



Università degli Studi dell'Aquila
Facoltà di Ingegneria

Tesi di Laurea Specialistica in Ingegneria Informatica e Automatica

Progettazione e sviluppo di un sistema mobile/web dedicato ad utenti con disabilità cognitive

Relatore:

Prof. Luigi Pomante

Laureando:

Luca Finocchio

Correlatore:

Ing. Francesco Tarquini

Anno Accademico 2013-2014

Alla mia famiglia

Indice

1	Introduzione	7
2	Progetto Casa Più	10
2.1	Il progetto Casa+	10
2.2	La tecnologia al servizio di Casa+	11
2.3	Tecnologie e servizi	13
2.4	Applicazioni Mobile	16
3	Caratteristiche principali di iOS	17
3.1	Struttura Architetturale	18
3.2	Livelli Architetturali	19
3.2.1	CoreOS	19
3.2.2	Core Services	22
3.2.3	Media	24
3.2.4	Cocoa Touch	26
4	Sviluppo di applicazioni Cocoa in iOS: principali meccanismi	28
4.1	Design Patterns	29
4.1.1	Model-View-Controller Design Pattern	30
4.1.2	Processo di istanziazione	32
4.1.2.1	Singleton Design Pattern: l'oggetto UIApplication . .	32
4.1.3	Composizione della Struttura	32
4.1.3.1	Adapter Design Pattern: i Protocolli	32

4.1.3.2	Composite Design Pattern: UIView	32
4.1.3.3	Decorator Design Pattern: le Categorie	33
4.1.4	Modello di Comportamento	34
4.1.4.1	Chain of Responsibility Design Pattern: UIResponder	34
4.1.4.2	Command Design Pattern: Target-Action	34
4.1.4.3	Mediator Design Pattern: UIViewController	36
4.1.4.4	Observer Design Pattern: le Notifiche	37
4.1.5	Delegate Design Pattern	37
4.2	Gestione della Memoria	39
4.2.1	Politiche di gestione per gli oggetti in memoria	39
4.2.1.1	MRR	41
4.2.1.2	ARC	41
4.3	Ambiente di sviluppo	42
4.3.1	Xcode	42
4.3.1.1	Storyboard	44
4.3.1.2	Interface Builder	45
5	Content Management System: Joomla!	47
5.1	Classificazione dei CMS	49
5.2	Joomla!	50
5.2.1	Struttura	50
5.2.2	Caratteristiche	51
5.2.3	Estensioni	52
5.3	I Componenti di Joomla!	54
5.3.1	MVC in Joomla!	54
5.3.2	Breve panoramica sulle funzionalit di Joomla!	56
5.3.2.1	Gestione delle richieste	56
5.3.2.2	Request	56
5.3.2.3	Sessioni	57
5.3.2.4	Factory Method	57
5.3.2.5	Librerie	58
5.3.2.6	Costanti predefinite	58
5.3.3	Il Database	58

5.3.3.1	Query	59
5.3.3.2	Processare i risultati	60
5.3.3.3	JTable	61
5.3.4	Design dei componenti	61
5.3.4.1	Postazione di lavoro	61
5.3.4.2	Componente di base	62
5.3.4.3	Costruzione di un Model	64
5.3.4.4	Costruzione di una View	66
5.3.4.5	Costruzione di un Controller	68
5.3.4.6	Internazionalizzazione	69
6	Progettazione del sistema	71
6.1	Introduzione	71
6.2	Applicazioni simili già esistenti sul mercato	72
6.3	Specifiche di progetto	72
6.3.1	Requisiti funzionali	73
6.3.2	Requisiti non funzionali	74
6.3.3	Requisiti informativi	75
6.4	Modellazione UML	76
6.4.1	Casi d'uso	77
6.4.2	Struttura Architetturale	86
7	Sviluppo del sistema	89
7.1	Mobile Application	89
7.1.1	Diagramma delle classi	90
7.1.1.1	Model	90
7.1.1.2	ViewControllers	90
7.1.2	Principali aspetti implementativi	94
7.1.2.1	Principali classi dell'applicazione	94
7.1.2.2	CoreLocation	108
7.1.2.3	Storyboard	113
7.1.2.4	PushNotifications	113
7.1.2.5	Preferenze	122

7.2	Server	125
7.2.1	Manifest file, installazione e URL base	125
7.2.2	Schema del DB	131
7.2.3	I Controllers	132
7.2.4	Le Views	136
7.2.5	I Models	138
7.2.6	La Table	140
7.2.7	Internazionalizzazione	141
7.2.8	Layout	142
7.3	Push Notifications Server	144
8	Conclusioni e sviluppi futuri	150
	Bibliografia	155

Capitolo **1**

Introduzione

In questi ultimi anni, gli smartphone, ovvero quei dispositivi che offrono oltre alle caratteristiche solite di un cellulare anche altre più complesse, simili a quelle di un computer, stanno conquistando quote sempre maggiori del mercato della telefonia mobile, aumentando di anno in anno. A questo bisogna aggiungere l'arrivo, in tempi ancora più recenti, di tablet computer che si vanno a collocare, almeno per alcuni utilizzi, nella fascia di mezzo tra gli smartphone ed i computer portatili, fondendo le caratteristiche di entrambi in un unico apparecchio più completo di uno smartphone e più maneggevole ed intuitivo di un laptop. Queste nuove tecnologie fanno sì che la ricerca di nuovi software sia sempre più rivolta verso il mondo dei dispositivi mobili. Tutti questi device di ultima generazione hanno un sistema operativo che spesso deriva da quelli utilizzati per i computer portatili, ma privato di tutte le funzionalità superflue, per renderlo più veloce, leggero ed intuitivo. Inoltre, la maggior parte di questi offre una connessione ad internet, un modulo GPS e permettono di installarvi migliaia di applicazioni diverse (utili e meno utili) sviluppate dal produttore stesso dello smartphone o da terze parti e che possono essere scaricate gratuitamente o pagando una piccola cifra.

Casa Più è un importante progetto, proposto da A.I.P.D.¹ sezione di L'Aquila, concepito ed organizzato al fine di verificare le reali possibilità e capacità di vivere in autonomia delle persone con disabilità cognitiva. Tale progetto prevede l'utilizzo

¹A.I.P.D. (Associazione Italiana Persone Down) - <http://www.aipd.it/>

di nuove soluzioni tecnologiche al fine di sviluppare una certa indipendenza nella gestione e nello svolgimento delle azioni quotidiane.

L'idea alla base di questo progetto di tesi è stata in primo luogo quella di creare un'applicazione per dispositivi mobili da inserire nel contesto Casa+ durante lo svolgimento di un'attività quotidiana quale può essere il fare la spesa, consultare l'ora o preparare una ricetta. Tra le tante opportunità disponibili in questo campo, si è scelto di realizzare il progetto in ambiente Apple poiché per Android, che rappresente l'altra grossa fetta del mercato mobile, esiste già un applicazione analoga. È stato preso in considerazione anche la già grande attenzione che i dispositivi quali iPhone e iPad prestano per utenti con disabilità cognitive. Inoltre Apple permette, oltre la distribuzione diretta del software agli utenti finali, la distribuzione delle applicazioni direttamente, attraverso l'App Store, senza intermediari. In quest'ottica, l'applicazione sviluppata potrà essere anche venduta ad altri utenti i cui ricavati saranno totalmente a supporto dell'A.I.P.D. sezione di L'Aquila.

Le caratteristiche uniche dell'applicazione, vista la particolare natura degli utilizzatori, rendono fondamentale un confronto con figure professionali in ambito educativo e con gli utenti stessi, che offrono preziose indicazioni per la scelta dei servizi da rendere disponibile in mobilità.

La creazione di un'applicazione che combinasse tutte le caratteristiche precedentemente descritte è stata resa possibile grazie alle tecnologie messe a disposizione dall'ambiente di sviluppo Xcode, creato dalla stessa Apple, insieme a framework esterni contenenti, ad esempio, funzioni specifiche per la sintesi vocale o per la gestione delle coordinate dell'utente. La componente software installata sui dispositivi lavora in congiunzione con il server Casa+ già utilizzato in ambiente desktop per lo svolgimento dell'attività in esame. Tale server, però, è stato oggetto di una riprogettazione, in modo da renderlo maggiormente estendibile, modulare e per facilitare il più possibile future installazioni su altre piattaforme. Questo risultato è stato ottenuto grazie all'utilizzo di un CMS, in particolare, tra tutti quelli disponibili, la scelta è ricaduta su Joomla!.

Si arriva dunque a questo documento di tesi, utile per comprendere meglio tutti i diversi passaggi che si sono susseguiti nei mesi di lavoro, nelle cui pagine vengono presentate le tecnologie proprie della piattaforma di sviluppo scelta. Sarà quindi

presentato il progetto, delineandone i requisiti, la struttura, le metodologie adottate per portarlo a compimento ed il suo funzionamento. Vengono qui elencate anche alcune problematiche o dubbi progettuali che è stato necessario risolvere per poter implementare al meglio il software. Nella parte relativa ai principali aspetti implementativi, vengono mostrati gli strumenti utilizzati e motivato brevemente il perchè della loro scelta per poi entrare nel dettaglio dell'implementazione facendo anche osservare, quando necessario, parti di codice relative alla parte prettamente applicativa e alla parte server. Il documento termina con un resoconto del lavoro svolto per arrivare alle conclusioni e ai possibili sviluppi futuri dell'applicazione o di parti di essa.

Capitolo 2

Progetto Casa Più

“*Casa Più*”: è una casa in più rispetto a quella familiare, con più relazioni, più strumenti, più opportunità per una vita quotidiana più indipendente.

Casa Più è un importante progetto, proposto da A.I.P.D. (Associazione Italiana Persone Down) sezione di L’Aquila, concepito ed organizzato al fine di verificare le reali possibilità e capacità di vivere in autonomia delle persone con disabilità cognitiva. Tale progetto prevede l’utilizzo di nuove soluzioni tecnologiche al fine di sviluppare una certa indipendenza nella gestione e nello svolgimento delle azioni quotidiane.

2.1 Il progetto Casa+

Casa Più (di seguito Casa+) è un progetto di residenzialità breve che si svolge in un appartamento provvisto di strumenti tecnologici rivolti a giovani con disabilità cognitive del territorio Aquilano. Questo progetto permette a coloro che vi aderiscono di essere accompagnati, da educatori, in un cammino comune di inserimento sociale e di crescita delle proprie autonomie, nella capacità di gestione di una casa, nell’organizzazione del tempo libero e dei propri impegni quotidiani. Tale opportunità aiuta ad imparare a vivere senza la costante presenza dei genitori e prepara così il proprio futuro di adulti, ma è anche un’opportunità per i genitori per vedere

il proprio figlio “*sotto una nuova luce*” e avere l’occasione concreta per iniziare a progettare un differente futuro per il proprio figlio.

Prepararsi al cambiamento, prevenire quindi, cercando di anticipare e di gestire prevedibili situazioni di difficoltà o di emergenza:

- per comprendere cosa si può fare da soli e in che cosa occorre essere aiutati;
- per imparare a condividere una casa con altre persone;
- per scoprire il piacere di saper fare e di sentirsi adulti.

Il progetto Casa+ si pone, quindi, come obiettivo quello di:

- **sperimentare** un modello di residenzialità con componenti domotiche per aiutare la sicurezza dentro e fuori lo spazio abitativo e per supportare la gestione della vita domestica;
- **individuare** una possibile risposta alla domanda “Dopo di noi?”, da parte dei genitori e parenti, sulla base delle considerazioni effettuate durante lo svolgimento del progetto;
- **aumentare la consapevolezza** nelle famiglie delle capacità dei propri figli e aiutare i processi di allontanamento;
- **valorizzare le potenzialità** delle persone con disabilità cognitive andando a ridurre per gradi l’assistenza da parte degli educatori;
- **arricchire** la formazione degli educatori con nuove competenze di tipo relazionale e tecnologico.

2.2 La tecnologia al servizio di Casa+

Casa+ mediante l’utilizzo di supporti tecnologici intende “accrescere la consapevolezza nei propri mezzi dell’individuo” affidando all’abitazione il compito di sviluppare le capacità organizzative della persona, ipotizzando una risposta attiva alla segnalazione di anomalie, allarmi o ritardi nel compimento di attività prefissate. In altre parole sono inseriti dei nuovi interlocutori tra la casa e la persona che permetto

di effettuare delle segnalazioni attraverso l'uso di sistemi acustici, visivo luminosi, visivo per immagini, tattili, ecc.... Per mezzo di tali sistemi di segnalazione si consente alla persona con disabilità di intervenire e rimediare alle proprie carenze, sviluppando un proprio sistema di adattamento agli eventi.

Per aumentare la sicurezza degli utenti dell'abitazione, ogni situazione di allarme è riportata a un centro di controllo e resa nota a persone dedicate alla tutela e all'educazione degli ospiti che occupano l'appartamento. L'utilizzo di tali dati permette inoltre la creazione di una casistica comportamentale utile per sviluppare un preciso metodo educativo. Altro elemento fondamentale è quello concernente la tutela dell'individuo nella gestione della vita quotidiana, che attraverso l'utilizzo di strumenti tecnologici consentirà di garantirne la sicurezza. L'abitazione deve essere un luogo sicuro per l'ospite, tanto da poter salvaguardare la salute e il benessere delle persone con interventi di attivazione/disattivazione automatica degli impianti domestici (idraulico, elettrico e termico) e di sicurezza.

Si può giungere alla definizione che un adeguato supporto tecnologico a ben progettate azioni educative può consentire a persone affette dalla sindrome di Down di sviluppare le proprie attitudini e di acquisire la consapevolezza delle proprie potenzialità. Detto questo, non si vuole dar troppo valore alla tecnologia nell'educazione delle persone alla vita quotidiana ma se essa è accompagnata dall'individuazione di un corretto e adeguato percorso educativo, la tecnologia può divenire uno strumento utile a rafforzare ed erogare con continuità e tempestività l'azione educativa stessa, permettendo il raggiungimento di risultati utili all'acquisizione della piena coscienza delle potenzialità individuali.

Quindi, ogni supporto tecnologico in tale ambiente dovrà essere percepito come un utile ausilio per migliorare la consapevolezza nei propri mezzi e acquisire una sempre maggiore sicurezza, sempre in maniera non invasiva e rispettosa della libertà individuale.

Si potrà concludere che l'impiego di tecnologie di supporto all'azione educativa avrà avuto successo se nel tempo i soggetti interessati potranno progressivamente ridurne l'utilizzo, avendo acquisito una consapevolezza diretta delle proprie potenzialità.

2.3 Tecnologie e servizi

Il progetto Casa+ offre un ambiente domestico con una serie di soluzioni tecnologiche di ausilio alla gestione delle azioni quotidiane e alla sicurezza degli occupanti.

L'abitazione, nel dettaglio, è dotata di una rete di sensori wireless (Wireless Sensor Network WSN) che permette l'interazione tra i vari dispositivi installati e mediante un opportuno protocollo di comunicazione consente la diffusione dei dati e degli eventi.

La scelta di impiegare una WSN è dovuta alle seguenti caratteristiche:

- una WSN è formata da dispositivi di piccola dimensione;
- ogni dispositivo è autonomo;
- ogni dispositivo coopera nell'esecuzione di una qualche applicazione di raccolta di informazioni dall'ambiente;
- grande disponibilità di microprocessori e microcontrollori;
- moduli dei microprocessori e microcontrollori che permettono la realizzazione di sistemi elettronici portatili in grado di acquisire, elaborare e trasmettere segnali di varia natura.

La Figura 2.1 mostra la pianta dell'abitazione dove sono installati i dispositivi che si andranno a descrivere in seguito:

- **sicurezza domestica** - l'abitazione è equipaggiata con una sensore di controllo della presenza di persone nelle diverse stanze. La rilevazione della presenza può avvenire oltre che mediante sensori di presenza anche attraverso l'uso di dispositivi radio indossati dagli occupanti, incorporati all'interno di un orologio da polso. I sensori di presenza sono, inoltre, interconnessi via radio in maniera tale da minimizzare il cablaggio dell'abitazione e di semplificare il dispiegamento del sistema. Le informazioni raccolte dai diversi dispositivi confluiscano in un nodo specializzato, chiamato sink, che le rende disponibili ad un'unità di elaborazione;



Figura 2.1: Abitazione Casa+

- **gestione del tempo** - la gestione del tempo o meglio una ridotta percezione del trascorrere del tempo rappresenta una delle maggiori difficoltà quotidiane per le persone affette da sindrome di Down. Per migliorare la consapevolezza del proprio tempo si offrono degli indicatori di durata che sono di tipo natura visiva, uditiva, tattile o una combinazione di questi stimoli;
- **assistenza alle azioni quotidiane** - mediante l'uso di ausili tecnologici si favorisce laacquisizione di capacità autonome e in particolar modo si vuole facilitare il compiere di azioni quotidiane quali sono la preparazione dei cibi per i pasti, il tenere in ordine labitazione, ecc In tale contesto si sperimenta l'utilizzo di metodologie basate sulla fornitura di indicazioni legate alle diverse azioni da compiere in sequenza con ausili audio/video, al fine di ridurne nel tempo la frequenza di utilizzo;
- **monitoraggio e controllo a distanza** - questo servizio permette di migliorare la sicurezza all'interno dell'abitazione delle persone affette da sindrome di Down, giacchè mediante un servizio di monitoraggio sono rese accessibili,

sfruttando l'infrastruttura del remoto (tramite internet), una parte delle informazioni raccolte nella struttura. Ciò consente, anche, ai familiari dei soggetti interessati di poter controllare ciò che avviene nell'abitazione. È importante rilevare che, per quanto riguarda le famiglie, vale quanto già più volte asserito per i soggetti affetti da sindrome di Down e cioè che anche i familiari dovranno utilizzare questo strumento per convincersi delle effettive potenzialità dei propri congiunti e, nel tempo, imparare a fidarsi di loro, evitando di limitarne le potenzialità per effetto di atteggiamenti di eccessiva protezione;

- **infrastruttura di comunicazione domestica e accesso da/verso l'esterno** - le comunicazioni dati da e verso la struttura abitativa avvengono mediante un accesso alla rete internet con una connessione ADSL. Questo punto è fondamentale per il fluire delle informazioni raccolte dalla rete interna all'abitazione, senza le quali non si potrebbero avere le funzionalità di monitoraggio e controllo a distanza. Si è, inoltre, sviluppata un'interfaccia che offre un accesso a contenuti utili per la gestione delle diverse attività previste (file audio, video, testi, ecc...). Ancora, al fine di migliorare la sicurezza dei soggetti interessati, si è predisposta una soluzione alternativa alla linea ADSL, soprattutto nel caso d'invio di segnali di allarme, predisponendo l'abitazione di un sistema che consente di inviare messaggi attraverso la rete radiomobile pubblica.

Oltre a predisporre di servizi e tecnologie l'abitazione, il progetto Casa+ intende offrire anche assistenza verso gli spostamenti delle persone affette da sindrome di Down permettendone di acquisire, mediante uno strumento educativo, la capacità di gestire i propri spostamenti in modo ordinato ed efficiente che rappresenta un presupposto fondamentale per accedere al mondo del lavoro e per raggiungere un livello superiore di autonomia. In tale ambito si colloca lo sviluppo di un'applicazione di tracking. In particolare, si predispone l'uso di dispositivi di dimensioni ridotte aventi i servizi di rilevamento GPS e di trasmissione di dati a pacchetto, come il GPRS. L'applicazione di tracking prevede la definizione di un'area geografica ritenuta congrua agli spostamenti tipici e/o previsti del soggetto che ha con sé il dispositivo. Nel caso in cui la posizione rilevata del dispositivo sia al di fuori dell'area definita, il dispositivo procederà nel seguente modo:

- segnalando, dapprima, in locale sul dispositivo stesso mediante segnali acustici, visivi o mediante altri mezzi quali la vibrazione;
- segnalando, se i meccanismi di feedback locali sono ignorati, mediante una telefonata o un sms la posizione a una o più persone affinchè ne siano informate e possano intervenire di conseguenza.

Il dispositivo sarà anche in grado attraverso l'uso d'istruzioni vocali di fornire indicazioni al soggetto interessato. Aspetto molto importante in caso di uscita dall'area definita consiste nella continuità di invio della posizione del dispositivo, a una o più persone preventivamente indicate, mediante sms e email. Infine un'altra applicazione connessa al tracking è legato al supporto che si fornisce per l'utilizzo dei mezzi pubblici, indicando in anticipo il percorso che s'intende compiere da parte del soggetto interessato. Avendo nota delle tratte che prevedono l'impiego di un mezzo pubblico e conoscendo le linee, il dispositivo fornisce un'indicazione ad esempio su quale è il numero del bus su cui salire e quale sia la fermata a cui scendere.

2.4 Applicazioni Mobile

Casa+ prevede lo sviluppo di una serie di applicazioni per dispositivi mobile a supporto delle attività quotidiane delle persone affette da sindrome di Down. Tali applicativi devono poter rendere accessibili alcuni dei servizi disponibili sull'interfaccia desktop. La progettazione e lo sviluppo di una delle applicazioni previste dal progetto Casa+ costituisce una parte di questa tesi.

Capitolo 3

Caratteristiche principali di iOS

iOS rappresenta la versione mobile del sistema desktop OS X, utilizzato in tutti i dispositivi mobili Apple (iPhone, iPod touch, iPad) e mette a disposizione le tecnologie necessarie per implementare applicazioni web e/o native, oltre che dare la possibilità agli utenti di usufruire di un'innumerabile quantità di applicazioni gratuite o a pagamento pronte per essere installate.

Giunto alla versione 7.0.6, rappresenta tuttora uno dei sistemi di punta in ambiente mobile hitech. Il suo successo è stato determinato dalla facilità d'uso e semplicità, due dei maggiori punti di forza, che da sempre hanno reso possibile un utilizzo immediato e rapido, anche grazie ad un'ottima fluidità del sistema; inoltre i modi in cui è stata ideata l'interfaccia grafica e si sono gestite le interazioni con i componenti hardware per supportare il touchscreen, hanno aumentato il grado di intuitività nei gesti eseguiti in un utilizzo giornaliero. Tale piattaforma mobile risulta essere molto sicura e robusta, rispecchiando una scelta progettuale, nonché una linea di percorrenza dell'azienda produttrice, ben precisa.

Come già accennato in precedenza, è possibile progettare e sviluppare applicazioni native mediante interfacce, tools e risorse, dove queste ultime sono molto vincolanti e stringenti; l'aspetto più interessante è la possibilità di giungere ad un livello di performance e ottimizzazione formidabile tramite l'interfacciamento con strutture e tecnologie di basso livello, perseguito metodi più rischiosi e difficoltosi, oppure affidarsi a metodi “pre-confezionati” (la norma, nei maggiori ambienti di sviluppo) che risultano sicuramente più rapidi, ma meno personalizzabili. Per

quanto riguarda gli aspetti architetturali si può notare una grande affinità presente fra l’ambiente desktop e quello mobile, mettendo in luce un buon grado di analisi e organizzazione, ma soprattutto percependo la volontà dell’azienda produttrice di procedere, nel corso degli anni, verso un’unificazione dei due ambienti, al fine di pervenire ad una eterogenea gamma di dispositivi, dal punto di vista dell’equipaggiamento hardware, ed obiettivi specifici, nei quali è in esecuzione il medesimo sistema software sovrastante.

3.1 Struttura Architetturale

L’architettura di iOS è simile a quella che si trova alla base di Mac OS X e, come essa, è strutturata in una serie di quattro livelli di astrazione, o layers, ognuno dei quali implementa funzionalità ben specifiche, per rendere semplice la scrittura di applicazioni che funzionano in modo coerente su dispositivi con differenti capacità hardware.

Nel livello più alto, il sistema operativo agisce come intermediario fra l’hardware sottostante e l’interfaccia grafica, mentre le varie applicazioni installate comunicano con i livelli sottostanti attraverso un ben determinato set di interfacce, aumentando considerevolmente la protezione delle applicazioni da eventuali modifiche hardware. Il livello più basso del sistema ospita i servizi e le tecnologie fondamentali da cui tutte le applicazioni dipendono.

La maggior parte delle interfacce di sistema sono rese disponibili in pacchetti speciali chiamati frameworks. Un framework consiste in un directory contenente una libreria dinamica di funzioni e risorse (header files, immagini etc.) a supporto di essa. In aggiunta ai frameworks, Apple rende disponibili alcune tecnologie nella forma di librerie dinamiche in formato standard; molte di esse sono appartenenti al livello più basso del sistema operativo e derivano da tecnologie Open Source, come naturale conseguenza del fatto che iOS, similmente al corrispondente sistema desktop di casa Apple, sia basato sulla piattaforma Unix.

3.2 Livelli Architetturali

Procedendo in un'analisi più accurata circa la composizione e la coesione delle stratificazioni che vanno a formare l'architettura di iOS, si possono individuare quattro differenti livelli di astrazione, come mostrato in Figura 3.1 ognuno dei quali verrà trattato in maniera dettagliata nelle sezioni successive.

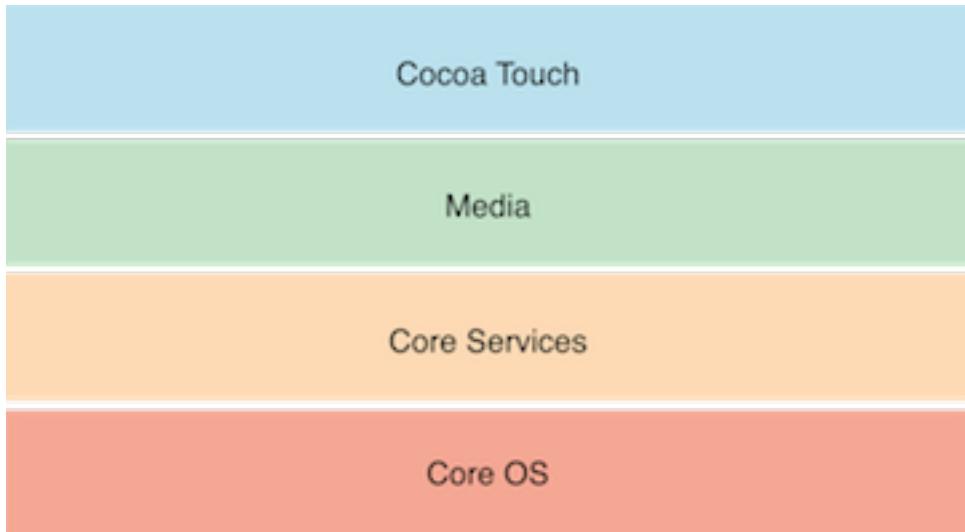


Figura 3.1: Livelli architetturali di iOS

3.2.1 CoreOS

È lo strato che permette di lavorare a diretto contatto con l'hardware sottostante ed è considerato il vero cuore del sistema operativo; infatti in esso sono presenti gli elementi considerati fondamentali, utilizzati poi dalle tecnologie dei livelli sovrastanti.

- **Accelerate Framework** - contiene le interfacce utilizzate per l'esecuzione di calcoli matematici, DSP ed anche per l'elaborazione di numeri molto grandi. Tale strumento ha il vantaggio di essere ottimizzato per tutte le configurazioni hardware presenti in dispositivi basati su iOS.
- **Core Bluetooth Framework** - consente agli sviluppatori di interagire specificamente con accessori Bluetooth a bassa energia (LE).

- **External Accessory Framework** - offre supporto per la comunicazione con la parte hardware di accessori o componenti collegati a dispositivi basati su iOS.
- **Security Framework** - mette a disposizione interfacce specifiche per gestire certificati, chiavi private o pubbliche e la generazione di numeri crittografati pseudo-casuali; tutto ciò in aggiunta alle caratteristiche di sicurezza già presenti, in modo da garantire un livello di sicurezza personalizzato per i dati delle applicazioni sviluppate.
- **Livello di Sistema** - comprende l'ambiente del kernel, i drivers, e le interfacce Unix di basso livello. La parte centrale, fulcro stesso di Mac OS X e iOS, che include il kernel e la base Unix, è noto come Darwin, un sistema operativo open source pubblicato da Apple. Il kernel, basato su Mach, è responsabile di ogni aspetto del sistema operativo. Gestisce il sistema di memoria virtuale, i threads, il file system, la rete e le comunicazioni tra processi. I driver di questo strato forniscono anche l'interfaccia tra l'hardware disponibile e i frameworks di sistema. Per motivi di sicurezza, l'accesso al kernel e ai driver è limitato a un numero limitato di frameworks e applicazioni di sistema.
- **Supporto per architetture a 64-bit** - iOS è stato inizialmente progettato per supportare file binari su dispositivi che utilizzano una architettura a 32-bit. In iOS 7, tuttavia, è stato introdotto il supporto per la compilazione, linking, e debugging di file binari su una architettura a 64-bit. Tutte le librerie di sistema e i framework sono “64-bit ready”, il che significa che possono essere utilizzati in entrambe le applicazioni a 32-bit e 64-bit. Quando queste vengono compilate per il runtime a 64 bit, le applicazioni possono essere eseguite più velocemente a causa della disponibilità di risorse del processore supplementari in modalità a 64 bit.

Per quanto concerne il kernel, il sistema operativo Darwin è costituito dal kernel XNU ; quest'ultimo è costituito da un'architettura a livelli che conta tre componenti principali, come è riportato in Figura 3.2. L'anello più interno del kernel, se così si può definire, si riferisce al livello Mach, derivante dal kernel omonimo alla versione 3.0, sviluppato presso la *Carnegie Mellon University*.

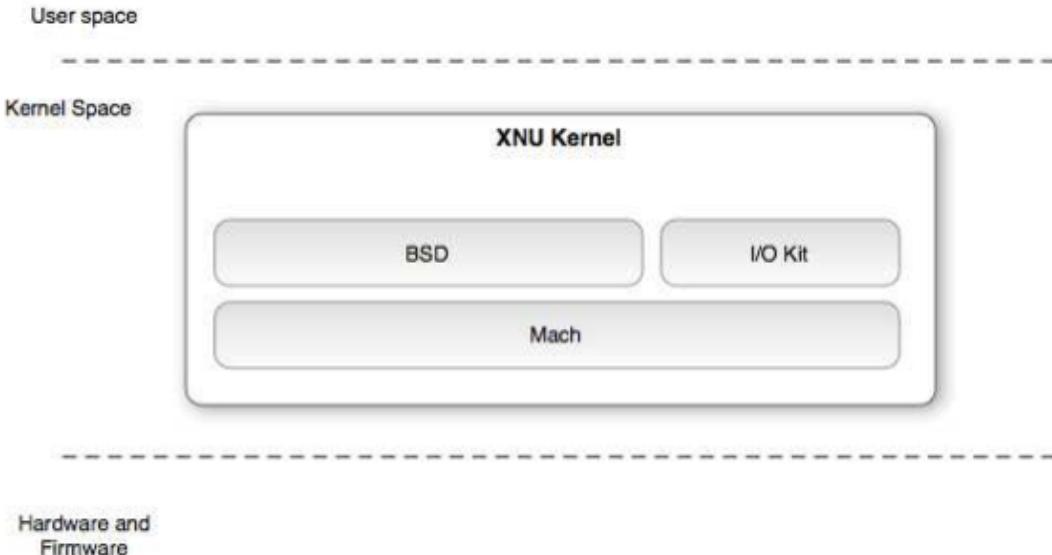


Figura 3.2: Architettura del kernel XNU

Il kernel Mach fu definito come microkernel, pensato come un sottile strato con lo scopo di fornire solamente i servizi fondamentali, come la gestione dei processori, dei threads e la schedulazione dei tasks, per i due componenti sovrastanti che completano l'XNU, rappresentati dal livello BSD e dall'I/O Kit.

La parte BSD si interpone tra il microkernel Mach e le applicazioni utente, implementando molte funzioni chiave del sistema operativo, quali la gestione dei processi, delle chiamate di sistema, del filesystem e del collegamento in rete. Tale layer si riferisce ad una parte del kernel che deriva dal sistema operativo FreeBSD 5, del quale rappresenta una porzione di codice e non un sistema completo in sè.

Il layer chiamato I/O Kit è in realtà un framework Object-Oriented necessario alla scrittura di drivers per i dispositivi ed altre estensioni del kernel; in particolare prevede un'astrazione dei sistemi hardware con classi base predefinite per supportarne diversi tipi, rendendo più semplice l'implementazione di nuovi drivers.

I drivers presenti nel livello di sistema consentono di fornire un'interfaccia interposta fra la specifica parte hardware e i frameworks di sistema anche se, per motivi di sicurezza, l'accesso al kernel e ai suddetti drivers è ristretto ad un set limitato di frameworks ed applicazioni. In fine, le interfacce Unix in questione sono basate

sul linguaggio C e forniscono supporto per il Threading (threads POSIX), la gestione della rete (sockets BSD), l'accesso al filesystem, la gestione dell'alimentazione, informazioni di localizzazione, gestione della memoria e relativa allocazione.

3.2.2 Core Services

Come suggerisce il nome assegnato, questo livello contiene i servizi di sistema fondamentali, utilizzati da tutte le applicazioni, spesso considerate utility. Le tecnologie chiave presenti si possono riassumere nelle aree di interesse che comprendono la programmazione concorrente, il commercio elettronico, la gestione e memorizzazione dei dati all'interno di database e presentazione/manipolazione delle informazioni ricevute o trasmesse.

- **iCloud Storage** - introdotto in iOS 5 consente lo storage *in-the-cloud* accessibile da tutti i dispositivi di uno stesso utente. Ciò vuol dire che un utente può visualizzare o modificare i propri dati senza che il trasferimento degli stessi avvenga in maniera esplicita.
- **Automatic Reference Counting (ARC)** - Introdotto in iOS 5, ARC è una funzionalità a livello di compilazione che semplifica il processo di gestione del ciclo di vita degli oggetti in Objective-C.
- **Grand Central Dispatch (GCD)** - introdotto nella versone 4.0 di iOS, rappresenta una tecnologia molto utilizzata per gestire lesecuzione delle varie attività (tasks) all'interno delle singole applicazioni.
- **In-App Purchase** - implementato tramite lo *Store Kit Framework*, il quale mette a disposizione le infrastrutture necessarie per i processi che coinvolgono le transazioni finanziarie utilizzate all'interno dell'iTunes Store. Esso offre la possibilità di vendere contenuti e servizi all'interno delle singole applicazioni, senza dover ricorrere obbligatoriamente ad una gestione centralizzata dei contenuti commerciali.
- **SQLite** - costituita da una libreria dedicata, permette di incorporare una versione “lite” del database SQL nelle applicazioni, senza bisogno di porre in

esecuzione separatamente un database remoto all'interno di un processo server; in particolare è possibile gestire tabelle e tuple attraverso un database locale contenuto in appositi files. Inoltre, anche se tale libreria è stata ideata per un uso generico, può essere ottimizzata al fine di innalzare il livello di performance.

- **Supporto XML** - tramite il Foundation Framework, mette a disposizione una classe che incapsula il concetto di parser per riuscire a trattare il testo all'interno del linguaggio di markup che rappresenta e descrive l'intera struttura di un documento.

Nelle tecnologie sopra descritte è spesso indispensabile l'utilizzo dei frameworks presenti in questo livello, alcuni di essi riportati di seguito.

- **Foundation Framework** - insieme ad UIKit è uno dei framework essenziali per la piattaforma mobile iOS, mentre tutti gli altri sono secondari e non indispensabili; ne sono un esempio le applicazioni a linea di comando, le quali utilizzano solo tali due frameworks. Le funzioni svolte ed i ruoli assunti dal framework Foundation si possono riassumere nella definizione di un livello base di classi che possono essere utilizzate per ogni tipo di programma. Per quanto riguarda gli obiettivi, esso si preoccupa di definire il comportamento base degli oggetti utilizzati ed introduce per essi delle convenzioni coerenti quali la gestione della memoria, il sistema di notifiche e il supporto alla loro mutevolezza, persistenza e distribuzione. Oltre a ciò, dà supporto per il sistema di localizzazione e fornisce alcune misure per rendere maggiormente portabile il sistema operativo stesso. Ad esso è strettamente legato il Core Foundation Framework, costituito da un insieme di interfacce basate sul linguaggio C, indirizzato alla gestione dei dati e servizi in applicazioni iOS in modo più specifico rispetto al Foundation Framework.
- **Core Location Framework** - permette di accedere alle informazioni di localizzazione e posizionamento all'interno delle applicazioni; in particolare fa uso della tecnologia GPS, o della triangolazione Wi-Fi, per ottenere la longitudine e latitudine correnti.
- **CFNetwork Framework** - basato su socket BSD, è costituito da un insieme interfacce in linguaggio C molto performanti, utilizzando paradigmi orientati

agli oggetti per lavorare con svariati protocolli di rete; la caratteristica di rilievo è rappresentata dalla possibilità di ottenere un controllo dettagliato dei vari stack di protocolli e rendere molto semplice il loro utilizzo grazie alle astrazioni introdotte.

- **Core Data Framework** - costituisce una tecnologia per la gestione del modello dei dati in applicazioni caratterizzate dall'utilizzo del pattern MVC. In linea di principio, è destinato ad un utilizzo nello sviluppo di applicazioni in cui il modello dei dati previsto è altamente strutturato.
- **Core Media Framework** - include in sè gli strumenti e i tipi di comunicazione audio/video di basso livello, utilizzati nel livello architettonico sovrastante, per esempio dal *AVFoundation Framework*.
- **Quick Look Framework** - fornisce un'interfaccia diretta per la visualizzazione di un'anteprima dei files non supportati direttamente; tale strumento risulta utile in applicazioni che scaricano contenuti dalla rete o sono portati a lavorare con files la cui fonte è sconosciuta.
- **Store Kit Framework** - offre supporto nella compravendita di contenuti e servizi che avvengono all'interno delle singole applicazioni; in particolare tale framework si focalizza sull'aspetto finanziario della transazione, assicurando che sia avvenuta correttamente e nel modo più sicuro possibile.

3.2.3 Media

Rappresenta lo strato che contiene tutte le funzionalità e le librerie per la gestione di video e audio. Mediante le tecnologie presenti, si è orientati verso la creazione della migliore esperienza multimediale raggiungibile su un dispositivo mobile. Inoltre, è necessario l'inserimento di una nota per quanto riguarda una tecnologia sviluppata chiamata AirPlay, la quale permette lo streaming audio verso la Apple TV oppure verso altoparlanti AirPlay di terze parti. Il supporto AirPlay è integrato in AVFoundation e CoreAudio Framework; questo implica che qualunque contenuto audio riprodotto usando uno dei due frameworks sopra citati è automaticamente reso idoneo per la distribuzione tramite AirPlay.

- **AVFoundation Framework** - utilizzato per la riproduzione e manipolazione di contenuti audio; con esso è possibile avere un ampio controllo su vari aspetti dei suoni riprodotti, come la possibilità di registrare audio e gestire le informazioni sulle sessioni sonore acquisite.
- **Core Audio** - offre supporto nativo per le operazioni sull'audio; in particolare supporta la manipolazione di audio a qualità stereo, la generazione, registrazione e mix nonché riproduzione dellaudio risultante.
- **Core Graphics, MIDI, Text, Video Frameworks** - come tutti i frameworks presenti in questo livello architettonico, permettono di avere un ampio controllo e possibilità di espressione nelle rispettive aree di interesse.
- **OpenAL (AudioLibrary) Framework** - rappresenta uno standard interpiattaforma per posizionare fonti audio in modo tridimensionale, tenendo in considerazione i parametri di disturbo che potrebbero influire. Viene utilizzato per implementare giochi o altre applicazioni che richiedano un audio posizionale in output, caratterizzate da alte prestazioni e soprattutto con un audio di altissima qualità.
- **OpenGL (Graphical Library) ES Framework** - e una potente libreria grafica, pensata per interfacciarsi direttamente con l'hardware e fornire al singolo programma una serie di primitive, più o meno essenziali, per lo sviluppo di applicazioni nel campo del rendering 2D e 3D. Le primitive considerate comprendono funzioni per la manipolazione dei pixel, per la proiezione di poligoni in 3D e per la gestione del movimento degli stessi, per la rappresentazione di luci etc....
- **Quartz Core Framework** - contiene le interfacce del Core Animation, una tecnologia avanzata per l'animazione e composizione utilizzata come via per il rendering ottimizzato al fine di implementare complesse animazioni ed effetti visivi.

3.2.4 Cocoa Touch

Rappresenta lo strato più vicino all'applicazione utente e i frameworks di questo livello supportano direttamente le applicazioni basate su iOS. Esso si occupa della gestione del touch e multi-touch, interpretando i differenti gesti (gestures) compiuti dall'utente finale mediante i *gesture recognizers*, oggetti collegati alle *view* (a loro volta definite come schermate visibili sul video), utilizzati per rilevare i tipi più comuni di gestures, come lo zoomin o la rotazione di elementi; non appena collegati, si può stabilire quale comportamento associare ad essa.

Oltre alla gestione del touch, molti dei frameworks del livello Cocoa Touch contengono specifiche classi genericamente denominate *View Controller*, per poter visualizzare interfacce standard di sistema. Tali componenti rappresentano particolari tipi di controller molto utilizzati per presentare e gestire un insieme di view; giocano un ruolo importante nella progettazione ed implementazione di applicazioni iOS perché forniscono un'infrastruttura per gestire i contenuti correlati alle view e coordinare la comparsa/scomparsa di queste ultime.

Un'ulteriore caratteristica di questo livello è data dal supporto a due differenti modalità in cui utilizzare le notifiche, le quali danno la possibilità di avvisare l'utente di nuove informazioni mediate un segnale sonoro o visivo. Il primo prende il nome di *Apple Push Notification Center* e utilizza un processo server per generare e distribuire la notifica inviatagli dal client, mentre il secondo, *Local Notification*, completa il meccanismo di notifiche sopra descritto dando la possibilità, in fase di progettazione, di generarle in locale, senza fare affidamento su un server esterno.

Infine, molto importante per gli argomenti correlati alle prestazioni di sistema, è il pieno supporto dato al Multitasking anche ad alto livello architetturale, per poter coordinare ed impostare azioni tipiche del livello applicativo, come la ricezione di notifiche ed il passaggio da uno stato attivo ad uno passivo.

Come in tutti gli altri livelli in cui è strutturato iOS, anche in Cocoa Touch si utilizzano innumerevoli frameworks per giungere agli obiettivi appena descritti, dei quali si riportano i più importanti.

- **GameKit Framework** - fornisce varie funzionalità separate, principalmente implementabili nelle applicazioni di tipo ludico.

- **MapKit Audio** - fornisce un’interfaccia per integrare le mappe direttamente nella finestre e view delle applicazioni, supportando anche l’inserimento di annotazioni, la possibilità di introdurre sovrapposizioni fra differenti tipologie di mappe e la ricerca tramite un’operazione di reversegeolocation al fine di determinare le informazioni per un determinato segnaposto, una volta note le coordinate geografiche.
- **iAd Framework** - consente all’applicazione di ottenere un introito mediante la visualizzazione di annunci pubblicitari nella forma di piccoli banner. Gli annunci sono incorporati all’interno delle view in formato standard e possono essere visualizzate in ogni momento.
- **UIKit Framework** - come già accennato nel livello Core Services, è uno dei due frameworks portanti nella struttura architettonica di iOS; in questo caso, il suo compito principale è mettere a disposizione tutte le classi atte alla costruzione e gestione dell’interfaccia utente nell’applicazione. Inoltre incorpora il supporto per caratteristiche specifiche di alcuni dispositivi, come l’accelerometro, la libreria fotografica dell’utente, informazioni sullo stato della batteria, sui sensori di prossimità etc. . . .

Capitolo **4**

Sviluppo di applicazioni Cocoa in iOS: principali meccanismi

È importante adottare un processo che renda possibile la realizzazione di un prodotto di qualità per rimediare ad errori o introdurre modifiche con relativa facilità e che potrebbe essere riutilizzato nella costruzione di altri sistemi, senza costringere ad una organizzazione dal punto di partenza. Al fine di concludere la fase di progettazione ottenendo un prodotto di qualità, è opportuna non solo la conoscenza, ma bensì la padronanza di concetti strutturali e stilistici, comuni a tutte le applicazioni Cocoa indirizzate alla piattaforma mobile iOS, che comprendono:

- i design patterns e come essi siano stati adattati rispetto alle forme standard;
- la gestione della memoria all'interno del sistema e delle applicazioni;
- i tools e le metodologie di sviluppo.

Per quanto riguarda la versione del sistema operativo, si è scelto di riferirsi ad iOS 7.0.6, corrispondente all'ultimo aggiornamento della piattaforma mobile, rilasciato a Febbraio 2014; ai ni della trattazione corrente, è sufficiente prendere in considerazione versioni non inferiori alla 7.0.

4.1 Design Patterns

Un design pattern descrive una soluzione generale a un problema di progettazione ricorrente, gli attribuisce un nome, astrae e identifica gli aspetti principali della struttura utilizzata per la soluzione del problema, identifica le classi e le istanze partecipanti e la distribuzione delle responsabilità, descrive quando e come può essere applicato. In breve, definisce un problema, i contesti tipici in cui si trova e la soluzione ottimale allo stato dell'arte.¹

Nelle fasi progettuali, dopo un attenta analisi delle problematiche alle quali è soggetta l'applicazione target, una buona conoscenza dei principali design patterns è fondamentale per la scrittura di codice finalizzato alla costituzione di componenti software il più possibile estendibili e riusabili, rendendo eventuali modifiche di facile attuazione, se futuri requisiti o esigenze lo richiedessero. Le applicazioni basate sull'utilizzo dei design patterns sono generalmente più eleganti ed efficienti, traducendosi molto spesso in una diminuzione delle linee di codice che portano al medesimo obiettivo. L'importanza che assumono i design patterns nell'ambiente di programmazione Cocoa è nettamente amplificata, data la loro pervasività nella maggior parte delle architetture e meccanismi in entrambe le piattaforme software (OS X, iOS) nelle quali il sistema di frameworks dedicati fornisce un'infrastruttura cruciale per l'applicazione e in molti casi rappresenta l'unica via di accesso alle risorse sottostanti.

Dato che ai pattern non è attribuito un significato assoluto, è possibile applicarli a casi di studio concreti con un certo grado di essibilità, come è nel caso di applicazioni mobile; questo è l'aspetto che ha permesso un approccio personalizzato, caratterizzando fortemente le applicazioni iOS da tutte le altre concorrenti.

Nell'ambiente Cocoa le classi associate ad un particolare framework e gli stessi linguaggi di programmazione o di runtime, implementano già molti dei design patterns catalogati; in questo modo l'ambiente di sviluppo è in grado di soddisfare gran parte delle esigenze di un normale sviluppatore, utilizzando uno o più di questi adattamenti ad un modello di progettazione. Sono inseriti alcuni tratti e caratteri-

¹Gamma, E., Helm, R., Johnson, R. e Vlissides, J., *Design Patterns - Elementi per il riuso di software ad oggetti*, Pearson Education Italia, 2002. ISBN 887192150X

stiche distintive in tali templates perchè il design che si vuole ottenere è fortemente influenzato da fattori come la capacità dei linguaggi adottati o da strutture architettoniche esistenti, indispensabili per la logica adottata nella piattaforma software. L'implementazione dei design patterns in Cocoa avviene in varie forme; infatti alcuni costituiscono caratteristiche proprie del linguaggio Objective-C, in altri casi l'istanza di un pattern è implementata all'interno di una classe o in un gruppo di classi correlate; altre volte il pattern adattato rappresenta l'architettura stessa di un framework principale.

4.1.1 Model-View-Controller Design Pattern

Il pattern Model-View-Controller (MVC) è molto diffuso nello sviluppo di sistemi software Object-Oriented per modellare e attribuire un ruolo preciso agli elementi correlati alla presentazione dell'interfaccia graca. Non 'e considerato propriamente un design pattern, ma piuttosto un *pattern architettonale*, perchè il suo utilizzo concerne l'organizzazione dell'architettura globale, classificando gli oggetti in base ai ruoli generici che ricoprono in un'applicazione, con l'obiettivo di giungere ad un disaccoppiamento delle varie parti che compongono il sistema. Gli oggetti che costituiscono applicazioni Object-Oriented conformi al pattern MVC presentano interfacce ben definite e risultano maggiormente riusabili, specie al variare dei requisiti applicativi. Nel pattern MVC si prendono in considerazione tre principali classi di oggetti, che si differenziano per il ruolo che ricoprono:

- **Model** - classe di oggetti che rappresenta il modello del dominio e fornisce metodi per potervi accedere; generalmente non dovrebbe essere correlata con problematiche relative alla presentazione dell'interfaccia;
- **View** - classe di oggetti con la funzione di visualizzare l'interfaccia utente, permettendo all'utente di interagire con essa, utilizzando indirettamente i dati dell'applicazione;
- **Controller** - classe di oggetti che agisce da intermediario fra View e Model. Tendenzialmente è responsabile del corretto accesso alla classe di oggetti Model, che è necessario al fine di visualizzarne il contenuto ed eventualmente modificarne alcune parti.

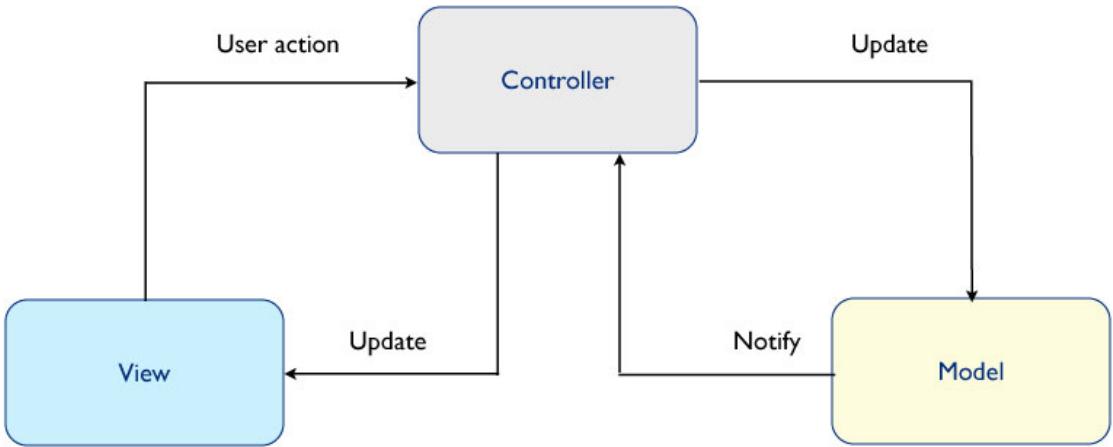


Figura 4.1: Struttura MVC: Soluzione Apple

Nella versione Cocoa, il pattern MVC presenta una differenza sostanziale: la totale separazione fra Model e View, resa possibile attribuendo maggiori responsabilità agli oggetti Controller. L'approccio adottato produce un aumento della complessità della parte relativa al controllo, che ora deve mediare il flusso dei dati fra Model e View in entrambi le direzioni. In iOS gli oggetti Controller sono modellati come oggetti *UIViewController*; per semplificare la gestione del modello dei dati e la coordinazione per la presentazione dell'interfaccia grafica, il framework UIKit fornisce diverse classi pronte per essere utilizzate, che si differenziano a seconda della modalità in cui predispongono la suddivisione dell'interfaccia utente. La classe *UIViewController* è essenziale in ogni applicazione mobile iOS perché in essa sono implementati molti dei metodi base per la gestione dei comportamenti comuni ad ogni applicazione; ad essa spesso è affiancato un delegato personalizzato, reso conforme ad una serie di protocolli specifici, in modo da rinviare la definizione ed implementazione di funzionalità caratteristiche delle classi che estendono e specializzano *UIViewController*. Le due parti restanti del pattern MVC, inerenti rispettivamente al modello dei dati e alla presentazione dell'interfaccia utente, costituiscono i componenti maggiormente riusabili in un'applicazione.

Il diagramma in Figura 4.1 rappresenta sinteticamente la struttura del pattern MVC, nella soluzione adottata in iOS.

4.1.2 Processo di istanziazione

4.1.2.1 Singleton Design Pattern: l'oggetto UIApplication

Tale pattern viene utilizzato per assicurare che una determinata classe sia presente in una sola istanza, e per fornire un accesso globale ad essa. Per garantire l'esistenza di un'unica istanza, la quale è oltretutto facilmente accessibile, la soluzione migliore è fare in modo che sia la stessa classe a tenere traccia di ciò. Utilizzo del Singleton nello sviluppo di software per dispositivi mobile diviene necessario quando si modella, tramite un oggetto, una risorsa fisica per usufruire delle funzionalità messe a disposizione, infatti, parecchie classi del framework Cocoa sono singleton. Queste includono NSFileManager, NSWorkspace, NSApplication, e, in UIKit, UIApplication. Un processo è limitato ad una istanza di queste classi. Quando un client chiede una istanza di una classe, si ottiene una istanza condivisa, che sarà stata creata alla prima richiesta.

4.1.3 Composizione della Struttura

4.1.3.1 Adapter Design Pattern: i Protocolli

Un Protocollo è essenzialmente la dichiarazione di una serie di metodi non associati ad alcuna classe, quindi se lo scopo è quello di permettere la comunicazione fra un oggetto client ed un altro oggetto, la quale risulta difficoltosa per un'incompatibilità a livello di interfacce, si può facilmente definire un protocollo, adottato dalla classe che si vuole adattare. In generale, per poter essere conforme al protocollo assegnato, è necessario implementare i metodi dichiarati necessari, tramite la direttiva @required, mentre si può decidere di lasciarne altri opzionali (@optional).

Utilizzando i protocolli si rende indipendente la dichiarazione di un set di metodi dalla gerarchia di classi.

4.1.3.2 Composite Design Pattern: UIView

Il Composite pattern permette la disposizione degli oggetti, caratterizzati dallo stesso tipo base, in una struttura gerarchica ad albero, dove ogni “nodo padre” può contenere uno o più “nodi figli”; tale relazione si presenta in maniera ricorsiva al-

l'interno della gerarchia. La struttura descritta termina con “nodi foglia”, ovvero con elementi che non contengono a loro volta alcun oggetto, ma che presentano la medesima interfaccia comune a tutti i componenti dell'albero. Nel framework Cocoa Touch si utilizza questo design pattern per modellare la struttura di cui è composta l'interfaccia grafica; in particolare ogni elemento, nodo, è rappresentato dall'oggetto *UIView*, utilizzato per visualizzare il contenuto dell'applicazione sullo schermo, definendo una porzione della nostra totale. Alla base della gerarchia è presente l'oggetto *UIWindow*, che modella l'idea di “contenitore” più esterno, con dimensioni pari a quelle dello schermo considerato.

Nella progettazione dell'oggetto *UIView*, Apple ha introdotto alcune specificazioni; ad eccezione fatta per l'oggetto *UIWindow*, ad ogni istanza della classe *UIView* è attribuita un'unica *superview* e zero o più *subview*, come mostrato in Figura 4.2 ad ogni view è associato un array di subview, dove l'ordinamento riflette la disposizione visibile sullo schermo, da quella disposta sullo sfondo, (posizione 0), a quella presente in primo piano. La composizione costituita dagli oggetti *UIView* gioca un ruolo fondamentale sia per la visualizzazione del contenuto, che per la risposta ad eventi. Ogni volta che è richiesta la visualizzazione di una porzione della finestra, il messaggio è inviato prima alla *superview* e successivamente alle *subviews*.

4.1.3.3 Decorator Design Pattern: le Categorie

Il pattern Decorator permette sostanzialmente di incorporare l'insieme dei comportamenti designati, senza dover modificare o adattare la struttura delle classi progettate in precedenza. Nelle piattaforme Apple è implementata una variante di tale pattern strutturale, riassunta nel concetto di *Categoria*. Una *Categoria* è una caratteristica propria del linguaggio Objective-C, che permette di aggiungere funzionalità ad una determinata classe, in termini di definizione delle interfacce dei metodi e relativa implementazione, senza ricorrere ad una sottoclasse.

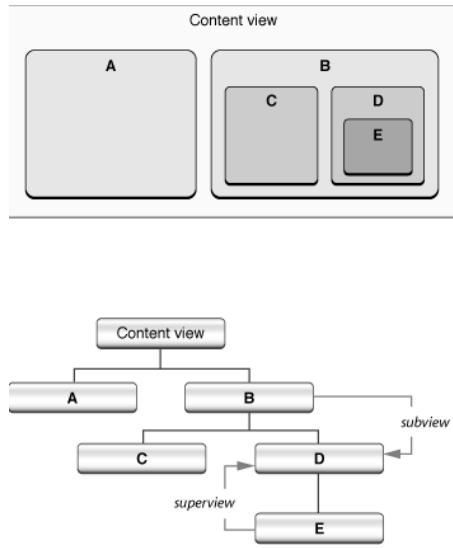


Figura 4.2: View Layers

4.1.4 Modello di Comportamento

4.1.4.1 Chain of Responsibility Design Pattern: UIResponder

L’idea principale alla base del Chain of Responsibility design pattern è di creare una struttura a catena, dove ogni oggetto detiene il riferimento a quello successivo, e tutti implementano il medesimo metodo per gestire in modi differenti una determinata richiesta, eseguita partendo dal primo oggetto della catena. Se uno di questi non è in grado di gestire la richiesta, la passerà al “ricevitore successivo”. Figura 4.3

4.1.4.2 Command Design Pattern: Target-Action

Il pattern Command disaccoppia un’azione, modellata come oggetto, e il ricevente che la esegue e viene implementato in Cocoa tramite il meccanismo Target-Action. Tale meccanismo risulta utile per la comunicazione fra uno o più elementi di controllo ed un qualunque altro oggetto, in modo tale che i primi mantengano le informazioni necessarie per inviare un messaggio al secondo, al verificarsi di un generico evento. L’evento che innesca lazione può essere di qualunque tipo, anche se il meccanismo targetaction è molto spesso usato in relazione ad oggetti di controllo come buttoni o sliders. Per quanto riguarda loggetto di controllo, il framework UIKit ha dichia-

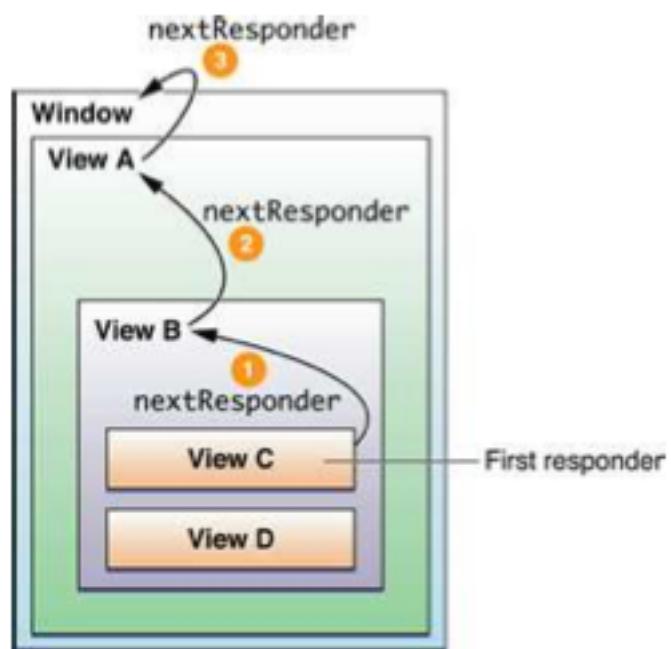


Figura 4.3: Responder Chain

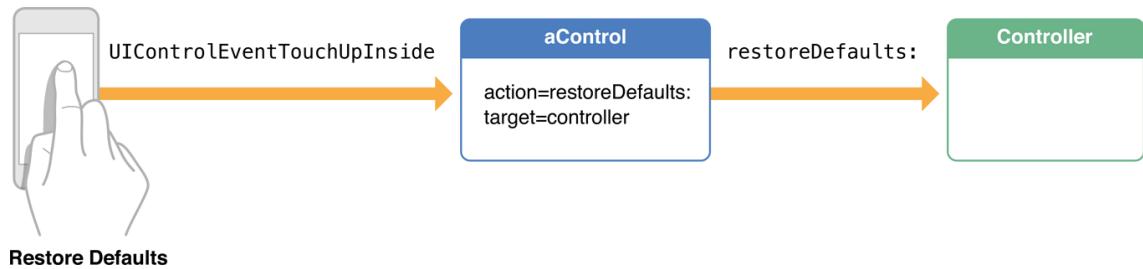


Figura 4.4: Esempio di Target-Action

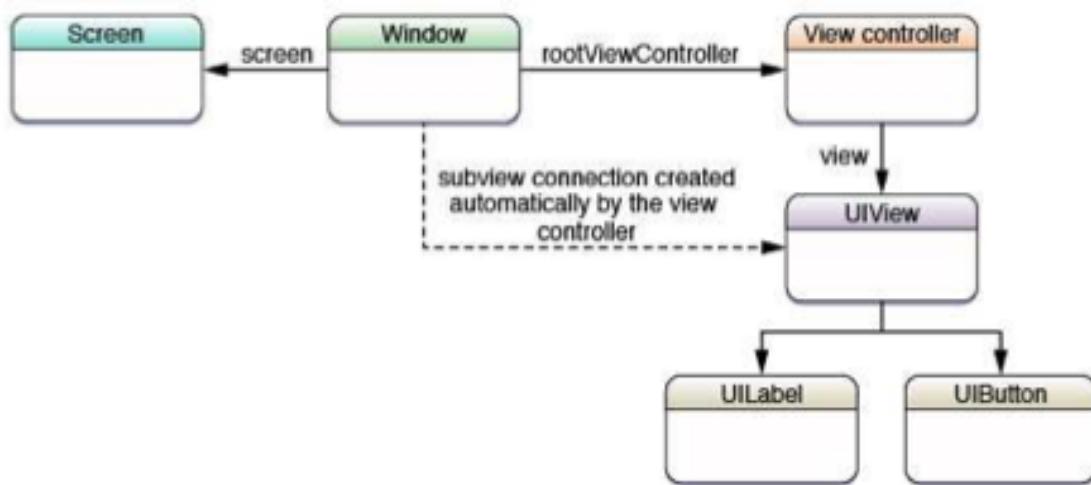


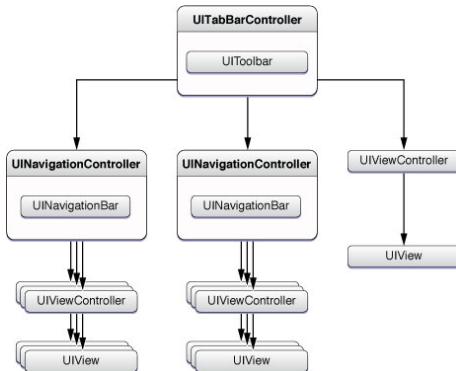
Figura 4.5: In generale ad UIWindow viene sempre aggiunto un UIViewController e la relativa UIView

rato ed implementato diverse classi di controllo, tutte ereditanti da UIControl, che definisce molti meccanismi targetaction specifici per iOS. Figura 4.4

4.1.4.3 Mediator Design Pattern: UIViewController

Il pattern Mediator è utilizzato nella piattaforma iOS soprattutto per organizzare le transizioni di differenti views. La classe UIViewController è al centro di questo design. Essa rappresenta una classe generale per gestire la view che contiene, con il compito specifico di mediare il flusso di dati che intercorre fra gli oggetti UIView e ciò che rappresenta il modello dei dati. Figura 4.5

La Figura 4.6 mostra le classi UIViewController disponibili nel framework UIKit.

Figura 4.6: **UIViewController**s disponibili in **UIKit**

Ogni View Controller, sia quelli forniti da iOS che quelli costruiti dagli sviluppatori, possono essere divisi in due categorie generali: i *Content View Controllers* e i *Container View Controllers*. In Figura 4.7 il Navigator Controller è un Container View Controller mentre glia altri sono Content View Controller.

4.1.4.4 Observer Design Pattern: le Notifiche

Il design pattern Observer definisce una dipendenza unoamolti fra due o più oggetti, in modo tale che ad ogni modifica allo stato dell'oggetto al quale si è interessati (il soggetto), corrisponda l'invio di un messaggio automatico a tutti gli oggetti "osservatore".

Nel framework Cocoa Touch sono state sviluppate alcune classi per utilizzare il pattern Observer senza doverlo implementare interamente, adattando tale pattern mediante la tecnologia delle Notifiche, modellate attraverso gli oggetti *NSNotification* e *NSNotificationCenter*. Figura 4.8

4.1.5 Delegate Design Pattern

La delegazione è un meccanismo mediante il quale un oggetto, chiamato host, detiene un riferimento ad un altro oggetto (il suo delegato) e periodicamente invia messaggi ad esso quando necessita di un input per compiere un determinato task.

Tale comunicazione è resa possibile mediante la dichiarazione, senza implementarli, di uno o più metodi che vanno a costituire un protocollo formale o informale;

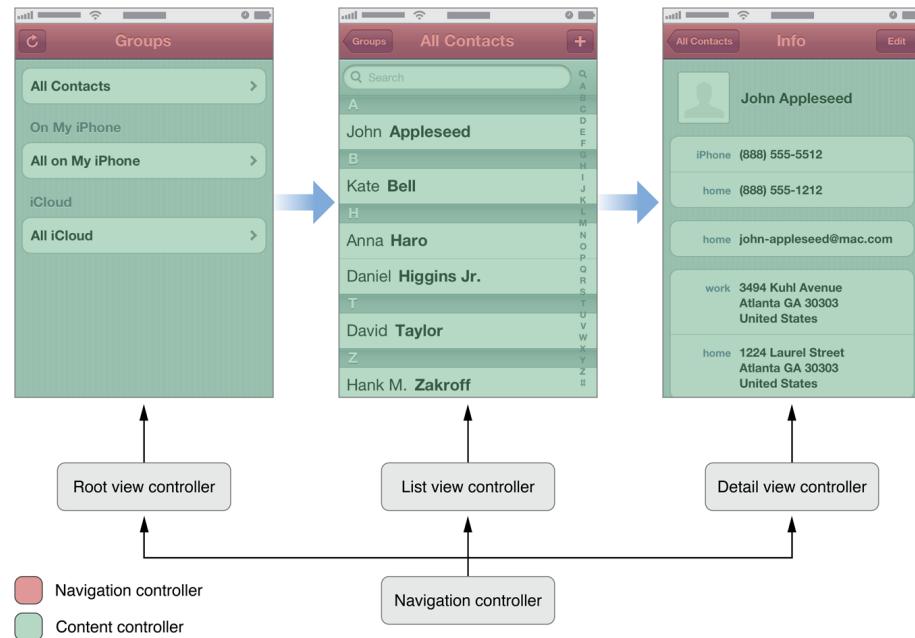


Figura 4.7: Esempi di Container View Controller e Navigator Controller

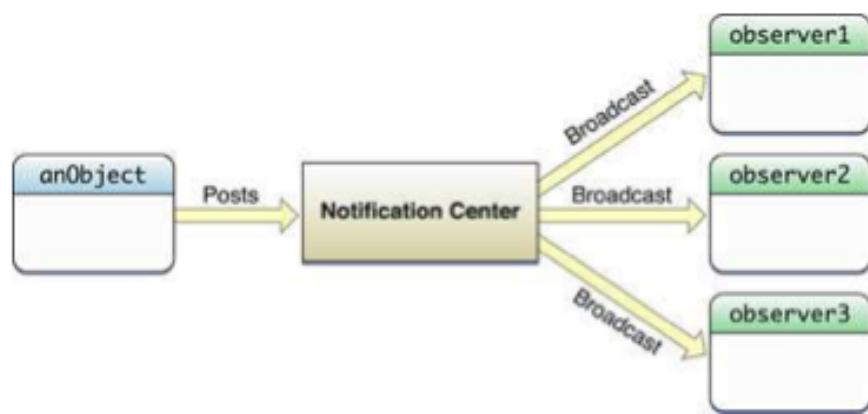


Figura 4.8: NSNotification e NSNotificationCenter

nell'approccio del protocollo formale (il più utilizzato) il delegato è incaricato di implementare i metodi indicati come obbligatori.

Il modello alla base del meccanismo adottato da Apple è conosciuto come pattern Delegate. Esso rappresenta principalmente una specializzazione del pattern Adapter, perché si riscontrano tutte le caratteristiche necessarie per essere considerato tale; infatti le funzioni del design pattern Adapter si riassumono nel converte l'interfaccia di una classe in un'altra che i clienti si aspettano di utilizzare, permettendo una cooperazione fra tali classi, cosa altrimenti non possibile a causa del sussistere di un'incompatibilità fra interfacce.

4.2 Gestione della Memoria

Si definisce *Gestione della Memoria a livello applicativo*, il processo di allocazione di memoria durante l'esecuzione di un programma, utilizzandola e rilasciandola quando non è più necessaria. Tipicamente è buona norma che un programma utilizzi un quantitativo di memoria che corrisponda al minimo necessario, come naturale risultato di una gestione efficiente delle risorse; questo è tanto più vero quanto più le risorse a disposizione sono limitate, come nel caso di iOS che prevede e definisce uno specifico meccanismo di gestione delle risorse in memoria, al quale sono affiancate una serie di politiche al riguardo (*Ownership Policy*) alle quali corrispondono diversi livelli di consapevolezza e responsabilità:

- *MRR (Manual Retain-Release)*
- *ARC (Automatic Reference Counting)*

4.2.1 Politiche di gestione per gli oggetti in memoria

L'insieme di politiche fissate per mantenere la validità del concetto di proprietà appena espresso sono implementate attraverso la nozione di *Reference Counting* (conteggio delle istanze, ovvero i riferimenti agli oggetti), un meccanismo o procedura che associa ad ogni oggetto Cocoa un intero rappresentante il numero di “proprietari” (in questo caso altri oggetti o frammenti di codice procedurale) interessati alla sua persistenza in memoria.

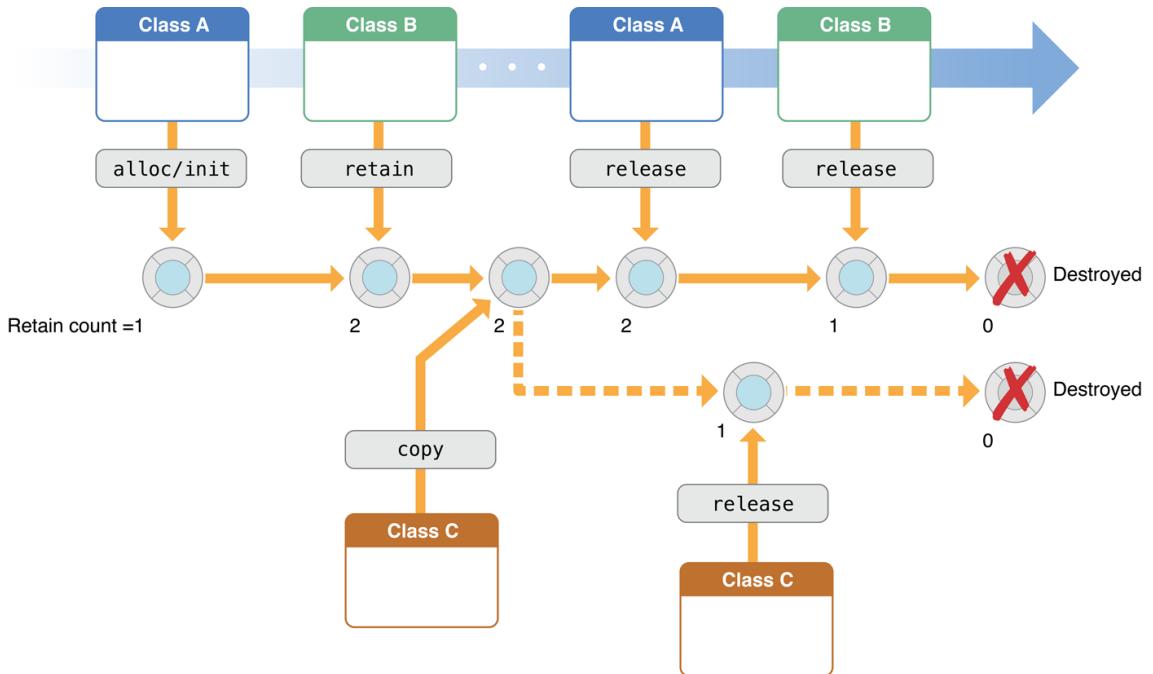


Figura 4.9: Il Retain Count in azione

Il Reference Counting è attuato tramite l'utilizzo di un set ristretto di istruzioni (messaggi) inviati agli oggetti da manipolare, con una politica sovrastante che fissa alcune semplici regole per ottenere un risultato consistente:

- Si detiene la proprietà su un oggetto che si è creato mediante le istruzioni che iniziano con *alloc*, *new*, *copy* o *mutableCopy*;
- In un qualsiasi momento è possibile richiedere la proprietà su un oggetto chiamando il metodo *retain*;
- È necessario rilasciare la proprietà che si detiene sull'oggetto inviando ad esso un messaggio di *release* o *autorelease*, dal momento in cui si sono concluse le operazioni con tale oggetto e non deve essere più utilizzare per i propri scopi.

Un esempio di tali meccanismo illustrato in Figura 4.9.

4.2.1.1 MRR

L'approccio denominato Manual Retain-Release (MRR) consente di gestire in maniera esplicita la memoria del livello applicativo, tenendo traccia di ciascun oggetto che si detiene. In questo caso è lo stesso sviluppatore a preoccuparsi di quali messaggi inviare per soddisfare correttamente la Object Ownership Policy, sempre utilizzando il modello di Reference Counting definito in precedenza.

Per ottimizzare l'utilizzo delle risorse o in casi in cui la performance applicativa risulti di primaria importanza è consigliato, se non necessario, prendere in considerazione l'utilizzo del metodo MRR per la gestione della memoria, perché è l'unico che consenta un'ampia gamma di opzioni e possibilità per ottenere una completa gestione delle risorse. Ovviamente, le grandi potenzialità offerte si traducono in un rischio e possibilità maggiore di compromettere il funzionamento dell'intera applicazione; per questo motivo spesso si preferisce affidarsi al metodo duale ARC.

4.2.1.2 ARC

ARC è l'acronimo di Automatic Reference Counting; utilizza la stessa procedura in MRR per implementare la Object Ownership Policy anche se, rispetto al metodo manuale accoppiato, inserisce automaticamente l'appropriato metodo di gestione della memoria a tempo di compilazione, rimuovendo la relativa implementazione dal codice sorgente.

Sostanzialmente ARC non introduce altre funzionalità rispetto ad MRR, ma aggiunge solamente una certa quantità di codice a tempo di compilazione per assicurare la persistenza degli oggetti in memoria solo per il tempo necessario, non oltre. Per questo motivo l'esecuzione di applicazioni dove viene utilizzato il metodo ARC sono pressoché identiche a quelle dove si utilizza correttamente il metodo MRR e le differenze nel comportamento dell'intera applicazione sono spesso trascurabili dato che l'ordinamento delle operazioni e la performance in entrambi i casi è molto simile.

La Figura 4.10 mostra un parallelo tra MRR e ARC:

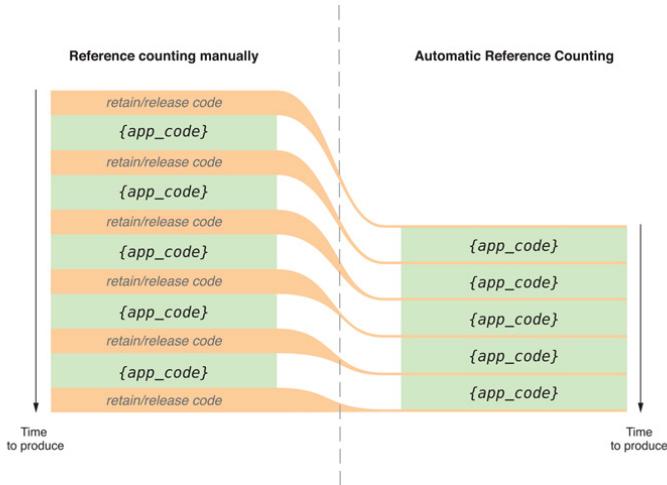


Figura 4.10: MRC vs ARC

4.3 Ambiente di sviluppo

4.3.1 Xcode

Xcode è un ambiente di sviluppo integrato (Integrated development environment, IDE) sviluppato da Apple Inc. per agevolare lo sviluppo di software per OS X e iOS. Questo fornisce gli strumenti per gestire l'intero flusso di lavoro, dallo sviluppo dell'applicazione, al collaudo, all'ottimizzazione fino alla sottomissione all'App Store. È fornito gratuitamente tramte lo stesso AppStore. Estende e rimpiazza il precedente tool di sviluppo della Apple, Project Builder, che era stato ereditato dalla NeXT. Xcode lavora in congiunzione con Interface Builder (anch'esso proveniente da NeXT) che permette agli sviluppatori che usano Carbon e Cocoa di disegnare interfacce grafiche per le applicazioni usando uno strumento grafico, senza la necessità di scrivere decine di righe di codice. Xcode include GCC, che è in grado di compilare codice C, C++, Objective-C/C++ e Java. Supporta ovviamente i framework Cocoa e Carbon, oltre ad altri. Durante al scrittura del codice Xcode è in grado di mostrare gli errori di sintassi, di logica, e suggerisce anche le opportune correzioni. La Figura 4.11 mostra l'interfaccia base di Xcode. Quest'ultima integra editing del codice, progettazione dell'interfaccia utente, asset management, testing e debugging all'interno di una singola finestra di lavoro. La finestra riconfigura il

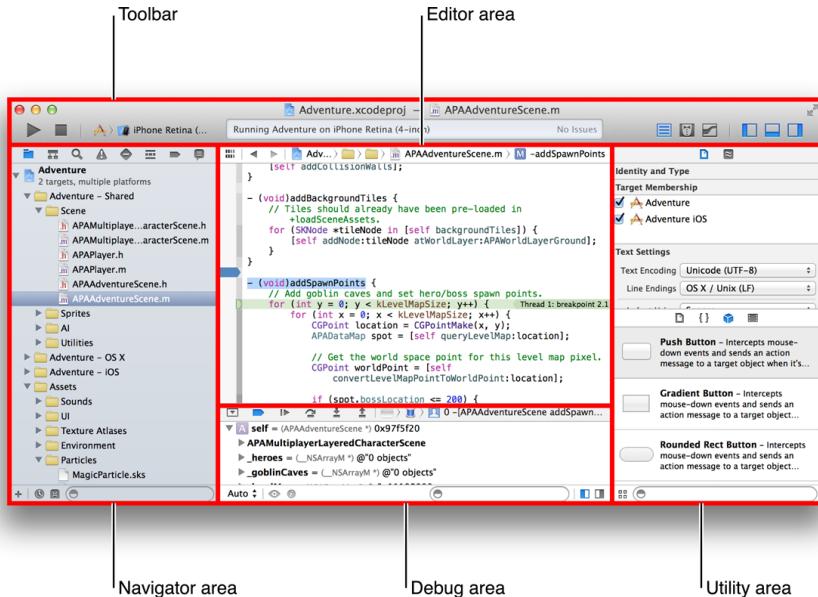


Figura 4.11: Xcode IDE

suo contenuto, a seconda di come si lavora. Ad esempio, se si seleziona un file in una zona, un editor appropriato si aprirà in un'altra zona. Oppure, se si seleziona un simbolo o un oggetto dell'interfaccia utente, la sua documentazione verrà visualizzata in un riquadro vicino.

Nel giugno del 2013 è stata rilasciata la versione 5.0 con alcune nuove funzioni che questa offre, quali ad esempio:

- **Automatic Configuration** - permette di configurare automaticamente le app per abilitare servizi Apple quali iCloud, Passbook o Game Center, direttamente all'interno dell'IDE;
- **Auto Layout** - consente di creare un'interfaccia per utente singolo che si adatta automaticamente alle dimensioni dello schermo, all'orientamento ed alla localizzazione;
- **Asset Management** - semplifica la gestione delle immagini; è possibile lavorare con tutte le versioni di ogni immagine, senza dover gestire files individuali di esse o memorizzare nomi di file convenzionali;

- **Source Control** - mostra sempre il “ramo” attivo del progetto al quale si sta lavorando e fornisce un accesso veloce per vedere, creare, e unire rami.

Un progetto Xcode è un archivio di tutti i file, le risorse e le informazioni richieste per la costruzione di uno o più prodotti software. Un progetto contiene tutti gli elementi utilizzati per costruire l'applicazione e mantiene le relazioni tra tali elementi. Contiene uno o più target, che specificano come costruire i vari prodotti. Un progetto definisce le impostazioni di compilazione di default per tutti i target del progetto.

Un file di progetto Xcode contiene le seguenti informazioni:

- i riferimenti a
 - ogni file sorgente (*header (.h)* e *implementation file (.m)*)
 - librerie, frameworks sia interni che esterni
 - risorse
 - XIB o Storyboard files
- le configurazioni per la compilazione del progetto
- elementi di debug

4.3.1.1 Storyboard

Storyboard è uno strumento che va a potenziare le possibilità di progettazione dell'interfaccia grafica rispetto a quanto disponibile in precedenza. Grazie a questo strumento si possono definire tutte le viste (schermate) dell'applicazione, la sequenza logica di navigazione da parte dell'utente e la modalità di presentazione delle viste stesse (per esempio tutto schermo o come popup modale). Tutto ciò è reso possibile grazie all'interfaccia di questo strumento che limita al minimo la scrittura di codice per la creazione e personalizzazione degli oggetti grafici come pulsanti, componenti dinamici (tabelle, pickerView) e sottoviste.

Vengono altresì semplificate altre funzionalità come ad esempio la gestione automatica di un sistema di navigazione tra le viste (NavigationController) e la gestione del TabBarController, la barra inferiore comune a tutte le viste. I comportamenti

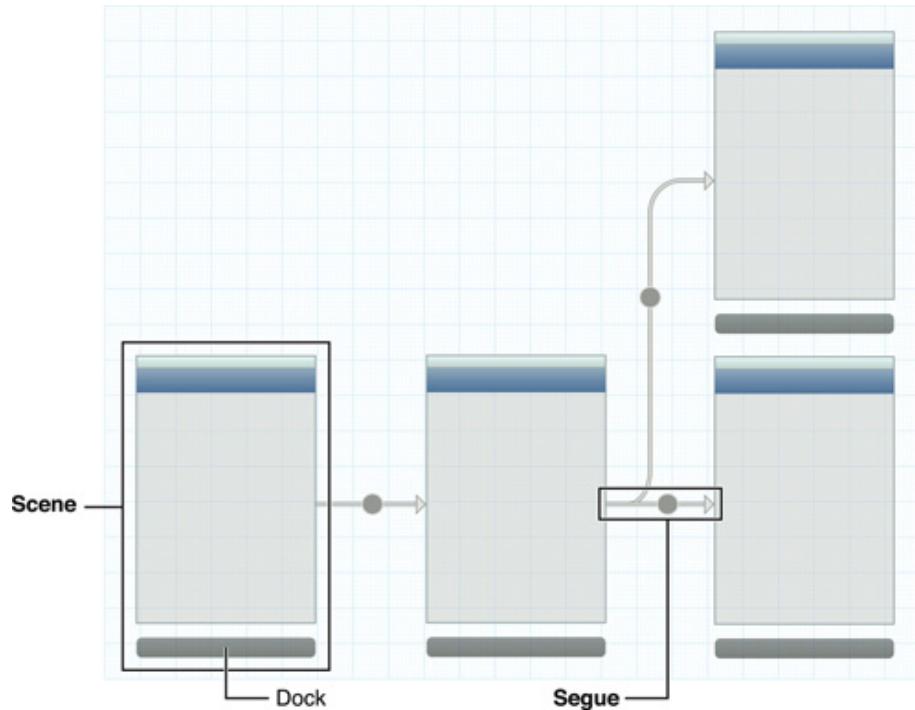


Figura 4.12: Storyboard

specifici delle singole viste, successivamente, vanno definiti nel codice delle classi di tipo `ViewController` seguendo il paradigma MVC.

Uno storyboard è composto da una sequenza di *Scene*, ciascuna delle quali rappresenta un `ViewController` e le sue `View`; le scene sono collegate da oggetti *Segue*, che rappresentano una transizione tra due `ViewController`.

La Figura 4.12 mostra un esempio di storyboard.

4.3.1.2 Interface Builder

Interface Builder appare per la prima volta nel 1988 come parte di NeXTSTEP 0.8. È stata una delle prime applicazioni commerciali che permetteva di disegnare interfacce e inserirci widget e menu usando il mouse.

Interface Builder mette a disposizione allo sviluppatore Objective-C palette o collezioni. Questi elementi dell'interfaccia grafica comprendono caselle di testo, tavole e menu a comparsa. Le palette di Interface Builder sono completamente estensibili: questo significa che è possibile sviluppare nuovi oggetti e aggiungerli alle palette.

Per costruire un’interfaccia lo sviluppatore deve semplicemente trascinare gli oggetti dalla paletta in una finestra o in un menu.

Le *actions* (o messaggi) che possono essere emesse dagli oggetti sono collegate ai *target* nel codice dell’applicazione e gli *outlet* (o puntatori) dichiarati nel codice dell’applicazione sono collegati a oggetti specifici. In questo modo tutta linizializzazione viene effettuata prima dell’esecuzione dell’applicazione; grazie a ciò, da una parte il processo di sviluppo viene semplificato e dall’altra vengono migliorate le prestazioni dell’applicazione.

Interface Builder registra l’interfaccia di un’applicazione come se fosse una directory (bundle), che contiene gli oggetti dell’interfaccia e le relazioni tra di essi utilizzate nell’applicazione. Questi oggetti sono raggruppati o in un file XML, o in un file stile-NeXT con estensione .nib.

Capitolo 5

Content Management System: Joomla!

Un Content Management System è un'applicazione che consente di variare i contenuti di un sito senza intervenire sulle pagine o sul database da cui sono lette le informazioni: una volta creata la struttura che dinamicamente produce le pagine, i contenuti possono essere inseriti e variati senza conoscenze tecniche di sviluppo.

Un CMS, più dettagliatamente, è un'applicazione lato server che utilizza un database preesistente per lo stoccaggio dei contenuti e d'informazioni aggiuntive (utilizzate poi per la corretta messa online); il sistema preleva contenuti e informazioni caricandoli online utilizzando il template grafico prescelto. L'applicazione è suddivisa in due parti:

- una sezione di amministrazione (*back end*), che serve ad organizzare e supervisionare la produzione dei contenuti;
- una sezione applicativa (*front end*), che l'utente web usa per fruire dei contenuti e delle applicazioni del sito.

I CMS possono essere realizzati tramite programmazione in vari linguaggi web tra cui, più comunemente, ASP, PHP, Microsoft.NET; il tipo di linguaggio adoperato è indifferente a livello di funzionalità.

Un CMS è essenzialmente formato da:

- **I Componenti di un CMS** - ovvero, quello che immaginiamo come un'unica applicazione in realtà è un insieme di componenti che lavorano in sinergia. Tra gli elementi più importanti troviamo:
 - la base dati contenente i contenuti;
 - il motore di pubblicazione delle pagine;
 - il sistema di gestione dei contenuti.
- **Il Database** - ovvero, l'archivio dei contenuti che saranno pubblicati sul sito. Normalmente contiene:
 - la struttura della parte dinamica del sito, controllabile dall'utilizzatore;
 - i contenuti degli articoli, della home e di tutte le parti in gestione all'utilizzatore;
 - i link ed banner visualizzati, con le informazioni collegate;
 - i dati di accesso.
- **Il Motore di Pubblicazione** - ovvero, la parte software deputata alla pubblicazione dei contenuti. Si distingue in due grandi categorie:
 - quelli che a fronte dell'inserimento di un nuovo contenuto, ne consentono immediatamente la visualizzazione;
 - quelli che necessitano di una fase di creazione delle pagine perché i nuovi contenuti siano visibili.
- **Il Gestore dei Contenuti** - ovvero, il front-end mediante il quale l'operatore gestisce i contenuti del sito. Le sue funzioni possono essere:
 - editor testuale o HTML dei contenuti;
 - gestione dei livelli di approfondimento;
 - gestione dei banner e dei link;
 - gestione della visibilità.

- **Banner e Link** - che consentono di posizionare sulle pagine banner e link per la gestione delle inserzioni. Su alcuni sistemi è possibile indicare ove devono essere posizionati i banner. Su altri è predisposta una parte fissa del layout ove sono inseriti a rotazione. Per i link, la scelta è tra sistemi che consentono di scegliere la posizione e quelli che li pongono raggruppati su una o più pagine.

5.1 Classificazione dei CMS

Come tutti i sistemi complessi, i CMS possono essere classificati secondo criteri differenti.

Una prima classificazione si basa sulla licenza d'uso con cui vengono distribuiti i vari CMS e a tale riguardo possiamo individuare due categorie di CMS: *Open Source* e *Proprietary*. Attualmente il mercato di questi software è dominato dalla categoria Open Source dato che risultano essere flessibili, estendibili e soprattutto economici. D'altro canto la documentazione e il supporto per questo tipo di CMS risultano essere piuttosto carenti, ma fortunatamente grande aiuto è offerto dalla comunità.

Un secondo criterio di classificazione può riferirsi alla tecnologia utilizzata per l'implementazione della dinamicità tipica dei CMS. Individuiamo così le seguenti categorie di CMS: *CMS php based*, *CMS java based*, *CMS asp based*, *CMS perl based*.

Infine, per differenziare ulteriormente i CMS, può considerarsi la tipologia di siti ospitati. Attualmente la rete è popolata da una vasta eterogeneità di contenuti e servizi, risulta quindi difficile se non impossibile fornire una classificazione esaustiva dei CMS a tale riguardo, dato che sono spesso sistemi modulari le cui funzionalità possono essere estese e/o modificate per mezzo di moduli aggiuntivi. È possibile però ricondurre la maggior parte dei CMS ad alcune macro categorie di seguito riportate:

- **Web CMS (WMCS)**- che forniscono un supporto generico nella gestione dei contenuti;
- **Transactional CMS (TCMS)** -che offrono supporto per la realizzazione di applicativi e-commerce;
- **Integrated CMS (IDMS)** - che offrono supporto alla gestione di contenuti e documenti interni a organizzazioni qualsiasi;
- **Pubblication CMS (PCMS)** - che offrono funzionalità specifiche per la gestione della pubblicazione di manuali, libri e simili;
- **Learning CMS (LCMS)** -dedicati alla realizzazione di sistemi e-learning web-based;

Si analizza ora in dettaglio il CMS utilizzato per lo sviluppo di parte del software che tale documento si propone di presentare: Joomla!

5.2 Joomla!

Joomla! è un software di content management (CMS) per la realizzazione di siti web, scritto completamente in linguaggio PHP. È pubblicato con licenza open source GNU GPL v.2 ed è totalmente gratuito: chiunque può scaricare ed utilizzare il software, inoltre, rispettando la suddetta licenza, è possibile modificare il codice ed adattarlo alle proprie esigenze.

Joomla! è formato diverse parti ben integrate tra loro, ognuna con il suo particolare compito, strutturate in modo tale che sia semplice aggiungere o togliere funzionalità al sistema. Molti aspetti, tra cui la facilità d'uso e l'estensibilità , hanno fatto di Joomla! il più popolare software per la costruzione di siti Web.

5.2.1 Struttura

Joomla! permette di creare e pubblicare siti internet dinamici non solo in modo semplice e veloce, ma anche con una buona sicurezza e elevate potenzialità. Grazie

alla presenza di un pannello di controllo ricco di icone è possibile esplorare tutte le funzionalità di Joomla! tra le quali spiccano l'inserimento dei contenuti e la configurazione globale delle caratteristiche del sito. Un aspetto molto importante è che tutte queste operazioni possono essere effettuate senza scrivere o modificare una sola linea di codice.

Altro aspetto rilevante è la presenza di un editor integrato (WYSIW-YG) che grazie alla sua interfaccia utente, simile a quella delle ben conosciute applicazioni Office, consente all'utente di creare agevolmente i contenuti che vuole realizzare.

È possibile paragonare l'intero CMS di Joomla! ad un sito diviso in due macrosezioni: il lato pubblico (*front end*) e il lato di amministrazione (*back end*).

Il front end è l'ambiente che tutti gli utenti possono vedere. Nel front end del sito un visitatore può navigare in tutte le pagine scegliendo quali parti del sito visitare e spostandosi tra di esse attraverso i menu. Un utente autorizzato, invece, può autenticarsi e compiere operazioni quali l'inserimento del testo ecc.; oppure può, sempre attraverso il login, diventare un utente registrato e visualizzare solo determinate pagine appositamente preparate o avere accesso ad aree riservate e così via.

Il back end, invece, è l'ambiente di tutta l'amministrazione del CMS nel quale può accedere, dopo aver eseguito il login, solo l'amministratore oppure altre persone da egli autorizzate.

La suddivisione dei due “ambienti” ha motivo di esistere in quanto è stata fatta per una maggiore sicurezza per il sito, scindendo in due parti distinte e separate gli ambienti. A dimostrazione di quanto appena scritto, un amministratore che effettua il login dal front end non ha automaticamente accesso al back end. Se volesse, infatti, entrare nell'area di amministrazione, dovrà nuovamente farsi riconoscere dal sistema.

5.2.2 Caratteristiche

Il CMS Joomla! è distribuito sotto forma di pacchetto compresso. È sufficiente scompattare l'archivio in una cartella pubblica di un server Web dotato di supporto

a PHP ed avere a disposizione un database MySQL per i dati del programma. Dopo un processo di installazione (più propriamente, di prima configurazione) di pochi minuti, il sito è operativo.

Tra le caratteristiche principali proposte ci sono:

- alto grado di personalizzazione grazie alle numerose estensioni moduli, componenti e plug-in disponibili sia come software libero che con altre licenze;
- caching delle pagine per incrementare le prestazioni;
- funzioni di Search Engine Optimization, per facilitare l'indicizzazione dei contenuti da parte dei motori di ricerca;
- feeding RSS, che permette ai visitatori di essere avvisati degli aggiornamenti dei contenuti mediante l'utilizzo di un feed reader;
- versione stampabile delle pagine;
- esportazione delle pagine in formato PDF;
- pubblicazione tipo Blog;
- sondaggi;
- ricerca testuale su tutti i contenuti inseriti;
- localizzazione internazionale, che permette la traduzione di ogni funzionalità del software nella propria lingua.

5.2.3 Estensioni

Uno dei punti di forza di Joomla! è la vivacità della comunità che lo supporta, sia in termini di discussione e capacità di aiuto che di ampia disponibilità di componenti aggiuntivi per personalizzare la funzionalità del motore.

Tutte le estensioni vengono distribuite sotto forma di pacchetti compressi, la cui installazione è gestita in maniera completamente automatica da uno script apposito, disponibile nella sezione di amministrazione del proprio sito Joomla!, che permette anche di disinstallare estensioni già installate.

Joomla! usa l'Extension Manager per organizzare le estensioni installate e per eseguire installazioni di nuove estensioni. Durante il processo di installazione Joomla! riconosce automaticamente il tipo di estensione che si sta installando.

Esistono tre tipi di estensioni: *componenti*, *moduli*, *plug-in*. Per modificare la rappresentazione delle informazioni Joomla! utilizza i *template*.

I componenti di Joomla! sono estensioni specifiche che permettono di aggiungere funzionalità complesse a un sito realizzato usando il CMS Joomla!. I componenti per Joomla! differiscono dai moduli essenzialmente per il livello di complessità supportato. Tradizionalmente, i moduli vengono utilizzati per implementare funzionalità elementari mentre i componenti possono aggregare più moduli per realizzare funzionalità più complesse e più complete. In generale, l'aggiunta di un componente corrisponde all'aggiunta di un'intera sezione al sito dove viene installato. A loro volta, secondo il medesimo schema modulare, i componenti possono essere usati da applicazioni che coprono livelli di funzionalità ancora più complessi. I componenti possono essere realizzati da qualsiasi utente di Joomla!. In rete si possono inoltre trovare componenti già pronti, prodotti da sviluppatori indipendenti.

I moduli di Joomla! sono estensioni che permettono l'aggiunta di piccole porzioni di HTML a un sito realizzato usando Joomla!. Sono usati per mostrare elementi di informazione o funzionalità interattive all'interno di un sito Joomla!, in maniera collaterale al contenuto principale. Si possono considerare come finestre aggiuntive attraverso le quali dare informazioni non necessariamente correlate alla pagina visualizzata, magari per mostrare le altre funzionalità del sito. I moduli recuperano le informazioni, o parti di informazioni definite attraverso parametri, e le visualizzano nella zona di loro competenza. All'utente viene data la possibilità di scegliere quali moduli visualizzare e dove collocarli all'interno del layout della pagina, in accordo con un template. Moduli sono anche i menu di navigazione all'interno di un sito Joomla!. Agendo direttamente nella sezione Gestione Moduli (Module Manager) dell'amministrazione, possono essere creati semplici moduli in HTML. Nel caso di script più complessi, essi sono in genere preparati per essere installati con le apposite procedure. Esistono moltissimi moduli di grande utilità già programmati e pronti all'uso, messi gratuitamente a disposizione nell'apposita sezione del sito ufficiale delle estensioni.

I plug-in sono porzioni di codice che, quando richiamate, attivano un programma, uno script o eseguono una specifica funzione. Spesso agiscono in background nell'intero sito. Possono essere semplicissimi come la funzione che sostituisce un certo testo digitato con una funzione codificata ma possono anche avere effetti molto più evidenti, come richiamare, all'interno delle finestre dei form per l'inserimento dei contenuti, le funzionalità di sofisticati editor di testo in modalità WYSIWYG elaborati da terze parti.

I template sono documenti HTML/CSS che contengono il codice necessario a guidare Joomla! e ad impaginare i contenuti: ad esempio contengono il codice che permette il caricamento dei vari moduli in posizioni definite, codice per caricare il cosiddetto mainbody (la zona in cui vengono presentati i contenuti principali generati da Joomla! o dai componenti aggiuntivi) e così via. Per ottenere l'aspetto desiderato molti template contengono anche una serie di immagini (per gli sfondi, i bordi eccetera). Ogni template può essere scaricato da appositi siti gratuitamente o a pagamento ed installato attraverso l'apposita area admin.

Nel prossimo capitolo verranno trattati in modo più approfondito i componenti e il loro sviluppo.

5.3 I Componenti di Joomla!

Un'estensione di Joomla! è un qualsiasi elemento che ne estende le funzionalità. Come precedentemente trattato sono tre i principali tipi di estensioni in Joomla!: componenti, moduli e plug-in. In questo capitolo verranno trattati più approfonditamente i componenti, che indubbiamente rappresentano le estensioni fondamentali di Joomla!.

5.3.1 MVC in Joomla!

Il modello MVC (Model-View-Control) è una delle novità nate con l'avvento della versione 1.5 di Joomla!. Infatti con la 1.5 sono stati riscritti dei file (in particolare

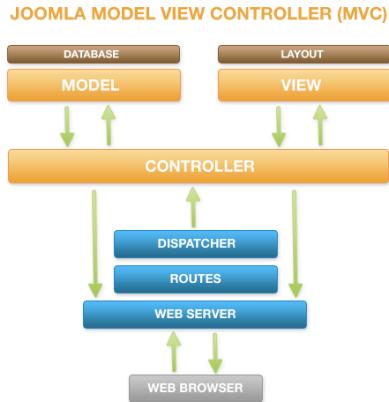


Figura 5.1: Il modello MVC in Joomla!

delle classi) del core del CMS, affinchè rispettasse i criteri di buona programmazione, tra cui l'utilizzo di alcuni pattern.

Come già detto, il model gestisce la logica dell'applicazione che comprende l'elaborazione e l'accesso ai dati. In modo formale si dice che il model incapsula i dati utilizzati dal componente. Poiché con Joomla! abbiamo una continua interazione con un database il model contiene per esempio i metodi per recuperare, aggiungere, rimuovere e aggiornare le informazioni delle tabelle. I dati per la maggior parte provengono dal database, ma possono provenire anche da altri fonti o servizi web. Il view organizza i dati, prendendoli dal model, in modo da essere visualizzati dall'utente. Infatti il suo compito è quello di generare l'output. Per esempio può implementare la pagina HTML che mostra il componente o il modulo. Il controller gestisce la parte delle richieste, le elabora e le indirizza verso l'oggetto destinatario. È il "vero responsabile" delle azioni degli utenti.

Ecco come avvengono le interazioni tra i diversi componenti: il controller determina il tipo di richiesta che gli arriva, la quale viene usata dal model per manipolare i dati e i dati risultanti dall'elaborazione vengono poi passati al view. Il Controller non espone i dati nel model, comanda i suoi metodi cambiando i dati, e poi li passa al view che li espone.

In questo modo si divide il layout dal codice che ha il compito di svolgere de-

terminate azioni, rendendo il sito più flessibile e potente. Una conseguenza molto importante dell'utilizzo di questo modello è data dalla possibilità di fare override. In particolare in Joomla l'implementazione del modello MVC è avvenuta utilizzando tre classi:

- JMolde;
- JView;
- JController.

Queste classi sono incluse nella libreria *joomla.application.component* ed ognuna di esse ha tutte le funzioni per assolvere appieno ai compiti che i tre diversi componenti devono assolvere.

5.3.2 Breve panoramica sulle funzionalità di Joomla!

5.3.2.1 Gestione delle richieste

L'Applicazione è un oggetto globale che viene usato da Joomla! per processare una richiesta. Le due classi interessate da un'applicazione sono *JSite* e *JAdministrator*: tramite queste due classi Joomla! soddisfa le richieste del front end e del back end.

L'oggetto Applicazione è sempre memorizzato nella variabile mainframe, una variabile globale alla quale è possibile accedere da funzioni e metodi dichiarandola globalmente. A differenza dell'oggetto Applicazione, per accedere ad altri oggetti globali è necessario utilizzare il metodo *JFactory::getInstance()*.

Le richieste di front end e di back end sono processate rispettivamente dai punti di accesso globali *index.php* e *administrator/index.php*. Quando vengono create estensioni per Joomla! non si devono assolutamente creare nuovi punti di accesso, in modo da garantire una serie di procedure che garantiscono sicurezza e stabilità.

5.3.2.2 Request

Lavorando i PHP, generalmente, sono utilizzate le variabili *\$_GET*, *\$_POST*, *\$_FILES*, *\$_COOKIE*, *\$_REQUEST*. In Joomla! inoltre viene utilizzata la classe statica *JRequest*. I due metodi maggiormente utilizzati della classe *JRequest* sono *JRequest::getVar()* e *JRequest::setVar()*: il loro uso è molto semplice: a *getVar()* viene

passato il nome della variabile da recuperare ed eventualmente il valore di default da assegnare se la variabile non è settata mentre, come si può intuire facilmente, a setVar() viene passato il nome della variabile da settare e il suo valore.

5.3.2.3 Sessioni

Le sessioni vengono utilizzate dalle applicazioni web per memorizzare facilmente dati relativi alla visita temporanea del client. In PHP si accede a questi dati attraverso la variabile superglobale `$_SESSION`. In Joomla! utilizziamo l'oggetto globale Sessione. I dati di sessione sono memorizzati in *namespace* e il namespace di default è `default`.

5.3.2.4 Factory Method

Nella programmazione ad oggetti, il Factory Method è uno dei design pattern fondamentali per l'implementazione del concetto di *factories*. Come altri pattern creazionali, esso indirizza il problema della creazione di oggetti senza specificarne l'esatta classe. Questo pattern raggiunge il suo scopo fornendo un'interfaccia per creare un oggetto, ma lascia che le sottoclassi decidano quale oggetto istanziare.

La creazione di un oggetto può, spesso, richiedere processi complessi la cui collocazione all'interno della classe di composizione potrebbe non essere appropriata. Esso può, inoltre, comportare la duplicazione di codice, richiedere informazioni non accessibili alla classe di composizione, o non provvedere un sufficiente livello di astrazione. Il Factory Method indirizza questi problemi definendo un metodo separato per la creazione degli oggetti; tale metodo può essere ridefinito dalle sottoclassi per definire il tipo derivato di prodotto che verrà effettivamente creato.

Joomla! possiede la classe `JFactory` che implementa il Factory Method: tale classe risulta molto importante poiché consente di accedere ed istanziare oggetti globali. Inoltre alcune classi di Joomla! implementano il pattern Singleton. Quindi, per instanziare un oggetto, bisogna dapprima provare con il metodo `JFactory`, poi con `getInstance()` (Singleton) e solo come ultima risorsa ricorrere direttamente al costruttore `new`.

5.3.2.5 Librerie

Joomla! include una serie di librerie molto utili se non indispensabili ai fini dello sviluppo. Per importare una libreria viene utilizzata la funzione *jimport()*; quando viene importata una libreria si può decidere se importarla completamente oppure solo una sua parte.

5.3.2.6 Costanti predefinite

All'interno di Joomla! esistono più di 400 costanti, molte delle quali provengono da librerie esterne. Una costante molto importante, ad esempio, è *_JEXEC* che è usata per assicurare che quando si includono dei file, questi ultimi siano inclusi da un punto di ingresso valido. La tabella in Figura 5.2 racchiude alcune delle costanti predefinite più importanti.

JPATH_ADMINISTRATOR	The path to the administrator folder.
JPATH_BASE	The path to the installed Joomla! site.
JPATH_CACHE	The path to the cache folder.
JPATH_COMPONENT	The path to the current component being executed.
JPATH_COMPONENT_ADMINISTRATOR	The path to the administration folder of the current component being executed.
JPATH_COMPONENT_SITE	The path to the site folder of the current component being executed.
JPATH_CONFIGURATION	The path to folder containing the configuration.php file.
JPATH_INSTALLATION	The path to the installation folder.
JPATH_LIBRARIES	The path to the libraries folder.
JPATH_PLUGINS	The path to the plugins folder.
JPATH_ROOT	The path to the installed Joomla! site.
JPATH_SITE	The path to the installed Joomla! site.
JPATH_THEMES	The path to the templates folder.
JPATH_XMLRPC	The path to the XML-RPC Web service folder.(1.5 only)

Figura 5.2: Alcune costanti predefinite di Joomla!

5.3.3 Il Database

Joomla! è sviluppato su un Database MySQL sul quale vengono memorizzati molti dei dati che stanno alla base di Joomla!: un'installazione base di Joomla! richiede all'incirca 30 tabelle.

Si accede al database di Joomla! utilizzando l'oggetto globale *JDatabase*. La classe *JDatabase* è una classe astratta che viene estesa con differenti driver di da-

tabase. In Joomla! sono presenti di base solamente due driver: *MySql* e *MySql*. Si accede all'oggetto globale `JDatabase` usando `JFactory`.

Quando creiamo estensioni di Joomla!, generalmente abbiamo bisogno di un database per memorizzare i dati, quindi risulta necessario estendere il database di Joomla! in maniera corretta:

- **Prefissi** - le tabelle del database di Joomla! hanno un prefisso, generalmente `jos_`, che aiuta ad usare lo stesso database per installazioni multiple di Joomla! Quando vengono scritte delle query SQL per una estensione si usa un prefisso che verrà poi sostituito con il prefisso attuale in runtime;
- **Convenzioni** - quando si creano le tabelle per una estensione vanno seguite alcune convenzioni standard. La più importante tra queste è il nome della tabella: tutte le tabelle devono usare il prefisso di tabella e devono iniziare con il nome dell'estensione. I nomi delle tabelle dovrebbero essere lowercase usando `_` come separatore di spazio;
- **Publishing** - si usa un campo di tabella chiamato *published* di tipo tiny-int(1) del valore 1 per indicare i dati da visualizzare, 0 viceversa;
- **Hits** - se si vuole tener traccia del numero di volte che un record viene visualizzato è possibile utilizzare un campo speciale chiamato *hits*, di tipo integer e di valore di default 0;
- **Ordering** - si utilizza il campo *ordering*, di tipo integer, per numerare la sequenza di record per determinare l'ordine con cui essi sono visualizzati.

5.3.3.1 Query

Quando si vuole effettuare una query al database si passa all'oggetto globale `JDatabase` la query da eseguire utilizzando il metodo `JDatabase::setQuery()`. Per eseguire la query si utilizza il metodo `JDatabase::query()`, analogo al metodo PHP `mysql_query()`. Se la query viene eseguita correttamente e produce un risultato in output, allora la risorsa sarà ritornata dalla funzione; se la query non deve produrre risultato la funzione ritornerà true se l'operazione è stata eseguita con successo.

Ci sono alcune regole da rispettare per scrivere correttamente delle query per il database di Joomla!:

- usare il prefisso #__ all'inizio di ogni nome di tabella;
- usare il metodo nameQuote() per incapsulare gli elementi della query;
- usare il metodo Quote() per incapsulare valori.

5.3.3.2 Processare i risultati

Si analizza ora come processare i risultati di una query al database di Joomla!. Risulta molto semplice utilizzare i metodi della classe JDatabase che consentono di presentare i risultati in formati diversi. I metodi utilizzati sono i seguenti:

- **loadResult():string** - questo metodo carica i valori del primo record tra i risultati; risulta utile quando vogliamo accedere ad un singolo campo in un record conosciuto;
- **loadResultArray(*numinarray*:int=0):array** - questo metodo carica una colonna *numinarray*, che viene usata per specificare quale colonna selezionare;
- **loadAssoc():array** - questo metodo carica il primo record come array associativo usando i nomi della colonna come chiavi per l'array;
- **loadAssocList(*key*:string):array** - questo metodo viene utilizzato per caricare un array di array associativi o un array associativo di associazioni. Se viene specificato il parametro *key* il metodo ritorna un array che utilizza *key* come chiave;
- **loadObject():stdClass** - questo metodo carica il primo record come oggetto usando i nomi dei campi come attributi;
- **loadObjectList(*key*:string):array** - questo metodo carica un array di oggetti della classe *stdClass*;
- **loadRow():array** - questo metodo carica il primo record come array. Risulta utile quando vogliamo lavorare su un solo record;

- **loadRowList(key:int):array** - questo metodo carica un array di array o un array associativo di array.

5.3.3.3 JTable

JTable è la potente classe astratta fornita da Joomla! con la quale è possibile operare molte funzioni di base sui record di una tabella. Per ogni tabella sulla quale si vuole usare la classe JTable è necessario creare una nuova sottoclasse.

Per creare nuove sottoclassi di JTable bisogna seguire alcune convenzioni che consentono di integrare l'estensione sviluppata nel framework Joomla!:

- il nome della classe deve essere uguale al singolare del nome della tabella con il prefisso Table;
- prima di utilizzare il metodo JTable::getInstance() è necessario riscrivere il costruttore di JTable in modo che tale costruttore abbia un solo parametro: l'oggetto database;
- si deve riscrivere il metodo check(), metodo utilizzato per validare i contenuti del buffer di ritorno dati da query.

Una volta creata la classe Table è possibile istanziare un oggetto attraverso il metodo getInstance().

5.3.4 Design dei componenti

Come precedentemente visto, i componenti sono costituiti da due elementi (front end e back end) al centro dei quali c'è il framework MVC. Si analizzano gli strumenti e l'approccio iniziale per lo sviluppo di componenti Joomla!.

5.3.4.1 Postazione di lavoro

Per lavorare sullo sviluppo di un componente per Joomla! bisogna innanzitutto crearsi una “postazione di lavoro”, cioè un supporto sul quale lavorare e testare il lavoro svolto. I tre strumenti principali che non possono mancare su tale postazione sono:

- **Editor di codice sorgente** - le estensioni di Joomla! consistono in codice sorgente. Il codice sorgente è un testo scritto in linguaggio di programmazione, ha bisogno di essere scritto e modificato. È necessario quindi utilizzare un editor di codice sorgente: può essere un'applicazione stand-alone oppure può essere costruito in un ambiente di sviluppo integrato;
- **Pacchetto Software XAMP** - XAMP indica una piattaforma per lo sviluppo di applicazioni web che prende il nome dalle iniziali dei componenti software con cui è realizzata: **X** sta per cross-platform, Server **A**pache http, **M**ySQL e **P**HP/**P**erl/**P**ython, i componenti principali per costruire una fattibile proposta generale di server Web.
- **Pacchetto Joomla!** - esistono due rami di sviluppo stabili per Joomla, la versione con supporto a lungo termine (Joomla! 2.5) che viene mantenuta ed aggiornata per circa due anni e la versione con supporto a breve termine (Joomla! 3.1) che contiene le maggiori novità e viene mantenuta ed aggiornata per soli 6 mesi.

Dopo aver scelto l'editor e installati i pacchetti XAMP e Joomla! si può procedere alla creazione del componente.

5.3.4.2 Componente di base

Un buon approccio iniziale è quello di creare un installer base che configuri un componente vuoto. Il codice presente in Codice 5.1 può essere utilizzato a tale scopo.

Questo codice può essere utilizzato creando un XML Manifest file usando la codifica UTF-8; una volta creato il file XML, quest'ultimo va inserito in un archivio ZIP in modo da poter essere installato in Joomla! attraverso l'Extension Manager.

I file che gestiscono la parte pubblica del componente saranno nella directory */components/com_myextension* e in questa cartella dovrà essere creato il controller *myextension.php* che sarà eseguito quando il componente sarà invocato da front end. Il back end del componente sarà in */administrator/components/com_myextension* e in questa cartella dovrà essere creato un altro controller, con lo stesso nome del precedente, che sarà eseguito quando il componente sarà invocato dal back end.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <install type="component" version="2.5.0">
3      <name>My Component</name>
4      <creationDate>Month Year</creationDate>
5      <author>Author's Name</author>
6      <authorEmail> Author's Email</authorEmail>
7      <authorUrl>Author's Website</authorUrl>
8      <copyright>Copyright Notice</copyright>
9      <license>Component License Agreement</license>
10     <version>Component Version</version>
11     <description>Component Description</description>
12     <administration>
13         <menu>My Component</menu>
14     </administration>
15     <install />
16     <uninstall />
17 </install>
```

Codice 5.1: Esempio installer XML

Una volta fatto ciò il front end sarà accessibile creando una nuova voce di menù, mentre il back end sarà accessibile dal menù componenti del pannello amministratore. Se il componente non è ancora stato associato ad una voce di menu può essere visto, in modo più semplice, richiamando la pagina *index.php* e passandogli il parametro *option* con il nome dell'estensione che si vuole visualizzare. Per accedere al front end, ad esempio, si dovrà richiamare la URL */index.php?option=com_myextension*.

Si analizza ora come creare dei semplici Model, View e Controller per trattarli più approfonditamente nel prossimo capitolo, che spiegherà in dettaglio lo sviluppo del componente descritto da questo documento.

5.3.4.3 Costruzione di un Model

Prima di iniziare a sviluppare un Model bisogna analizzare come denominarli: Joomla! segue una particolare convenzione nel nominare Model Controller e View. Per nominare un Model, ad esempio, si deve rispettare la seguente sintassi: NomeDelComponente+Model+NomeEntità. Una volta creato il Model deve essere collocato nella cartella */model* del componente.

Si procede ora alla creazione di un Model di esempio per il componente “My Extension”; il nome dell’entità sarà Entity. Il Model si chiamerà quindi *MyextensionModelEntity* e sarà collocato in *model/entity.php*.

Tutti i Model estendono la classe astratta JModel. In realtà, come vedremo nel prossimo capitolo, i Model ereditano da due classi astratte *JModelAdmin* e *JModelList* le quali ereditano direttamente da JModel; inoltre in base alla classe da cui ereditano varierà la convenzione Joomla! adottata nel nominarli. Quello che segue in Codice 5.2 è un esempio che mostra una semplice implementazione della classe.

La riga `defined('JEXEC')` o `die('Restricted Access')`; serve solamente ad impedire che il file possa essere richiamato direttamente dall’utente, senza passare dai meccanismi di Joomla.

La riga con `jimport` carica la classe JModel. Per ridurre il carico del sistema Joomla! non carica automaticamente tutto il codice necessario, ma bisogna richiederlo con questa sintassi.

La classe in Codice 5.3 verrà dotata ora di un semplice metodo `getEntity()`.

```

1 <?php
2 //Impedisce l'accesso diretto al file
3 defined('_JEXEC') or die('Restricted Access');
4 //Importa la classe principale JModel
5 jimport( 'joomla.application.component.model' );
6 /**
7  *Entity Model
8 */
9 class MyextensionModelEntity extends JModel{
10 }
```

Codice 5.2: Esempio classe Model

```

1 class MyextensionModelEntity extends JModel{
2     public function __construct($config = array()){
3         if (empty($config['filter_fields'])) {
4             $config['filter_fields'] = array('id', 'name_entity');
5         }
6         parent::__construct($config);
7     }
8
9     function getEntity(){
10         $db = JFactory::getDBO();
11         $query = $db->getQuery(true);
12         $query->select('id, name_entity');
13         $query->from('#_myextension_entity');
14         $db->setQuery($query);
15         $result = $query->loadObject();
16         return $result;
17     }
18 }
```

Codice 5.3: Esempio classe Model

```

1 <?php
2 //Impedisce l'accesso diretto al file
3 defined('_JEXEC') or die('Restricted Access');
4 //Importa la classe principale JModel
5 import( 'joomla.application.component.view' );
6 /**
7  *Entity View
8 */
9 class MyextensionViewEntity extends JView{
10 }

```

Codice 5.4: Esempio classe View

Il Model è ora utilizzabile ed è possibile recuperare un record dalla tabella #_myextension_entity. Analogamente all'accesso ai dati, si utilizza il Model per modificarli o per eliminare interi record.

5.3.4.4 Costruzione di una View

Le View sono separate le une dalle altre: ogni View ha la sua propria cartella nella directory /views. Insieme alla cartella si definisce un diverso file per ogni documento che la View supporta: HTML, PDF, feed, Se si definisce una View per un documento HTML si deve creare anche una cartella /tmpl che conterrà il layout per la visualizzazione.

Come per i Model, anche nel nominare le View è necessario seguire una convenzione di Joomla!, si scriverà quindi: NomeDelComponente+View+NomeEntità. La classe inoltre deve essere collocata in un file chiamato view.TipoDiDocumento.php. Tutte le View estendono la classe JVView. Segue un esempio, in Codice 5.4, che mostra una semplice implementazione di una View per l'entità Entity.

Come si può osservare dal codice, è necessario importare la libreria *joomla.application.view*.

Il metodo più importante di ogni classe View è il metodo *display()* che risulta già definito nella classe JVView: in generale in questo metodo si interroga il Model

```

1 class MyextensionViewEntity extends JView{
2     function display(){
3         //interroga il model
4         $this->items = $this->get('Entity');
5         //display
6         parent::display();
7     }
8 }
```

Codice 5.5: Esempio classe View

```

1 <table width="100" border="0" cellspacing="0" cellpadding="0">
2     <tr>
3         <th>Name</th>
4         <td><?php $this->items->name_entity?></td>
5     </tr>
6     <tr>
7         <th>ID</th>
8         <td><?php $this->items->id?></td>
9     </tr>
10 </table>
```

Codice 5.6: Esempio template HTML

per i dati, si personalizza il documento, e, infine, si visualizza. Si implementa ora, in Codice 5.5 un metodo display() per l'esempio precedente.

La funzione get() di JVView si limita a richiamare la funzione del model specificata come parametro, sempre con il prefisso get(). In questo caso verrà, pertanto, chiamata la funzione getEntity() del model. Il risultato ritornato è poi assegnato alla variabile \$items di JVView. Come è possibile vedere, da nessuna parte è stato detto di istanziare un Model. Questo viene fatto in automatico da JVView.

Infine, si crea un semplice layout, come quello in Codice 5.6, per visualizzare i dati appena recuperati in HTML, chiamato default.php che sarà contenuto nella cartella views/tmpl.

```

1 <?php
2 //Impedisce l'accesso diretto al file
3 defined('_JEXEC') or die('Restricted Access');
4 //Importa la classe principale JModel
5 import( 'joomla.application.component.controller' );
6 /**
7  *Entity Controller
8 */
9 class MyextensionControllerEntity extends JController{
10 }

```

Codice 5.7: Esempio classe Controller

5.3.4.5 Costruzione di un Controller

I Controller possono essere usati in molti modi diversi. Il pattern MVC afferma che, ai fini di un applicazione, potrebbe bastare un solo Controller; in realtà risulta molto utile implementare più Controller: uno per ogni entità.

I Controller estendono la classe astratta JController che viene importata dalla libreria *joomla.application.component.controller*. Più specificatamente, come si vedrà nel prossimo capitolo, i Controller ereditano da due classi astratte JControllerAdmin e JControllerForm a seconda dei compiti che il Controller riveste nel componente. Entrambe queste classi ereditano direttamente da JController.

I Controller utilizzano ed eseguono le task, cioè stringhe che identificano ciò che si vuole far fare all'applicazione. Associata al Controller c'è una mappa delle task che associa ai metodi dei nomi convenzionali. In JController è presente un metodo speciale chiamato execute(): questo metodo viene utilizzato per eseguire una task: (\$controller ->execute(task)).

Anche i Controller hanno una particolare convenzione per i nomi. Si scriverà quindi: NomeDelComponente+Controller+NomeEntità. Normalmente i Controller sono collocati nella cartella controllers o, se esiste un unico controller, si trova nella cartella principale del componente in un file chiamato controller.php. L'esempio in Codice 5.7 mostra una semplice implementazione di un Controller per l'entità Entity.

```

1 class MyextensionControllerEntity extends JController{
2     function display(){
3         $modelEntity = $this->getModel('Entity');
4         //display
5         parent::display();
6     }
7 }
```

Codice 5.8: Esempio classe Controller

```

1 COSTANTE1="Traduzione"
2 COSTANTE2="Traduzione"
```

Codice 5.9: Esempio di file di localizzazione

Ci sono molti metodi della classe JController che possono essere riscritti. Il più comunemente riscritto, come mostra il Codice 5.8, è il metodo display(): questo metodo istanzia un oggetto View, collega un Model e visualizza il risultato. Esistono due importanti variabili richieste che vengono usate dal metodo display() per stabilire ciò che si deve fare: La vista richiesta e il layout da usare.

5.3.4.6 Internazionalizzazione

La localizzazione di un software è l'insieme dei processi necessari per rendere un software utilizzabile da altre culture o da persone che parlano altre lingue.

Joomla! 2.5 supporta nativamente dei meccanismi di localizzazione che consentono di creare siti multilingua, ma il suo supporto si estende anche ai singoli componenti, mettendo a disposizione un meccanismo relativamente semplice per tradurre le varie parti nelle differenti lingue.

Purtroppo il meccanismo scelto non si basa sullo standard gettext presente sui sistemi Linux e supportato in modo egregio da PHP, ma su un sistema proprio di Joomla! basato su dei file che assomigliano a semplici file di configurazione che assomigliano al Codice 5.9:

Il significato di questo file è abbastanza semplice: sostituisci la stringa COSTANTE*n* con la corrispondente traduzione. Questa sostituzione è effettuata da una classe

apposita di Joomla! che si chiama *JText*, il cui costruttore stesso vuole una costante come parametro di input e restituisce la corrispondente traduzione basandosi sulla lingua selezionata dall'utilizzatore del sito.

Capitolo 6

Progettazione del sistema

6.1 Introduzione

Nell’ambito del progetto Casa+ è previsto lo sviluppo di un’applicazione per dispositivi mobili a supporto delle attività quotidiane delle persone affette da sindrome di Down. Dato che esiste già un’applicazione di questo tipo per piattaforma android, si è deciso di scegliere iOS per il proseguo di questo lavoro, in modo da coprire la maggior parte del mercato mobile, dominato per l’appunto da questi due colossi. Tale applicativo deve poter rendere accessibile in mobilità alcuni dei servizi disponibili in ambiente desktop:

- il servizio di gestione di una lista della spesa;
- il servizio di gestione di un ricettario;
- il servizio di visualizzazione dell’orario;
- il servizio denominato “Rubrica dell’euro”.

La lista della spesa è lo strumento attraverso il quale i soggetti interessati realizzano una lista di prodotti da acquistare presso un particolare rivenditore. La particolarità di questo strumento è la possibilità di realizzare una propria lista sulla base dei dati forniti dagli educatori, ricevere aggiornamenti in tempo reale su eventuali modifiche eseguite in ambiente desktop nonché la possibilità di ricevere delle

notifiche quando, in mobilità, ci si trova in prossimità di un rivenditore per ognuno dei prodotti nella lista.

Il ricettario è lo strumento attraverso il quale i soggetti interessati ottengono tutte le informazioni necessarie alla preparazione di una ricetta. È fornita, oltre ai dettagli generali (es. calorie, numero di persone, utensili e ingredienti), una lista degli step da seguire con le relative istruzioni e timer, in modo da guidare l'utente il più possibile e fornire un adeguato supporto alla gestione del tempo.

L'orologio è lo strumento attraverso il quale i soggetti interessati visualizzano una coppia di orologi, uno digitale e uno analogico. Lo strumento deve permettere la lettura da parte del sistema dell'ora corrente.

La rubrica dell'euro, infine, è lo strumento attraverso il quale i soggetti interessati vengono informati di quali prodotti è possibile acquistare con un determinato taglio di moneta in euro.

La progettazione e lo sviluppo di tale sistema è argomento di questo documento di tesi.

6.2 Applicazioni simili già esistenti sul mercato

Nel particolare contesto Casa+ non è risultata di grande importanza l'analisi delle applicazioni simili già presenti sul mercato in quanto lo sviluppo dell'interazione con l'applicazione è stato effettuato in stretta collaborazione con gli educatori dei ragazzi e, secondo le loro direttive, si è costruito un sistema sostanzialmente *adhoc* per il particolare tipo di utente.

6.3 Specifiche di progetto

La progettazione del software è avvenuta in seguito ad una fase di studio sullo scenario cui l'applicazione è destinata e sulle specifiche delle funzionalità e delle caratteristiche richieste. È stata richiesta la realizzazione di un software per dispositivi mobili che consenta l'utilizzo di alcuni dei servizi disponibili in ambiente desktop, visualizzabili al sito , per utenti affetti da sindrome di Down con disabilità cognitive.

L'applicativo è stato realizzato per l'uso su smartphone e tablet di ultima generazione. In particolare si è progettato un'applicazione per il sistema iOS versione 7.x. All'applicazione, inoltre, è richiesta la necessità di comunicare con l'applicativo desktop mediante connessione dati a pacchetto o wireless.

6.3.1 Requisiti funzionali

1. Il sistema deve comunicare con il database remoto
 - (a) Il sistema deve permettere l'inserimento di dati nel db remoto
 - (b) Il sistema deve permettere la modifica di dati nel db remoto
 - (c) Il sistema deve permettere la cancellazione di dati nel db remoto
2. Il sistema deve fornire notifiche
 - (a) Il sistema deve fornire notifica della modifica della lista della spesa presente nel db remoto
 - (b) Il sistema deve fornire notifica sulla posizione dell'utente
3. Il sistema deve gestire una lista della spesa
 - (a) Il sistema deve permettere l'inserimento/eliminazione di un prodotto nella lista
 - (b) Il sistema deve permettere l'eliminazione contemporanea di tutti gli elementi della lista
 - (c) Il sistema deve permettere l'inserimento della quantità di un prodotto nella lista
 - (d) Il sistema deve permettere di marcare un prodotto come acquistato
 - (e) Il sistema deve permettere di visualizzare i dettagli di un prodotto nella lista
4. Il sistema deve mostrare l'orario
 - (a) Il sistema deve permettere di visualizzare un orologio analogico

- (b) Il sistema deve permettere di visualizzare un orologio digitale
 - (c) Il sistema deve permettere di leggere l'ora corrente
5. Il sistema deve gestire un ricettario
- (a) Il sistema deve permettere di visualizzare i dettagli di una ricetta della lista
 - (b) Il sistema deve permettere di visualizzare gli utensili di una ricetta della lista
 - (c) Il sistema deve permettere di visualizzare gli ingredienti di una ricetta della lista
 - (d) Il sistema deve permettere di visualizzare tutti gli step di una ricetta in ordine di posizione
 - (e) Il sistema deve permettere di aggiungere utensili e ingredienti di una ricetta alla lista della spesa
6. Il sistema deve mostrare i prodotti acquistabili con un dato taglio di moneta in euro
- (a) Il sistema deve permettere di visualizzare tutti i tagli di monete disponibili
 - (b) Il sistema deve permettere di visualizzare una lista di prodotti o in alternativa un messaggio che indica che non è possibile acquistarne nessuno
7. Il sistema deve fornire la possibilità di stampa della lista della spesa attraverso una stampante compatibile

6.3.2 Requisiti non funzionali

1. Il sistema deve poter essere utilizzato su dispositivi Apple iPhone, iPad, iPod Touch
2. Il sistema deve seguire le direttive sull'interazione fornite dagli educatori
3. Il sistema deve seguire le direttive grafiche fornite dagli educatori

4. Il sistema deve fornire feedback vocale laddove richiesto dagli educatori
5. Il sistema deve permettere la personalizzazione da parte degli educatori

6.3.3 Requisiti informativi

1. Le informazioni di una categoria sono:
 - (a) nome
 - (b) immagine
2. Le informazioni di una posizione sono:
 - (a) nome
3. Le informazioni di un negozio sono:
 - (a) nome
 - (b) posizione (latitudine, longitudine)
 - (c) indirizzo
 - (d) telefono
4. Le informazioni di un prodotto sono:
 - (a) nome
 - (b) una lista di categorie
 - (c) posizione
 - (d) immagine
 - (e) quantità
 - (f) prezzo
 - (g) una lista di negozi
5. Le informazioni di una categoria di ricette sono:
 - (a) nome

- (b) immagine
6. Le informazioni di uno step di una ricetta sono:
- (a) posizione (numero dello step)
 - (b) descrizione
 - (c) immagine
 - (d) timer (0 timer, 1 countdown)
 - (e) tempo
7. Le informazioni di una ricetta sono:
- (a) nome
 - (b) descrizione
 - (c) una lista di step
 - (d) una lista di ingredienti
 - (e) una lista di utensili
 - (f) categoria
 - (g) difficoltà
 - (h) immagine
 - (i) calorie
 - (j) numero di persone
8. La lista della spesa è composta da una lista di prodotti

6.4 Modellazione UML

Contemporaneamente all'ideazione delle interfacce il processo di sviluppo del software è proseguito con la realizzazione di diagrammi UML. In questa fase dello sviluppo sono stati analizzati gli scenari d'utilizzo del software da parte degli utenti mediante elaborazione dei *Casi d'Uso (UC Use Cases)*, la progettazione di una base di dati locale che consenta di essere sincronizzata con la base di dati remota e la definizione del diagramma delle classi.



Figura 6.1: Attori

6.4.1 Casi d'uso

Gli attori sono illustrati in Figura 6.1

L'applicativo Casa+ consente all'utente la composizione di una lista della spesa mediante aggiunta o rimozione dei prodotti e la definizione delle loro quantità, il marcamento dei prodotti che sono stati acquistati, la stampa della lista della spesa, la consultazione del ricettario, la visualizzazione della Rubrica dell'euro e la visualizzazione dell'orologio. In Figura Figura 6.2 è mostrato lo scenario.

Casa+

- **Caso d'uso:** Componi Lista della spesa
 - **Descrizione:** Il sistema deve permettere l'inserimento/eliminazione di un prodotto nella lista e il settaggio della relativa quantità
 - **Attori:** Utente, Educatore
 - **Precondizioni:** -
 - **Attivazione:** -
 - **Scenario:**
 1. l'utente sceglie una categoria
 2. l'utente sceglie un prodotto
 3. se la configurazione lo consente il prodotto viene aggiunto direttamente alla lista
 4. **altrimenti** viene presentata una schermata con i dettagli del prodotto dalla quale è possibile aggiungere il prodotto nella lista
 5. il sistema memorizza i dati

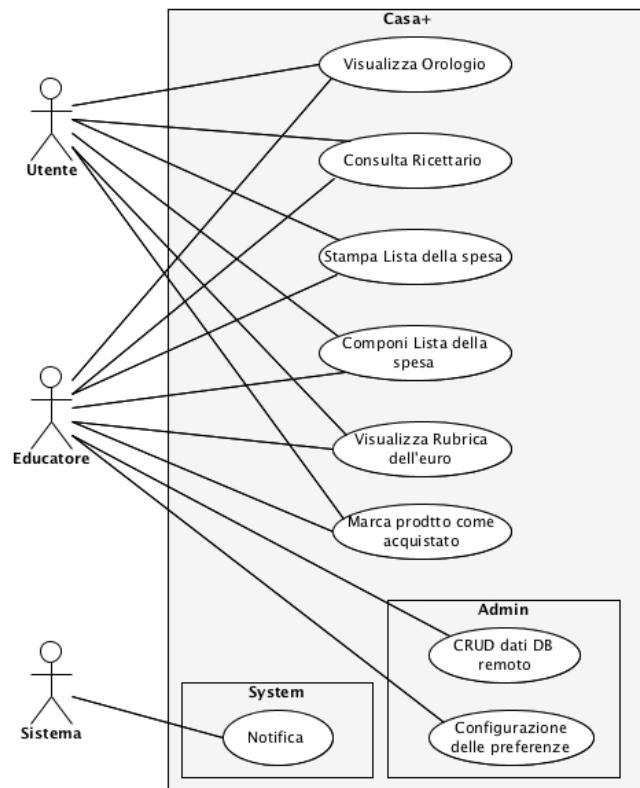


Figura 6.2: Casi d'uso

- **Caso d'uso:** Marca prodotto come acquistato

- **Descrizione:** Il sistema deve permettere di marcare un prodotto come acquistato
- **Attori:** Utente, Educatore
- **Precondizioni:**
 1. è stata creata una lista di prodotti
 2. sono state definite le quantità di prodotti
- **Attivazione:** -
- **Scenario:**
 1. l'utente sceglie un prodotto



2. se la configurazione lo consente il prodotto viene marcato come acquistato
3. altrimenti viene presentata una schermata con i dettagli del prodotto dalla quale è possibile marcato il prodotto come acquistato
4. il sistema memorizza i dati

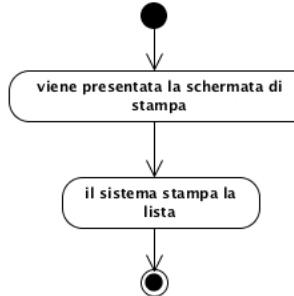
- **Caso d'uso:** Stampa della lista

- **Descrizione:** Il sistema deve fornire la possibilità di stampa della lista della spesa attraverso una stampante compatibile acquistato
- **Attori:** Utente, Educatore
- **Precondizioni:**
 1. è stata creata una lista di prodotti
 2. sono state definite le quantità di prodotti
- **Attivazione:** -



– Scenario:

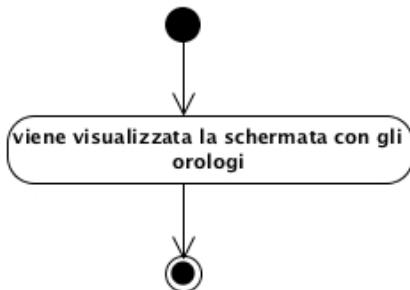
1. viene presentata la schermata di stampa
2. il sistema stampa la lista



• Caso d'uso: Visualizza orologio

- Descrizione: Il sistema deve fornire la possibilità di visualizzare una coppia di orologi, uno analogico e uno digitale
- Attori: Utente, Educatore

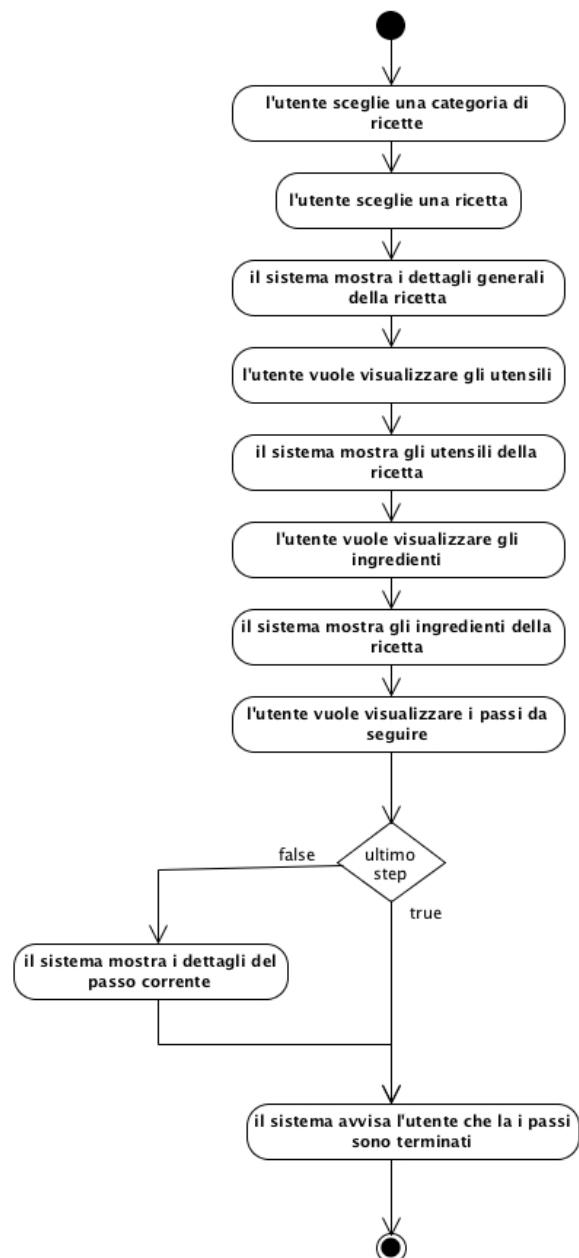
- **Precondizioni:** -
- **Attivazione:** -
- **Scenario:**
 1. viene presentata la schermata con gli orologi



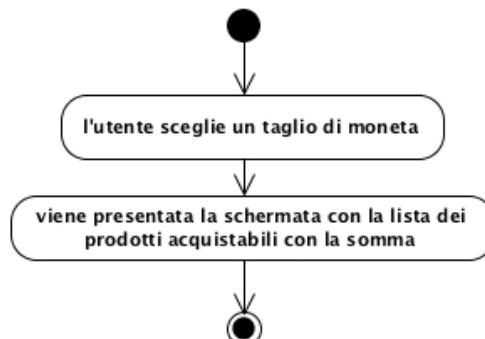
- **Caso d'uso:** Consulta ricettario

- **Descrizione:** Il sistema deve fornire la possibilità di selezionare una ricetta e consultarne i relativi dettagli quali: gli ingredienti, gli utensili e i passi da seguire
- **Attori:** Utente, Educatore
- **Precondizioni:** -
- **Attivazione:** -
- **Scenario:**
 1. l'utente sceglie una categoria di ricette
 2. l'utente sceglie una ricetta
 3. il sistema mostra i dettagli generali della ricetta
 4. l'utente vuole visualizzare gli utensili
 5. il sistema mostra gli utensili della ricetta
 6. l'utente vuole visualizzare gli ingredienti
 7. il sistema mostra gli ingredienti della ricetta

8. l'utente vuole visualizzare i passi da seguire
9. **finchè** il passo corrente non Il sistema deve fornire la possibilità l'ultimo il sistema mostra i dettagli del passo corrente
10. **altrimenti** il sistema avvisa l'utente che la i passi sono terminati



- **Caso d'uso:** Consulta Rubrica dell'euro
 - **Descrizione:** Il sistema deve fornire la possibilità, una volta selezionato un taglio di moneta, di visualizzare quali prodotti è possibile comprare con la somma selezionata
 - **Attori:** Utente, Educatore
 - **Precondizioni:** –
 - **Attivazione:** –
 - **Scenario:**
 1. l'utente sceglie un taglio di moneta
 2. viene presentata la schermata con la lista dei prodotti acquistabili con la somma selezionata



Admin

- **Caso d'uso:** Configurazione delle preferenze
 - **Descrizione:** Il sistema deve permettere la modifica delle preferenze
 - **Attori:** Educatore
 - **Precondizioni:** –
 - **Attivazione:** –
 - **Scenario:**

1. l'educatore sceglie le impostazioni
2. il sistema memorizza le impostazioni



- **Caso d'uso:** CRUD dati DB remoto

- **Descrizione:** Il sistema deve permettere di eseguire operazioni CRUD su tutti i dati presenti sul database remoto
- **Attori:** Educatore
- **Precondizioni:** -
- **Attivazione:** -
- **Scenario:**
 1. l'educatore sceglie cosa modificare
 2. l'educatore inserisce i dati richiesti
 3. il sistema memorizza i dati



System

- **Caso d'uso:** Notifica

- **Descrizione:** Il sistema deve permettere di ricevere aggiornamenti in tempo reale su eventuali modifiche eseguite in ambiente desktop sulla lista della spesa nonché di ricevere delle notifiche quando, in mobilità, ci si trova in prossimità di un rivenditore per ognuno dei prodotti nella lista.
- **Attori:** Sistema
- **Precondizioni:** -
- **Attivazione:** -
- **Scenario:**
 1. il sistema monitora lo stato osservato
 2. il sistema notifica all'utente del cambiamento dello stato osservato
 3. l'utente comunica al sistema come comportarsi



6.4.2 Struttura Architetturale

Le scelte architettoniche hanno tenuto conto di due caratteristiche fondamentali che l'applicazione deve avere:

- l'accesso in mobilità dei dispositivi client, permettere cioè ai client di accedere dovunque si trovino;
- la possibilità di utilizzo di diverse tipologie di client, ovvero permettere a diversi dispositivi di accedere all'applicazione.

Viste queste caratteristiche, e per tutto quello che si è detto precedentemente in questo capitolo, si è scelto di implementare il sistema con un'architettura client-server.

Attualmente l'applicativo desktop è realizzato mediante tecnologia web, in particolare attraverso l'utilizzo dei linguaggi HTML, CSS, Javascript e PHP per il web server e MySQL per il database. Il punto debole di questa soluzione è l'esportabilità. Se si volesse infatti installare l'applicativo su un altro server, occorrerebbe eseguire diversi step quali: copiare tutti i file sul server, installare e popolare il database ed infine configurare il server, con almeno i parametri per la connessione alla base dati. Senza contare che tali azioni devono necessariamente essere compiute da persone che conoscano le tecnologie sopraelencate.

Per questo motivo si è deciso di effettuare una “ristrutturazione” dell'applicativo trasformandolo in un componente Joomla!. In questo, modo si eliminerà la com-

plessità dovuta all'installazione che si ridurrebbe al caricamento di un componente sul CMS. Altro vantaggio derivante dalla nuova soluzione è l'estensibilità: volendo in futuro aggiungere funzionalità all'applicativo basterà creare nuovi componenti ed installarli, senza dover assolutamente mettere mano ai file sul server o al database. A tutto questo possiamo aggiungere gli altri vantaggi derivanti dall'utilizzo di un CMS:

- possibilità di aggiornare un sito Web senza essere esperti di HTML;
- possibilità di realizzare in modo facilitato l'architettura dei dati progettata, attraverso la definizione di diverse sezioni e categorie in cui classificare gli articoli;
- possibilità di separare nettamente i dati dalla loro rappresentazione;
- possibilità di gestire in modo semplice la pubblicazione e la rimozione delle informazioni;
- integrazione con contenuti provenienti da diverse fonti come database o RSS;
- gestione degli utenti con mailing list e messaggistica;
- funzionalità di ricerca dei contenuti che vanno oltre la disposizione in categorie.

In Figura 6.3 viene presentata più in dettaglio l'architettura mostrando il contenuto delle componenti che la compongono.

La comunicazione tra il client e il server, infine, sarà effettuata attraverso *json*: il client effettua le richieste di tipo *http-post* mentre il server risponde al client attraverso documenti json.

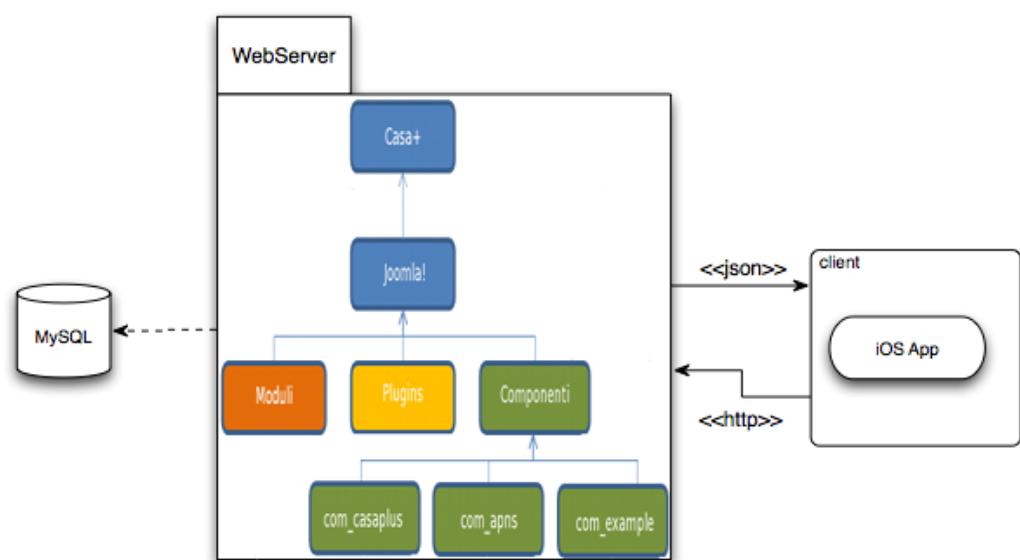


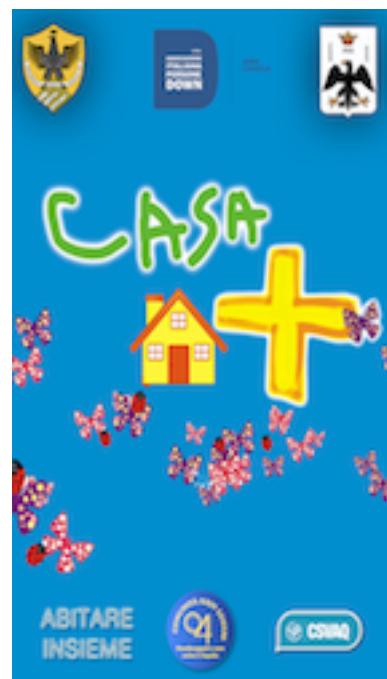
Figura 6.3: Schema architettrale

Capitolo 7

Sviluppo del sistema

In questo capitolo verrà descritta la parte di sviluppo del lato client e del lato server del sistema *casa+* descritto nel capitolo precedente.

7.1 Mobile Application



7.1.1 Diagramma delle classi

7.1.1.1 Model

Il diagramma in Figura 7.1 mostra le classi che, incapsulando lo stato dell'applicazione, definiscono i dati e le operazioni che possono essere eseguite su questi. Quindi definisce le regole di business per l'interazione con i dati, esponendo alla View ed al Controller rispettivamente le funzionalità per l'accesso e l'aggiornamento.

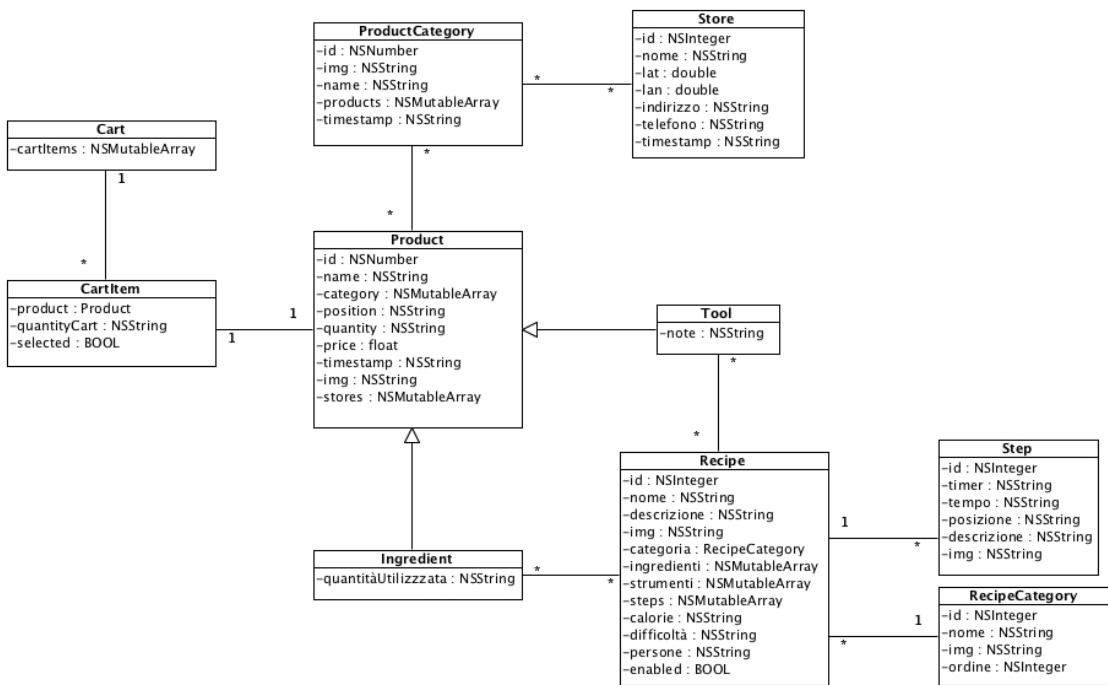


Figura 7.1: Diagramma delle classi model

7.1.1.2 ViewControllers

La complessità del diagramma delle classi non 'e riproponibile in una singola immagine per una visualizzazione ottimale del contenuto. Per tale motivo si è preferito mostrare gli esplosi dello schema complessivo della struttura dell'applicativo i che riassume in grandi linee l'interazione tra le diverse componenti dell'applicazione

UIViewControllers Lista della spesa

La Figura 7.2 mostra l'insieme delle classi ViewController che gestiscono la lista della spesa, e le relazioni che ci sono tra di esse.

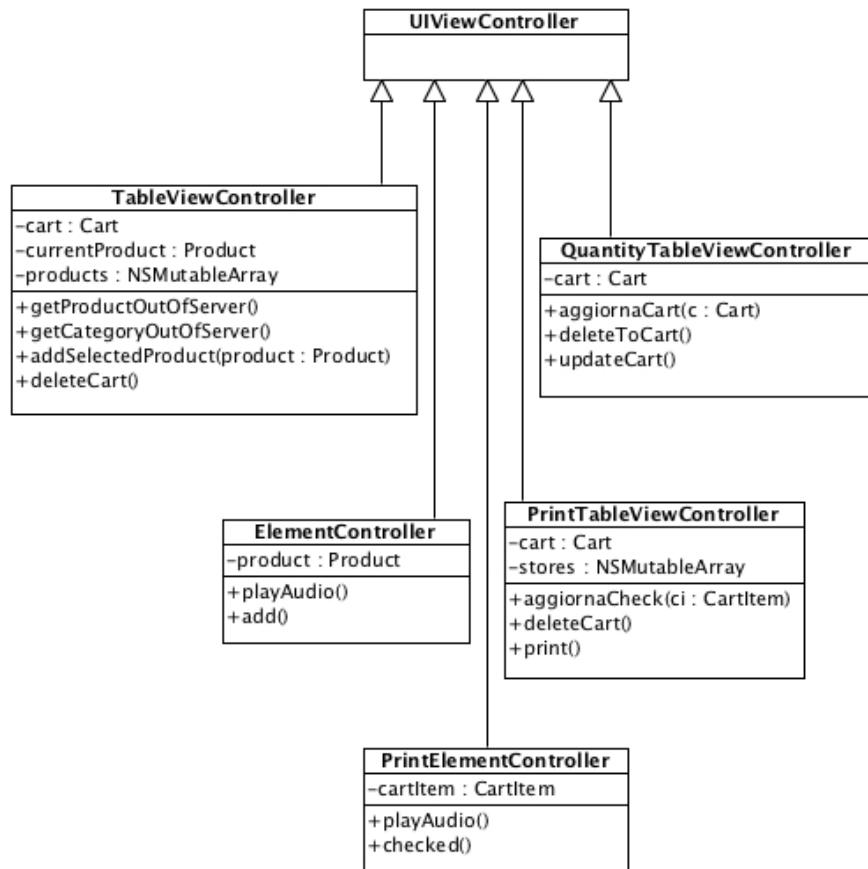
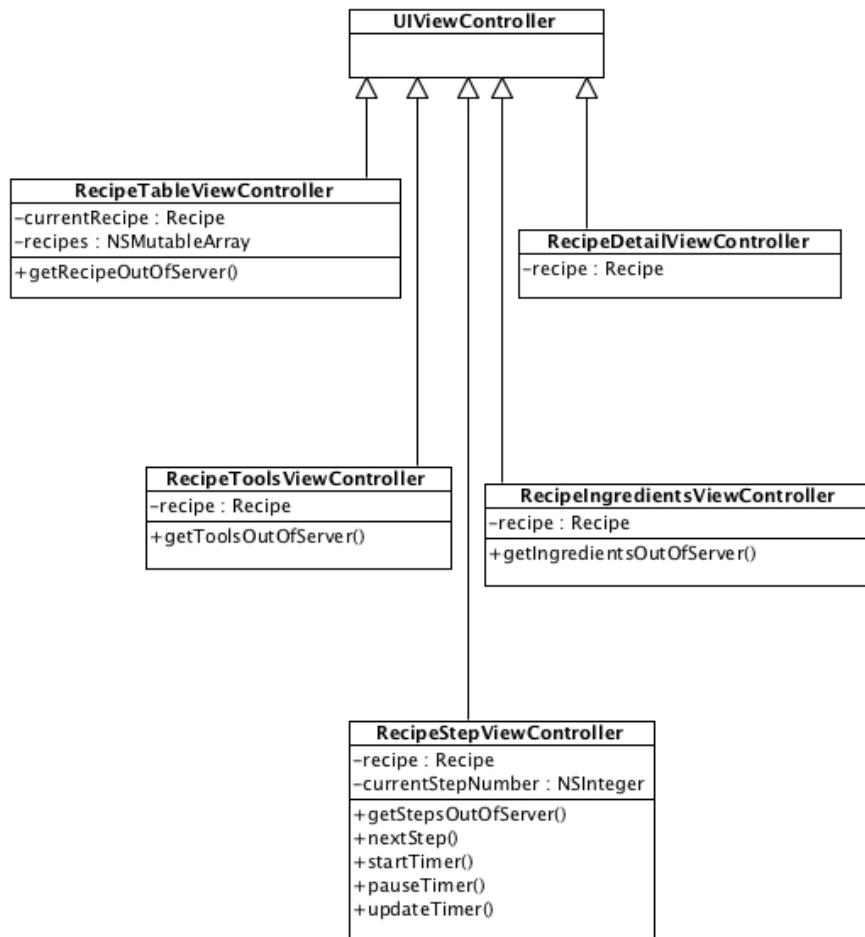


Figura 7.2: UIViewControllers Lista della spesa

UIViewControllers Ricettario

La Figura 7.3 mostra i pacchetti ViewController che compongono la gestione del ricettario.

Figura 7.3: `UIViewController`s Ricettario

`UIViewController`s Login

In Figura 7.4 è riportato il diagramma delle classi `ViewController` che si occupano della gestione dell'accesso da parte degli utenti all'applicazione.

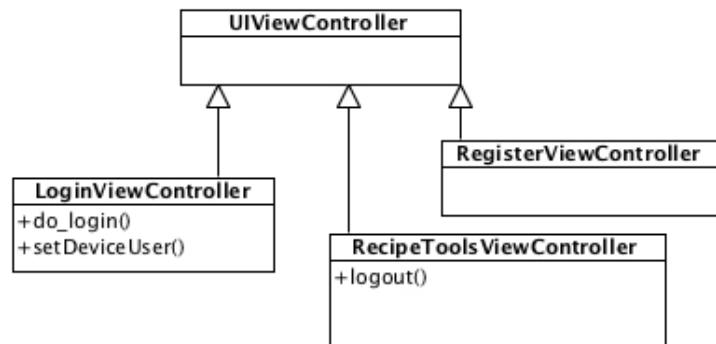


Figura 7.4: `UIViewControllerAnimated Login`

UIViewControllerAnimated Rubrica dell'euro e L'Orologio

Infine, in Figura 7.5 e Figura 7.6 sono riportati i diagrammi delle classi View-Controller che si occupano rispettivamente della gestione del servizio della rubrica dell'euro e della visualizzazione degli orologi.

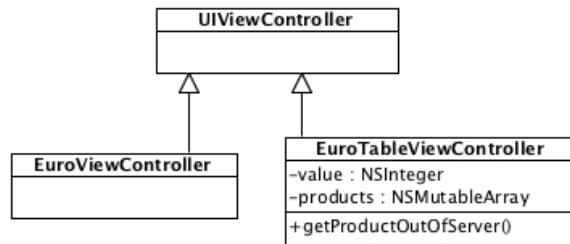


Figura 7.5: `UIViewControllerAnimated Rubrica dell'euro`

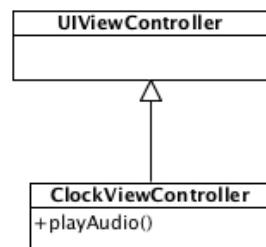


Figura 7.6: `UIViewControllerAnimated Orologi`

7.1.2 Principali aspetti implementativi

7.1.2.1 Principali classi dell'applicazione

- **TableViewController**

Questa classe che estende *UIViewController* gestisce la vista per la selezione dei prodotti che compongono la lista della spesa. Al suo interno è implementata una *UITableView* che contiene i prodotti suddivisi per categoria e ne gestisce la visualizzazione secondo le preferenze.

```

1   //
2   // ViewController.h
3   // CasaPlus
4   //
5   // Created by Luca Finocchio on 04/11/13.
6   // Copyright (c) 2013 GLDeV. All rights reserved.
7   //
8
9
10  #import "productTableViewCellHeader.h"
11  #import "MBProgressHUD.h"

12  @interface ViewController : UIViewController <SectionHeaderViewDelegate,
13      UITableViewDelegate, UITableViewDataSource, UISearchBarDelegate,
14      MBProgressHUDDelegate>

15  @property (strong, nonatomic) NSMutableArray *products;
16  @property (strong, nonatomic) IBOutlet UITableView *tableView;

17  - (void)getProductOutOfServer;
18
19  - (void) getCategoryOutOfServer;

20
21
22  @end

```

```
1 //  
2 // ViewController.m  
3 // CasaPlus  
4 //  
5 // Created by Luca Finocchio on 04/11/13.  
6 // Copyright (c) 2013 GLDeV. All rights reserved.  
7 //  
8  
9 #import "TableViewCellController.h"  
10 #import "Product.h"  
11 #import "productTableView.h"  
12 #import "Category.h"  
13 #import "productTableViewHeader.h"  
14 #import "UIImageView+AFNetworking.h"  
15 #import "AFNetworking.h"  
16 #import "AppDelegate.h"  
17 #import "ElementController.h"  
18 #import "Cart.h"  
19 #import "CartItem.h"  
20 #import "QuantityTableViewCellController.h"  
21  
22  
23 @interface ViewController()  
24 {...}  
25  
26 @property (strong, nonatomic) NSMutableArray *filteredProduct;  
27 @property (strong, nonatomic) Product *currentProduct;  
28 @property (nonatomic,assign) NSInteger openSectionIndex;  
29 @property (nonatomic,strong) NSMutableArray* categories;  
30 @property (nonatomic,strong) NSMutableArray* sectionInfoArray;  
31 @property (nonatomic, strong) Cart *cart;  
32 @property (nonatomic,assign) NSInteger tmpid;
```

```
34 @property (strong, nonatomic) IBOutlet UIBarButtonItem *deleteCart;
35 @property (strong, nonatomic) IBOutlet UISearchBar *searchbar;
36 @property (strong, nonatomic) IBOutlet UIBarButtonItem *refresh;
37 @property (strong, nonatomic) IBOutlet UIBarButtonItem *avanti;

38 @end

40 @implementation ViewController

42 - (id)initWithCoder:(NSCoder *)decoder
43 { ... }

44 - (void)viewDidLoad
45 { ... }

48 - (void)viewDidAppear:(BOOL)animated
49 { ... }

50 -(void) viewWillDisappear:(BOOL)animated
51 { ... }

54 #pragma mark - Table View Delegate
55 ...
56
57 #pragma mark Section header delegate
58 ...
59
60 #pragma mark - Menage Segue
61 ...
62
63 #pragma mark - UISearchBarDelegate
64 ...
65
66 #pragma mark - Manage refresh
67 ...
```

```

68 #pragma mark - Custom

70 - (void)getProductOutOfServer
{
72     HUD = [[MBProgressHUD alloc]
73             initWithView:self.navigationController.view];
74     [self.navigationController.view addSubview:HUD];
75     HUD.delegate = self;
76     HUD.labelText = @"Prodotti";
77     HUD.detailsLabelText = @"Caricamento in corso ...";
78     HUD.square = YES;
79     [HUD show:YES];

80     // Imposto i parametri della richiesta GET
81     AFHTTPRequestOperationManager *manager =
82         [AFHTTPRequestOperationManager manager];
83     manager.responseSerializer.acceptableContentTypes =
84         [NSSet setWithObject:@"text/html"];
85     NSDictionary *params = @{@"option": @"com_casaplus",
86                             @"task": @"products.get_product"};
87
88     // Eseguo la richiesta
89     [manager GET:[NSString stringWithFormat:@"%@%@%@",

90             [defaults objectForKey:@"url"], @"index.php"]
91             parameters:params success:
92             ^(AFHTTPRequestOperation *operation, id responseObject) {

93
94         // Recupero i prodotti dal json e li inserisco in un array
95         NSMutableArray *productsTMP =[[NSMutableArray alloc] init];
96         NSMutableDictionary *jsonDict =
97             (NSMutableDictionary *) responseObject;
98         NSArray *products = [jsonDict objectForKey:@"products"];
99
100        [products enumerateObjectsUsingBlock:
101            ^(id obj,NSUInteger idx, BOOL *stop){


```

```

104         if ([[obj objectForKey:@"id"] integerValue] != self.tmpid){
105             Product *product = [[Product alloc] init];
106             product.id = [obj objectForKey:@"id"];
107             product.name = [[obj objectForKey:@"nome"]
108                             capitalizedString];
109             product.category = [[NSMutableArray alloc] init];
110             [product.category addObject:
111                 [[obj objectForKey:@"categoria"] uppercaseString]];
112             product.price = [[obj objectForKey:@"prezzo"] floatValue];
113             product.img = [obj objectForKey:@"img"];
114             self.tmpid = [[obj objectForKey:@"id"] integerValue];
115             [productsTMP addObject:product];
116         } else {
117             [[[productsTMP lastObject] category] addObject:
118                 [[obj objectForKey:@"categoria"] uppercaseString]];
119         }];
120         self.products = [productsTMP copy];
121         [self getCategoryOutOfServer];
122         [self.tableView reloadData];
123     }
124     failure:^(AFHTTPRequestOperation *operation, NSError *error) {
125         // Mostro un alert con il messaggio di errore
126         UIAlertView *message = [[UIAlertView alloc]
127             initWithTitle:[NSString stringWithFormat:
128                             @"%@: %ld",@"Errore nella connessione al server"
129                             ,(long)operation.error.code]
130             message: @"Controlla che la tua connessione internet
131                     sia attiva o che l'indirizzo del server sia corretto"
132             delegate:nil cancelButtonTitle:@"OK" otherButtonTitles:nil];
133         [message show];
134     }];
135 }
136 @end

```

I metodi `getProductOutOfServer` e `getCategoryOutOfServer` consentono il recupero dei prodotti e delle categorie dal server e si occupano di costruire due array contenenti gli oggetti `Product` e `ProductCategory` rispettivamente. In particolare, i metodi eseguono una richiesta http-post in modo da recupero i file JSON dal server: se la richiesta va a buon fine si procede con la creazione degli array ciclando un array temporaneo generato dal JSON stesso.

Poichè la classe `TableViewController` è una classe di tipo `UIViewController` il suo compito è quello di governare una particolare vista. In Figura 7.7 è mostrata la View generata e controllata dalla classe `TableViewController`.

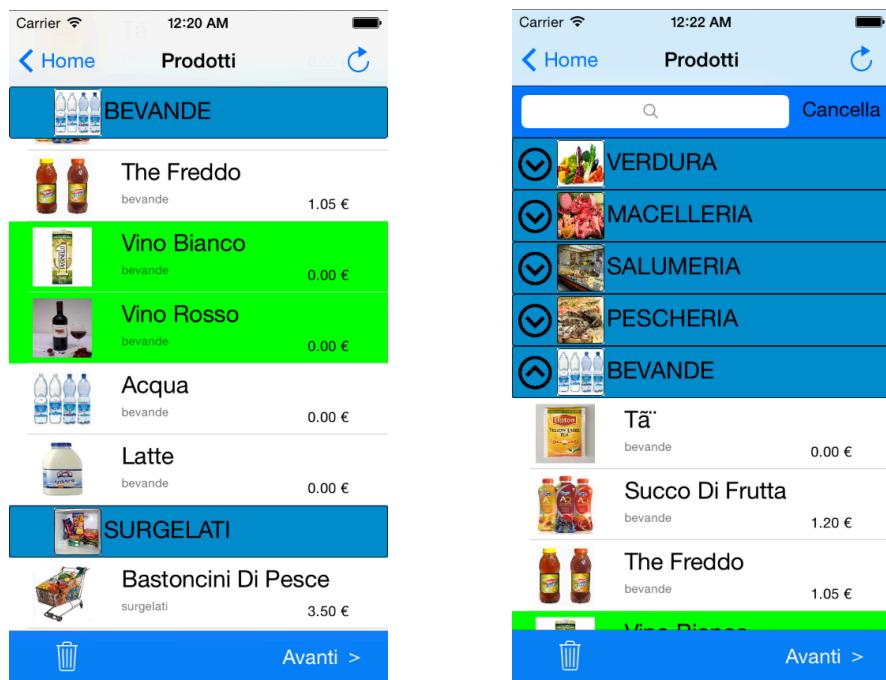


Figura 7.7: `TableViewController`

• `ClockViewController`

Questa classe anch'essa estende `UIViewController` e si occupa della gestione degli orologi e della lettura dell'ora corrente.

```
//  
2 // ClockViewController.h  
// clock  
4 //  
// Created by Luca Finocchio on 04/11/13.  
6 // Copyright (c) 2013 GLDeV. All rights reserved.  
//  
8  
9  
10 #import <UIKit/UIKit.h>  
11 #import "ClockView.h"  
12  
13 @interface ClockViewController : UIViewController  
14 {  
15     NSTimer *timer;  
16 }  
17  
18 @property (strong, nonatomic) IBOutlet ClockView *clockView;  
19 @property (strong, nonatomic) IBOutlet UILabel *clockDigital;  
20 @property (strong, nonatomic) IBOutlet UIButton *speak;  
21  
22 @end
```

```
1 //  
2 // ClockViewController.m  
3 // clock  
4 //  
5 // Created by Luca Finocchio on 04/11/13.  
6 // Copyright (c) 2013 GLDeV. All rights reserved.  
7 //  
8  
9 #import "ClockViewController.h"  
10 #import "AppDelegate.h"  
11 #import <AVFoundation/AVFoundation.h>  
12  
13 @implementation ClockViewController{...}  
14  
15 - (void)playAudio:(id)sender  
16 {  
17     // Imposto i parametri del sintetizzatore vocale  
18     AVSpeechSynthesizer *synth =  
19         [[AVSpeechSynthesizer alloc] init];  
20     NSDateComponents *dateComponents =  
21         [[NSCalendar currentCalendar] components:  
22             (NSHourCalendarUnit | NSMinuteCalendarUnit | NSSecondCalendarUnit)  
23             fromDate:[NSDate date]];  
24     NSInteger minutes = [dateComponents minute];  
25     NSInteger hours = [dateComponents hour];  
26     AVSpeechUtterance *utterance =  
27         [AVSpeechUtterance speechUtteranceWithString:  
28             [NSString stringWithFormat:@"Sono le ore %ld e %ld",
29             (long)hours, (long)minutes]];  
30     [utterance setRate:0.16];  
31     [utterance setVoice:[AVSpeechSynthesisVoice voiceWithLanguage:@"it-IT"]];  
32     [synth speakUtterance:utterance];  
33 }  
34  
35 @end
```

Il metodo *playAudio* consente all'applicazione di leggere l'ora utilizzando il framework AVFoundation. Gli oggetti coinvolti sono AVSpeechSynthesizer e AVSpeechUtterance, ed in particolare è stato necessario settarli a quest'ultimo alcuni parametri come la stringa da leggere, la velocità di lettura e la lingua utilizzata.

In Figura 7.8 è mostrata la View generata e controllata dalla classe appena descritta.

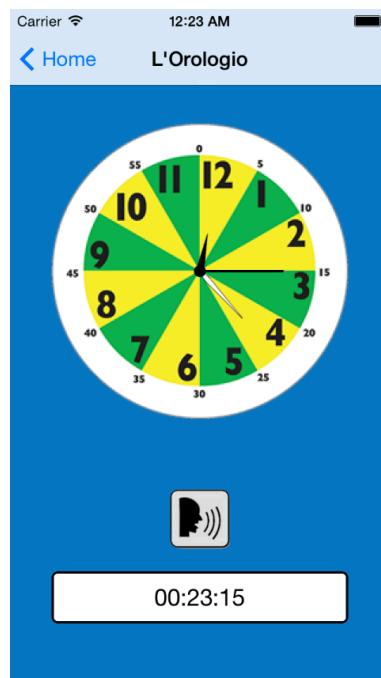


Figura 7.8: ClockViewController

- **EuroViewController**

Questa classe si occupa della visualizzazione di tutti i tagli di monete in Euro disponibili tramite una view scrollabile orizzontalmente.

```
1 //  
2 // EuroViewController.h  
3 // CasaPlus  
4 //  
5 // Created by Luca Finocchio on 23/12/13.  
6 // Copyright (c) 2013 GLDeV. All rights reserved.  
7 //  
8  
9 #import <UIKit/UIKit.h>  
10  
11 @interface EuroViewController : UIViewController <UIScrollViewDelegate>  
12  
13 @property (strong, nonatomic) IBOutlet UIScrollView *scrollView;  
14 @property (strong, nonatomic) IBOutlet UIPageControl *pageControl;  
15  
16 @end
```

```
//  
2 // EuroViewController.m  
// CasaPlus  
4 //  
// Created by Luca Finocchio on 23/12/13.  
6 // Copyright (c) 2013 GLDeV. All rights reserved.  
//  
8  
#import "EuroViewController.h"  
10 #import "EuroView.h"  
#import "EuroTableViewController.h"  
12  
@interface EuroViewController()  
14 {...}  
  
16 @property (nonatomic) float euro_value;  
  
18 @end  
  
20 @implementation EuroViewController  
  
22 -(id)initWithCoder:(NSCoder *)aDecoder  
{...}  
24  
- (void)viewDidLoad  
{  
    [super viewDidLoad];  
    self.automaticallyAdjustsScrollViewInsets = NO;  
    NSUInteger numberPages = 7;  
    self.scrollView.pagingEnabled = YES;  
    self.scrollView.contentSize =  
        CGSizeMake(CGRectGetWidth(self.scrollView.frame) * numberPages,  
        CGRectGetHeight(self.scrollView.frame));  
32
```

```
36     self.scrollView.showsHorizontalScrollIndicator = NO;
37     self.scrollView.showsVerticalScrollIndicator = NO;
38     self.scrollView.scrollsToTop = NO;
39     self.scrollView.delegate = self;
40     self.pageControl.numberOfPages = numberPages;
41     self.pageControl.currentPage = 0;
42
43     [self loadScrollViewWithPage:0];
44     [self loadScrollViewWithPage:1];
45     [self loadScrollViewWithPage:2];
46     [self loadScrollViewWithPage:3];
47     [self loadScrollViewWithPage:4];
48     [self loadScrollViewWithPage:5];
49     [self loadScrollViewWithPage:6];
50 }
51
52 #pragma mark - Manage for segue
53 ...
54
55 - (EuroView *)createViewAtIndex:(int)index
56 {
57     EuroView* euroView = [[EuroView alloc]
58         initWithFrame:self.scrollView.frame page:index];
59     [euroView.img1 addTarget:self action:@selector(toTable:)
60         forControlEvents:UIControlEventTouchUpInside];
61     [euroView.img2 addTarget:self action:@selector(toTable:)
62         forControlEvents:UIControlEventTouchUpInside];
63     [euroView.img3 addTarget:self action:@selector(toTable:)
64         forControlEvents:UIControlEventTouchUpInside];
65     return euroView;
66 }
```

```
66 - (void) toTable:(UIButton *)sender
{
    self.euro_value = sender.tag;
    [self performSegueWithIdentifier:@"euroTable" sender:self];
}
70
72 - (void)loadScrollViewWithPage:(NSUInteger)page
{
    EuroView *ev = [self createViewAtIndex:(int)page];

    if (ev.superview == nil)
    {
        CGRect frame = self.scrollView.frame;
        frame.origin.x = CGRectGetWidth(frame) * page;
        frame.origin.y = 0;
        ev.frame = frame;

        [self.scrollView addSubview:ev];
    }
}
86
88 - (void)scrollViewDidEndDecelerating:(UIScrollView *)scrollView
{
    CGFloat pageWidth = CGRectGetWidth(self.scrollView.frame);
    NSUInteger page =
        floor((self.scrollView.contentOffset.x -
               pageWidth / 2) / pageWidth) + 1;
    self.pageControl.currentPage = page;

    [self loadScrollViewWithPage:page - 1];
    [self loadScrollViewWithPage:page];
    [self loadScrollViewWithPage:page + 1];
}
98 }
```

```
100 - (void)gotoPage:(BOOL)animated
101 {
102     NSInteger page = self.pageControl.currentPage;
103
104     [self loadScrollViewWithPage:page - 1];
105     [self loadScrollViewWithPage:page];
106     [self loadScrollViewWithPage:page + 1];
107
108     CGRect bounds = self.scrollView.bounds;
109     bounds.origin.x = CGRectGetWidth(bounds) * page;
110     bounds.origin.y = 0;
111     [self.scrollView scrollRectToVisible:bounds animated:animated];
112 }
113
114 - (IBAction)changePage:(id)sender
115 {
116     [self gotoPage:YES];
117 }
118 @end
```

In Figura 7.9 è mostrata la View generata e controllata dalla classe sopra descritta.



Figura 7.9: ClockViewController

7.1.2.2 CoreLocation

CoreLocation è il Framework che serve per dialogare con il location services integrato nei dispositivi mobile apple, ovvero il chip del GPS. La classe che lo interfaccia con l'HW è la `CLLocationManager`, che informa un oggetto delegato, una volta presi i dati dal GPS. L'utilizzo di tale framework prevede l'import del framework stesso `#import <CoreLocation/CoreLocation.h>` e il setup del delegate che deve occuparsi della gestione di tutto ciò `CLLocationManagerDelegate` che si occupa di ottenere i dati sul gps e fornirli tramite dei metodi opzionali del protocollo.

La classe che implementa tale protocollo è `PrintTableViewController`

```
//  
2 // PrintTableViewController.h  
// CasaPlus  
4 //  
// Created by Luca Finocchio on 17/12/13.  
6 // Copyright (c) 2013 GLDeV. All rights reserved.  
//  
8  
#import <UIKit/UIKit.h>  
10 #import <CoreText/CoreText.h>  
#import <CoreGraphics/CoreGraphics.h>  
12 #import <CoreLocation/CoreLocation.h>  
#import "Cart.h"  
14  
@protocol sendDataProtocol <NSObject>  
16  
- (void)aggiornaCart:(Cart *)ci;  
18  
@end  
20  
22 @interface PrintTableViewController : UIViewController<UITableViewDelegate,  
UITableViewController, UIPrintInteractionControllerDelegate,  
CLLocationManagerDelegate>  
24  
26 @property (strong, nonatomic) Cart *cart;  
@property (strong, nonatomic) IBOutlet UITableView *tableView;  
@property (strong, nonatomic) IBOutlet UIBarButtonItem *cancella;  
@property (strong, nonatomic) IBOutlet UIBarButtonItem *print;  
@property (nonatomic, strong) CLLocationManager *locationManager;  
30 @property (strong, nonatomic) NSMutableArray *stores;  
32 @property(nonatomic,assign)id delegate;  
34  
@end
```

```
//  
2 // PrintTableViewController.m  
// CasaPlus  
4 //  
// Created by Luca Finocchio on 17/12/13.  
6 // Copyright (c) 2013 GLDeV. All rights reserved.  
//  
8  
9 #import "PrintTableViewController.h"  
10 #import "Product.h"  
11 #import "CartItem.h"  
12 #import "UIImageView+AFNetworking.h"  
13 #import "AFNetworking.h"  
14 #import "printTableView.h"  
15 #import "PrintElementController.h"  
16 #import "Store.h"  
17  
18 @interface PrintTableViewController()  
19 {...}  
20 @property (strong, nonatomic) UILabel *label;  
21 @end  
22  
23 @implementation PrintTableViewController  
24  
25 -(id)initWithCoder:(NSCoder *)aDecoder  
26 {...}  
27  
28 - (void)viewDidLoad  
29 {...}  
30  
31 - (void)viewDidUnload  
32 {...}  
33  
34 -(void) viewWillDisappear:(BOOL)animated  
35 {...}
```

```
36 #pragma mark - Table view data source
...
38
# pragma mark UIPrintInteractionControllerDelegate
...
40
# pragma mark - Manage for segue
...
44
# pragma mark - CLLocationManagerDelegate
46
- (void)locationManager:(CLLocationManager *)manager
    didFailWithError:(NSError *)error{}
48
50 - (void)locationManager:(CLLocationManager *)manager
    didUpdateToLocation:(CLLocation *)newLocation
    fromLocation:(CLLocation *)oldLocation
{
54     NSMutableArray *tmp = [[NSMutableArray alloc] init];
55     for (Store* store in self.stores)
56     {
57         CLLocation *location = [[CLLocation alloc]
58             initWithLatitude:store.lat longitude:store.lon];
59         CLLocationDistance distance =
60             [newLocation distanceFromLocation:location];
61         if (distance < 300)
62         {
63             [self updateWithEvent:[NSString stringWithFormat:
64                 @"Sei vicino a %@", store.nome]];
65             [tmp addObject:store];
66         }
67     }
68     [self.stores removeObjectsInArray:tmp];
69 }
```

La classe *PrintViewController* si occupa inoltre della gestione delle notifiche locali.

```

70 - (void)updateWithEvent:(NSString *)event
{
71
72     // Update the icon badge number.
73     [[UIApplication sharedApplication] applicationIconBadgeNumber++];

74
75     //Istanzia la variabile per impostare la local notification
76     UILocalNotification *notification = [[UILocalNotification alloc] init];

77
78     //Imposto il fireDate e il timezone
79     notification.fireDate = [NSDate date];
80     notification.timeZone = [NSTimeZone defaultTimeZone];

81
82     //Messaggio che verr visualizzato se la nostra applicazione spenta
83     notification.alertBody = [NSString stringWithFormat:@"%@", event];

84
85     //Nome del pulsante per avviare la nostra applicazione
86     notification.alertAction = @"Dettagli";

87
88     //Qui possiamo impostare un suono,
89     //per semplicità impostiamo il suono di default
90     notification.soundName = UILocalNotificationDefaultSoundName;

91
92     NSDictionary *infoDict = [NSDictionary
93         dictionaryWithObjectsAndKeys:event,@"event", nil];
94     notification.userInfo = infoDict;

95
96     //Imposto la local notification
97     [[UIApplication sharedApplication]
98      scheduleLocalNotification:notification];
99 }
100 @end

```

7.1.2.3 StoryBoard

La Figura 7.10 illustra il diagramma Storyboard utilizzato per prototipare e realizzare l’interfaccia grafica e l’interazione con l’applicazione da parte degli utenti. Ad ogni schermata è associato un particolare ViewController mentre le connessioni tra le varie schermate indicano le transizioni tra le stesse.

Oltre al layout per iPhone è stato realizzato anche un layout per iPad come mostra lo storyboard in Figura 7.11

Lo scambio dei dati tra i vari ViewController avviene attraverso l’implementazione del metodo

1  - `(void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender`

nel quale è possibile specificare i dati da passare da un ViewController ad un altro.

7.1.2.4 PushNotifications

Ci sono 3 componenti essenziali da implementare per gestire le notifiche push in applicazioni iOS:

- alcuni metodi dedicati alle Push Notification all’interno dell’AppDelegate.m dell’applicazione;
- un back-end su un Web Server on line con *PHP* e *MySQL*
- profili e certificati dell’applicazione, che hanno abilitata la funzione Push Notification.

Il meccanismo delle notifiche push si basa sull’infrastruttura dell’Apple Push Notification Service (nel seguito APNs). Tale infrastruttura consiste in un server Apple denominato Push Notification Server (PNS) che fa da dispatcher di tutti i messaggi di notifica inviati a tutti i dispositivi iOS.

L’APNs consente al dispositivo iOS dell’utente di restare continuamente connesso con il PNS utilizzando una connessione TCP/IP. Una volta che il meccanismo è

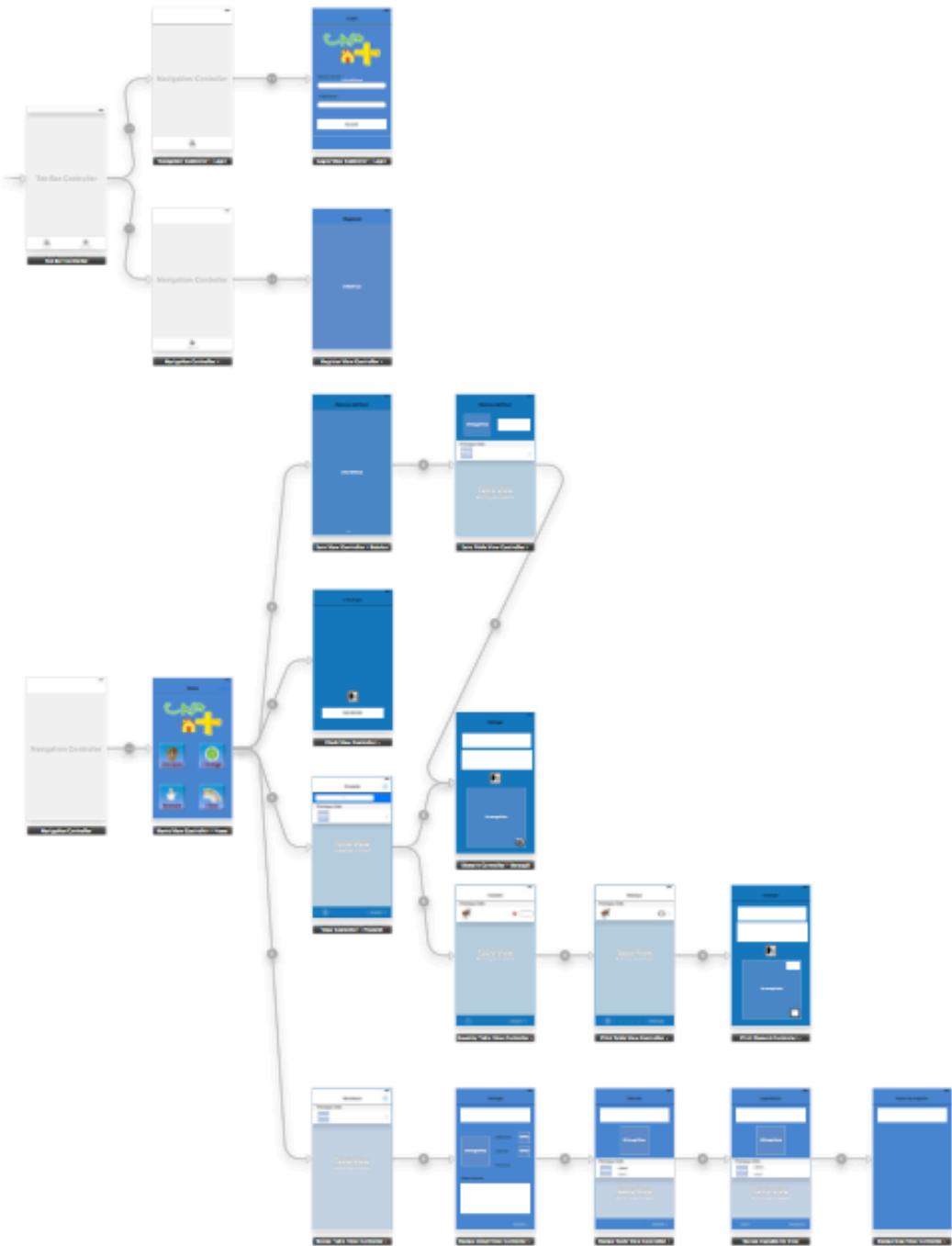


Figura 7.10: Storyboard Diagram iPhone

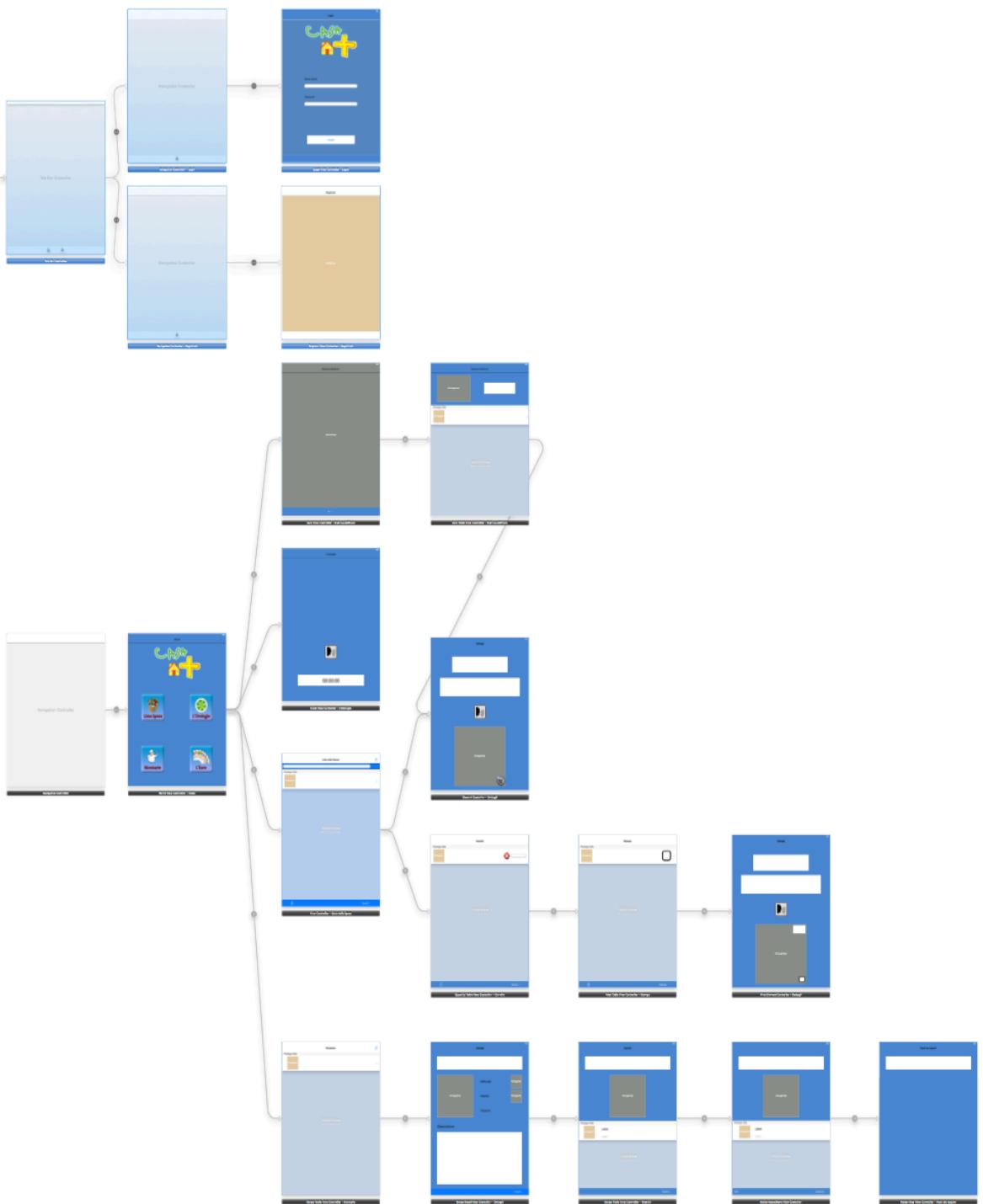


Figura 7.11: Storyboard Diagram iPad

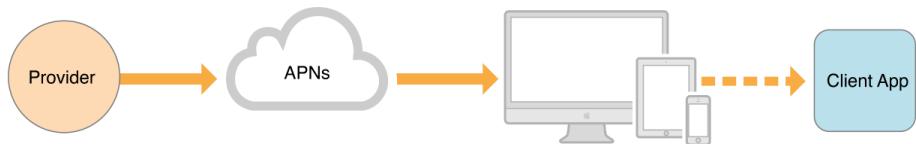


Figura 7.12: Push Notifications

funzionante, quando si desidera inviare una notifica push all'applicazione installata sul dispositivo dell'utente, è necessario contattare il servizio APNs Apple affinchè consegna il messaggio alla precisa applicazione installata sul dispositivo desiderato.

Per rendere funzionale il dialogo con l'Apple Push Notification service (APNs), è compito dello sviluppatore scrivere un'applicazione su un web server che faccia da provider delle notifiche push e che comunichi proprio con tale server Apple. L'applicazione provider invierà il messaggio al server APNs, che a sua volta recapiterà il messaggio, attraverso la connessione TCP/IP, a tutti i dispositivi che hanno l'applicazione installata e che hanno dato il consenso alla ricezione delle notifiche.

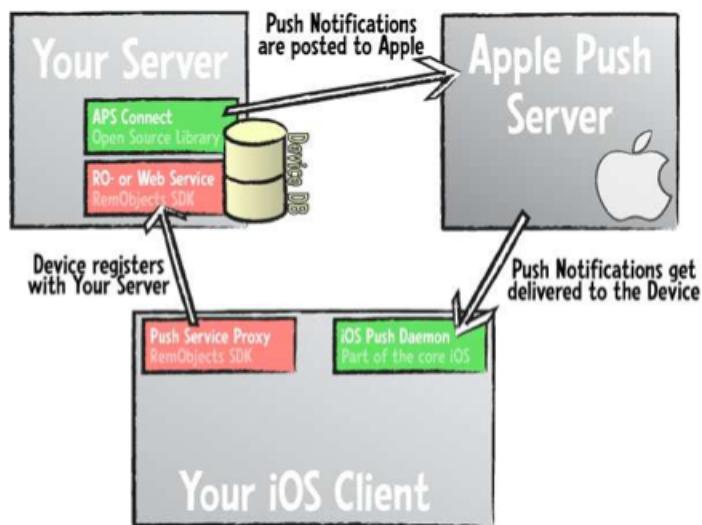


Figura 7.13: APNs-App-Server

Di seguito sono riportati i metodi implementati nell'appDelegate per la gestione delle PushNotifications sul client.

```

1 - (BOOL)application:(UIApplication *)application
2     didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
3 {
4     ...
5
6     #define UUID_USER_DEFAULTS_KEY @“UUID”
7
8     if ([defaults objectForKey:UUID_USER_DEFAULTS_KEY] == nil)
9     {
10         [defaults setObject:[[[self class] GetUUID]
11                         forKey:UUID_USER_DEFAULTS_KEY];
12         [defaults synchronize];
13     }
14
15     [defaults synchronize];
16
17     [[UIApplication sharedApplication] registerForRemoteNotificationTypes:
18      (UIRemoteNotificationTypeBadge | UIRemoteNotificationTypeSound |
19       UIRemoteNotificationTypeAlert)];
20
21
22     return YES;
23 }
24
25 - (void)application:(UIApplication*)application
26     didRegisterForRemoteNotificationsWithDeviceToken:(NSData*)deviceToken
27 {
28
29 #if !TARGET_IPHONE_SIMULATOR
30
31     // Get Bundle Info for Remote Registration
32     // (handy if you have more than one app)
33     NSString *appName = [[[NSBundle mainBundle] infoDictionary]
34                           objectForKey:@“CFBundleDisplayName”];
35     NSString *appVersion = [[[NSBundle mainBundle] infoDictionary]
36                           objectForKey:@“CFBundleVersion”];
37
38 }
```

```

34 // Check what Notifications the user has turned on.
35 NSUInteger rntypes = [[UIApplication sharedApplication]
36             enabledRemoteNotificationTypes];
37
38 // Set the defaults to disabled unless we find otherwise...
39 NSString *pushBadge = (rntypes & UIRemoteNotificationTypeBadge)
40             ? @"enabled" : @"disabled";
41 NSString *pushAlert = (rntypes & UIRemoteNotificationTypeAlert)
42             ? @"enabled" : @"disabled";
43 NSString *pushSound = (rntypes & UIRemoteNotificationTypeSound)
44             ? @"enabled" : @"disabled";
45
46 UIDevice *dev = [UIDevice currentDevice];
47 NSString *deviceUuid =
48             [defaults objectForKey:UUID_USER_DEFAULTS_KEY] ;
49 NSString *deviceName = dev.name;
50 NSString *deviceModel = dev.model;
51 NSString *deviceSystemVersion = dev.systemVersion;
52
53 // Prepare the Device Token for Registration(remove spaces and < >)
54 NSString *deviceToken1 = [[[deviceToken description]
55             stringByReplacingOccurrencesOfString:@("<" withString:@(""))
56             stringByReplacingOccurrencesOfString:@(">" withString:@(""))
57             stringByReplacingOccurrencesOfString: @" " withString: @("")];
58
59 NSString *user;
60 if ([defaults objectForKey:@"username"])
61     user = [defaults objectForKey:@"username"];
62 else
63     user = @"not_logged";

```

```

64 // Imposto i parametri della richiesta
65 AFHTTPRequestOperationManager *manager =
66     [AFHTTPRequestOperationManager manager];
67 manager.responseSerializer.acceptableContentTypes =
68     [NSSet setWithObject:@"text/html"];
69 NSDictionary *params = @{@"option": @"com_apns", @"task":
70     @"devlists.addDevice", @"appName": appName, @"appVersion":
71     appVersion, @"pushBadge": pushBadge, @"pushAlert": pushAlert,
72     @"pushSound": pushSound, @"devUid": deviceUuid, @"devName":
73     deviceName, @"devModel": deviceModel, @"devS0":
74     deviceSystemVersion, @"devToken": deviceToken1, @"user":user};
75
76 // Esegue la richiesta
77 [manager POST:[NSString stringWithFormat:@"%@",  

78     [defaults objectForKey:@"url"], @"index.php"] parameters:params  

79     success:^(AFHTTPRequestOperation *operation, id responseObject)  

80 {
81     if ([defaults objectForKey:@"token"] == nil) {  

82         [defaults setObject:deviceToken1 forKey:@"token"];  

83         [defaults synchronize];  

84     }
85 } failure:^(AFHTTPRequestOperation *operation, NSError *error) {  

86 }];
87
88 #endif
89
90 }
91
92 - (void)application:(UIApplication*)application  

93     didFailToRegisterForRemoteNotificationsWithError:(NSError*)error
94 {
95     #if !TARGET_IPHONE_SIMULATOR
96         NSLog(@"Failed to get token, error: %@", error);
97     #endif
98 }

```

```
- (void)application:(UIApplication *)application
    didReceiveRemoteNotification:(NSDictionary *)userInfo
{
#if !TARGET_IPHONE_SIMULATOR

    NSString *alertMsg;
    NSString *badge;
    NSString *sound;
    if( [[userInfo objectForKey:@"aps"] objectForKey:@"alert"] != NULL)
    {
        alertMsg = [[userInfo objectForKey:@"aps"] objectForKey:@"alert"];
    }
    else
    { alertMsg = @"{no alert message in dictionary}";
    }

    if( [[userInfo objectForKey:@"aps"] objectForKey:@"badge"] != NULL)
    {
        badge = [[userInfo objectForKey:@"aps"] objectForKey:@"badge"];
    }
    else
    { badge = @"{no badge number in dictionary}";
    }

    if( [[userInfo objectForKey:@"aps"] objectForKey:@"sound"] != NULL)
    {
        sound = [[userInfo objectForKey:@"aps"] objectForKey:@"sound"];
    }
    else
    { sound = @"{no sound in dictionary}";
    }
}
```

```
130 // Mostro un alert a seconda della richiesta
131 if (application.applicationState==UIApplicationStateActive)
132 {
133     AudioServicesPlaySystemSound (kSystemSoundID_Vibrate);
134     AudioServicesPlaySystemSound (1007);
135     NSString* alert_msg = alertMsg;
136     UIAlertView *alert;
137
138     if ([alert_msg rangeOfString:@"spesa"].location != NSNotFound) {
139         alert = [[UIAlertView alloc] initWithTitle:@"CasaPi"
140                                         message:alert_msg delegate:self
141                                         cancelButtonTitle:@"Cancella"
142                                         otherButtonTitles:@"Aggiorna", nil];
143     } else {
144         alert = [[UIAlertView alloc] initWithTitle:@"CasaPi"
145                                         message:alert_msg delegate:nil
146                                         cancelButtonTitle:@"OK"
147                                         otherButtonTitles:nil];
148     }
149     [alert show];
150 } else if ([alertMsg rangeOfString:@"spesa"].location != NSNotFound)
151 {
152     NSString* alert_msg = alertMsg;
153     UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"CasaPi"
154                                         message:alert_msg delegate:self
155                                         cancelButtonTitle:@"Cancella"
156                                         otherButtonTitles:@"Aggiorna", nil];
157     [alert show];
158 }
159 #endif
160 }
161
162 @end
```

7.1.2.5 Preferenze

La maggior parte delle app spesso espone preferenze agli utenti in modo che questi possano personalizzare l'aspetto e il comportamento delle stesse. Anche in questo caso, vogliamo che gli utenti abbiano a disposizione delle “manopole” per custumizzare l'applicazione, in particolare si vuole poter agire sui seguenti parametri:

- l'indirizzo IP del server sul quale è presente il database e che gestisce le notifiche;
- lo sfondo dell'orologio;
- se i prodotti della lista sono raggruppati per categoria o meno;
- se l'infomazione sulla categoria dei prodotti è visibile o no;
- se è possibile aggiungere un prodotto alla lista della spesa direttamente o passando per i dettagli.

La maggior parte delle preferenze vengono memorizzate in locale utilizzando il *Cocoa preferences system* noto come *user defaults system*.

Lo user defaults system è progettato per la memorizzazione di tipi di dato semplici come stringhe, numeri, date, valori booleani, URL, e cos via, in una *property list*. L'uso di una property list implica anche che è possibile organizzare le preferenze utilizzando array e dizionari. È anche possibile memorizzare altri oggetti in una property list codificandoli prima in un oggetto NSData. Un'app iOS è in grado di visualizzare le preferenze tramite un'altra app, Impostazioni, che rappresenta la soluzione ideale per collocare le preferenze che l'utente non ha bisogno di configurare frequentemente. Per visualizzare le preferenze in Impostazioni, l'applicazione deve includere una risorsa speciale chiamata *Settings bundle* che definisce le preferenze di visualizzazione, il modo corretto per visualizzarle, e le informazioni necessarie per registrare le scelte dell'utente.

Un setting bundle è banalmente chiamato Settings.bundle e risiede nella directory principale dell'app. Questo contiene uno o più *Settings page files* che descrivono le singole pagine delle preferenze. Ogni file di questo tipo viene memorizzato in formato iPhone Settings property-list, che è un formato di file strutturato come in Figura 7.14.

CasaPlus > CasaPlus > Supporting Files > Settings.bundle > Root.plist > No Selection		
Key	Type	Value
iPhone Settings Schema	Dictionary	(2 items)
Preference Items	Array	(8 items)
Item 0 (Group – GENERALE)	Dictionary	(2 items)
Item 1 (Text Field – URL)	Dictionary	(8 items)
Autocapitalization Style	String	None
Autocorrection Style	String	No Autocorrection
Default Value	String	http://192.168.1.3:8888/joomla/
Text Field Is Secure	Boolean	NO
Identifier	String	url
Keyboard Type	String	Alphabet
Title	String	URL
Type	String	Text Field
Item 2 (Group – LISTA DELLA)	Dictionary	(2 items)
Item 3 (Toggle Switch –	Dictionary	(4 items)
Item 4 (Toggle Switch –	Dictionary	(4 items)
Type	String	Toggle Switch
Title	String	Visualizza categorie
Identifier	String	pref_show_category
Default Value	Boolean	YES
Item 5 (Toggle Switch –	Dictionary	(4 items)
Item 6 (Group – OROLOGIO)	Dictionary	(2 items)
Type	String	Group
Title	String	OROLOGIO
Item 7 (Multi Value – Scegli uno	Dictionary	(6 items)
Strings Filename	String	Root

Figura 7.14: Esempio di Settings page files

Di seguito è presentato un frammento di codice utile per la gestione delle preferenze.

```

1 ...
2
3 @implementation AppDelegate{
4     NSUserDefaults *defaults;
5 }
6
7 - (BOOL)application:(UIApplication *)application
8     didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
9 {
10     defaults = [NSUserDefaults standardUserDefaults];
11     if (![defaults objectForKey:@"firstRun"])
12     {
13         [defaults setValue:[NSDate date] forKey:@"firstRun"];
14         [defaults setBool:NO forKey:@"pref_visual_grouped"];
15         [defaults setObject:@"http://localhost:8888/joomla/" forKey:@"url"];
16         [defaults setBool:YES forKey:@"pref_show_category"];
17         [defaults setBool:YES forKey:@"pref_direct_selection"];
18         [defaults setValue:@"0" forKey:@"clock_background"];
19     }
20
21 #define UUID_USER_DEFAULTS_KEY @"UUID"
22 if ([defaults objectForKey:UUID_USER_DEFAULTS_KEY] == nil)
23 {
24     [defaults setObject:[[self class] GetUUID]
25                     forKey:UUID_USER_DEFAULTS_KEY];
26     [defaults synchronize];
27 }
28
29 [defaults synchronize];
30 ...
31     return YES;
32 }
```

Infine, in Figura 7.15 è mostrata la sezione dell'app Impostazioni relativa a Casa+.



Figura 7.15: Pagine delle impostazioni di Casa+

7.2 Server

In questo sezione verrà analizzato lo sviluppo del lato server di Casa+ attraverso la descrizione del DB utilizzato e del componente Joomla! denominato *com_casaplus*.

7.2.1 Manifest file, installazione e URL base

Come spiegato nel capitolo 5, un componente Joomla! può essere visualizzato richiamando la pagina index.php e passandogli il parametro option con il nome dell'estensione che si vuole visualizzare. Quando Joomla! riceve tale parametro:

- cerca la cartella in cui è installato il componente richiesto;
- esegue il file php con lo stesso nome del componente;
- applica il template al contenuto html che ne risulta.

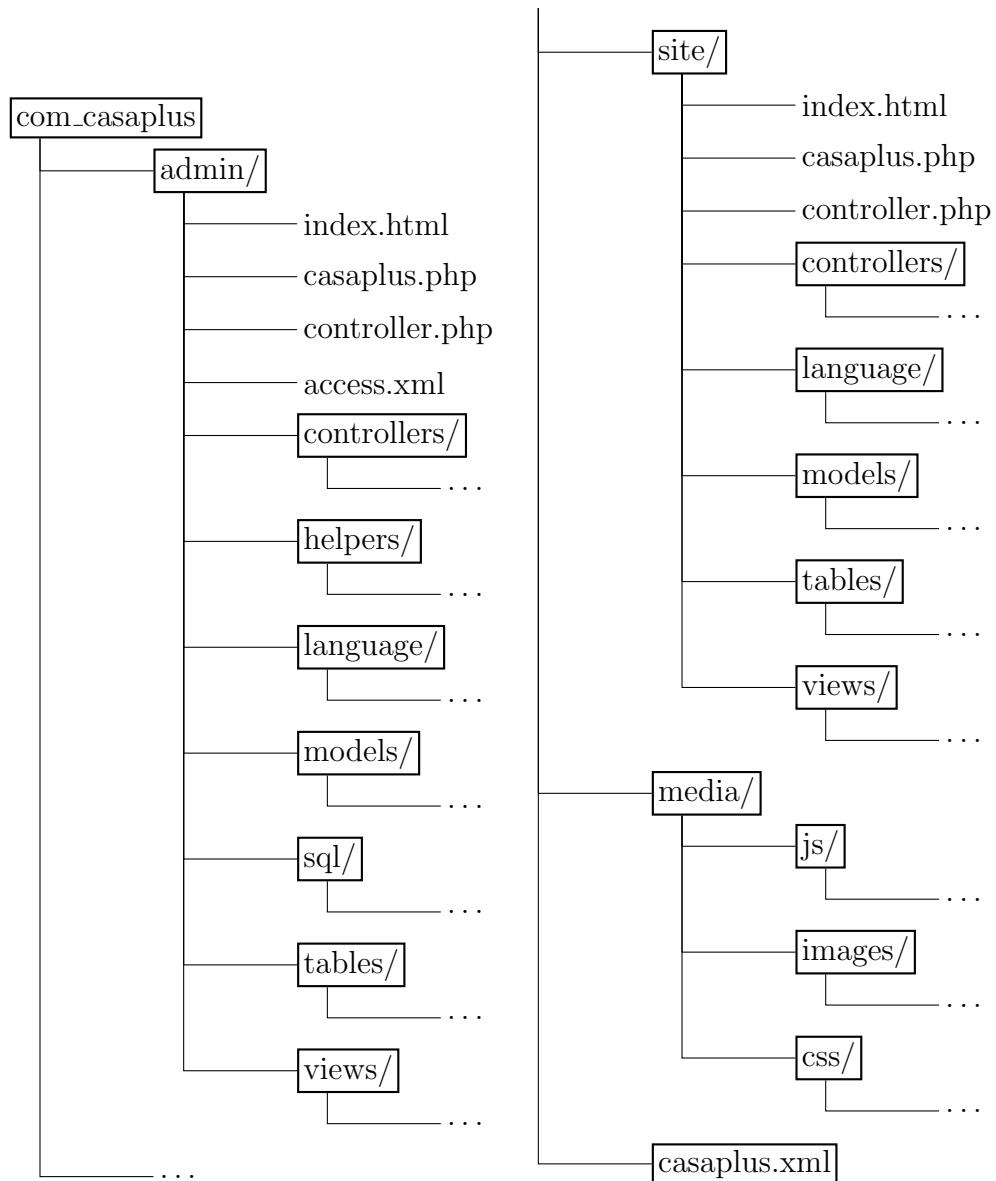
Richiamando, ad esempio, la URL */index.php?option=com_casaplus* verranno eseguite le seguenti operazioni:

- viene ricercata l'estensione nel database. Poichè option inizia con com_ si tratterà di un componente, ma questa informazione, così come il nome casaplus sono memorizzate in una tabella del database;
- viene eseguito il file /components/com_casaplus/casaplus.php;
- il risultato viene inserito nella parte principale del template della pagina;
- la pagina HTML completa viene restituita al browser.

Per la parte di amministrazione avverrà un'operazione analoga.

Leggendo le operazioni sopra descritte si nota che, ad un certo punto, il componente viene ricercato nel database. Questo fa capire come non sia sufficiente creare le cartelle e i file indicati affinchè tutto funzioni, ma si debba aggiungere delle informazioni nel database. In realtà basta aggiungere un record in una tabella, tuttavia è più semplice preparare il pacchetto di installazione del componente e lasciar fare tutto a Joomla!: installazione dei file e modifiche al database.

Si prepara dunque una cartella com_casaplus, in cui verrà sviluppato il componente, con la seguente struttura:



I file *index.html* servono esclusivamente a non far vedere il contenuto se viene direttamente richiamata una cartella.

Il file *casaplus.xml* avrà il seguente contenuto:

Ecco la spiegazione del file:

- i parametri del tag `install` indicano che si tratta di un componente e che il file xml è scritto nella versione 2.5.0 di Joomla!;

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <install type="component" version="2.5.0" method="upgrade">
3      <name>CasaPlus</name>
4      <author>Luca Finocchio</author>
5      <authorEmail>luca.finocchio@gmail.com</authorEmail>
6      <authorUrl></authorUrl>
7      <creationDate>November 2013</creationDate>
8      <version>0.0.1</version>
9      <description>COM_CASAPLUS_DESCRIPTION</description>
10     <files folder="site">
11         <filename>index.html</filename>
12         <filename>casaplus.php</filename>
13         <filename>controller.php</filename>
14         <folder>models</folder>
15         <folder>views</folder>
16         <folder>controllers</folder>
17         <folder>language</folder>
18         <folder>tables</folder>
19     </files>
20     <install folder="admin">
21         <sql>
22             <file charset="utf8" driver="mysql">
23                 sql/install.mysql.utf8.sql</file>
24             </sql>
25         </install>
```

```
26 <uninstall folder="admin">
27   <sql>
28     <file charset="utf8" driver="mysql">
29       sql/uninstall.mysql.utf8.sql</file>
30   </sql>
31 </uninstall>
32 <update>
33   <schemas>
34     <schemapath type="mysql">sql/updates</schemapath>
35   </schemas>
36 </update>
37 <media destination="com_casaplus" folder="media">
38   <folder>images</folder>
39   <folder>css</folder>
40   <folder>js</folder>
41   <folder>sound</folder>
42   <folder>mespeak</folder>
43 </media>
44 <administration>
45   <menu>COM_CASAPLUS</menu>
46   <files folder="admin">
47     <filename>index.html</filename>
48     <filename>casaplus.php</filename>
49     <filename>controller.php</filename>
50     <folder>controllers</folder>
51     <folder>language</folder>
52     <folder>models</folder>
53     <folder>sql</folder>
54     <folder>tables</folder>
55     <folder>views</folder>
56     <folder>helpers</folder>
57   </files>
58 </administration>
59 </install>
```

- il primo tag files, al di fuori del tag administration, elenca i file da copiare nella parte del sito pubblico (la cartella */components/com_casaplus*). L'attributo folder specifica che i file indicati si trovano all'interno della cartella site nel pacchetto di installazione;
- il tag install contiene un file install.sql che indica le query da eseguire al momento dell'installazione del pacchetto;
- il tag install contiene un file uninstall.sql che indica le query da eseguire al momento dell'eliminazione del pacchetto dal database di Joomla!;
- il tag update contiene dei file update.sql che indicano le query da eseguire al momento degli aggiornamenti del pacchetto;
- il tag media contiene i file immagine o css che dovranno essere aggiunti per la corretta visualizzazione dei template;
- il tag administration delimita i file da installare nella parte di amministrazione (la cartella */administrator/components/com_casaplus*);
- il tag file all'interno di administration ha lo stesso significato, ma i file saranno presi dalla cartella admin del pacchetto, come indicato dall'attributo folder;
- I tag filename indicano i file che dovranno essere contenuti direttamente nella cartella */administrator/components/com_casaplus*;
- i tag folder indicano le directory da creare all'interno della cartella principale */administrator/components/com_casaplus*; queste directory conterranno il cuore del componente, ovvero Models, Views, Controllers e Tables oltre ad eventuali file .sql che conterranno le query e ad eventuali file per modificare il linguaggio del componente.

Per installare il pacchetto su Joomla! è sufficiente zippare il tutto, ovvero la directory com_casaplus, e utilizzare il pannello amministrativo di Joomla! che gestisce in maniera automatica l'installazione di nuovi componenti.

7.2.2 Schema del DB

La seguente Figura 7.16 mostra lo schema della base dati presente sul server.

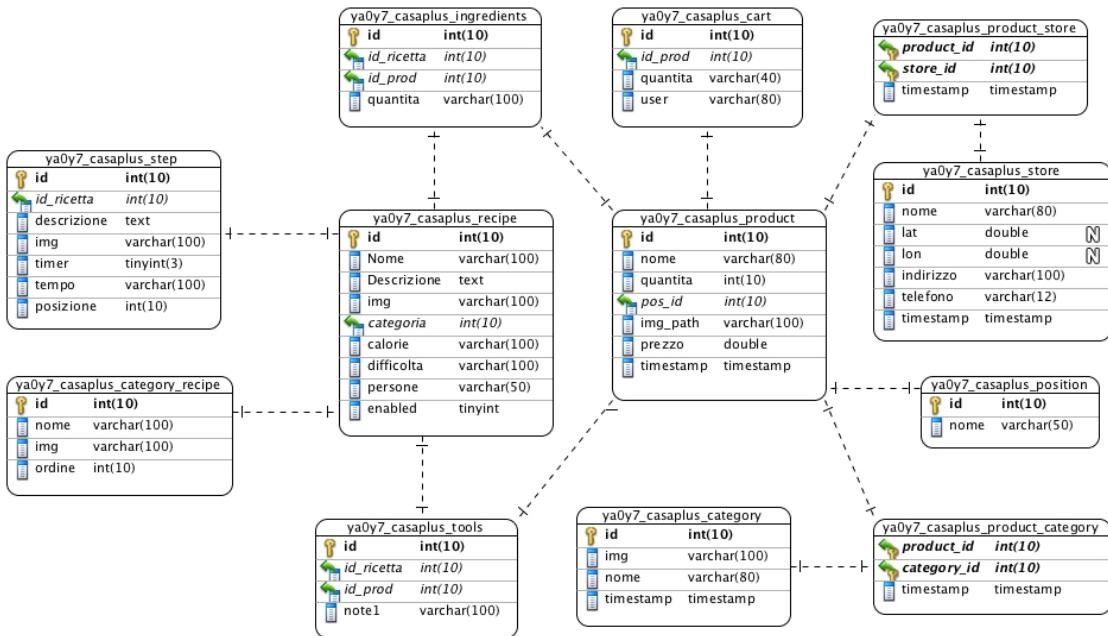


Figura 7.16: Schema del DB Casa+

I codici seguenti sono i responsabili della creazione delle tabelle e della loro cancellazione e si trovano rispettivamente in admin/sql/install.sql e admin/sql/uninstall.sql.

In fase di installazione verrà eseguita la query:

```

CREATE TABLE IF NOT EXISTS '#_casaplus_product' (
1   'id' int(11) NOT NULL auto_increment,
2   'cat_id' int(11) NOT NULL,
3   'nome' varchar(80) NOT NULL,
4   'quantita' int(11) NOT NULL,
5   'pos_id' int(11) NOT NULL,
6   'img_path' varchar(100) NOT NULL,
7   'prezzo' double NOT NULL,
8   'timestamp' timestamp NOT NULL default CURRENT_TIMESTAMP
9       on update CURRENT_TIMESTAMP,
10      PRIMARY KEY ('id')
11 ) ENGINE=MyISAM DEFAULT CHARSET=latin1 AUTO_INCREMENT=297 ;
12 ...

```

In fase di disinstallazione verrà eseguita la query:

```

1 DROP TABLE IF EXISTS '#_casaplus_product';
2 ...

```

7.2.3 I Controllers

Si procede ora allo studio della struttura del componente sviluppato, analizzando una piccola parte del lato back end del componente.

Il punto di accesso globale al back end del componente è il file casaplus.php contenuto nella cartella admin.

Viene istanziata la classe controller del componente. Il nome passato è il nome del componente stesso e costituisce il prefisso di tutte le classi che si creeranno. Tale controller, pertanto, si chiamerà CasaplusController.

Con il metodo *execute()* viene eseguita l'operazione indicata dal parametro task, recuperato tramite un apposito metodo di Joomla! che filtra anche eventuali attacchi per variabili mal formattate. Nel caso questo parametro non sia specificato viene

```

1 <?php
2
3 defined('_JEXEC') or die('Restricted access');
4 JLoader::register('CasaPlusHelper', dirname(__FILE__)
5     . DS . 'helpers' . DS . 'casaplus.php');
6
7 Jimport('joomla.application.component.controller');
8
9 $controller = JController::getInstance('Casaplus');
10 $input = JFactory::getApplication()->input;
11 $controller->execute($input->getCmd('task'));
12
13 $controller->redirect();

```

eseguita l'operazione *display()*. Dal punto di vista del codice viene chiamato il metodo *display()* del controller.

Il file del controller appena istanziato si chiamerà controller.php e anch'esso è contenuto all'interno della cartella admin.

Il metodo *display()* va a riscrivere il metodo omonimo della classe padre: esso imposterà la View di default e richiamerà il metodo *display* della classe padre, che si occuperà della visualizzazione.

Il metodo *products()* è molto simile al metodo *display*, con la differenza che non utilizzerà la View di default ma imposterà come View che si dovrà occupare della visualizzazione dei contenuti una View di nome *Products*; dopodichè richiamerà il metodo *display* della classe padre.

Si analizzano ora i due controller che gestiscono l'entità *Product* del componente. Questi sono detti controller specifici e vanno ricercati all'interno nella cartella controllers. Essi si chiameranno *CasaplusControllerProducts* e *CasaplusControllerProduct*. L'uso di due controller potrebbe risultare strano, ma la spiegazione di questo è legata al fatto che gli sviluppatori di Joomla! hanno già realizzato, come al solito, buona parte del codice necessario affinchè i controller gestiscano i dati nel database, ma hanno deciso di suddividere quelle che sono le operazioni di inserimento e modifica da altre operazioni, come quelle di cancellazione.

```

1 <?php
2
3 defined('_JEXEC') or die('Restricted access');
4 import('joomla.application.component.controller');
5
6 class CasaplusController extends JController{
7
8     public function categories($cachable = false){
9         JRequest::setVar('view',
10             JRequest::getCmd('view', 'Categories'));
11         parent::display($cachable);
12     }
13
14     public function products($cachable = false){
15         JRequest::setVar('view',
16             JRequest::getCmd('view', 'Products'));
17         parent::display($cachable);
18     }
19
20     ...
21 }
```

Il motivo di questa decisione è legato al fatto che inserimento e modifica sono legate ad un form che raccoglie i dati e agiscono sempre su un solo record per volta, mentre operazioni come la cancellazione non richiedono visualizzazioni aggiuntive e possono operare anche su più record contemporaneamente.

Per far questo Joomla! mette a disposizione le classi JControllerForm e JControllerAdmin. I seguenti file verranno creati nella cartella controllers:

- products.php

```

1 <?php
2
3 defined('_JEXEC') or die('Restricted access');
4 jimport('joomla.application.component.controlleradmin');
5
6 class CasaplusControllerProducts extends JControllerAdmin{
7
8     public function getModel($name = 'Product',
9         $prefix = 'CasaplusModel', $config = array()){
10        return parent::getModel($name, $prefix, $config);
11    }
12}

```

- product.php

```

1 <?php
2
3 defined('_JEXEC') or die('Restricted access');
4 jimport('joomla.application.component.controllerform');
5 jimport( 'joomla.application.component.helper' );
6
7 class CasaplusControllerProduct extends JControllerForm{
8
9     public function __construct($config = array())
10    {
11        $this->view_list = 'products';
12        parent::__construct($config);
13    }
14
15    public function save(){
16        ...
17    }
18}

```

Nel primo file è stata ridefinita la funzione *getModel()*. Questo è necessario in quanto il codice per la cancellazione si trova nel model che gestisce anche inserimento e modifica e che è definito nel model di tipo JModelAdmin (che si vedrà in seguito). La funzione si limita a specificare dei parametri predefiniti che chiamano il model “errato”, ovvero quello del controller CasaplusControllerProduct.

Il secondo file, invece, ridefinisce il metodo *save()*, poichè occorre del codice extra per il salvataggio dell’immagine.

7.2.4 Le Views

Le Views saranno contenute all’interno della cartella views, e ognuna di esse sarà contenuta da una cartella con lo stesso nome dell’entità a cui la view fa riferimento.

La View principale del componente sarà quella con il nome del componente stesso, si chiamerà dunque, secondo le convenzioni di Joomla, CasaplusViewCasaplus, e sarà contenuta in *views/casaplus/view.html.php*.

La classe implementa il metodo *display* per la visualizzazione dei template. Tale metodo ne richiama altri due, implementati anch’essi all’interno della classe: *setToolbar()* e *setDocument()* che servono ad impostare alcune delle proprietà della pagina che si andrà visualizzare. Il metodo *display* si occupa della visualizzazione del template associato alla view. Tale template verrà trattato più avanti.

Tornando alla trattazione dell’entità Product, si è visto che essa ha due controller. Di conseguenza avrà anche due view, ognuna associata ad un controller. La View di nome CasaplusViewProducts sarà contenuta in *views/products/view.html.php* mentre quella di nome CasaplusViewProduct in *views/product/view.html.php*.

```
1 <?php
2
3 defined('_JEXEC') or die('Restricted access');
4 jimport( 'joomla.application.component.view' );
5
6 class CasaplusViewCasaplus extends JView
7 {
8     function display($tpl = null){
9         JHtml::_('behavior.framework');
10        JHtml::stylesheet('com_casaplus/admin.stylesheet.css',
11                         array(), true);
12
13        $this->addToolBar();
14        $this->setDocument();
15        $this->items = $this->get('Items');
16
17        parent::display($tpl);
18    }
19
20    protected function addToolBar($total=null){
21        JToolBarHelper::title(
22            JText::_('COM_CASAPLUS_MAINTENANCE_MANAGER'),
23            'maintenance_manager'
24        );
25
26        function setDocument(){
27            $document = JFactory::getDocument();
28            $document->addStyleDeclaration(
29                '.icon-48-maintenance_manager {background-image:
30                    url(..../media/com_casaplus/images/icon-48-cpanel.png);
31                }'
32            );
33        }
34    }
35}
```

7.2.5 I Models

In base a quanto spiegato della struttura MVC ci aspettiamo che nel model sia presente tutto il codice necessario a recuperare i dati da una determinata sorgente (lo stesso database Joomla!), in modo che tali dati possano essere disponibili per la View.

Questo è esattamente ciò che avviene, tuttavia, se parliamo del database MySQL già utilizzato da Joomla! per la sua gestione standard, tutto il codice è già stato implementato in una classe che deriva dal *JModel* base. Tale classe è **JModelList** e, come si può dedurre dal nome, contiene il codice necessario a visualizzare una lista di elementi prelevandoli da una tabella del database MySQL. Il Model che si occupa del singolo elemento, viceversa, è **JModelAdmin**, che contiene il codice necessario per la creazione o la modifica di record.

Si analizzano ora i due Model CasaplusModelProducts e CasaplusModelProduct, che ereditano rispettivamente da JModelList e JModelAdmin. Tali Model saranno contenuti in *models/products.php* e *models/product.php*.

- CasaplusModelProducts

Viene definita esclusivamente una funzione che restituisce un oggetto di tipo JDatabaseQuery per cui sono già state impostate le informazioni di base per costruire la query, ovvero l'elenco dei campi di select e il nome della tabella da cui prelevarli.

```

1 <?php
2 defined('JEXEC') or die();
3 jimport( 'joomla.application.component.modellist' );
4
5 class CasaplusModelProducts extends JModelList{
6     ...
7     function getListQuery(){
8         $db = JFactory::getDBO();
9         $query = $db->getQuery(true);
10        $query->select('p1.id, p1.nome as nome, p2.nome as posizione
11                      , p1.quantita as quantita, p1.img_path as img,
12                      p1.prezzo as prezzo');
13        $query->from('#__casaplus_product AS p1,
14                      #__casaplus_position AS p2');
15        $query->where('p1.pos_id = p2.id');
16        $query->order($this->getState('list.ordering', 'p1.id') .
17                      ' ' . $this->getState('list.direction', 'ASC'));
18        return $query;
19    }
20    ...
21}

```

- CasaplusModelProduct

In esso abbiamo è definita la funzione *getForm()*, in quanto si tratta di un metodo astratto e la sua assenza causerebbe un errore. Questo metodo è presente poichè il Model eredita da JModelAdmin, classe che implementa i metodi di inserimento e cancellazione dal database, che a sua volta eredita da JModelForm. Questa classe mette a disposizione dei metodi che consentono di definire i campi da visualizzare nel form tramite un apposito file XML.

```

1 <?php
2 defined('_JEXEC') or die();
3 jimport( 'joomla.application.component.modeladmin' );
4
5 class CasaplusModelProduct extends JModelAdmin{
6
7     public function getForm($data = array(), $loadData = true){
8         $form = $this->loadForm('com_casaplus.product', 'product',
9             array('control' => 'jform', 'load_data' => $loadData));
10        if (!$form){
11            return false;}
12        else{
13            return $form;}
14    }
15
16    public function loadFormData(){
17        $data = $this->getItem();
18        return $data;
19    }
20
21 }

```

Questa possibilità viene completata con funzionalità aggiuntive che prevedono la creazione di campi personalizzati e la validazione dei dati inseriti nel form.

Non viene definito altro, ma il controller si aspetta di trovare una classe di tipo *JTable* il cui nome, questa volta contrariamente al solito, è TableProduct, in quanto il model si chiama Product. Questa classe viene cercata nel file product.php presente nella cartella tables.

7.2.6 La Table

All'interno della cartella models, viene creata la cartella tables e, al suo interno il file product.php con il seguente contenuto.

```

1 <?php
2
3 defined('_JEXEC') or die('Restricted access');
4
5 class TableProduct extends JTable{
6     function __construct( &$amp;db ) {
7         parent::__construct('#__casaplus_product', 'id', $db);
8     }
9 }
```

L'unica operazione che viene fatta è riscrivere il costruttore per passare il nome della tabella e quello della chiave primaria in modo che tutte le query siano composte in modo corretto.

7.2.7 Internazionalizzazione

Nei file precedentemente analizzati, ad esempio nel manifest e nelle views, è possibile notare:

- l'utilizzo della classe **JText**;
- alcune costanti proprie del componente iniziano con COM_CASAPLUS;
- alcune costanti generali definite da Joomla! e che vengono utilizzate per uniformità.

Con le costanti sono a posto è possibile creare i file di traduzione nel posto giusto: viene creata, all'interno della cartella admin, la cartella language e, al suo interno la cartella it-IT. Al suo interno viene inserito il file *it-IT.com_casaplus.ini* con il seguente contenuto:

```
1 COM_CASAPLUS="CasaPiu'"  
2  
3 COM_CASAPLUS_PRODUCT_NAME = "Nome"  
4 COM_CASAPLUS_PRODUCT_QUANTITY="Quantita'"  
5 COM_CASAPLUS_PRODUCT_PRICE="Prezzo"  
6 COM_CASAPLUS_PRODUCT_POSITION="Posizione"  
7 COM_CASAPLUS_PRODUCT_IMG_PATH="Immagine"  
8 COM_CASAPLUS_PRODUCT_STORE = "Negozi"
```

Joomla!, durante l'installazione del componente, non utilizza il file precedente, ma ne vuole uno apposta che utilizzerà per tradurre le stringhe utilizzate nel file manifest: tale file è *it-IT.com-maintenance.sys.ini* che verrà posizionato sempre nella stessa cartella.

Per trasformate le informazioni visualizzate, ad esempio, in inglese, basterà creare altri file come i precedenti, aggiungendo la cartella en-EN all'interno della cartella language e impostare, dal pannello amministrativo di Joomla!, come lingua di back end l'inglese.

7.2.8 Layout

Per lo sviluppo del layout si è scelta una veste grafica semplice e immediata. I template si occupano della generazione del codice html vero e proprio. Di seguito vengono riportati alcuni screenshot per meglio comprendere il lavoro svolto.

Capitolo 7

The screenshot displays two windows from a Joomla backend administration interface:

- Gestione Dispensa - Prodotti**: A grid view of products. Each row contains a checkbox, the product name, its category, its position, an image thumbnail, quantity, price, and a store selection dropdown.
- Modifica Prodotto**: A detailed edit form for the product "Riso". It includes fields for Name (Riso), Position (Dispensa Sportello1), Quantity (1), Price (1.6), and an image upload field (empty). On the right, there are lists for Category (checkboxes for PASTA, PER APPARECCHIARE, UTENSILI DA CUCINA, etc.) and Store (checkboxes for GALLUCCI, GLOBO CALZATURE, L'AQUILONE).

Figura 7.17: Esempio di backend

The screenshot shows the Joomla frontend with the following components:

- Header**: Shows the Joomla logo and Open Source Content Management text.
- Left Sidebar**: Includes a "Menu Principale" with links like Home, L'orologio, Euro, etc., and a "Login Form" with fields for Nome utente, Password, Ricordami, and Log in.
- Content Area**:
 - L'Orologio**: A digital clock displaying 16:02:00.
 - Altri sfondi**: A circular background image labeled "Manda Verde-Giallo".
 - On/Off Orologio Digitale**: A button to toggle the digital clock.
 - Lista della Spesa**: A table for a shopping list with columns for Immagine, Prodotto, Quantità, and Elimina. It lists items like SUCCO DI FRUTTA, SALSICCE FRESCHE, and PATATE.
 - Crea lista della spesa**: A button to create a new shopping list.
 - Lista di tutti i prodotti**: A link to view all products.
 - VERDURA**: A section showing a thumbnail of vegetables and a list of items: PATATE, ZUCCHINI, CAROTE, and RICOTTA.

Figura 7.18: Esempio di frontend

7.3 Push Notifications Server

Il sistema Casa+ deve essere in grado di inviare una notifica ai dispositivi in caso di modifica, da parte dell'applicazione desktop, della lista dei spesa. In questo modo è possibile notificare all'utente, ovunque esso si trovi purchè connesso alla rete, che sono state effettuate delle modifiche alla lista dei prodotti. Nonchè, deve essere notificato all'utente quando si è in prossimità di un negozio che vende un qualsiasi prodotto presente nella lista.

In conformità con i discorsi fatti in precedenza per l'applicativo desktop, anche questa parte di server responsabile della creazione e della gestione delle notifiche push, è stata implementata come componente di Joomla!. Come risultato, è stato ottenuto un componente completamente indipendente per la gestione delle notifiche push di apple e di conseguenza, riusabile in qualsiasi altro sistema. Per estendere ulteriormente il riuso del componente, questo è stato sviluppato non solo per Joomla! 2.5 ma anche per la versione successiva del CMS, la 3.

Nello specifico, le funzionalità messe a disposizione dal componente, denominato *com_apns* sono:

- salvataggio del *device token* e di altre informazioni generali dei dispositivi
- modifica delle impostazioni sui certificati
 - tipo di certificato da utilizzare (production o developer)
 - caricamento dei certificati
 - password per la cifratura dei certificati
- invio delle notifiche via frontend del componente
- invio delle notifiche via richiesta http-post

Di seguito verrà riportato un frammento di codice, relativo alla versione 2.5, responsabile dell'invio delle notifiche a tutti i dispositivi registrati.

```
1 <?php
2
3 defined('_JEXEC') or die('Restricted access');
4 jimport('joomla.application.component.controlleradmin');
5
6 class ApnsControllerSend extends JControllerAdmin{
7     ...
8     // manda una notifica a tutti i dispositivi registrati
9     // tramite la relativa interfaccia web
10    public function sendtoall()
11    {
12        $model = $this->getModel();
13        $items = $model->getAllDevices();
14        $tmp = $model->getCertificate();
15        $cer = $tmp[0];
16
17        $num_rows = count($items);
18        if($num_rows>0){
19
20            if ($cer->certificato == 'Developer')
21                $passphrase = $cer->password;
22            else
23                $passphrase = $cer->passwordPro;
24
25            if (isset($_POST['textArea'])){
26                $message = $_POST['textArea'];
27            } else {
28                $message = 'Nuova notifica';
29            }
30        }
31    }
32}
```

```

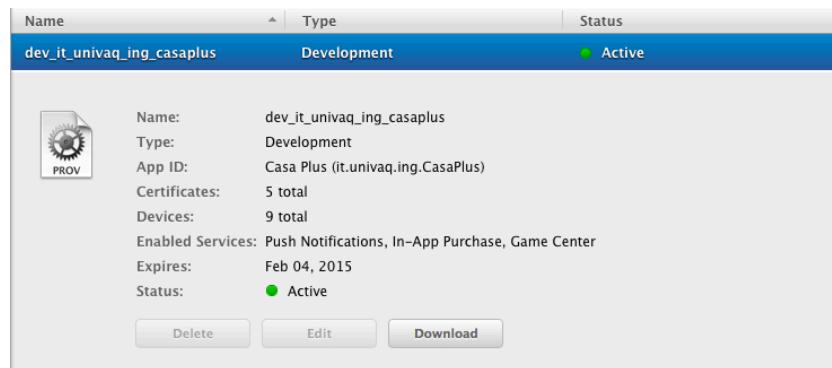
30     $ctx = stream_context_create();
31
32     if ($cer->certificato == 'Developer' &&
33         $cer->developer != ""){
34         stream_context_set_option($ctx, 'ssl', 'local_cert',
35             'media/com_apns/certificates/dev/'.$cer->developer);
36     }else if ($cer->certificato == 'Production' &&
37         $cer->production != ""){
38         stream_context_set_option($ctx, 'ssl', 'local_cert',
39             'media/com_apns/certificates/prod/'.$cer->production);
40     }else {
41         JFactory::getApplication()->enqueueMessage('Seleziona o
42             carica un certificato', 'error');
43         $this->setRedirect('index.php?option=com_apns&
44             task=sendtoall');
45         $this->redirect();
46     }
47
48     stream_context_set_option($ctx, 'ssl', 'passphrase',
49         $passphrase);
50
51     $i=0;
52     do {
53         $i++;
54         if ($cer->certificato == 'Developer'){
55             $fp = stream_socket_client(
56                 'ssl://gateway.sandbox.push.apple.com:2195', $err,
57                 $errstr, 60, STREAM_CLIENT_CONNECT|
58                 STREAM_CLIENT_PERSISTENT, $ctx);
59         }else {
60             $fp = stream_socket_client(
61                 'ssl://gateway.push.apple.com:2195', $err,
62                 $errstr, 60, STREAM_CLIENT_CONNECT|
63                 STREAM_CLIENT_PERSISTENT, $ctx);
64         }
65     }while (!$fp || $i==5);

```

```

64      $result = false;
65      foreach ($items as &$row){
66          $body['aps'] = array(
67              'alert' => $message,
68              'sound' => 'default',
69              'badge' => $row->badgeCount+1 );
70
71          $payload = json_encode($body);
72          $msg = chr(0) . pack('n', 32) . pack('H*', $row->devToken) .
73                  pack('n', strlen($payload)) . $payload;
74          $result = fwrite($fp, $msg, strlen($msg));
75          $model->updateBadge($row->devToken, $row->badgeCount+1);
76      }
77
78      $application = JFactory::getApplication();
79      if (!$result){
80          $application->enqueueMessage(
81              JText::_('COM_APNS_ERROR_MESSAGE'), 'error');
82      } else {
83          fclose($fp);
84          $application->enqueueMessage(
85              JText::_('COM_APNS_MESSAGE'));
86      }
87
88      $this->setRedirect('index.php?option=com_apns&task=sendtoall');
89      $this->redirect();
90  } else {
91      JFactory::getApplication()->enqueueMessage(
92          JText::_('COM_APNS_NO_DEVICE_MESSAGE'), 'error');
93      $this->setRedirect('index.php?option=com_apns&task=sendtoall');
94      $this->redirect();
95  }
96 }
97 ...
98 }
```

Osserviamo infine che la gestione delle notifiche push è subordinata alla creazione e gestione di certificati che permettono ai server Apple di riconoscere e validare l'identità dello sviluppatore che ha implementato la gestione delle notifiche push nella propria applicazione. Tali certificati sono richiedibili ad Apple attraverso il portale iOS Provisioning Portal¹.



Questi però non sono sufficienti, prima di poter essere effettivamente utilizzati, i certificati deve essere cifrati con un particolare procedimento, che non sarà descritto nel proseguo di questa trattazione, ma è possibile visionare al seguente link Wikibooks home. Al termine del procedimento sarà ottenuto un file con estensione .pem.

Nella Figura 7.3 è possibile vedere come il componente realizzato permetta il caricamento di questo file.

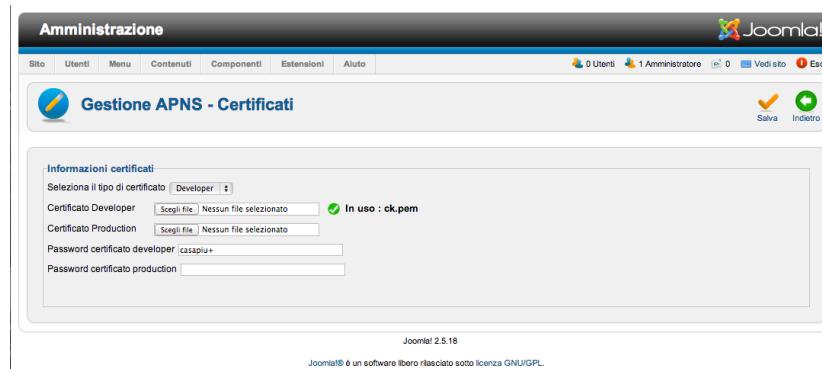


Figura 7.19: Caricamento del file ck.pem

¹<https://developer.apple.com/>

In Figura 7.20 e Figura 7.21 vengono mostrate rispettivamente, un esempio di ricezione di una notifica push dell'applicazione Casa+ e degli screenshot del componente.

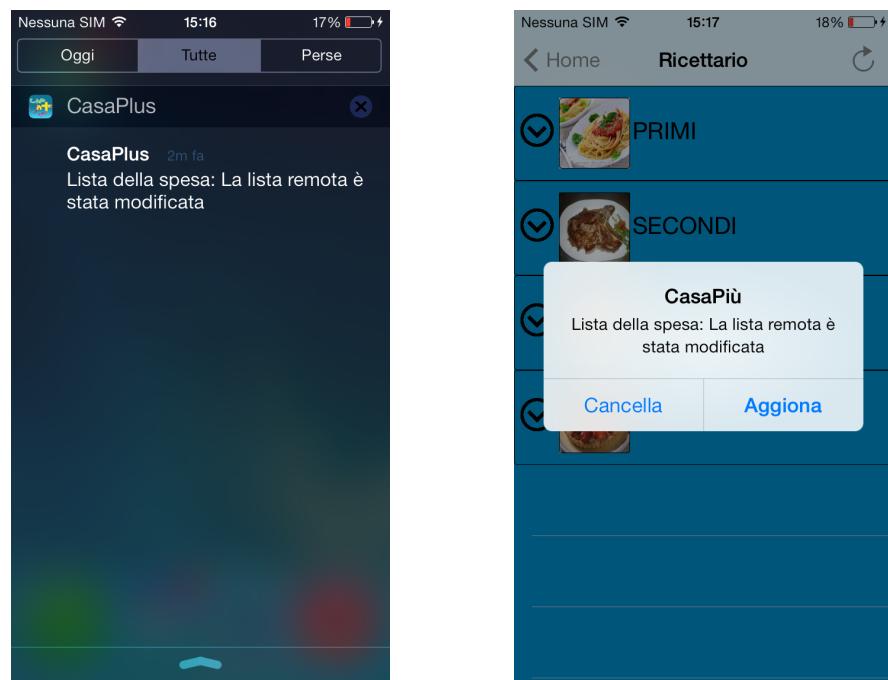


Figura 7.20: Esempio di ricezione di una notifica push

Figura 7.21: Backend e frontend del componente com_apns

Capitolo 8

Conclusioni e sviluppi futuri

Casa+, come detto all'inizio di questo lavoro di tesi, è un importante progetto concepito ed organizzato al fine di vericare le reali possibilità e capacità di vivere in autonomia delle persone con disabilità cognitiva. Al centro del progetto c'è l'utilizzo di nuove soluzioni tecnologiche al fine di sviluppare una certa indipendenza nella gestione e nello svolgimento delle azioni quotidiane. A tale proposito è stato creato un sistema mobile/web con lo scopo di gestire alcune di queste routine giornaliere quali gestire una lista della spesa, consultare un ricettario, consultare l'ora e accrescere la propria consapevolezza sul valore del denaro.

Il sistema è stato realizzato tramite un'applicazione iOS, che implementa le funzionalità sopraelencate presenti anche in un'applicativo web. Quest'ultimo è stato realizzato tramite CMS, in particolare Joomla!, partendo dalle funzionalità e dai contenuti presenti nel portale casa+ preesistente.

Quello di casa+ è un progetto in fase di espansione, dalla sola provincia dell'Aquila, ad esempio, si è già iniziato a lavorare a Roma, con più di centro ragazzi. Per questo motivo è stata fondamentale una ristrutturazione del portale facendo migrare tutto su Joomla!. In questo modo si è raggiunto un grado di modularità ed espandibilità incredibilmente elevato.

Basti pensare che l'installazione dell'applicativo web si è ridotta in questo modo al caricamento di due componenti su Joomla!, operazione semplice e veloce da eseguire anche per chi non è esperto di tecnologie web. Non solo, aggiungere nuove funzionalità non richiederà di mettere le mani ne sul database esistente né tanto

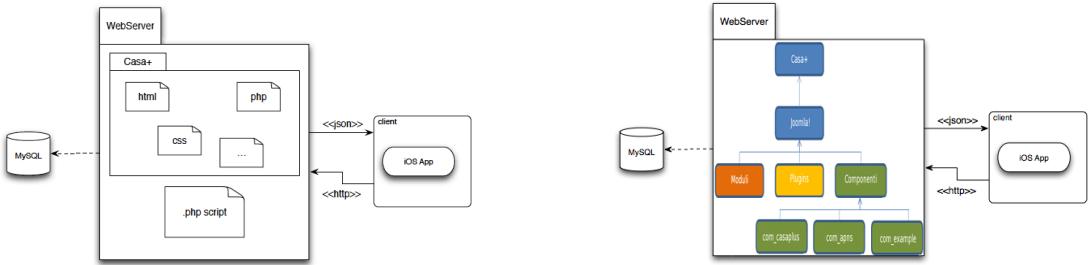
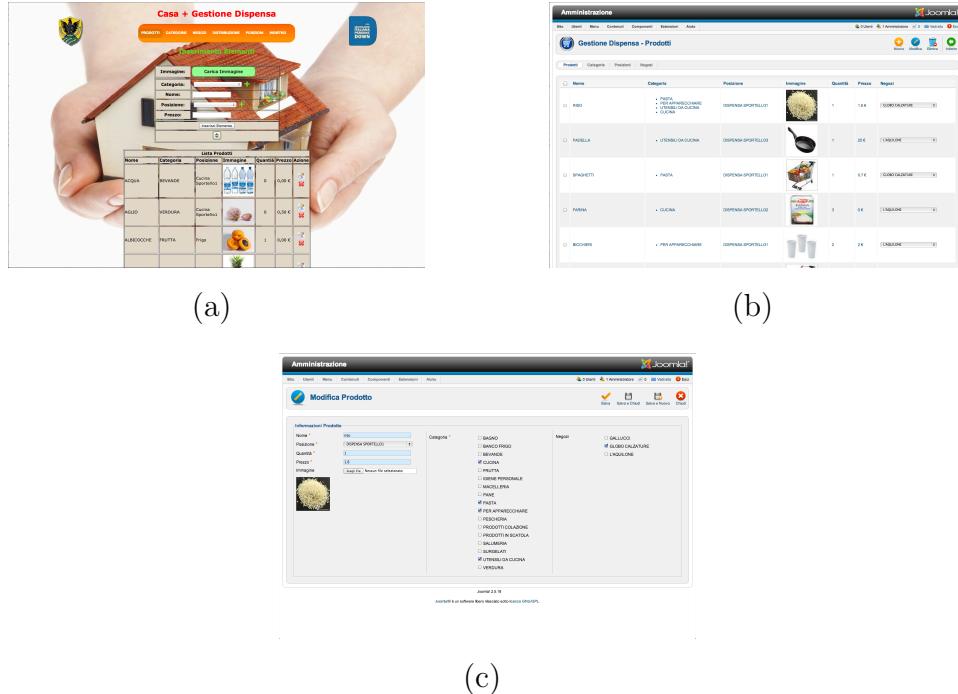


Figura 8.1: Vecchia e nuova architettura a confronto

meno sui le presenti sul server: anche in questo caso il tutto si riduce alla creazione di un componente. Tale processo favorisce anche leventuale aggiornamento degli applicativi poichè basterà distribuire a tutti i pacchetti contenenti le nuove versioni.

Mettendo per un attimo da parte questi aspetti architettonici, la ristrutturazione del server ha coinvolto anche alcuni importanti grafici, che hanno portato l'applicativo desktop ad avere una nuova veste più pulita e funzionale. Di seguito sono riportate tre figure prese rispettivamente dal vecchio (Figura 8.2(a)) e dal nuovo backend (Figura 8.2(b) e Figura 8.2(c)).

Possiamo notare subito come la visualizzazione dei prodotti sia molto più nitida e chiara e come sia più intuitivo e funzionale eliminare o modificare prodotti. Inoltre per l'inserimento e la modifica dei prodotti stessi, si è creato un layout a parte, a differenza del precedente che unisce le varie funzionalità.



Anche a livello di front end sono state apportate delle migliorie come è possibile apprezzare dalle immagini in Figura 8.3 e Figura 8.4, che mostrano la finestra dei dettagli della stessa ricetta e l'inserimento di prodotti nella lista della spesa

In conclusione, il progetto casa+ ha da offrire parecchi spunti per quanto riguarda eventuali sviluppi futuri, quelli che risultano più evidenti sono le spensione dell'applicativo web tramite la realizzazione di nuovi componenti e l'implementazione di nuove applicazioni che sfruttino al meglio tutte le innovazioni e i progressi che la tecnologia sta compiendo nel campo del mobile. Rimandando vicini al lavoro svolto, invece, due possibili sviluppi sono uno la ristrutturazione del layout per iPad mentre l'altro consisterebbe nel posticipare il caricamento delle immagini del componente com_casaplus una volta completata l'installazione di quest'ultimo, in modo da rendere questa operazione molto più veloce e performante.

Capitolo 8

The screenshot shows a Joomla! CMS page with a blue header featuring the Joomla! logo and the text "We do Joomla!". Below the header is a menu titled "Menu Principale" with links to Home, Euro, Orologio, Ricettario, lista della spesa, invia a selezionati, and invia.

The main content area displays a recipe card for "AMATRICIANA per 5 persone". The card includes a title, a "Difficoltà" icon (easy), a "Cucina" icon (Italian), and a "Categorie" icon (Food). The ingredients listed are:

Ingredienti per 5 persone		
	PANCETTA	1
	PARMIGIANO GRATUGLIATO	1
	OILIO	2 CUCCHIAI
	SALE FINO	3 CUCCHIANI
	SALE GROSSO	3 CUCCHIAI
	SPAGHETTI	1 PACCO
	PASSATA DI POMODORO	1 BOTTIGLIA

At the bottom of the card are three buttons: "Crea lista Spesa" (Create shopping list), "Iniziamo!" (Let's start!), and another "Iniziamo!" button.

To the right of the card is a sidebar titled "AMATRICIANA per 5 persone" with sections for "Difficoltà" (Easy) and "Categorie" (Food). It also features a "Crea lista della spesa" button.

The footer contains a "Lista degli ingredienti" table with the following data:

	Nome	Quantità
	PANCETTA	1
	PASSATA DI POMODORO	1 BOTTIGLIA
	PARMIGIANO GRATUGLIATO	1
	OILIO	2 CUCCHIAI
	SALE FINO	2 CUCCHIAINI
	SALE GROSSO	2 CUCCHIAI
	SPAGHETTI	1 PACCO

Figura 8.3: Confronto tra vecchia e nuova interfaccia: dettagli di una ricetta

The screenshot shows a Joomla! CMS page with a blue header featuring the Joomla! logo and the text "We do Joomla!". Below the header is a menu titled "Menu Principale" with links to Home, Euro, Orologio, Ricettario, lista della spesa, invia a selezionati, and invia.

The main content area displays a shopping list titled "Lista della Spesa". The list includes two items:

Immagine	Prodotto	Quantità	Elimina
	ARISTA DI MAIALE	<input type="text"/>	
	THE FREDDO	<input type="text"/>	

At the bottom of the list is a "Crea lista della spesa" button.

To the right of the list is a sidebar titled "Lista di tutti i prodotti" with categories: VERDURA, MACELLERIA, SALUMERIA, PESCHERIA, SALMONE, BEVANDE, and BEVANDE. Each category has a checkbox next to it.

The footer contains a "Lista degli ingredienti" table with the following data:

	Nome	Quantità
	ARISTA DI MAIALE	1
	THE FREDDO	1
	VERDURA	1
	MACELLERIA	1
	SALUMERIA	1
	PESCHERIA	1
	SALMONE	3.5 €
	BEVANDE	1

Figura 8.4: Confronto tra vecchia e nuova interfaccia: lista della spesa

Bibliografia

- [1] Gamma, E., Helm, R., Johnson, R. e Vlissides, J., *Design Patterns - Elementi per il riuso di software ad oggetti*, Pearson Education Italia, 2002 ISBN 887192150X
- [2] Joomla! Developer Network, <http://www.joomla.org/>
- [3] Developing a MVC Component, <http://docs.joomla.org/>
- [4] iOS Developer Library, <https://developer.apple.com/devcenter/ios/index.action>
- [5] Apple Push Notification Services, <http://www.raywenderlich.com/32960/apple-push-notification-services-in-ios-6-tutorial-part-1>
- [6] R. Alesii, F. Graziosi, S. Marchesani, C. Rinaldi, M. Santic, F. Tarquini, *Short range wireless solutions enabling ambient assisted living to support people affected by the Down syndrome*, Center of Excellence DEWS, University of L'Aquila, L'Aquila, Italy