

# **UNIVERSITÀ DEGLI STUDI DI SALERNO**



Facoltà di Scienze Matematiche Fisiche e Naturali

---

Tesi di Laurea di I livello in  
***INFORMATICA***

**Miniaturizzazione di applicazioni Java Desktop in App  
Mobile: gestione delle tabelle**

**Relatore**

*Prof. Michele Risi*

**Candidato**

*Egidio Giacoia*

*Matr. 051210/2376*

---

*Anno accademico 2016/2017*

# Sommario

<b>1.</b>	<b>Introduzione.....</b>	<b>1</b>
<b>2.</b>	<b>Le tecnologie .....</b>	<b>4</b>
2.1	Le interfacce grafiche in Java .....	4
2.1.1	<i>Le classi AWT e SWING.....</i>	4
2.1.2	<i>I Container.....</i>	6
2.1.3	<i>I Top Level Container .....</i>	6
2.1.4	<i>I Component .....</i>	7
2.1.5	<i>La gestione degli eventi.....</i>	7
2.2	Le differenze nello sviluppo di applicazioni in ambiente mobile.....	7
2.2.1	<i>L'applicazione Nativa .....</i>	8
2.2.2	<i>L'applicazioni Web .....</i>	9
2.2.3	<i>L'applicazione Irida.....</i>	10
2.3	Google Android e framework.....	11
2.3.1	<i>L'Android SDK.....</i>	12
2.4	Introduzione a PhoneGap.....	12
2.4.1	<i>Che cos'è PhoneGap.....</i>	13
2.4.2	<i>Come funziona .....</i>	13
2.4.3	<i>Il plugin di comunicazione in PhoneGap.....</i>	14
2.4.4	<i>L'utilizzo del plugin nel processo di miniaturizzazione .....</i>	15
2.4.5	<i>La struttura di un progetto PhoneGap .....</i>	15
2.5	La libreria Hammer.js .....	17
<b>3.</b>	<b>Il processo di miniaturizzazione.....</b>	<b>18</b>
3.1	La strategia e il processo di migrazione.....	18
3.2	La libreria di supporto pAwt.....	20
3.2.1	<i>La gerarchia delle pAWT.....</i>	21
3.2.2	<i>Il repository.....</i>	22
3.2.3	<i>Come recuperare il frame radice.....</i>	22
3.2.4	<i>La gestione degli eventi nelle pAWT .....</i>	24
3.3	Il flusso di comunicazione .....	25
3.3.1	<i>Da PhoneGap verso Java .....</i>	25
3.3.2	<i>Da Java verso PhoneGap .....</i>	27
3.4	L'implementazione del plugin di comunicazione.....	29

3.5	Il porting della UI.....	31
3.5.1	<i>L'implementazione del parser DOM.....</i>	31
3.5.2	<i>L'integrazione con PhoneGap e Plugin.....</i>	33
3.5.3	<i>La creazione dell'Activity nel progetto PhoneGap .....</i>	35
<b>4.</b>	<b>Generazione del layout .....</b>	<b>37</b>
4.1	Il porting dei componenti grafici.....	37
4.2.1	<i>Il pulsante.....</i>	37
4.1.2	<i>La label .....</i>	37
4.1.3	<i>L'area di testo.....</i>	38
4.1.4	<i>La checkbox (radio button) .....</i>	39
4.1.5	<i>Il menù .....</i>	39
4.2	L'impaginazione del layout.....	42
4.2.1	<i>Un esempio: Rubrica.....</i>	44
<b>5.</b>	<b>La gestione della tabella .....</b>	<b>47</b>
5.1	La tabella in Java .....	47
5.2	La tabella in pAWT .....	50
5.2.1	<i>I costruttori di pTable .....</i>	50
5.2.2	<i>I metodi di pTable .....</i>	52
5.3	Introduzione al parsing della tabella .....	53
5.3.1.	<i>Come identificare la tabella nel codice.....</i>	53
5.3.2	<i>La gestione degli eventi della tabella in HTML .....</i>	54
5.3.3	<i>Le proprietà grafiche della tabella.....</i>	58
5.3.4	<i>La gestione dell'impaginazione della tabella .....</i>	60
5.4	Il parsing della tabella.....	63
5.4.1	<i>La fase di parsing: metodo createHeaderTable.....</i>	63
5.4.2	<i>La fase di parsing: metodo createRows.....</i>	64
5.4.3	<i>La fase di parsing: metodo createTableCSS .....</i>	65
5.5	La Comunicazione da Java verso PhoneGap.....	66
5.5.1	<i>Il metodo setProperty.....</i>	67
5.5.2	<i>Il metodo setRowSelection.....</i>	70
5.5.3	<i>Il metodo insertRow.....</i>	72
5.5.4	<i>Altri metodi .....</i>	76
5.6	La comunicazione da PhoneGap verso Java.....	77
<b>6.</b>	<b>Caso di studio.....</b>	<b>82</b>

6.1	ListShop.....	82
6.1.1	<i>Il codice Java AWT.....</i>	82
6.1.2	<i>La fase di parsing.....</i>	84
6.1.3	<i>Simulazione Java-Mobile: inserimento di una riga .....</i>	88
6.1.4	<i>Simulazione Java-Mobile: editing di una cella.....</i>	90
6.1.5	<i>Simulazione Java-Mobile : rimozione di una riga .....</i>	91
6.2	L'Impaginazione di una tabella di grandi dimensioni.....	94
7.	<b>Conclusioni .....</b>	96

# 1. Introduzione

---

Nell'ultimo decennio con il progredire della tecnologia il mercato degli **smartphone** continua ad espandersi ed evolversi sempre più rapidamente, diventando uno strumento fondamentale nella vita quotidiana delle persone. Oltre alla vendita dei dispositivi mobile cresce, di conseguenza, anche il mercato delle applicazioni messe a disposizione per un vasto bacino di utenti.

Al giorno d'oggi gli utenti che si connettono ad Internet da un device mobile hanno superato quelli che utilizzano un computer desktop, soprattutto grazie all'impiego delle app mobile. Le aziende per stare al passo con i tempi stanno focalizzando il proprio interesse su questo tipo di tecnologie. Tuttavia questo implica un notevole sforzo, in termini di risorse e tempo, nel riscrivere completamente da zero i sistemi pensati per essere eseguiti su **ambiente desktop**, trasformandoli in sistemi per essere eseguiti su **ambiente mobile**.

L'idea di base dell'attività di tesi nasce proprio da quest'ultimo concetto, ovvero di **miniaturizzazione** di un software, i.e., trasformare un'applicazione desktop in un'applicazione utilizzabile sullo smartphone. Tale attività è legata alla **portabilità** (dall'inglese *porting*), che consiste nell'adattare il software in modo tale che un programma applicativo possa essere eseguito in un ambiente computazionale diverso da quello per cui era stato progettato inizialmente. Le cause che possono portare ad effettuare richieste di operazioni di porting (ovvero creazione/scrittura di un port) sono dovute, per esempio, a delle diversità di utilizzo delle CPU, delle API (Application Programming Interface) dei sistemi operativi, dell'hardware o dall'incompatibilità dell'implementazione del linguaggio di programmazione sull'ambiente su cui deve essere compilato il programma (ambiente target). L'intero processo può risultare un'attività complessa e costosa: tutto è legato maggiormente dal divario tecnologico tra l'ambiente di origine e l'ambiente di destinazione, dal tipo di componente software da "portare" e dagli strumenti con cui esso è stato costruito.

Un altro aspetto da considerare nel processo di sviluppo riguarda le scelte degli investitori e sviluppatori per lanciare la propria applicazione nel mercato. Devono stabilire la strategia e il modello di business da adottare, la scelta dei collaboratori per lo sviluppo delle applicazioni, i budget necessari e gli obiettivi di guadagno con il fine di velocizzare il **Time to Market**. Infatti, la scelta del modello di sviluppo dell'applicazione mobile gioca un ruolo fondamentale: esistono soluzioni molto veloci ma che non hanno buone performance (**web app**), mentre altre che lavorano nell'ambiente nativo in modo tale da avere alte prestazioni ma sostenendo costi elevati (**app nativa**).

Una soluzione a tali problematiche è il processo di miniaturizzazione e le tecnologie sui cui si basa, che permettono di realizzare un meccanismo semi-automatico che trasforma un'applicazione desktop Java eseguibile solo sul computer, in un'applicazione eseguibile sul sistema operativo mobile Android, attraverso l'uso del framework PhoneGap. La soluzione consiste nella creazione di un wrapper per l'applicazione Java di partenza in modo tale da renderla compatibile con il nuovo ambiente in cui sarà eseguita senza modificarne la logica di business, utilizzando una **libreria di supporto** e un **middleware di comunicazione** per collegare l'interfaccia utente miniaturizzata con tutte le funzioni presenti all'interno del codice considerato. Tutto ciò porta a definire un' **applicazione** tecnicamente **ibrida**, con alcune peculiarità:

- miglioramento delle performance in quanto la logica è nativa;
- minimizzazione dello sviluppo in quanto esso parte da una applicazione già esistente.

L'obiettivo centrale del lavoro di tesi è quello di ampliare il processo di miniaturizzazione: il sistema definito permette di sviluppare nuovi componenti dell'interfaccia grafica utente gestibili in maniera automatica. L'attività è di estendere la libreria di supporto con una nuova componente per la **gestione di tabelle e delle sorgenti di dati**. Lo sviluppo di tale componente richiede una notevole sforzo di comprensione su come avviene l'interazione con l'utente e sul funzionamento e gestione dei dati in quanto la componente Java corrispondente è un oggetto composto (formato da intestazioni, colonne, righe e celle) ed opera con numerose classi di supporto che rendono arduo il processo di porting.

Il lavoro di tesi è organizzato come di seguito:

- Nel secondo capitolo verranno descritte le tecnologie coinvolte nel processo di miniaturizzazione. In particolar, saranno ampiamente descritte ed analizzate le classi grafiche di Java, le principali differenze nello sviluppo di applicazioni mobile, il sistema operativo Android, il framework PhoneGap ed i suoi plugin, ed infine la libreria Hammer.
- Nel terzo capitolo sarà spiegato in dettaglio l'intero processo di miniaturizzazione: verrà descritta la libreria di supporto pAWT e il plugin di comunicazione che permette la comunicazione tra l'applicazione Android e il codice Java.
- Il quarto capitolo è dedicato alla fasi di parsing dei componenti dell'interfaccia utente (UI) disponibili in Java e l'impaginazione del layout.

- Nel quinto capitolo sarà mostrato in dettaglio il processo di generazione del componente relativo la tabella. Verranno spiegate le classi aggiunte alla libreria di supporto e si analizzerà nel dettaglio l'algoritmo utilizzato per effettuare il parsing.
- Nel sesto capitolo verranno mostrati alcuni casi di studio implementati per verificare la validità del processo di miniaturizzazione e della app mobile generata.
- L'ultimo capitolo sarà dedicato alle conclusioni, i problemi riscontrati e gli sviluppi futuri.

## 2. Le tecnologie

---

Nel seguente capitolo verranno descritte e presentate le tecnologie usate, i sistemi operativi ed i vari ambienti di sviluppo coinvolti.

### 2.1 Le interfacce grafiche in Java

In questo paragrafo si introdurranno l'interfaccia grafica (**GUI** - Graphical User Interface) in *Java* (che rappresenta il principale strumento di interazione tra utente e applicazione), le classi, le gerarchie e la struttura di un'applicazione contenente elementi grafici i quali contribuiscono a migliorare l'usabilità<sup>1</sup> di un programma.

#### 2.1.1 *Le classi AWT e SWING*

La libreria Java contenente classi e le interfacce fondamentali per il rendering grafico è l'**Abstract Window Toolkit (AWT)**. Queste classi consentono di realizzare GUI complesse e di definire l'interazione attraverso la specifica di elementi e di gestori degli stessi.

Essa contiene:

- gli elementi base per applicazioni *GUI*, come finestre, check box, text box pulsanti;
- l'interfaccia tra il sistema a finestre nativo e l'applicazione *Java*;
- il package `java.awt.datatransfer` per poter utilizzare gli Appunti e le operazioni Drag and Drop;
- le interfacce per la gestione degli eventi e delle periferiche come mouse e tastiera;
- l'Interfaccia Nativa *AWT*, che permette alle procedure compilate in linguaggio nativo di operare su un *Canvas Java*;
- la possibilità di eseguire applicazioni del sistema operativo, come il browser o il gestore di posta elettronica da un'applicazione *Java*.

---

<sup>1</sup> L'usabilità è definita dall'ISO (International Organization for Standardization), come l'efficacia, l'efficienza la soddisfazione con le quali determinati utenti raggiungono determinati obiettivi in determinati contesti.

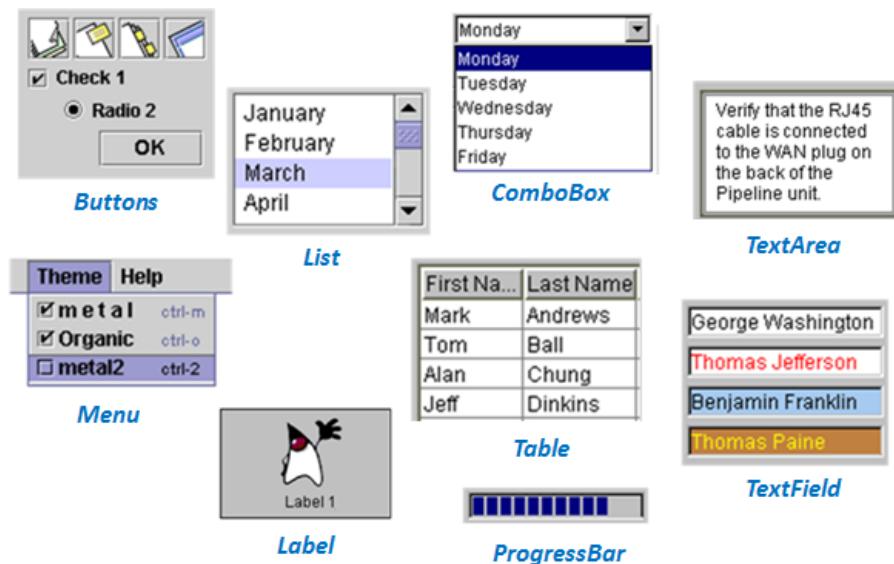


Figura 2. 1 - Esempi di componenti in Java

Ogni volta che il programmatore crea un componente *AWT* e lo inserisce in un’interfaccia grafica, il sistema *AWT* posiziona sullo schermo un oggetto grafico.

Qualsiasi volta che l’utente manipola un elemento dell’interfaccia grafica, un’apposita routine (scritta in codice nativo) crea un apposito oggetto Event e lo inoltra al corrispondente oggetto Java (ascoltatore), in modo da permettere al programmatore di gestire il dialogo con il componente e le azioni dell’utente con una sintassi completamente *Object Oriented* e indipendente dal sistema sottostante.

Il limite più grosso di *AWT* è che questo delega al sistema operativo la creazione e l’implementazione interna delle finestre e dei componenti, mentre il più moderno *Swing* utilizza, dovunque possibile, codice scritto direttamente in Java.

Per questo motivo, nel 1998, con l’uscita del *JDK 1.2*, venne introdotto il package **Swing**, i cui componenti sono stati realizzati completamente in Java, ricorrendo unicamente alle primitive di disegno più semplici, tipo “traccia una linea” o “disegna un cerchio”. Il codice java che disegna un pulsante *Swing* sullo schermo di un PC produrrà lo stesso identico risultato su un *Mac* o su un sistema *Linux*. Questa architettura risolve alla radice i problemi di uniformità visuale, visto che la stessa identica libreria viene ora utilizzata, senza alcuna modifica, su qualunque *JVM*.

Le classi *Swing* sono definite nel pacchetto *javax.swing*, mentre le *AWT* in *java.awt*.

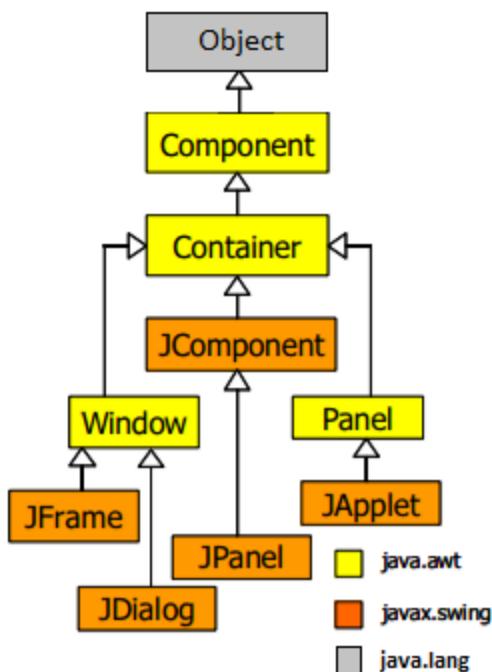


Figura 2.2 - Diagramma UML di base del package SWING

La figura 2.2 riassume molto sinteticamente le classi base del package `javax.swing` e come queste derivino da classi AWT.

Ogni oggetto grafico (una finestra, un bottone, un campo di testo, ecc ...) è implementato come classe del package `javax.swing`. Come si può notare tutte le classi derivano dalla classe "radice" `Object` contenuta nel package `java.lang`.

### 2.1.2 I Container

Un oggetto **Container** è un componente che può a sua volta contenere altri componenti. Esso tiene traccia dei **Component** che vengono aggiunti al suo interno attraverso una lista, la quale definirà anche l'ordine con cui saranno posizionati.

Tra i metodi messi a disposizione dalla classe **Container**, vale la pena citare:

- `add(Component comp);`
- `add(Component comp, int index);`

Se il metodo `add` viene invocato senza fornire un indice, il componente sarà aggiunto alla fine della lista (e quindi mostrato per ultimo).

### 2.1.3 I Top Level Container

I top level container sono i componenti all'interno dei quali si creano le interfacce grafiche: ogni programma grafico ne possiede almeno uno, di solito un **Frame**, che rappresenta la finestra principale. Un frame viene solitamente

usato come finestra principale per il programma. La disposizione degli elementi all'interno del frame viene gestita dai **Layout Manager** messi a disposizione da Java.

#### 2.1.4 I Component

Un **Component** è un oggetto avente una rappresentazione grafica che può essere mostrata sullo schermo e può interagire con l'utente. Esempi di **Component** sono i **Button**, le **Checkbox** ed altri elementi tipici dell'interfaccia utente.

#### 2.1.5 La gestione degli eventi

La gestione degli eventi grafici in *Java* segue il paradigma **event-delegation** (conosciuto anche come *event-forwarding*). Ogni oggetto grafico è predisposto ad essere sollecitato in qualche modo dall'utente e ad ogni sollecitazione genera eventi che vengono inoltrati ad appositi ascoltatori, che reagiscono agli eventi secondo i desideri del programmatore.

L'*event-delegation* presenta il vantaggio di separare la sorgente degli eventi dal comportamento a essi associato: un componente non sa (e non è interessato a sapere) cosa avverrà al momento della sua sollecitazione: esso si limita a notificare ai propri ascoltatori che l'evento che essi attendevano è avvenuto, e questi provvederanno a produrre l'effetto desiderato.

## 2.2 Le differenze nello sviluppo di applicazioni in ambiente mobile

Sviluppare applicazioni in ambiente mobile, in particolare **un'applicazione nativa**, richiede la conoscenza dell'utilizzo di alcuni strumenti software ed ovviamente una buona conoscenza del linguaggio di programmazione con cui sarà sviluppata. L'applicazione, una volta completa, verrà installata sul dispositivo e l'utente finale potrà interagire con essa. Tuttavia un'applicazione per smartphone non deve necessariamente risiedere sullo smartphone stesso, infatti è possibile sviluppare **Applicazioni Web** a cui si accede direttamente dal browser del dispositivo digitandone *l'URL*.

Nei seguenti sottoparagrafi esamineremo entrambe le tipologie di sviluppo di applicazioni mobile, la cui fusione ha dato luogo a framework di sviluppo di **applicazioni ibride** come **PhoneGap**.

### 2.2.1 L'applicazione Nativa

Con il termine "Applicazione Nativa" si intende un'applicazione mobile, sviluppata con codice e libreria proprietarie, che viene scaricata (gratuitamente o a pagamento) e installata sugli smartphone.

E' un software realizzato ad hoc per una piattaforma. Ciò comporta l'utilizzo del linguaggio di programmazione adatto, l'installazione di una *SDK (Software Development Kit)* e la configurazione di eventuali piattaforme di sviluppo legate al sistema target. Per alcune piattaforme proprietarie è necessario utilizzare hardware adeguati per compilare le applicazioni.

#### Vantaggi

- **Interagire** con tutto l'**HW** e **SW** del dispositivo fornendo un maggiore controllo nella gestione degli eventi;
- Incremento delle **prestazioni**: velocità di accesso alle funzionalità, affidabilità e migliore reattività oltre che una risoluzione superiore che assicura un'esperienza migliore all'utente;
- Non richiedono una connessione internet, permettendo di lavorare **offline**;
- **Look & Feel simile** alle app già installate di default, dunque rispettano le caratteristiche distintive del design di ciascuna piattaforma;
- **Notifiche Push**: permettono di avvisare gli utenti e di attirare la loro attenzione ogni volta che lo desideriamo, come per esempio per avvisare di un nuovo contenuto oppure a scopo promozionale;
- **Visibilità** dell'app garantita dall'essere pubblicati su uno store.

#### Svantaggi

- **Enorme impiego di tempo** nello sviluppo;
- **Alta conoscenza del linguaggio** nativo di programmazione e conoscenza del dispositivo target;
- **Alto budget** da investire;
- Occupazione della **memoria** del dispositivo;
- Soggetta ad **approvazione** dal Market;

### 2.2.2 L'applicazioni Web

Con il termine "Applicazione Web" si intende un'applicazione risiedente in un Server Web alla quale si accede tramite un browser Internet o un altro programma con funzioni di navigazione operante secondo gli standard del World Wide Web. In generale si parla di applicazione Web quando la funzione svolta dalla pagina è più che la semplice consultazione: il suo contenuto è in genere dinamico ed interattivo. Un'applicazione Web si basa su elementi software standard, indipendenti dalle caratteristiche della particolare applicazione e dalla piattaforma software e hardware su cui viene eseguita. Questo permette agli sviluppatori di riutilizzare le conoscenze dei linguaggi per lo sviluppo di pagine web come **HTML**, **CSS** e **JAVASCRIPT** creando applicazioni che possono essere eseguite su qualunque smartphone, a patto che siano dotati di un browser web e di una connessione ad Internet.

#### Vantaggi

- **Manutenzione centralizzata:** poiché c'è solo un punto da cui vengono frui i servizi della Web App ( il server su cui è installata ) è immediato rendere disponibili aggiornamenti e upgrade a tutti;
- **Cross-platform:** utilizzabile su qualsiasi dispositivo;
- **Interfaccia adattabile** in base al dispositivo, utilizzando un qualsiasi browser;
- **Economica** e ridotti tempi di realizzazione;
- **Ridotta complessità:** molti dei linguaggi usati sono ben documentati ed esistono numerosi framework che ne rendono speditivo lo sviluppo;
- **Istallazione sul dispositivo non necessaria;**
- **Non** deve essere sottoposta al processo di **approvazione** del Market di turno;

#### Svantaggi

- **Limiti nelle capacità:** molto difficile riprodurre nel browser applicazioni molto complesse;
- Necessità delle **connessione** ad Internet: una web app ha bisogno di un server per recuperare informazioni e rendere disponibili determinate funzioni;
- Funzionalità limitate in quanto è **indipendente** dall'**HW** e **SW** del dispositivo;
- Impossibilità di pubblicazione sugli store e quindi una sostanziale **perdita di visibilità** che questi ultimi offrono;

### 2.2.3 L'applicazione Ibrida

Le "applicazioni ibride" sono applicazioni che hanno un'architettura che unisce caratteristiche delle applicazioni native a quelle delle applicazioni web creando uno strato intermedio tra le due. In questo modo, è possibile continuare a sviluppare applicazioni utilizzando linguaggi orientati al Web ed allo stesso tempo superare le limitazioni imposte da questa tipologia di sviluppo, sfruttando le **API** messe a disposizione dal framework<sup>2</sup> per accedere alle native del dispositivo come la fotocamera, il rilevatore *GPS*, l'accelerometro, ecc....

Distinguiamo tre fasi durante lo sviluppo di applicazioni ibride:

#### 1. L'applicazione web viene creata.

L'applicazione web viene creata sfruttando *HTML*, *CSS*, *JavaScript* e l'insieme di *API* messe a disposizione dal framework.

#### 2. L'applicazione web inserita in una Web View.

Nel livello che si colloca tra applicazione web ed applicazione nativa è presente una **WebView**<sup>3</sup> dentro alla quale sarà eseguita l'applicazione web. Questo livello è cruciale in quanto ha una duplice funzionalità:

- Comunicare con l'applicazione web attraverso le API
- Comunicare (attraverso codice nativo) con il dispositivo per soddisfare le richieste dall'applicazione web.

#### 3. Il codice sorgente dell'applicazione nativa viene generato.

Senza la necessità di apportare alcuna modifica, potrà essere generata l'applicazione nativa per ogni tipo di piattaforma supportata dal framework di sviluppo.

#### Vantaggi

- **Forte adattabilità** a diversi tipi di piattaforme;
- **Risparmio di tempo** e **denaro** per lo sviluppo rispetto ad un app nativa;
- **Non** necessaria una **connessione** ad Internet: funzionano in maniera autonoma anche offline senza collegarsi ad un server;
- **Accesso** alle **funzioni del dispositivo** in uso con qualche variazione;
- **Aggiornamenti più facili** e adattabili alle diverse piattaforme rispetto all'app nativa;
- Possibilità di **pubblicazione** sul market.

---

<sup>2</sup> PhoneGap `e un esempio di framework che permette di sviluppare applicazioni ibride

<sup>3</sup> Una **WebView** `e un elemento grafico dentro al quale viene mostrata una pagina web.

Svantaggi

- **Performance inferiori** rispetto alle app native;
- **Tempo di sviluppo superiore** rispetto alle web app;
- **Minore facilità d'utilizzo e stabilità** dell'applicazione,
- Necessita di essere installata sul dispositivo, occupando **memoria**;
- La **programmazione** è più **complessa** rispetto alle web app;
- Soggetta ad **approvazione** dal Market.

Tipo APP	Integrazione col device	Tempi e costi di sviluppo	Funzionamento Offline	Modalità di distribuzione	Approvazione dallo store	Performance
NATIVA	SI	ALTI	SI/a scelta	STORE	SI	ALTE
WEB	NO	BASSI	NO	Solo WEB	NO	BASSE
IBRIDA	SI	MEDI	Alcune parti	STORE	SI	BASSE

Figura 2.3 - Tabella riassuntiva delle caratteristiche delle applicazioni di sviluppo per mobile

### 2.3 Google Android e framework

**Android** è un sistema operativo open source per dispositivi mobili, basato sul kernel Linux. Fu inizialmente sviluppato da *Android Inc.*, startup acquisita nel 2005 da **Google**. Il 5 novembre 2007 l'Open Handset Alliance presentò Android, costruito sulla versione 2.6 del Kernel Linux.

La piattaforma è basata sul kernel Linux, usa il database *SQLite*, la libreria dedicata *SGL* per la grafica bidimensionale e supporta lo standard *OpenGL ES 2.0* per la grafica tridimensionale. Le applicazioni vengono eseguite tramite la *Dalvik virtual machine*, una Java virtual machine adattata per l'uso su dispositivi mobili. Android è fornito di una serie di applicazioni preinstallate: un browser, basato su *WebKit*, una rubrica e un calendario.

Per iniziare a programmare su Android bisogna utilizzare alcuni strumenti software, tutti reperibili su Internet a costo zero, ovvero:

- un **JDK**, il kit di sviluppo per la tradizionale programmazione Java, visto che questa è la tecnologia con cui realizzeremo i nostri programmi;
- un **IDE** (ambiente di sviluppo integrato) che includa possibilmente tutti gli strumenti necessari al programmatore. Al momento, gli *IDE* più usato per programmare con Android sono: **Android Studio**, la soluzione ufficiale e quindi preferibile, ed **Eclipse**, sempre meno utilizzato ma che gode ancora di un folto bacino di utenza.

### 2.3.1 L'Android SDK

Tra gli strumenti appena citati, non è stato nominato un elemento fondamentale che merita, però, una menzione speciale: l'**Android SDK**<sup>4</sup>. Questo è il vero pacchetto di strumenti che ci permetterà di vedere realizzati i nostri programmi per Android. Esso è composto da varie entità:

- **Android APIs**: il cuore dell'SDK, in quanto sono lo strumento da cui partire per programmare un'applicazione Android;
- **Development tools**: semplici tool che permettono di compilare, debuggare e testare in modo ottimale un'applicazione;
- **The Android Virtual Device Manager and Emulator**: l'SDK fornisce un emulatore che simula un dispositivo Android. Poichè il device emulabile è altamente personalizzabile sia come caratteristiche hardware che software, questo strumento è molto utile in fase di test.
- **Documentazione** dove è incluso anche del codice di esempio.

## 2.4 Introduzione a PhoneGap

In questo capitolo verrà introdotto **PhoneGap** e la sua architettura, saranno poi presentati gli strumenti di sviluppo utilizzati nel corso della mia attività e le tecnologie coinvolte per la realizzazione della miniaturizzazione del software.

Come già anticipato nel Capitolo 2.3.3 dove si parlava di **applicazioni ibride**, PhoneGap è proprio uno dei framework creati per sviluppare questa tipologia di applicazioni e che sarà ampiamente utilizzato nel processo di miniaturizzazione.

---

<sup>4</sup> Android Software Development Kit, reperibile gratuitamente da <https://developer.android.com/>

### 2.4.1 Che cos'è PhoneGap

PhoneGap è un framework open source creato da Nitobi Software che funge da ponte tra applicazioni native e applicazioni web.



Figura 2. 4 – Filosofia di PhoneGap

L'obiettivo è cercare di realizzare lo slogan "*Write once, Port Everywhere*" fornendo agli sviluppatori, anche sprovvisti di conoscenze specifiche riguardanti le diverse piattaforme, un insieme di librerie statiche che permetta loro di scrivere applicazioni cross-platform nel modo più semplice e veloce possibile. Sviluppata l'applicazione utilizzando **HTML**, **CSS** e **JAVASCRIPT**, ne viene fatto il **porting** verso le varie piattaforme installando gli ambienti di sviluppo relativi e compilando la web-app realizzata. Il framework fornisce diverse piattaforme mobile, tra cui: **Android**, **iOS**, e **Windows Phone**.

### 2.4.2 Come funziona

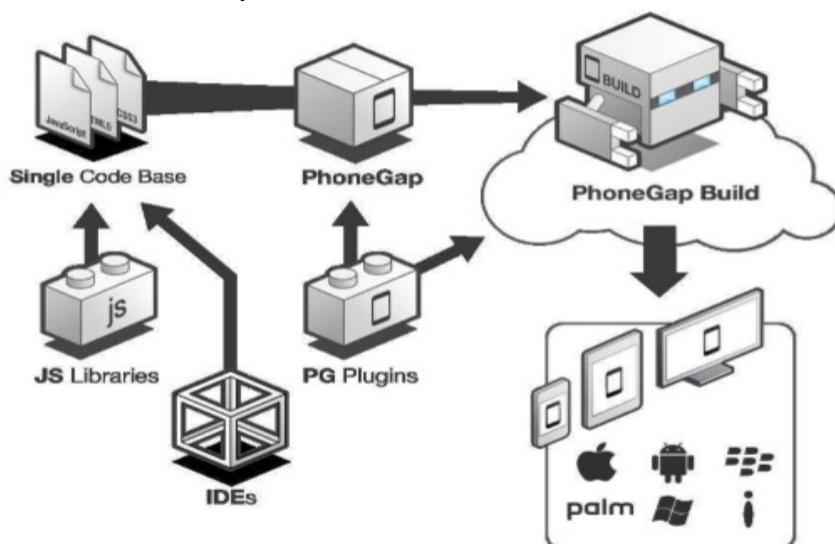


Figura 2. 5 - Il funzionamento di PhoneGap

**PhoneGap** non è altro che un **wrapper**, un contenitore, che permette agli sviluppatori di incorporare le applicazioni web all'interno di applicazioni native di diverse piattaforme. In questo modo, con la sola conoscenza di **HTML, CSS e JavaScript**, è possibile sviluppare un'applicazione web che ha il vantaggio di poter comunicare con il dispositivo utilizzandone le librerie native, grazie alle **API** fornite dal framework PhoneGap, al quale viene delegato il compito di tradurre le richieste ricevute tramite le API nel codice nativo della piattaforma su cui è eseguita l'applicazione, in modo del tutto trasparente allo sviluppatore.

Phonegap fa da ponte tra il sistema operativo e la web application realizzata dallo sviluppatore. In base alla tipologia di piattaforma con la quale dovrà interfacciarsi, l'implementazione di aggancio (fornita dallo stesso framework) sarà di conseguenza sviluppata in *Objective C* per *iPhone*, in *Java* per *Android*, e così via.

In pratica esiste un runtime basato su *WebKit*, in cui vengono iniettate le componenti statiche. Il risultato sarà un pacchetto composto da due elementi principali con differenti responsabilità che però cooperano tra loro per fornire delle funzioni a valore aggiunto. Nel caso specifico il runtime si occupa di dialogare direttamente con il dispositivo e le parti statiche offrono l'interfaccia verso l'utente. L'uso di *JavaScript* e di *AJAX* consente poi di dare vita alle applicazioni che possono avere comportamenti complessi e comunicazioni verso endpoint remoti realizzando di fatto un'applicazione completa in cui il client è un elemento fondamentale in linea con le moderne applicazioni Web 2.0. Il funzionamento è schematizzato nella figura 2.5.

#### 2.4.3        *Il plugin di comunicazione in PhoneGap*

Esplorando le API di PhoneGap abbiamo avuto modo di vedere come coprire la maggior parte delle esigenze di sviluppo di un'app mobile. Le API ci consentono di utilizzare funzioni native mediante *JavaScript* ignorando i dettagli implementativi di ciascuna piattaforma mobile. Ma cosa fare se abbiamo bisogno di una funzionalità nativa non prevista dalle API? In questo caso possiamo far ricorso ai plugin.

Un **plugin** è un componente software che consente di mappare una funzione *JavaScript* in una funzionalità nativa: esso è costituito da un'unica interfaccia *JavaScript* e da tante implementazioni dipendenti dalla specifica piattaforma mobile. In realtà tutte le API di PhoneGap sono implementate secondo questo modello, quindi un plugin non fa altro che estendere le API aggiungendo nuove funzionalità (figura 2.6).

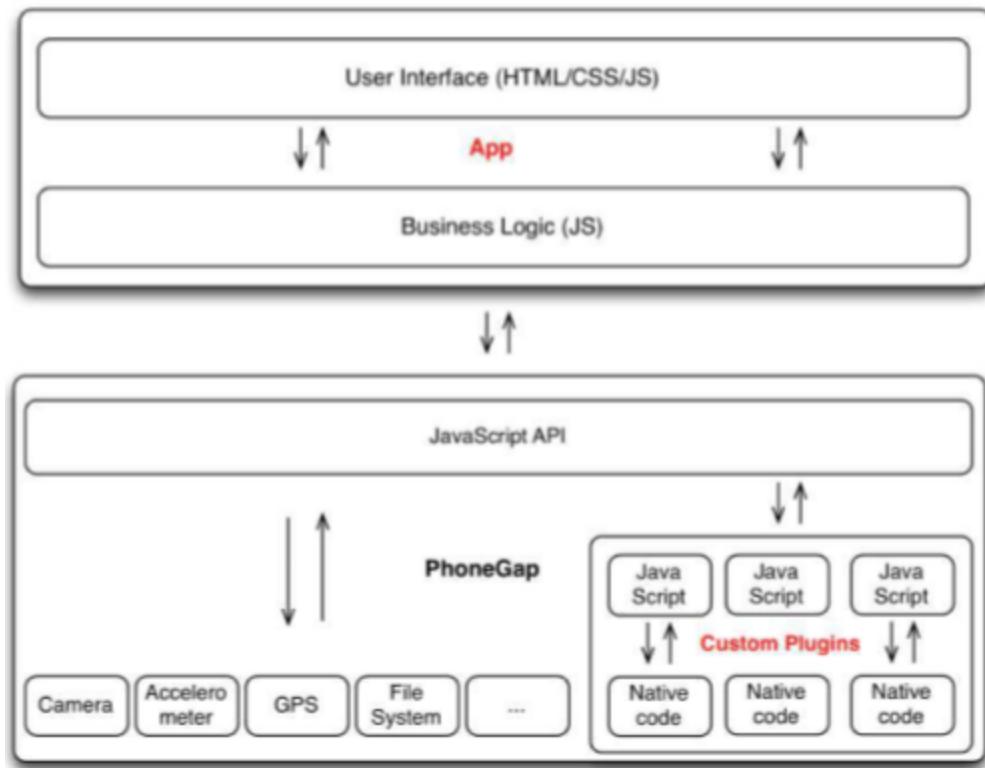


Figura 2. 6 - Schema architettonico PhoneGap con plugin

#### 2.4.4 L'utilizzo del plugin nel processo di miniaturizzazione

Come verrà spiegato in maggior dettaglio nei prossimi capitoli, la nuova interfaccia grafica in *HTML* che verrà generata durante la fase di parsing del processo di miniaturizzazione, sarà dunque gestita da PhoneGap, che si occuperà della sua visualizzazione sul dispositivo mobile.

Il plugin si comporterà come un **adapter** tra la logica dell'applicazione originale e la nuova interfaccia grafica in *HTML*, mantenendo attive le comunicazioni con il codice nativo: ogni componente grafico presente nella vecchia *UI* sarà sostituito da uno in *HTML* che ne preserverà le funzionalità originali.

#### 2.4.5 La struttura di un progetto PhoneGap

Affinché un progetto PhoneGap possa essere compilato correttamente è necessario che questo contenga directory e file dai nomi predeterminati. Tale struttura (figura 2.7) viene creata in parte dall'*ADT*, al momento della creazione di un normale progetto Android.

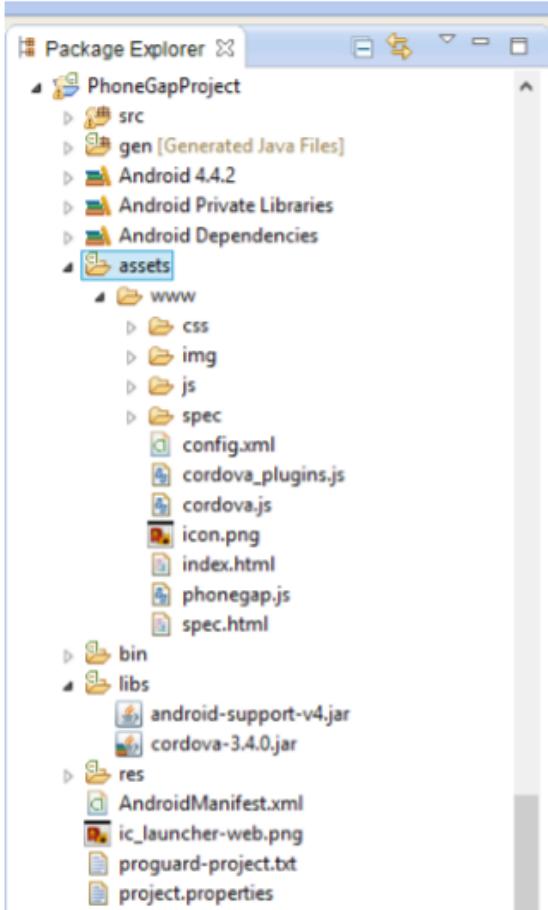


Figura 2. 7 - Struttura di un progetto PhoneGap

Analizziamo file e cartelle contenute nel progetto:

- **src/main**: directory contenente i sorgenti;
- **gen**: directory contenente i file generati dall'ADT. Il file R.java, in particolare contiene un riferimento ad ogni risorsa contenuta nella cartella res del progetto;
- **android 4.4.2 e android-support-v4.jar**: librerie di Android che contengono le API per la versione del sistema operativo 4.4.2 (*Android KitKat*);
- **res**: directory contenente tutte le risorse dell'applicazione come, ad esempio, immagini, layout e stringhe di test di un progetto Android. Le risorse sono definite utilizzando il formato XML;
- **assets/www**: il cuore di tutto il progetto, dove si trovano i files HTML e JavaScript (con relative librerie) che vengono utilizzati per la creazione e gestione dell'interfaccia grafica. In particolare, il file **index.html** rappresenta una normale pagina HTML in cui è definito il layout dell'interfaccia utente. Nella cartella **css** saranno aggiunti i fogli di stile, in fonts eventuali font aggiuntivi, mentre in **/img** le immagini ed in **/js** il

codice *JavaScript* che interfaccia la nostra applicazione con l'infrastruttura nativa attraverso l'uso del plugin. Notiamo, infine, il file **config.xml** che contiene le impostazioni relative alla nostra app.

- **libs**: directory in cui sono inserite le librerie aggiuntive di *Cordova-PhoneGap* che contengono le *API* per la versione 3.4.0. Il file è stato aggiunto manualmente nel progetto;
- **AndroidManifest.xml**: il file che descrive l'applicazione e dichiara tutti i suoi componenti.

## 2.5 La libreria Hammer.js

**Hammer** è una libreria *JavaScript* che permette di interagire con i contenuti Web attraverso azioni classiche eseguite su un touch screen: *tap*, *hold*, *drag*, *ping*, *swipe*, ecc. Tipicamente una pagina web non intercetta tutti gli eventi generati sullo schermo di smartphone e tablet, a parte il semplice *tap*<sup>5</sup>.



Figura 2. 8 - Esempi di gestures supportate da hammer.js

La **compatibilità con i browser più diffusi** (compresi quelli mobile), insieme alla **semplicità di utilizzo**, determinano il punto di forza di questa libreria. Nella pagina ufficiale di Hammer<sup>6</sup>, è possibile effettuare il download, consultare la documentazione e i relativi esempi.

Per iniziare ad utilizzare la libreria, basta includerla alla pagina *HTML* e creare una nuova istanza.

```

1. var hammertime = new Hammer(ElementoHTML);
2. hammertime.on('swipeleft', function(ev){
3.     console.log(
4.         "Rilevato swipe verso sinistra sull'oggetto
5.         "+ElementoHTML);
6. });

```

Come si può intuire dal codice auto-esplicativo, quando in prossimità di un certo elemento della pagina, in questo caso **ElementoHTML**, viene fatto uno **swipe** verso sinistra, viene stampato un messaggio nella console.

<sup>5</sup> Il tap `e la gestione equivalente al click, viene eseguita attraverso il touch.

<sup>6</sup> Pagina ufficiale: <http://hammerjs.github.io>.

# 3. Il processo di miniaturizzazione

---

L'obiettivo prefissato di questo progetto è quello di mettere in pratica un buona strategia per effettuare il porting da un'applicazione Java ad una applicazione Android e sviluppare la tecnologia di supporto utile per la realizzazione di tale compito.

Questo richiede la definizione e l'implementazione di un middleware di comunicazione per collegare la nuova interfaccia utente con tutte le funzioni presenti all'interno del codice considerato. Tutte le operazioni visualizzate dall'interfaccia devono essere reindirizzate al componente middleware, che stabilisce un collegamento, una comunicazione tra la nuova interfaccia utente migrata e la business logic dell'applicazione Java.

Come analizzato nel capitolo 2.2 il progetto può considerarsi una applicazione tecnicamente ibrida:

- miglioramento delle performance in quanto la logica è nativa;
- minimizzazione dello sviluppo, opera su un'applicazione già esistente .

## 3.1 La strategia e il processo di migrazione

Effettuare il **porting** da un'applicazione Java ad una applicazione Android, non implica una riprogrammazione radicale, ma piuttosto una sorta di adattamento del codice alla nuova piattaforma, lasciando inalterati i requisiti di carattere funzionale del sistema.

L'adattamento è dovuto sostanzialmente al fatto che Android e Java non forniscono lo stesso development kit, e quindi le stesse *API*. In questo senso la parte di user interface e data storage deve essere completamente riscritta. Per quel che riguarda la UI, l'*SDK* di Android prevede l'utilizzo dei componenti Activiy, Fragment, Layout e Widget, mentre nei software Java si utilizzano le AWT o le più recenti *Swing*.

L'idea è quella di implementare un wrap per i componenti grafici di Java, mantenendone comunque inalterata la logica: la nuova *GUI* includerà gli oggetti della classe AWT o *Swing* presenti all'interno del codice in grado di accedere comunque alle funzionalità originali attraverso il componente middleware.

Il motore di tutto il processo di migrazione può essere concentrato sul lavoro di parsing del programma Java di partenza, per la generazione dei nuovi componenti grafici che verranno elaborati da PhoneGap e la **generazione dinamica** dei file *HTML* necessari per la costruzione di una nuova interfaccia che permetta al programma di gestire gli eventi associati ad ogni elemento grafico presente, ad esempio: frame, button, textbox, dialog, ecc...

Il plugin generato sarà in grado di mettere in comunicazione l'*HTML* con Java in entrata e uscita, attraverso delle funzioni *JavaScript*. Vedremo in seguito come avviene la **comunicazione bidirezionale** tra questi due mondi e come sono strutturate le classi che compongono la libreria di supporto creata per gestione dei componenti e degli eventi collegati.

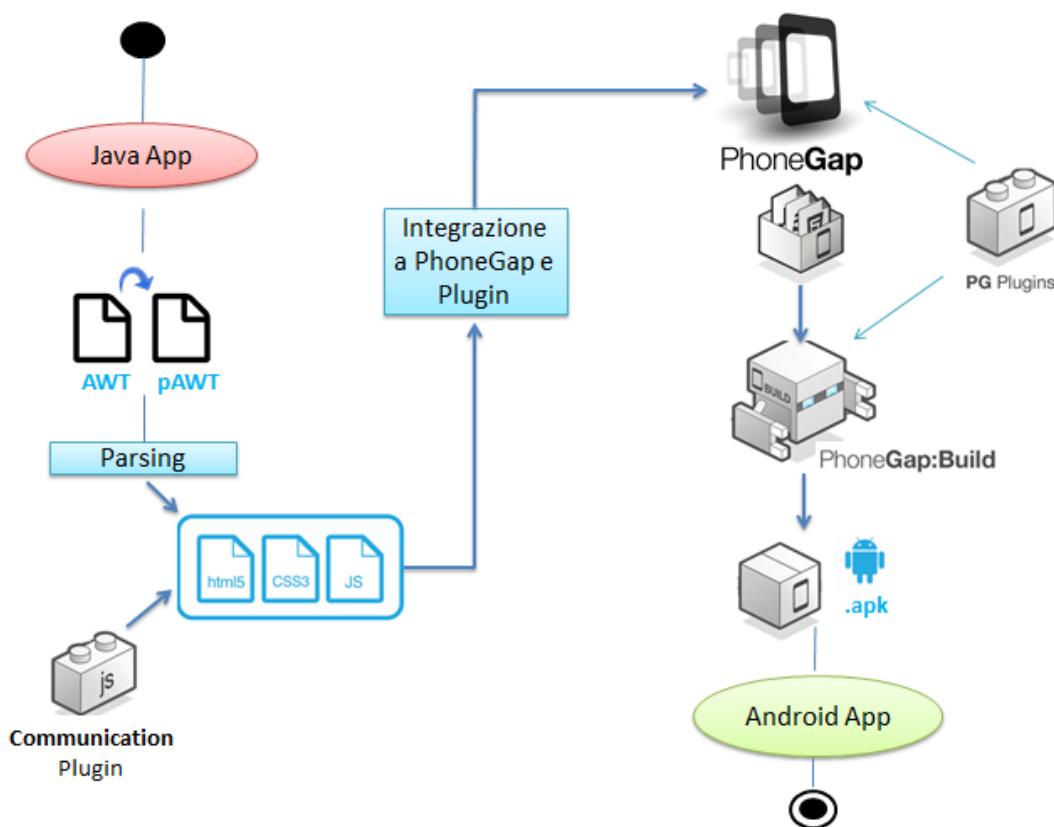


Figura 3.1 - Diagramma delle attività del processo di miniaturizzazione

La figura 3.1 mostra l'intero processo svolto come un diagramma delle attività, dove vengono riportati tutti gli step principali per la miniaturizzazione e le sotto-attività necessarie per la conversione di un software scritto in Java verso il sistema operativo mobile Android.

### 3.2 La libreria di supporto pAwT

Partendo da un programma Java, il processo di trasposizione richiede di determinare tutti i punti che non vanno più bene con l'applicazione mobile: per quanto riguarda l'interfaccia grafica, si è cercato, quindi, di creare un mapping tra le classi AWT o Swing (ad esempio Frame, Button, Textfield, ecc...) ed una loro corrispondente classe definita nella libreria di supporto che permetta di interagire ed operare con PhoneGap.

È stato necessario creare un **wrapper** per questi oggetti: ogni componente grafico presente, appartenente alle classi Java sarà opportunamente rinominato e sostituito con altri oggetti che costituiscono la libreria di supporto importata.

Analizziamo un esempio contenente componenti grafici AWT, come in figura:

```
import java.awt.*;
import mini.pawt.*;

public class Example{
    private Frame mainFrame;
    private Button button1;
    private TextField tf1;

    public Example(){
        mainFrame=new Frame("Java AWT Example");
        button1=new Button("Button1");
        tf1=new TextField(3);
        mainFrame.add(button1);
        mainFrame.add(tf1);
    }

    button1.addActionListener(
        new ActionListener() {
            public void actionPerformed(ActionEvent e){
                myFunction(a.getText(),b.getText());
            }
        });
}

import mini.pawt.*;
public class Example{
    private pFrame mainFrame;
    private pButton button1;
    private pTextField tf1;

    public Example(){
        mainFrame=new pFrame("Java AWT Example");
        button1=new pButton("Button1");
        tf1=new pTextField(3);
        mainFrame.add(button1);
        mainFrame.add(tf1);
    }

    button1.addActionListener(
        new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                myFunction(a.getText(),b.getText());
            }
        });
}
```

Figura 3. 2 - Esempio del cambiamento apportato al codice

La figura 3.2 mostra il cambiamento effettuato nel codice di una classe generica di esempio, dove vengono dichiarati, inizializzati ed aggiunti al frame principale alcuni elementi grafici che appartengono ad AWT di Java. Le frecce gialle evidenziano i cambiamenti effettuati, cioè l'operazione di rinomina di tali componenti (da Frame a pFrame, da Button a pButton, ecc...), lasciando completamente inalterato il codice dell'evento associato e la funzione ad esso collegata (myFunction).

Queste nuove classi, organizzate in un framework, andranno a mappare negli oggetti grafici compatibili con PhoneGap, mantenendo la stessa logica e funzionalità della classe di appartenenza, in modo tale da permettere agli eventi associati di operare correttamente con la nuova interfaccia.

Tale processo di modifica potrà essere facilmente automatizzato in futuro, cioè facendo in modo che attraverso un processo di strumentazione del codice sia possibile run-time questa operazione.

Possiamo definire le **pAWT** come un vero e proprio strumento di comunicazione tra l'applicazione Java nativa e la UI migrata, in entrata e in uscita.

Le classi che compongono questa libreria hanno, dunque, la funzione di mappare un elemento grafico appartenente alle classi di Java, con il rispettivo elemento *HTML*, replicando il comportamento degli oggetti grafici Java e, quindi, di gestire e modellare l'interfaccia grafica *HTML*.

### 3.2.1 La gerarchia delle pAWT

Il package **mini.pawt** fornisce una serie di classi wrapper che ruotano intorno alla libreria grafica di Java. Nella figura è raffigurata la gerarchia delle pAWT: ogni classe che rappresenta il componente presente (frame, pulsanti, campi di testo, ecc...) deriva dalla classe astratta **pComponent**, dentro la quale sono definiti dei metodi vuoti per la gestione degli eventi che dovranno essere implementati dalle sottoclassi.

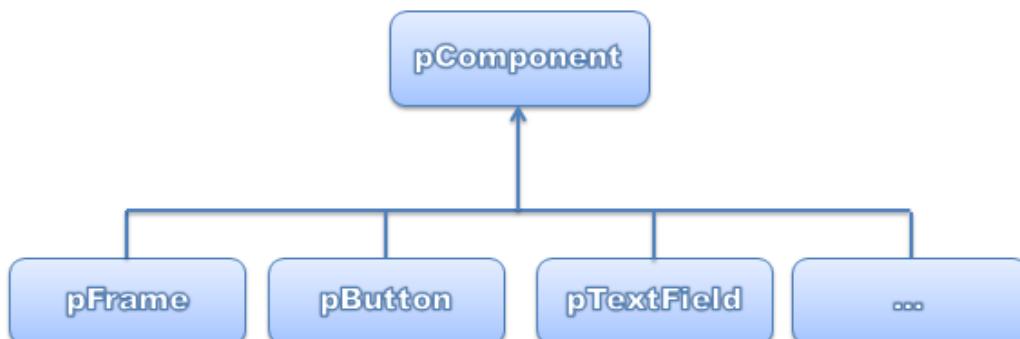


Figura 3. 3 - Gerarchia delle pAWT

L'elemento fondamentale che compone l'interfaccia grafica è, dunque, il componente: ogni oggetto grafico che apparirà sul display del nostro dispositivo mobile, come la finestra principale, pulsanti, campi di testo, checkbox, ecc... è un componente, ed ognuno di questi dovrà essere inserito dentro un frame "contenitore", che li raggruppi.

Per questo motivo, la classe **pFrame**, è forse la classe più importante della gerarchia: oltre a svolgere la funzione "contenitore", essa si occupa del dispatch degli eventi agli altri componenti.

### 3.2.2

### *Il repository*

Tutti gli elementi di tipo pComponent che vengono aggiunti man mano al mainFrame principale durante il normale flusso di esecuzione dell'applicazione Java, verranno memorizzati in una struttura dati di supporto appositamente creata e contenuta nella classe pFrame.

In particolare, ogni volta che bisogna aggiungere un componente alla classe pFrame, bisogna chiamare il metodo *void add(pComponent c)* che avrà il compito di richiamare il metodo addComponent della classe Repository che avrà lo scopo di aggiungere il componente in una struttura Hash Table.

```
1. Repository rep = new Repository();  
2. public void add(pComponent c) {  
3.     rep.addComponent(c._title, c);  
4. }
```

La funzione che si occupa di inserire l'elemento nella struttura appena creata è la seguente:

```
1. class Repository {  
2.     Hashtable<String, pComponent> components = new  
        Hashtable<String, pComponent>();  
3.     public void addComponent(String key, pComponent component) {  
4.         components.put(key, component);  
5.     }  
6.     ...  
7. }
```

Ogni elemento grafico di tipo pComponent presente nella tabella hash, sarà così richiamabile in qualunque momento, utilizzando la chiave specificata dalla variabile **key** di tipo **String** che contiene l'id del componente.

### 3.2.3

### *Come recuperare il frame radice*

Durante la fase di parsing vengono analizzate tutte le classi Java del programma da miniaturizzare, che implementano i componenti grafici. L'approccio standard per eseguire il porting della UI prevede che ogni schermata dell'applicazione (come, ad esempio un oggetto *java.awt.Frame*, nel caso di AWT) venga convertita in una nuova Activity in Android. Ogni volta che si usa un'app generalmente si interagisce con una o più "pagine", mediante le quali si consultano dati o si immettono input.

È possibile, quindi, che l'applicazione Java da voler miniaturizzare contenga più di un frame, ed è necessario generare un metodo per il recupero dello stesso, per ciascun frame presente nelle classi.

C'è bisogno, dunque, di un metodo che si occupi di recuperare l'elemento radice della gerarchia degli elementi grafici, in modo da poter interagire con tutto l'albero dei componenti. Il lavoro del parser sarà quello di aggiungere questo metodo in ogni classe in cui è presente un frame, come in figura 3.4:

```

import mini.pawt.*;

public class Example{

    private pFrame mainFrame;
    private pButton button1;
    private pTextField tf1;

    public Example(){
        mainFrame=new pFrame("Java AWT Example");
        button1=new pButton("Button1");
        tf1=new pTextField(3);
        mainFrame.add(button1);
        mainFrame.add(tf1);
    }

    button1.addActionListener(
        new ActionListener() {
            public void actionPerformed(ActionEvent e){
                myFunction(a.getText(),b.getText());
            }
        }
    );
}
}


public pComponent getTopComponent(){
    return mainFrame;
}
}

```

Figura 3. 4 - Aggiunta del metodo `getTopComponent()` nel codice

dopo la sostituzione dei componenti grafici di Java con i pComponent che costituiscono la nuova libreria creata, come abbiamo visto in precedenza, durante la fase di parsing viene aggiunto il metodo `getTopComponent()`, che restituisce l'elemento al vertice della gerarchia.

### 3.2.4 La gestione degli eventi nelle pAWT

Ogni volta che l'utente esegue un'azione, un clic del mouse, la pressione di un tasto sulla tastiera, la modifica di una finestra o la selezione di un elemento da un menu, viene generato un evento

Per gestire gli eventi bisogna distinguere tra:

- **Sorgente dell'evento (source)** la componente dell'interfaccia utente che ha generato l'evento Pulsanti, voci di menu, barre di scorrimento, etc
- **Ricevitore dell'evento (listener)** la classe che intercetta l'evento e definisce le azioni intraprese a seguito del verificarsi di particolari eventi

Il modello ad eventi di Java segue un'architettura molto semplice e intuitiva dal punto di vista implementativo. Tale modello prevede la possibilità per ogni componente di generare un evento. Ad ogni evento generato corrisponde una classe specifica, che ne descrive l'origine e la causa dell'evento stesso, cioè viene fornito un resoconto completo sull'evento generato. La gestione logica dell'evento viene affidata ad una sua interfaccia specifica, detta listener, che contiene i metodi da implementare, come nell'esempio:

```
1. button1.addActionListener(  
2.     new ActionListener() {  
3.         public void actionPerformed(ActionEvent e) {  
4.             myFunction(...);  
5.         }  
6.     }  
7. );
```

Per quanto riguarda le classi che compongono la nostra libreria di supporto, **ActionListener** ed **ActionEvent** fanno parte di **pAWT** ed hanno funzionalità simili a quelle di Java. Quando si verifica un evento, cioè quando l'utente compie un'azione utilizzando uno dei componenti dell'interfaccia grafica, come la pressione di un tasto, o l'immissione di un testo, questo viene elaborato e gestito nella classe **pFrame**, che si occuperà del dispatch degli eventi ai suoi sotto-componenti.

Ad esempio, alla pressione di un bottone, il metodo *fire(dest)* del **pFrame** si occuperà del recupero del componente che ha generato l'evento, e ad invocare il metodo *fire()* sul componente destinatario dell'evento.

```
1. public void fire(String dest) {  
2.     rep.click(dest);  
3. }
```

Il compito sarà quindi delegato alla **Repository** (rep) che dopo aver trovato il componente dest (al suo interno o all'interno delle repository di altri componenti contenuti in essa), ne invoca il metodo *fire()*.

### 3.3 Il flusso di comunicazione

#### 3.3.1 Da PhoneGap verso Java

Ora che abbiamo introdotto le classi che fanno parte della libreria pAWT, cerchiamo di capire come avviene la comunicazione tra la Web View che costituisce l'interfaccia grafica di PhoneGap e queste classi (Figura 3.4).

Le *API* messe a disposizione da PhoneGap ci consentono di utilizzare funzioni native mediante JavaScript, ignorando i dettagli implementativi di ciascuna piattaforma mobile. Ma cosa fare se abbiamo bisogno di una funzionalità nativa non prevista dalle *API*? In questo caso possiamo far ricorso ai plugin.

Un **plugin** è un componente software che consente di mappare una funzione JavaScript al codice nativo, estendendo le *API* per aggiungere nuove funzionalità. Questo è alla base del sistema di comunicazione (bidirezionale) tra l'interfaccia grafica in PhoneGap(Web View) e la nostra applicazione nativa.

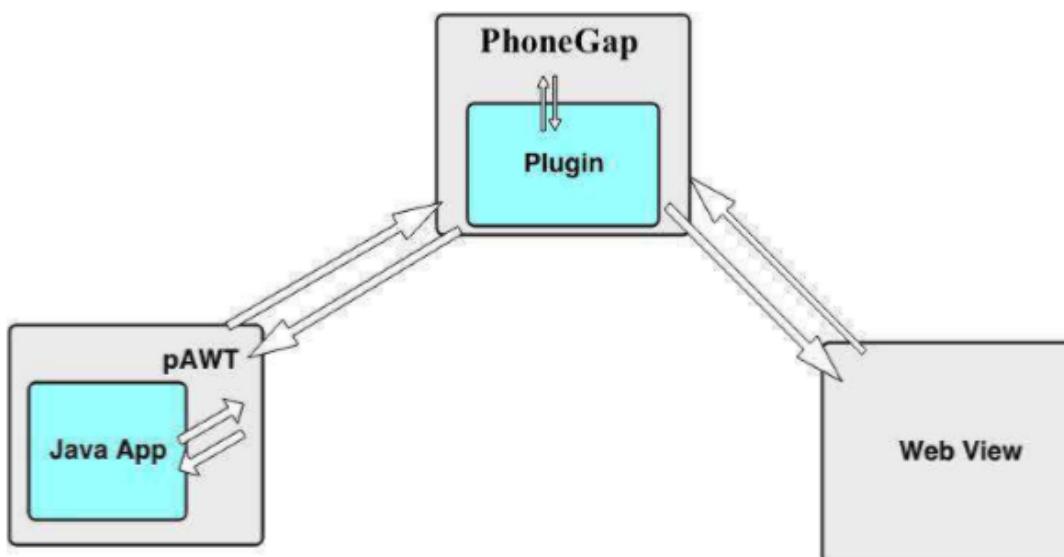


Figura 3. 5 - Schema del processo di comunicazione

Grazie alle classi “wrapper” appartenenti alla libreria delle pAWT, si ha la possibilità di interagire (in entrata e in uscita) con l'interfaccia, attraverso la mappatura delle funzioni *JavaScript* associate ai componenti *HTML* presenti nella UI migrata, con i corrispettivi componenti pAWT. Tali funzioni permettono al plugin di istanziare degli oggetti in linguaggio nativo, passare dei parametri e riprendere dei valori da passare all'interfaccia utente.

PhoneGap attraverso la chiamata al metodo *cordova.exec()* permette il passaggio di controllo al codice Java. Quando per cui il codice *JavaScript* associato agli elementi grafici presenti nel codice *HTML* andrà a chiamare questo metodo.

Ad esempio, il codice *HTML* di un button sarà creato in questo modo:

```
<BUTTON class="button" id="sum" onClick="app.fire(id)" type="button">sum</BUTTON>
```

Si nota che *onClick* chiama la funzione *app.fire(id)*.

```
1. fire: function(id,successCallback, errorCallback) {
2.         cordova.exec(
3.             successCallback,
4.             errorCallback,
5.             'AWTPlugin',
6.             'fire',
7.             [
8.                 {
9.                     "id": id
10.                }
11. }
```

I primi due parametri specificano che funzioni eseguire in caso di successo o errore lato codice *JavaScript*, una stringa che rappresenta la classe o il servizio sottostante (“*AWTPlugin*” nel nostro caso), un’altra stringa che indica l’evento generato dal componente e un array di parametri passati dalla funzione, in cui viene specificato l’id del componente *HTML* che lo ha generato.

In questo caso si tratta del click di un bottone: il plugin, dopo che gli è stato restituito l’elemento radice (*mainFrame*) attraverso la chiamata al metodo *getTopComponent()* è in grado di invocare l’evento “*fire*” della classe *pFrame*, dopo aver recuperato nella tabella hash il *pButton* richiesto, identificato dal suo *id*.

Al metodo *exec* visto precedentemente verrà passato l’id del bottone che ha scatenato l’evento (in questo caso “*button1*”) e l’evento “*fire*” che verrà catturato dalla classe *pFrame*, che ha il compito di invocare il metodo *fire()* sull’oggetto *pButton* identificato da suo *id*.

Per quanto riguarda gli elementi grafici che richiedono l’immissione di informazioni da parte dell’utente, come avviene, ad esempio per un campo di testo, una label, una checkbox, ecc... occorrerà, in tal caso, passare informazioni aggiuntive alla funzione *exec* che riguardano i campi immessi in *input*.

Analizziamo il caso di un campo di testo, rappresentato dal seguente codice HTML:

```
1. <INPUT id="equals"
2. onkeyup="app.insert(id,document.getElementById(id).value)"
3. placeholder="equals" type="text">
```

L'evento *onkeyup* che viene eseguito ogni volta che l'utente rilascia un tasto della tastiera, è in grado di invocare la funzione *JavaScript insert(id,document.getElementById(id).value)* alla quale, oltre all'identificativo del componente che ha generato l'evento ("textfield1" in questo caso), verrà passato all'array della funzione *exec()* anche il valore immesso da tastiera. Il metodo *document.getElementById(id)* restituisce l'elemento avente l'id specificato.

Vediamo ora come cambia la funzione *cordova.exec()*, con il passaggio delle informazioni inserite dall'utente:

```
1. insert: function(id,a,successCallback, errorCallback) {
2.   cordova.exec(
3.     successCallback,
4.     errorCallback,
5.     'AWTPPlugin',
6.     'insert',
7.     [
8.       {
9.         "id": id,
10.        "element": a
11.      }
12.    ]
13.  );
14.}
```

Oltre all'id del componente *HTML* che ha generato l'evento, abbiamo bisogno anche dell'informazione immessa dall'utente per poterla elaborare. Il plugin generato provvederà al passaggio di tale informazione alla classe pFrame, la quale, a sua volta, dopo aver recuperato dalla tabella hash la pTextField identificata dall'id passato per argomento, invocherà la funzione *setText()* sull'oggetto pTextField destinatario dell'evento.

### 3.3.2 Da Java verso PhoneGap

Abbiamo visto, dunque, come è possibile far comunicare l'*HTML* di **PhoneGap** con *Java*, cioè attraverso l'uso del plugin ed alcune funzioni *JavaScript* di supporto.

Ma ciò che rende possibile la comunicazione con l'ambiente nativo in entrata ed in uscita è la funzione *sendJavascript()*, inclusa nelle librerie di Cordova: questa

permette l'invocazione di una funzione *JavaScript* direttamente dal codice nativo, utile per modificare l'interfaccia *HTML* dalla classe del componente *pAWT* che lo desidera.

Come è emerso dal precedente esempio del campo di testo, abbiamo bisogno di poter operare sull'interfaccia direttamente dalla classe *pTextField*, per poter inviare l'informazione richiesta dall'utente all'interfaccia grafica *HTML*.

La funzione *sendJavascript()* è in grado di invocare una funzione *JavaScript* (lato PhoneGap), che sarà in grado di operare sul componente *HTML* identificato dal suo id passato come parametro ed il valore da voler restituire, indicato da "value":

```
1. public void setText(String value) {  
2.     if(!value.equals(_value)) {  
3.         _value = value;  
4.         try {  
5.             CordovaWebView wv = Constants.view;  
6.             if(wv != null){  
7.                 wv.sendJavascript("app.setta('"  
8.                     +_title+"', '"  
9.                     +_value+"')");  
10.            }  
11.        } catch(Exception e) {}  
12.    }  
13. }
```

La funzione *JavaScritp* che andrà a modificare e scrivere sull'interfaccia *HTML* è la seguente:

```
1. setta: function(id, val) {  
2.     var obj = document.getElementById(id);  
3.     if(obj) {  
4.         if( obj.value != val)  
5.             obj.value = val;  
6.     }  
7. }
```

### 3.4 L'implementazione del plugin di comunicazione

Per implementare il plugin lato codice nativo su piattaforma mobile Android, occorre una classe Java che estenda la classe “**CordovaPlugin**” di PhoneGap, in cui viene effettuato l’override di uno dei suoi metodi, cioè “*execute()*”, che è quello che verrà richiamato dal JavaScript engine con la *exec()* vista precedentemente.

Il metodo, che sarà il cuore del nostro plugin è così definito:

```
1. public boolean execute(String action, JSONArray args,
2.                           CallbackContext callbackContext)
```

Il primo parametro *action* indica l’azione che vogliamo far eseguire al nostro plugin, il secondo parametro, invece, rappresenta un array JSON di parametri passati dalla funzione JavaScript *exec()* e la callback da dover restituire in caso di fallimento o di successo.

Facendo riferimento agli esempi precedenti, per quanto riguarda gli eventi scatenati da un bottone e da un campo di testo, un’implementazione di questo metodo potrebbe essere la seguente:

```
1. public boolean execute(String action, JSONArray args,
2.                           CallbackContext callbackContext) throws JSONException {
3.
4.     try {
5.         Constants.view = webView;
6.         Log.d("awtrun", "Plugin action call! : " + action);
7.
8.         if (action.equals("insert")) {
9.             JSONObject arg_object = args.getJSONObject(0);
10.            String id = arg_object.getString("id");
11.            String value = arg_object.getString("element");
12.            component.change(id, value);
13.        } else if (action.equals("fire")) {
14.            JSONObject arg_object = args.getJSONObject(0);
15.            String id = arg_object.getString("id");
16.            component.fire(id);
17.        }
18.
19.        return true;
20.    } catch (Exception e) {
21.        callbackContext.error(e.getMessage());
22.        return false;
23.    }
24. }
```

Viene, dunque, effettuato il dispatch dei metodi: in caso di inserimento da parte dell’utente in un campo di testo presente nella UI grafica, la funzione JavaScript

invocata dal lato *HTML*, sarà la *change()*, oppure la *fire()* nel caso di un click su un bottone, come abbiamo visto precedentemente.

Vengono quindi recuperate tutte le informazioni utili dall'array *JSON*, come l'id del componente che ha generato l'evento, oppure il testo immesso dall'utente (indicato dalla chiave "*myElement*"), invocando il rispettivo metodo delle classi *pTextField* o *pButton* appartenenti alla libreria delle *pAWT* (*change()* o *fire()*, a seconda che si tratti di un inserimento in un campo di testo, o il click di un bottone). Il tutto sarà gestito, come spiegato precedentemente, nella classe *pFrame* che si occuperà del recupero del componente dalla tabella hash e dell'invocazione del metodo su quell'oggetto.

La funzione *change()* contenuta nella classe *pFrame*, è la seguente:

```
1. public void change(String dest, String value) {  
2.     rep.change(dest,value);  
3. }
```

Rep è il riferimento alla classe *Repository*. Il metoto *change()* della classe *Repository* avrà il ruolo di ricercare il componente nella hashtable e richiamare il metoo *setText()* del componente.

```
1. public void change(String key, String value) {  
2.     pComponent component = find(key);  
3.     if(component != null && component instanceof pTextField){  
4.         pTextField f = (pTextField)component;  
5.         f.setText(value);  
6.     }  
7. }
```

### 3.5 Il porting della UI

Dopo aver chiarito come avviene la comunicazione tra l'HTML di PhoneGap e Java nativo (e viceversa), occupiamoci della realizzazione della pagina HTML, che contiene l'interfaccia migrata su PhoneGap. La creazione del file HTML, che definisce l'intero layout dell'interfaccia in PhoneGap, viene generata dinamicamente in base al numero degli oggetti grafici presenti nel programma di origine: ogni elemento contenuto nel codice Java dovrà avere una sua corrispondenza in codice HTML.

```
.....  
public Example(){  
.....  
    button1=new pButton("button1");  
    textfield1=new pTextField(7);  
.....  
}  
  
<HTML>  
<HEAD>  
<TITLE>  
JAVA AWT Example  
</TITLE>  
</HEAD>  
<BODY>  
<BUTTON id="button1" onClick="function()" type="button">Title</BUTTON>  
<INPUT id="a" onkeyup="function1()" size="3" type="text">  
</BODY>  
</HTML>
```

Figura 3. 6 - Esempio di codice da Java ad HTML

#### 3.5.1 L'implementazione del parser DOM

Lo strumento più adatto per generare dinamicamente componenti in HTML è il **DOM** (*Document Object Model*), letteralmente modello a oggetti del documento, è una forma di rappresentazione dei documenti strutturati come modello orientato agli oggetti.

Quando, ad esempio, un nuovo pulsante ed il campo di testo verranno istanziati nel costruttore, il lavoro del parser sarà quello di inserire i due elementi nella sua giusta collocazione all'interno del DOM.

Allo scopo creiamo un parser DOM (che in Java è rappresentato dalla classe `DocumentBuilder`) sfruttando il metodo `newDocumentBuilder()` della classe `DocumentBuilderFactory`:

```
1. DocumentBuilder docFactory =  
   DocumentBuilderFactory.newInstance();  
2. DocumentBuilder docBuilder = docFactory.newDocumentBuilder();  
3. doc = docBuilder.newDocument();
```

In Java, utilizzando le librerie `org.w3c.dom.*` e `javax.xml.*` abbiamo tutti gli strumenti per fare riferimento ad un oggetto DOM e creare dinamicamente oggetti *HTML* al suo interno.

Istanziamo un oggetto Document: l'oggetto Document fornisce un modo generico di rappresentare documenti *HTML* ed è l'elemento che contiene tutti gli altri elementi del DOM.

```
1. Document document = docBuilder.newDocument();
```

Quindi creiamo gli elementi con il nome tag specificato secondo la seguente sintassi

```
1. Element elemento = document.createElement(nomeTag);
```

dove:

- *elemento* è l'oggetto Element che verrà creato;
- *nomeTag* è una stringa che specifica il tipo di elemento che verrà creato.

e appendiamo l'elemento all'albero di DOM.

Ad esempio:

```
1. Element rootElement = document.createElement("HTML");  
2. document.appendChild(rootElement);  
3. Element head = doc.createElement("HEAD");  
4. rootElement.appendChild(head);
```

Possiamo anche definire per un elemento degli attributi secondo la seguente sintassi:

```
1. element.setAttribute(s, v)
```

il cui significato è crea l'attributo “*s*” e imposta il suo valore a “*v*” o aggiorna il valore di *s* se già esistente.

Ad esempio:

```
1. Element link = document.createElement("LINK");
2. link.setAttribute("rel", "stylesheet");
3. link.setAttribute("type", "text/css");
4. link.setAttribute("href", "css/index.css");
5. head.appendChild(link);
```

Una volta ottenuta la rappresentazione DOM del file *HTML* per scrivere in un file usiamo un Transformer identità (non compie alcuna operazione sul file).

Il seguente esempio mostra come ottenere la trasformazione:

```
1. TransformerFactory transformerFactory =
   TransformerFactory.newInstance();
2. Transformer transformer =
   transformerFactory.newTransformer();
3. transformer.setOutputProperty(OutputKeys.METHOD, "html");
4. DOMSource domSource = new DOMSource(document);
5. StreamResult outputstream = new StreamResult(new
   File(filename));
6. transformer.transform(domSource, outputstream );
```

Come si può vedere dall'esempio come sorgente abbiamo utilizzato un oggetto di tipo **DOMSource**. Siccome vogliamo scrivere in un file allora abbiamo utilizzato come oggetto di output uno **StreamResult**. Il metodo statico *newInstance()* istanzia un oggetto di tipo **TransformerFactory**, che poi istanzia un oggetto di tipo **Transformer**. L'oggetto **transformer** infine, grazie al metodo *transform()*, trasforma il contenuto del documento DOM nel file html.

Osserviamo inoltre che l'istruzione (3) garantisce che nel file *HTML* vengano scritti il testo e i tag del documento DOM.

### 3.5.2 *L'integrazione con PhoneGap e Plugin*

La logica di programmazione che sta dietro ad ogni applicazione PhoneGap è molto semplice: una pagina *HTML* ed uno script *JavaScript*. Con questi soli due elementi è possibile realizzare un'applicazione che sia platform-dipendente. L'approccio usato da Cordova è, dunque, unico nel suo genere, perché fonde le capacità native di un dispositivo con la possibilità di realizzare applicazioni usando gli standard web, dando così vita alle applicazioni ibride. Grazie quindi alla sua versatilità, PhoneGap si è rivelato lo strumento perfetto in questo lavoro.

A questo punto si è giunti all'ultimo passo, cioè l'integrazione dell'intero progetto in PhoneGap.

Di seguito si riporta il workflow per eseguire questa operazione:

- settare i riferimenti al plugin nel file *config.xml*;
- costruire la classe Java che costituisce il plugin;
- costruire l'interfaccia JavaScript che useremo per richiamare il plugin.

Nel capitolo 2.4.5 abbiamo già discusso della struttura di un progetto PhoneGap e di come sono organizzati i file e le cartelle che lo costituiscono. All'interno del progetto Android, nella cartella **res/xml**, bisogna settare i riferimenti al nostro plugin all'interno del file **config.xml**.

Questo file xml contiene un tag **plugins**, all'interno del quale sono presenti un insieme di tag **plugin** che rappresentano tutti i vari elementi del framework PhoneGap e di terze parti che sono attualmente attivi nell'applicazione. In sostanza bisogna aggiungere un nuovo tag **plugin** come di seguito:

```
1. <plugin name="AWTRun" value="com.awtrun.AWTPlugin" />
```

Questo tag contiene due attributi: “*name*” che rappresenta il nome attraverso cui verrà referenziato il nostro plugin all'interno dell'engine JavaScript e “*value*” che invece costituisce il percorso completo della classe Java che realizza il componente nativo voluto. Chi già conosce il linguaggio Java sa che i nomi dei file che rappresentano le classi terminano con l'estensione *.java*, in questo caso non è necessaria.

La classe Java che costituisce il plugin dovrà estendere **CordovaPlugin**:

```
1. public class AWTPlugin extends CordovaPlugin
```

Questa classe conterrà un riferimento all'oggetto della classe Java nativa da miniaturizzare ed un riferimento all'oggetto mainframe.

```
1. public class AWTPlugin extends CordovaPlugin {  
2.     private Object exp;  
3.     private pFrame component;  
4.  
5.     public AWTPlugin() {  
6.         exp = new Example();  
7.         component = (pFrame) ((Example) exp).getTopComponent();  
8.  
9.         public boolean execute(String action, JSONArray  
10.            args,CallbackContext callbackContext){  
11.             ....  
12.         }  
13.     }
```

A questo punto si è giunti all'ultimo dei 3 punti elencati, cioè la costruzione dell'interfaccia JavaScript che permetterà di richiamare il codice Java appena creato.

Nella cartella **assets/www/js** sarà presente un (o più) file *.js* dove sono definite le funzioni *JavaScript* di supporto. Un tag *HTML* farà riferimento a questo file nel codice *HTML* che costituisce l'interfaccia, come di seguito:

```
1. <SCRIPT type="text/javascript" src="js/index.js" ></SCRIPT>
```

### 3.5.3 La creazione dell'Activity nel progetto PhoneGap

La creazione di un'activity in un progetto PhoneGap necessita di due passaggi fondamentali:

- estendere la classe Activity, appartenente al framework Android;
- registrare l'Activity nell'*AndroidManifest.xml* mediante l'uso dell'apposito tag XML *<activity>*.

Creiamo, quindi, una classe che estende **Activity** ed implementa l'interfaccia **CordovaInterface**: al suo interno implementiamo l'override del metodo *onCreate()*, invocando il metodo omonimo della classe base. Questa operazione è assolutamente obbligatoria.

```
1. public class AWTRun extends Activity implements  
CordovaInterface{  
  
2. CordovaWebView cwv;  
  
3. @Override  
4. public void onCreate(Bundle savedInstanceState) {  
5.     super.onCreate(savedInstanceState);  
6.     this.setContentView(R.layout.main);  
7.     Log.d("awtrun","AWTRun Started");  
8.     getWindow().addFlags(  
9.         WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);  
10.    cwv = (CordovaWebView) findViewById(R.id.tutorialView);  
11.    cwv.loadUrl("file:///android asset/www/index.html");  
12. }  
13. ...  
14. }
```

Alla riga (6) specifichiamo quale sarà il volto dell'activity, cioè il suo layout. Alla riga (8-9) con il metodo *getWindow().addFlags* manteniamo attivo il display del dispositivo durante l'esecuzione dell'applicazione, utilizziamo il flag *FLAG\_KEEP\_SCREEN\_ON*.

Per la visualizzazione sul display utilizziamo una **CordovaWebView**, un tipo di *View* che permette di visualizzare pagine web, nella quale viene gestita la logica applicativa (10-11).

Il tag <ACTIVITY> contenuto nel file **AndroidManifest.xml** che configura la nostra applicazione appare così:

```
1. <activity
2.     android:name="com.awtrun.AWTRun"
3.     android:label="@string/app_name" >
4.     <intent-filter>
5.         <action android:name="android.intent.action.MAIN" />
6.         <category
7.             android:name="android.intent.category.LAUNCHER" />
8.     </intent-filter>
9. </activity>
```

# 4. Generazione del layout

## 4.1 Il porting dei componenti grafici

### 4.2.1 Il pulsante

Per fare il parsing di un pulsante (**button**) bisogna innanzitutto capire come bisogna rappresentarlo in *HTML*. Un pulsante in *HTML* viene generato così:

```
1. <BUTTON class="button" id="sum" onClick="app.fire(id)" type="button">submit</BUTTON>
```

Il risultato sarà un pulsante che graficamente sarà così:

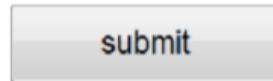


Figura 4. 1 - Esempio di un bottone in Android

Per la generazione del codice *HTML* in modo automatico, è stato aggiunto un metodo alla classe *parseHTML*.

```
1. public static void addButton(String name){  
2.     Element button = doc.createElement("BUTTON");  
3.     button.setAttribute("type", "button");  
4.     button.setAttribute("class", "button");  
5.     button.setAttribute("id", name);  
6.     button.appendChild(doc.createTextNode(name));  
7.     button.setAttribute("onClick", buttonEvent);  
8.     elements.put(name, button);  
9. }
```

Il metodo *addButton()* prende in input il nome del pulsante che sarà anche il suo identificativo per recuperare il riferimento al pulsante. Alla riga (2) creiamo il TAG *<Button>*, successivamente aggiungiamo gli attributi *type*, *class*, *id* e *onClick* utilizzando il metodo *setAttribute()*.

### 4.1.2 La label

Un altro componente *AWT* da dover considerare riguarda la **label**. Le labels sono delle strutture di interfaccia usate per mostrare del testo statico e non dinamico (ad esempio nelle dialog in cui viene chiesta la login), i campi di testo (*TextField*, dinamici ed interattivi con l'utente) possono essere nominati o identificati con una classica label. Per creare una label si utilizza un

sottoinsieme della **pTextField** rendendola *read-only* e quindi non modificabile dall'utente. Una piccola modifica alla pTextField con una variabile comportamentale che la rende solo di lettura e senza margini di scrittura (concettualmente molto più simile ad una label AWT) renderebbe il tutto perfettamente aderente con la Label delle AWT.

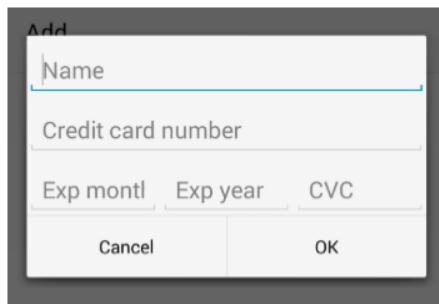


Figura 4. 2 - Esempio di labels in Android

#### 4.1.3 L'area di testo

Per l'area di testo (**TextField**), utilizzata ad esempio per l'input del programma, il codice *HTML* utilizzato è:

```

1. <DIV class="areaditestotext">
2. <LABEL><INPUT id="a" value="" type="text" onkeyup="app.insert(id,document.getElementById(id).value)" placeholder="a" /></LABEL>
3. </DIV>
```

Il risultato grafico del codice sopra è:



Figura 4. 3- Esempio di area di testo in Android

Il codice Java utilizzato per generare il codice *HTML* è:

```

1. public static void addTextField(int size, String title){
2. Element list = doc.createElement("DIV");
3. list.setAttribute("class", "areaditestotext");
4. Element label = doc.createElement("LABEL");
5. Element textField = doc.createElement("INPUT");
6. textField.setAttribute("type", "text");
7. textField.setAttribute("placeholder", title);
8. textField.setAttribute("id", title);
9. textField.setAttribute("onkeyup", textEvent);
10. label.appendChild(textField);
11. list.appendChild(label);
12. elements.put(title, list);
13. }
```

#### 4.1.4 La checkbox (radio button)

La gestione della **checkbox** (o **radio button**) è comunque argomento di studio se si vuole realizzare un'interfaccia *HTML* completa e soddisfacente. Le *AWT* hanno le *java.awt.Checkbox* che permettono all'utente di effettuare la scelta tra un insieme di opzioni preconfezionate dal programmatore. Sotto *HTML* esiste un equivalente semplice e di facile implementazione:

```
1. <input type="checkbox">
```

Per quanto riguarda le *pAWT*, la classe **pCheckbox** di facile implementazione che estende le *Checkbox* delle *AWT* ed implementa il traduttore durante la fase di parsing *HTML* all'interno di un form.

Nessun *JS* è richiesto per la gestione delle checkbox dato che nessuna funzione è associata a questo componente, essendo predisposto al data entry e non alla gestione eventi.



Figura 4. 4 - Esempio di checkbox in Android

#### 4.1.5 Il menù

Se ad una applicazione vogliamo aggiungere un **menù** dobbiamo seguire questi tre punti:

- Creare gli oggetti **MenuItem**;
- Creare gli oggetti **Menu** ed attaccarvi i **MenuItem**;
- Creare una **MenuBar** e attaccare i **Menu**;

*l'HTML* utilizzato per il menu a tal proposito offre la possibilità di creare un elenco utilizzando i tag *<ul>* e *<li>*.

In generale possiamo inserire diversi livelli con `<li>` all'interno del tag `<ul>`, creando delle strutture "ad albero", utili a definire oggetti come menu. Per farlo è sufficiente inserire un nuovo elenco all'interno di un elemento.

```

1. <ul>
2.   <li>File
3.     <ul>
4.       <li id="Apri" onclick="app.fire(id)">Apri</li>
5.       <li id="Chiudi" onclick="app.fire(id)">Chiudi</li>
6.     </ul>
7.   </li>
8.   <li>Modifica
9.     <ul>
10.      <li id="Annulla"
11.        onclick="app.fire(id)">Annulla</li>
12.      <li id="Ripristina"
13.        onclick="app.fire(id)">Ripristina</li>
14.    </ul>
15.  </li>
16. </ul>

```

Per generare l'HTML del menu, si utilizzano due metodi ricorsivi. Il primo metodo `void parse_menu(ArrayList<pMenu> bar)`; prende in input la barra del menu, quindi un'arraylist contenente tutti i **pMenu** utilizzati.

```

1. public static void parse_menu(ArrayList<pMenu> bar) {
2.   int i;
3.   Element ul = doc.createElement("ul");
4.   span_menu.appendChild(ul);
5.   for(i=0;i<bar.size();i++) {
6.     Element li = doc.createElement("li");
7.     ul.appendChild(li);
8.     li.appendChild(doc.createTextNode(bar.get(i).getName()));
9.     li.setAttribute("onclick", "insert_menu(this)");
10.    if(bar.get(i).has_item())
11.      parse_menu_item(li, bar.get(i).getArray_menu());
12.  }
13. }

```

Alla riga (3), si crea il primo tag `<ul>` principale, poi alla riga (5), viene utilizzato il ciclo for per scorrere tutto l'arraylist. Alla riga (10), dopo aver creato il tag `<li>`, controllo con il metodo `boolean has_item()` se il **pMenu** contiene un sottomenu. Se li contiene chiamo il secondo metodo ricorsivo che andrà a generare l'html di tutti i **sottoMenu** del **pMenu**.

```

1. private static void parse_menu_item(Element tag_li,
   ArrayList<pMenuItem> pmenui){
2.     int i;
3.     Element ul = doc.createElement("ul");
4.     tag_li.appendChild(ul);
5.     for(i=0;i<pmenui.size();i++){
6.         Element li = doc.createElement("li");
7.         ul.appendChild(li);
8.         li.appendChild(doc.createTextNode(
9.             pmenui.get(i).getName()));
10.        li.setAttribute("id", pmenui.get(i).getName());
11.        li.setAttribute("onclick", "app.fire(id)");
12.        if(pmenui.get(i).has_item())
13.            parse_menu_item(li, pmenui.get(i).getArray_menu());
14.    }
15. }

```

Notiamo che questo metodo è molto simile al primo metodo, con la differenza che in input abbiamo bisogno anche del Tag `<li>` di riferimento dove poter creare un sottomenu partendo di nuovo dal tag `<ul>`; e come secondo input, l'ArrayList prende in input solo **pMenuItem**.

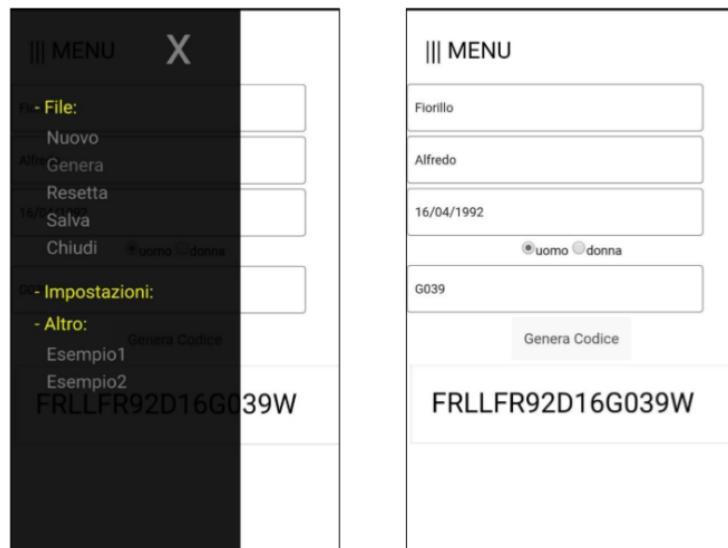


Figura 4. 5 - Esempio di menu implementato in Android

## 4.2 L'impaginazione del layout

Il processo della generazione del layout, se affrontato nella sua interezza può risultare piuttosto complesso: si vuole generare il codice sorgente dell'applicazione mobile dove spesso gli elementi che compongono l'interfaccia grafica devono essere riorganizzati (eventualmente in più Frame) per migliorare l'usabilità da mobile.

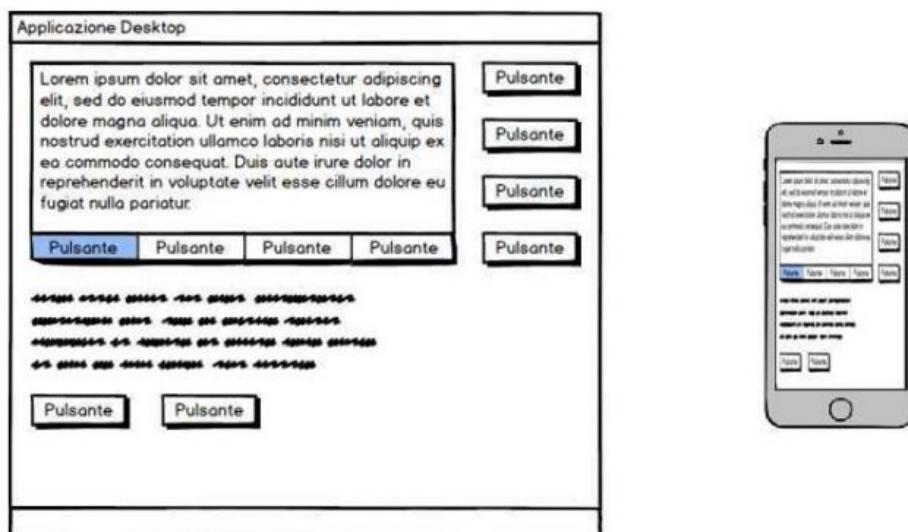


Figura 4.6 - Confronto tra spazio a disposizione su desktop e mobile

Nella figura 4.6 viene evidenziato come un'interfaccia grafica pensata per essere visualizzata su desktop risente dello spazio ridotto quando viene trasformata in applicazione mobile, senza considerare che non vi è alcuna garanzia che il layout generato venga rimpicciolito in proporzione a quello originale.

Semplifichiamo il processo di generazione del layout focalizzandoci su un problema per volta, suddividendolo in fasi:

- **Grafo delle interazioni degli elementi**

Il codice sorgente dell'applicazione di partenza viene analizzato per ricercare tutti gli elementi che compongono l'interfaccia grafica. Al termine di questa fase si ottiene un grafo  $G=(V,E)$  dove:  $V$  è l'insieme dei vertici: i nomi dei riferimenti degli oggetti.  $E$  è l'insieme degli archi: dati due vertici  $i, j \in V$  si ha una coppia  $(i, j) \in E \iff i \text{ interagisce con } j$ .

- **Partizionamento dei vertici del grafo**

In questa seconda fase, l'insieme dei vertici  $V$  del grafo delle interazioni viene partizionato sfruttando sia l'insieme degli archi  $E$ , sia eventuali vincoli aggiuntivi imposti. Ogni blocco di questa partizione determinerà un Frame dell'applicazione mobile.

- **Generazione dell'interfaccia mobile**

In quest'ultima fase il codice sorgente dell'applicazione viene nuovamente analizzato per trasformare ogni Component nel corrispettivo pComponent e generare il codice dell'applicazione mobile tenendo presente la suddivisione in più frame fatta nella fase precedente.

Al termine di queste fasi, l'applicazione è pronta per essere compilata ed eseguita su mobile.

Al termine della generazione dell'interfaccia mobile, il parser dovrà creare una pagina **index.html** contenente tutti i Frame che la compongono. Un Frame è rappresentato da un *DIV* (elemento *HTML*) che, grazie alla classe *CSS "frame"*, occupa tutto lo spazio a disposizione nello schermo del dispositivo.

In questo modo, pur essendo contenuti più "frame" nella stessa pagina, l'utente che naviga tra questi *DIV* avrà la sensazione di spostarsi da più Activity. Inoltre, assegnando un id univoco al div che rappresenta il frame, si potrà fare uso degli (anchor) per reindirizzare l'utente in una schermata specifica.

Per aggiungere un nuovo frame, si utilizza il codice *HTML* seguente, dove è possibile notare che a ciascun frame viene assegnato un id univoco.

```
1. <DIV class="frame" id="frame0">
2.   <!-- Elementi contenuti nel Frame -->
3. </div> ;
```

Per quanto riguarda la suddivisione in più Frame dell'applicazione, l'icona del Menu dovrà comparire in tutti i Frame. Quindi, subito dopo il tag che rappresenta il Frame, è stato aggiunta l'icona del Menu che richiama la funzione *openNav()* una volta cliccato.

```
1. <DIV class="frame" id="frame0">
2. <div class="scritta_menu" onclick="openNav()"> MENU</div>
```

La funzione *openNav()* non fa altro che apparire il blocco di Menu identificato dall'id "myNav". Viceversa *closeNav()* farà scomparire il menu impostando la larghezza del tag a 0.

```
1. function openNav() {
2.   document.getElementById("myNav").style.width = "70%";
3. }
4. function closeNav() {
5.   document.getElementById("myNav").style.width = "0%";
6. }
```

### 4.2.1 Un esempio: Rubrica

Vediamo come esempio di analizzarsi l'interfaccia grafica di un form per l'inserimento di un contatto nella rubrica.



Figura 4. 7 - L'interfaccia grafica Rubrica in Java

Poiché i campi sono raggruppati per categorie, il mainFrame dovrà contenere tre oggetti di tipo Panel, i quali verranno aggiunti ad esso, ovvero:

1. *panelAnagrafica*, dove saranno aggiunti oggetti di tipo TextField per l'inserimento di dati anagrafici: nome, cognome, residenza e codiceFiscale.
2. *panelContatti*, dove saranno aggiunti oggetti di tipo TextField per l'inserimento dei vari recapiti: skype, twitter, numero, email, sitoWeb, facebook.
3. *panelPulsanti*, dove saranno aggiunti oggetti di tipo Button per la funzione di salvataggio di un contatto nella rubrica (pulsante salva) e per il reset dei campi (pulsante reset).

Analizziamo le fasi necessarie per il processo di generazione del layout:

#### 1. Generazione del grafo

Effettuando il parsing sul file *Rubrica.java* vengono aggiunte tutte le componenti identificandole come vertici del grafo e tutti i metodi che interagiscono fra loro come archi del grafo.

```
VERTICI={  
    mainFrame, panelPulsanti, panelAnagrafica, panelContatti,  
    salva, reset, residenza, codiceFiscale,  
    nome, cognome, skype, twitter, numero, email, sitoWeb,  
    facebook  
}
```

```
ARCHI = {
    (panelContatti, skype), (panelContatti, twitter),
    (panelContatti, numero), (panelContatti, email),
    (panelContatti, sitoWeb), (panelContatti, facebook),
    (panelPulsanti, salva), (panelPulsanti, reset),
    (panelAnagrafica, nome), (panelAnagrafica, cognome),
    (panelAnagrafica, residenza),
    (panelAnagrafica, codiceFiscale),
    (mainFrame, panelAnagrafica), (mainFrame, panelContatti),
    (mainFrame, panelPulsanti), (reset, panelContatti),
    (reset, panelAnagrafica)
}
```

## 2. Partizionamento del grafo:

Al termine della Fase 1 abbiamo ottenuto il grafo delle interazioni.

Osservando l'immagine in Figura 4.7, è facile rendersi conto che trasformare l'intera interfaccia grafica in una singola schermata mobile significherebbe creare una lunga lista di campi di testo che l'utilizzatore dell'applicazione sarebbe costretto a scorrere prima di arrivare al pannello dei pulsanti.

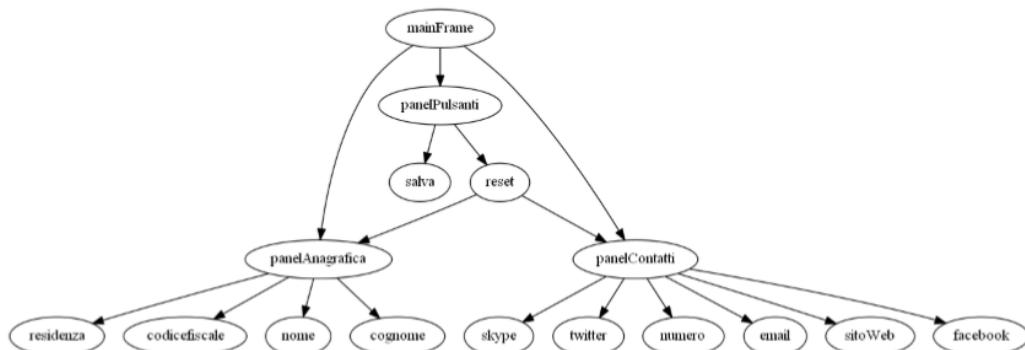


Figura 4.8 - Rappresentazione grafica del grafo prima della partizione

L'obiettivo di questa fase è partizionare l'interfaccia grafica il più possibile, senza perdere di vista l'unico vincolo: i due pulsanti reset e salva devono essere contenuti nello stesso blocco. Tale vincolo viene creato con opportune classi e a questo punto il processo di partizionamento viene avviato ottenendo:

```
BLOCCO 1 = {
    panelAnagrafica, residenza, cognome, panelPulsanti,
    reset, nome, salva, mainFrame, codiceFiscale
}
BLOCCO 2 = {
    skype, twitter, numero, facebook,
    panelContatti, sitoWeb, email
}
```

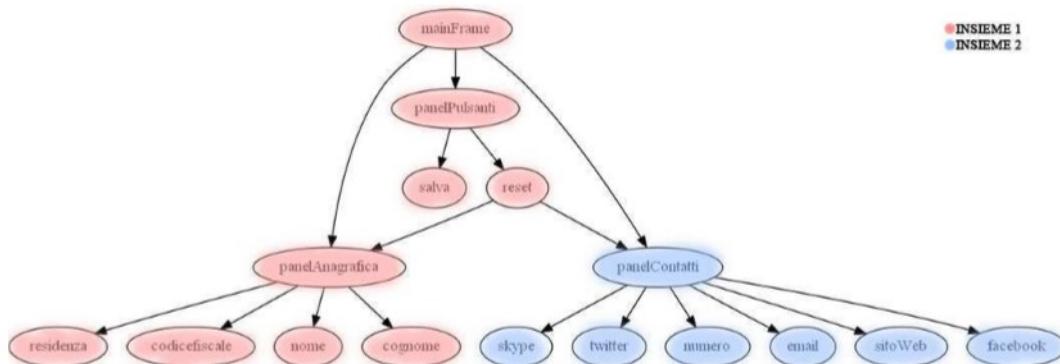


Figura 4. 9 - Rappresentazione grafica del grafo partizionato

### 3. Generazione del codice sorgente

In accordo alle fasi precedenti viene avviato il processo di parsing che crea il file *index.html*. Ecco come si presenta l’interfaccia dell’applicazione mobile:

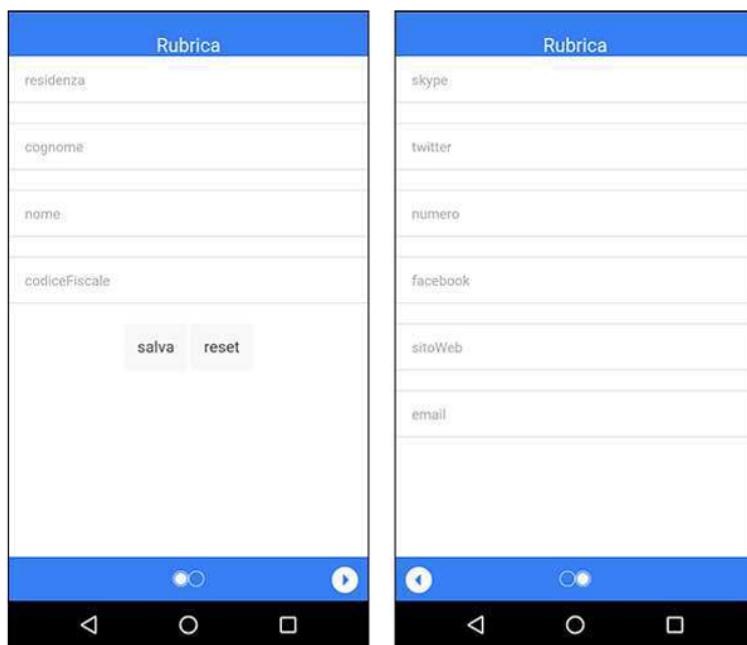


Figura 4. 10 - L'interfaccia grafica mobile Rubrica

Come si può notare per permettere all’utente di navigare tra le schermate, è stata inserita la barra di navigazione in basso alla finestra dell’applicazione strutturata in questo modo:

- Ai lati della barra compaiono delle frecce nel caso in cui vi è la presenza di schermate successive/precedenti.
- Al centro sono presenti dei bullets che simboleggiano il numero di schermate.

# 5. La gestione della tabella

Nel seguente capitolo verrà analizzata come è stata strutturata la creazione della componente tabella nella libreria **pAWT**, mostrandone la generazione durante la fase di **parsing**, la gestione degli eventi ad essa associati e come avviene la comunicazione bidirezionale tra **PhoneGap** e codice nativo **Java**.

Rispetto alle componenti presenti nelle *pAWT*, la componente **pTable** ha richiesto una maggiore comprensione sul funzionamento in quanto la classe Java corrispondente, **JTable**, opera con numerose classi di supporto il che hanno reso molto onerosa l'implementazione e l'apprendimento di tale componente.

Un'altra differenza, rispetto alle componenti precedenti, è che si cercato di ottenere un'alta compatibilità tra *JTable* e *pTable* implementato e riadattando numerosi metodi.

Tutto ciò verrà spiegato nel dettaglio nel corso del capitolo.

## 5.1 La tabella in Java

La tabella in Java viene rappresentata dalla classe **JTable**, contenuta nel package *javax.swing*, attraverso cui è possibile visualizzare un modello di dati, eventualmente permettendo all'utente di modificarlo.

Ecco come viene mostrata all'utente una semplice view dei dati:



Figura 5. 1 - L'interfaccia grafica di una tabella in Java

La tabella raffigurata precedentemente , viene costruita semplicemente dichiarando i nomi delle colonne in un *array di stringhe*:

```
1. String[] columnNames = {
2.     "First Name",
3.     "Last Name",
4.     "Sport",
5.     "# of Years",
6.     "Vegetarian"
7. };
```

e i dati sono inizializzati e memorizzati in un *array di oggetti bidimensionale*:

```
1. Object[][] data = {
2.     {"Kathy", "Smith",
3.      "Snowboarding", new Integer(5), new Boolean(false)},
4.     {"John", "Doe",
5.      "Rowing", new Integer(3), new Boolean(true)},
6.     {"Sue", "Black",
7.      "Knitting", new Integer(2), new Boolean(false)},
8.     {"Jane", "White",
9.      "Speed reading", new Integer(20), new Boolean(true)},
10.    {"Joe", "Brown",
11.      "Pool", new Integer(10), new Boolean(false)}
12. };
```

La tabella in fine viene costruita utilizzando i due array precedenti nel seguente modo:

```
JTable table = new JTable(data, columnNames);
```

Il vantaggio di questo costruttore è che è molto facile da usare. Tuttavia ha anche svantaggi: ogni cella è modificabile, tratta tutti lo stesso tipo di dati (come stringhe); richiedono i dati della tabella in un array o un vettore ecc...

Nella costruzione di una tabella la classe **JTable**, per superare anche queste restrizioni, può utilizzare delle classi di supporto contenute nel pacchetto apposito *javax.swing.table* ognuna avente diverse responsabilità: gestione dei dati, delle colonne e selezione dei dati.

Per comprendere meglio come tali classi operano tra di loro verrà mostrato nella figura 5.2 le zone d'interesse su cui lavorano tramite una semplice vista di una tabella:

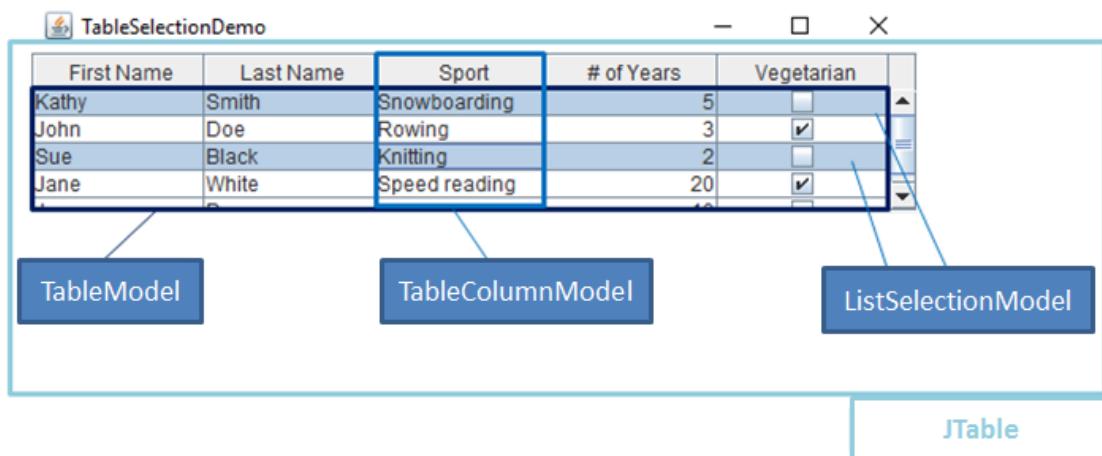


Figura 5.2 - Illustrazione delle classi che utilizza JTable

Come si può notare **JTable** collabora con tre classi, le quali vengono istanziate nei suoi costruttori:

- **TableModel:**

è un interfaccia che specifica i metodi che JTable utilizzerà per interrogare un modello di dati tabulari. Le implementazioni che utilizzano tale interfaccia sono `AbstractTableModel`, classe astratta che fornisce un' implementazione predefinita di alcuni metodi, e la sua sottoclasse `DefaultTableModel`, che utilizza un vettore di vettori per memorizzare il valore degli oggetti delle celle.

- **TableColumnModel:**

è un'interfaccia che definisce i requisiti per un oggetto “modello di colonna” adatto all’uso di JTable. Tale interfaccia viene implementata in maniera standard per un JTable nella classe `DefaultColumnModel`.

- **ListSelectionModel:**

questa interfaccia rappresenta lo stato corrente della selezione, la cui viene modellata come un insieme di intervalli indicizzati.

`DefaultListSelectionModel` è la classe di default che implementa tale interfaccia.

## 5.2 La tabella in pAWT

Come detto, la corrispondente classe di *JTable* definita in **pAWT** è **pTable**. Sono state riportate anche tutte le classi contenute nel pacchetto *javax.swing.table* andando a definire nella libreria **pAWT** un sotto-package *pwat.table*.

La classe **pTable**, come il resto delle componenti, estende la classe astratta **pComponent**, ridefinendone opportunamente i metodi, ed implementa l’interfaccia **TableModelListener**, utile a gestire eventi scatenati sul modello di dati come inserimento e rimozione di una riga che verranno spiegati in seguito.

In sintesi seguiranno i costruttori e i metodi implementati da **pTable** che rispecchiano la firma e la logica della classe *JTable*, integrando il tutto alle componenti della libreria e al plugin.

### 5.2.1 I costruttori di *pTable*

La classe *pTable* supporta numerosi costruttori i quali richiamano il costruttore principale. Quest’ultimo impostai i riferimenti interni agli oggetti *TableModel*,  *TableColumnModel* e *ListSelectionModel* (se i parametri sono nulli li istanzia di default utilizzando appositi metodi di supporto contenuti all’interno della classe). Per semplicità riportiamo soltanto il codice del costruttore principale:

```

1. public pTable(TableModel dm, TableColumnModel cm,
2.                 ListSelectionModel sm) {
3.     _title = "Tabella"+this.NUMBER;
4.     NUMBER++;
5.     if(dm == null)
6.         _dataModel = createDefaultDataModel();
7.     else _dataModel = dm;
8.     if (cm == null) {
9.         _columnModel = createDefaultColumnModel();
10.        createDefaultColumnsFromModel();
11.    }
12.    else _columnModel = cm;
13.    if (sm == null)
14.        _selectionModel = createDefaultSelectionModel();
15.    else _selectionModel = sm;
16.
17.    initializeLocalVars();
18.}
```

Così come ogni componente della nostra libreria anche **pTable** ha un id univoco (3), “*\_title*”, costruito attraverso la concatenazione della stringa “*Tabella*” e una variabile statica, impostata a 0 di default, che viene incrementata ad ogni nuova istanza di **pTable**.

Dopo aver settato i vari riferimenti, il costruttore principale richiama un metodo privato, *initializeLocalVars()*, utile a settare alcune proprietà iniziali della tabella come: colore di sfondo, primo piano, margini e altezza righe ecc...

```
1. protected void initializeLocalVars() {  
2.     this.showHorizontalLines = true;  
3.     this.showVerticalLines = true;  
4.     this.isRowHeightSet = false;  
5.     this.rowSelectionAllowed = true;  
6.     setColumnSelectionAllowed(false);  
7.     setCellSelectionEnabled(true);  
8.     this.rowMargin = 20; //JTable default 1  
9.     this.rowHeight = 50; //JTable default 16  
10.    this.editingColumn = -1;  
11.    this.editingRow = -1;  
12.    this.gridColor = "BLACK";  
13.    this.background = "GAINSBORO";  
14.    this.foreground = "BLACK";  
15.    this.selectionBackground = "SKYBLUE";  
16.    this.selectionForeground = "WHITE";  
17.}
```

### 5.2.2 I metodi di pTable

Analizziamo adesso i metodi contenuti nella classe **pTable** (Figura 5.3). Possiamo osservare che mantengono la stessa firma di quelli di *JTable* e conservano la stessa logica permettendo inoltre la comunicazione con il plugin.

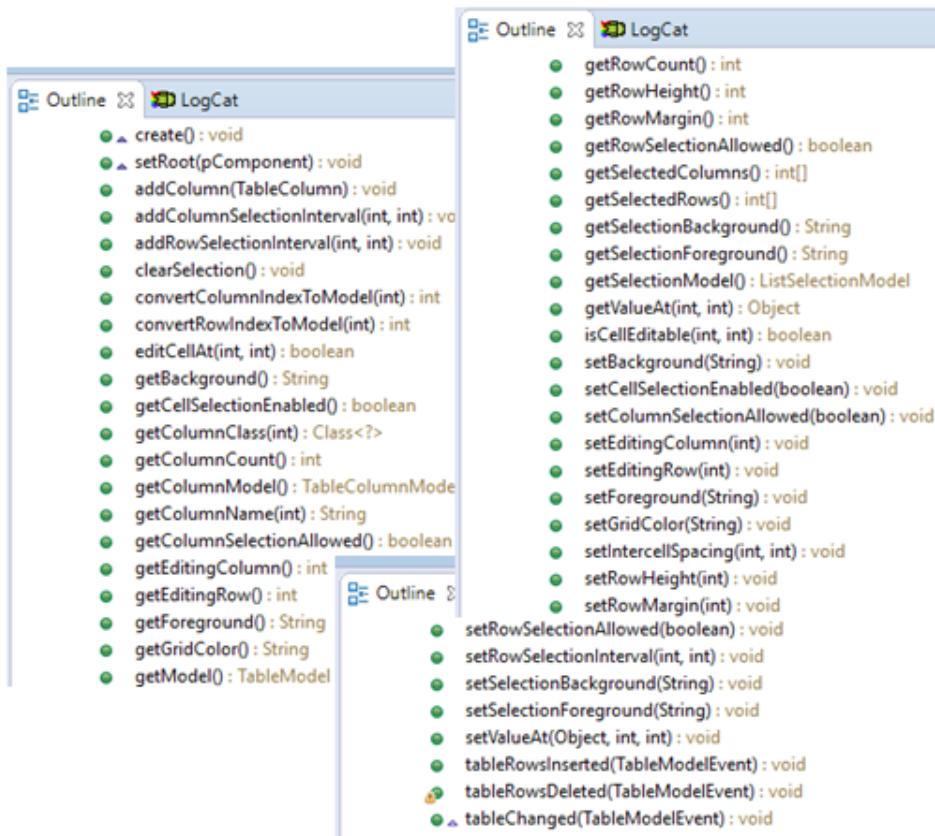


Figura 5.3 - Metodi implementati in pTable

Come si può vedere dall'immagine sono stati implementati molti dei metodi della classe *JTable*:

- alcuni sono semplice metodi come *getter* e *setter*;
- differenti metodi invece sono privati per impostare proprietà interne come *initialLocalVars()*, *createDefaultTableModel()*, ecc...
- altri invece devono prestare particolari attenzioni in quanto vanno a delineare il layout della tabella e quindi vengono implementati in modo tale da permettere la comunicazione bidirezionale tra Java e Phonegap
- altri ancora, come il metodo *tableChanged()*, definito dall'interfaccia *TableModelListener*, è necessario per ricevere l'evento scatenato dal modello di dati in seguito all'inserimento o rimozione di una riga, per esempio, per poi richiamare di conseguenza *tableRowsInserted()* oppure *tableRowsDeleted()*.

### 5.3 Introduzione al parsing della tabella

Come spiegato nel capitolo 3.5 la fase di **parsing** consiste nel generare il documento *HTML* comprendente i riferimenti ai file *JavaScript* e *CSS* che bisogna utilizzare.

Per capire come si presenta la struttura di base di una tabella in *HTML* vediamo un esempio:

```
1. <table border="1">
2.   <tr>
3.     <th>Firstname</th>
4.     <th>Lastname</th>
5.     <th>Age</th>
6.   </tr>
7.   <tr>
8.     <td>Jill</td>
9.     <td>Smith</td>
10.    <td>50</td>
11.   </tr>
12.   <tr>
13.     <td>Eve</td>
14.     <td>Jackson</td>
15.     <td>94</td>
16.   </tr>
17. </table>
```

Dal codice si può notare il tag `<table>` (1-17) che definisce una tabella; ogni riga viene determinata con il tag `<tr>`(2-6 , 7-11, 12-16); mentre l'intestazione della tabella viene rappresentata dal tag `<th>` (3, 4, 5). Per impostazione predefinita, le intestazioni sono definite in grassetto e centrate. Infine un dato/cella viene delineato con il tag `<td>` (8, 9, 10, 13, 14, 15).

#### 5.3.1. *Come identificare la tabella nel codice*

Come abbiamo visto nel capitolo 3.5.1 ad un **elemento-tag** possiamo definire degli attributi tramite il metodo `setAttribute()`.

Questi risultano utili per poter ottenere dei riferimenti agli elementi-tag all'interno di uno script *JS* oppure per definire delle regole *CSS* (selettore).

Occorre , per cui, specificare per ognuno degli elementi-tag della tabella, siano essi `<th>`,`<tr>` o `<td>`, delle proprietà **id** e **class** utilizzando un determinato formato di creazione in quanto la fase di parsing avviene in maniera dinamica.

Qui di seguito verrà fornita una tabella riassuntiva del formato degli attributi:

TAG	ATTRIBUTO	FORMATO	ESEMPIO
<TABLE>	<i>ID</i>	idTabella	Tabella0
	<i>CLASS</i>	Tab	Tab
<TR> Header	<i>ID</i>	idTabella+-rowHeader	Tabella0- rowHeader
	<i>CLASS</i>		
<TH>	<i>ID</i>	idTabella+- colHeader+indiceColonna	Tabella0- colHeader2
	<i>CLASS</i>	idTabella+-tabHeader"	Tabella0- tabHeader
<TR> Default	<i>ID</i>	idTabella+-row"+indiceRiga	Tabella0- row1
	<i>CLASS</i>	idTabella+-row"	Tabella0- row
<TD> Default	<i>ID</i>	idTabella+- col"+indiceRiga+", "+indiceColonna	Tabella0- col1,2
	<i>CLASS</i>	idTabella+-column"	Tabella0- column

Figura 5. 4 – Formato degli attributi ID e CLASS degli elementi-tag della tabella

### 5.3.2 La gestione degli eventi della tabella in HTML

La registrazione di un evento per un elemento HTML avviene in due modi:

- **inline:** l'evento viene registrato all'interno del tag come un attributo il cui valore potrà essere una chiamata ad una funzione oppure qualche semplice riga *JavaScript*;

```

1. <BUTTON class="button" id="sum"
2. onClick="funzione()" type="button">sum</BUTTON>

```

- **addEventListener()**: tale metodo viene richiamato lato JS dall'elemento-tag su cui bisogna registrare un evento, definendone il tipo, l'operazione da eseguire (chiamata a funzione/funzione anonima), come effettuare la delegazione degli eventi (capturing/bubbling).

```

1. document.getElementById("sum").addEventListener(
2.     "click", funzione, false
3. );

```

Per registrare gli eventi nel caso della tabella si è utilizzata la seconda tecnica. Oltre a ciò è stata utilizzata la libreria **Hammer.js** descritta già nel capitolo 2.7 per definire alcuni eventi.

Lo script “**tableEvent.js**” si occupa di registrare gli eventi a righe e celle che vengono generate durante la fase di parsing della tabella utilizzando delle funzioni, ovvero:

- **addFireRowListener()**

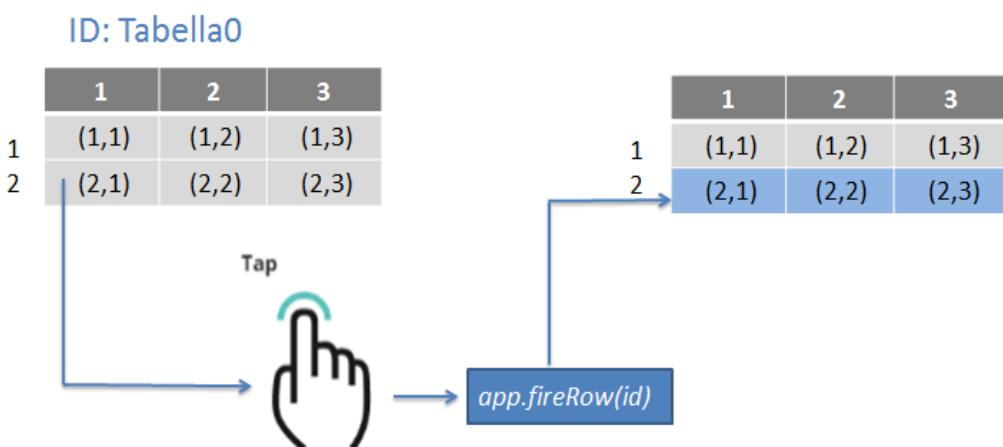


Figura 5. 5 - Illustrazione della gestione della selezione di una riga

si occupa di creare un oggetto *hammer* sull'elemento-tag *<tr>* passato come parametro in modo tale che sul “**tap**” dell'utente venga richiamata la funzione JavaScript “*app.fireRow(id)*” che sarà responsabile della selezione/de-selezione della riga cliccata dall'utente.

```

1. function addFireRowListener(obj, id) {
2.     mapTapHammer[id] = new Hammer(obj);
3.     mapTapHammer[id].on('tap', function(ev) {
4.         app.fireRow(id);
5.     });
6. }

```

- [addFireCellListener\(\)](#)

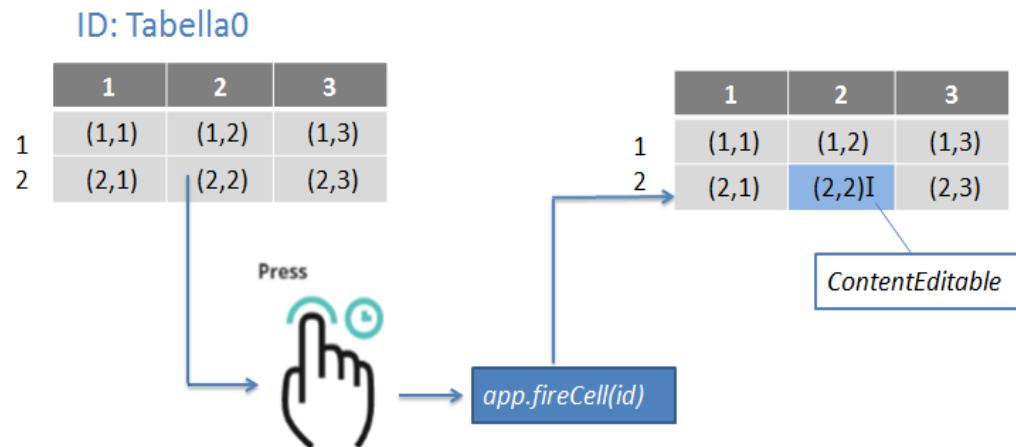


Figura 5. 6 - Illustrazione della gestione di selezione di una cella per l'editing

si occupa di creare un oggetto *hammer* sull'elemento-tag `<td>` passato come parametro in modo tale che sul "press" dell'utente venga richiamata la funzione JavaScript "`app.fireCell(id)`" che sarà responsabile della selezione della cella abilitandola all'editing da parte dell'utente.

```

1. function addFireCellListener(obj, id, value){
2.   mapPressHammer[id] = new Hammer(obj);
3.   mapPressHammer[id].on('press', function(ev) {
4.     app.fireCell(id, value);
5.   });
6. }
```

- [addChangeListener\(\)](#)

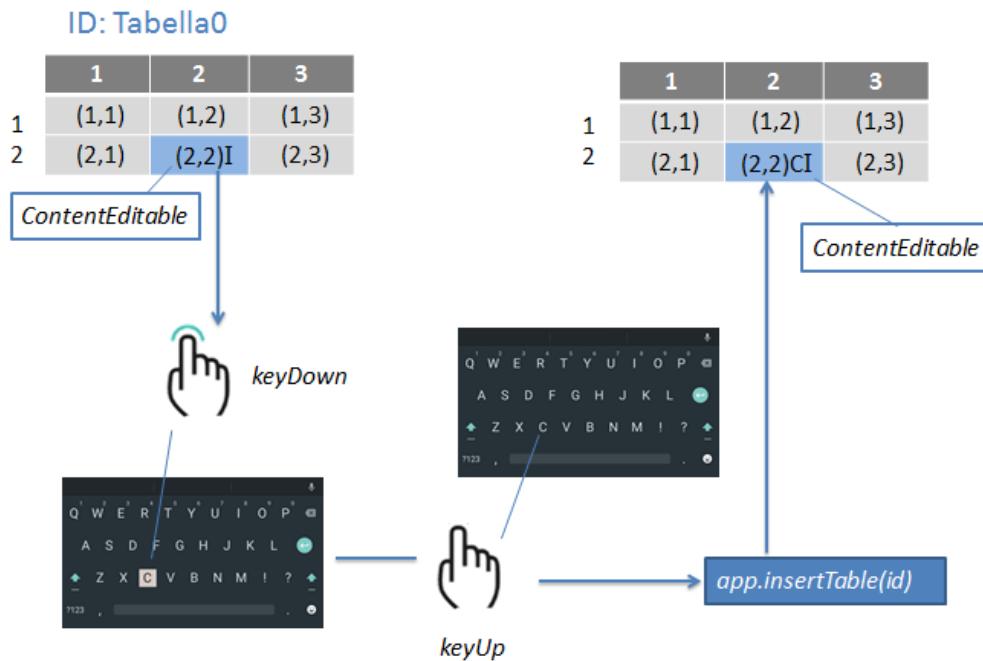


Figura 5. 7 - Illustrazione della gestione delle modifiche del valore di una cella

si occupa di registrare sull'elemento colonna passato come parametro l'evento "keyup", in modo tale che l'utente ogni qualvolta dopo aver inserito un carattere in input viene richiamata la funzione JavaScript "app.insertTable(id)" che sarà responsabile nell'impostare il valore immesso dall'utente nella cella che sta editando ed in particolare permette di aggiornare il valore della cella lato Java del modello di dati.

```

1. function addChangeListener(obj, id){
2.     var contenteditable =
3.         document.querySelector('[contenteditable]');
4.     obj.addEventListener('keyup',function(e){
5.         app.insertTable(id,
6.                         contenteditable.textContent);
7.     },false);
8. }

```

Dopo aver visto quali sono i metodi che registrano gli eventi, vediamo come essi vengono richiamati, analizzando “**tableEvent.js**”:

```

1. mapTapHammer = {};
2. mapPressHammer = {};

3. var tables = document.getElementsByTagName("table");
4. for(var i=0; i<tables.length; i++) {
5.     var tds = tables[i].getElementsByTagName("td");
6.     for(var j=0; j<tds.length; j++) {
7.         var idElm = tds[j].id;
8.         var valueElm = tds[j].innerHTML;

9.         addFireCellListener(tds[j], idElm, valueElm);
10.    }

11.    trs = tables[i].getElementsByTagName("tr");
12.    for(k=1; k<trs.length; k++) {
13.        addFireRowListener(trs[k], trs[k].id);
14.    }
15. }
```

I due hashmap (1-2) servono a contenere gli elementi su cui vengono registrati gli eventi appartenenti alla libreria **Hammer.js** in modo tale da poterli recuperare in seguito.

Si ritrovano tutti gli elementi-tag *<table>* (3) e per ognuno di essi: righe e celle (4-15). All'interno dei cicli vengono richiamati i metodi necessari a registrare gli eventi sugli elementi-tag *<tr>* (13) e *<td>*(9).

Il metodo *addChangeListener()* viene richiamato in un altro script, ovvero quando viene chiamata la funzione *app.fireCell ()* che abilità la cella all'editing su cui viene registrato l'evento *keyup* visto in precedenza.

### 5.3.3 Le proprietà grafiche della tabella

Come ribadito durante la fase di parsing vengono aggiunti i file CSS che contengono la formattazione del documento. Viene aggiunto anche il file CSS “**tableCSS.css**”, il quale viene creato/aggiornato durante la fase di parsing di una tabella, come vedremo, vengono scritte le relative regole CSS all'interno del file.

Nel definire le regole CSS degli elementi-tag dobbiamo distinguere, così come analizzato durante la gestione degli eventi, quando una riga/cella è selezionata da quando non lo è.

Un' esempio aiuterà a capire il concetto:

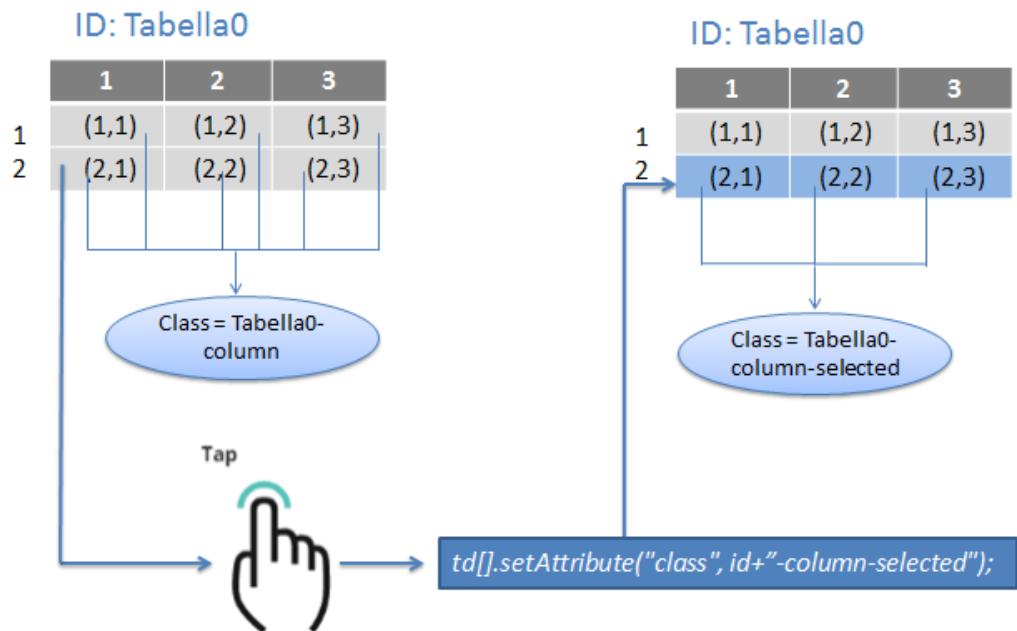


Figura 5. 8 - Illustrazione del cambiamento del set di regole CSS al momento della selezione della riga

L'immagine 5.7 mostra inizialmente una tabella 2x3. Ogni cella ha la proprietà `class` impostata a “`Tabella0-column`” in accordo al formato definito nel paragrafo 5.3.1.. In seguito ad un evento, in questo caso la selezione della riga, vengono richiamate le funzioni necessarie dove, in particolare, impostano una nuova classe “`Tabella0-column-selected`” alle celle utile dunque ad evidenziare la selezione.

Per l'esempio, il file CSS scritto durante la fase di parsing è così definito:

```

1. .Tabella0-column {background-color: grey}
2. .Tabella0-column-selected {background-color:blue}

```

Per mostrare l'avvenuta selezione bisogna modificare le regole CSS che riguardano quelle celle, per cui viene definito un set di regole appositamente per quando avviene la selezione (“`idTabella-column/row-selected`”) in modo tale da permettere un cambiamento di classe “al volo”.

Dunque per scrivere/leggere le regole bisogna utilizzare come selettori delle regole CSS i formati `ID` e `CLASS` visti precedentemente, in modo tale da poter riferirsi agli elementi-tag.

Aggiorniamo la tabella 5.4 con i formati di classe che riguardano la selezione di riga e colonna:

TAG	ATTRIBUTO	FORMATO	ESEMPIO
<i>&lt;TR&gt; Selezione</i>	<i>ID</i>	idTabella"-row"+indiceRiga	Tabella0- row1
	<i>CLASS</i>	idTabella"-row-selected"	Tabella0- row- selected
<i>&lt;TD&gt; Selezione</i>	<i>ID</i>	idTabella"- col"+indiceRiga+", "+indiceColonna	Tabella0- col0,1
	<i>CLASS</i>	idTabella"-column-selected"	Tabella0- column- selected

Figura 5. 9 - Formato degli attributi ID e CLASS degli elementi-tag selezionati della tabella

### 5.3.4 La gestione dell'impaginazione della tabella

Le dimensioni di una tabella possono essere notevoli in quanto vengono utilizzate per contenere grandi quantità di dati. Un problema che può incorrere è quello che sul device mobile nel frame della *WebView* la tabella potrebbe non essere visualizzata interamente (si pensi per esempio ad una tabella con molte colonne).

Per risolvere questo problema viene mostrata all'utente una barra di scorrimento (**Scrollbar**) ogni qualvolta le dimensioni della tabella superano lo spazio messo a disposizione del frame. Tutto questo avviene tramite l'utilizzo di specifiche regole CSS definite su un elemento **<DIV>** che fa da contenitore alla tabella.

Quindi durante la fase di parsing della tabella bisogna impostare anche queste caratteristiche.

Per una maggiore comprensione del problema e su come risolverlo esaminiamo un caso di esempio:

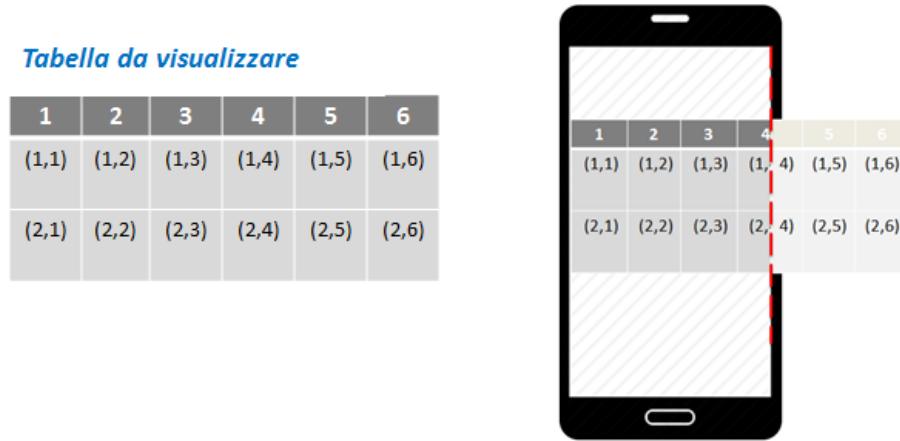


Figura 5. 10 - Illustrazione del problema di visualizzazione della tabella nel frame

vogliamo visualizzare sul dispositivo mobile una tabella 2x6, come mostrato in figura 5.10, ma a causa delle grandi dimensioni non riusciamo a visualizzarla interamente nel frame.

Per evitare ciò viene definito un elemento <DIV> che contiene tale tabella nel seguente modo:

```

1. <DIV class="divTab">
2.   <TABLE border="1" class="tab" id="Tabella0">
3.     .
4.   </TABLE>
5. </DIV>

```

Come si nota dal codice viene definito al <DIV> l'attributo *class* con valore "divTab". Tale classe è utile per definire le regole CSS, contenute nel file *index.css*, necessarie ad effettuare lo scrolling della tabella:

```

1.      .tabdiv{
2.          overflow-x: auto !important;
3.      }
4.
5.      .tabdiv::-webkit-scrollbar{
6.          background-color: #B2B2B2;
7.      }
8.
9.      .tabdiv::-webkit-scrollbar-thumb:window-inactive,
10.     .tabdiv::-webkit-scrollbar-thumb {
11.         background: #8F8F8F
12.     }
13.     .divTab::-webkit-scrollbar-button {
14.         background-size: 100%;
15.         background-color: #FFFFFF;
16.         height: 10px;
17.         width: 10px;
18.         -webkit-box-shadow: inset 1px 1px 2px rgba(0,0,0,0.2);
19.     }
20.
21.     .divTab::-webkit-scrollbar-button:horizontal:increment {
22.         background-image: url(../img/right_arrow.png);
23.         background-repeat: no-repeat;
24.     }
25.

```

Il risultato ottenuto è la visualizzazione della *Scrollbar* posta sotto alla tabella (figura 5.11), in modo tale da segnalare all'utente la presenza della tabella che va oltre il frame di visualizzazione.

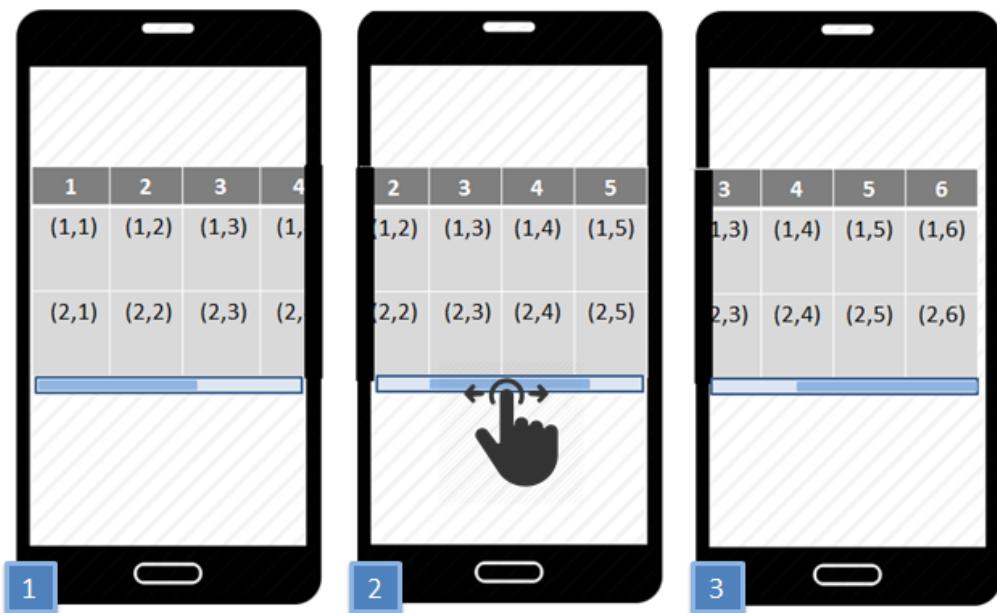


Figura 5. 11 - Illustrazione del funzionamento della barra di scrolling per la visualizzazione dell'intera tabella

## 5.4 Il parsing della tabella

Prima di addentrarci su come avviene la fase di parsing della tabella bisogna specificare che il tutto avviene mediante il metodo statico “*addTable()*”, richiamato all’interno del metodo, ereditato da **pComponet**, “*create()*” di **pTable**.

```

1. public void create() {
2.     parserHTML.addTable(_title, this);
3. }
```

Nel codice sottostante, il metodo “*addTable()*” definisce l’ elemento-tag *<table>* , la radice della tabella, impostando gli attributi *id*, *class* e il bordo della tabella (4-7). Per definire le righe, intestazioni , celle e le regole CSS, richiama al suo interno altri metodi fondamentali (8-10) che verranno analizzati di seguito.

Infine l’elemento-tag *<table>* viene aggiunto all’interno di un elemento-tag *<div>*, per gestirne l’impaginazione come spiegato nel paragrafo 5.3.4, che a sua volta viene aggiunto nell’*hashmap* degli elementi del documento *HTML* (12).

```

1. public static void addTable(String title, pTable table){
2.     Element divHTML = doc.createElement("DIV");
3.     divHTML.setAttribute("class", "divTab");
4.     Element tableHTML = doc.createElement("TABLE");
5.     tableHTML.setAttribute("id", title);
6.     tableHTML.setAttribute("class", "tab");
7.     tableHTML.setAttribute("border", "1");
8.     create_headerTable(tableHTML, table);
9.     create_rows(tableHTML, table);
10.    create_tableCSS(table, title);
11.    divHTML.appendChild(tableHTML);
12.    elements.put(title, divHTML);
13. }
```

### 5.4.1 La fase di parsing: metodo *createHeaderTable*

Il primo metodo ad essere richiamato come si può vedere dallo snippet precedente è il metodo “*create\_HeaderTable()*” il cui scopo è quello di definire la struttura in *HTML* dell’intestazione ed agganciarla all’elemento-tag *<table>*, passato come parametro nella funzione. Il secondo parametro della funzione è un riferimento al nostro oggetto **pTable** per poter ottenere dati relativi a comporre la struttura *HTML*.

Di seguito il codice Java del metodo:

```

1. private static void create_headerTable(Element tableHTML,
2.                                         pTable table){
3.     int num_colonne = table.getColumnCount();
4.     boolean flagHeader = false;
5.     Element trHeader = null;
6.     for(int k=0; k<num_colonne; k++){
7.         if(k==0) {
8.             trHeader = doc.createElement("TR");
9.             trHeader.setAttribute("id", table._title+"-
10.                                     rowHeader");
11.            flagHeader= true;
12.        }
13.        Element th = doc.createElement("TH");
14.        th.setAttribute("id", table._title+"-colHeader"+k);
15.        th.setAttribute("class", table._title+"-tabHeader");
16.        String colonnaNome = table.getColumnName(k);
17.        if(colonnaNome == null)
18.            th.appendChild(doc.createTextNode(""));
19.        else th.appendChild(doc.createTextNode(colonnaNome));
20.        trHeader.appendChild(th);
21.    }
22.    if(flagHeader == true) tableHTML.appendChild(trHeader);
23. }
```

Recuperando il numero di colonne della tabella (3) vengono create le celle `<th>`, settandone l'id (14), classe (15) e il valore (16-19); ogni cella viene appesa alla riga `<tr>` (20) creata alla riga (7-12). Dopo di che la riga d'intestazione viene appesa all'elemento-tag `<table>` (22).

#### 5.4.2 La fase di parsing: metodo `createRows`

Discorso analogo vale per il metodo `create_Rows()`: anziché costruire una singola riga che contiene le celle dell'intestazione, qui vengono create più righe (5-20) impostandone i vari attributi (7-8), e, iterativamente, vengono create le celle (9-18), settati gli attributi (11-12) e appese alla riga corrispondente (17), ognuna delle quali viene aggiunta all'elemento-tag `<table>`(19):

```

1. private static void create_rows(Element tableHTML, pTable
2.                                     table) {
3.     int num_righe = table.getRowCount();
4.     int num_colonne = table.getColumnCount();
5.     for(int i=0; i<num_righe; i++) {
6.         Element tr = doc.createElement("TR");
7.         tr.setAttribute("id", table._title+"-row"+i);
8.         tr.setAttribute("class", table._title+"-row");
9.         for(int j=0; j<num_colonne; j++) {
10.             Element td = doc.createElement("TD");
11.             td.setAttribute("id", table._title+"col"+i+"-"+j);
12.             td.setAttribute("class", table._title+"-column");
13.             String str = (String) table.getValueAt(i, j);
14.             if(str == null)
15.                 td.appendChild(doc.createTextNode(""));
16.             else td.appendChild(doc.createTextNode(str));
17.             tr.appendChild(td);
18.         }
19.         tableHTML.appendChild(tr);
20.     }
21. }
```

### 5.4.3 La fase di parsing: metodo *createTableCSS*

Con i metodi precedenti la struttura fisica della tabella è completa, non resta che settare le proprietà di layout. Se ne occuperà proprio l'ultimo metodo chiamato ovvero “*createTableCSS()*” il quale obiettivo è quello di creare un file CSS, di cui si è discusso nel paragrafo 5.3.3, “**tableCSS.css**”, e scrivere le regole della tabella.

Anche per il metodo *addTableCSS()* vengono passati come parametri l’oggetto **pTable** e l’id della tabella.

Inizialmente viene creato il file **tableCSS.css** in cui verranno scritte le regole CSS di ogni tabella (2). Si associa uno stream al file utilizzando la classe *FileOutputStream* (3), indicando che occorre scrivere alla fine del file (parametro impostato a *true*). Viene controllata l’esistenza del file: se non esiste viene creato un nuovo file altrimenti si utilizza il file presente aggiungendo le regole in coda a quelle già presenti all’interno (4-6). Per scrivere effettivamente le regole CSS nel file si utilizza un oggetto statico *PrintWriter* che prende in input lo stream definito (7).

```

1. try {
2.     File f = new File(tableCSS);
3.     FileOutputStream fos = new FileOutputStream(f, true);
4.     if(!f.exists()){
5.         f.createNewFile();
6.     }
7.     pw = new PrintWriter(fos);
    . . .
```

Una volta creato il file, si utilizza un metodo privato `writeCSSProperty()` per scrivere una regola CSS specifica.

```

1. public static void writeCssProperty(String selector,
2. String property, String newValue){
3.     pw.append(""+selector+"{" +property+ ":" +newValue+"}");
4.     pw.println();
5. }
```

Continuando ad analizzare il metodo `createTableCss()` vengono definite le regole CSS utilizzando proprio tale metodo di supporto:

```

. . .
1. writeCssProperty("." +table._title +"-column",
2.                     "color", table.foreground);
3. writeCssProperty("." +table._title +"-column",
4.                     "border-color", table.gridColor);
5. writeCssProperty("." + table._title +"-column",
6.                     "background-color", table.background);
7. writeCssProperty("." + table._title +"-row",
8.                     "height", "+" +table.getRowHeight() +"px");
9. writeCssProperty("." +table._title +"-column",
10.                      "border-width", "1px");
. . .
```

Come si può notare il metodo `writeCssProperty()` prende in input 3 parametri: un selettore, una proprietà e un valore. I selettori utilizzati sono selettori di classe in accordo al formato discusso nel paragrafo 5.3.1.1.

Una volta completata l'operazione di scrittura delle regole CSS, gli oggetti `PrintWriter` e `FileOutputStream` vengono chiusi:

```

. . .
1. pw.close();
2. fos.close();
3. } catch (IOException e) {
4.     e.printStackTrace();
5. }
```

Bisogna far presente che durante il parsing del metodo `createPage()`, responsabile nel creare la struttura *HTML* del documento, viene controllato se esiste il file **table.CSS** e in tal caso cancellato in modo tale da evitare l'utilizzo di regole definite precedentemente.

## 5.5 La Comunicazione da Java verso PhoneGap

In accordo al capitolo 3.3.1 vediamo nel dettaglio i metodi di **pTable** che si occupano di invocare la funzione `sendJavascript()` per poter comunicare con il documento *HTML* di **PhoneGap**.

### 5.5.1 Il metodo *setProperty*

Vari metodi presenti nella classe **pTable** impostano delle proprietà utili per costruire il layout grafico della tabella come il colore di sfondo, primo piano, dei bordi, altezza delle righe ecc...

Tutto ciò dal lato *HTML* si traducono in regole *CSS* per la tabella. I metodi invocano lato *Java* la funzione JS “*app.setProperty*” utilizzando proprio la funzione *sendJavascript()*.

Un esempio di tali metodi è *setBackground(String bg)*, che imposta il colore di sfondo, ecco il codice:

```

1. public void setBackground(String bg) {
2.     if (bg == null) {
3.         throw new IllegalArgumentException(
4.                 "New color is null");
5.     }
6.     if(!bg.equals(this.background)) {
7.         String oldBg = this.background;
8.         this.background = bg;
9.         try {
10.             CordovaWebView wv = Constants.view;
11.             if(wv != null){
12.                 String str = "app.setProperty"
13.                     + "("
14.                     + title+"", "" //table id
15.                     +"background"+ "", ""//property change
16.                     +oldBg+"", "" //old value
17.                     +bg+"", "" //new value
18.                     +"column" //type
19.                     +")";
20.                 wv.sendJavascript(str);
21.             }
22.         } catch(Exception e) {}
23.     }
24. }
```

Alle righe (1-8) viene controllato e settato il valore del background della classe **pTable**. Per poter comunicare il nuovo background al componente *HTML* `<table>` bisogna utilizzare la funzione *sendJavascript(String str)* che invocherà la funzione JS *app.setProperty()* avente come parametri: l'id della tabella, la proprietà *CSS* che si desidera cambiare, il vecchio valore di background, il nuovo valore di background e un ultimo valore utile ad impostare correttamente il selettore della regola *CSS*.

In generale la funzione *app.setProperty()* permette, dunque, di impostare un valore di una regola *CSS* che abbia il selettore di classe, nei formati evidenziati in 5.3.1., e come attributo *property*.

La ricerca della regola CSS da modificare avviene recuperando tutti i file CSS linkati nel documento:

```
1. var styleSheets = window.document.styleSheets;
2. var styleSheetsLength = styleSheets.length;
```

Dopo di che su ognuno di essi vengono recuperate tutte le regole definite all'interno analizzandole una per una,

```
1. for(var i = 0; i < styleSheetsLength; i++) {
2.   var classes = styleSheets[i].rules ||
3.               styleSheets[i].cssRules;
4.   if (!classes)
5.     continue;
6.   var classesLength = classes.length;
7.   for (var x = 0; x < classesLength; x++) {
.....
```

trovando quella da modificare in base ai parametri passati in input ovvero:

```
1. if(selection != "selection"){
2.   if (classes[x].selectorText == "."+id.toLowerCase()+
3.       "-"+type.toLowerCase()) {
4.     classes[x].style[property.toLowerCase()] = newValue;
5.   }
6.   if(classes[x].selectorText == ".+"+id.toLowerCase()
7.       +"-"+type.toLowerCase()+"-selected"){
8.     if(property.toLowerCase() != "background" &&
9.         property.toLowerCase() != "color"){
10.       classes[x].style[property.toLowerCase()] =
11.           newValue;
12.     }
13.   }
14. }
15. else{
16.   if(classes[x].selectorText == "."+id.toLowerCase()
17.       +"-"+type.toLowerCase()+"-selected"){
18.     classes[x].style[property.toLowerCase()] =
19.       newValue;
}
```

Una ricreazione degli step eseguiti:

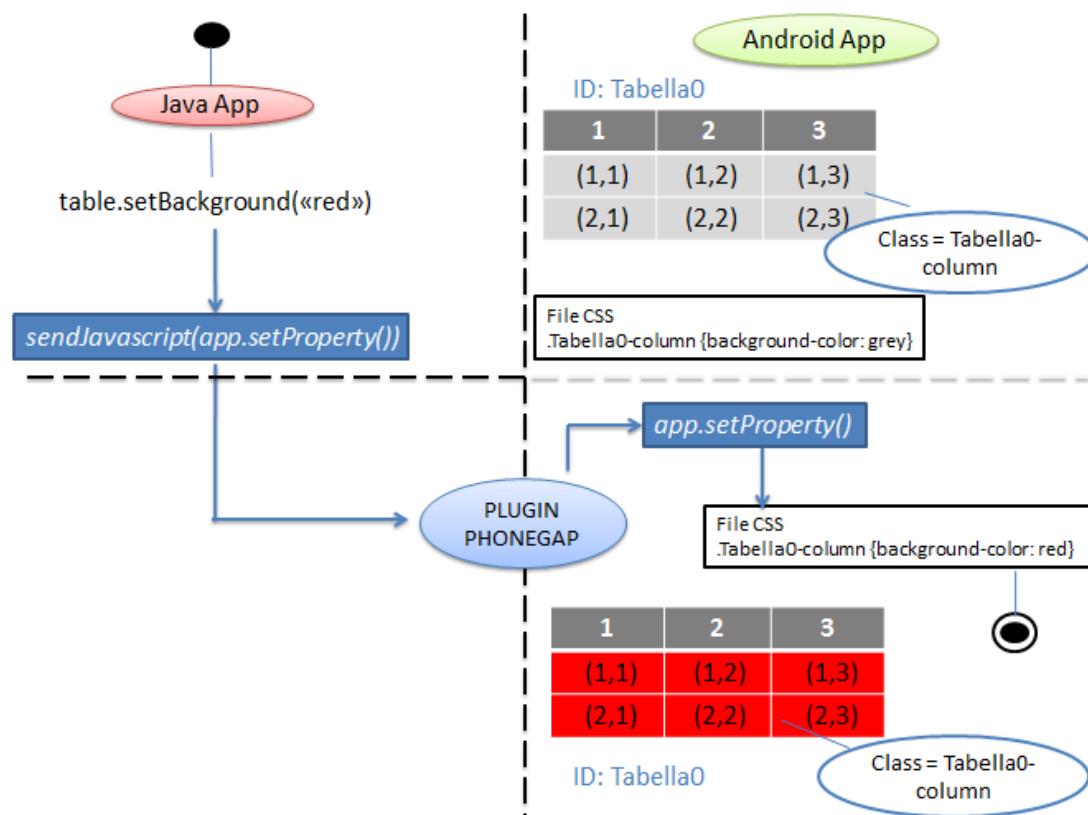


Figura 5. 12- Diagramma illustrativo della modifica di una proprietà della tabella da Java a Mobile

Nel caso dell'esempio nel settare la proprietà di background viene modificata la regola CSS (del file `tableCSS.css`) che ha come selettore `".Tabella0-column"` per impostare la proprietà del background-color di ciascuna cella.

### 5.5.2 Il metodo *setRowSelection*

Altro metodo richiamato lato Java nella funzione *sendJavascript()* è il metodo *setRowSelection()*: esso viene richiamato nel metodo di **pTable** *setRowSelectionInterval()* come mostrato qui di seguito.

```

1. public void setRowSelectionInterval(int index0, int index1) {
2.     int[] rowsOld = this.getSelectedRows();
3.     int oldIndex=-1;
4.     if(rowsOld.length != 0){
5.         oldIndex = rowsOld[rowsOld.length-1];
6.     }
7.     getSelectionModel().setSelectionInterval(boundRow(index0),
8.                                              boundRow(index1));
9.     try {
10.         CordovaWebView wv = Constants.view;
11.         if(wv != null){
12.             String str = "app.setRowSelection"
13.                 + "("
14.                 + _title+"'", "'"
15.                 +oldIndex+"'", "'"
16.                 +index0
17.                 +")";
18.             wv.sendJavascript(str);
19.             if(oldIndex == index0){
20.                 getSelectionModel().clearSelection();
21.             }
22.         }
23.     } catch(Exception e) {}
24. }
```

La logica è analoga a quella del metodo *setBackground()* visto nell' esempio precedente. Le righe (1-8) determinano la selezione delle righe.

Le righe (10-18) costruiscono la richiesta da invocare nella funzione *sendJavascript()*, ovvero richiamano la funzione *setRowSelection()* passando come parametri l'id della tabella, l'indice della vecchia riga selezionata e l'indice della nuova riga selezionata. Infine viene controllato se l'indice della riga precedente e quella nuova siano le stesse (19-21); se così fosse allora bisogna ripulire la selezione in modo tale da permettere la de-selezione di una riga.

Analizziamo ora la funzione JavaScript `app.setRowSelection()`:

```

1. setRowSelection: function(id, rowsOld, rowsNew) {
2.     var tds =
3.         document.getElementById(id).getElementsByName("td");
4.     for(var j=0; j<tds.length; j++) {
5.         var result = tds[j].id.split("-");
6.         var cell = [];
7.         result[1].match(/\d+/g).forEach(function(i,j){
8.             cell[j]=parseInt(i);
9.         });
10.        if(cell[0] == rowsOld) {
11.            tds[j].setAttribute("class", ""+id+"-column");
12.            tds[j].removeAttribute("contenteditable");
13.        } else if(cell[0] == rowsNew) {
14.            tds[j].setAttribute("class",
15.                id+"-column-selected");
16.            tds[j].removeAttribute("contenteditable");
17.        } else {
18.            tds[j].setAttribute("class",
19.                id+"-column");
20.            tds[j].removeAttribute("contenteditable");
21.        }
22.    }
23. }
```

L'id della tabella viene utilizzato per recuperare tutte le celle della tabella (2-3). A questo punto vengono analizzate in un ciclo for (4-22), dove per ognuna di esse viene estratto (5-8) dal proprio id la posizione all'interno della tabella in quanto l'id è costruito nel formato per esempio: "Tabella0-col0,1" , che sta ad indicare la cella della tabella con id Tabella0 la cui posizione è alla riga 0 e colonna 1. Analizziamo ora le condizioni espresse nelle righe (9-21):

- *le righe (9-12)* controllano che l' indice della riga della cella sia uguale all'indice delle riga precedentemente selezionata (9); se così fosse allora viene effettuata un cambiamento di classe impostando come nuova classe quella di default della cella;
- *le righe (13-16)* controlla che l' indice della riga della cella sia uguale all'indice delle riga che deve essere selezionata (13); se così fosse allora viene effettuata un cambiamento di classe impostando come nuova classe quella della cella selezionata;
- *le righe (17-21):* se non rientra nei casi precedenti, viene impostata la classe di default della cella.

Ripercorriamo tramite la seguente illustrazione gli step, su un esempio di selezione dell'intervallo della seconda riga:

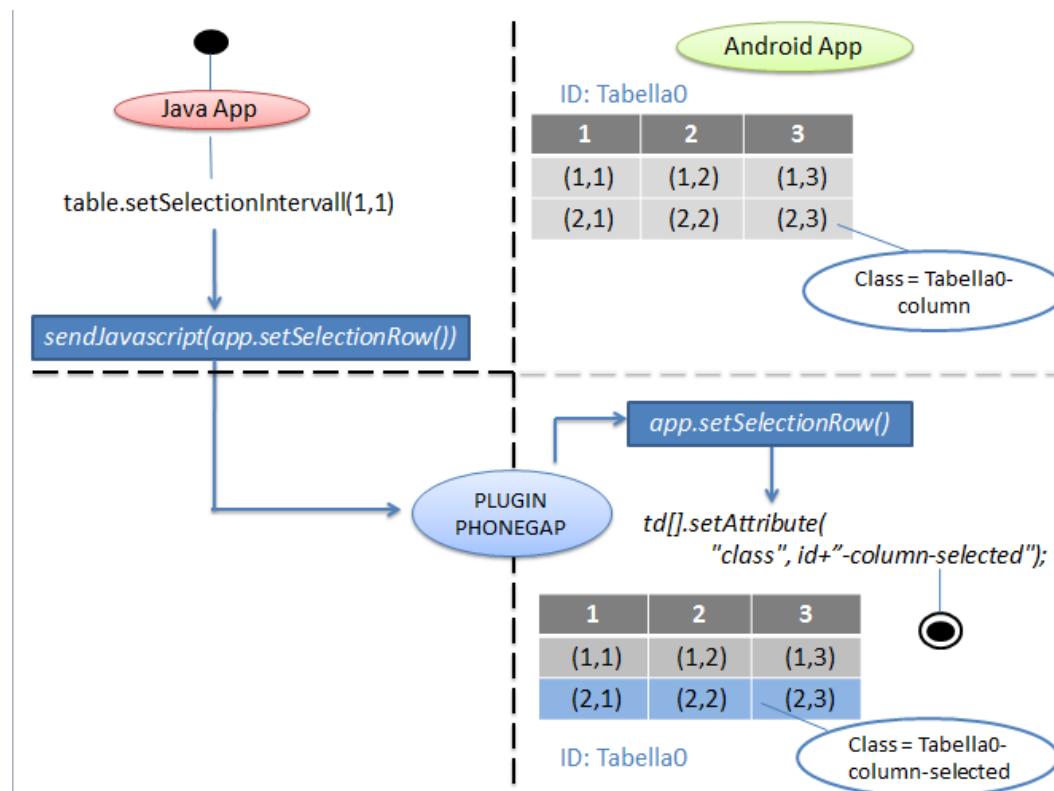


Figura 5. 13 - Diagramma illustrativo della selezione di una riga da Java a Mobile

### 5.5.3 Il metodo `insertRow`

Come discusso precedentemente, tra i vari metodi presenti nella classe `pTable` vi è un metodo in particolare, ovvero: `tableChanged()`. Per capire come tale metodo si comporta forniamo un esempio di utilizzo.

Consideriamo che nel nostro codice Java si stia lavorando sul modello di dati di una tabella e vogliamo che si aggiungi a tale modello una nuova riga, cioè

```

1. private pTable tabella;
2. private DefaultTableModel model;
3. ...
4. public Esempio(){
5. ...
6.     tabella = new pTable(model);
7.     model.addTableModelListener(tabella);
8. ...
9.     Object[] obj = {"cella0", "cella1", "cella2"};
10.    model.addRow(obj);
11. ...
12. }
```

A questo punto una volta che il metodo `addRow()` della classe `DefaultTableModel` ha terminato viene invocato un metodo `fireTableRowsInserted()` metodo ereditato dalla classe `AbstractTableModel` che al suo interno richiama `fireTableChanged()` con una serie di valori.

In questo modo viene notificato l'avvenuto evento, in questo caso un **INESERT**, al metodo `tableChanged()` dell'oggetto `pTable`:

```

1. @Override
2. public void tableChanged(TableModelEvent e) {
3.     if (e.getType() == TableModelEvent.INSERT) {
4.         tableRowsInserted(e);
5.         return;
6.     }
7.     if (e.getType() == TableModelEvent.DELETE) {
8.         tableRowsDeleted(e);
9.         return;
10.    }
11. }
```

In base all'evento ricevuto si stabilisce che tipo di azione eseguire (inserimento o cancellazione); nel nostro esempio si invoca il metodo `tableRowsInserted()` appartenente alla classe `pTable`.

```

1. public void tableRowsInserted(TableModelEvent e) {
2.     try{
3.         CordovaWebView wv = Constants.view;
4.         if(wv != null){
5.             String data = "{";
6.             for(int i=0; i<getColumnCount(); i++){
7.                 data += "\\" + i + "\\":\\" + getValueAt(e.getFirstRow(), i) + "\\";
8.                 Int c=i+1;
9.                 if(c != getColumnCount()) data+="," ;
10.            }
11.            data += "}";
12.            String str = "app.insertRow"
13.            + "("
14.            + title + ", "
15.            + data
16.            + ")";
17.            wv.sendJavascript(str);
18.        }
19.    }
20. } catch(Exception exc) {}
21. }
```

In input riceve un oggetto `TableModelEvent` utile a recuperare le informazioni sull'indice della riga appena inserita. Le righe (5-12) costruiscono una stringa in formato `JSON` per `JavaScript` contenente l'indice della colonna e il valore della cella della riga appena inserita: nel nostro esempio sarà:

“{“0”：“cella0”, “1”：“cella1”, “2”：“cella2”}”.

Mentre le righe (13-18) costruiscono la richiesta da invocare nella funzione *sendJavascript()*, cioè la funzione *app.insertRow()* che verrà spiegata nel dettaglio ora.

```
1.     insertRow: function(id, data) {
2.         var tab = document.getElementById(id);
3.         var trs = tab.getElementsByTagName("tr");
4.         var index = trs.length -1;
5.         var row = tab.insertRow(trs.length);
6.         row.setAttribute("id", id+"-row"+index);
7.         row.setAttribute("class", "+id+-row");
8.         addFireRowListener(row, row.id);
9.         var obj = JSON.parse(data);
10.        for(var k in obj) {
11.            var cell = row.insertCell(k);
12.            cell.innerHTML = "+obj[k]";
13.            cell.setAttribute("id", "+id+-" +
14.                            col"+index+", "+k);
15.            cell.setAttribute("class", "+id+-column");
16.            addFireCellListener(cell, cell.id,
17.                                cell.innerHTML);
18.        }
19.    }
```

I parametri passati alla funzione sono l'id della tabella e la stringa in formato *JSON* creata precedentemente. Viene creato un nuovo elemento *<TR>* (2-7) aggiungendolo alla tabella corrispondente a tale id e il relativo evento (8). Le righe (9-17) infine crea e appende alla riga *<TR>* appena creata i nuovi elementi *<TD>* in base alla stringa *JSON* contenente i valori appositi e ne vengono registrati gli eventi (16). In questo modo si completa l'inserimento di una nuova riga ed essa potrà essere visualizzata nel layout del documento *HTML*.

Riportiamo anche in questo caso i vari passaggi :

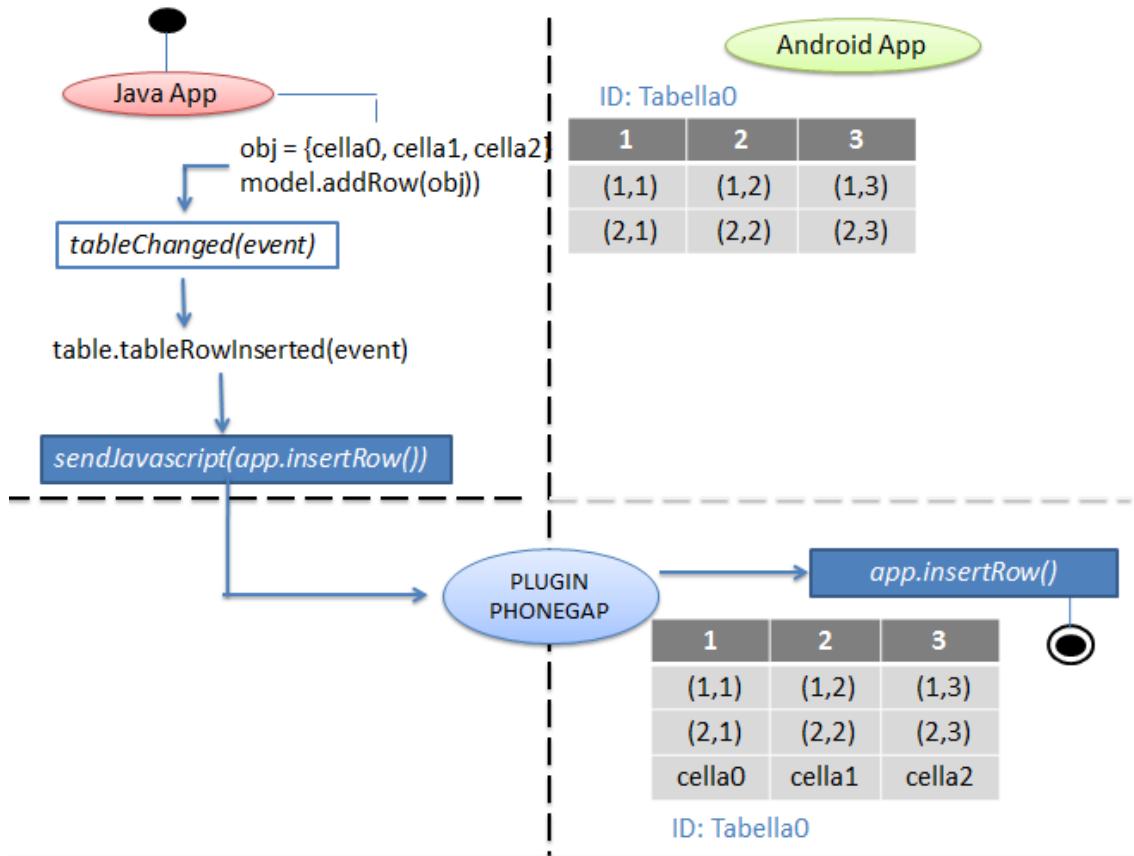


Figura 5. 14 - Diagramma illustrativo dell'inserimento di una riga da Java a Mobile

### 5.5.4 Altri metodi

La classe **pTable** contiene molti altri metodi, il cui comportamento risulta essere simile ai precedenti, che invocano funzioni *JavaScript*.

Per completezza nella seguente tabella vengono schematizzati tutti i metodi **pTable** e le corrispondenti funzioni lato *JavaScript* che richiamano:

Metodo pTable	Funzione Javascript
setBackground()	setProperty(id, background, oldbg, bg, column)
setForeground()	setProperty(id, color, oldfg, fg, column)
setGridColor()	setProperty(id, border-color, oldgd, gd, column)
setIntercellSpacing()	setProperty(id, padding-left, old, width, column) setProperty(id, padding-right, old, width, column)
setRowHeigth	setProperty(id, heigth, oldH, rowHeight, row)
setRowMargin()	setProperty(id, padding-top, oldM, rowMargin, column) setProperty(id, padding-bottom, oldM, rowMargin, column)
setRowSelectionInterval()	setRowSelection(id, oldIndex, newIndex)
setSelectionBackground()	setProperty(id, background, oldbg, bg, column, selected)
setSelectionForeground()	setProperty(id, color, oldfg, fg, column, selected)
setValueAt()	setValue(id, row, column, value)
tableRowsInserted()	insertRow(id, data)
tableRowsDeleted()	removeRow(id, index)

Figura 5. 15 - Tabella riassuntiva dei metodi di pTable e relative funzioni js richiamate

## 5.6 La comunicazione da PhoneGap verso Java

Come ben sappiamo per permettere la comunicazione tra la *WebView* di **Phonegap** e le classi **pAwt** ci viene incontro il **plugin: PhoneGap** attraverso la chiamata *cordova.exec()* passa il controllo al codice Java, come spiegato nel capitolo 3.3.2.

Le funzioni aggiunte che sono chiamate dalla *cordova.exec()* per la gestione della tabella risultano:

- **FireRow**

```

1. fireRow: function(id, row, successCallback, errorCallback) {
2.     cordova.exec(
3.         successCallback,
4.         errorCallback,
5.         'AWTPlugin',
6.         'fireRow',
7.         [
8.             {
9.                 "id": id,
10.                "row": row
11.            }
12.        ]
13.    );
14. }
```

Nell'array di parametri passati dalla funzione viene specificato l'id del componente *HTML* che ha generato l'evento “**fireRow**”, l'indice della riga selezionata dall'utente per poterla elaborare.

- **FireCell**

```

1. fireCell: function(id, row, column, successCallback,
2. errorCallback) {
3.     cordova.exec(
4.         successCallback,
5.         errorCallback,
6.         'AWTPlugin',
7.         'fireCell',
8.         [
9.             {
10.                "id": id,
11.                "row": row,
12.                "column": column
13.            }
14.        ]
15.    );
16. }
```

Nell'array di parametri passati dalla funzione viene specificato l'id del componente *HTML* che ha generato l'evento “**fireCell**”, l'indice della riga e

della colonna utili ad identificare la cella selezionata dall'utente per poterla elaborare.

- **InsertTable**

```
1. insertTable: function(id, row, column, value, successCallback,
2.   errorCallback) {
3.     cordova.exec(
4.       successCallback,
5.       errorCallback,
6.       'AWTPPlugin',
7.       'insertTable',
8.       [
9.         {
10.           "id": id,
11.           "row": row,
12.           "column": column,
13.           "value": value
14.         }
15.     );
}
```

Nell'array di parametri passati dalla funzione viene specificato l'id del componente *HTML* che ha generato l'evento “**fireCell**”, l'indice della riga e della colonna utili ad identificare la cella selezionata dall'utente per poterla elaborare, il valore della contenuto dalla cella.

Nella nostra classe lato Java che estende la classe “**CordovaPlugin**” al metodo “**execute()**”, per quanto riguarda gli eventi scatenati dalla tabella, andranno aggiunti all’implementazione nuove *action*:

```

1. else if (action.equals("fireRow")){
2.
3.     JSONObject arg_object = args.getJSONObject(0);
4.     String id = arg_object.getString("id");
5.     String row = arg_object.getString("row");
6.     component.fireRow(id, row);
7.
8. } else if (action.equals("fireCell")){
9.
10.    JSONObject arg_object = args.getJSONObject(0);
11.    String id = arg_object.getString("id");
12.    String row = arg_object.getString("row");
13.    String column = arg_object.getString("column");
14.    component.fireCell(id, row, column);
15.
15.} else if (action.equals("insertTable")){
16.
17.    JSONObject arg_object = args.getJSONObject(0);
18.    String id = arg_object.getString("id");
19.    String row = arg_object.getString("row");
20.    String column = arg_object.getString("column");
21.    String value = arg_object.getString("value");
22.    component.change(id, row, column, value);
23. }

```

Per poter effettuare il dispatch dei metodi bisogna aggiornare anche la classe **pFrame** della nostra libreria **pAWT**, che si occuperà del recupero del componente dalla tabella hash e dell'invocazione del metodo su quell'oggetto.

Di seguito vediamo il codice da aggiungere a **pFrame**:

```

1. public void fireRow(String id, String row) {
2.     rep.fireRow(id, row);
3. }

4. public void fireCell(String id, String row, String column) {
5.     rep.fireCell(id, row, column);
6. }

7. public void change(String id, String row, String column,
8.     String value) {
9.     rep.change(id, row, column, value);
}

```

I metodi della classe **pFrame** chiamano i metodi delle classe **Repository** la quale si occupa di recuperare la componente occorrente ed intraprendere le azioni richieste su di essa. Dunque alla classe **Repository** andranno aggiunti i seguenti metodi:

```
1.  public void fireRow(String key, String posRow) {
2.      pComponent component = find(key);
3.      if(component != null && component instanceof pTable){
4.          pTable t = (pTable)component;
5.          if(t.getRowSelectionAllowed()){
6.              t.setRowSelectionInterval(Integer.parseInt(posRow),
7.              Integer.parseInt(posRow));
8.          }
9.      }
10. }

11. public void fireCell(String key, String posRow, String
12. posColumn) {
13.     pComponent component = find(key);
14.     if(component != null && component instanceof pTable){
15.         pTable t = (pTable)component;
16.         if(t.isCellEditable(Integer.parseInt(posRow),
17.                         Integer.parseInt(posColumn))){
18.             t.editCellAt(Integer.parseInt(posRow),
19.                         Integer.parseInt(posColumn));
20.         }
21.     }
22. }

23. public void change(String key, String posRow, String
24. posColumn, String value) {
25.     pComponent component = find(key);
26.     if(component != null && component instanceof pTable){
27.         pTable t = (pTable)component;
28.         t.getModel().setValueAt(value,
29.                         Integer.parseInt(posRow),
30.                         Integer.parseInt(posColumn));
31.     }
32. }
```

# 6. Caso di studio

In questo capitolo viene illustrato il processo di miniaturizzazione applicato ad casi di studio ponendo attenzione all'utilizzo della componente tabella.

## 6.1 ListShop



Figura 6. 1 - L'interfaccia grafica ListShop di Java

Il programma Java mostra, come in figura, tre *JTextField*, due *JButton*, un *JTable*.

La tabella contiene già una riga con dei valori. Attraverso questo semplice programma è possibile effettuare tre operazioni:

- **Inserimento di una riga:**  
compilare i campi, inserendo un oggetto, il costo e il negozio, e cliccando sul pulsante “Aggiungi” viene aggiunta una nuova riga nella tabella.
- **Rimozione di una riga:**  
selezionando una riga della tabella con un click e schiacciando il pulsante “Rimuovi”, viene rimossa dalla tabella la riga appena selezionata.
- **Modificare il valore di una cella:**  
Effettuando un doppio click su una cella è possibile modificarne il valore;

### 6.1.1 Il codice Java AWT

Diamo uno sguardo a come è stato strutturato il codice Java della classe **ListShop** che utilizza i package *AWT* e *SWING*. Oltre ai vari import delle librerie necessarie, vengono definiti all'interno della classe i riferimenti necessari:

```

1. public class Listashop {

2.     private JFrame mainFrame;
3.     private JTable tabella;
4.     private JTextField oggetto;
5.     private JTextField costo;
6.     private JTextField negozio;
7.     private JButton aggiungi;
8.     private JButton rimuovi;
9.     private DefaultTableModel model;
10.    private String[] columnName = {
11.        "Oggetto", "Costo", "Negozio"};
12.    private Object[][] data = {
13.        {"Giacca", "60", "Sottotono" },
14.    };
15.
16.    ....

```

Dopodichè troviamo il costruttore ListShop il cui compito è quello di costruire l'interfaccia grafica: vengono creati gli oggetti e aggiunti ai panelli;

```

.....
1. public Listashop()
2. {
3.     model = new DefaultTableModel(data,columnName);
4.     mainFrame = new JFrame("Java AWT Listashop");
5.     tabella = new JTable(model);
6.     oggetto = new JTextField("Oggetto", 12);
7.     costo = new JTextField("Costo", 6);
8.     negozio = new JTextField("Negozio", 12);
9.     aggiungi = new JButton("Aggiungi");
10.    rimuovi = new JButton("Rimuovi");
11.
12.    JPanel panelUp = new JPanel();
13.    panelUp.setLayout(new FlowLayout());
14.    panelUp.add(oggetto);
15.    panelUp.add(costo);
16.    panelUp.add(negozio);
17.    panelUp.add(aggiungi);
18.    panelUp.add(rimuovi);
19.
20.    JPanel panelTable = new JPanel();
21.    panelTable.setLayout(new BorderLayout());
22.    panelTable.add(tabella.getTableHeader(),
23.                  BorderLayout.PAGE_START);
24.    panelTable.add(tabella, BorderLayout.CENTER);
25.
26.    ....

```

Infine tali panelli vengono aggiunti al frame principale, al quale vengono impostati dimensioni, layout e visibilità.

```

public ListaShop() {
    ....
    1. mainFrame.setLayout(new BorderLayout());
    2. mainFrame.add(panelUp, BorderLayout.PAGE_START);
    3. mainFrame.add(panelTable, BorderLayout.CENTER);
    4. mainFrame.setVisible(true);
    5. mainFrame.pack();
    ....
}

```

Non resta che impostare gli *ActionListener* sui pulsanti “aggiungi” e “rimuovi” che sono responsabili di conseguenza di aggiungere e rimuovere una riga al modello di dati. Per lanciare l’applicazione eseguiamo il metodo *main()*.

```

.....
1.     model.addTableModelListener(tabella);
2.
3.     aggiungi.addActionListener(new ActionListener() {
4.         public void actionPerformed(ActionEvent e) {
5.             Object[] obj = {oggetto.getText(),
6.                             costo.getText(), negozio.getText()};
7.             model.addRow(obj);
8.         }
9.     });
10.
11.    rimuovi.addActionListener(new ActionListener() {
12.        public void actionPerformed(ActionEvent e) {
13.            int[] rows = tabella.getSelectedRows();
14.            for(int i: rows){
15.                model.removeRow(i);
16.            }
17.        }
18.    });
19. }

20. public static void main(String[] args) {
21.     new ListaShop();
22. }
23.
}

```

### 6.1.2 La fase di parsing

Dopo aver creato la classe wrapper *ListaShop* in **pAWT**, viene creato durante la fase di parsing il documento HTML: vengono settati i riferimenti ai file CSS all’interno del tag <HEAD>.

```

1. <HTML>
2.   <HEAD>
3.     <META http-equiv="Content-Type" content="text/html;
4.       charset=UTF-8">
5.     <LINK href="css/index.css" rel="stylesheet"
6.       type="text/css">
7.     <LINK href="css/ionic.css" rel="stylesheet"
8.       type="text/css">
9.     <LINK href="css/ionicons.min.css" rel="stylesheet"
10.       type="text/css">
11.     <LINK href="css/menu/menulaterale.css" rel="stylesheet"
12.       type="text/css">
13.     <LINK href="css/tableCSS.css" rel="stylesheet"
14.       type="text/css">
15.     <META content="user-scalable=no, initial-scale=1,
16.       maximum-scale=1, minimum-scale=1, width=device-
17.       width, height=device-height, target-
densitydpi=device-dpi" name="viewport">
18.   <TITLE>index.html</TITLE>
19. </HEAD>

```

....

Viene poi definito il corpo del documento dove ogni oggetto *pAWT* viene sostituito con il corrispondente oggetto *HTML*. Così come spiegato nel capitolo 4.2 nel seguente esempio l'applicazione viene suddivisa in due frame: il primo contiene i campi necessari all'inserimento di una nuova riga e il pulsante "aggiungi";

```

1. <BODY onhashchange="updateFooter();" onload="defaultFrame();">
2.   <div class="frame overlay" id="myNav">
3.     <a class="closebtn" href="javascript:void(0)"
4.       onclick="closeNav()">x</a>
5.     <span class="overlay-content">
6.       <ul></ul>
7.     </span>
8.   </div>
9.   <DIV class="frame" id="frame0">
10.    <div class="scritta_menu" onclick="openNav()>|||
11.      MENU</div>
12.    <DIV class="areaditestो">
13.      <LABEL><INPUT
14.        id="Oggetto" onkeyup=
15.          app.insert(id,document.getElementById(id).value)
16.          placeholder="Oggetto" type="text"></LABEL>
17.    </DIV>

```

```
18.     <DIV class="areaditestostoclienti">
19.         <LABEL><INPUT
20.             id="Costo" onkeyup="
21.                 app.insert(id,document.getElementById(id).value)
22.                 placeholder='Costo' type='text'></LABEL>
23.     </DIV>
24.     <DIV class="areaditestostoclienti">
25.         <LABEL><INPUT
26.             id="Negozio" onkeyup="
27.                 app.insert(id,document.getElementById(id).value)
28.                 placeholder='Negozio' type='text'></LABEL>
29.     </DIV>
30.     <BUTTON class="button" id="Aggiungi"
31.         onClick="app.fire(id)" type="button">Aggiungi</BUTTON>
32. </DIV>
```

il secondo contiene la tabella con i dati, il pulsante "rimuovi".

```
1. <DIV class="frame" id="frame1">
2.   <div class="scritta_menu" onclick="openNav()">||| MENU</div>
3.   <BUTTON class="button" id="Rimuovi" onClick="app.fire(id)"
4.     type="button">Rimuovi</BUTTON>
5.   <TABLE border="1" class="tab" id="Tabella0">
6.     <TR id="Tabella0-rowHeader">
7.       <TH class="Tabella0-tabHeader"
8.         id="Tabella0-colHeader0">Oggetto</TH>
9.       <TH class="Tabella0-tabHeader"
10.        id="Tabella0-colHeader1">Costo</TH>
11.       <TH class="Tabella0-tabHeader"
12.         id="Tabella0-colHeader2">Negozio</TH>
13.     </TR>
14.     <TR class="Tabella0-row" id="Tabella0-row0">
15.       <TD class="Tabella0-column"
16.         id="Tabella0-col0,0">Giacca</TD>
17.       <TD class="Tabella0-column"
18.         id="Tabella0-col0,1">60</TD>
19.       <TD class="Tabella0-column"
20.         id="Tabella0-col0,2">Sottotono</TD>
21.     </TR>
```

Alla fine viene aggiunta la sezione che identifica la presenza dei frame e i file JavaScript per la gestione delle componenti.

```
....  
1. <DIV class="bar bar-positive bar-footer" id="footer">  
2.   <BUTTON class="hide button button-clear icon-left ion-  
3.     android-arrow-dropleft-circle" id="buttonLeft"></BUTTON>  
4.   <DIV class="title" id="framesBullets"></DIV>  
5.   <BUTTON class="hide button button-clear icon-right ion-  
6.     android-arrow-dropright-circle" id="buttonRight"></BUTTON>  
7. </DIV>  
8. <SCRIPT src="js/hammer.min.js"  
9.   type="text/javascript"></SCRIPT>  
10.<SCRIPT src="js/index.js"  
11.  type="text/javascript"></SCRIPT>  
12.<SCRIPT src="js/test.js"  
13.  type="text/javascript"></SCRIPT>  
14.<SCRIPT src="js/cordova.js"  
15.  type="text/javascript"></SCRIPT>  
16.<SCRIPT src="js/phonegap.js"  
17.  type="text/javascript"></SCRIPT>  
18.<SCRIPT src="js/ionic.bundle.min.js"  
19.  type="text/javascript"></SCRIPT>  
20.<SCRIPT src="js/menu/menulaterale.js"  
21.  type="text/javascript"></SCRIPT>  
22.<SCRIPT src="js/tableEvent.js"  
23.  type="text/javascript"></SCRIPT>  
24. </BODY>  
25.</HTML>
```

Riportiamo anche il file CSS creato durante la fase di parsing:

```
.Tabella0-TabHeader { padding: 5px }  
.Tabella0-TabHeader { font-weight: BOLD }  
.Tabella0-TabHeader { text-align: CENTER }  
.Tabella0-TabHeader { background-color: ALICEBLUE }  
.Tabella0-TabHeader { border-color: BLACK }  
.Tabella0-TabHeader { border-style: SOLID }  
.Tabella0-TabHeader { border-width: 1px }  
.Tabella0-column-selected { padding-right: 1px }  
.Tabella0-column-selected { padding-left: 1px }  
.Tabella0-column-selected { padding-bottom: 20px }  
.Tabella0-column-selected { padding-top: 20px }  
.Tabella0-row-selected { height: 50px }  
.Tabella0-column-selected { background-color: SKYBLUE }  
.Tabella0-column-selected { border-color: BLACK }  
.Tabella0-column-selected { color: WHITE }  
.Tabella0-column-selected { border-style: solid }  
.Tabella0-column-selected { border-width: 1px }  
.Tabella0-column { padding-right: 1px }  
.Tabella0-column { padding-left: 1px }  
.Tabella0-column { padding-bottom: 20px }
```

```
.Tabella0-column { padding-top: 20px }
#Tabella0 { border-collapse: collapse }
.Tabella0-column { border-style: solid }
.Tabella0-column { border-width: 1px }
.Tabella0-row { height: 50px }
.Tabella0-column { background-color: GAINSBORO }
.Tabella0-column { border-color: BLACK }
.Tabella0-column { color: BLACK }
```

Facendo partire l'applicazione sul cellulare, si presenterà in questo modo:

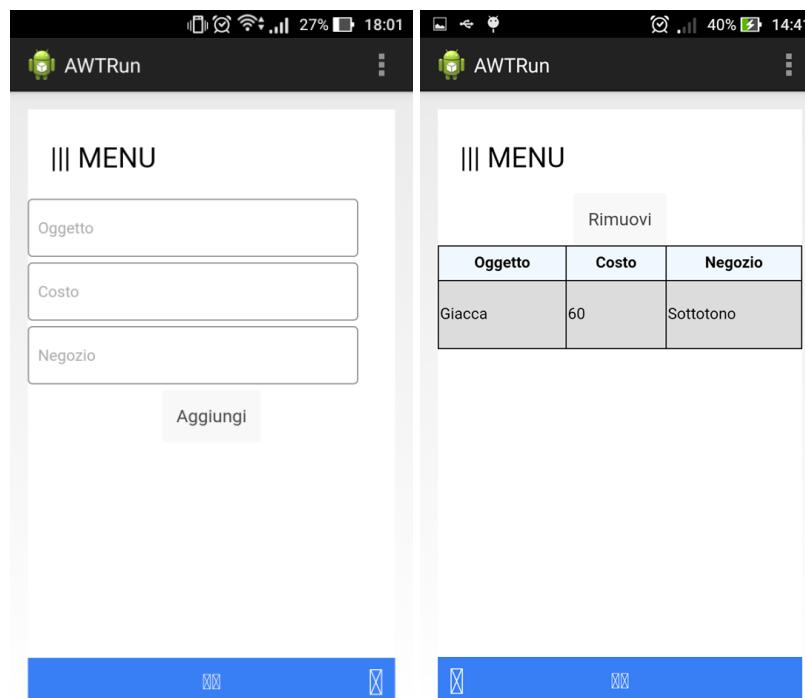


Figura 6. 2 - L'interfaccia grafica ListShop su Android

### 6.1.3      *Simulazione Java-Mobile: inserimento di una riga*

Compiliamo i campi necessari specificando l' oggetto “*Jeans*”, il costo “35”e il negozio in cui è stato acquistato “*Zara*”. Infine clicchiamo sul pulsante “*aggiungi*” per completare l'operazione (Figura 6.3 e 6.4).



Figura 6. 3 - Esempio di completamento dei campi e click per l'aggiunta di una riga in Java

## CAPITOLO 6 Caso di studio

Quello che accade è che sul bottone è stato registrato un actionlistener la cui azione è quella di aggiungere al modello di dati una riga:

```
Object[] obj = {  
    oggetto.getText(),  
    costo.getText(),  
    negozio.getText()};  
model.addRow(obj);
```

Tutto ciò comporta l'invocazione del metodo di pTable *tableChanged()*, il quale invoca la funzione *tableRowsInserted()* che si occupa di recuperare le informazioni necessarie sulla riga inserita nel modello di dati ed invoca la funzione *sendjavascript()* richiamando *app.insertRow(id,data)*, dove il parametro *id* è "Tabella0" mentre *data* è una stringa nel formato JSON {"0":"Jeans","1":"35","2":"Zara"}.



Figura 6.4 - Esempio di completamento dei campi e click per l'aggiunta di una riga in Android

Tale funzione *JavaScript* si occuperà di aggiungere dinamicamente alla tabella *HTML* la nuova riga con le informazioni che ha ricevuta dal lato Java e registrare gli eventi supportati (Figura 6.6). Se vogliamo fare un paragone dal punto di vista statico in *HTML* avremo:

```
<TR class="Tabella0-row" id="Tabella0-row1">  
    <TD class="Tabella0-column"  
        id="Tabella0-col1,0">Jeans</TD>  
    <TD class="Tabella0-column"  
        id="Tabella0-col1,1">35</TD>  
    <TD class="Tabella0-column"  
        id="Tabella0-col1,2">Zara</TD>  
</TR>
```

Oggetto	Costo	Negozio
Giacca	60	Sottotono
Jeans	35	Zara

Figura 6.5 - Esempio di visualizzazione della tabella dopo l' inserimento della riga in Java

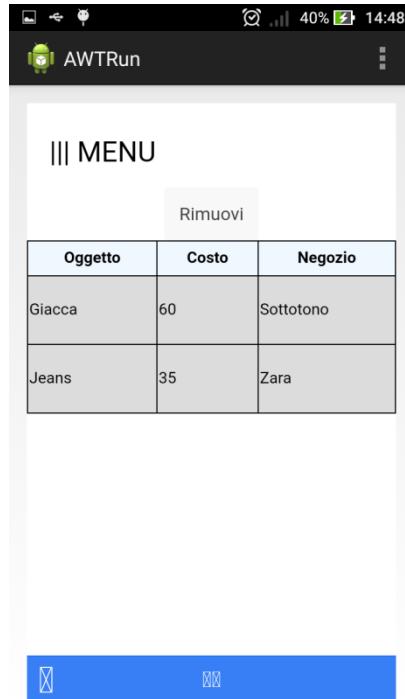


Figura 6. 6 - Esempio di visualizzazione della tabella dopo l'inserimento della riga in Android

#### 6.1.4      *Simulazione Java-Mobile: editing di una cella*

Possiamo accorgerci di aver commesso un errore nella riga appena inserita e dunque provvediamo a modificare la cella corrispondente effettuandone la selezione, in Java AWT con un doppio click mentre sull'app Android con un "press", in modo tale da rendere la cella editabile. Per esempio modifichiamo il valore del costo della seconda riga.

Quello che succede è che dal lato JS viene chiamato il metodo `app.setCellSelection(id, editingRow, editingColumn)` quando si scatena l'evento "press" sulla cella passando come id la stringa "Tabella0", `editingRow` l'indice della riga "1" e `editingColumn` l'indice della colonna "1". Tali informazioni sono utili al metodo per permettere la selezione della cella ed impostare la proprietà `contentEditable` a "true" per permettere la modifica della cella, mostrando sul dispositivo mobile la tastiera.

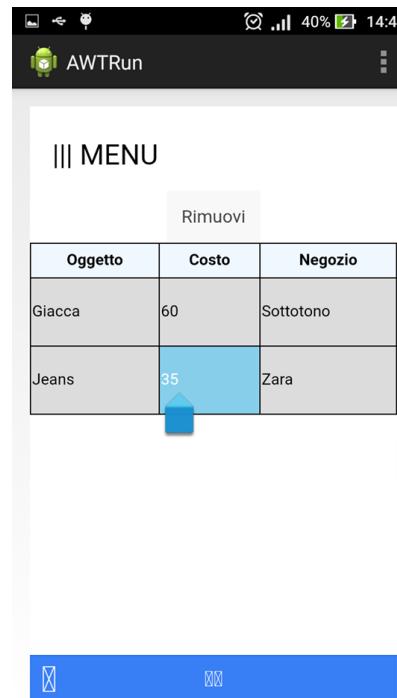


Figura 6. 7 - Esempio di una cella selezionata per l'editing in Android

## CAPITOLO 6 Caso di studio

Jeans	35	Zara	<b>Aggiungi</b>	<b>Rimuovi</b>
Oggetto		Costo		Negozi
Giacca	60		Sottotono	
Jeans	35		Zara	

Figura 6. 8 - Esempio di una cella selezionata per l'editing in Java

A questo punto effettuiamo la modifica della cella inserendo il nuovo valore "40".

Ad ogni rilascio di un carattere della tastiera Android si scatena l'evento "keyup" registrato sulla cella che si sta editando in modo tale invocare la funzione app.insertTable(id,value), con *id* impostato ad "Tabella0" e *value* a seconda del contenuto della cella dopo il rilascio del tasto della tastiera Android (dopo aver inserito l'ultimo risulta essere uguale ad "40").

Questa funzione andrà a richiamare la funzione

*Cordova.exec() awtPlugin.insertTable(id , row, column, value)*, con parametri "Tabella0", "1", "1" e "35" rispettivamente in modo tale da comunicare con il lato Java per impostare la modifica anche al modello di dati della tabella.

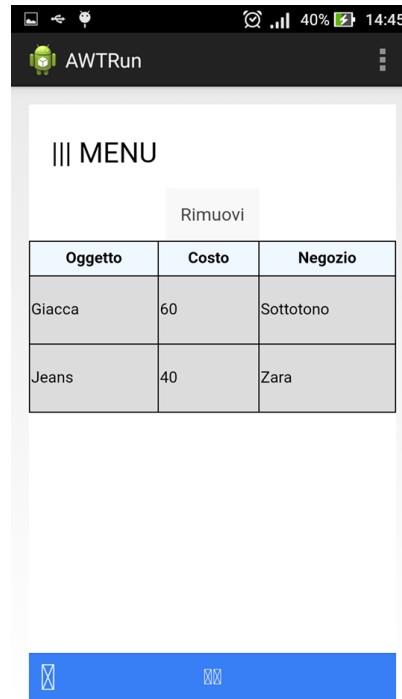


Figura 6. 9 - Esempio di visualizzazione della tabella dopo che la cella è stata modificata in Android

Jeans	35	Zara	<b>Aggiungi</b>	<b>Rimuovi</b>
Oggetto		Costo		Negozi
Giacca	60		Sottotono	
Jeans	40		Zara	

Figura 6. 10 - Esempio di visualizzazione della tabella dopo che la cella è stata modificata in Java

### 6.1.5 Simulazione Java-Mobile : rimozione di una riga

Nel voler rimuovere, ad esempio, la prima riga della tabella bisogna selezionarla in Java AWT con un click su un qualsiasi cella della riga mentre effettuiamo un tap sull'app mobile analogamente.

## CAPITOLO 6 Caso di studio

Jeans	35	Zara
Oggetto	Costo	Negozio
Giacca	60	Sottotono
Jeans	40	Zara

Figura 6. 11 - Esempio di selezione della riga in Java

Quello che succede è che dal lato JS viene chiamato il metodo `app.setRowSelection(id, rowOld, rowNew)` quando si scatena l'evento “*tap*” su un qualsiasi cella passando come id la stringa “*Tabella0*”, *rowOld* l'indice della riga precedentemente selezionata , nel nostro caso nessuna, “-1” per default e *rowNew* l'indice della riga da selezionare “0”. Tali informazioni sono utili al metodo per permettere la selezione della riga, così come spiegato nel paragrafo 5.5.2.



Figura 6. 12 - Esempio di selezione della riga in Android

Una volta effettuata la selezione della riga clicchiamo/tap sul pulsante “*Rimuovi*”.

Jeans	35	Zara
Oggetto	Costo	Negozio
Giacca	60	Sottotono
Jeans	40	Zara

Figura 6. 13 - Esempio di click sul pulsante rimuovi dopo aver selezionato la riga in Java

Quello che accade è che sul bottone è stato registrato un actionlistener la cui azione è quella di rimuovere dal modello di dati la riga selezionata:

```
int[] rows = tabella.getSelectedRows();
for(int i: rows){
    model.removeRow(i);
}
```

## CAPITOLO 6 Caso di studio

Tutto ciò comporta l'invocazione del metodo di pTable *tableChanged()*, il quale invoca la funzione *tableRowsRemoved()* che si occupa di recuperare la riga rimossa dal modello di dati ed invoca la funzione *sendjavascript()* richiamando *app.removeRow(id, row)*, dove i parametri hanno valore “*Tabella0*”, “0” rispettivamente, passando le informazioni sulla riga da rimuovere.

Dunque la funzione *app.removeRow(id, row)* si occuperà di rimuovere dalla tabella HTML la riga selezionata e rimuovere gli eventi ad essa associati. Ecco come si presenta la tabella

aggiornata:



Figura 6. 14 - Esempio di tap sul pulsante rimuovi dopo aver selezionato la riga in Android



Figura 6. 15 - Visualizzazione della tabella dopo aver rimossa la riga in Java

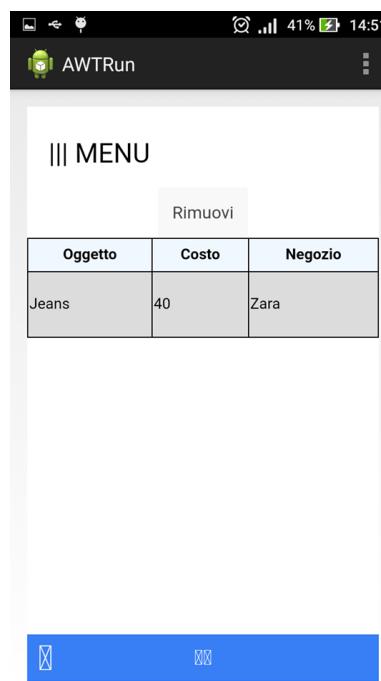


Figura 6. 16 - Visualizzazione della tabella dopo aver rimossa la riga in Android

## 6.2 L'Impaginazione di una tabella di grandi dimensioni

Vediamo adesso un caso in cui la tabella non rientri nello spazio di visualizzazione del device mobile, in accordo con quello spiegato nel paragrafo 5.3.4.

Effettuiamo una modifica dell'esempio precedente *ListShopExtended* dove alle colonne "Oggetto", "Costo", "Negozio" aggiungiamo due colonne, "Percentuale di Sconto", "Data" e "Luogo"; dopodiché popoliamo la tabella, ovvero:

```

1. private String[] columnName = {
2.     "Oggetto", "Costo", "Negozio", "Percentuale di Sconto",
3.     "Data", "Luogo"
4. };
5. private Object[][] data = {
6.     {"Giacca", "60", "Sottotonno", "Nessuna",
7.      "14/06/2017", "Battipaglia"},
8.     {"Maglione", "20", "Belli&Comodi", "10%",
9.      "22/06/2017", "Nocera"},
10.    {"Scarpe", "100", "Calzature Giordano", "20%",
11.      "28/06/2017", "Nemoli"},
12.};

```

Il resto del codice Java rimane invariato. L'interfaccia grafica mostrata sul dispositivo mobile generata è la seguente:



**Figura 6. 17 - L'interfaccia grafica ListShopExtended su Android**

Come si può notare la tabella non viene visualizzata interamente all'interno del frame, per cui viene mostrato una barra di scorrimento che permette di traslarla

## CAPITOLO 6 Caso di studio

orizzontalmente permettendo all'utente la visualizzazione completa di tutti i suoi contenuti.

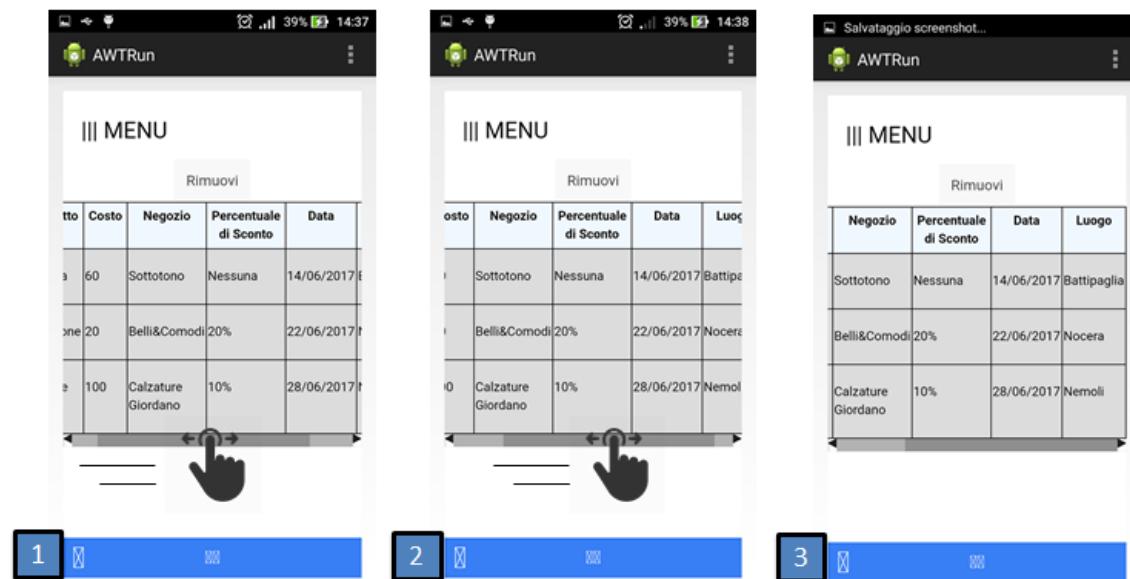


Figura 6. 18 - Illustrazione dello scrolling della tabella su Android

## 7. Conclusioni

---

Ci sono vantaggi e svantaggi legati alla miniaturizzare di un software per le aziende, le quali sentono sempre più il bisogno di espandersi realizzando i loro applicativi anche in versione mobile. Lavorando in ambiente nativo si riescono ad avere applicazioni performanti e pieno accesso alle funzionalità del device, ma ciò richiede un'ottima conoscenza del linguaggio di programmazione dell'ambiente e, soprattutto, costi e tempi di sviluppo notevoli. Invece, se si sceglie di sviluppare l'applicazione utilizzando tecnologie Web e framework si riducono costi e tempi di sviluppo ed inoltre si ottiene un'applicazione cross-platform, ma le prestazioni e l'accesso alle funzionalità del dispositivo sono ridotte considerevolmente.

La soluzione adottata per effettuare il processo di miniaturizzazione arriva a definire una conclusione efficace: utilizzando un **plug-in** e un **framework**, è stato possibile realizzare una **web app** visibile su ogni cellulare con ottimi risultati a livello di performance rispetto alle altre applicazioni, limitandone tempi e costi di sviluppo.

Il lavoro di tesi svolto è stato quello di sviluppare un modulo per estendere il progetto di miniaturizzazione, in particolare la gestione delle tabelle. È stato realizzato un algoritmo ricorsivo che permette di effettuare il parsing di ogni componente della tabella (intestazione, righe e celle) presente nell'applicazione Java. Inoltre sono state definite delle direttive per delineare un'interfaccia user-friendly ed è stata implementata la logica di interazione tra l'utente e la tabella sfruttando il plugin di comunicazione, con il fine di migliorare l'user-experience e aumentare la compatibilità tra l'applicazione desktop e l'applicazione mobile. Gli esempi e i casi di studio analizzati hanno mostrato la validità del processo e la capacità di utilizzare la UI sul mobile.

I possibili sviluppi futuri riguardano:

- **Rendere automatico il processo di miniaturizzazione.**

Esso richiede che alcuni passaggi siano eseguiti manualmente, ma nonostante tutto si è comunque delineato in tutte le sue parti dando indicazione di tutte le attività di pianificazione per la migrazione e delle linee guida.

- **Estensione del motore di generazione del layout.**

Una limitazione del progetto riguarda l'impaginazione del layout in quanto l'aggiunta di componenti di grandi dimensioni nella costruzione dell'interfaccia grafica non riescono ad essere visualizzate all'interno del

frame. Una possibile soluzione a questo problema è quello di estendere il frame qualora un componente copra un'aria più grande rispetto ad esso e segnalarlo all'utente.

- **Estensione della libreria pAWT.**

Aggiungere le componenti maggiormente utilizzate nelle applicazioni Java based in modo tale da integrare nella libreria tutte le classi presenti nel pacchetto *AWT* di *Java*. In particolare si possono trattare negli sviluppi futuri la componente *Dialog* e la possibilità di poter gestire più interfacce contemporaneamente.

## Bibliografia

- [1] SoloTablet. Cara app ma quanto mi costi! In azienda e come sviluppatore – <http://www.solotablet.it/tablet-impresa/ambiti-di-applicazione/cara-app-ma-quanto-mi-costi-in-azienda-e-come-sviluppatore>
- [2] Wikipedia. Porting – Wikipedia, l'enciclopedia libera. <https://it.wikipedia.org/wiki/Porting>, 2017.
- [3] Wikipedia. Smartphone – Wikipedia, l'enciclopedia libera. <https://it.wikipedia.org/wiki/Smartphone>, 2017.
- [4] Oracle. Component. – <https://docs.oracle.com/javase/7/docs/api/java.awt/Component.html>.
- [5] Oracle. Container. – <https://docs.oracle.com/javase/7/docs/api/java.awt/Container.html>.
- [6] Oracle. How To Use Tables – <https://docs.oracle.com/javase/tutorial/uiswing/components/table.html>
- [7] Oracle. JTable – <https://docs.oracle.com/javase/7/docs/api/javax/swing/JTable.html>
- [8] Mary Ercolini. Introduzione alle applicazioni web – [http://www.cs.unipr.it/Informatica/Corsi/2003-04/ICT\\_Azienda\\_D05\\_ApplicazioniWeb.pdf](http://www.cs.unipr.it/Informatica/Corsi/2003-04/ICT_Azienda_D05_ApplicazioniWeb.pdf)
- [9] Rhubbit. Sviluppo app native, ibride e web – <https://www.rhubbit.it/app-native-ibride-web/>
- [10] Luigi Nittoli. Miniaturizzazione di applicazioni Java in Android, 2015.
- [11] Luigi Marano. Miniaturizzazione dell'interfaccia utente da applicazioni desktop a mobile, 2016
- [12] Alfredo Fiorillo. Miniaturizzazione dell'interfaccia utente da applicazioni desktop a mobile, 2017

## Documentazione online

- [1] Android Developer Official Site – <http://developer.android.com/index.html>
- [2] Android Developer Blog – <http://android-developers.blogspot.it>
- [3] PhoneGap Developer Portal – <http://phonegap.com/developer>
- [4] w3schools – <http://www.w3schools.com/>
- [5] Oracle Java SE Documentation – <https://docs.oracle.com/javase/7/docs/>