ALMA MATER STUDIORUM
UNIVERSITÀ DEGLI STUDI DI BOLOGNA

Seconda Facoltà di Ingegneria
Corso di Laurea in Ingegneria Elettronica, Informatica e delle Telecomunicazioni

DAL CODICE AI MODELLI: ANALISI DI UN CASO DI STUDIO

Elaborata nel corso di: Ingegneria del Software

Tesi di Laurea di:
ANDREA PAGLIARANI

Relatore:
Prof. ANTONIO NATALI

ANNO ACCADEMICO 2011–2012
SESSIONE II

PAROLE CHIAVE

modelli

myJourney

analisi

collaudo

controller

Indice

Capitolo 1

Introduzione

Al giorno d'oggi il mobile é uno dei settori su cui il mercato si sta concentrando maggiormente e risulta essere in continua espansione; sempre piú numerose sono le aziende che focalizzano il loro business plan in questo campo. Le piattaforme attualmente piú gettonate sono Android e iOS, ma ve ne sono altre, come ad esempio Windows Phone, in forte ascesa (vedi Figura 1.1); inoltre, si pensa che molte ne nasceranno ancora nei prossimi anni. In questo ambito, le software house tendono tipicamente a progettare e realizzare le apps sapendo a priori quale tecnologia useranno per la loro implementazione. Questo approccio ha come conseguenza diretta la perdita di contatto col problema nella sua essenza, in quanto l'attenzione tende a focalizzarsi direttamente sul progetto nella particolare tecnologia di interesse. L'analisi del problema e la progettazione finiscono per ridursi ad un'unica fase, contemporanea in alcuni casi anche all'implementazione, con notevoli inconvenienti per quanto riguarda la pianificazione del lavoro e con la quasi impossibilitá di scindere le considerazioni di analista e progettista al fine di poter valutare alternative al progetto. Capita di frequente che, in assenza di un'analisi indipendente dalla tecnologia, l'architettura logica e i problemi evidenziati per una piattaforma non siano piú validi per un'altra; ma, qualora non si sentisse l'esigenza di fare il porting del prodotto, questo non costituirebbe alcun tipo di problema. Tuttavia, oggigiorno l'esigenza di realizzare applicazioni multi-piattaforma é molto forte, come testimonia anche la nascita di ambienti, come PhoneGap, capaci di tradurre automaticamente

un progetto scritto in HTML, CSS e JavaScript in diverse tecnologie mobile. Eppure, dal momento che HTML, CSS e JavaScript sono esse stesse delle tecnologie, il rischio é che concettualmente non cambi nulla, in quanto si finirebbe con buona probabilitá per non cogliere ugualmente la vera essenza del problema, generando un software non portabile in ambienti diversi da quelli supportati da PhoneGap. Si profilano quindi due alternative percorribili quando si possiede un progetto platform-dependent e si ha la necessitá di portarlo su un'altra tecnologia: o si ricomincia daccapo riprogettando totalmente il sistema nel nuovo ambiente di interesse; oppure si cerca di astrarre dalla particolare piattaforma che si andrá ad utilizzare, cogliendo solamente le caratteristiche fondamentali del problema in questione, descrivendolo mediante modelli.
In questo lavoro si analizzerá un caso di studio reale, che riflette lo scenario suddetto: un'applicazione da me realizzata in un primo momento in ambiente iOS seguendo un approccio di tipo code-based. In seguito, la necessitá di fare il porting del prodotto sulla piattaforma Android mi ha spinto ad analizzare criticamente il mio precedente operato, valutando la possibilitá di reingegnerizzare il sistema adottando un approccio di tipo model-driven. In particolare, nel corso dell'esposizione si partirá con la presentazione del caso di studio e del processo che mi ha condotto alla produzione dell'artefatto in iOS, del quale saranno poi evidenziati i limiti che mi hanno indotto a modificare la strategia progettuale. A tal proposito, ci si concentrerá sull'analisi e la modellazione di alcune delle funzionalitá del sistema, che faranno da campione. Si mostrerá infine come il modello del problema frutto del refactoring possa essere adattato alle due tecnologie, mettendo in luce in particolare i vantaggi dell'approccio model-driven.

Worldwide Mobile Communications Device Open OS Sales to End Users by OS (Thousands of Units)

OS	2010	2011	2012	2015
Symbian	111,577	89,930	32,666	661
Market Share (%)	37.6	19.2	5.2	0.1
Android	67,225	179,873	310,088	539,318
Market Share (%)	22.7	38.5	49.2	48.8
Research In Motion	47,452	62,600	79,335	122,864
Market Share (%)	16.0	13.4	12.6	11.1
iOS	46,598	90,560	118,848	189,924
Market Share (%)	15.7	19.4	18.9	17.2
Microsoft	12,378	26,346	68,156	215,998
Market Share (%)	4.2	5.6	10.8	19.5
Other Operating Systems	11,417.4	18,392.3	21,383.7	36,133.9
Market Share (%)	3.8	3.9	3.4	3.3
Total Market	**296,647**	**467,701**	**630,476**	**1,104,898**

Source: Gartner (April 2011)

Figura 1.1: La figura [1] mostra, per ogni sistema operativo mobile raffigurato, il volume delle vendite in migliaia di unitá di dispositivi relativamente agli anni 2010, 2011 e 2012, piú la previsione per l'anno 2015. Si puó notare che Android e iOS stiano attualmente dominando il settore, ma che Windows Phone e altre nuove tecnologie siano in forte ascesa e continueranno ad esserlo nei prossimi anni. Symbian invece risulta oramai destinato al declino.

Capitolo 2

Caso di studio: "myJourney"

Il caso di studio preso in considerazione è "myJourney", che come dicevamo è un'applicazione mobile che in un primo momento ho sviluppato in ambiente iOS. In seguito, la necessità di portarla sulla piattaforma Android mi ha spinto a rianalizzare criticamente il mio precedente operato, evidenziandone i limiti e cercando un approccio alternativo a quello seguito in precedenza.

2.1 Requisiti

Prima di spiegare cosa effettivamente poteva essere migliorato è opportuno mettere in luce i requisiti alla base di "myJourney".
Essa è una app molto utile a tutti coloro che partono per una vacanza o che vogliono avere novità sul viaggio che un'altra persona sta effettuando. Chi ha in programma di organizzare una gita può, previa registrazione e login, crearla all'interno dell'applicazione, attribuendole un nome e assegnandole un identificativo univoco, e elencare al sistema tutte le tappe che la costituiranno (le quali si possono aggiungere anche successivamente alla conferma della creazione). Come è ragionevole pensare, è possibile creare più viaggi, ma solo uno alla volta potrà essere attivato. In seguito all'attivazione, all'apertura dell'applicazione l'utente verrà geolocalizzato per permettere a tutti coloro che lo stanno seguendo di vedere la sua posizione corrente.

Inoltre, chi è in viaggio può lasciare commenti riguardanti i luoghi che sta visitando, scattare foto, commentarle e vedere i commenti lasciati dagli amici che lo stanno seguendo; infine, quando si giunge al termine della vacanza è possibile comunicarlo al sistema, che permetterà all'utente di attivare nuovi viaggi qualora espressamente richiesto.
Come già anticipato, non solo questo sistema è utile a chi decide di andare in vacanza, ma anche a chi vuole avere notizie inerenti ad un viaggio di un proprio conoscente, che può essere consultato semplicemente conoscendo il relativo identificativo, senza necessariamente essere autenticati; tuttavia, per commentare le foto scattate è necessario fare login. Chi segue un viaggio ha accesso alla relativa mappa, comprensiva delle tappe programmate inizialmente e aggiornate dopo ogni modifica (aggiunta da parte dell'utente o geolocalizzazione in posti differenti da quelli prestabiliti); può vedere e commentare (se loggato) tutte le foto, nonchè accedere alla cronologia di tutti gli eventi degni di nota (foto e commenti rilasciati dal viaggiatore).

2.2 Requisiti funzionali e non funzionali

Tutti quelli presentati finora possono essere racchiusi nella categoria dei requisiti funzionali, che descrivono le funzionalità del sistema software, in termini di servizi che esso deve fornire e di come deve reagire ad input e situazioni particolari. Il primo artefatto "myJourney" risultava funzionante e perfettamente coerente con i requisiti funzionali. Ciononostante, essi di per sè non sono sufficienti se si mira alla produzione di un software di qualità, che, per essere definito tale, deve rispettare anche i cosiddetti requisiti non funzionali (vedi Figura 2.1). Come vedremo più avanti nel corso dell'esposizione, "myJourney", essendo stata progettata unicamente nell'ottica dell'ambiente iOS, risulta poco portabile in altre piattaforme come ad esempio Android e ha enormi problemi di interoperabilità col mondo esterno (GUIs, server o altre entità differenti dal cuore dell'applicazione).

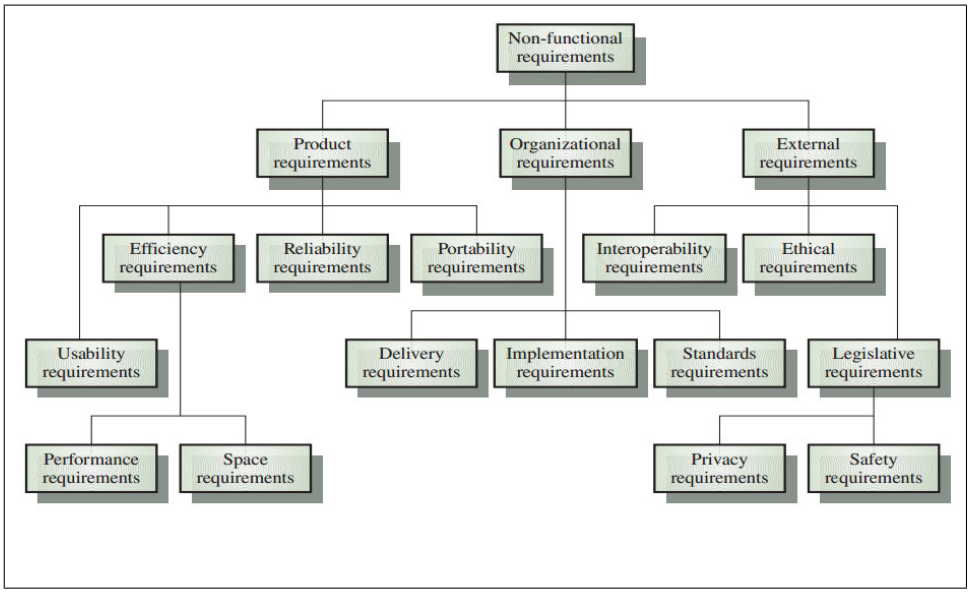

Figura 2.1: La figura mostra una classificazione dei requisiti non funzionali. Fonte: Università di Roma [2].

2.3 Caratteristiche primo artefatto "myJourney"

Dal momento che l'app oggetto dell'analisi di questo lavoro è nata in ambiente iOS, ci si aspetterebbe che essa sia improntata al pattern Model View Controller, che è pervasivo all'interno di questo ambiente di programmazione; invece in "myJourney" i tre elementi risultano non essere ben definiti.
Il Model è totalmente assente: all'interno dell'applicazione non sono individuabili entità del dominio. I dati che provengono dalle GUIs vengono direttamente utilizzati dentro il sistema oppure inviati al server, senza passare per la mediazione di un'entità che abbia quantomeno il compito di validarli, dando ad essi un contesto e una semantica. Così pure i dati che giungono dal server, che vengono direttamente visualizzati sulle GUIs senza salvataggio.
Dal canto suo, il Controller dovrebbe essere quella parte del sistema incaricata di interagire con le View, facendo in modo che le modifiche del Model vengano correttamente visualizzate su di esse. In "my-

Journey" questa mansione viene ricoperta dai View Controllers, che tuttavia non si limitano al loro compito, svolgendo anche numerose funzioni non di loro competenza, quali:

- implementare direttamente le funzionalità dell'applicazione;
- raccogliere i dati in input dalle GUI;
- gestire le connessioni col server;
- definire la rappresentazione dei dati scambiati col server;
- effettuare controlli utili a validare la consistenza dei dati provenienti dalle GUIs e/o dal server.

In definitiva, il prototipo realizzato risulta totalmente code-based, con la diretta e grave conseguenza di una assoluta mancanza di flessibilità, che comporta il rischio di una quasi totale reingegnerizzazione del sistema qualora dovesse cambiare qualche requisito. Si tenga presente che la flessibilità è figlia di una buona organizzazione della struttura del software, il quale dovrebbe essere costituito da due dimensioni ortogonali tra loro: la computazione interna all'applicazione e l'interazione con il mondo esterno. La sfera computazionale è opportuno sia svolta da componenti, ognuno dei quali avente funzioni ben distinte dagli altri; inoltre, è bene che essi non si occupino delle interazioni col mondo esterno, in modo da esserne indipendenti, di qualunque natura esso sia. Invece in "myJourney" tutto è svolto dai ViewControllers e, quindi, computazioni e interazioni appaiono fuse tra loro, a tal punto che è impossibile individuare componenti con precise funzioni. "myJourney" risulta non solo fortemente technology-dependent, ma anche dipendente dalle GUIs e dal particolare server con cui interagisce; non è quindi pensabile poter portare il prototipo funzionante in iOS su altre piattaforme senza prima averlo riprogettato completamente (vedi lo schema in figura 2.2).

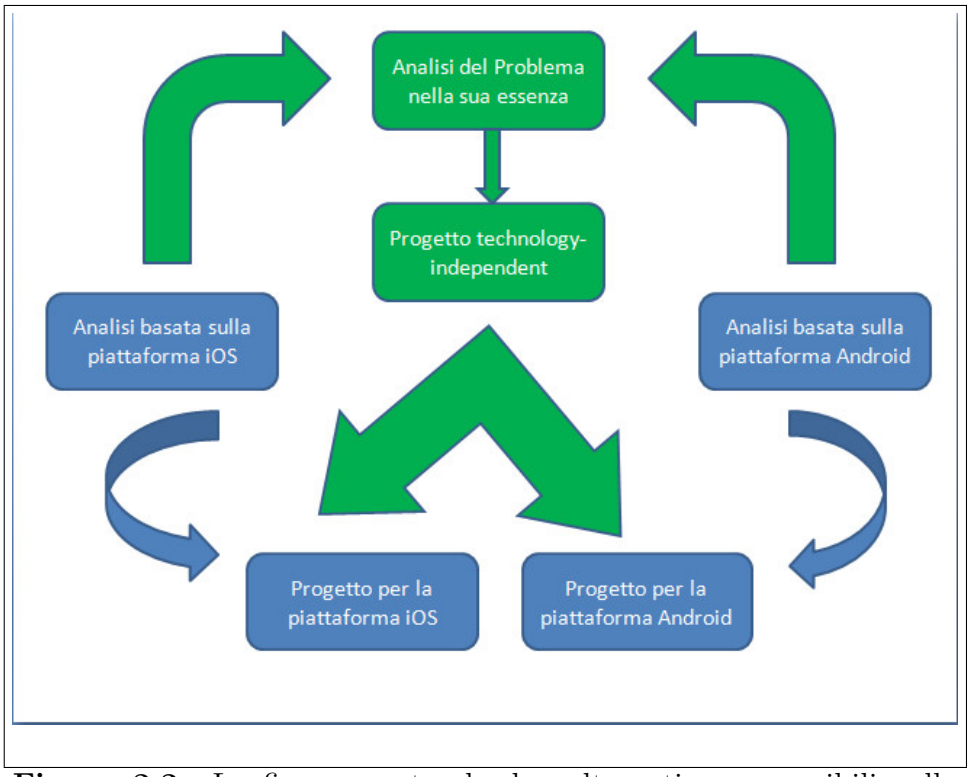

Figura 2.2: La figura mostra le due alternative percorribili nello sviluppo di un sistema software: o lo si analizza e progetta per la particolare tecnologia di interesse (come visibile dalle figure in blu), con la necessità di ripartire daccapo ogniqualvolta si debba produrre lo stesso artefatto per un'altra tecnologia; oppure ci si eleva ad un livello più astratto (riportato dalle figure in verde), capace di cogliere le caratteristiche del problema nella sua essenza, per giungere poi ad una progettazione il più possibile technology-independent, mappabile così con relativa semplicità nelle varie tecnologie di interesse. E' quest'ultimo l'approccio che si seguirà e di cui si evidenzieranno i vantaggi in questo lavoro.

Capitolo 3

Reingegnerizzazione di "myJourney"

L'obiettivo di questo lavoro è ora quello di riaffrontare il problema con un approccio model-driven, valutandone i vantaggi rispetto alla precedente soluzione code-based, al fine dell'abilitazione del porting. Concentriamo la nostra attenzione solo su alcune delle funzionalità evidenziate, che faranno da campione. In particolare, prendiamo in analisi l'operazione di login, quella di creazione di un viaggio e quella di recupero di un viaggio precedentemente creato.

3.1 Analisi dei requisiti

3.1.1 Casi d'uso

Per prima cosa riportiamo di seguito due Use Case Diagram (Figure 3.1 e 3.2) che mettono in luce le funzionalità del sistema a cui un agente esterno, come ad esempio un utente grazie alla mediazione di GUIs, può accedere.

3.1.2 Domain model

Ricordando i requisiti precedentemente esposti e tenendo conto delle funzionalità prese in considerazione in questo lavoro, possono essere

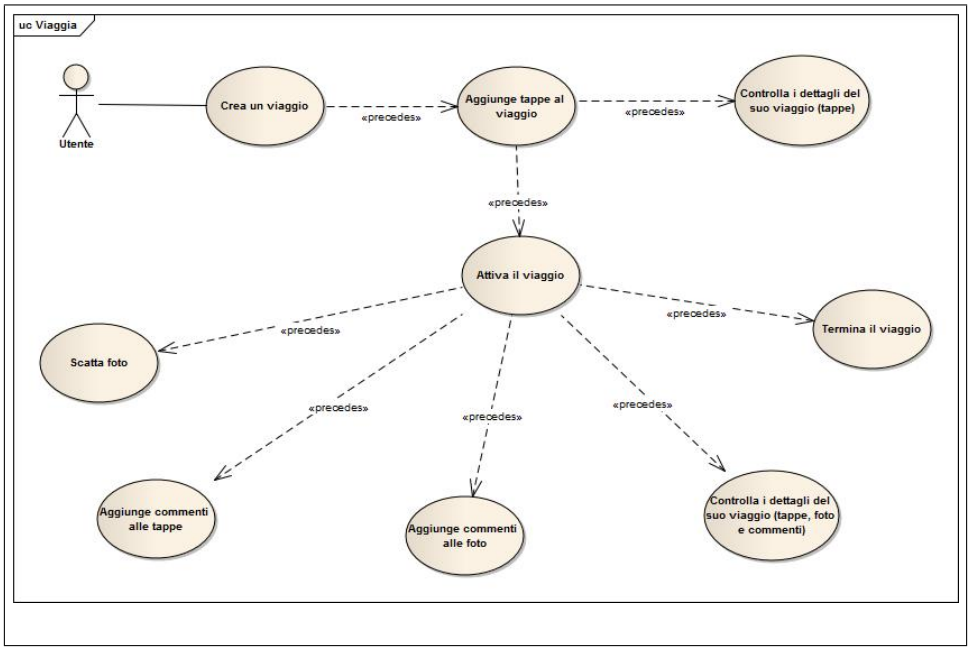

Figura 3.1: Il diagramma mostra la sequenza di operazioni che un utente può svolgere durante e dopo la creazione di un viaggio: prima crea il viaggio e ne aggiunge le tappe; poi, se ancora non lo ha attivato, può comunque consultarne il promemoria, altrimenti può anche scattare foto e aggiungere commenti, finchè non deciderà di terminare il viaggio.

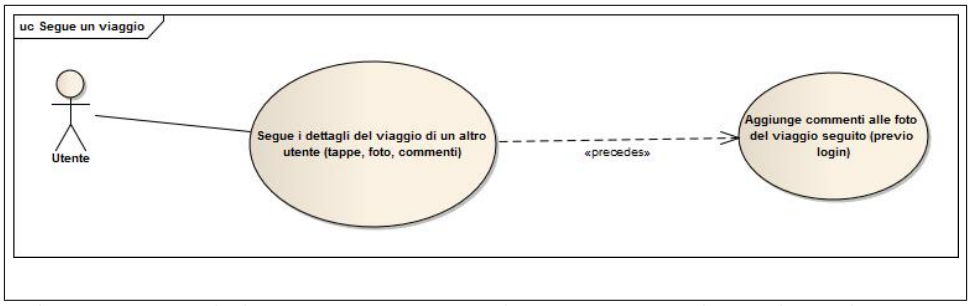

Figura 3.2: Il diagramma mostra le operazioni che può svolgere un utente che sta seguendo il viaggio di un amico: è possibile seguirne i dettagli e, se loggati, anche commentare le foto.

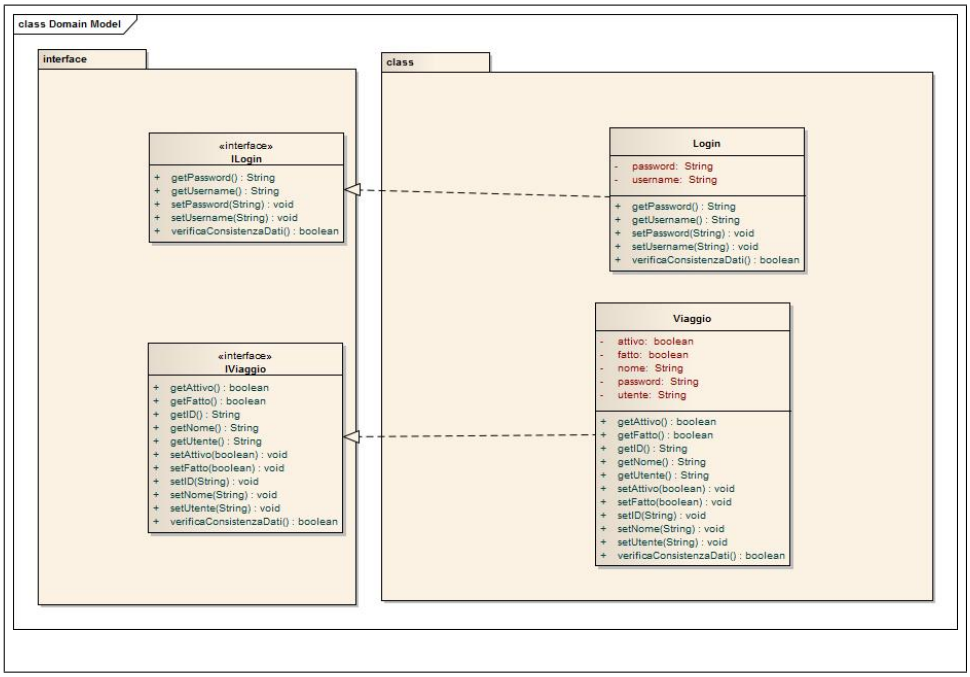

Figura 3.3: Diagramma UML rappresentante il domain model. La figura mostra le entità emergenti dai requisiti e l'elenco delle loro funzionalità.

messe in evidenza due entità di dominio fondamentali: Login e Viaggio. Queste sono in grado di salvare e recuperare le informazioni riguardanti rispettivamente le credenziali di autenticazione di un utente e i dati di base tramite i quali è possibile caratterizzare un viaggio; inoltre, esse possono in ogni momento valutare la consistenza del proprio stato interno. Login e Viaggio vanno così a formare il domain model, che nella logica MVC (Model View Controller) si incarna nel Model, ovvero la parte di sistema che fattorizza importanti proprietà e dati utili all'applicazione, totalmente assente nel primo prototipo di "myJourney".
Per capire cosa le entità del dominio dovranno essere in grado di fare si può fare riferimento ai diagrammi UML in figura 3.3 , che evidenziano la lista delle loro operazioni; mentre per comprendere meglio la semantica di tali funzionalità si rimanda ai piani di testing in allegato, di cui si riporta uno stralcio nelle figure 3.4 e 3.5.

```
public final void testVerificaConsistenzaDati1(){
    try{
        login.setUsername("andrea");
        login.setPassword("pagliarani");
    }catch(Exception e){}
    assertTrue("testVerificaConsistenzaDati1", login.verificaConsistenzaDati());
}
```

Figura 3.4: Il piano di testing in figura ci dice che i dati attualmente presenti all'interno dell'entità Login sono consistenti se prima qualcuno ha opportunamente settato i campi username e password. Altri piani di testing sono invece atti alla verifica che se qualche pezzo di informazione fosse assente i dati non risulterebbero più consistenti.

```
public final void testVerificaConsistenzaDati1(){
    try{
        viaggio.setUtente("andrea");
        viaggio.setID("gita");
        viaggio.setNome("londra");
    }catch(Exception e){}
    assertTrue("testVerificaConsistenzaDati1", viaggio.verificaConsistenzaDati());
}
```

Figura 3.5: Il piano di testing in figura ci dice che i dati attualmente presenti all'interno dell'entità Viaggio sono consistenti se prima qualcuno ha opportunamente settato i campi utente, id e nome. Se qualche pezzo di informazione tra quelli suddetti non fosse presente i dati perderebbero la loro consistenza. Negli altri piani di testing è possibile vedere che un Viaggio è consistente anche se si settano opportunamente i campi fatto e attivo, che non dovranno mai essere posti entrambi a true.

3.2 Analisi del problema

Da un'analisi accurata del problema emerge che l'applicazione in questione, oltre a racchiudere in sè numerose funzionalità, ha il compito di interagire con due entità esterne: una GUI e un server. L'obiettivo è quello di mantenere il più possibile l'indipendenza tra la computazione interna all'applicazione e l'interazione con il mondo esterno.

3.2.1 Business logic

In questi casi la prima cosa da fare è concentrarsi su cosa deve fare l'applicazione e su come il problema spinge ad organizzare la sua business logic, ovvero l'insieme delle funzioni logiche che nel loro complesso la caratterizzano, demandando ad un secondo momento la gestione delle interazioni con gli agenti esterni. Come già sottolineato in analisi dei requisiti, è emerso che le entità fondamentali del sistema sono Login e Viaggio: proviamo ad analizzarle separatamente, per cercare di capire chi dovrà interagire con esse e in seguito a quali circostanze.

Login

L'entità Login non nasce nello stesso istante in cui viene avviato il sistema, ma dovrà essere creata on demand nel momento in cui un agente esterno (nel nostro caso un utente) tenti di autenticarsi. Pertanto, il problema induce a pensare, con l'aiuto del pattern Factory, ad un altro soggetto incaricato di creare Login solamente quando necessario (vedi Figura 3.6): tale entità sarà ribattezzata CreatoreLogin. E' importante rimarcare che per il solo fatto che si è sentita l'esigenza di introdurre un CreatoreLogin, il sistema ora si compone di un altro tassello, che è in grado, quando qualcuno lo esige, di costruire un Login e che risulta quindi essere un qualcosa di totalmente nuovo rispetto a quanto era stato espressamente richiesto dai requisiti. Tuttavia, l'inserimento di questa nuova entità porta con sè due problemi che in precedenza non erano nemmeno stati considerati e che hanno entrambi importanza capitale:

- La factory dove trova i dati che le occorrono per creare Login?

- Che rappresentazione devono avere i dati all'interno dell'applicazione?

Cerchiamo innanzitutto di risolvere la prima questione, lasciando momentaneamente il secondo quesito in sospeso.

Come strutturare le Factory

Esistono due alternative per fare in modo che la factory crei il suo prodotto finale:

- fornirle tutti i dati necessari alla creazione dell'entità;
- ordinarle di creare l'entità lasciando che essa si procuri autonomamente i dati di cui ha bisogno.

Nel primo caso avremo bisogno di una nuova entità, che sia in grado di procurarsi i dati di cui la factory ha bisogno per svolgere il suo compito e che abbia un riferimento ad essa. Nel secondo caso invece sarà CreatoreLogin stesso a dover interagire con un'altra parte del sistema capace di recuperare gli elementi costitutivi del Login. Quest'ultima alternativa ha il notevole vantaggio che la factory, se qualcosa dovesse andare storto prima o durante la creazione, potrebbe interromperla, possibilmente senza aver recuperato inutilmente tutti i dati, cosa che sarebbe impossibile scegliendo la prima opzione, in cui essi verrebbero procacciati indipendentemente dalla buona riuscita della successiva operazione di creazione. D'altra parte, adottando la seconda possibilità avremmo una dipendenza diretta della factory dall'entità in grado di recuperare i dati; tuttavia, è noto dai requisiti che essi giungeranno da una GUI o, volendo essere più generali possibile, proverranno comunque dall'esterno dell'applicazione. Pertanto, il legame che si avrebbe è un prezzo da pagare più che ragionevole, perchè lega la factory ad un'altra parte del sistema, che tuttavia non fa parte della business logic, la qual cosa graverebbe sulla flessibilità, ma è un ente che regola le interazioni col mondo esterno, di qualunque natura esso sia. Infatti, come dicevamo noi sappiamo che i dati saranno recuperati tramite l'interazione di un utente con una GUI, ma nulla vieta di pensare che un domani essi possano giungere dalla rete o in un altro modo ancora. Questo spinge a valutare la possibilità, sfruttando il

pattern Adapter, di avere tanti adapter, i quali avranno un'interfaccia immutabile, cosicchè CreatoreLogin possa rivolgersi a loro in modo standard; ognuno di essi poi avrà il compito di interfacciare il sistema col particolare agente esterno in questione, recuperando i dati richiesti da una GUI piuttosto che da remoto o da altre fonti.

Viaggio e Applicazione

L'altra entità fondamentale di "myJourney" emersa dall'analisi dei requisiti è il Viaggio, per il quale le considerazioni da fare sono in parte differenti da quelle già discusse per il Login. Infatti, un Viaggio (inteso come l'entità descritta in analisi dei requisiti) andrà istanziato in due particolari situazioni: quando qualcuno direttamente ne richiede la creazione interagendo con l'applicazione; oppure quando il sistema vuole recuperare dall'esterno un Viaggio già precedentemente creato. La seconda esigenza risponde al problema che "myJourney" è un'applicazione mobile e, come tale, destinata a girare su dispositivi aventi una disponibilità di memoria relativamente ridotta; per questo motivo è bene che il salvataggio di dati in locale sia minimo e, di conseguenza, è preferibile che essi siano per lo più memorizzati in remoto.
Le due situazioni sono molto diverse tra loro, ma al contempo si può notare la presenza di una funzionalità fattorizzabile, che è appunto quella tramite la quale viene istanziato un nuovo Viaggio a partire dai dati che lo compongono. Questo compito viene gestito dall'entità che in questo lavoro è stata ribattezzata CreatoreViaggio, che, analogamente a CreatoreLogin, ha bisogno di recuperare alcuni dati per fare a dovere il proprio lavoro. Perciò, anche in questo caso, memori delle considerazioni precedenti, avremo un opportuno adapter che verrà utilizzato a questo scopo.
Uno dei requisiti fondamentali per la creazione di un Viaggio è che un utente deve essere necessariamente autenticato per poter portare a compimento con successo l'operazione; detto in altri termini, Login deve essere inevitabilmente istanziato prima di Viaggio. La gestione di questo aspetto esula dai compiti della factory, che riceve il Login all'atto della creazione e si limita ad utilizzarlo al fine di produrre l'entità Viaggio; qualora non le fosse assegnato un Login, CreatoreViaggio non riuscirebbe a portare a termine l'operazione, ma si limiterebbe a

segnalarlo, senza introdurre politiche in grado di arginare il problema. Risulta piuttosto chiaro che tali procedimenti dovranno essere attuati da un'altra entità, il cui compito sarà quello di fare da collante tra i componenti del sistema, coordinandoli in modo tale da soddisfare lo scopo ultimo per il quale esso è stato creato. Per questi motivi il problema ci impone di immettere nel sistema un nuovo ente, a cui è stato assegnato il nome di Applicazione (vedi Figura 3.7), che offrirà alle factory gli adapter di cui hanno bisogno e che coordinerà le loro attività. L'Applicazione imporrà l'autenticazione nel caso in cui dovesse essere richiesta la creazione di un viaggio prima di aver fatto login. Si noti come questa sia a tutti gli effetti una politica di sistema e, come tale, non implementabile da un componente che si occupa solamente di una ben determinata funzionalità. Si noti altresì che è sempre il problema che ci spinge a trovare delle contromisure, perchè è opportuno che non solo l'intera applicazione, ma anche le singole operazioni non falliscano quando evitabile a priori; il sistema deve risultare il più possibile human-friendly.
Tornando a livello di funzionalità, sarà sempre l'entità Applicazione a far sì che la creazione di un nuovo viaggio differisca dal recupero di uno già esistente. Infatti, in entrambi i casi la factory CreatoreViaggio si preoccupa solamente di produrre una nuova entità Viaggio, ignorando completamente la natura della sorgente che le ha fornito i dati e restando anche all'oscuro di come il Viaggio sarà utilizzato da altri componenti del sistema. Quindi, in base a ciò che dicono i requisiti, Applicazione procurerà alla factory un adapter in grado di recuperare i dati in input da una GUI nel caso in cui si vorrà creare un viaggio totalmente nuovo o un adapter che interagirà con un server qualora si voglia recuperare un viaggio già precedentemente creato ma attualmente non presente nella memoria interna al sistema.

3.2.2 La rappresentazione dei dati

Come già avevamo evidenziato in precedenza, un altro problema da risolvere riguarda quale struttura devono avere i dati all'interno dell'applicazione. Questo tema è di fondamentale importanza, in quanto, perchè l'esecuzione di ogni funzionalità vada a buon fine, tra le varie entità del sistema devono essere scambiati diversi dati. Quindi,

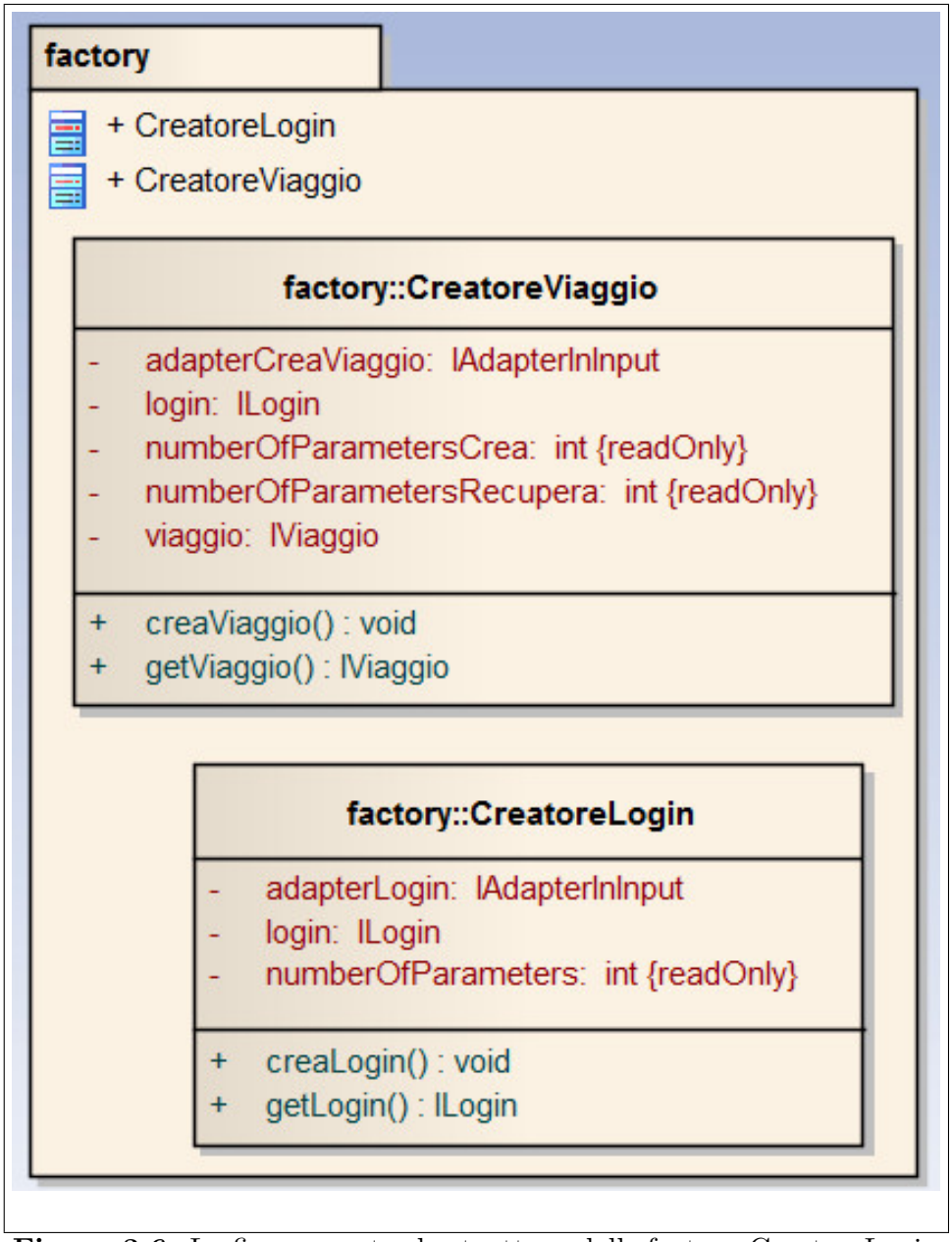

Figura 3.6: La figura mostra la struttura delle factory CreatoreLogin e CreatoreViaggio.

Figura 3.7: La figura mostra la struttura dell'entità Applicazione, che funge da collante tra i vari componenti del sistema.

una cosa già appare ovvia: qualunque sia la loro formattazione, essa deve poter essere compresa da tutti gli enti che entrano in gioco. Per questo motivo è opportuno che vi sia una classe di supporto in grado di regolare la rappresentazione dei dati, in modo tale che ogni classe del sistema, invece di introdurre proprie politiche, facilmente mal interpretabili dalle altre, deleghi ad essa questa incombenza. E' importante rimarcare che questa è una classe di sistema, statica e riferibile da tutte le altre. Di rilevanza ancor maggiore è sottolinearne il compito: infatti, questa classe, che in "myJourney" abbiamo ribattezzato Formatter (vedi Figura 3.8), incapsula al suo interno un vero e proprio linguaggio di rappresentazione dei dati, che può essere liberamente utilizzato dalle varie entità senza che esse ne conoscano i dettagli implementativi interni. In questo lavoro, quando un'entità necessita di più dati per eseguire correttamente il proprio compito (ad esempio una factory), si è pensato di inviarglieli contemporaneamente; pertanto, dovranno esserci dei separatori che indichino ad essa il confine tra un'unità di informazione ed un'altra. Formatter mette quindi a disposizione delle operazioni che permettono di separare i dati o comporli opportunamente qualora necessario.

3.2.3 La suddivisione in package

Nel corso dell'analisi appena condotta sono state messe in luce numerose entità, ognuna con dei compiti ben precisi e distinti da quelli delle altre. Tenendo conto di quanto detto, se ne possono delineare più tipologie, ciascuna inglobabile all'interno di un package (vedi Figura 3.9):

- le domain entity (Login e Viaggio);
- le factory (CreatoreLogin e CreatoreViaggio);
- la glue del sistema (Applicazione);
- gli adapter (IAdapterInInput e IAdapterInOutput);
- i support (Formatter).

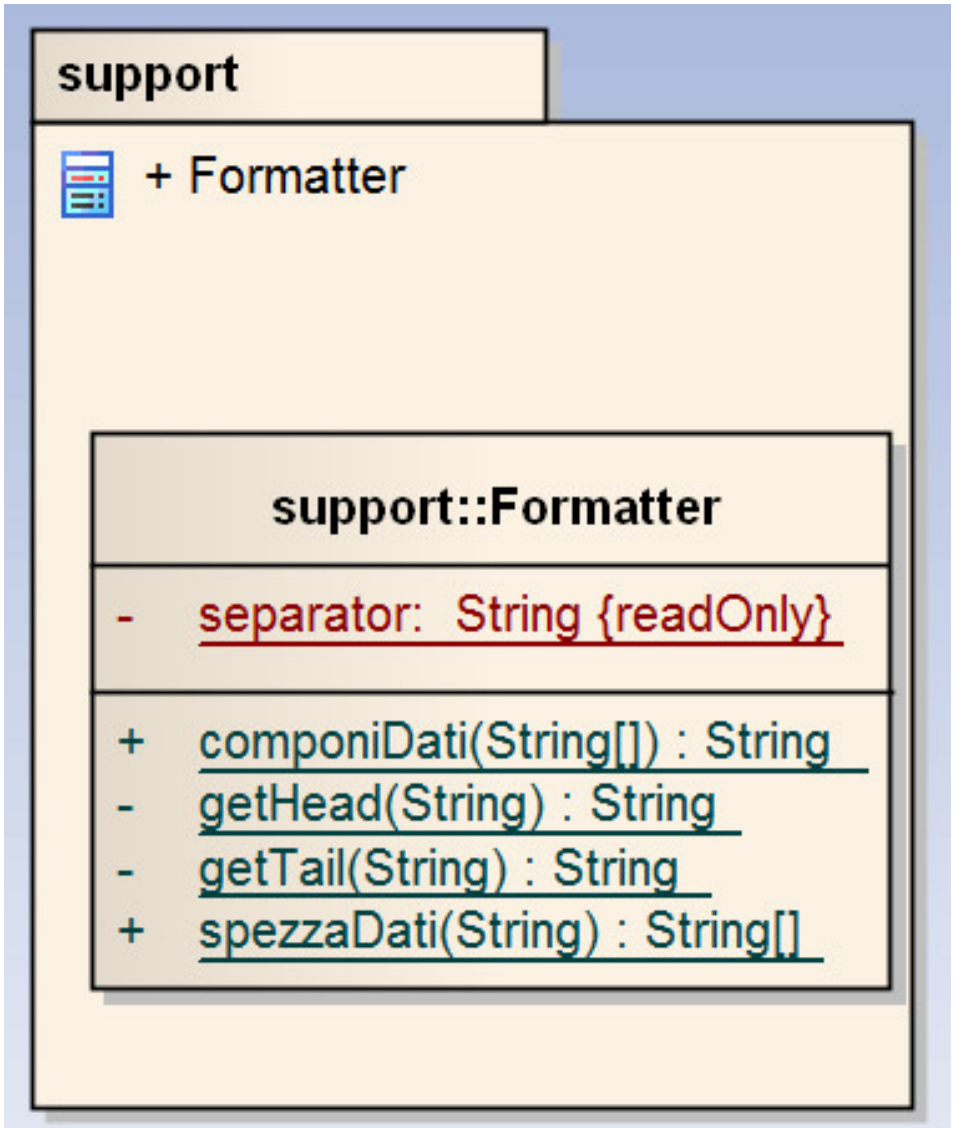

Figura 3.8: La figura mostra la struttura dell'entità Formatter, ovvero la classe di sistema che definisce la rappresentazione dei dati.

Le domain entity costituiscono il Model, ovvero la rappresentazione formale dei dati che saranno inseriti, scambiati e recuperati coerentemente con gli obiettivi dell'applicazione. Le factory sono le uniche entità aventi la facoltà di istanziare una domain entity quando necessario. La glue si occupa di mantenere consistente lo stato dell'intero sistema, coordinando le attività delle singole sottoparti che realizzano le varie funzionalità. Gli adapter fungono da interfaccia tra "myJourney" e il mondo esterno e, quindi, permettono di rendere l'applicazione indipendente dal tipo di entità che vorranno interagire con essa e dal tipo di comunicazione che dovrà instaurarsi tra loro (come risulterà più chiaro quando si parlerà nel dettaglio dell'interazione tra app e agenti esterni). I support forniscono delle regole valide per tutti i componenti del sistema e tassative per il suo corretto funzionamento. La suddivisione in package è un ottimo metodo per ripartire concettualmente i compiti di ogni parte del sistema; in questo modo risulta anche più semplice il collaudo poichè, delineando le dipendenze tra i vari package, è possibile testare separatamente le varie parti e, una volta che una di esse si dimostra essere funzionante, non sarà più necessario apportarle modifiche e ci si potrà concentrare anche sulle unità che dipendono direttamente o indirettamente da essa. Inoltre, la separazione attuata mostra chiaramente la business logic dell'applicazione, che è costituita da domain entity, factory e glue, con l'aggiunta dei support.

3.2.4 Pianificazione del collaudo

Per collaudo si intende una serie di operazioni messe in atto in maniera tale da verificare il corretto funzionamento del sistema; esso costituisce la parte finale della produzione dell'artefatto, tramite la quale avviene la validazione del prodotto. Tuttavia, è bene ricordare che tutte le considerazioni precedenti sono frutto dell'analisi del problema, che ha evidenziato solo cosa il sistema è in grado di fare (quella che in Ingegneria del software è definita architettura logica), prima ancora che abbia inizio la sua effettiva realizzazione. La modellazione UML è molto utile a fornire un'idea su quali saranno struttura, interazione e comportamento del sistema nel suo complesso e anche dei suoi componenti costitutivi; nonostante ciò, in UML è impossibile dare una

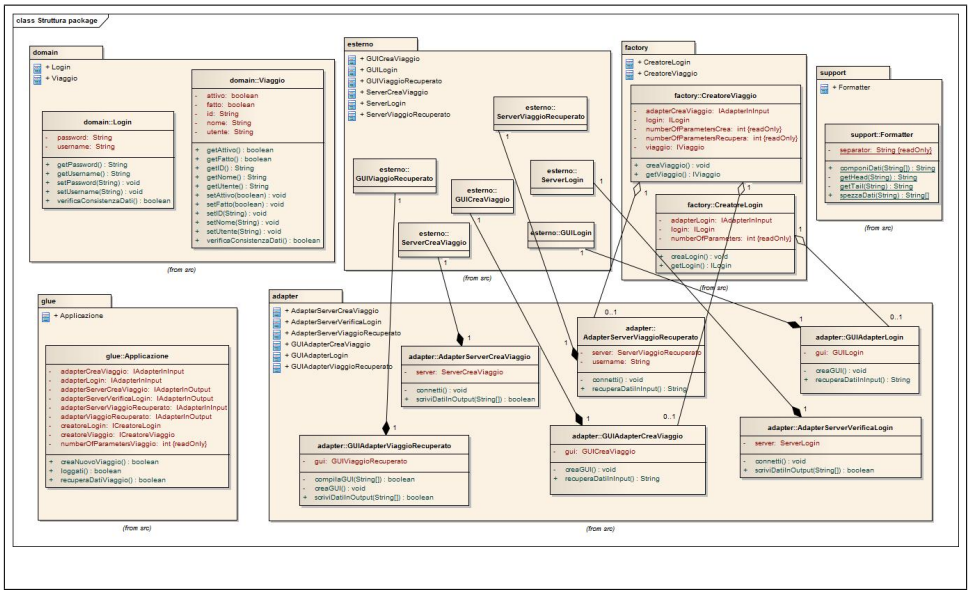

Figura 3.9: La figura mostra la struttura dei package scaturenti dal problema e le relazioni tra le entità. Notare che le factory non sono vincolate ad essere legate necessariamente agli adapter raffigurati, in quanto l'unica cosa importante è che siano sempre agganciate ad un IAdapterInInput, indipendentemente dal fatto che esso faccia da ponte con un server, una GUI o altro. Invece, gli adapter, qualunque sia la loro natura, per quanto detto devono essere in 1-1 col mondo esterno e devono collegare la business logic dell'applicazione con esso.

semantica alle operazioni. Per colmare questi limiti di espressività nasce quella che è chiamata pianificazione del collaudo, che formalizza ulteriormente ciò che già era apparso nei diagrammi, stabilendo in maniera più chiara cosa ci si aspetta da ogni singola entità e da ogni sua singola operazione.
Come già osservato, in analisi dei requisiti erano già stati messi a punto i piani di collaudo per le domain entity. A questo livello, invece, ciò che ci interessa maggiormente è la validazione delle factory e dell'entità che le coordina. Infatti, il fine dei piani di testing è la verifica della business logic dell'applicazione; una volta che essa funziona resterà solamente il problema di metterla in comunicazione col mondo esterno, progettando gli opportuni adapter utili a questo scopo. In fase di analisi però le interazioni con l'esterno non sono rilevanti e quindi è possibile simularle tramite l'utilizzo di mock objects, ossia entità in grado di riprodurre il comportamento di quelle che effettivamente svolgeranno il loro compito una volta che il sistema sarà ultimato. Nel caso affrontato in questo lavoro, sono stati usati dei mock objects al posto delle GUIs e dei server, i quali restituiscono risultati cablati, atti alla verifica della condotta dell'applicazione. Gli adapter, da parte loro, permettono soltanto il traferimento dei dati tra l'app e il mondo esterno, costituito appunto da tali mock objects. Per queste ragioni, le uniche parti per cui pianificare realmente i testing sono le factory e l'Applicazione, passando anche per la validazione di Formatter.
A questo punto una precisazione appare doverosa: mentre quelli descritti in fase di analisi dei requisiti sono degli Unit Testing, quelli di cui abbiamo appena parlato in analisi del problema, oltre ad essere Unit Testing, sono veri e propri Integration Testing. A renderli profondamente diversi è il fatto che i primi servono a testare singole parti del sistema, indipendenti dalle altre; invece, gli Integration Testing combinano più Unit Testing al fine di testare il loro funzionamento complessivo, dipendente dalle politiche del sistema. In particolare, le factory sono sia Unit Testing che Integration Testing, in quanto sono unità del nostro sistema che comunque hanno il compito di creare entità del dominio, che perciò devono già essere collaudate e funzionanti. Invece, l'Applicazione è a tutti gli effetti un Integration Testing, perchè assembla in sè le varie funzionalità di "myJourney", che devono operare sia singolarmente che nel loro insieme.

```
public final void testSpezzaDati1(){
    String separator=";;";
    String primo="a";
    String secondo="b";
    String datiUniti=primo+separator+secondo;
    String[] datiSpezzati=Formatter.spezzaDati(datiUniti);
    assertTrue("testSpezzaDati1", datiSpezzati[0].equals(primo) &&
                                    datiSpezzati[1].equals(secondo));
}

public final void testComponiDati1(){
    String primo="a";
    String secondo="b";
    String[] datiSpezzati=new String[]{primo, secondo};
    String datiUniti=Formatter.componiDati(datiSpezzati);
    assertTrue("testComponiDati1", datiUniti.equals(primo+";;"+secondo));
}
```

Figura 3.10: La figura mostra la semantica delle operazioni dell'entità Formatter. spezzaDati() prende in ingresso una stringa e produce un array di stringhe ottenuto spezzando la stringa in input ogniqualvolta incontra i caratteri usati come separatori, ovvero ";;". componiDati() fa esattamente l'operazione opposta. Oltre a questi, vi sono altri piani di testing che definiscono meglio il comportamento delle due funzioni che, se non ricevono almeno un parametro in ingresso diverso da una stringa vuota, restituiscono un riferimento a null.

Nelle figure 3.10, 3.11, 3.12 e 3.13 è riportato qualche esempio della pianificazione del collaudo delle entità che costituiscono la business logic dell'applicazione. L'intero piano di testing ricordiamo essere consultabile in allegato a questo lavoro.

3.2.5 L'interazione con il mondo esterno

Una volta sistemata la business logic dell'applicazione, è possibile curarne le interazioni col mondo esterno. E' bene pensare che il nostro obiettivo consista nel rendere il sistema technology-independent e, perchè lo sia, non è sufficiente una logica di business accuratamente progettata. Infatti, come già accennato in precedenza, occorrono degli adapter con i quali l'applicazione dovrà interagire; questi saranno in mapping biunivoco con la particolare entità esterna che in quel determinato momento è incaricata di svolgere la funzione richiesta. Questa

```
public final void testCreaLogin(){
    String username="andrea";
    String password="pagliarani";
    try {
        creatoreLogin.creaLogin();
    } catch (Exception e) {
        fail();
    }
    assertTrue("testCreaLogin", creatoreLogin.getLogin().getUsername().equals(username) &&
                                creatoreLogin.getLogin().getPassword().equals(password));
    try {
        creatoreLogin.creaLogin();
        fail();
    } catch (Exception e) {
        assertTrue("testCreaLogin", true);
    }

}
```

Figura 3.11: La figura mostra il significato dell'operazione creaLogin() della factory CreatoreLogin. La prima volta che viene chiamata essa dovrebbe restituire un Login caratterizzato dall'username e dalla password recuperati in input, che nel caso specifico sono cablati in un mock object e il cui contenuto corrisponde a quello delle variabili "username" e "password" dichiarate all'inizio del test. Se poi per qualche motivo venisse chiesta nuovamente alla factory la creazione di un ulteriore Login, essa lancerebbe un'eccezione, in totale coerenza con il pattern Singleton, che le impone di non mantenere più di un Login attivo contemporaneamente.

```
public final void testRightCreaViaggio1(){
    String nome="londra";
    String id="gita";
    boolean fatto=false;
    boolean attivo=false;
    try {
        creatoreProprioViaggio.creaViaggio();
    } catch (Exception e) {
        fail();
    }
    assertTrue("testRightCreaViaggio1", creatoreProprioViaggio.getViaggio().getNome().equals(nome) &&
            creatoreProprioViaggio.getViaggio().getID().equals(id) &&
            creatoreProprioViaggio.getViaggio().getUtente().equals(login.getUsername()) &&
            creatoreProprioViaggio.getViaggio().getFatto()==fatto &&
            creatoreProprioViaggio.getViaggio().getAttivo()==attivo);
    try {
        creatoreProprioViaggio.creaViaggio();
        fail();
    } catch (Exception e) {
        assertTrue("testRightCreaViaggio1",true);
    }
}
```

Figura 3.12: La figura mostra il significato dell'operazione creaViaggio() della factory CreatoreViaggio. Essa sarà del tutto indipendente dalla fonte dei dati: infatti, qui la factory è stata chiamata "creatoreProprioViaggio" per indicare che siamo nel caso in cui si sta cercando di creare un Viaggio dall'interno dell'applicazione e, perciò, stando ai requisiti, i dati saranno recuperati da una GUI. Ma in allegato si può trovare un altro piano di testing pressochè identico per "creatoreViaggioRecuperato", un'altra istanza della factory, che avrà un riferimento non ad un adapter verso una GUI, bensì ad uno legato ad un server (con la medesima interfaccia). Considerando in entrambi i casi i dati immessi o recuperati (in questo momento da un mock object) coincidenti con il contenuto delle variabili "nome", "id", "fatto" e "attivo" dichiarate all'inizio del test, si può vedere che una prima invocazione di creaViaggio porti alla creazione di una nuova entità Viaggio, costituita dai pezzi di informazione suddetti, con in aggiunta l'username di colui che si era autenticato al sistema precedentemente. Invece, una seconda invocazione di creaViaggio causa il lancio di un'eccezione in quanto quella factory ha già esaurito il suo compito.

```
public final void testCreaNuovoViaggio(){
    assertTrue("testCreaNuovoViaggio", app.creaNuovoViaggio());
}

public final void testLoggati(){
    assertTrue("testLoggati", app.loggati());
}

public final void testRecuperaDatiViaggio(){
    assertTrue("testRecuperaDatiViaggio", app.recuperaDatiViaggio());
}
```

Figura 3.13: La figura mostra il piano di testing dell'entità Applicazione, che deve solo valutare che tutto vada a buon fine, essendo in grado di coordinare le azioni degli altri componenti del sistema. Tutte e tre le operazioni, se correttamente eseguite, daranno il valore "true" in output.

organizzazione comporta che, se muta il mondo esterno, dovranno farlo anche gli adapter, mentre il cuore del sistema, come abbiamo detto, dovrà restare inalterato. Perchè si possano raggiungere gli obiettivi gli adapter dovranno avere delle interfacce immutabili, in modo tale da permettere all'applicazione di accedere alle loro funzionalità in modo standard, qualunque sia l'adapter concreto attualmente in uso nel sistema. Poi sarà il particolare adapter correntemente utilizzato a realizzare l'effettivo collegamento con ciò che sta al di fuori dell'applicazione. Di tutta la logica interna, in questo modo, non verrà modificato quasi nulla, tranne l'entità Applicazione che, dovendo mantenere consistente lo stato dell'intero sistema, conoscerà con quali adapter concreti esso dovrà interfacciarsi e passerà un riferimento ad essi alle factory che lo necessitano. Quindi, la factory non sa e non le interessa sapere con quale adapter stia veramente interagendo; ciò che realmente è importante è che, qualsiasi entità sia, essa sappia erogare le funzionalità dichiarate nella propria interfaccia.
In "myJourney" avremo due diverse tipologie di adapter (vedi Figura 3.14):

- uno che sia in grado di recuperare dati in input;
- uno che sappia scrivere dati in output.

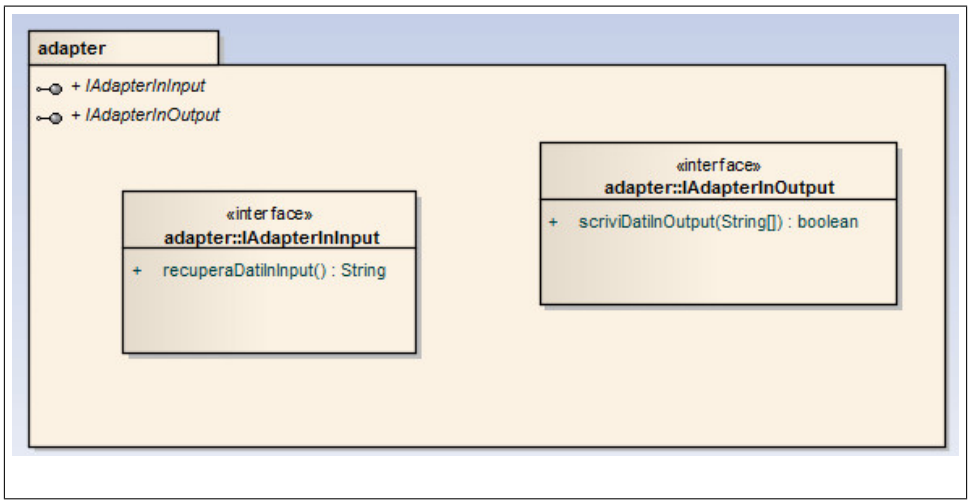

Figura 3.14: La figura mostra la struttura degli Adapter, ovvero le entità in grado di collegare la business logic dell'applicazione al mondo esterno.

Essi, a livello di interfaccia, saranno indipendenti dal fatto che i dati vengano letti/scritti da/su una GUI, un file, un'applicazione autonoma, un server (di qualunque tipo esso sia: jsp, servlet, pagina php, ecc.), ecc.

3.2.6 Dall'architettura logica al progetto

Una volta definita l'intera architettura logica del sistema si può proseguire passando alla progettazione. Ricordiamo che la prima ci descrive cosa esso dovrà essere in grado di fare, mentre solamente con la seconda saremo in grado di dire come queste cose dovranno effettivamente essere fatte. Il vantaggio di un'analisi accurata sta nel fatto che la maggior parte delle problematiche dovrebbero già essere state evidenziate ed arginate prima ancora di mettere mano al progetto; infatti, se possibile, sarebbe bene che almeno la business logic possa rimanere pressochè inalterata. In ogni caso, le scelte del progettista non devono alterare ciò che l'analista aveva affermato; se questo dovesse succedere non si potrebbe più procedere col progetto ma sarebbe necessario rivedere la fase di analisi del problema al fine di risolvere le incongruenze. Nelle applicazioni mobile capita spesso che l'architettura logica frutto

dell'analisi del problema sia già sufficiente a spiegare non solo il cosa, ma anche il come; detto altrimenti, la business logic raramente subisce delle variazioni nel progetto. Basti pensare al fatto che un sistema, per il solo fatto che deve girare su un dispositivo mobile, è soggetto a molte più problematiche (ad esempio la disponibilità ridotta di memoria) di quante non ce ne siano per una normale applicazione per computer desktop. Anche in "myJourney" si può dire che non siano necessarie rivoluzioni e, di fatto, possiamo considerare l'architettura logica emergente dal problema come architettura di progetto. Tuttavia, è importante aggiungere una precisazione, ovvero stabilire quali siano le parti del progetto technology-independent e quali invece quelle technology-dependent. A tal proposito, possiamo ritenere l'intera business logic, comprensiva di domain entity, factories, support e entità Applicazione, completamente technology-independent; invece, gli adapter risultano essere technology-dependent.
In seguito vedremo come mappare questo progetto su alcune piattaforme concrete, come ad esempio iOS e Android.

3.3 Vantaggi dell'approccio model-driven rispetto a quello code-based

Il primo approccio seguito nella produzione di "myJourney" lo possiamo definire quasi totalmente code-based, ovvero una soluzione che si articola sul codice della particolare tecnologia sulla quale si cercava di sviluppare l'artefatto. Questa strategia progettuale ha portato una confusione enorme. Infatti, la prima cosa che salta all'occhio è che non sono presenti entità del dominio e, di conseguenza, non vi è neanche l'ombra del Model. E' come se l'intera applicazione fosse un unico blocco, totalmente privo di flessibilità, in cui ogni parte risulta avere forti dipendenze da tante altre. Un solo cambio dei requisiti sarebbe una vera e propria catastrofe, perchè si dovrebbe mettere mano a tutto il sistema. Questo è già un prezzo che in molte situazioni (compreso in "myJourney") non si è disposti a pagare. A ciò si aggiunge anche che una soluzione code-based è totalmente technology-dependent; questa è stata la vera causa scatenante che mi ha spinto a cambiare radicalmente approccio, in quanto il primo artefatto era perfettamen-

te funzionante sulla piattaforma iOS, ma non era portabile. Nessun componente può essere mantenuto tale in altri ambienti, semplicemente perchè non vi sono componenti realmente presenti e distinguibili in modo netto dagli altri; non sono individuabili enti autonomi in grado di svolgere precisi compiti. Solo queste motivazioni risulterebbero sufficienti a motivare un cambio radicale di prospettiva, ma ad esse si può aggiungere anche un altro aspetto, che è molto fastidioso per chi dovrà implementare e collaudare il sistema. Infatti, terminata la progettazione, da cui emergerà l'architettura del sistema con tutti i suoi elementi, la successiva traduzione del progetto in una specifica piattaforma potrebbe portare ad avere delle difficoltà tecniche, anche notevoli se si è sviluppatori inesperti per un determinato ambiente di programmazione. Perciò potrebbe essere necessario fare del debug: in un sistema ben progettato ed organizzato in parti aventi ognuna una mansione particolare, risulta relativamente semplice capire quale/quali entità non sta/stanno funzionando a dovere, anche perchè ci si può servire della pianificazione del collaudo precedentemente effettuata. Ma, se non c'è un progetto, non ci sono entità e, pertanto, sono assenti anche i piani di collaudo, trovare un errore, anche il più banale, potrebbe essere un vero e proprio inferno. Come risultato, se in partenza si poteva essere convinti che sfruttando una metodologia agile il lavoro avrebbe occupato un tempo più breve, alla fine ci si può rendere conto di aver ottenuto l'esatto contrario, per di più avendo in mano un prodotto di scarsa qualità.
Per quanto detto si è adottato in seconda battuta un approccio model-driven, che vede la modellazione del sistema come il punto di partenza del processo di produzione del software. Utilizzando diagrammi UML è possibile produrre un modello del problema totalmente technology-independent, in cui si ha una descrizione sintattica (ricordiamo che la semantica viene aggiunta dai piani di testing) delle entità che, in primo luogo i requisiti e in secondo il problema, ci spingono ad introdurre in quella che costituirà la business logic dell'applicazione in questione. I vari componenti avranno ognuno il loro compito ben preciso e la loro composizione a livelli permetterà di erogare le funzionalità in modo tale che ciascuno svolga solamente la parte di propria competenza, demandando agli altri le restanti operazioni. Un chiaro esempio di questa organizzazione è il fatto che in "myJourney" le factory si preoccupino

solo di creare le domain entity, delegando gli adapter al recupero dei dati a loro necessari per lo svolgimento corretto del proprio lavoro. Dal modello del problema poi si articolerà il progetto, che dovrà essere effettuato cercando sempre di individuare entità ed elementi che possano non dipendere dalla particolare piattaforma di realizzazione. Chiaramente non è da escludere che si possano trovare parti del sistema inevitabilmente dipendenti dalla tecnologia, le quali andranno prima progettate e poi mappate, a mano o mediante l'utilizzo di un tool, sulle piattaforme di interesse. A tal proposito, non va sottovalutato un aspetto che tratteremo più approfonditamente nei prossimi capitoli: qualora sia necessario introdurre all'interno del sistema elementi technology-dependent, diventa fondamentale per il progettista avere una buona conoscenza dell'architettura in questione per capire quali parti potranno essere mantenute inalterate e quali invece andranno modificate affinchè tutto possa non solo funzionare, ma anche essere organizzato secondo i principi dell'Ingegneria del software.

Capitolo 4

Il porting in iOS

Dopo l'opera di reingegnerizzazione in chiave model-driven che ci ha condotto ad un'architettura ben definita, vediamo come muta la struttura del primo artefatto. Infatti, il nuovo progetto non solo sará utile per fare il porting su Android, che ricordiamo essere l'obiettivo primario che ha aperto la strada al refactoring, ma sará anche fondamentale per capire come modificare la precedente implementazione in iOS di "myJourney". Prima della realizzazione, peró, il progettista dovrá valutare come mappare l'architettura scaturita dal progetto su questa piattaforma. Risulta abbastanza chiaro che, in assenza di un tool capace di eseguire il mapping in iOS direttamente, colui che progetta il sistema dovrá per forza di cose avere una buona conoscenza della tecnologia in questione, di cui andremo ora a sottolineare alcune delle peculiaritá che dovranno essere considerate per poter eseguire il porting.

4.1 iOS

iOS é il sistema operativo dei dispositivi touch Apple. Tipicamente, le applicazioni sviluppate in questo ambiente adottano in modo pervasivo il pattern MVC (Model View Controller) come é possibile notare dalla figura 4.1. Il Model é la parte del sistema che racchiude i dati dell'applicazione, mentre il Controller é colui che si occupa in ogni istante di visualizzare sulle View ció che proviene dal Model. In iOS esistono delle entitá particolari che prendono il nome di View-

Controller, che hanno il compito di gestire le interazioni tra una View e il resto del sistema. Essi, a dispetto del nome, risultano essere a tutti gli effetti degli InputAndViewController in quanto spesso, oltre a trasferire sulle View le modifiche del Model, capita che acquisiscano da esse anche degli input in grado di cambiarlo. Quando si andrá a progettare il sistema, sebbene si utilizzeranno in tutte le situazioni dei ViewController, sará bene fare attenzione alla differenza concettuale tra un'entitá che ad una modifica del Model fa seguire un cambiamento della View e un'altra che fa esattamente l'operazione opposta.
Un altro aspetto importante di iOS consiste nel fatto che per un'applicazione viene predisposto uno stack che contiene i riferimenti ai vari ViewController. All'avvio di un'app viene fatta la push del RootViewController, ovvero del primo ViewController, che é colui che ha il compito di amministrare la View con cui l'utente si troverá inizialmente ad interagire. In seguito, si puó passare da questo ad un altro ViewController semplicemente tramite un'altra push, in modo da porre il nuovo riferimento nella prima posizione dello stack. A questo punto, come é facile immaginare, per ritornare al ViewController precedente é sufficiente un'operazione di pop. Esistono anche altri modi per fare lo switch tra ViewController, ma non li tratteremo nel dettaglio poiché esulano dagli obiettivi di questo lavoro.

4.2 L'abstraction gap tra la piattaforma iOS e Objective-C

Per la scrittura di applicazioni in ambiente iOS il linguaggio utilizzato é Objective-C [4], un'estensione a oggetti del linguaggio C, in cui sono state introdotte caratteristiche sintattiche e semantiche che derivano da Smalltalk (in particolar modo la comunicazione tramite scambio di messaggi) per includere in esso alcune delle astrazioni dell'object-oriented programming, come l'incapsulamento, l'ereditarietá, la riusabilitá. Tuttavia, Objective-C non riflette direttamente l'architettura della piattaforma iOS: per esempio, in quest'ultima esiste il concetto di ViewController, che invece é ignoto a livello del linguaggio. Questo consente di affermare che conoscere il linguaggio di programmazione non é sufficiente al fine di realizzare applicazioni in iOS; a questo

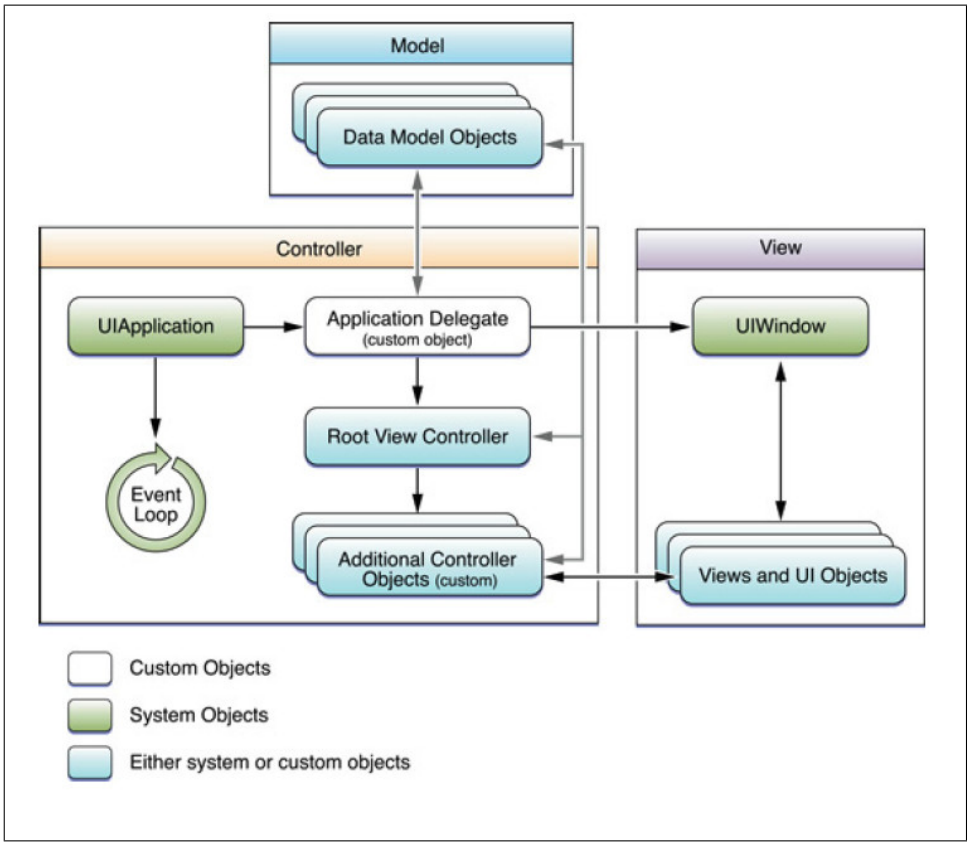

Figura 4.1: Struttura di una generica applicazione su iOS. Fonte: iOS App Programming Guide [3].

scopo occorre avere anche una nozione relativamente approfondita dell'architettura dell'ambiente, in modo tale da colmare l'abstraction gap presente. Sarebbe auspicabile per una piattaforma avere un linguaggio appositamente creato in modo da incapsulare nativamente le proprie peculiaritá. Non avendolo a disposizione, in iOS bisogna colmare il gap concettuale tramite l'utilizzo di librerie, che pertanto dovranno essere padroneggiate unitamente al linguaggio per essere in grado di sviluppare applicazioni.

4.3 Parte technology-independent e parte technology-dependent

Osservando attentamente l'architettura prodotta in precedenza, possiamo notare che la business logic di "myJourney" non dovrá subire particolari modifiche per poter entrare in ambiente iOS. Infatti, le domain entity, le factory e Formatter sono semplici oggetti che realizzano funzionalitá independenti dalla particolare piattaforma; pertanto, sará solamente necessario tradurle nel linguaggio Objective-C. Il discorso si fa invece piú complicato non appena si introduce l'interazione del cuore dell'applicazione con l'ambiente esterno. Come giá sottolineato, qualora occorra rapportarsi ad una View si avrá per forza di cose un elemento di dipendenza dalla tecnologia, ovvero il ViewController; discorso analogo nel caso il mondo esterno fosse costituito da un server, in quanto le connessioni vengono gestite tramite Framework messi a disposizione dall'ambiente di programmazione. Dal punto di vista delle factory queste considerazioni non sono rilevanti, dal momento che esse hanno solo bisogno di ricevere i dati in input a loro necessari per completare il proprio lavoro, indipendentemente dalla reale entitá che svolgerá per loro tale compito. L'unica parte del sistema che a design-time conosce con quali elementi esterni esso dovrá interagire é l'oggetto Applicazione, il quale ha il compito di coordinare tutto ció che fa parte della logica di business, creando entitá e sfruttando le loro capacitá quando necessario. Applicazione sará technology-independent per quanto riguarda l'interfaccia delle proprie operazioni, che resterá la medesima; tuttavia, l'effettiva implementazione sará

technology-dependent perché essa avrá dei compiti aggiuntivi, come quello di permettere lo switch tra un ViewController ed un altro.

4.4 La nuova versione di "myJourney"

Dopo queste riflessioni appare evidente che l'aver adottato inizialmente un approccio model-driven abbia reso particolarmente flessibile la struttura di "myJourney", a tal punto che la business logic risulta completamente technology-independent e che le varie entitá che entrano in gioco possono essere adattate con relativa facilitá in caso di un banale cambio di requisiti. Vediamo ad esempio quale aspetto assumerá il nostro sistema in base ai suoi requisiti iniziali (vedi Figura 4.2). Dal momento che all'avvio dell'applicazione si deve specificare da quale ViewController partire, bisogna crearne uno ad hoc detto AppViewController, che funzioni come un IAdapterInInput, in quanto avrá il solo compito di raccogliere l'input dell'utente e di fare eseguire ad Applicazione la rispettiva operazione. AppViewController sará dotato di una semplice GUIApp con tre pulsanti, ognuno dei quali scatenerá l'esecuzione di una funzionalitá, che ricordiamo essere la possibilitá di fare login, di creare un viaggio o di recuperarne uno giá esistente.
Per quanto riguarda l'autenticazione, Applicazione si serve di CreatoreLogin, a cui passerá un riferimento a LoginViewController, incaricato del recupero dei dati; prima di istanziare la factory, é altresí importante che Applicazione ponga il nuovo ViewController al primo posto dello stack mediante una push. Terminata la creazione della nuova entitá Login, Applicazione potrá recuperarne i dati e inviarli, al fine di valutarne la correttezza, al server, sfruttando l'azione di AdapterServerVerificaLogin; infine, mediante una pop, si ritornerá al RootViewController.
Ragionamenti analoghi si seguiranno per la creazione e il recupero di un Viaggio. Essendo due operazioni concettualmente differenti, che devono avvenire in stati diversi del sistema, i ViewController che governano rispettivamente l'input per la creazione e l'output per la visualizzazione dei dati relativi ad un Viaggio devono essere due enti distinti. Invece, nel caso in cui al termine della creazione del Viaggio si fossero dovuti mostrare i dati relativi, l'input e il successivo output

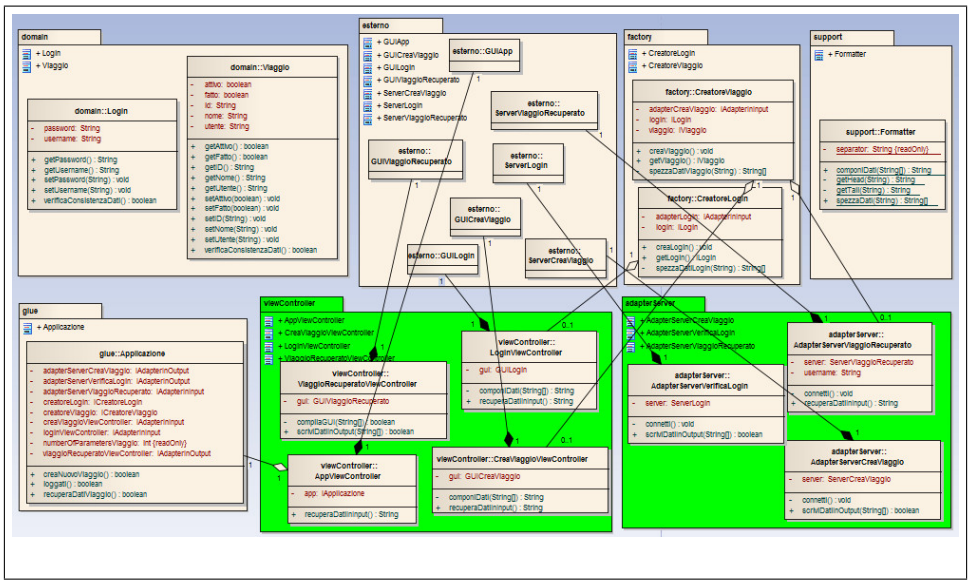

Figura 4.2: La figura mostra la struttura dei package nel progetto in iOS. Si noti come l'unica parte technology-dependent sia quella che comprende gli adapter che fanno da ponte tra la business logic e il mondo esterno, i quali sono stati suddivisi nei package "viewController" (che include le entitá che si rapportano alle GUI) e "adapterServer" (che contiene gli enti che gestiscono le connessioni in remoto). Tutti gli altri componenti sono rimasti inalterati addirittura dal modello del problema prodotto quando la piattaforma di realizzazione non era ancora stata considerata.

sarebbero potuti entrambi essere gestiti dalla medesima entitá.

Capitolo 5

Il porting in Android

Adesso possiamo vedere come mappare il progetto di "myJourney" in Android. Noteremo che parte delle considerazioni che abbiamo fatto per iOS resteranno valide anche in questo caso, mentre ve ne saranno altre totalmente nuove, che derivano direttamente dalle caratteristiche peculiari di questo ambiente di programmazione. Come per il precedente porting in iOS, per tradurre l'architettura del progetto in un'altra che possa essere poi implementata in Android é essenziale che il progettista conosca approfonditamente tale piattaforma al fine di elaborare un progetto che rimanga fedele alla business logic giá sviscerata e allo stesso tempo tenga conto delle problematiche presenti in questa particolare tecnologia.

5.1 Android

Android é un sistema operativo per dispositivi mobili diventato di proprietá della Google a partire dal 2005. Analizzando le novitá presenti in Android, salta subito all'occhio l'introduzione di 4 importanti concetti che possono essere utilizzati all'interno delle applicazioni:

- Le Activity;
- i Service;
- i Broadcast Receiver;

- i Content Provider.

Le Activity sono quei blocchi di un'applicazione che interagiscono con l'utente utilizzando lo schermo e i dispositivi di input messi a disposizione dallo smartphone. Solamente un'Activity alla volta puó essere in foreground in ogni istante; questo fa sí che in ogni momento vi sia sempre uno e un solo gestore delle interazioni con l'utente.
I Service sono una sorta di Activity in background, con la grande differenza che essi non regolano le interazioni con l'utente ma si occupano come dice il nome stesso di svolgere dei servizi utili al corretto funzionamento della particolare applicazione momentaneamente attiva o dell'intero sistema.
Un Broadcast Receiver viene utilizzato quando si intende intercettare un particolare evento, attraverso tutto il sistema. Ad esempio lo si puó usare se si desidera compiere un'azione quando si scatta una foto o quando parte la segnalazione di batteria scarica.
I Content Provider sono utilizzati per esporre dati e informazioni. Costituiscono un canale di comunicazione tra le differenti applicazioni installate nel sistema.
Vi é poi un altro supporto di particolare interesse: gli AsyncTask, ovvero una classe messa a disposizione da Android per evitare di gestire i thread manualmente. Questa risponde all'esigenza che il thread principale non va utilizzato per compiere operazioni eccessivamente onerose, ma deve restare a disposizione dell'utente in modo tale da mantenere attiva e reattiva l'interfaccia grafica. Un caso rilevante in cui si fa largo uso degli AsyncTask riguarda le connessioni in remoto, che, in questo modo, possono essere gestite parallelamente all'interazione con l'utente; in particolare, in caso di mancato arrivo della risposta grazie agli AsyncTask il problema sará arginato, mentre in assenza di essi l'intera applicazione resterebbe bloccata. In questo lavoro, di tutti questi elementi prenderemo in considerazione solo le Activity e gli AsyncTask, ma anche gli altri concetti sono molto importanti non solo per avere un'idea degli aspetti innovativi rintracciabili in Android, ma anche per evidenziare un altro problema con cui giá ci siamo scontrati parlando di iOS.

5.2 L'abstraction gap tra la piattaforma Android e Java

Activity, Service, Broadcast Receiver e Content Provider sono elementi peculiari dell'ambiente Android, che non sono presenti altrove. Per cui, come abbiamo giá asserito trattando le problematiche presenti in iOS, ci farebbe comodo poter disporre di un linguaggio di programmazione in grado di riflettere questi concetti innovativi; peccato che anche in Android non lo si possegga. Il linguaggio utilizzato dalla piattaforma Android é Java, un linguaggio di programmazione ad oggetti che nativamente non supporta le suddette entitá; dunque anche in questo caso ci troveremo di fronte ad un abstraction gap tra l'ambiente di programmazione e il relativo linguaggio. La soluzione al problema risulta essere analoga a quella vista in precedenza: sono state realizzate delle librerie con lo scopo di colmare il dislivello presente ed é stata messa a disposizione degli sviluppatori la documentazione necessaria alla loro comprensione. Tuttavia, va comunque sottolineato che questo costituisce un ulteriore ostacolo al compimento del processo di produzione del software, perché chi dovrá progettare e implementare per l'ambiente Android dovrá essere al corrente di come accedere alle librerie, oltre a conoscere in profonditá Java.

5.3 Le Activity e gli Intent

Come abbiamo visto nell'introduzione alla piattaforma Android, una delle sue caratteristiche piú innovative é la presenza delle Activity. Per certi versi esse possono essere accomunate ai ViewController di iOS, ma d'altra parte presentano anche caratteristiche nuove, meritevoli di particolare attenzione. Analogamente ai ViewController, le Activity sono le uniche entitá che possono interagire con le View e, quindi, sono quelle parti del sistema che regolano i rapporti con quella frazione di mondo esterno che é costituita dall'utente. Un'altra similitudine con i ViewController riguarda la loro gestione: per ogni applicazione o, per essere piú precisi, per ogni task, dal momento che un'applicazione potrebbe contenere al proprio interno piú di un task, viene mantenuto in memoria uno stack, all'interno del quale sono memorizzati i riferimenti

alle Activity. Per avviare un'applicazione viene lanciato un task e nello stack relativo viene inserito un riferimento alla prima Activity che entrerá in foreground. Per passare da un'Activity ad un'altra verrá fatta una push di quest'ultima nello stack, mentre per fare una pop é necessario far terminare l'Activity che in quel determinato momento é in foreground, recuperando quella in background posta immediatamente sotto di essa nello stack (vedi Figura 5.1). Fin qui niente di nuovo rispetto a ció che succede ai ViewController in iOS.

5.3.1 Intent espliciti e Intent impliciti

Tuttavia, le differenze, anche notevoli, nascono dalla diversa maniera di gestire le applicazioni nelle due piattaforme. In iOS ogni applicazione é racchiusa in una propria sandbox e non puó in alcun modo interagire con le altre, al contrario di Android, dove invece vi possono essere forme di comunicazione. In Android tutto questo é possibile grazie al meccanismo degli Intent, una specie di messaggi che il sistema invia ad un'applicazione quando si aspetta che questa faccia qualcosa; essi possono essere di due tipi: espliciti o impliciti. I primi vincolano un'Activity a richiamarne un'altra ben nota a compile-time, proprio come succede in iOS tra ViewController; l'aspetto innovativo invece sta nella seconda tipologia citata. Infatti, con gli Intent impliciti é possibile richiamare un'Activity in base a delle caratteristiche di interesse; quindi, l'associazione all'Activity concretamente utilizzata per lo scopo dichiarato dall'Intent avviene a runtime e, cosa fondamentale, potrebbe riguardare le Activity dell'intero sistema, non solo quelle dell'applicazione attualmente in uso. Tutto ció é realizzabile grazie al Manifest, un file unico per ogni applicazione, in cui tra le altre cose vengono dichiarate le Activity che ne faranno parte, con i relativi intent-filter ove andranno specificate le caratteristiche dell'Intent a cui quella determinata Activity risponde. Vedremo poi come questo meccanismo si possa mettere in pratica in "myJourney". Per ora é importante sottolineare che, grazie a questa organizzazione di Android, se nel sistema fosse presente un'Activity in grado di svolgere un certo compito, essa potrebbe essere riutilizzata ovunque ce ne fosse il bisogno, semplicemente richiamandola, senza necessitá di duplicarla o di crearne ogni volta una diversa. Questo risulta essere perfettamen-

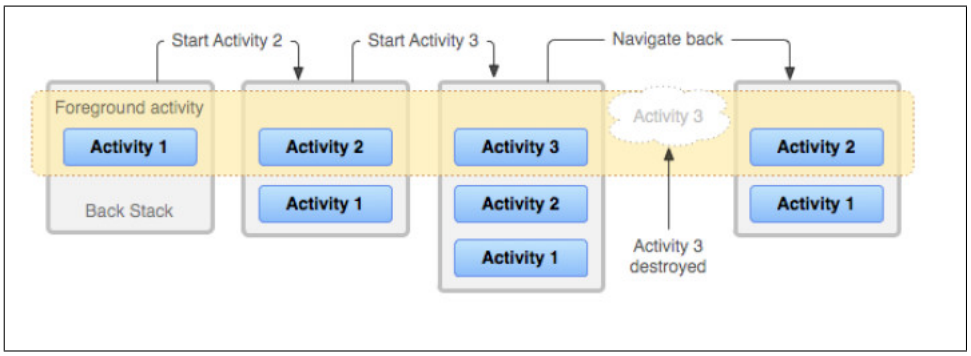

Figura 5.1: La figura mostra come viene gestito lo stack di un task in Android [5].

te coerente con i principi dell'Ingegneria del software, che ha tra i suoi obiettivi anche la possibilitá di riusare ovunque richiesto componenti software funzionanti e collaudati.

5.4 La nuova versione di "myJourney"

Come per iOS, anche per tradurre il progetto nella piattaforma Android si puó vedere la business logic rimanere completamente immutata (vedi Figura 5.2). Infatti, le domain entity, le factory e Formatter, svolgendo funzionalitá platform-independent, dovranno soltanto essere tradotte nel linguaggio Java, cosa che potrebbe essere fatta senza problema alcuno da un tool, ma che, in assenza di esso, non presenta comunque particolari difficoltá. Ancora una volta sará invece necessario introdurre le considerazioni relative alla tecnologia nel momento in cui dovremo progettare gli adapter che faranno da ponte tra l'applicazione e il mondo esterno. In base alle riflessioni relative all'ambiente Android viste in precedenza, possiamo pensare ragionevolmente che ogniqualvolta si debba interagire con una View sia istanziata un'Activity, mentre quando occorrerá dialogare con server remoti saranno utilizzati degli AsyncTask. In entrambi i casi le factory non si accorgeranno di nulla, in quanto l'unica cosa che a loro interessa é che qualcuno sia in grado di recuperare i dati che a loro servono, sia esso un AsyncTask, un'Activity o un'altra entitá ancora. Come detto nel caso iOS, occorrerá creare un'Activity ad hoc da cui partire, che

gestirá una View dotata di tre pulsanti per azionare le rispettive funzionalitá di Applicazione. Oltre a quella iniziale avremo poi altre tre Activity, con il compito rispettivamente di acquisire i dati utili al login o alla creazione di un viaggio e di mostrare le informazioni inerenti al viaggio che l'utente aveva precedentemente creato all'interno dell'applicazione.
Ricordando ció che abbiamo detto riguardo le Activity e gli Intent, si ritiene opportuno assegnare ad ognuna di esse degli Intent impliciti, in modo tale sia da rendere piú flessibile la struttura interna di "myJourney", non vincolando un'Activity ad interagire con un'altra nota a priori, ma con una qualsiasi benché in grado di svolgere le stesse funzioni, sia da permettere il riuso di questi componenti anche da altre applicazioni. Nel Manifest relativo a "myJourney" avremo la dichiarazione di tutte le Activity che ne fanno parte, che specificheranno nei loro intent-filter una Action, che spiega in generale di cosa si occupa l'Intent, e una Category, che aggiunge un pezzo di informazione per precisare maggiormente il suo compito. Pertanto, possiamo pensare che ActivityLogin abbia una ACTION=LOGIN e una CATEGORY=DEFAULT, mentre per ActivityCreaViaggio e ActivityViaggioRecuperato, che potrebbero entrambe avere una ACTION=VIAGGIO, la distinzione della Category diventa fondamentale per separare i loro obiettivi: la prima avrá dunque CATEGORY=CREA, mentre la seconda CATEGORY=RECUPERA.

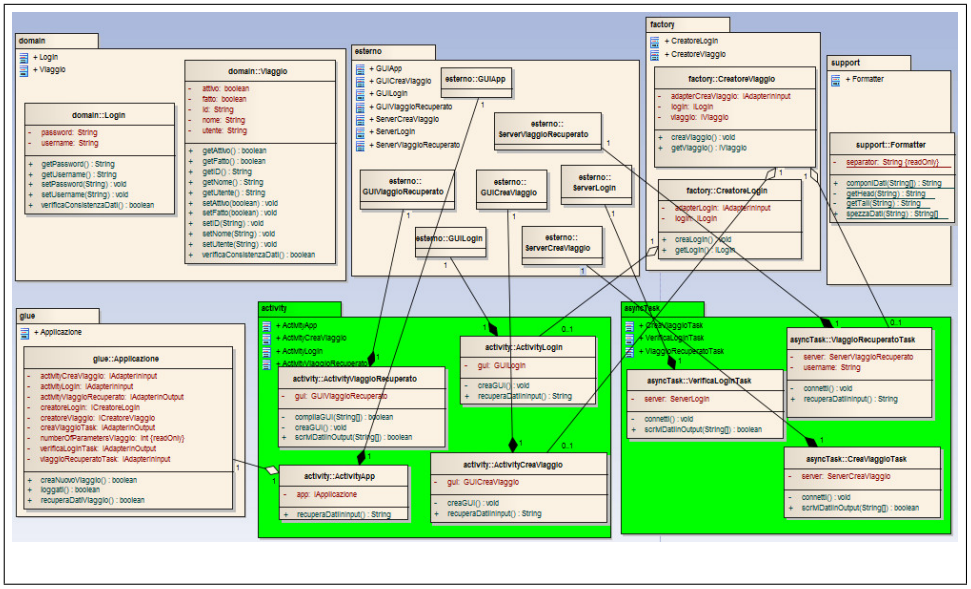

Figura 5.2: La figura mostra la struttura dei package nel progetto in Android. Si noti come le uniche parti technology-dependent siano il package “activity” e quello “asyncTask”, mentre tutte le altre siano rimaste inalterate addirittura dal modello del problema prodotto quando la piattaforma di realizzazione non era ancora stata considerata. Si noti altresí come la tecnologia non incida direttamente sulla logica di business dell’applicazione, bensí su quelle entitá che permettono i collegamenti del sistema con il mondo esterno.

Capitolo 6

Conclusioni

In conclusione abbiamo visto che l'approccio code-based seguito durante la produzione del primo artefatto ha causato non pochi problemi: l'assenza del Model e del resto della business logic ha reso l'applicazione technology-dependent e priva di flessibilitá. Un semplice cambio dei requisiti, l'attivitá di debug o la necessitá di eseguire il porting su un'altra piattaforma comportano uno sforzo enorme per riorganizzare l'intero sistema. Adottando invece una strategia model-driven é possibile essere piú flessibili di fronte a queste problematiche: utilizzando la modellazione in termini di diagrammi UML, all'interno del sistema si possono evidenziare piú parti, aventi tra loro compiti differenti. In questo modo, i vari componenti possono essere sviluppati e testati separatamente, abbattendo i costi di produzione; lo stesso cambio dei requisiti, se non comporta stravolgimenti, potrebbe anche avere come risultato solo una diversa composizione delle medesime entitá.
Tuttavia, anche l'approccio model-driven non é esente da problemi; infatti, tipicamente, in un team di sviluppo aziendale analista, progettista e implementatore non sono la stessa persona e questo non garantisce che il codice rispecchi fedelmente il modello del problema e il progetto che da esso scaturisce. Per evitare questi ostacoli sarebbe auspicabile disporre di tool in grado di produrre il codice direttamente dai modelli ed eseguire eventuali update a livello di questi ultimi, in modo da evitare che qualcuno possa stravolgere le implementazioni senza che nelle precedenti fasi di analisi e progettazione ne rimanga traccia (nell'ottica di quella che tipicamente viene definita la "crisi del

software").
Un altro aspetto degno di nota é relativo alle piattaforme iOS e Android: per sviluppare applicazioni si é costretti ad utilizzare come linguaggi di programmazione rispettivamente Objective-C e Java, che non riflettono nativamente le caratteristiche dei relativi ambienti. Pertanto, come abbiamo visto affrontando il problema del porting, per progettare completamente l'applicazione con riferimento alla tecnologia, al progettista non é sufficiente conoscere il linguaggio di programmazione, ma gli occorre sapere anche come la particolare piattaforma é organizzata al proprio interno, in modo da colmare gli abstraction gap che gli si presentano dinanzi. Questo caso rende desiderabile la presenza di un linguaggio ad hoc, capace di rispecchiare fedelmente le peculiaritá dello specifico ambiente di programmazione.

Bibliografia

[1] Figura vendite dispositivi mobile:
www.gadgetvenue.com/wp-content/uploads/2011/04/smartphone-numbers.png

[2] Figura Requisiti non funzionali:
http://www.uniroma2.it/didattica/ISW1/deposito/03-Requisiti_software.pdf

[3] Figura struttura app iOS:
http://developer.apple.com/library/ios/#/iphone/conceptual/iphoneosprogrammingguide/AppArchitecture/AppArchitecture.html#//apple_ref/doc/uid/TP40007072-CH3-SW2

[4] Objective-C:
http://it.wikipedia.org/wiki/Objective_C

[5] Figura gestione stack Android:
http://android.litrin.net/guide/components/tasks-and-back-stack.html

Elenco delle figure