Università degli studi
di Modena e Reggio Emilia

Dipartimento di Scienze Fisiche,
Informatiche e Matematiche
Corso di Laurea in Informatica

Progetto e sviluppo in Swift di un'applicazione ludica per dispositivi mobili basata su dati semantico-lessicali

Diana Sighinolfi

Tesi di Laurea

Relatore:
Ing. Riccardo Martoglia

Anno Accademico 2014-2015

Ringraziamenti

Ringrazio l'Ing. Riccardo Martoglia per la disponibilità e la pazienza dimostrata durante questi mesi di lavoro.
Un ringraziamento particolare a Rossano ed Anna che mi hanno sostenuto ed aiutato costantemente.
Ringrazio anche Serena, Mirko e Fabrizio per i consigli ed i suggerimenti relativi alla stesura di questa tesi di laurea.
Infine ringrazio i miei genitori che hanno creduto in me ed hanno sempre appoggiato le mie scelte.

Parole chiave

Apple
Swift
Game design
Codice mobile
Analisi semantica

Indice

Introduzione **1**

I Stato dell'arte **3**

1 Introduzione al linguaggio Swift **5**
1.1 Caratteristiche del linguaggio 5
1.1.1 Tipi di dato 6
1.1.2 Cicli 7
1.1.3 Statement 7
1.1.4 Funzioni 7
1.2 Strumenti di sviluppo 9
1.2.1 Xcode 9
1.2.2 Playground 10
1.2.3 Gestione della memoria 10

2 Analisi semantica del testo **13**
2.1 WordNet 13
2.1.1 La matrice lessicale 15
2.2 Le relazioni 16
2.2.1 Relazioni lessicali 16
2.2.2 Relazioni semantiche 16
2.3 MultiWordNet 18

II L'applicazione **21**

3 Progettazione **23**
3.1 Analisi del gioco 23
3.1.1 Meccaniche del gioco 24

3.2 Schemi di gioco 25

4 Implementazione **27**
4.1 Processamento dei dati 28
4.2 Algoritmo di generazione schemi 30
4.2.1 Pseudocodice 31
4.2.2 Interrogare il database 32
4.3 Scheletro dell'applicazione 34
4.4 Esempio di partita 35
4.5 Playtesting 40
4.5.1 Esempi 40

5 Conclusione **43**
5.1 Sviluppi futuri 44

Acronimi **45**

Bibliografia **47**

Introduzione

La multinazionale americana Apple Inc., fondata 39 anni or sono, è sempre stata un'azienda all'avanguardia per quanto riguarda le tecnologie sviluppate al suo interno e pertanto si è affermata sul mercato nazionale tramite prodotti di lusso come iPhone, iPad, iPod e Mac.

Nonostante il colosso statunitense faccia parlare di sé ogni 6 mesi per il rilascio di nuovi modelli disponibili, la rivoluzione che ha scatenato un notevole aumento di programmatori Apple è stata il rilascio di un nuovo linguaggio di programmazione, *Swift*. L'obiettivo dell'azienda di Cupertino è infatti quello di far affermare, nel minor tempo possibile, il linguaggio tra i programmatori, in modo da soppiantare completamente Objective-C, finora l'unico linguaggio di programmazione usato per sviluppare le applicazioni per i dispositivi Apple.

Swift si è subito conquistato il primo posto tra i programmatori a tempo pieno di applicazioni per iOS e OS X, abituati alla sintassi dell'Objective-C, riuscendo anche ad entrare nella Top 10 dei linguaggi più utilizzati nel 2015. È stato introdotto nei corsi di programmazione nella maggior parte delle università statunitensi e in alcuni atenei italiani perché considerato il linguaggio più promettente dei prossimi decenni.

Si è pertanto deciso, attraverso questa Tesi di Laurea, di approfondire lo studio del nuovo linguaggio di programmazione tramite lo sviluppo di un'applicazione iOS.

In particolare, nello sviluppo dell'applicazione, si sono approfondite tecniche di elaborazione semantica dei dati attraverso l'utilizzo di una collezione di termini relazionati fra loro, anche detta *thesaurus*.

La presente Tesi è strutturata in cinque capitoli, suddivisi in due parti. Nella prima parte sono spiegati gli argomenti e gli strumenti utilizzati durante la fase di ricerca mentre la seconda parte è relativa all'implementazione dell'applicazione.
Il documento si presenta pertanto così:
Parte prima: Stato dell'arte

- **Capitolo 1: Introduzione al linguaggio Swift**. Il seguente capitolo ha lo scopo di illustrare in maniera sintetica, ma comunque ricca di esempi, la sintassi e l'uso di Swift anche attraverso confronti con altri linguaggi di programmazione conosciuti.

- **Capitolo 2: Analisi semantica del testo**. In questo capitolo si parla brevemente della struttura di un database lessicale come WordNet e di come i termini del vocabolario possono relazionarsi fra loro attraverso l'uso degli insiemi semantici.

Parte seconda: L'applicazione

- **Capitolo 3: Progettazione**. Questo capitolo è relativo alla progettazione del gioco (*game design*) che si è sviluppato con relativa spiegazione degli elementi che lo compongono.

- **Capitolo 4: Implementazione**. Si analizzerà il nocciolo del progetto su cui si basa la seguente Tesi; tratta di come sono stati utilizzati gli strumenti descritti nella prima parte per poter realizzare l'applicazione progettata nel capitolo 3.

- **Capitolo 5: Conclusione**. Descrive le conclusioni finali a cui si è giunti alla fine del lavoro di ricerca ed i possibili sviluppi futuri.

Parte I

Stato dell'arte

Capitolo 1

Introduzione al linguaggio Swift

1.1 Caratteristiche del linguaggio

Swift è un nuovo linguaggio di programmazione rilasciato da Apple Inc. nella versione per sviluppatori il 2 giugno 2014 e, per gli utenti normali, insieme alla release del nuovo sistema operativo mobile iOS 8 il 17 settembre 2014.

Swift si pone come obiettivo primario quello di essere un linguaggio fresco e veloce da apprendere, soprattutto per i programmatori alle prime armi, riuscendo tuttavia ad integrarsi perfettamente con i linguaggi preesistenti risultando familiare anche agli sviluppatori Objective-C.

Swift è un linguaggio basato principalmente sul C e l'Objective-C prendendo le parti migliori di ognuno. In particolare è visto come il linguaggio che intende risolvere i principali problemi che si presentavano in Objective-C.

La primissima caratteristica che risalta è il livello di astrazione raggiunto dal linguaggio. Se Objective-C poteva ora essere paragonato a Java grazie all'aggiunta dei framework per la gestione della memoria senza l'uso dei puntatori, Swift, per la sua somiglianza con i linguaggi

di scripting quali Python e Ruby, può essere addirittura annoverato tra essi.

1.1.1 Tipi di dato

In Swift ci sono i consueti tipi di dato di base (numeri e caratteri) a cui si aggiungono gli array, i dictionary e le tuple (che permettono di ritornare valori multipli), come nella maggior parte dei linguaggi dinamici.

Tuttavia, a differenza dei linguaggi di scripting, la tipizzazione dei dati risulta invece essere molto forte, grazie all'uso del *type checking* praticamente immediato che avviene appena finita di digitare una riga di codice nell'IDE Xcode e alla sua caratteristica di essere un linguaggio *type safe*, cioè non permette ad un cast implicito di produrre un crash del programma, pertanto quasi sempre vengono richiesti cast espliciti.

Un'altra particolarità di Swift è la possibilità di utilizzare i valori opzionali. Ciò significa che una variabile può contenere il valore di ritorno di una funzione se questa restituisce qualcosa (un tipo di dato definito) oppure potrebbe non contenere nulla nel caso la funzione non restituisca per un qualsiasi motivo il dato richiesto; in questo caso alla variabile viene assegnato il valore *nil*. Per esemplificare la spiegazione, ecco una piccola applicazione:

```
1  let stringaIntera = "123"
2  let stringaNonIntera = "abc"
3
4  /* La funzione toInt() ritorna un valore opzionale intero */
5  let valoreOpzionale : Int? = stringaIntera.toInt()
6  println(valoreOpzionale)        // qui stampa "Optional(123)"
7
8  /* Qui la funzione toInt non puo' convertire il dato in un
9  * intero pertanto ritorna nil */
10 valoreOpzionale = stringaNonIntera.toInt()
11 println(valoreOpzionale)        // qui stampa "nil"
```

Infine, la versatilità del linguaggio ci permette anche di ridefinire gli operatori di base, come in questo esempio, oppure di definirne di nuovi personalizzati:

```
/* Definiamo un nuovo tipo di dato */
struct numeroBinario{ var val : Int }

/* Ridefiniamo l'operatore somma trasformandolo in somma
* binaria : 0+0=0, 0+1=1, 1+0=1, 1+1=0 */
func + (op1 : numeroBinario, op2 : numeroBinario) -> Int {
    if op1.val != op2.val{
        return 1
    }
    return 0
}
/* Dichiariamo due numeri binari con valore 1 */
let a = numeroBinario(val:1); let b = numeroBinario(val:1)
println(a+b)     // Stampa 0
```

1.1.2 Cicli

I cicli presenti sono quelli classici del C come il *while*, il *do-while* e il *for-condition-increment*; inoltre viene aggiunto il ciclo *for-in* che grazie all'uso dell'operatore *in* permette di ciclare su un oggetto iterabile senza preoccuparsi della dimensione dell'oggetto e del passo di incremento.

1.1.3 Statement

Swift supporta i costrutti condizionali *if-else* e *switch-case* intesi come in C tranne per la caratteristica dello *switch-case* di non avere il fallthrough implicito, quindi una volta eseguita l'istruzione scelta non prosegue con le istruzioni successive e pertanto non necessita dell'interruzione `break` come in C o in Java.

1.1.4 Funzioni

Le funzioni in Swift hanno una signature del tipo

```
func NomeFunzione(variabile: TipoVariabile) -> TipoRitorno
```

prendono quindi in ingresso le variabili dichiarandone il tipo proprio come nella maggior parte dei linguaggi tipizzati.

I parametri d'ingresso sono generalmente passati per valore, creando una copia; per passare invece i parametri per riferimento si utilizzano la parole chiave `inout` da anteporre al nome della variabile interessata.

In Swift non esistono le funzioni variadiche come in C, tuttavia queste possono avere parametri variabili, cioè prendere in ingresso zero o più valori di un determinato tipo semplicemente aggiungendo i puntini di sospensione dopo il primo parametro indicato.

Infine le funzioni possono avere come argomento altre funzioni, dette *anonime* perché senza prototipazione, queste prendono il nome di *closure*. Le closure si possono infatti intendere come i blocchi in C o in Objective-C, servono principalmente per recuperare riferimenti di costanti e variabili dal contesto in cui sono definite.

A differenza dell'Objective-C, Swift possiede molteplici funzioni integrate che possono essere utilizzate come meglio si crede per svariate tipologie di problemi; queste funzioni servono in generale a semplificare azioni che comporterebbero l'uso di un ciclo come per esempio la funzione *map* che compie una trasformazione lineare sui dati presi in ingresso (tipicamente un array o una tupla) del tipo $map\langle U\rangle((T) \mapsto U) \mapsto U[\,]$, qui sotto un esempio d'uso:

```
1 let prezzi = ["10", "20", "35", "40"]
2 /* Aggiungiamo "$" accanto ai prezzi */
3 var prezzi_dollaro = prezzi.map("{$\($0)}")
4 println(prezzi_dollaro) // stampa  "[$10, $20, $35, $40]"
```

La riga 3 mostra come sia possibile aggiungere a tutti gli elementi dell'array il simbolo del dollaro $ senza l'ausilio di un ciclo.

Allo stesso modo la funzione *reduce* opera una trasformazione del tipo $reduce(\text{initial:}U, combine(U,T) \mapsto U) \mapsto U$, mentre la funzione *filter* permette di selezionare in un array solo gli elementi che verificano una certa condizione, un po' come la clausola `where` in SQL.

1.2 Strumenti di sviluppo

Per sviluppare applicazioni per iOS e OS X è necessario disporre di alcuni strumenti fondamentali. Per compilare i sorgenti bisogna essere in possesso delle SDK relative al linguaggio con cui si intende sviluppare l'applicazione, in questo caso Swift; queste librerie vengono distribuite insieme all'ormai famosissimo ambiente di sviluppo *Xcode*. Per poter pubblicare le applicazioni sullo store dedicato è tuttavia richiesto un account *Apple Developer*, che prevede un contributo annuale di 99$.

Infine, almeno fino a questo momento, l'unica limitazione imposta da Apple Inc. è stata quella di distribuire gli strumenti di sviluppo solo agli utenti Apple, in quanto non era possibile scindere il compilatore dall'ambiente di sviluppo integrato, installabile solo su un computer con sistema operativo OSX originale. Tuttavia, alla *Worldwide Developers Conference* dello scorso 8 giugno 2015, Apple ha annunciato che la prossima versione di Swift 2.0 (prevista per fine 2015) sarà rilasciata sotto licenza *open-source* con il relativo compilatore Swift e le librerie per iOS, OS X e Linux. Questo però non permetterà ai programmatori sprovvisti di un Mac di sviluppare applicazioni per iOS e OS X, né di accedere all'ambiente di sviluppo Xcode; sarà possibile però utilizzare Swift per scrivere qualsiasi altro tipo di applicazione, in particolare le *web application* perché Swift si proprone anche come un potente linguaggio di programmazione web.

1.2.1 Xcode

Apple, come detto in precedenza, prevede dunque lo sviluppo delle sue applicazioni legate al suo IDE Xcode. Qui infatti i programmatori potranno trovare l'ambiente adatto per implementare i programmi con il supporto relativo. Le funzionalità previste dal software sono infatti molteplici e sicuramente la più utilizzata è quella del *Assistant Editor* che consiste nell'autocompletamento durante la scrittura prevedendo anche la signature delle funzioni.

L'ambiente è suddiviso in varie macro aree: il *navigator*, l'*editor*, la *toolbar*, la *utility* ed il fondamentale *debug*. In quest'ultimo è presente anche la console per verificare l'output del programma, oltre ai relativi controlli del debugger dovuti ai breakpoint.

Xcode include anche un comodissimo controllo di versione GIT o Subversion con cartella locale all'interno del progetto stesso. Inoltre, per progetti medio-piccoli, è presente anche la funzionalità *snapshot*, che permette di salvare una versione del programma per potervi tornare indietro in caso di problemi futuri.

1.2.2 Playground

Sicuramente la novità più piacevole di Xcode 6 è quella del *playground*. Basta creare infatti un file con estensione ".playground" per poter aver accesso ad un ambiente interattivo che permette di provare il codice senza compilarlo ed eseguirlo; qui infatti è possibile vedere in anteprima cosa produce il codice e testare parti di applicazioni con facilità. È anche il modo più semplice per apprendere il linguaggio e "giocare" con esso, da qui infatti il nome playground.

1.2.3 Gestione della memoria

La gestione della memoria in Swift non è lasciata al programmatore, come accade invece in C, bensì viene affidata all'*Automatic Reference Counting* che prevede l'utilizzo di un contatore per ogni oggetto che contiene il numero degli oggetti che lo referenziano.

Il reference counting è molto veloce a differenza del garbage collector utilizzato da altri linguaggi di programmazione quali Java e Python; i contatori vengono infatti decrementati non appena si esce da un blocco di codice o si perde il riferimento ad una variabile.

L'unico svantaggio dell'uso dei contatori è il rischio che un'istanza di una classe non raggiunga mai il contatore delle referenziazioni a zero perché si è creato uno *strong reference cycle*, cioè un ciclo dovuto a referenziazioni marcate forti. Un esempio di come ciò può accadere, riportato nella pagina accanto, è spiegato nel libro [App14]:

```
1  class Person {
2      let name: String
3      init(name: String) { self.name = name }
4      var apartment: Apartment?
5      deinit { println("\(name) is being deinitialized") }
6  }
7
8  class Apartment {
9      let number: Int
10     init(number: Int) { self.number = number }
11     var tenant: Person?
12     deinit {
13         println("Apartment #\(number) is being deinitialized")
14     }
15 }
16
17 /* Dichiariamo due variabili opzionali */
18 var john: Person? ; var number73: Apartment?
19
20 /* Creiamo due istanze delle rispettive classi */
21 john = Person(name: "John Appleseed")
22 number73 = Apartment(number: 73)
23
24 john!.apartment = number73
25 number73!.tenant = john
26
27 /* Proviamo a deallocare la memoria senza successo */
28 john = nil
29 number73 = nil
```

Nella riga 24 viene creato un riferimento forte tra l'oggetto `Apartment` e l'attributo apartment dell'oggetto `Person`; il simbolo `!` serve per forzare il compilatore al recupero dell'attributo all'interno della classe `Person`, `john` è infatti dichiarato come variabile opzionale, senza il punto esclamativo il compilatore proverebbe a ritornare l'istanza della classe `Person` e non gli attributi al suo interno proprio perché l'istanza è di tipo opzionale. Quando poi (riga 28) si perde il riferimento della variabile `john`, il reference counter dell'oggetto puntato dalla variabile non scende a zero, bensì passa da 2 a 1 e pertanto l'oggetto non può essere deallocato

dall'ARC (non viene eseguita la funzione `deinit`). Lo stesso avviene se si cerca di deallocare l'oggetto `Apartment` mettendo a nil la variabile `number73`.

Per ovviare al problema, è stato introdotta la parola chiave `weak` da aggiungere prima della dichiarazione dell'attributo della classe in questione. Se nella classe `Apartment` la variabile fosse stata definita tramite una relazione debole:

```
weak var tenant: Person?
```

L'istruzione `john = nil` avrebbe prodotto la distruzione dell'oggetto di tipo `Person` e il programma avrebbe quindi stampato l'istruzione prevista nella funzione `deinit` vista alla riga 5.

Capitolo 2

Analisi semantica del testo

I dati possono essere raggruppati in due grandi macro-aree: i dati *strutturati* (tipicamente organizzati in un DBMS) ed i dati *non strutturati* (testo libero). Quest'ultimo tipo necessita di strumenti particolari per poter estrapolare le informazioni che contiene, come l'elaborazione dei documenti prima di utilizzarli ed i *thesauri*, che racchiudono un insieme di parole con una possibile descrizione per ognuna di esse.

In questo capitolo verrà descritto il thesaurus *Wordnet* che è stato utilizzato per lo sviluppo dell'applicazione descritta nel capitolo 3 ed approfondita nel capitolo successivo relativo all'implementazione.

2.1 WordNet

Nel 1985, un gruppo di psicolinguisti dell'università di Princeton, New Jersey USA, ha iniziato lo sviluppo di un database lessicale sulla base delle scoperte di George Amirtage Miller e Philip Johnson-Laird, autori dell'opera *Linguaggio e percezione* del 1976. Il progetto aveva lo scopo di realizzare un dizionario non pensato per essere utilizzato tramite ricerche alfabetiche, bensì sulla base di concetti.

WordNet è il risultato di questo esperimento che ha portato a formulare teorie molto più complesse ed ambiziose rispetto agli obiettivi iniziali cui si erano preposti i fondatori.

Oggi è un *thesaurus* che, come accennato precedentemente, non è altro un dizionario contenente lemmi con la loro relativa descrizione sotto forma di database lessicale. Concettualmente, un thesaurus può essere definito come un enorme grafo contenente diversi termini (i nodi del grafo) collegati fra loro attraverso varie relazioni (gli archi fra i nodi). WordNet, progettato in lingua inglese (poi tradotto in molteplici lingue), viene distribuito gratuitamente dal sito dell'università con libera licenza di utilizzo previa citazione degli autori ([MilG95] e [FelC98]) e il riferimento al sito http://wordnet.princeton.edu/. Al sito http://wordnetweb.princeton.edu/perl/webwn è disponibile un tool online, liberamente utilizzabile, per testare le funzionalità e verificare le relazioni fra i lessemi.

Il database lessicale WordNet non è un semplice dizionario di termini inglesi, è un sistema lessicale di riferimento il cui disegno è inspirato alle teorie psico-linguistiche contemporanee sulla memoria lessicale umana. I termini, infatti, non sono disposti seguendo l'ordine alfabetico (cosa che accosterebbe parole con significati fra loro anche molto diversi), ma per affinità di significato.

La più grande differenza tra WordNet ed un dizionario standard è quello delle divisione del bagaglio lessicale in quattro categorie sintattiche: nomi, verbi, aggettivi ed avverbi, organizzate tra di loro in termini di significato piuttosto che per forma di parola. La scelta dell'ordinamento del thesaurus è dettata però dal problema riguardante un thesaurus organizzato in ordine alfabetico: i termini ridondanti. Se una parola W_x ed una W_y sono sinonime fra loro, sia la coppia (W_x, W_y) che la coppia (W_y, W_x) sarebbero inserite nel database, una volta sotto la lettera x ed una sotto la lettera y, creando il fenomeno della ridondanza.

Ogni categoria sintattica è suddivisa in diversi insiemi di sinonimi a cui viene associato un significato comune (*sense*) condiviso da tutti i termini, anche se, ovviamente, un termine può possedere più di un significato ed essere presente contemporaneamente in più di un insieme

o in più di una categoria sintattica. All'interno del progetto WordNet l'insieme dei lessemi che condivide uno stesso significato viene chiamato *synset*.

Questi insiemi, a differenza di un qualsiasi vocabolario, sono collegati fra loro tramite diversi tipi di relazioni; nel progetto che è stato realizzato vengono utilizzate soprattutto le relazioni di *ipernimia* e *iponimia* che verranno approfondite in seguito.

2.1.1 La matrice lessicale

Per comprendere le relazioni e le associazioni che vi possono essere tra vari synset è necessario fare un passo indietro e spiegare in che modo i lemmi ed i sense sono collegati fra loro.

Un termine, inteso come lemma facente parte di un determinato lessema, può avere, come accennato in precendenza, più di un significato ed in questo caso si è di fronte ad un caso di *polisemia*; d'altra parte un dato significato può essere associato a due o più termini, rendendoli *sinonimi*.

Tabella 2.1: La matrice lessicale.

Forme / Significati	L_1	L_2	L_3	L_4
S_1	$E_{1,1}$			
S_2		$E_{2,2}$		
S_3		$E_{3,2}$	$E_{3,3}$	
S_4				$E_{4,4}$

Questi due problemi di lessicografia possono essere visti come due aspetti complementari della cosiddetta *matrice lessicale*, in cui sulle colonne troviamo i *lemmi* e sulle righe i *synset*. Nella tabella 2.1 si possono osservare chiaramente le due relazioni lessicali:

- **sinonimia**: se in una riga ci sono più entry significa che i due termini associati si riferiscono allo stesso significato (come nel caso delle celle (3,2) e (3,3))

- **polisemia**: se in una colonna ci sono più entry significa che lo stesso termine è associato a più significati (come nel caso delle celle (2,2) e (3,2))

2.2 Le relazioni

In WordNet troviamo due tipi di relazioni in base al fatto che queste siano tra lemmi o tra synset. Nel primo caso parliamo di *relazioni lessicali*, nel secondo invece di *relazioni semantiche*. Non possono esistere relazioni fra un lemma ed un synset o fra operandi appartenenti a diverse categorie sintattiche (ad esempio fra un nome ed un verbo). In seguito verranno analizzate le relazioni utilizzate per lo sviluppo dell'applicazione in Swift descritta nel capitolo 4.

2.2.1 Relazioni lessicali

Le relazioni lessicali coinvolgono due o più forme di parola, i cosiddetti lemmi e non comprendono dunque le relazioni tra synset.

La relazione lessicale utilizzata nel progetto è quella di **sinonimia**, che viene espressa in WordNet non come un collegamento fra due parole bensì con il fatto che due parole appartengano o meno allo stesso synset. Se infatti due o più termini si trovano nella stessa riga della matrice semantica, spiegata in precedenza, sono fra loro sinonimi. Lo stesso "synset" risulta essere una parola composta dai termini *synonimous* (sinonimo) e *set* (insieme), creando quindi l'idea dell'insieme dei sinonimi.

2.2.2 Relazioni semantiche

Le relazioni semantiche risultano invece meno immediate di quelle lessicali poiché gli operandi che ne fanno parte sono due synset, quindi vi è un ulteriore passaggio logico che permette di collegare i vocaboli finali fra loro tramite le relazioni semantiche.

Queste relazioni sono molteplici, ma per lo sviluppo del gioco, ne sono state utilizzate solamente due: la relazione di **iponimia** e quella

di **ipernimia**, presenti solamente nelle categorie sintattiche di nomi e verbi.

La prima è una relazione che può essere definita come una specificazione di un concetto più ampio che sta al di sopra di esso. In termini matematici si potrebbe definire come $x \in X = \{x_1, x_2, ..., x_n\}$ e $y \in Y = \{y_1, y_2, ..., y_n\}$, con X e Y synset, dove x è una parte di y. Nel caso di verbi invece la relazione di iponimia prende il nome di *troponimia*: x è un troponimo di y, se compiere l'azione di x implica in qualche modo fare anche y. Per fare un esempio più pratico, in termini informatici si potrebbe intendere la relazione di iponimia come la relazione di ereditarietà fra classi: l'iponimo è la sottoclasse che eredita dalla classe padre e si specializza in qualcosa.

La relazione di ipernimia è invece la relazione opposta rispetto a quella di iponimia, è il concetto generale che viene esemplificato e specializzato dalle sottocategorie, gli iponimi.

Cerchiamo ora di capire meglio le relazioni di ipernimia ed iponimia grazie ad un sottografo di WordNet, con un esempio reale:[1]

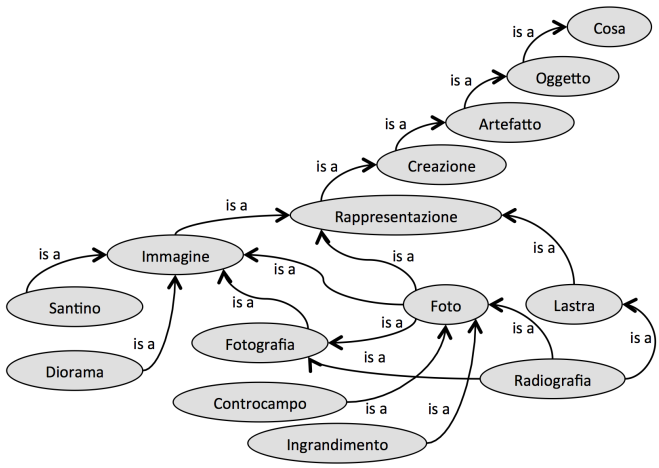

Figura 2.1: Esempio di relazioni di ipernimia e iponimia.

[1]Il grafo è stato realizzato utilizzando le relazione previste dal sito http://multiwordnet.fbk.eu/online/multiwordnet.php.

Come si vede nella figura 2.1, gli archi sono collegamenti semantici fra synset tramite la relazione *is-a*, cioè la relazione di ipernimia. Questo significa che le parole che hanno una freccia in ingresso sono in qualche modo antenate delle parole ad esse collegate. Per esempio *lastra* e *immagine* non sono collegate fra loro direttamente però possiedono un antenato comune, il termine *rappresentazione*, rendendo le due parole fra loro iponime.

2.3 MultiWordNet

Per l'applicazione sviluppata era tuttavia necessario l'utilizzo di un dizionario in lingua italiana, si è quindi scelto di utilizzare un database lessicale basato su WordNet, scaricabile gratuitamente con licenza Creative Commons 3.0 dal sito http://compling.hss.ntu.edu.sg/omw/ in concessione d'uso con redistribuizione.

Il progetto MultiWordNet [PiaBG02] contiene circa 40 mila lemmi, 61 mila sense e 34 mila synset e viene distribuito in formato XML (*eXtensible Markup Language*), un linguaggio basato su *tag* che permette una classificazione dei dati in maniera semi-strutturata.

L'organizzazione dei dati è strutturata in maniera gerarchica. Rappresentabile, in questo modo, tramite una rappresentazione ad albero:

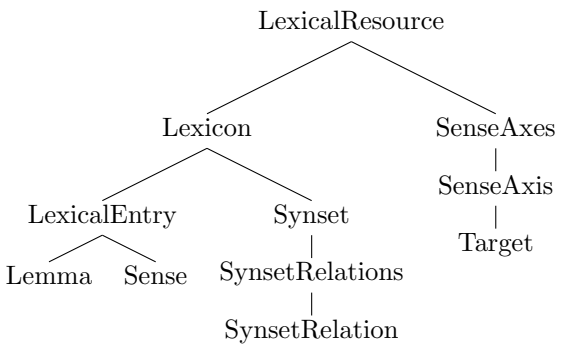

Figura 2.2: MultiWordNet visto ad albero.

In XML questo viene tradotto così:

```
<LexicalResource>
    <Lexicon>
        <LexicalEntry>
            <Lemma ... />
            <Sense ... />
        <\LexicalEntry>
        ...
        <Synset>
            <SynsetRelations>
                <SynsetRelation ... />
                ...
            <\SynsetRelations>
        <\Synset>
        ...
    <\Lexicon>
    <SenseAxes>
        <SenseAxis>
            <Target ... />
            ...
        <\SenseAxis>
    <\SenseAxes>
<\LexicalResource>
```

L'analisi completa di come questi dati sono stati estrapolati ed utilizzati nell'applicazione, verrà effettuata nel capitolo 4.

Parte II

L'applicazione

Capitolo 3

Progettazione

Per consolidare le basi acquisite durante lo studio e l'approfondimento del linguaggio Swift, è stata sviluppata una semplice applicazione iOS che ripropone il gioco conosciuto con il nome de "*La Ghigliottina*" reso famoso dal programma televisivo "*L'eredità*", in onda su Raiuno.

3.1 Analisi del gioco

Lo scopo del gioco è quello di indovinare la parola nascosta e, per farlo, vengono fornite 5 parole ad essa collegate. Per rendere ancora più difficile il gioco, le parole non sono assegnate all'inizio della partita ma devono essere scelte mano a mano dal giocatore tra due parole alla volta: se si sceglie la parola giusta si prosegue con la prossima coppia di parole, se invece la parola scelta è quella sbagliata il montepremi viene dimezzato (da qui il nome "*Ghigliottina*").

La modalità di gioco viene rappresentata in dettaglio, tramite un diagramma di flusso, nella figura 3.1.

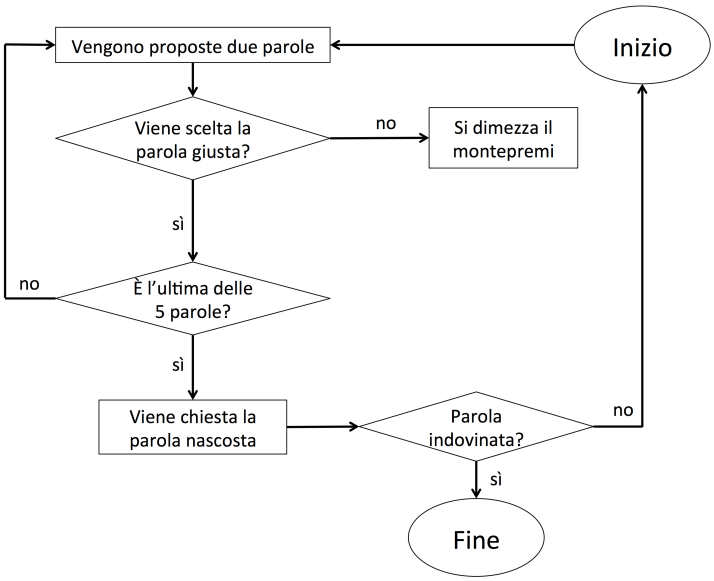

Figura 3.1: Diagramma di flusso rappresentante la modalità di gioco.

3.1.1 Meccaniche del gioco

Analizziamo ora nel profondo la tipologia di gioco che si intende sviluppare facendo riferimento alla suddivisione delle meccaniche del gioco previste nella disciplina che prende il nome di *Game studies*.

Gli oggetti del gioco sono per noi le *parole* che possono avere l'attributo di iponimo della parola nascosta oppure di sinonimo della parola giusta.

Gli stati a cui possono accedere gli oggetti sono solamente due: "selezionato" o "scartato", in base alla scelta dell'utente in ogni momento.

Le azioni e le scelte possibili del giocatore vengono racchiuse in quello che prende il nome di "*meaningful play*", cioè l'insieme delle azioni operative (proprie del giocatore) e di quelle risultanti (riguardanti le strategie di gioco).

L'elemento casuale è fondamentale in ogni gioco. Qui è rappresentato dalla scelta randomica della parola nascosta da indovinare nonché di quelle "avversarie" che vengono proposte per fuorviare il giocatore.

3.2 Schemi di gioco

La particolarità del remake del gioco è quella di non avere all'inizio della partita gli schemi di gioco già pronti che vengono precaricati dal sistema.

Quasi tutti i giochi, soprattutto quelli mobile che tengono conto della scarsa disponibilità delle risorse hardware del dispositivo in uso, prevedono l'uso di schemi di gioco fissi e predefiniti dal programmatore. Questo approccio apporta molti vantaggi dal punto di vista di una game application, primo dei quali la velocità di esecuzione perché permette il precaricamento dello schema successivo in memoria centrale; tuttavia il limite degli schemi definiti a priori è quello di essere statici e quindi non variabili tra una partita e l'altra.

La scelta implementativa della generazione automatica degli schemi è dettata quindi dalla volontà di realizzare un sistema automatico che, selezionata una parola a caso, scelga autonomamente le 5 coppie di parole che completano lo schema di gioco. La spiegazione completa dell'algoritmo verrà completata nel capitolo 4.

Capitolo 4

Implementazione

Per lo sviluppo dell'applicazione si è scelto di seguire un approccio, in riferimento all'ingegneria del software, di tipo evolutivo. Si tratta di un modello altamente flessibile, adatto per linguaggi a prototipazione veloce come Swift, che permette uno sviluppo trasversale delle specifiche del software, della scrittura stessa del codice e della sua validazione successiva.

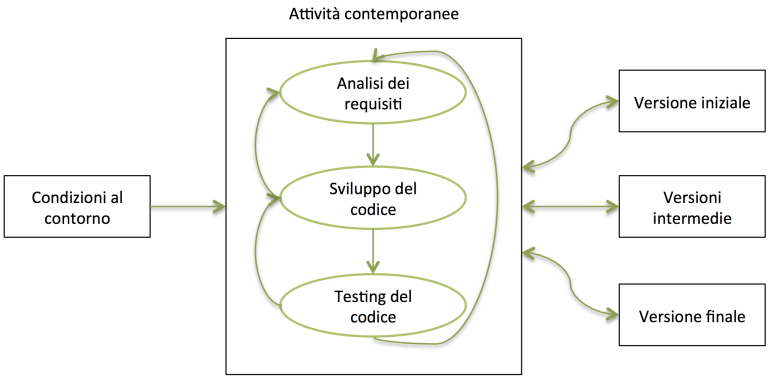

Figura 4.1: Modello evolutivo.

Il modello evolutivo prevede infatti una prima analisi dei requisiti seguita dallo sviluppo dell'applicazione ed il suo testing, a seguito del

quale i requisiti vengono messi in discussione ed eventualmente corretti od integrati con nuovi elementi emersi durante le fasi precedenti. Si continua in questo modo fino a quando non si produce, volta per volta, una versione del programma ritenuta completa e quindi dichiarata finale. Il susseguirsi delle varie fasi è descritto nella figura 4.1.

Si è scelto questo modello per la continua revisione che viene impegata sull'intera applicazione in quanto Swift, nonostante sia un progetto iniziato 5 anni fa, è un linguaggio in continua evoluzione con release, beta e definitive, che vengono rilasciate ogni 3-6 mesi.

Per lo sviluppo dell'applicativo è stata utilizzata la versione di Swift 1.1, rilasciata lo scorso 22 Ottobre 2014. Tuttavia successivamente si è scelto di aggiornare l'appicazione alla major version 1.2 dell'8 aprile 2015 per integrarla con una nuovo wrapper [FarF14] in Swift scritto appositamente per permettere l'interfacciamento di un'applicazione con un database sottostante.

4.1 Processamento dei dati

I dati previsti da MultiWordNet, come accennato nel capitolo 2, sono in formato XML. Per interfacciare un'applicazione con questo formato in genere si utilizza un *parser*, cioè un software in grado di effettuare l'analisi sintattica di un documento XML nonché di determinare se esso risulta essere ben formato.

I parser XML possono dividersi in due grandi categorie in base alle API (*Application Programming Interfaces*) che implementano:

- **DOM** (*Document Object Model*): parser che dopo aver analizzato l'intero documento XML, generano una struttura interna navigabile ad albero.

- **SAX** (*Simple API for XML*): parser che restituiscono una risposta solo dopo il verificarsi di un evento, che deve essere catturato e gestito dall'applicazione; non costruiscono nessuna struttura ad albero e quindi sono molto più veloci.

I parser di tipo SAX sono in genere utilizzati quando si ha a che fare con file di grandi dimensioni perché, non costruendo un albero interno, non occupa così tanta memoria principale, salvando solo le informazioni necessarie per quel dato evento. Tuttavia, un parser di questo tipo, non creando gli oggetti "nodi" tipici dell'albero sintattico di DOM, non è in grado di realizzare relazioni fra essi a meno di interventi espliciti da parte del programmatore che, per ottenere questo risultato, dovrebbe comunque salvare in memoria buona parte dei dati annullando così il beneficio prodotto dal parser in termini di consumo di risorse.

Vista l'importanza delle relazioni fra i vari nodi del grafo per il database lessicale MultiWordNet e per l'applicazione che si intendeva sviluppare, si è quindi scelto di utilizzare un parser di tipo DOM per avere a disposizione un albero da poter utilizzare per il processamento dei dati.

Considerando tuttavia il dispositivo finale per cui l'applicazione è stata sviluppata e le sue caratteristiche tecniche, non si è ritenuto opportuno utilizzare i dati appena estrapolati dal parser mantenendoli in memoria principale per non influire sui tempi di risposta complessivi del programma. Per questo motivo, parte dei dati (quella necessaria per l'applicazione) è stata elaborata e trasformata per essere inserita in un database di tipo SQLite all'interno del dispositivo finale.

L'albero sintattico della figura 2.2 è stato tradotto nelle seguenti tabelle SQL in cui possiamo vedere le chiavi primarie e i riferimenti che collegano le tabelle fra loro:

L'elaborazione dei dati dal formato iniziale XML a quello finale SQL è stato realizzato tramite una semplice applicazione realizzata in PHP (*Hypertext Preprocessor*) che estrapola gli oggetti del file XML tramite il parser XMLsimple e li inserisce all'interno di istruzioni SQL atte ad essere eseguite tramite script per il popolamento delle tabelle stesse.

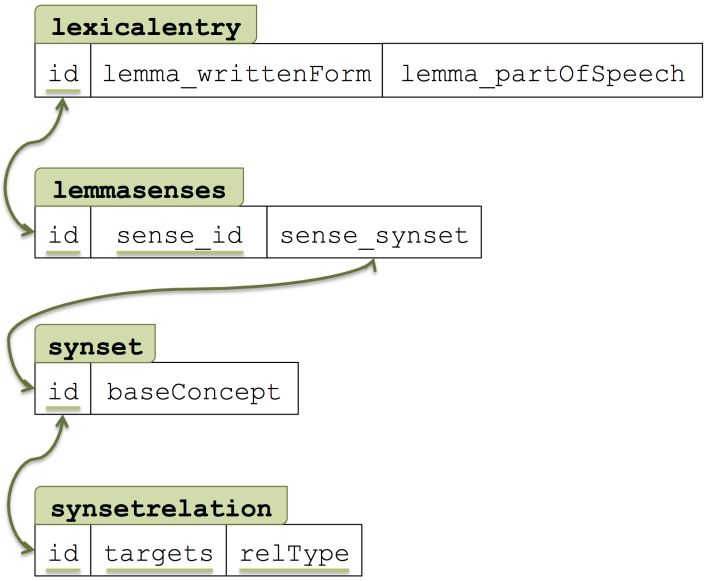

Figura 4.2: Schema delle tabelle in SQL.

4.2 Algoritmo di generazione schemi

Come accennato nella sezione 3.2 relativa agli schemi di gioco del precedente capitolo, la particolarità del sistema progettato è quella di determinare a tempo di esecuzione le parole facenti parte della partita attuale calcolandole sulla base delle loro relazioni semantiche e scegliendole a caso fra le candidate.

In questo modo ogni partita sarà diversa dalle altre senza la possibilità di annoiare il giocatore nel caso riconosca un possibile schema; ciò può accadere nei giochi basati su schemi predefiniti in cui la componente casuale è ridotta al minimo con il rischio di riuscire ad eludere le difficoltà del gioco stesso basandosi su tattiche date dalla determinazione del comportamento del programma in quel particolare schema.

4.2.1 Pseudocodice

L'algoritmo utilizzato risulta essere efficace nonostante la sua semplicità intrinseca. Illustriamolo tramite pseudocodice ed analizziamolo in dettaglio:

```
1  vocaboli <- {}
2  while lemmi is not empty
3      lemma <- random(lemmi)
4      count <- 0
5      for s in synset(lemma)
6          if s(relType) = "hypo"
7              count ++
8          if count >= 5
9              vocaboli <- vocaboli + lemma
10     lemmi <- lemmi - lemma
11 parola_nascosta <- random(vocaboli)
12 indizi <- {}
13 intrusi <- {}
14 while synset(parola_nascosta) is not empty
15     s <- random(synset)
16     indizio <- lemma(s)
17     while intruso not parola_nascosta and intruso not in indizi
18         intruso <- random(vocaboli)
19     indizi <- indizi + indizio
20     intrusi <- intrusi + intruso
21     synset(parola_nascosta) <- synset(parola_nascosta) - s
22 return parola_nascosta, indizi, intrusi
```

Analizziamo ora il codice. Innanzitutto suddividiamolo in base al recupero degli oggetti finali che servono per costruire uno schema di gioco completo: la *parola nascosta*, le *parole-indizio* e le *parole-intruso*.

- **Parola nascosta**: è la parola che il giocatore deve indovinare alla fine della partita, se la indovina ha vinto il montepremi virtuale ed ottenuto punti. La parola nascosta viene selezionata a caso tra tutte le parole del dizionario che possiedono almeno 5 synset collegati ad altri synset tramite una relazione di iponimia.

- **Parola-indizio**: è una delle 5 parole che vengono proposte mano a mano all'utente durante il gioco; le parole-indizio sono quelle che devono essere selezionate per evitare di dimezzare il montepremi a

disposizione. Le parole-indizio vengono scelte tra le parole associate ai synset iponimi selezionati nel passo precedente.

- **Parola-intruso**: è una delle 5 parole che sono visualizzate insieme ad una parola-indizio per indurre il giocatore ad una scelta i due termini; se selezionata dimezza il montepremi. Le parole-intruso vengono selezionate in maniera randomica tra tutte le parole del vocabolario escluse la parola nascosta e le parole-indizio.

4.2.2 Interrogare il database

L'algoritmo descritto viene implementato, per la maggior parte, tramite query in SQL effettuate attraverso il wrapper [FarF14] e per la restante parte in Swift.

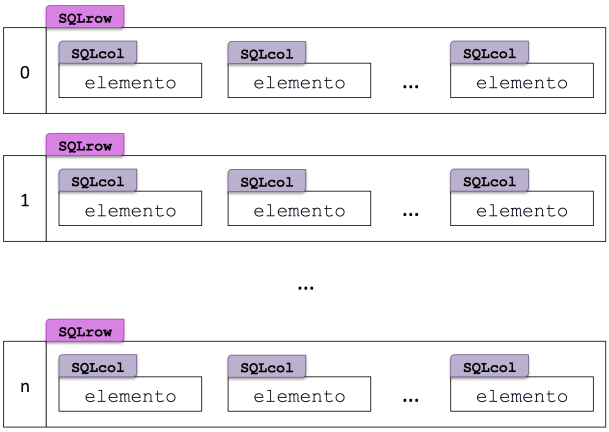

Figura 4.3: Matrice dei risultati.

Il wrapper implementa un ponte che collega il database SQLite all'applicazione restituendo i risultati delle query in oggetti di tipo `SQLrow`, array contenenti le righe della tabella restituite dalla query. In realtà il risultato della query risulta essere una matrice di oggetti `SQLrow` e `SQLcol` annidati fra loro come illustrato nella figura 4.3, pertanto l'accesso ad un elemento della matrice non può avvenire in maniera diretta tramite indici

di tipo `elem = matrix[Int][String]` perché le colonne della matrice non sono indipendenti dalla riga che le contiene.

Inoltre il wrapper implementa le funzioni necessarie per la conversione dei dati (asString(), asInt(), asDouble(), asData(), asDate()) provenienti dal database e recuperati tramite il campo dell'oggetto `SQLcol` incapsulato all'interno dell'oggetto di tipo `SQLrow`.

Per esempio, la lista dei vocaboli destinati a svolgere il ruolo di parola nascosta è stata recuperata in questo modo: [1]

```
1  let db = SQLiteDB.sharedInstance()
2  var array_tuple: [(writtenForm:String,
3                     synset:String,
4                     targets:String)] = []
5  let q = "select lex.lemma_writtenForm,
6                 lem.sense_synset,
7                 s1.targets
8                 from lexicalentry lex,
9                 lemmasenses lem,
10                synsetrelation s1
11                where s1.reltype='hypo'
12                and s1.id = lem.sense_synset
13                and lex.id = lem.id
14                and lex.lemma_writtenForm <> 'GAP!'
15                and lex.lemma_writtenForm <> 'PSEUDOGAP!'
16                and s1.id <> s1.targets and 6<= (
17                select count (distinct s2.targets)
18                       from synsetrelation s2
19                       where s1.id=s2.id
20                       and s2.reltype='hypo')"
21 let parole = db.query(\(q))
22
23 for i in 0...parole.count-1{
24     let riga_parola = parole[i].data
25     let tupla = (
26     writtenForm:riga_parola["lemma_writtenForm"]!.asString(),
27     synset:riga_parola["sense_synset"]!.asString(),
28     targets:riga_parola["targets"]!.asString())
29     array_tuple.append(tupla)
30 }
```

[1] La sintassi della query nelle righe 5-20 è relativa a SQLite.

4.3 Scheletro dell'applicazione

Per realizzare la semplice interfaccia grafica del gioco sono state utilizzate le librerie *Cocoa Touch* incluse sia in Objective-C che in Swift.

Xcode mette a dispositione uno strumento molto ultile per la creazione delle scene relative al pattern architetturale *Model-View-Controller* (chiamato comunemente MVC) si tratta dello *storyboard*. Tramite questo tool si vuole fornire un modo semplice, grafico e, per quanto possibile, privo di codice per disegnare le interfacce grafiche delle applicazioni e per navigare tra i viewcontroller che la compongono.

La grafica realizzata è di questo tipo:

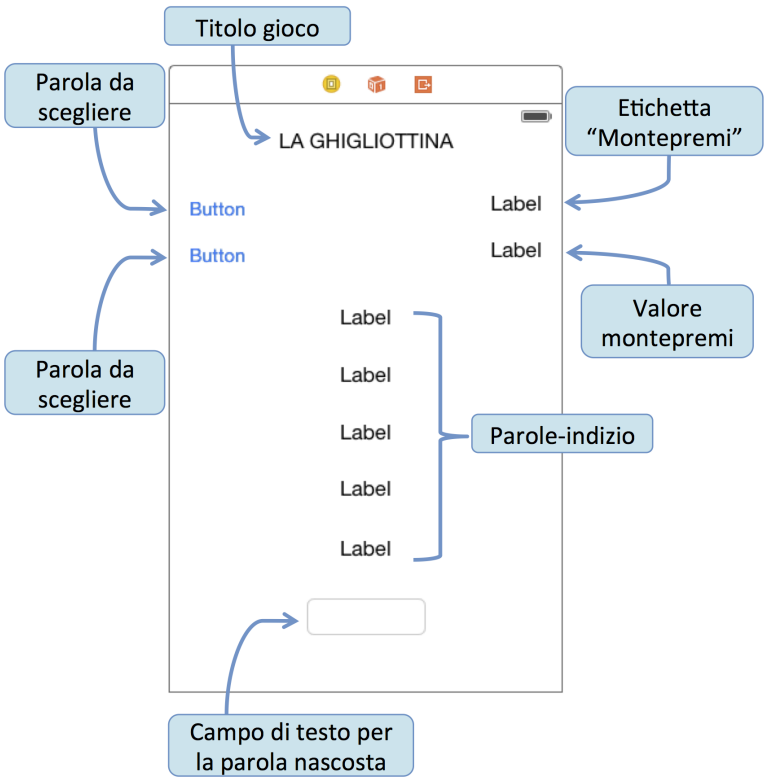

Figura 4.4: Interfaccia dell'applicazione.

Il collegamento degli oggetti relativi all'interfaccia grafica di tipo *User Interface* (`UILabel`, `UIButton`, `UITextBorderStyle`, ...) che fanno parte della libreria Cocoa risulta molto semplice nel codice con la loro dichiarazione nel ViewController tramite il tag `@IBOutlet`. D'altra parte, per associare una funzione ad un determinato oggetto dell'interfaccia basta creare un collegamento del tipo *drag and drop* dallo storyboard con le funzioni definite nel codice precedute dal tag `@IBAction`.

4.4 Esempio di partita

Per comprendere meglio la dinamica del gioco, si riporta una partita completa documentata attraverso schermate dell'applicazione stessa.

(a) (b)

Figura 4.5: Scelta della prima parola.

Come si nota nella figura 4.5, la scelta della parola-intruso (in questo caso *ladruncolo*) comporta non solo il dimezzarsi del montepremi a disposizione ma determina anche il colore del testo della parola-indizio (verde se è stata scelta, rossa se si è scelta la parola-intruso).

(a) (b)

Figura 4.6: Seconda e terza parola.

Nella figura 4.6 si scelgono due parole-indizio (a): *bolo* e (b): *regno*; pertanto la loro etichetta è verde in modo da ricordare al giocatore che ha scelto le parole giuste senza dimezzare ulteriormente il montepremi.

(a) (b)

Figura 4.7: Scelta delle ultime due parole.

La partita in questione prevede, come si può notare nella figura 4.7(b), cinque parole-indizio che dovrebbero aiutare il giocatore ad indovinare la parola nascosta che si cela nella schermata successiva.

Le parole-indizio sono tutte parole iponime della parola nascosta:

- competenza
- bolo
- regno
- privata
- pallottola

(a) (b)

Figura 4.8: Parola nascosta indovinata.

Una volta selezionate le parole-indizio non resta che scrivere nel campo di testo la parola nascosta pensata nell'apposito campo di testo in fondo alla schermata ed infine premere sul pulsante "Parola nascosta" per vedere se la parola è stata indovinata o meno.

Si accede così ad una nuova *view* (vedi figura 4.8(b)) organizzata in maniera molto semplice: qui vi sono infatti solamente tre `UILabel`, una con la descrizione "La parola nascosta era", una contenente la parola nascosta ed una che dice se la parola è stata indovinata o meno. Infine in fondo alla view è presente un `UIButton` che permette di tornare alla view precedente e giocare una nuova partita.

Se il giocatore indovina la parola nascosta scrivendola nel campo di testo (figura 4.8(a)), come avviene in questo caso con la parola *sfera*,

verrà visualizzata una schermata come la figura 4.8(b) in cui l'etichetta centrale declama "Hai indovinato la parola!".

(a) (b)

Figura 4.9: Parola nascosta non indovinata.

Se invece l'utente ha scritto nel campo di testo la parola sbagliata, il sistema risponde con un "Non hai indovinato la parola.", ovviamente sempre preceduto dalla parola nascosta che il giocatore doveva indovinare.

4.5 Playtesting

Il *playtesting* permette di individuare problemi durante l'implementazione del progetto stesso nonché di verificare che il lavoro svolto è in linea con gli obiettivi iniziali.

A tal proposito, i test effettuati si basano sull'osservazione diretta da parte del game designer su un campione di utenti finali a cui sono state fornite le 5 parole-indizio con l'unica richiesta di indovinare la parola nascosta che le accomuna.

In un primo momento, l'algoritmo prevedeva l'utilizzo di termini ipernimi come parole-indizio, tuttavia si è visto che la maggior parte degli utenti intervistati non erano in grado di indovinare l'iponimo relativo; inoltre, trovare un iponimo comune a vari ipernimi risultava essere computazionalmente elevato.

Per questo motivo si è scelto di utilizzare come parola nascosta un ipernimo delle parole-indizio, che risultano pertanto essere iponimi della parola da indovinare. I test effettuati dopo tale modifica hanno confermato la correttezza di questo procedimento; quasi un utente su due era infatti in grado di indovinare la parola nel tempo a disposizione (1 minuto), mentre due su tre riuscivano ad arrivare alla soluzione con un proroga di tempo.

4.5.1 Esempi

Per verificare il corretto funzionamento dell'applicazione e dell'algoritmo descritto nel paragrafo 4.2.1, si riportano alcuni esempi reali di schemi di gioco generati automaticamente dal programma.

Tabella 4.1: Esempio 1

Parola nascosta	Indizi	Intrusi
PRINCIPIO	DETTAME	RISPUNTARE
	FONDAMENTO	PELVI
	CAPOSALDO	CATENA DI MONTAGGIO
	MENARCA	PREBENDARIO
	INSORGENZA	OKAPI

Tabella 4.2: Esempio 2

Parola nascosta	Indizi	Intrusi
BUCO	CECO	SEDERSI
	OCCHIAIA	PIEDE
	ORBITA	IRRIFLESSIBILITÀ
	ANTRO	KAMIKAZE
	SERRATURA	BORRAGINE

Tabella 4.3: Esempio 3

Parola nascosta	Indizi	Intrusi
CARATTERE	DIGRAMMA	CLASSE OPERAIA
	INIZIALE	RIOTTENIMENTO
	ZETA	INSUFFICIENZA
	STAMPATELLO	CINQUE
	SPIRITO	INTERRUTTORE

Tabella 4.4: Esempio 4

Parola nascosta	Indizi	Intrusi
FATTO	MINIMO	COPERTINA
	MENO	DEPLIANT
	NODO	ACCASCIAMENTO
	ARGOMENTO	CHIESA
	ESPERIENZA	QUAGGA

Tabella 4.5: Esempio 5

Parola nascosta	Indizi	Intrusi
COMPAGNIA	INTESA	ANGOSTURA
	MASSONERIA	PERBENE
	BALDORIA	SARDINA
	BISBOCCIA	AUTOMATICO
	MASCHERATA	INIQUAMENTE

Tabella 4.6: Esempio 6

Parola nascosta	Indizi	Intrusi
SEMPLICE	FESSACCHIOTTO	CUCCHIAIO
	SCIOCCONE	MANTIDE RELIGIOSA
	MAMMALUCCO	LEONE MARINO
	PINCO	BORSEGGIATORE
	STOLTO	RIGUADAGNARE

Tabella 4.7: Esempio 7

Parola nascosta	Indizi	Intrusi
FONDO	FONDALE	IPNOTICO
	ALVEO	DISORDINATO
	LETTO	SUPINAZIONE
	FEUDO	MOLO
	BARONIA	FLANELLA

Capitolo 5

Conclusione

Gli obiettivi di questa tesi erano quelli di studiare ed approfondire il linguaggio Swift, nonché di analizzare ed utilizzare dati raggruppati in un database semantico-lessicale.

Il primo obiettivo è stato raggiunto attraverso la realizzazione di un'applicazione completa per iOS in Swift con l'intento di comprendere appieno il funzionamento del linguaggio e del suo ambiente di sviluppo. Per quanto riguarda l'utilizzo dei dati relativi al database lessicale, compreso nel secondo obiettivo, si è esaminato il thesaurus MultiWordNet, la sua struttura interna e le relazioni di natura semantica che ne collegano i concetti.

In aggiunta, si è deciso di integrare gli obiettivi richiesti all'interno di una *game application*, cioè si è sviluppata un'applicazione ludica che utilizza i dati dal database semantico-lessicale. Pertanto si sono analizzate le meccaniche relative alla progettazione di giochi per dispositivi mobili, compresi nella disciplina chiamata *game studies*, le strutture e le regole previste per un gioco in ambito di *game design*.

Infine, l'applicazione sviluppata riesce, attraverso l'uso di una *façade* (un design pattern che incapsula le funzionalità relative al calcolo degli schemi di gioco all'interno di un'interfaccia), a determinare una *user experience* semplice ed efficace che nasconda all'utente tutti i dettagli

implementativi effettuati a tempo di esecuzione. In questo modo, il giocatore non si rende conto della mole di calcolo che viene effettuata affinché il sistema risponda e lo schema di gioco sia pronto, resta in attesa del gioco che possiederà ad ogni riavvio una nuova parola da indovinare, sempre diversa.

5.1 Sviluppi futuri

L'applicazione sviluppata è stata realizzata con il solo scopo didattico e pertanto non è stata migliorata in tutti i suoi punti. In futuro tuttavia è possibile prevederne un miglioramento attraverso vari aspetti:

- **Efficienza**: i tempi di risposta del sistema sono per ora dell'ordine dei secondi, attraverso migliorie all'algoritmo utilizzato o tramite l'utilizzo di dati in *cache* per la seconda esecuzione del programma questi tempi potrebbero essere notevolmente abbassati.
- **Grafica**: l'interfaccia grafica dell'applicazione, per mancanza di tempo, non è stata personalizzata a dovere per renderla più *user friendly*.
- **Punti**: l'introduzione dei punti esperienza per il giocatore, con il relativo salvataggio di questi e il loro caricamento ad ogni avvio del gioco, permetterebbe un filo continuo tra le varie partite con la possibilità di accedere alle classifiche.
- **Sfide con gli amici**: trasformando l'applicazione in una web application o prevedendola installata su un server, si potrebbero creare sfide con gli amici dei Social Network con tempi di risposta real time.

Acronimi

API *Application Programming Interfaces*

ARC *Automatic Reference Counting*

DOM *Document Object Model*

PHP *Hypertext Preprocessor*

SAX *Simple API for XML*

XML *eXtensible Markup Language*

Bibliografia

[App14] "*The Swift Programming Language*" di Apple Inc., Editore Apple Inc. Giugno 2014.

[BerM14] "*Game design. Gioco e giocare tra teoria e progetto*" di Marisa Bertolo e I. Mariani, Pearson Editore Marzo 2014.

[MilG95] "*WordNet: A Lexical Database for English*" di George A. Miller, Editore Communications of the ACM Vol. 38, 1995.

[FelC98] "*WordNet: An Electronic Lexical Database*" di Christiane Fellbaum, Editore Cambridge, MIT Press, 1998.

[MisD02] "*Riscrittura di interrogazioni XML: un approccio basato sull'analisi semantica degli schemi*" di Daniele Miselli, Tesi di Laurea, 2002.

[MilBFGM93] "*Introduction to WordNet: An On-line Lexical Database*" di George A. Miller, Richard Beckwith, Christiane Fellbaum, Derek Gross, e Katherine Miller, 1993.

[PiaBG02] "*MultiWordNet: Developing an Aligned Multilingual Database*" di Emanuele Pianta, Luisa Bentivogli e Christian Girardi, 2002.

[GirC04] "*http://multiwordnet.fbk.eu/english/faq_it.php*" sito realizzato da Chirstian Girardi, 2004.

[FarF14] "*https://github.com/FahimF/SQLiteDB*" di Fahim Farook, Giugno 2014.

[MarM13] "*B4-XMLParsing*" slides di Gestione avanzata dell'informazione, di Riccardo Martoglia e Federica Mandreoli, 2013-2014.