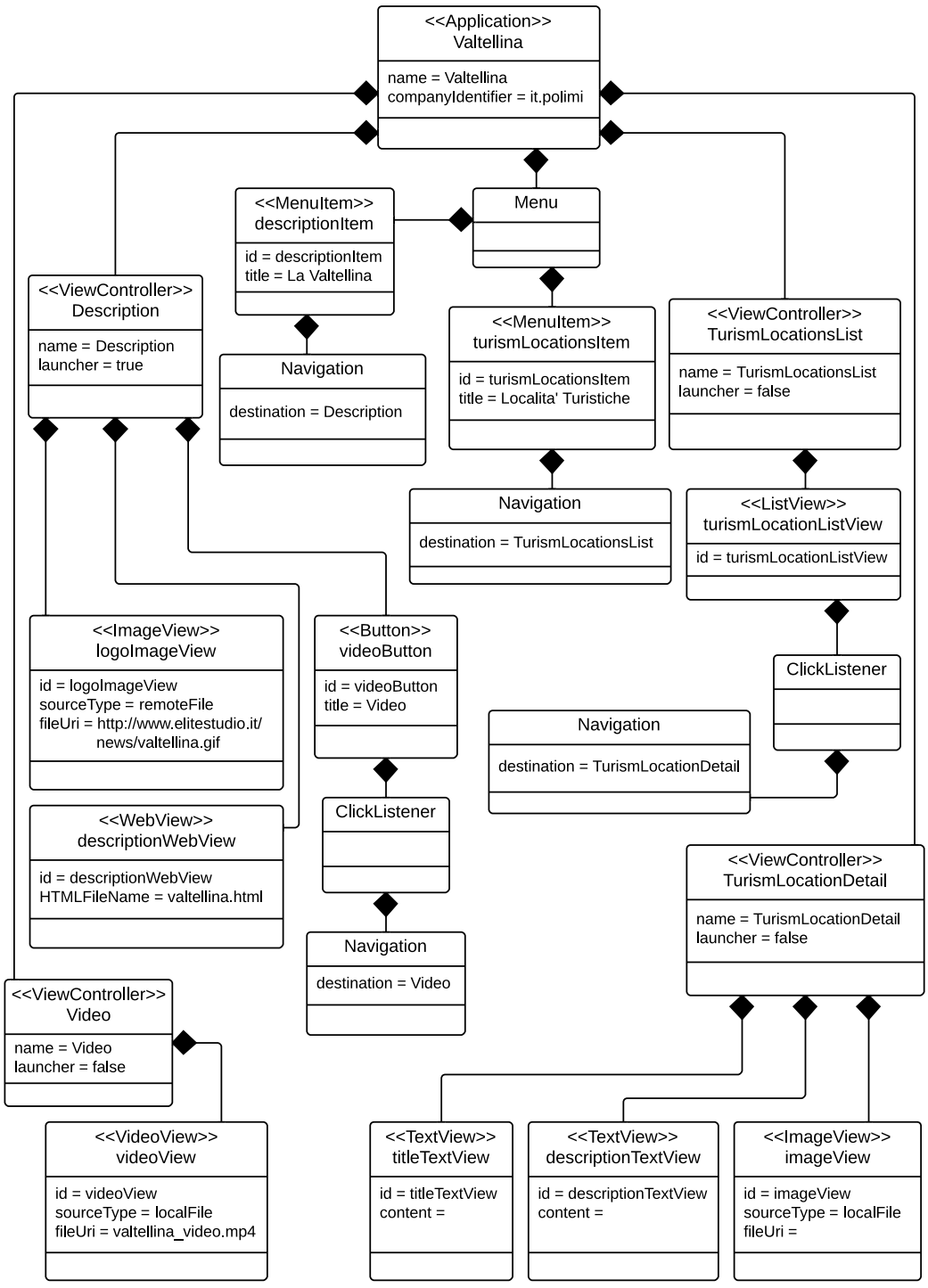

Figura 6.1: *Valtellina*: istanza del metamodello implementativo

definire tutti gli elementi dell'istanza riportati in Figura 6.1, producendo così la versione finale del file Valtellina.xmi.
Per questioni di spazio non abbiamo riportato in figura gli elementi ListViewCell di cui è composta la ListView del ViewController TurismLocationsList. Per evitare di doverle inserire manualmente via codice completiamo il file .xmi, aggiungendone una per ciascuna delle località turistiche che intendiamo mostrare, perciò sfruttando la relazione listViewCells su turismLocationListView definiamo undici ListViewCell, a cui assegniamo rispettivamente come *title* i seguenti nomi di località: "Aprica", "Bormio", "Chiesa Valmalenco", "Livigno", "Madesimo", "Santa Caterina Valfurva", "Teglio", "Tirano", "Val Masino", "Piantedo" e "Dazio". Per ciascuna di queste inseriamo nella cartella user_files un file .txt, contenente una descrizione generica della specifica località, e un'immagine da mostrare nel ViewController di dettaglio di ogni località.

Dopo essersi accertati di aver inserito nell'apposita directory user_files tutte le risorse referenziate dagli elementi dell'istanza (file .txt ed immagini per le celle della turismLocationListView, file valtellina.html per la descriptionWebView e file valtellina_video.mp4 per la videoView), la fase successiva è quella di traduzione del modello in codice, operazione che può essere avviata lanciando il workflow engine, ovvero facendo click con il tasto destro del mouse su uno dei file .mwe presenti nella cartella workflow e selezionando *Run As MWE Workflow* dal menu a tendina che appare sullo schermo. Prima di avviare la generazione di codice bisogna configurare il file MWE in modo che legga il modello dell'applicazione di cui si desidera generare i sorgenti. In base al file .mwe scelto verranno generati i sorgenti nativi dell'applicazione per la piattaforma Android, per la piattaforma iOS o per entrambe. Perciò, dopo aver scelto quale file utilizzare, è necessario aprire tale file e cambiare il percorso che porta alla lettura del modello modificando il valore della property *model* che si trova nelle prime righe. Ai fini di verificare il corretto funzionamento del generatore di codice e valutarne la sua utilità scegliamo di produrre i sorgenti per entrambe le piattaforme, utilizzando il file *iOSAndAndroidGenerator.mwe*: apriamo quindi tale file e settiamo il percorso del file Valtellina.xmi nella proprietà model.

Ora che abbiamo presentato a livello generale l'approccio model-driven allo sviluppo dell'applicazione Valtellina, forniamo una valutazione quantitativa dello strumento, confrontando il codice prodotto dal generatore con quello che, a seguito di modifiche manuali, definisce l'applicazione finale. Come anticipato in precedenza effettueremo una valutazione distinta per le versioni Android e iOS dell'applicazione.

Valutazione della versione Android

Al momento dell'importazione nell'ambiente di sviluppo Eclipse dei sorgenti Android prodotti dal generatore di codice, emerge la necessità di perfezionare il layout e la grafica dei ViewController. In particolare lo sviluppatore dovrà riposizionare e modificare, secondo le proprie preferenze, gli elementi dell'interfaccia grafica, modificando il contenuto dei file .xml che definiscono il layout dell'applicazione. Oltre a questo genere di modifiche, ai fini di completare le funzionalità che l'applicazione dovrà offrire all'utente finale, il programmatore dovrà implementare le parti algoritmiche mancanti. Nel caso in analisi abbiamo preferito infatti trascurare la ridefinizione dell'interfaccia grafica, che può essere personalizzata rapidamente con l'editor grafico reso disponibile dal plugin ADT di Eclipse, incentrandoci così sul completamento della logica applicativa mancante.

Per l'applicazione Valtellina possiamo considerare completamente funzionali i ViewController Description e Video, in quanto il contenuto e il layout del primo sono definiti principalmente attraverso il file valtellina.html, copiato dal generatore all'interno dei sorgenti e referenziato dalla WebView interna al ViewController, mentre il secondo ospita una VideoView che riproduce un video salvato in locale. Oltre alla revisione dei layout dei ViewController rimane da settare il contenuto delle due TextView e dell'ImageView presenti all'interno del ViewController TurismLocationDetail, sulla base dell'elemento che è stato cliccato nella ListView del ViewController TurismLocationsList. Perciò abbiamo memorizzato in due array tutti i file di testo e le immagini relative alle località turistiche, quindi abbiamo implementato la logica che permette di passare titolo, testo e immagine della località turistica selezionata dalla lista al ViewController di dettaglio. All'interno di quest'ultimo abbiamo dovuto poi aggiungere manualmente il codice Java necessario a settare dinamicamente i contenuti delle due TextView e dell'ImageView, sulla base delle informazioni passate.

Nella Tabella 6.1 sono riportati i valori delle due misure Physical SLOC e Logical SLOC per i file Java, ovvero quelli che contengono la logica applicativa dell'applicazione, del progetto Valtellina prodotto dal generatore di codice. In Tabella 6.2 sono riportati i valori degli stessi file in seguito alle modifiche effettuate nell'ambiente di sviluppo Eclipse per completare l'applicazione: di fianco ad ogni conteggio è riportata anche la percentuale di codice coperta da ciascun file prodotto dal generatore di codice. Per citare un esempio, il file *DescriptionActivity.java* presenta una copertura del 100% sia per quanto riguarda le Physical SLOC che le Logical SLOC, in quanto i valori misurati nel file autogenerato e nel file completo coincidono.

File	Physical SLOC	Logical SLOC
DescriptionActivity.java	81	48
ImageDownloadingTask.java	75	39
MenuActivity.java	30	20
TursimLocationDetailActivity.java	52	32
TurismLocationListActivity.java	79	46
Utils.java	23	15
VideoActivity.java	61	39

Tabella 6.1: *Valtellina Android generata*: SLOC dei file Java

File	Physical SLOC		Logical SLOC	
	#	%	#	%
DescriptionActivity.java	81	100	48	100
ImageDownloadingTask.java	75	100	39	100
MenuActivity.java	30	100	20	100
TursimLocationDetailActivity.java	66	78.7	47	68.08
TurismLocationListActivity.java	89	88.76	53	86.79
Utils.java	23	100	15	100
VideoActivity.java	61	100	39	100

Tabella 6.2: *Valtellina Android completa*: SLOC dei file Java

In Figura 6.2 sono riportati graficamente i valori SLOC di entrambe le tabelle. Come è possibile notare la maggior parte delle Activity risultano essere già complete per come vengono generate dal generatore, e le classi interessate dalle modifiche necessarie al completamento dell'applicazione sono *TurismLocationListActivity* e *TurismLocationDetailActivity* che richiedono la definizione della logica applicativa di passaggio dei parametri dalla prima alla seconda Activity e del settaggio degli elementi grafici nella seconda.
Non abbiamo riportato i valori di conteggio per i singoli file XML interni al progetto Android dell'applicazione per via del fatto che abbiamo trascurato la ridefinizione della veste grafica. Infatti, a meno di alcuni riposizionamenti all'interno delle schermate, non abbiamo apportato alcuna modifica a questi file dopo l'importazione del progetto in Eclipse.
In Figura 6.3 è riportato un grafico che mostra un confronto quantitativo ad alto livello tra il codice dell'applicazione prodotta dal generatore e quello dell'applicazione completa: risulta evidente che la quantità di righe e statement dei file XML, dove è compreso anche il Manifest, è invariata nelle due versioni dell'applicazione Android, mentre emergono alcune variazioni nei file Java.

In base ai dati raccolti sembra che il generatore di codice permetta davvero, almeno nel caso di Valtellina, di risparmiare buona parte del tempo richiesto per lo sviluppo completo dell'applicazione. Senza considerare alcuna modifi-

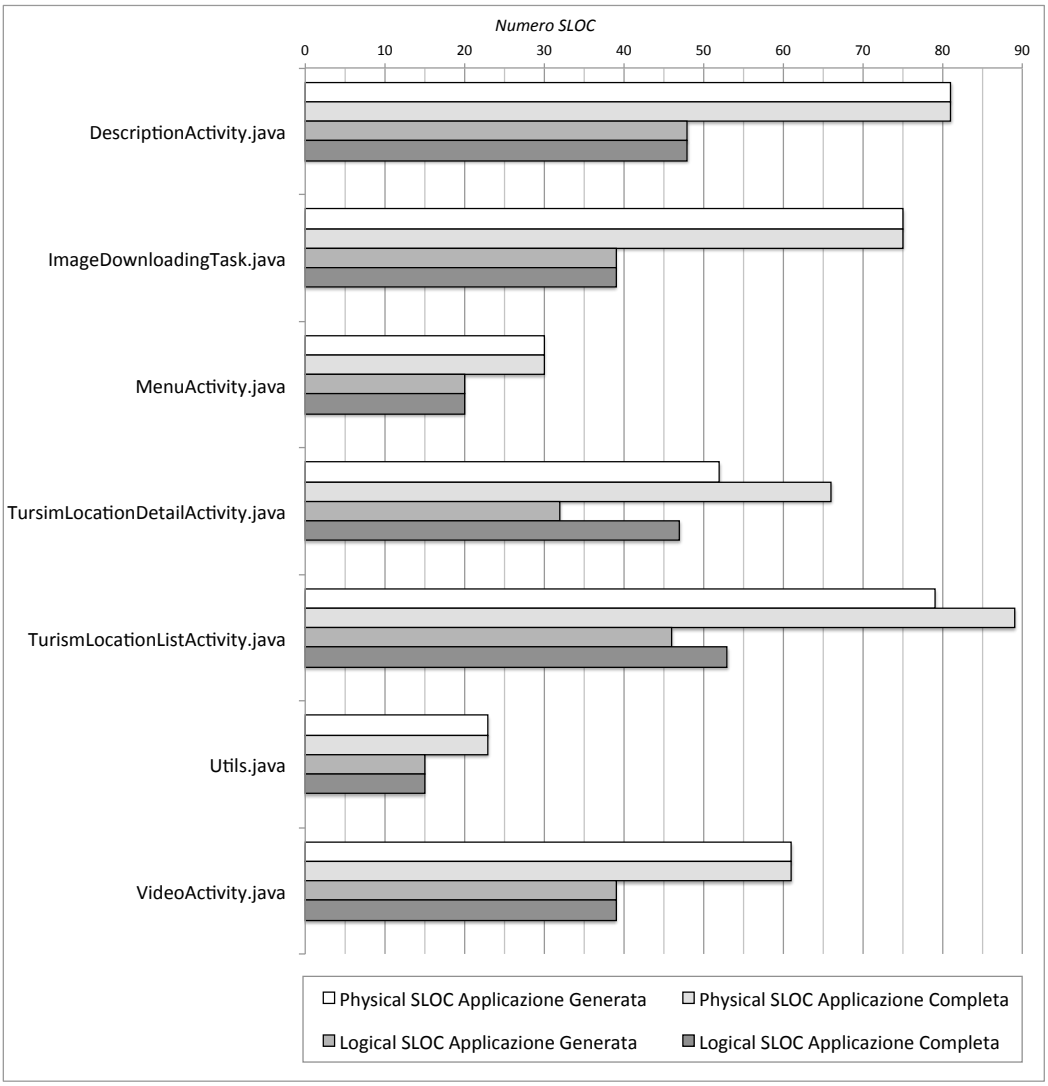

Figura 6.2: *Valtellina Android*: confronto SLOC dei file Java

ca all'interfaccia grafica, il generatore di codice produce 510 righe di codice sulle 534 righe che compongono l'applicazione completa e 326 statement sui 348 complessivi, arrivando così a coprire il 95,5% delle Physical SLOC e il 93,7% delle Logical SLOC. Questo risultato è significativo se vogliamo considerare la copertura della logica applicativa, che costituisce la parte sulla quale ci siamo concentrati maggiormente al momento della definizione dei template Xpand.

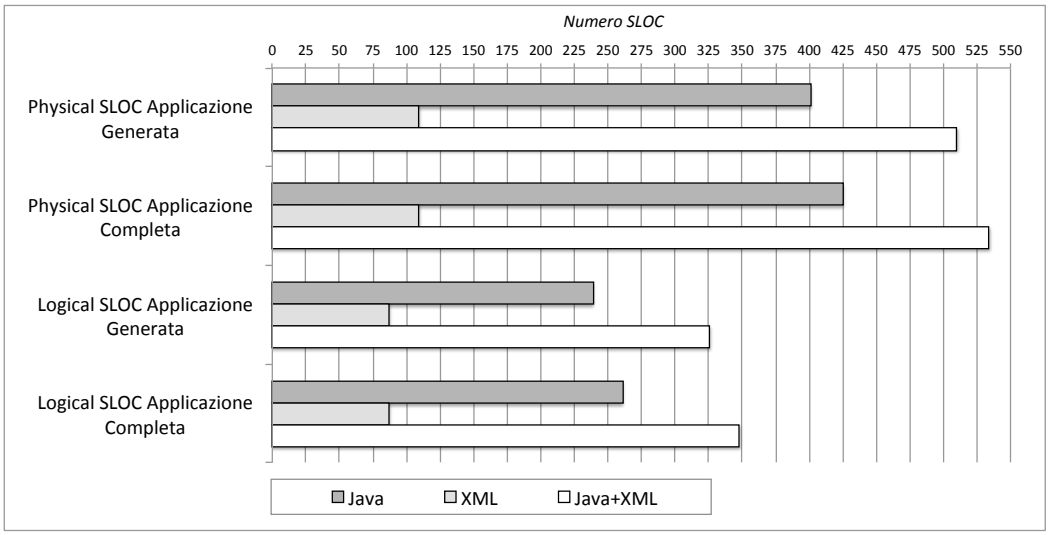

Figura 6.3: *Valtellina Android*: confronto SLOC complessive

Valutazione della versione iOS

Per quanto riguarda il layout e la grafica dei ViewController, la differenza principale rispetto ai sorgenti Android risiede nel fatto che occorre modificare un unico file, ovvero la Storyboard. Queste modifiche avvengono sfruttando l'Interface Builder, uno strumento integrato in Xcode, che offre un editor grafico intuitivo e ricco di opzioni utili. Per quanto riguarda i ViewController, TurismLocationsList e TurismLocationDetail, oltre alla definizione dei due array per recuperare i file di testo e le immagini dobbiamo definire la logica applicativa che determina il passaggio delle informazioni dal primo ViewController al secondo e quella per il settaggio dinamico dei contenuti dei componenti grafici del ViewController di dettaglio.

Le considerazioni relative al codice iOS prodotto sono simili a quelle viste per la versione Android. Anche in questo caso abbiamo confrontato i valori Physical SLOC e Logical SLOC del codice generato con quelli del codice completo per i file più significativi del progetto. Oltre a considerare i file Objective-C, che definiscono buona parte della logica applicativa, abbiamo valutato anche la Storyboard, che costituisce un elemento di fondamentale importanza nello sviluppo di applicazioni per iOS e che, a differenza dei file XML del layout in Android, definisce sia il layout dell'applicazione che la logica applicativa delle navigazioni tra ViewController.

In Tabella 6.3 sono riportati i valori di conteggio relativi ai file dell'applicazione prodotta dal generatore, mentre in Tabella 6.4 ci sono quelli relativi

File	Physical SLOC	Logical SLOC
DescriptionViewController.m	79	29
VideoViewController.m	79	28
TurismLocationsListViewController.m	131	45
TurismLocationDetailViewController.m	72	20
ImageDownloadingTask.m	36	15
DescriptionViewController.h	5	2
VideoViewController.h	6	3
TurismLocationsListViewController.h	6	2
TurismLocationDetailViewController.h	5	2
ImageDownloadingTask.h	6	3
MainStoryboard.storyboard	259	159

Tabella 6.3: *Valtellina iOS generata*: SLOC dei file .m, .h e della Storyboard

File	Physical SLOC		Logical SLOC	
	#	%	#	%
DescriptionViewController.m	79	100	29	100
VideoViewController.m	79	100	28	100
TurismLocationsListViewController.m	170	77.05	60	75
TurismLocationDetailViewController.m	75	96	29	69
ImageDownloadingTask.m	36	100	15	100
DescriptionViewController.h	5	100	2	100
VideoViewController.h	6	100	3	100
TurismLocationsListViewController.h	6	100	2	100
TurismLocationDetailViewController.h	8	62,5	5	40
ImageDownloadingTask.h	6	100	3	100
MainStoryboard.storyboard	237	—	162	98.1

Tabella 6.4: *Valtellina iOS completa*: SLOC dei file .m, .h e della Storyboard

ai file dell'applicazione modificata con le percentuali di copertura del codice prodotto con il generatore di codice rispetto a quello completo. In Figura 6.4 è riportato un grafico che mostra i valori SLOC dei file .m e della Storyboard. Come è possibile notare, anche nel caso dell'applicazione iOS, gli unici file che sono stati modificati, escludendo la Storyboard, sono i due relativi alle ViewController delle località turistiche.
Osservando il grafico potrebbe sorprendere l'assenza del valore in percentuale di copertura, motivato da un abbassamento del valore delle Physical SLOC relative al file della Storyboard. Grazie all'utilizzo di DiffMerge, un software per il confronto tra file, abbiamo riscontrato che il motivo di questo cambiamento inaspettato è da ricercare nel generatore di codice che, sfruttando il template Xpand finalizzato alla produzione della Storyboard, produce un file contenente alcuni spazi bianchi, che non impattano sul corretto funzionamento dell'applicazione. Al momento dell'importazione dei sorgenti in Xcode, e

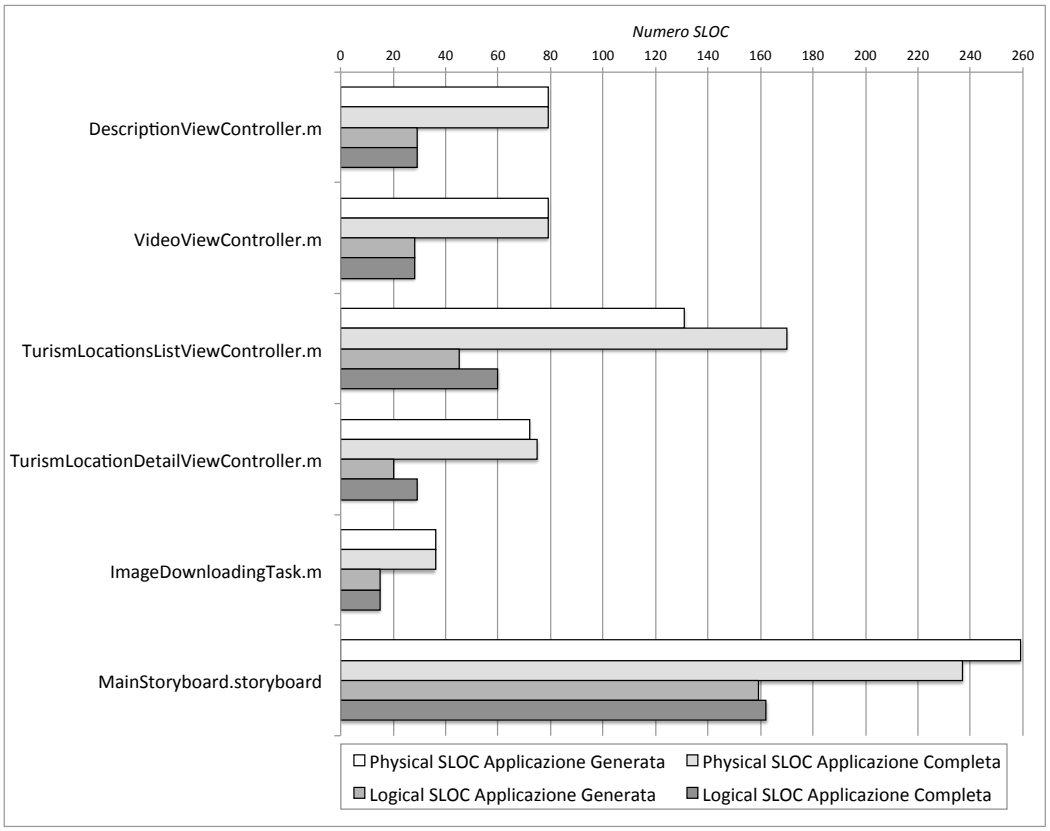

Figura 6.4: *Valtellina iOS*: confronto SLOC dei file .m e della Storyboard

in particolare all'apertura del file della Storyboard, quello che si verifica è una pulizia automatica del codice, con rimozione degli spazi bianchi generati e conseguente riduzione del numero delle Physical SLOC.
Se confrontiamo invece il numero di Logical SLOC dell'applicazione generata con quello dell'applicazione completa, possiamo riscontrare un leggero aumento nell'applicazione completa: ciò è dovuto all'inserimento di alcuni outlet nel TurismLocationDetailViewController, necessari per settare correttamente gli elementi grafici manualmente via codice.

In Figura 6.5 è riportato un grafico che mostra un confronto quantitativo ad alto livello tra il codice dell'applicazione prodotta dal generatore e quello dell'applicazione completa, dove risulta evidente la piccola differenza esistente nel numero di righe e nel numero di statement del codice prodotto e del codice completo. Senza considerare alcuna modifica all'interfaccia grafica, il generatore di codice produce 684 righe di codice sulle 707 complessive e 308 statement sui 338 complessivi, arrivando così a coprire il 96,7% delle Physical SLOC e il 91% delle Logical SLOC. Possiamo quindi affermare che, anche

per la versione iOS, il generatore ha permesso di risparmiare buona parte del tempo di sviluppo dell'applicazione. E' bene tenere presente però che anche in questo caso i valori di conteggio sono stati determinati senza apportare particolari modifiche all'interfaccia grafica esistente, che può essere ridefinita rapidamente attraverso lo strumento Interface Builder di Xcode.

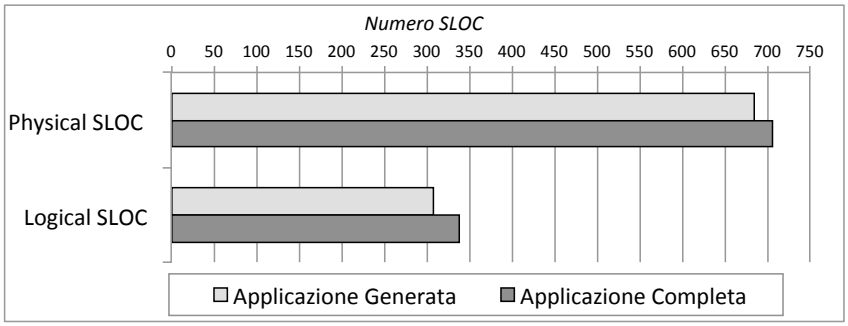

Figura 6.5: *Valtellina iOS*: confronto SLOC complessive

6.1.2 Applicazione Multimedia

Multimedia è un'applicazione pensata per permettere all'utente di scattare delle fotografie, registrare dei filmati con la videocamera del dispositivo ed effettuare registrazioni vocali. L'applicazione, inoltre, mette a disposizione una Gallery per mostrare i file multimediali prodotti, divisi per categoria, in modo da permettere all'utente di vedere ed eventualmente cancellare le foto, i video o i file audio che sono stati salvati in memoria.

Così come per Valtellina, ai fini di chiarire la struttura dell'applicazione e quale sarà il contenuto del file .xmi che utilizzeremo come input per la fase di traduzione del modello in codice, abbiamo prodotto un'istanza del meta-modello implementativo, mostrata nelle Figure 6.6 e 6.7. Abbiamo dovuto spezzare il modello su due figure per problemi di spazio.
Come è possibile notare, in Figura 6.6 sono stati indicati i tre ViewController e i relativi elementi che permettono di sfruttare i componenti hardware del dispositivo per generare e salvare in memoria un nuovo file multimediale (immagine, video o audio). Questi ViewController sono raggiungibili attraverso i MenuItem che compongono il menu dell'applicazione indicato in figura.
Oltre a tutti questi elementi è stato inserito il ViewController Gallery, che sarà il primo ad essere mostrato all'apertura dell'applicazione e che può essere raggiunto selezionando la relativa voce del menu. Attraverso i tre Button che compongono Gallery è possibile raggiungere i ViewController PhotoGrid,

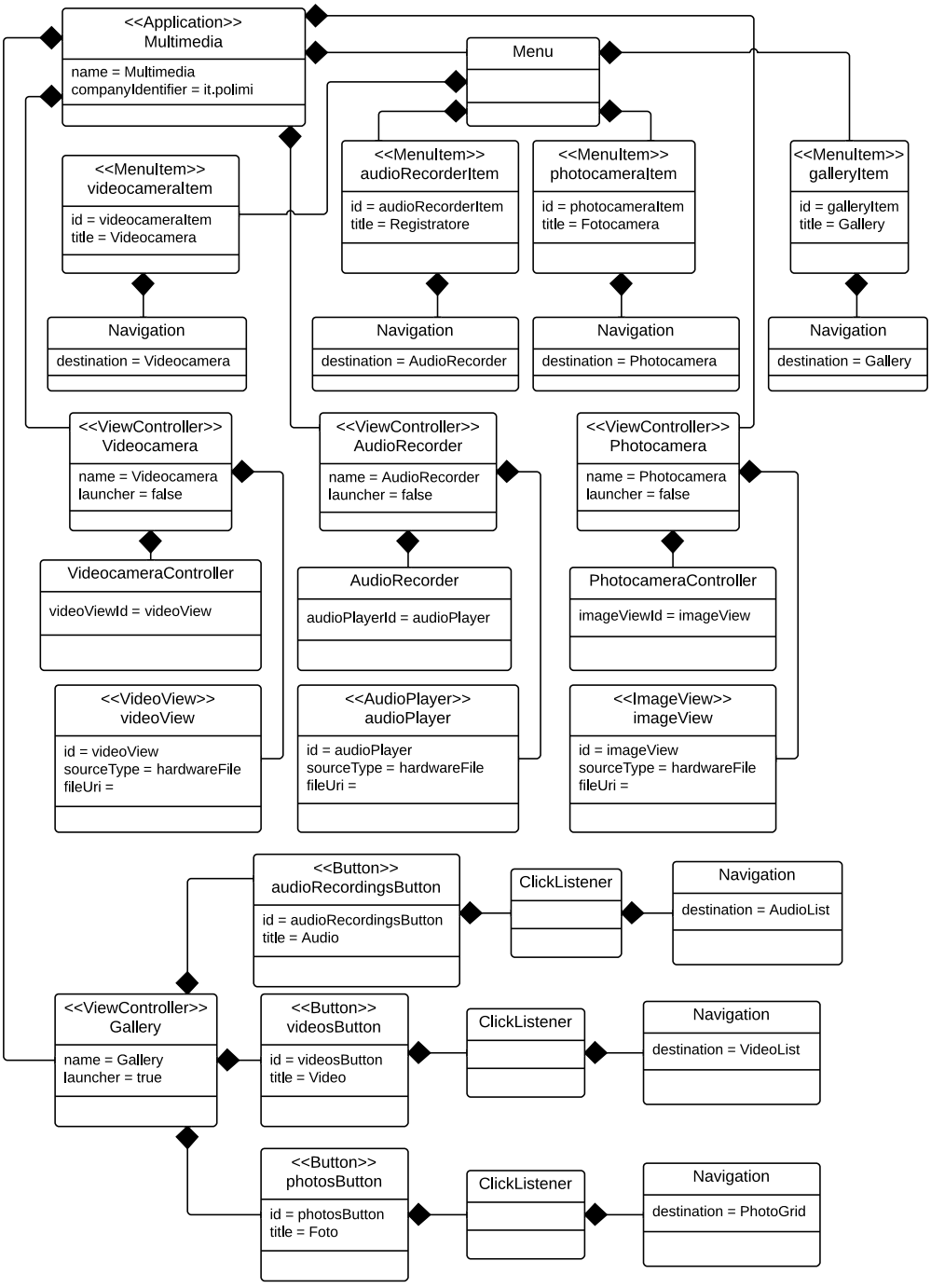

Figura 6.6: *Multimedia*: istanza del metamodello implementativo - parte 1

VideoList e AudioList, che sono riportati in Figura 6.7 e che mostrano rispettivamente la lista di tutte le foto, i video e i file audio salvati in memoria.

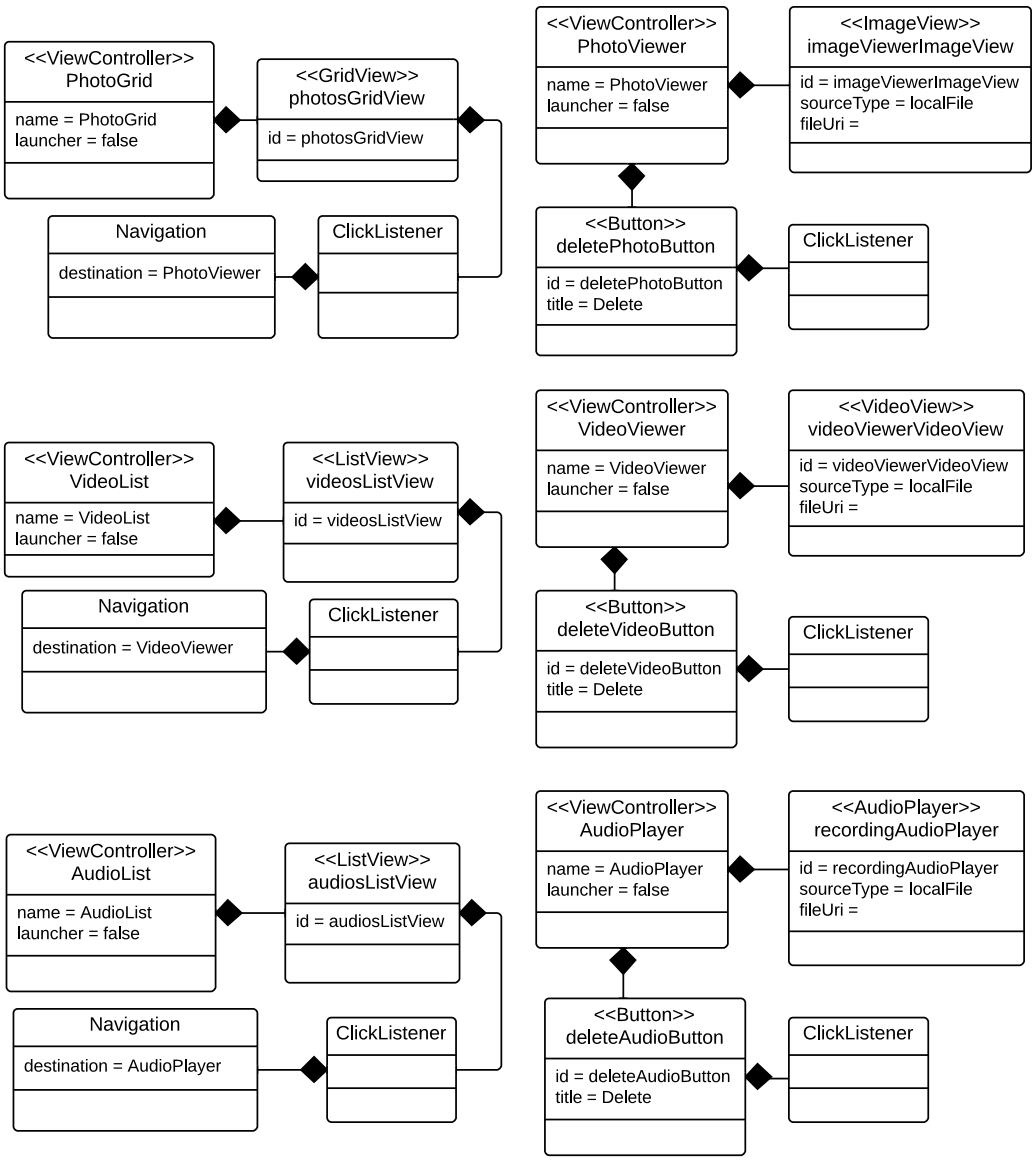

Figura 6.7: *Multimedia*: istanza del metamodello implementativo - parte 2

La selezione di una voce all'interno di una di queste tre liste porta all'apertura di un ViewController contenente i dettagli e che consente la cancellazione dello specifico file multimediale.

Dopo aver generato il file .xmi e inserito gli elementi in accordo con il modello presentato nelle due figure, abbiamo generato i sorgenti Android e iOS corrispondenti, le cui valutazioni sono riportate di seguito.

Valutazione della versione Android

Escludendo il riposizionamento degli elementi grafici nei layout dei ViewController, abbiamo dovuto implementare manualmente il recupero dei file multimediali dalla memoria all'apertura dei ViewController PhotoGrid, VideoList e AudioList, oltre alla logica applicativa relativa al passaggio dei parametri necessari verso i ViewController di dettaglio e quella per settare dinamicamente il contenuto dello specifico componente grafico all'interno dei ViewController PhotoViewer, VideoViewer e AudioPlayer.
Per quanto riguarda il ViewController PhotoGrid, esso è stato progettato per mostrare in una griglia le miniature delle fotografie scattate dall'utente, quindi, oltre alla necessità di definire via codice il recupero delle immagini dalla memoria, è stato necessario creare manualmente un file XML, chiamato *item_imagesgrid.xml*, per definire il layout delle celle della griglia. Questo file contiene una semplice ImageView, che abbiamo settato manualmente nell'Activity contenente la griglia implementando un adapter alternativo a quello predefinito, in modo da gestire l'inserimento di foto, e non di testi, all'interno delle GridViewCell. Infine, aggiungendo poche righe di codice, abbiamo implementato anche la cancellazione dalla memoria interna di un particolare file multimediale, da svolgersi al momento del click da parte dell'utente finale dell'apposito bottone Delete presente all'interno dei tre ViewController di dettaglio.

Nelle tabelle 6.5 e 6.6 sono riportati i valori delle Physical SLOC e Logical SLOC presenti all'interno dei file del progetto Android, generato e completo rispettivamente. Nella seconda tabella sono riportate anche le percentuali di codice coperto con i sorgenti prodotti dal generatore di codice per ognuno dei file presi in analisi.
Entrando nel dettaglio di quanto riportato nelle tabelle e consultando gli stessi dati sul grafico di Figura 6.8 possiamo notare che sono state modificate solo le Activity *AudioListActivity.java*, *AudioPlayerActivity.java*, *PhotoGridActivity.java*, *PhotoViewerActivity.java*, *VideoListActivity.java* e *VideoViewerActivity.java*, in accordo con quanto appena esposto. Per quanto riguarda i file XML che definiscono il layout delle Activity, così come per Valtellina, i file esistenti non hanno subito alcuna variazione significativa, poiché abbiamo preferito trascurare la ridefinizione della veste grafica, che per come viene generata permette di avere un'applicazione comunque funzionale.

File	Physical SLOC	Logical SLOC
AudioListActivity.java	78	50
AudioPlayerActivity.java	106	66
AudioRecorderActivity.java	165	101
GalleryActivity.java	87	32
MenuActivity.java	35	26
PhotocameraActivity.java	106	68
PhotoGridActivity.java	78	49
PhotoViewerActivity.java	63	40
Utils.java	28	15
VideocameraActivity.java	117	76
VideoListActivity.java	78	49
VideoViewerActivity.java	73	46

Tabella 6.5: *Multimedia Android generata*: SLOC dei file Java

File	Physical SLOC		Logical SLOC	
	#	%	#	%
AudioListActivity.java	85	91.8	57	87.7
AudioPlayerActivity.java	110	96.4	70	94.3
AudioRecorderActivity.java	165	100	101	100
GalleryActivity.java	87	100	32	100
MenuActivity.java	35	100	26	100
PhotocameraActivity.java	106	100	68	100
PhotoGridActivity.java	119	65.5	87	56.3
PhotoViewerActivity.java	71	88.7	48	83.3
Utils.java	28	100	15	100
VideocameraActivity.java	117	100	76	100
VideoListActivity.java	85	91.8	56	87.5
VideoViewerActivity.java	75	97.3	48	95.8

Tabella 6.6: *Multimedia Android completa*: SLOC dei file Java

In Figura 6.9 è riportato un grafico con i valori delle SLOC totali. Considerando sia i file Java che quelli XML sono state generate 1274 Physical SLOC e 827 Logical SLOC, mentre nell'applicazione completa sono state contate 1343 Physical SLOC e 928 Logical SLOC. Quindi il generatore ha coperto il 94,9% delle righe di codice e l'89% degli statement. Anche considerando i soli file Java la percentuale di statement coperti risulta essere del 90,3%, infatti abbiamo contato 684 statement complessivi, di cui 618 risultano essere autogenerati.

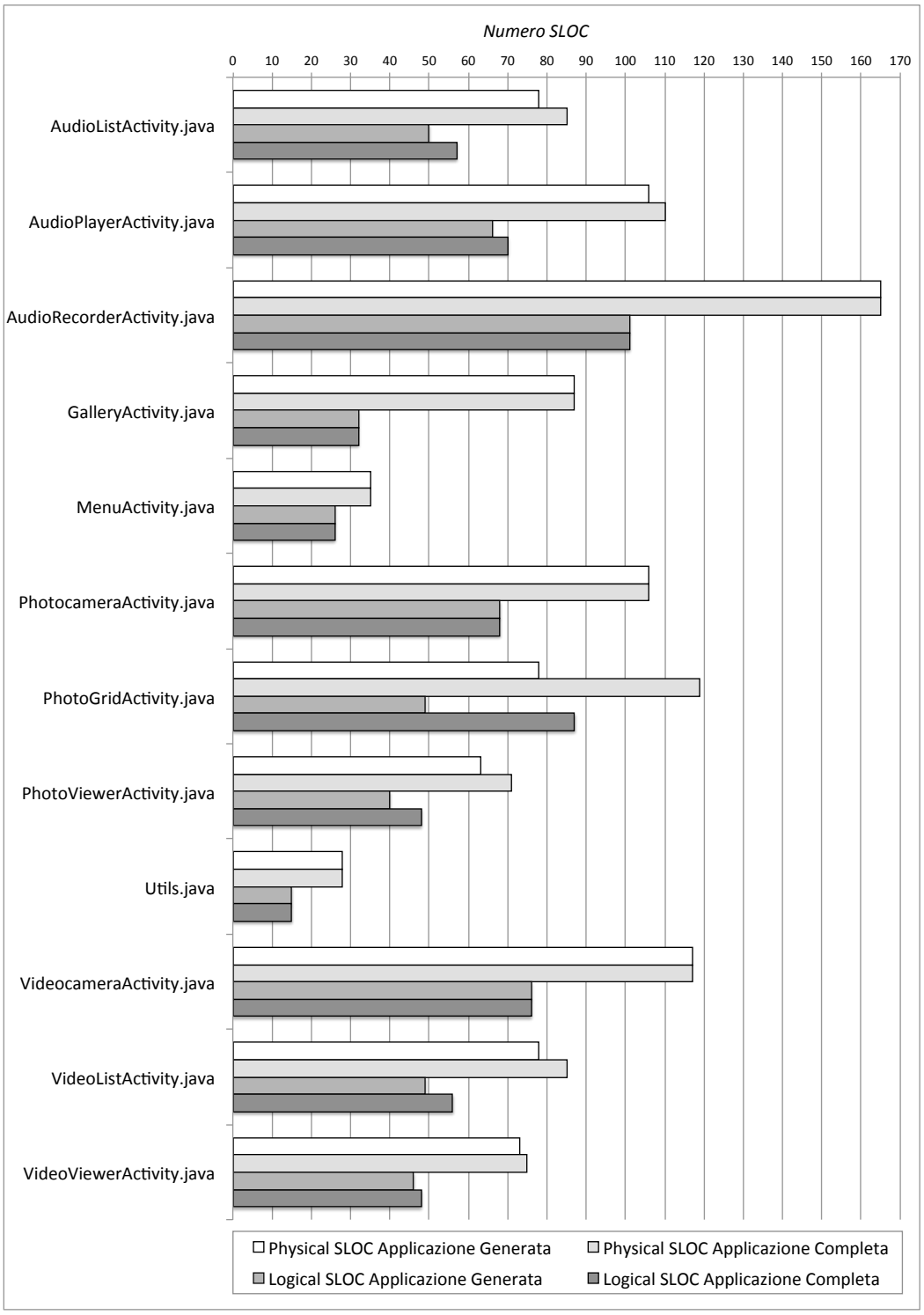

Figura 6.8: *Multimedia Android*: confronto SLOC dei file Java

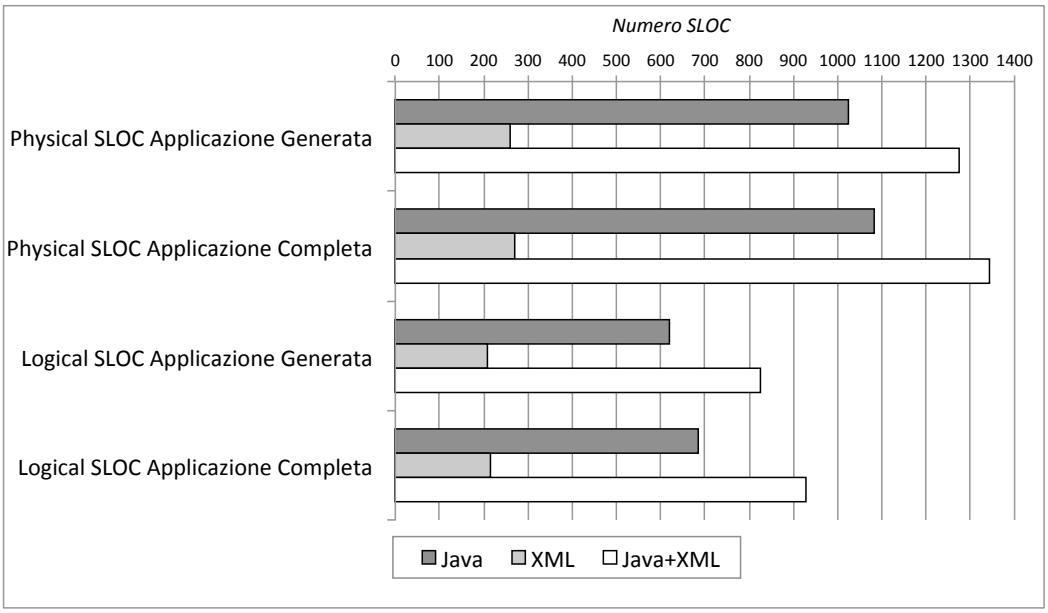

Figura 6.9: *Multimedia Android*: confronto SLOC complessive

Valutazione della versione iOS

Molte delle considerazioni presentate per i sorgenti Android prodotti valgono per i sorgenti iOS dell'applicazione, con la differenza, considerata anche per l'applicazione Valtellina, che occorre modificare un unico file, la Storyboard, per la definizione del layout grafico, in alternativa a tanti file XML.
Osservando la Tabella 6.7, la Tabella 6.8 e i grafici in Figura 6.10 e in Figura 6.11, si può notare che i ViewController che hanno subito delle modifiche corrispondono alle Activity modificate nel progetto Android. In aggiunta a *MainStoryboard.storyboard*, si tratta in particolare dei file *PhotoGridViewController.m*, *VideoListViewController.m*, *AudioListViewController.m*, *PhotoViewerViewController.m*, *VideoViewerViewController.m* ed *AudioPlayerViewController.m*. Per non appesantire troppo la lettura delle due tabelle abbiamo preferito non riportare i valori di conteggio relativi ai file .h dei ViewController, che comunque risultano invariati nell'applicazione generata e completa. Osservando nelle due tabelle i valori di Logical SLOC e Physical SLOC corrispondenti ad uno stesso file è possibile riscontrare notevoli discostamenti: queste differenze sono in parte dovute alla presenza di alcuni commenti o spazi bianchi nei file generati, ma la causa principale risiede nel modo in cui abbiamo definito gli statement all'interno delle classi Objective-C del progetto. Laddove possibile abbiamo infatti cercato di compattare in una singola riga di codice più chiamate a funzione, in modo da evitare la dichiarazione e utilizzo di un numero eccessivo di variabili.

File	Physical SLOC	Logical SLOC
VideocameraViewController.m	138	59
PhotocameraViewController.m	132	53
AudioRecorderViewController.m	145	68
GalleryViewController.m	72	20
PhotoGridViewController.m	119	45
VideoListViewController.m	118	44
AudioListViewController.m	118	45
PhotoViewerViewController.m	75	21
VideoViewerViewController.m	83	29
AudioPlayerViewController.m	98	32
CustomCollectionViewCell.m	24	6
MainStoryboard.storyboard	720	448

Tabella 6.7: *Multimedia iOS generata*: SLOC dei file .m e della Storyboard

File	Physical SLOC		Logical SLOC	
	#	%	#	%
VideocameraViewController.m	138	100	59	100
PhotocameraViewController.m	132	100	53	100
AudioRecorderViewController.m	145	100	68	100
GalleryViewController.m	72	100	20	100
PhotoGridViewController.m	132	90.2	55	81.8
VideoListViewController.m	129	91.5	52	84.6
AudioListViewController.m	128	92.2	53	84.9
PhotoViewerViewController.m	76	98.7	27	77.8
VideoViewerViewController.m	86	96.5	32	90.6
AudioPlayerViewController.m	103	95.1	37	86.5
CustomCollectionViewCell.m	24	100	6	100
MainStoryboard.storyboard	694	—	476	94.1

Tabella 6.8: *Multimedia iOS completa*: SLOC dei file .m e della Storyboard

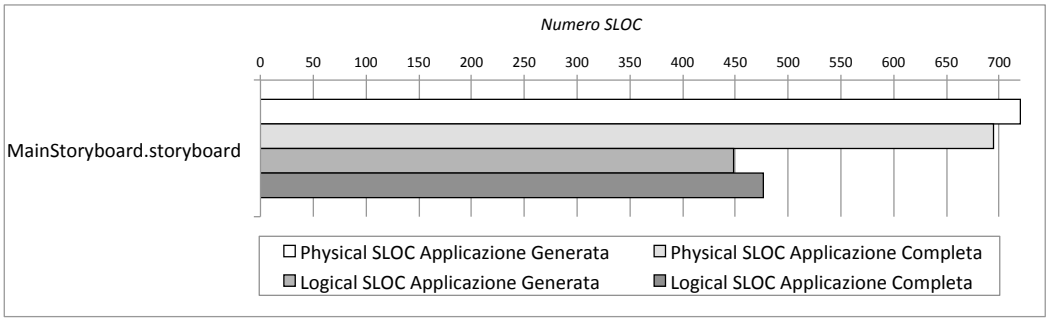

Figura 6.10: *Multimedia iOS*: confronto SLOC della Storyboard

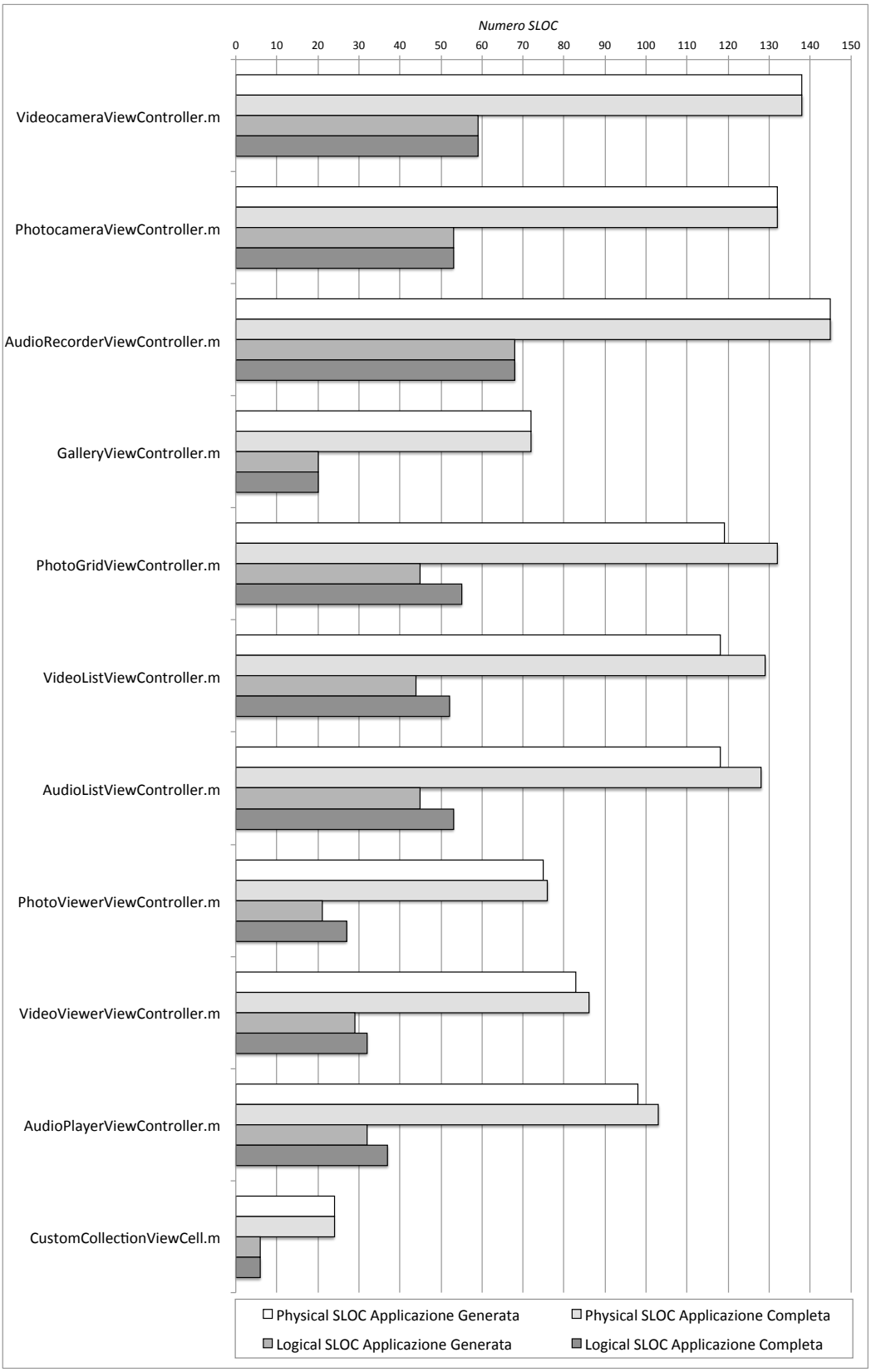

Figura 6.11: *Multimedia iOS*: confronto SLOC dei file .m

Per il file MainStoryboard.storyboard la Tabella 6.7, la Tabella 6.8 e la Figura 6.10 mostrano che il valore delle Physical SLOC è inferiore nell'applicazione completa: così come per l'applicazione Valtellina, il motivo di questa diminuzione è ancora da ricercarsi nella pulizia automatica del codice XML interno alla Storyboard al momento dell'importazione dei sorgenti in Xcode. Confrontando invece il valore delle due Logical SLOC, si osserva un numero maggiore di statement nell'applicazione completa, determinato dal fatto che abbiamo aggiunto alcuni outlet per la gestione via codice dei contenuti degli elementi grafici che costituiscono la gallery dell'applicazione.

Il progetto dell'applicazione Multimedia completa, come mostrato in Figura 6.12, richiede, per i file .m, .h e per la Storyboard, un totale di 1931 Physical SLOC, di cui il generatore di codice ha prodotto rispettivamente 1910 Physical SLOC, coprendo cosi il 98,9% delle righe di codice. Considerando invece le Logical SLOC, l'applicazione completa richiede in totale 969 statement, di cui 476 risiedono nel file della Storyboard, mentre il generatore di codice ha prodotto 899 Logical SLOC totali, di cui 448 interni alla Storyboard. Perciò la percentuale di statement coperta per la totalità del codice risulta essere circa del 92,8%, mentre per la Storyboard il generatore è riuscito a produrre il 94% degli statement. Questi risultati vanno considerati tenendo sempre presente che sono state escluse considerazioni di restyling della veste grafica dell'applicazione.

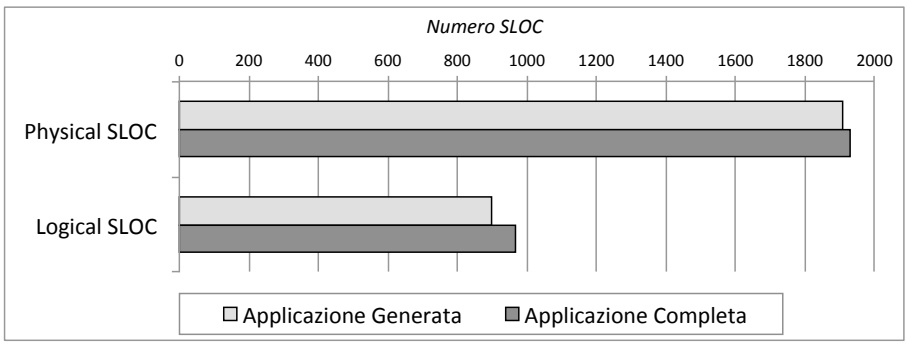

Figura 6.12: *Multimedia iOS*: confronto SLOC complessive

6.1.3 Osservazioni

Il fatto di aver analizzato unicamente due applicazioni, peraltro prive di parti algoritmiche particolarmente complesse, non ci permette certamente di generalizzare la validità dei risultati ottenuti rispetto a una qualsiasi applicazione. In generale possiamo però confermare con i dati ottenuti l'effettiva utilità dello strumento nel mondo dello sviluppo di applicazioni per dispositivi mobili,

tenendo presente le limitazioni che caratterizzano l'approccio suggerito. Prima tra queste ricordiamo l'impossibilità di modellare completamente le parti algoritmiche dell'applicazione: come vedremo nell'analisi qualitativa, utilizzare il generatore di codice per lo sviluppo di applicazioni con una logica applicativa articolata richiederebbe infatti agli sviluppatori di implementare buona parte della logica applicativa.
Ricordiamo inoltre che i dati per la valutazione quantitativa sono stati raccolti trascurando considerazioni di ridefinizione della veste grafica dell'applicazione generata. Abbiamo già detto infatti che lo strumento attualmente non consente la definizione di una veste grafica personalizzata per l'applicazione e i componenti grafici inseriti nel modello vengono posizionati nei sorgenti generati l'uno di seguito all'altro sullo schermo. Non escludiamo naturalmente la possibilità di lavorare su questi aspetti in futuro.

6.2 Valutazione qualitativa

Dopo aver cercato di fornire una valutazione quantitativa del generatore di codice, analizzando i sorgenti delle due semplici applicazioni Valtellina e Multimedia, in questa sezione proviamo a valutare qualitativamente come l'uso del generatore possa impattare sullo sviluppo di un'applicazione mediamente complessa.

L'ambito di utilizzo per cui questo progetto è nato ed è stato portato avanti è quello dello sviluppo di applicazioni native. In particolare il tool è stato pensato in modo da permettere ai team che si occupano dello sviluppo di una stessa applicazione per piattaforme diverse, di condividere la fase di progettazione dell'applicazione, definendo un'istanza del metamodello e producendo i sorgenti Android e iOS corrispondenti, che costituiscono un punto di partenza avanzato della fase di implementazione dell'applicazione. L'opportunità di avere uno strumento che permette di definire applicazioni parzialmente o completamente funzionali attraverso un approccio model-driven può velocizzare il processo di sviluppo dell'applicazione. Sebbene il principale beneficio sia la diminuzione dei tempi per lo sviluppo, l'utilizzo del tool può impattare anche sull'effort richiesto dai membri dei team di sviluppo, che in questo modo non si trovano a dover partire da zero nella fase di design e di implementazione.

Oltre alla definizione di un'applicazione totalmente nuova, il tool può essere usato anche nel momento in cui esiste già una versione dell'applicazione e si intende rinnovarla, oppure nel caso in cui è già stata sviluppata per una specifica piattaforma e si rende necessario implementare la stessa applicazio-

ne per un sistema operativo differente. Per quanto riguarda la definizione di una nuova versione di un'applicazione sviluppata per una certa piattaforma, difficilmente l'utilizzo del tool può essere la soluzione più indicata: è da valutare se il rinnovamento con un approccio model-driven di quella specifica applicazione implica una ridefinizione di parti esistenti nella vecchia versione oppure no. Diverso è il caso della definizione di una versione della stessa applicazione per una piattaforma diversa, dove l'utilizzo del tool può determinare un punto di partenza interessante nella fase di sviluppo.

Per valutare qualitativamente l'impatto del tool sullo sviluppo di più versioni di una stessa applicazione abbiamo considerato l'applicazione **Franciacorta**, di cui esistono già le versioni iOS e Android. La prima è presente sull'Apple Store dal 2011 all'url `http://itunes.apple.com/it/app/franciacorta/id441208456?mt=8`, mentre la seconda è stata rilasciata sul Google Play Store nell'Aprile di quest'anno ed è scaricabile dall'url `http://play.google.com/store/apps/details?id=net.franciacorta.android`.

Figura 6.13: *Franciacorta*: due schermate dell'applicazione Android

In Figura 6.13 sono riportati alcuni screenshot della versione Android, ma la struttura e le schermate delle due applicazioni sono molto simili, nonostante rispecchino le caratteristiche di design della specifica piattaforma di appartenenza. Gli screenshot riportati in figura sono quelli relativi alle news inerenti il territorio, ma l'applicazione comprende altre sezioni in cui si possono visualizzare gli itinerari pedonali e ciclo-pedonali della zona, i locali e le attività commerciali, alcune fotografie, un video e una descrizione del territorio e del consorzio di vini Franciacorta.

Immaginiamo ora di avere a disposizione la versione iOS dell'applicazione e di voler sviluppare quella Android, immedesimandoci così nel team di sviluppo che si è dedicato alla realizzazione della stessa all'inizio di quest'anno. Attraverso il tool è possibile definire in maniera semplice il modello dell'applicazione, sulla base della struttura dell'applicazione iOS esistente. Non possiamo chiaramente aspettarci di riuscire a produrre un modello che possa garantire la generazione dell'applicazione completa, infatti uno dei limiti principali della fase di modellazione è quello di non permettere allo sviluppatore di definire le parti algoritmiche dell'applicazione.

Innanzitutto Franciacorta presenta numerosi task asincroni, finalizzati a salvare e recuperare informazioni dalla memoria del dispositivo, e a gestire uno scambio dati con un server remoto. Per esempio, nella sezione delle notizie, rappresentata in Figura 6.13, in caso di connessione a Internet disponibile l'applicazione richiede la lista aggiornata delle news al server centrale, in caso contrario estrae gli ultimi dati memorizzati dalla memoria interna del dispositivo. Quello che è possibile fare a livello di modello è limitarsi a definire i nomi di questi task asincroni e rimandare alla fase successiva di revisione dei sorgenti prodotti l'implementazione delle parti algoritmiche.
Oltre a queste limitazioni è richiesta la definizione della logica applicativa che determina la transizione da un ViewController contenente una lista al ViewController di dettaglio: lo stesso problema riscontrato nelle applicazioni Valtellina e Multimedia descritte in precedenza, che in genere richiede però l'aggiunta di poche righe di codice. Infine l'applicazione presenta un'interfaccia grafica articolata, che il generatore non permette certamente di riprodurre automaticamente. In questo caso è quindi richiesto uno sforzo sostanzioso allo sviluppatore che intende utilizzare il generatore di codice e replicare la veste grafica dell'applicazione iOS originale.

Le considerazioni viste valgono naturalmente anche nel caso della situazione opposta, ovvero lo sviluppo della versione iOS a partire da quella Android, e possono essere generalizzate allo sviluppo di una qualsiasi applicazione per una o più piattaforme. Quindi il principale vantaggio dell'approccio model-driven suggerito riguarda il fatto di poter definire in poco tempo l'istanza del metamodello e quindi avere immediatamente sorgenti corrispondenti da cui partire per completare l'applicazione. Nonostante possa rivelarsi uno strumento decisamente interessante a disposizione degli sviluppatori, non dimentichiamo il fatto che presenta anche delle limitazioni, legate principalmente all'impossibilità di modellare le parti algoritmiche e personalizzare al momento della modellazione la veste grafica dell'applicazione.

6.3 Conclusioni

In questo capitolo abbiamo dato una valutazione quantitativa e qualitativa al risultato del lavoro di tesi, che si concretizza nella produzione di uno strumento in grado di offrire un approccio alternativo a quelli cross-platform di cui abbiamo ampiamente discusso inizialmente.
La valutazione quantitativa è stata svolta considerando due piccole applicazioni, dove i risultati sembrano essere rilevanti. Sebbene per entrambi il generatore copra più del 90% dei sorgenti necessari, raggiungendo un ottimo risultato, bisogna tenere presente che considerare solo due applicazioni non particolarmente estese, non è statisticamente sufficiente a dimostrare la positività dell'impatto del tool sullo sviluppo di applicazioni mobili.
Da un punto di vista qualitativo abbiamo invece fatto delle considerazioni generali sulle possibili situazioni in cui è indicato l'utilizzo dello strumento. Nello specifico abbiamo visto quali possono essere i principali benefici e svantaggi che nascono dall'impiego di un approccio model-driven allo sviluppo di applicazioni mobili. Tra le limitazioni dello strumento abbiamo in particolare riscontrato il fatto di non riuscire a coprire le parti algoritmiche e definire una grafica personalizzata. Analizzando l'applicazione Franciacorta abbiamo inoltre considerato un caso concreto in cui si possa giudicare necessario lo sviluppo di una nuova versione di un'applicazione esistente.

Un punto sul quale non ci siamo soffermati in questo capitolo è la valutazione dell'utilizzo del solo metamodello astratto, che può avvenire per la definizione della struttura ad alto livello di un'applicazione da sviluppare per una o più piattaforme. Un metamodello a così alto livello si presenta come un generico strumento di modellazione per dispositivi mobili di facile interpretazione, in quanto permette di modellare in maniera molto semplice e generica una qualsiasi applicazione, astraendo dalla specifica piattaforma per cui la si vuole realizzare. In quanto tale non è da escludere la possibilità che possa offrire dei benefici se utilizzato come strumento stand-alone, magari prima di istanziare il metamodello implementativo.

Capitolo 7

Conclusioni

Il lavoro svolto nel corso della tesi ha riguardato in particolare lo sviluppo di due metamodelli, astratto e implementativo, di supporto alla fase di design di applicazioni mobili, e la produzione di uno strumento che prende in input il modello dell'applicazione e produce in output i sorgenti Android e iOS corrispondenti. Per quanto riguarda Android, i template sono stati definiti per produrre applicazioni compatibili con qualsiasi smartphone con livello API minimo pari a 8 e scegliendo come target il livello API 17, per iOS invece i template sono stati definiti in modo da produrre applicazioni compatibili con iPhone e iPod Touch, scegliendo come target la versione 6 di iOS.

Lo sviluppo del metamodello astratto è stato svolto considerando sempre l'obiettivo di creare uno strumento generale che potesse risultare utile nella progettazione di qualsiasi applicazione mobile, mentre per quanto riguarda la generazione dei sorgenti abbiamo posto l'attenzione sulle due piattaforme attualmente più diffuse, cercando di tradurre ogni elemento modellato in pezzi di codice che, in un'ottica Model-View-Controller, tendano a coprire il più possibile la parte di controllo, ovvero di logica applicativa. Abbiamo preferito non concentrarci particolarmente sulla definizione della grafica dell'applicazione, che nei casi Android e iOS può essere facilmente definita attraverso gli editor grafici messi a disposizione dai rispettivi ambienti di sviluppo Eclipse e Xcode. Per la parte View ci siamo quindi limitati a generare i file necessari alla definizione di una grafica essenziale, dove gli elementi dell'interfaccia utente vengono disposti linearmente nel layout dei ViewController.
Come abbiamo avuto modo di capire nel capitolo relativo alla valutazione, l'utilizzo del generatore di codice permette comunque in generale di coprire buona parte delle righe di codice necessarie allo sviluppo completo di un'applicazione, presentandosi come alternativa all'approccio cross-platform, di cui si parla molto negli ultimi anni nonostante le limitazioni che lo caratterizzano.

Sebbene il metamodello astratto possa presentarsi come uno strumento completo, il metamodello implementativo e il generatore di codice in generale richiedono uno studio e una revisione continua nel tempo, in quanto sono strettamente legati all'aggiornamento delle versioni delle piattaforme supportate, che per ora sono Android e iOS, ma che in futuro potrebbero comprenderne delle nuove. L'aggiornamento costante dello strumento è indispensabile per produrre applicazioni compatibili con le nuove versioni dei vari sistemi operativi: la struttura modulare del progetto di tesi, che permette di individuare immediatamente tutti i segmenti di codice che mappano uno specifico componente del metamodello sui sorgenti da generare, è stata definita appositamente per venire incontro a esigenze di questo tipo. La necessità di un aggiornamento continuo negli anni del progetto potrebbe costituire uno dei principali ostacoli al suo utilizzo in futuro, ma le valutazioni svolte nella fase finale del lavoro di tesi mostrano la possibilità di trarre dei benefici importanti sia dall'utilizzo dei metamodelli sia da quello del generatore di codice, entrambi intesi come utili strumenti di supporto allo sviluppo di un'applicazione per dispositivi mobili.

7.1 Sviluppi Futuri

L'ambito entro cui è stato portato avanti il lavoro di tesi è relativamente nuovo e sono numerosi gli sviluppi futuri che potrebbero ampliare il nostro progetto. Per questo motivo, per entrambe le fasi di modellazione e implementazione del lavoro di tesi, abbiamo scelto di utilizzare linguaggi e tecnologie standard. Quindi la definizione dei due metamodelli è basata sull'estensione del linguaggio UML, mentre per definire lo strumento di generazione del codice abbiamo sfruttato i framework EMF e oAW, che sono comunemente utilizzati nel contesto dello sviluppo di software model-driven.

Innanzitutto, per entrambe le fasi di modellazione e generazione di codice, si potrebbe valutare la possibilità di supportare anche la piattaforma Windows Phone 8, che costituisce il terzo player nel mercato degli smartphone, o qualcuna delle altre piattaforme esistenti, seppur meno diffuse.
Inoltre, nel quarto capitolo di questo documento abbiamo visto che alcuni degli elementi concreti definiti nel metamodello astratto non sono stati inclusi nel metamodello implementativo per diversi motivi, quindi si potrebbe proseguire con l'implementazione degli elementi mancanti, eventualmente riconsiderando anche gli elementi del metamodello già mappati nel codice.
Un ulteriore sviluppo potrebbe riguardare l'estensione dei dispositivi supportati dalle applicazioni prodotte col generatore di codice. Infatti, come anticipato in precedenza, abbiamo dato priorità alla produzione di applica-

zioni compatibili con smartphone, considerando secondari dispositivi come tablet e iPad, che potrebbero essere supportati in futuro.
Infine, utilizzando gli strumenti messi a disposizione dal framework GMF, si potrebbe pensare di rivedere e migliorare la veste grafica del generatore di codice, che attualmente mostra l'editor grafico standard di EMF. L'idea potrebbe essere quella di definire un editor grafico articolato, che possa essere di grande aiuto anche alla modellazione, per l'applicazione che si intende realizzare, di un'interfaccia grafica personalizzata e che non si limiti quindi solo a permettere la definizione degli elementi grafici da inserire nei ViewController.

Bibliografia

[1] Andre Charland and Briand Leroux. Mobile application development: Web vs. native. *Communication of the ACM*, 54(5):49 – 53, May 2011.

[2] C.P Rahul Raj and Seshu Babu Tolety. A study on approaches to build cross-platform mobile applications and criteria to select appropriate approach. *India Conference (INDICON), 2012 Annual IEEE*, pages 625–629, Dec.

[3] M. Palmieri, I. Singh, and A. Cicchetti. Comparison of cross-platform mobile development tools. *Intelligence in Next Generation Networks (ICIN), 2012 16th International Conference on*, pages 179–186, Oct.

[4] J. Ohrt and V. Turau. Cross-platform development tools for smartphone applications. *Computer*, 45(9):72–79, 2012.

[5] Apps Builder Srl. Apps builder website. `http://ibuildapp.com`, citato in Maggio 2013.

[6] Inc. iBuildApp. ibuildapp website. `http://ibuildapp.com`, citato in Maggio 2013.

[7] Adobe Systems Inc. Phonegap website. `http://phonegap.com/2012/05/02/phonegap-explained-visually/`, citato in Aprile 2013.

[8] MoSync AB. Mosync website. `http://www.mosync.com/documentation/manualpages/mosync-reload`, citato in Aprile 2013.

[9] Motorola Solution Inc. Motorola solution website. `http://docs.rhomobile.com/rhodes/introduction`, citato in Aprile 2013.

[10] Xamarin Inc. Xamarin website. `http://xamarin.com/how-it-works`, citato in Aprile 2013.

[11] Corona Labs Inc. Corona website. `http://www.coronalabs.com`, citato in Agosto 2013.

[12] Seregon Solution Inc. Dragonrad website. http://dragonrad.com/overview, citato in Aprile 2013.

[13] Bup-Ki Min, Minhyuk Ko, Yongjin Seo, Seunghak Kuk, and Hyeon-Soo Kim. A uml metamodel for smart device application modeling based on windows phone 7 platform. *TENCON 2011 - 2011 IEEE Region 10 Conference*, pages 201–205, Nov.

[14] Minhyuk Ko, Yong-Jin Seo, Bup-Ki Min, Seunghak Kuk, and Hyeon Soo Kim. Extending uml meta-model for android application. *Computer and Information Science (ICIS), 2012 IEEE/ACIS 11th International Conference on*, pages 669–674, 30 2012-June 1.

[15] Microsoft. Microsoft developer network. http://msdn.microsoft.com, citato in Luglio 2013.

[16] TechGenix Ltd. Windows phone 7 security implications. http://www.windowsecurity.com/articles-tutorials/misc_network_security/Windows-Phone-7-Security-Implications.html, citato in Luglio 2013.

[17] Google Inc. Activities. http://developer.android.com/guide/components/activities.html, citato in Luglio 2013.

[18] Google Inc. Service. http://developer.android.com/reference/android/app/Service.html, citato in Luglio 2013.

[19] Google Inc. Content provider. http://developer.android.com/reference/android/content/ContentProvider.html, citato in Luglio 2013.

[20] Inc. Object Management Group. Omg - uml website. http://www.omg.org, citato in Maggio 2013.

[21] Uml-Diagrams.org. Uml diagrams website - core elements. http://www.uml-diagrams.org/uml-core.html, citato in Luglio 2013.

[22] The Eclipse Foundation. Epsilon. http://www.eclipse.org/epsilon/, citato in Luglio 2013.

[23] The Eclipse Foundation. Eclipse modeling framework project (emf). http://www.eclipse.org/modeling/emf/?project=emf#emf, citato in Luglio 2013.

[24] openArchitectureWare.org. openarchitectureware. http://www.openarchitectureware.org/index.php, citato in Luglio 2013.

[25] Sven Efftinge, Peter Friese, Arno Hase, Dennis Hubner, Clemens Kadura, Bernd Kolb, Jan Kohnlein, Dieter Moroff, Karsten Thoms, Markus Volter, Patrick Schonbach, Moritz Eysholdt, Steven Reinisch, Darius Jockel, Andre Arnold, and contributors. Xpand reference. *Xpand*, 2010.

[26] Swen Efftinge and Markus Voelter. Openarchitectureware 4.1 workflow engine reference. *OpenArchitectureWare*.

Lista degli acronimi

3G	Third Generation
ADT	Android Developer Tools
API	Application Programming Interface
APK	Android Package
CSS	Cascading Style Sheets
EMF	Eclipse Modeling Framework
EMP	Eclipse Modeling Project
GMF	Graphical Modeling Framework
GPS	Global Positioning System
GUI	Graphical User Interface
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
HW	Hardware
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
MVC	Model View Controller
MWE	Modeling Workflow Engine
NFC	Near Field Communication
oAW	openArchitectureWare
RIM	Research in Motion
SDK	Software Development Kit
SLOC	Source Lines of Code
SysML	Systems Modeling Language
UML	Unified Modeling Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
WebML	Web Modeling Language
WP7	Windows Phone 7
XMI	XML Metadata Interchange
XML	eXtensible Markup Language