UNIVERSITÀ DEGLI STUDI DI PADOVA

FACOLTÀ DI INGEGNERIA

CORSO DI LAUREA MAGISTRALE
IN INGEGNERIA INFORMATICA

TESI DI LAUREA

RACCOLTA E ANALISI DEI DATI ASSISTITE DAL CALCOLATORE PER I PAZIENTI DI UNA SALA POST-OPERATORIA

RELATORE: PROF. CARLO FANTOZZI
CORRELATORE: DOTT. ALBERTO GRASSETTO

LAUREANDO: AURELIEN F. BANTIO LONTSI

ANNO ACCADEMICO 2010/2011

Sommario

L'oggetto di questa tesi riguarda la raccolta e l'analisi dei dati assistite dal calcolatore per i pazienti della sala post-operatoria (più conosciuta sotto il nome inglese *Recovery Room*) dell'Ospedale dell'Angelo di Mestre.
Verrà presentata la progettazione dell'applicazione mobile che è stata sviluppata a tale scopo, utilizzando il sistema operativo Android. Successivamente si discuterà l'analisi effettuata, utilizzando delle tecniche di classificazione, sui dati dei pazienti, utile per rilevare l'eventuale possibilità di predire l'andamento del loro stato durante il ricovero.

The object of this thesis is a computer-assisted collection and analysis of patient data in the recovery room of the Hospital of the Angel of Mestre.
The mobile application that was developed for this purpose, using the Android operating system, will be presented. Later the analysis of the patient data will be discussed. Classification techniques will be utilized since they are useful to detect whether it is feasible to predict the outcome of patient's state during their hospitalization in the recovery room.

Indice

Capitolo 1

Introduzione Generale

La parte iniziale del lavoro si è concentrata sulla realizzazione di un'applicazione funzionante su un dispositivo mobile. Questa applicazione permette di raccogliere in modo operativo e veloce i dati dei pazienti ricoverati nella *Recovery Room*.

La seconda parte si è concentrata sul trasferimento dei dati raccolti dal dispositivo mobile verso una macchina server una volta che il paziente viene dimesso dalla sala. Dai dati trasferiti sul server, viene generato (creazione *on the fly*) un file in formato pdf contenente le informazioni raccolte su ogni paziente durante il suo ricovero nella sala. Una copia di questo file viene mandata automaticamente in stampa per poi essere inclusa nella cartella clinica del paziente e un'altra viene archiviata. Sempre dai dati trasferiti, viene aggiornata un file cumulativo in formato *comma-separated values* (csv)[1] che raccoglie i dati di tutti i pazienti che passano per la Recovery Room. Questo file potrà essere comodamente consultato dai medici usando un adeguato programma per i fogli elettronici.

La terza parte si è concentrata su un'analisi dei dati raccolti usando degli algoritmi di *data mining* per studiare l'eventuale possibilità di predire l'andamento dello stato dei pazienti a partire dalle informazioni su di loro raccolte all'arrivo in sala.

L'esposizione del lavoro è articolato come segue.

[1] Formato di file basato su file di testo utilizzato per l'impostazione ed esportazione (ad esempio da fogli elettronici o database) di una tabella di dati.

- Nel capitolo 2 viene data una descrizione della Recovery Room e una breve descrizione dei parametri del paziente che vengono monitorati presso l'Ospedale dell'Angelo di Mestre.
- Nel capitolo 3 vengono presentati gli obiettivi del lavoro e le scelte effettuate per raggiungerli.
- Nel capitolo 4 viene presentato brevemente il sistema operativo *Android* che è stato scelto per sviluppare l'applicazione mobile.
- Nel capitolo 5 viene presentata la fase di progettazione del sistema.
- Nel capitolo 6 viene proposta la realizzazione vera e propria dell'applicazione, analizzando anche alcune parti del codice del programma.
- Nel capitolo 7 viene presentata la parte di lavoro relativa alla gestione dei dati nel lato server e alla stampa della scheda del paziente.
- Nel capitolo 8 vengono proposte le tecniche usate per l'analisi e successivamente vengono presentati i risultati ottenuti.
- Nel capitolo 9 viene dato spazio ad alcune riflessioni sul lavoro complessivo ed ai possibili sviluppi futuri.

Capitolo 2

Recovery Room

In questo capitolo verranno illustrati la *Recovery Room* e i parametri del paziente che vengono monitorati in questo ambiente. In particolare verrà presentata la gestione della *Recovery Room* dell'ospedale dell'Angelo di Mestre.

2.1 Origine della Recovery Room

Negli ultimi 50 anni lo sviluppo delle tecniche chirurgiche e di anestesia e rianimazione ha permesso un importante sviluppo della chirurgia ed un ampiamento delle sue indicazioni; contemporaneamente è risultata molto più impegnativa l'assistenza postoperatoria. Il periodo che segue l'intervento è gravato da un elevato numero di incidenti e complicanze. Questo ha portato all' organizzazione di una Sala di Ricovero Postoperatorio (SRP), o Recovery Room (RR), che rappresenta un'area che consente il graduale e sicuro recupero completo delle funzioni vitali nell'immediato postoperatorio. La SRP è caratterizzata da un'ampia tecnologia che permette il monitoraggio cardiocircolatorio e respiratorio, l'assistenza ventilatoria, il controllo termico del paziente, il recupero postoperatorio del sangue, il controllo metabolico e infusionale e l'impostazione della terapia antalgica per il dolore acuto postoperatorio. La RR rappresenta anche un elemento determinante per il percorso intraospedaliero di molti pazienti nel postoperatorio. Nella pratica quotidiana, infatti, quando il risveglio del paziente è completato e

le funzioni vitali sono stabili (secondo uno schema specifico di scores o recovery scores), il paziente viene trasferito nel reparto di degenza. Nel caso in cui permangano problematiche relative alle funzioni vitali, la soluzione più sicura è trasferire il paziente in terapia intensiva.
Tutti i pazienti sottoposti ad intervento chirurgico transitano attraverso la RR ad eccezione di quelli trattati in regime di day-surgery e di quelli destinati alla terapia intensiva in modo programmato.

2.2 Fasi cliniche e parametri monitorati

Quando il paziente giunge in RR l'anestesista e/o l'infermiere di sala forniscono all'infermiere che lo prende in carico tutte le informazioni utili alla corretta gestione dell'immediato periodo postoperatorio. La sorveglianza postoperatoria comprende la periodica valutazione dello stato di coscienza, delle funzioni respiratoria, cardiocircolatoria e neuromuscolare, della temperatura, del dolore, della diuresi, dei drenaggi chirurgici oltre al trattamento di eventuali complicanze (nausea e vomito, brivido, aritmie, emorragia, ecc.). I parametri monitorati devono essere trascritti in cartella.

2.2.1 Funzioni e parametri monitorati

Vengono di seguito precisate le funzioni e i parametri monitorati nell'immediato periodo postanestesiologico:

- *Stato di coscienza (sedazione, agitazione e delirio)*: il controllo dello stato di coscienza fa parte, insieme alla verifica dei parametri cardiovascolari e respiratori, della valutazione postoperatoria iniziale all'arrivo in RR. L'insorgenza di uno stato confusionale acuto postoperatorio è una condizione clinica associata a un aumento della mortalità, a complicanze postoperatorie e comporta inoltre l'aumento della durata della degenza ospedaliera.
  Lo stato di coscienza e i riflessi protettivi devono essere valutati clinicamente con periodicità non superiore ai 15 minuti.

- *Funzione respiratoria*: l'ipossiemia è uno degli eventi più temuti nel periodo postoperatorio; essa è probabilmente il meccanismo più co-

munemente responsabile del verificarsi di outcome avversi nel periodo postoperatorio. Il monitoraggio pulsiossimetrico in RR consente una diagnosi e quindi un trattamento precoce dell'ipossiemia e delle conseguenze negative ad essa correlate; tuttavia non c'è evidenza scientifica sufficiente a dimostrare che l'uso del pulsiossimetro in sala di risveglio migliori l'outcome dei pazienti in termini di complicanze cardiorespiratorie, infettive e neurologiche[1].
Durante la fase di risveglio devono essere attentamente valutati la pervietà delle vie aeree, il pattern respiratorio (frequenza respiratoria ed escursione toracica) e la concentrazione o saturazione di ossigeno nel sangue (SpO2) con pulsiossimetro. La pronta disponibilità in RR di un sistema di monitoraggio capnografico consente di individuare precocemente gli episodi di ipopnea-apnea e non è inficiato dall'eventuale somministrazione di ossigeno.

- *Funzione cardiocircolatoria*: la maggior parte degli eventi cardiovascolari avversi si verifica entro le due ore successive all'intervento chirurgico; questi interessano circa il 7% dei pazienti e sono rappresentati prevalentemente da fenomeni aritmici (tachicardia, bradicardia) ed emodinamici (ipertensione, ipotensione). Tachicardia e ipertensione correlano con un incremento del rischio di ricovero in terapia intensiva e di mortalità postoperatoria.
  Durante il ricovero in RR, si procede al controllo della frequenza cardiaca (FC), della pressione arteriosa (PA) e del tracciato elettrocardiografico al fine di individuare e quindi trattare precocemente eventuali alterazioni cardiocircolatorie.

- *Funzione neuromuscolare*: il persistere di un blocco neuromuscolare residuo è un fenomeno ancora oggi frequente nel periodo postoperatorio. Sono ben note le gravi conseguenze che possono derivarne: riduzione della capacità di deglutire e quindi di proteggere le vie aeree dall'aspirazione, una maggiore incidenza di episodi ostruttivi a carico delle prime vie aeree, una riduzione della risposta ventilatoria all'ipossiemia, causata dall'effetto inibitorio diretto esercitato dal blocco neuromuscolare residuo sull'attività chemorecettoriale del glomo

carotideo, un aumento della morbosità e della mortalità per complicanze respiratorie. Il controllo clinico della funzione neuromuscolare durante il recupero dell'anestesia risulta pertanto di primaria importanza.
Il monitoraggio clinico del grado di risoluzione del blocco neuromuscolare deve essere effettuato in tutti i pazienti trattati con bloccanti neuro-muscolari di tipo non depolarizzante al momento dell'ingresso in RR e prima della dimissione.

- *Diuresi e bilancio idroelettrolitico*: lo squilibrio idroelettrolitico si riscontra nel 20% dei pazienti ed è stato riconosciuto come un importante fattore correlato alla mortalità e morbilità postoperatoria. Il corretto approccio al paziente dovrebbe essere dunque personalizzato in termini di volume, velocità d'infusione e composizione elettrolitica, evitando sia carichi eccessivi che la deplezione idrica.
  Lo stato volemico, l'equilibrio elettrolitico e la diuresi devono essere valutati attentamente durante la degenza in RR in pazienti selezionati sulla base delle patologie croniche associate e del tipo e durata dell'intervento chirurgico.

- *Temperatura (T°)*: l'ipotermia accidentale lieve, intesa come una riduzione della temperatura centrale Tc < 36°C, interessa circa il 50% dei pazienti nell'immediato periodo postoperatorio. In relazione al grado di ipotermia raggiunta possono essere necessarie anche 2-5 ore per un completo ripristino della normotermia. Si ritiene dunque utile, nei soggetti a rischio di ipotermia, il monitoraggio della temperatura corporea.
  In pazienti selezionati sulla base dell'età (in particolare le fasce estreme), della costituzione fisica, del tipo e durata di intervento, delle perdite ematiche e fluide e delle comorbidità (endocrinopatie in particolare), si deve misurare la temperatura corporea all'ingresso e prima della dimissione del paziente dall'RR.

- *Brivido*: la principale causa dell'insorgenza di brivido postoperatorio è rappresentata dalla risposta termoregolatrice all'ipotermia postope-

ratoria; alla sua genesi, tuttavia, concorrono anche altri fattori quali: il cattivo controllo del dolore, la liberazione di citochine pirogeniche in seguito al danno tissutale, l'alcalosi respiratoria, la riduzione del tono ortosimpatico, la vasoplegia in corso di anestesia spinale o epidurale. Dal brivido derivano: notevole disagio per il paziente, interferenze con il monitoraggio della pressione arteriosa e dell'ECG, aumento della pressione endooculare, tachipnea, aumento del consumo di O2 e della produzione di CO2, aumento del lavoro cardiaco, riduzione della gittata cardiaca, aumento delle richieste metaboliche, acidosi lattica, aumento della pressione intracranica.
Il mantenimento della normotermia rappresenta l'intervento cardine per la prevenzione ed il trattamento del brivido postoperatorio.

- *Dolore*: lo scarso controllo del dolore postoperatorio comporta gravi conseguenze di ordine psicologico e fisiologico: aumento della morbilità, aumentati tempi di degenza fino all'insorgenza di dolore cronico e peggioramento della qualità di vita. Il dolore deve essere mantenuto entro il livello target di Visual Analogic Scale (VAS) < 3 e la frequenza delle rilevazioni deve essere rapportata al tipo di analgesia adottato.
  Il dolore deve essere monitorato con scale a punteggio con frequenza minima del rilievo ogni 15 minuti in tutti i pazienti ricoverati in RR sottoposti ad intervento chirurgico o a manovre diagnostico-terapeutiche invasive.

- *Nausea e vomiti postoperatori*: la nausea e il vomito postoperatorio (PONV - Post Operative Nausea and Vomiting) sono tuttora una delle complicanze più frequenti e più temute dai pazienti. I PONV infatti, prolungano il tempo di permanenza del paziente in RR, sono causa di discomfort e incidono negativamente sulla qualità delle cure. La letteratura riporta un'incidenza di PONV che varia dal 20 al 30% dei pazienti sottoposti ad anestesia generale e arriva fino al 70% in quelli ad alto rischio. Sono stati proposti diversi punteggi predittivi per determinare quali sono i fattori di rischio del PONV e per identificare i pazienti maggiormente esposti. All'interno dell'RR è necessaria una

valutazione routinaria della comparsa di nausea e/o vomito postoperatorio.
All'interno dell'RR è necessaria una valutazione routinaria della comparsa di nausea e/o vomito postoperatorio.

- *Drenaggi chirurgici*: in lettereatura non sono riportati casi sufficienti per valutare l'impatto del monitoraggio dei drenaggi chirurgici nell'immediato periodo postoperatorio al fine di quantificare eventuali sanguinamenti (i drenaggi servono per monitorare il sanguinamento o le raccolte purulente). Alcune fonti tuttavia affermano che la valutazione dei drenaggi permette la rapida intercettazione dell'insorgere di complicanze e riduce gli outcome avversi.
  Si riconosce l'utilità di monitorare l'entità dei sanguinamenti e lo stato dei drenaggi eventualmente presenti in tutti i pazienti che accedono all'RR.

2.2.2 Criteri di dimissione

Ogni RR deve avere criteri ben definiti per il trasferimento del paziente in reparto. La responsabilità del trasferimento del paziente dalla RR al reparto di degenza è dell'anestesista.
Si ritiene che il paziente possa essere trasferito al reparto di degenza quando sono soddisfatti i seguenti criteri:

- adeguato stato di coscienza (deve essere cosciente senza stimolazione eccessiva);
- capacità di mantenere la pervietà delle vie aeree in modo autonomo e presenza dei riflessi protettivi (deglutizione e tosse) delle vie aeree;
- ventilazione e ossigenazione soddisfacenti (cioè frequenza respiratoria nella norma SpO2 accettabile);
- stabilità emodinamica: dev'essere stata esclusa la presenza di sanguinamento persistente, i valori di polso e pressione arteriosa devono essere vicini ai normali valori preoperatori e la perfusione periferica de-

v'essere adeguata (cute calda e rosea, tempo di riempimento capillare < 3 secondi);

- buon controllo del dolore;
- assenza di nausea e vomito;
- normotermia;
- completo recupero dell'attività motoria e della forza muscolare.

Nella valutazione del paziente sono usate scale a punteggi che:

- rendono obiettivi e confrontabili i rilievi;
- permettono di seguire il recupero delle funzioni vitali nel tempo;
- documentano la raggiunta stabilizzazione.

Un esempio di scala a punteggio è riportato in figura 2.1. La dimissibilità del paziente deve essere documentata in cartella clinica.

Per maggiore dettagli su questa parte si consiglia la consultazione del documento [2], che è stato elaborato da un Gruppo di Lavoro (GdL) costituito da un nucleo di membri del Gruppo di Studio della Società Italiana di Anestesia Analgesia Rianimazione e Terapia Intensiva (SIAARTI) per la Sicurezza in Anestesia e Terapia Intensiva con spiccato interesse per le tematiche inerenti l'argomento in oggetto, affiancato da riconosciuti esperti nel settore.

2.3 Gestione dei pazienti nell'RR dell'Ospedale dell'Angelo di Mestre

L'ospedale dell'Angelo di Mestre ha la particolarità di avere 16 sale operatorie, tutte raggruppate in un stesso blocco operatorio. Possiede una *Recovery Room* che conta 8 posti e serve tutte le 16 sale operatorie (10 circa funzionanti contemporaneamente).

Attività	Capace di muovere 4 estremità volontariamente o a comando	2
	Capace di muovere 2 estremità volontariamente o a comando	1
	Incapace di muovere le estremità volontariamente o a comando	0
Respirazione	Capace di respirare a fondo e di tossire liberamente	2
	Dispnea o respirazione limitata	1
	Apnea	0
Circolazione	PA ± 20% dei valori preoperatori	2
	PA ± 20 - 49% dei valori preoperatori	1
	PA ± 50% dei valori preoperatori	0
Coscienza	Completamente sveglio	2
	Risvegliabile alla chiamata	1
	Non risvegliabile	0
Saturazione d'ossigeno	In grado di mantenere una SpO_2 > 92% in aria ambiente	2
	Necessario ossigeno per mantenere SpO_2 > 90%	1
	SpO_2 <90% anche con supplemento d'ossigeno	0
	Punteggio totale	
Il paziente può essere trasferito in reparto di degenza quando ottiene un punteggio totale minimo di 8 in due valutazioni successive, in assenza di punteggio uguale a zero per le singole voci.		

Figura 2.1: Scala a punteggi di Aldrete

I pazienti, dopo l'intervento chirurgico, permangono nella RR da un minimo di 30 minuti ad un massimo di 3 ore. Vi è un infermiere dedicato, coadiuvato da un altro infermiere al bisogno.
L'infermiere deve rilevare i parametri del paziente e chiedere l'intervento del medico in caso di necessità. Il medico è generalmente l'anestesista che lo ha operato oppure l'anestesista dedicato alle urgenze oppure ancora l'anestesista responsabile del blocco operatorio.
Un anestesista firma sempre la dimissione del paziente.
Quasi tutti i pazienti sottoposti ad intervento chirurgico trascorrono il periodo post-operatorio nella sala risveglio ad eccezione di quelli che subiscono interventi di chirurgia minore in anestesia locale e blanda sedazione; tuttavia la decisione di una *fast track recovery* è sempre a discrezione dell'anestesista che segue il paziente.
Ogni giorno accedono alla RR tra i 10 e i 20 pazienti.
Alla dimissione la scheda con i dati registrati viene allegata alla cartella del paziente. Non esiste un archivio con i dati di tutti i pazienti transitati in

RR.
La scheda utilizzata per la trascrizione dei parametri dei pazienti nella recovery room dell'ospedale dell'Angelo di Mestre è cartacea ed è riportata nelle figure 2.2 e 2.3. In questa scheda inoltre, sono presenti i parametri "ASA"e "Bromage Scale":

- ✓ **ASA (o Punteggio di Stato Fisico)**
  È uno score ideato dalla società americana degli anestesisti ("American Society of Anesthesiologists", ASA) e viene utilizzato per valutare lo stato di salute pre-operatorio di un paziente.
  Lo score ASA permette di valutare il rischio anestesiologico e di ottenere un parametro predittivo di mortalità e morbosità peri-operatoria. Il suo utilizzo offre inoltre la possibilità di studiare e determinare i fattori che influiscono nell'infezione post-operatoria (IPO) e i principi di prevenzione.
  I pazienti sono classificati con una scala che va da 1 a 5. Uno score maggiore o uguale a 3 è considerato come un fattore di rischio anestesiologico e anche di IPO. La scala è esplicitata nella tabella 2.1.

ASA score	**Stato di salute del paziente**
1	Paziente sano, in buona salute. Pazienti esenti da malattie organiche o psichiche.
2	Malattia sistemica leggera; paziente che presenta una patologia in stadio iniziale per esempio: lieve ipertensione arteriosa, lieve anemia, bronchite cronica lieve.
3	Malattia sistemica severa o invalidante, paziente che presenta un patologia di grado severo per esempio: angina moderata, diabete, ipertensione grave, scompenso cardiaco debuttante.
4	Paziente che presenta una patologia di grado severo tale da inficiare la prognosi quoad vitam, per esempio: angina a riposo, insufficienza sistemica pronunciata (polmonare, renale, epatica, cardiaca...).
5	Paziente moribondo la cui speranza di vita con o senza intervento chirurgico non supera le 24 ore.

Tabella 2.1: Classificazione dello stato del paziente secondo lo score ASA.

- ✓ **Bromage Scale**
  È uno score usato per valutare il blocco motorio di un paziente. Per

RECOVERY ROOM – Ospedale dell'Angelo – Dipartimento di Anestesia e Rianimazione

Paziente _____ Data di nascita _____ Reparto _____
Intervento _____ Anestesista _____ Anestesia _____
ASA _____ Allergie _____ Note _____

Ora di arrivo_____ **SpO2**_____ **PA**_____ **FC**____ T°_____
Infusioni in atto _____
Monitoraggio: SpO2 – EGA – NIBP – ABP – FC – PVC – T° - Diuresi
Bromage Scale arrivo: 0 1 2 3
Provvedimenti terapeutici _____
_____
_____
_____

Drenaggi	***arrivo***	***60'***	***120'***	***180'***
Diuresi				

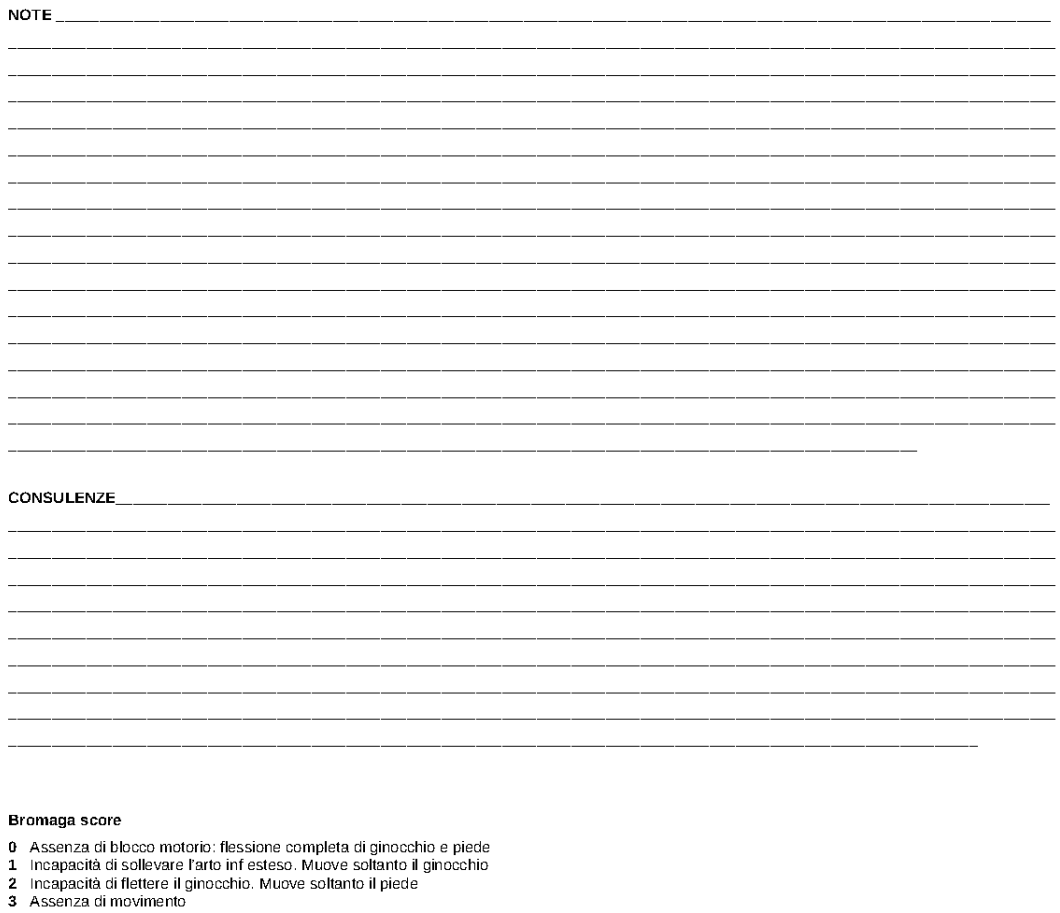
RECOVERY SCORE		Arrivo	15'	30'	60'	90'	120'	180'	Dimissione
Livello di Coscienza									
Sveglio e orientato	2								
Risvegliabile su chiamata	1								
Responsivo a stimolazione	0								
Respiratorio									
Eupnoico e in grado di tossire validamente	5								
Bradi o tachipnoico e/o respiro superficiale e/o incapace di tossire validamente	2								
Dispnoico e/o SpO2 < 90% in AA	0								
Emodinamica									
PAS ± 20% rispetto al preoperatorio e a ritmo sinusale (o comunque invariato rispetto il preoperatorio)	5								
PAS ± 20-50% rispetto al preoperatorio e/o tachicardico (FC > 100bpm) e/o bradicardico (FC < 50bpm)	2								
PAS ± 50% rispetto al preoperatorio e/o aritmico (aritmia non presente nel preoperatorio) e/o necessità di supporto	0								
Dolore									
Assente (VAS ≤ 3)	2								
Moderato (VAS 4-6)	1								
Severo (VAS ≥ 7)	0								
PONV									
Assente	2								
Nausea	1								
Vomito	0								
Brivido									
Assente	2								
Moderato	1								
Intenso	0								
TOTALE									

BROMAGE SCALE dimissione: 0 1 2 3

Ora di dimissione: _____ **SpO2** ____ **PA**_____ **FC** ____

Responsabile dimissione Dr./Dr.ssa _____ **Rilevamento parametri I.P.** _____

Figura 2.2: Pagina 1 - scheda utilizzata nell'RR dell'Ospedale dell'Angelo di Mestre


NOTE

CONSULENZE

Bromaga score

0 Assenza di blocco motorio: flessione completa di ginocchio e piede
1 Incapacità di sollevare l'arto inf esteso. Muove soltanto il ginocchio
2 Incapacità di flettere il ginocchio. Muove soltanto il piede
3 Assenza di movimento

Figura 2.3: Pagina 2 - scheda utilizzata nell'RR dell'Ospedale dell'Angelo di Mestre

questo score l'intensità del blocco motorio viene quantificata valutando la capacità del paziente di muovere gli arti inferiori.
I pazienti sono classificati con una scala che va da 0 a 3. Tale scala è esplicitata nella tabella 2.2.

Bromage score	**Blocco motorio del paziente**
0	Assenza di blocco motorio: flessione completa di ginocchio e piede.
1	Incapacità di sollevare l'arto inferiore esteso. Muove soltanto il ginocchio.
2	Incapacità di flettere il ginocchio. Muove soltanto il piede.
3	Assenza di movimento.

Tabella 2.2: Scala di valutazione del Bromage Score.

Capitolo 3

Obiettivi del lavoro e scelte effettuate per raggiungerli

In questo capitolo verranno illustrati gli obiettivi del lavoro; successivamente verranno presentate le scelte che sono state effettuate per raggiungerli.

3.1 Obiettivi del lavoro

Lo scopo principale di questo lavoro di tesi è stato quello di:

- realizzare un'applicazione funzionante su un dispositivo mobile per la raccolta delle informazioni dei pazienti della Recovery Room, informatizzando così il processo di raccolta dati che è ancora gestito in modo cartaceo (paragrafo 3.1.1);

- gestire il trasferimento dei dati delle persone ricoverate verso un server, in modo da creare un archivio delle informazioni sui pazienti che sono stati ospitati nella Recovery Room (paragrafo 3.1.2);

- partendo dai primi dati in archivio, costruire dei predittori capaci di prevedere l'andamento dei pazienti in RR, dopo aver preso nota dei loro parametri all'ingresso in sala (paragrafo 3.1.3).

3.1.1 Applicazione per la raccolta dei dati

Finora i parametri dei pazienti vengono raccolti dagli infermieri usando una scheda cartacea strutturata in modo da permetter loro di rilevare i parametri fissi (età, tipo anestesia, tipo intervento, ecc.), i parametri relativi al monitoraggio di arrivo e di dimissione dei pazienti e i loro valori degli score che sono monitorati periodicamente durante il loro soggiorno nella Recovery Room. L'applicazione per lo stesso scopo deve essere di facile utilizzo, intuitiva e non rallentare la fase di raccolta dei dati. Gli obiettivi di questa parte sono centrati sulla creazione dei formulari usando dei layout (viste) ergonomiche in modo da facilitare al meglio la compilazione e quindi l'inserimento dei dati dei pazienti in database.
Considerando il contesto lavorativo in cui l'infermiere deve passare da paziente a paziente per registrare i parametri da monitorare, la scelta si è portata chiaramente su un'applicazione per dispositivi mobili invece di un'applicazione per i personal computer.

3.1.2 Trasferimento dei dati e la loro gestione lato server

Gli obiettivi di questa parte sono stati i seguenti:

- tenere una traccia delle informazioni sui pazienti che hanno soggiornato nella Recovery Room per un periodo di tempo. Precedentemente non si teneva nessuna informazione del paziente all'uscita della Recovery Room. I parametri oggetti del monitoraggio erano inscritti su una scheda cartacea che veniva inserita nella cartella clinica che accompagnava il paziente all'uscita della Recovery Room. Si è reso opportuno trovare un modo per tenere traccia dell'andamento dello stato dei pazienti passati per la Recovery Room. Ai medici servivano due modi di consultazione dei dati dei pazienti: uno per poter consultare i dati di un singolo paziente e uno che gli permettesse di avere a disposizione i dati di tutti i pazienti usando ad esempio un foglio di calcolo elettronico.

- Creazione di un file strutturato con tutte le informazioni del paziente: ora i dati dei pazienti non devono più essere inseriti direttamente su

una scheda cartacea ma è sempre necessario avere una scheda riassuntiva dell'evoluzione dello stato del paziente durante il suo soggiorno in RR; tale scheda deve essere poi inserita all'interno della cartella clinica del paziente che l'accompagna all'uscita della Recovery Room.

3.1.3 Costruzione dei predittori

Gli obiettivi di questa parte di lavoro erano quelli di costruire dei predittori richiesti dai medici che volevano poter prevedere l'andamento degli scores dei pazienti all'ingresso in sala. Una volta entrato in sala, a un paziente vengono rilevati i parametri fissi (età, tipo intervento, tipo anestesia, ecc.) e i parametri relativi al monitoraggio di arrivo (pressione arteriosa, temperatura, bromage scale, ecc.) ed in base a questi parametri si cerca di prevedere l'andamento degli scores del paziente durante il suo soggiorno in Recovery Room.
Questo è stato suddiviso in due parti: una prima parte di studio di fattibilità dei predittori e una seconda parte di costruzione vera e propria dei predittori.

3.2 Scelte effettuate per raggiungere gli obiettivi del lavoro

3.2.1 Scelta del sistema operativo

All'inizio della fase di progettazione sono state valutate le due piattaforme al momento più diffuse nel mondo dei dispositivi mobili: iOS di Apple e Android di Google. La prima differenza tra i due sistemi è il fatto che iOS è un sistema operativo proprietario, protetto e utilizzabile solo su prodotti Apple come iPhone, iPod touch e iPad; Android invece è un sistema rilasciato sotto licenze open-source [3]: Apache 2.0 [4] e GPLv2 [5]. È quindi disponibile in rete anche il codice sorgente del sistema operativo[1], il quale può essere modificato e redistribuito a piacere; motivo per cui esistono tanti

[1] http://source.android.com/source/downloading.html

terminali mobili Android-based in circolazione. L'unico dispositivo tablet basato su iOS disponibile al momento dell'inizio del lavoro era iPad. Per quanto riguarda i dispositivi tablet basati su Android, c'erano già moltissimi terminali con una grande varietà di prezzi anche se non provenivano da costruttori noti del mercato dell'hardware. In più c'erano dei tablet annunciati dai più noti costruttori come Samsung, Dell, Archos, ecc.
I due sistemi operativi studiati avevano entrambi tutti i requisiti necessari per la realizzazione dell'applicazione, nel senso che giravano su dispositivi touch screen, si potevano realizzare delle applicazioni con interfacce utenti ergonomiche e di buona qualità, supportavano entrambi il collegamento wireless e il protocollo FTP per il trasferimento dei dati. Le tre caratteristiche che hanno fatto la differenza tra i due sistemi sono le seguenti:

- *le dimensione dell'iPad*: l'iPad è a tutt'ora l'unico tablet su cui gira iOS. Purtroppo le sue dimensioni (schermo da 9,7 pollici) non sono molto adatte al contesto lavorativo di riferimento. Tenerlo con una mano e lavorarci in piedi (come fanno gli infermieri) è difficile;
- *la varietà di scelta dei dispositivi basati su Android*: tralasciando anche la grande varietà dei prezzi, i dispositivi tablet disponibili in quel momento e quelli annunciati erano svariate e con dimensioni di schermi diversi;
- *strumenti di sviluppo*: mentre gli strumenti di sviluppo delle applicazioni sotto Android sono disponibili gratuitamente, quelli dell'iOS sono protetti da una licenza a pagamento.

Queste differenze hanno portato a scegliere Android come sistema operativo di sviluppo dell'applicazione. Nel capitolo successivo (capitolo 4) si ritornerà sul sistema operativo Android presentandone le caratteristiche con maggior dettaglio.

3.2.2 Idee per la realizzazione dell'applicazione mobile

Un punto fondamentale dell'applicazione doveva essere la sua ergonomia; si doveva curare abbastanza l'interfaccia utente in modo da minimizzare il

tempo di compilazione dei dati dei pazienti.
Per rispondere a tale obiettivo, la prima idea è stata quella di organizzare le diverse parti della scheda in più interfacce utenti o viste in modo da rendere comprensibile e visibile ogni parte della scheda. La suddivisione della scheda ritenuta buona è stata la seguente:

- *dati fissi del paziente e parametri monitorati all'arrivo*: sono i dati registrati direttamente all'ingresso del paziente nella Recovery Room;
- *gli scores*: sono dati che sono monitorati periodicamente (recovery scores e valori dei drenaggi);
- *i dati di dimissione*: raggruppano i parametri monitorati alla dimissione e le informazioni sullo stato finale del paziente e sul personale che si è occupato di lui durante il soggiorno in sala;
- *altre informazioni*: sono dati come provvedimenti terapeutici, allergie, note e osservazioni varie sul paziente. Possono essere inserite in qualsiasi momento del soggiorno del paziente nella sala.

Sempre nell'ottica di avere un prodotto che semplificasse il processo di raccolta dei dati, si è pensato a minimizzare il più possibile l'inserimento dei dati testuali, cioè a ridurre al minimo l'inserimento dei caratteri, massimizzando l'uso dei campi con liste a scelte multiple e dei campi autocompletabili.

3.2.3 Scelte per il trasferimento dei dati e per la loro gestione nel lato server

Durante il soggiorno del paziente in Recovery Room, tutti i suoi dati sono salvati nel dispositivo. Alla dimissione i dati sono recuperati dal dispositivo e trasferiti su un server usando il protocollo FTP supportato dal sistema. I file costruiti e trasferiti per ogni paziente sono i seguenti:

- Un file in formato csv di cui il contenuto è inserito in coda ad un altro file csv globale che contiene i dati di tutti i pazienti passati nella RR in modo da permettere ai medici di consultare tali dati con un foglio di calcolo elettronico.

- Un file strutturato in formato HTML che viene creato in modo da organizzare i dati di ogni paziente come necessitano i medici, quindi in maniera simile alla scheda cartacea. Questa scelta si è resa indispensabile poichè non esisteva ancora una libreria pdf per la creazione dei file strutturati sotto Android, per cui si è usata una strategia alternativa che consiste nella creazione di un file strutturato in HTML direttamente sul dispositivo, cosicché, una volta che il file viene trasferito sul server, si utilizza uno script per trasformare il file in formato pdf.

Procedendo in questo modo si è riusciti a fornire ai medici la possibilità di consultare ugualmente sia i dati del singolo paziente (usando direttamente il loro file pdf riassuntivo), che quelli dell'insieme dei pazienti passati per la RR (aprendo direttamente il file csv globale con un programma per i fogli di calcolo elettronico).
Per rispondere all'obiettivo di avere anche una versione cartacea del riassunto dello stato dei pazienti in RR da mettere nella loro cartella clinica alla dimissione, la soluzione è stata quella di mandare in stampa una copia del file in formato pdf creato per ogni paziente prima della dimissione.

3.2.4 Scelte per la costruzione dei predittori

Scopo di questa parte del lavoro è usare algoritmi di classificazione per scoprire delle funzioni o costruire degli alberi di decisione che mappano i casi di pazienti in classi predefinite. In questo contesto, i parametri d'ingresso del processo di classificazione sono i parametri del paziente registrati al suo arrivo in RR: età, reparto di provenienza, anestesia, ASA, SpO2, PA massima, PA minima, FC, Bromage Scale.
I parametri del paziente di cui si vuole predire l'andamento sono: livello di coscienza, respirazione, emodinamica, dolore, PONV e brivido. Per ognuno di questi parametri si sono definite due classi: Good (buono) e Bad (non buono).
Lo scopo è di costruire degli alberi di decisione per predire l'andamento di ognuno dei parametri citati (cioè predire se il parametro avrà stato buono o non buono durante la permanenza del paziente in RR.), in funzione di quelli registrati al suo arrivo in RR.

Per rispondere a questo obiettivo, si è scelto di usare l'applicazione C4.5 [6] realizzata da John Ross Quinlan, basato sull'algoritmo *Decision Tree Induction* di Earl Hunt [7] per la costruzione degli alberi di decisione.

Capitolo 4

Introduzione al sistema operativo Android

In questo capitolo verrà presentato il sistema operativo Android seguito da una breve descrizione di alcune parti che costituiscono la sua piattaforma. Successivamente verranno presentati i componenti principali di un'applicazione eseguibile su questo sistema.

4.1 Origine di Android

Android è un sistema operativo progettato per dispositivi mobili basato sul kernel Linux. Fu inizialmente sviluppato da Android Inc, una startup statunitense fondata nell'ottobre del 2003 e acquistata da Google nell'agosto del 2005. Lo sviluppo del sistema viene sostenuto oggi da Open Handset Alliance (OHA), una coalizione di aziende composta tra le altre da: Google, Intel, HTC, Motorola, Samsung, LG, Nvidia, Qualcomm, T-Mobile e altre aziende del settore dell'informatica e delle telecomunicazioni, che ha il fine di sviluppare applicazioni e hardware per smart- phone.
Il sistema operativo è stato annunciato il 5 novembre 2007 come primo prodotto dell'OHA, e rilasciato il 12 novembre. La prima versione commerciale di un cellulare fornito della prima versione del sistema operativo Android è stata rilasciata il 22 ottobre del 2008 negli Stati Uniti, sotto il nome

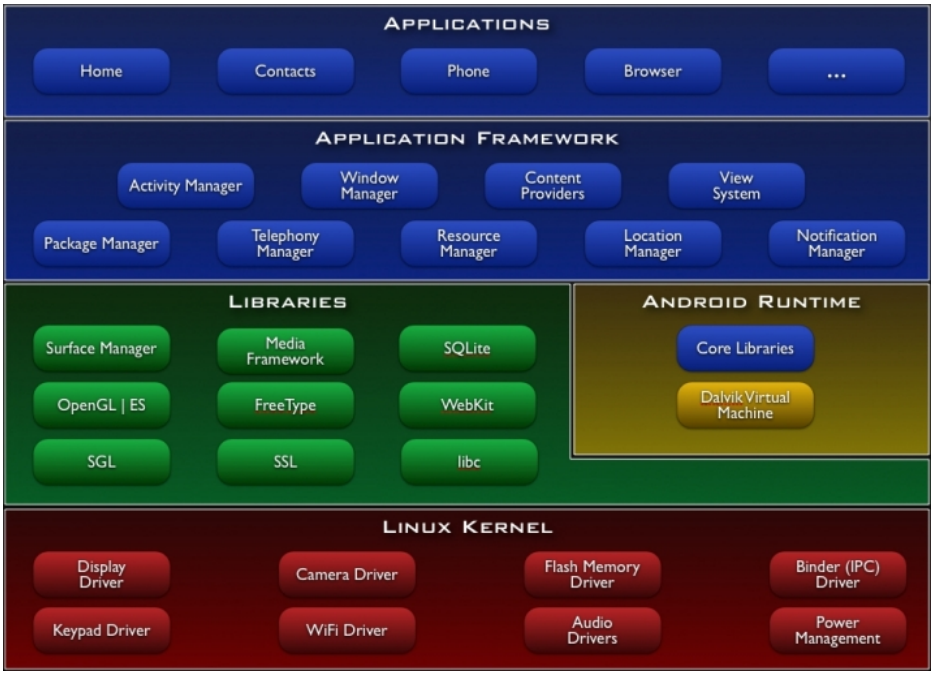


Figura 4.1: Struttura del sistema operativo Android

di T-Mobile G13[1]. L'hardware del telefono è stato realizzato dalla compagnia di Taiwan High Tech Computer (HTC). Da allora il sistema ha subito sviluppi continui fino all'ultima versione dichiarata come stabile (versione 2.3 – Android Gingerbread).

4.2 Architettura del sistema

Come i sistemi operativi per i personal computer, anche Android è organizzato secondo il paradigma a pila. Uno schema della sua struttura è rappresentato nella figura 4.1.

- **Strato principale: *Kernel Linux***
  Come si può vedere, lo strato inferiore dell'architettura è composto dal kernel Linux (la versione del kernel usata è la 2.6), che gestisce i servizi di base del sistema quali la gestione della sicurezza, della

[1] http://t-mobileg1.com/

memoria, dei processi, della coda di rete, dei driver. Il kernel agisce da strato di astrazione fra l'hardware e il resto del software.

- **Secondo strato**
  Questo strato contiene 2 sotto-strati, l'Android Runtime ed il sotto-strato Libraries delle librerie.

  ✓ **Android Runtime**
  Questo sotto-strato include un insieme di librerie che fornisce la maggior parte delle funzionalità disponibili nelle librerie di base del linguaggio di programmazione JAVA. Ogni applicazione Android gira in un proprio processo, con la propria istanza della macchina virtuale Dalvik (un equivalente della macchina virtuale Java adatta per l'uso su dispositivi mobili). Dalvik è stato scritto in modo che un dispositivo possa fare girare più macchine virtuali in modo efficiente. La macchina virtuale Dalvik esegue file nel formato Dalvik Executable (.dex) che sono fatti in modo da ottimizzare il consumo di memoria durante l'esecuzione dell'applicazione. In effetti le classi sono prima compilate da un compilatore Java e poi trasformate in formato .dex da un apposito strumento chiamato "dx". La Dalvik si basa sul kernel Linux per le funzionalità di base come il *threading* e la gestione a basso livello della memoria.

  ✓ **Libraries**
  Questo sotto-strato contiene un insieme di librerie C/C++ usate da alcuni componenti del sistema Android. Lo strato applicativo (*Application Framework*) costituisce un'interfaccia tra queste librerie e gli sviluppatori delle applicazioni. Le librerie principali sono le seguenti:

  - *System C library*: un'implementazione della libreria *standard C (libc)* specificatamente implementata per i sistemi Linux di tipo embedded.
  - *Media Libraries*: queste librerie sono basate su *PacketVideo's OpenCore* e supportano i più noti formati audio e video.

- *Surface Manager*: gestisce l'accesso al sottosistema del display e permette di comporre grafici in 2D e 3D.
- *LibWebCore*: un moderno motore di visualizzazione delle pagine web utilizzato dal browser di Android.
- *SGL*: un motore grafico 2D.
- *3D libraries*: un'implementazione basata sulle API *OpenGL ES 1.0*. Queste librerie usano gli acceleratori 3D se disponibili o inclusi.
- *FreeType*: per il rendering di bitmap e immagini vettoriali.
- *SQLite*: un potente e leggero motore di database relazionale disponibile per tutte le applicazioni.

- **Terzo strato:** ***Application Framework***
  Come detto precedentemente, questo strato è un'interfaccia tra le librerie e gli sviluppatori. Android offre agli sviluppatori la possibilità di creare applicazioni estremamente ricche e innovative consentendo loro di sfruttare appieno le funzionalità del dispositivo.
  Questo livello dello stack è necessario per rendere disponibili i servizi del sistema operativo alle applicazioni al livello superiore. Come detto in precedenza, queste API sono comuni sia alle applicazioni ufficiali che a quelle sviluppate in modo indipendente. L'obiettivo fondamentale nella progettazione di questo livello è il riuso del codice. Ovvero ogni applicazione può rendere disponibile alle altre servizi specifici permettendo così la progettazione in modo semplice di servizi nuovi basati su quelli preesistenti. Gli elementi più importanti del framework sono:

  - *View System*: è l'insieme di oggetti grafici che permettono la costruzione dell'interfaccia con l'utente[2]
  - *Content Providers*: un content provider è un elemento che permette la condivisione di dati fra applicazioni. La gestione dei dati memorizzati è lasciata all'implementazione del singolo provider mentre tutti devono implementare interfacce comuni per

[2]Per una collezione di esempi di view consultare l'indirizzo: http://code.google.com/android/reference/view-gallery.html

permettere l'esecuzione di query e l'inserimento di dati a ogni applicazione.

- *Resource Managers*: permettono l'accesso a tutti gli oggetti non costituiti da codice sorgente, quali immagini, file di configurazione dell'interfaccia, stringhe etc.
- *Notification Managers*: permettono alle applicazioni di presentare all'utente notifiche di eventi asincroni che avvengono in background.
- *Activity Managers*: gestiscono il **life-cycle** delle activity, le quali sono oggetti che rappresentano una singola operazione che l'utente può eseguire. Le activity, essendo la base di ogni programma sviluppato in Android, verranno descritte in modo approfondito in seguito (paragrafo 4.3.2).

- **Quarto strato: *Applications***
  Questo strato contiene un insieme di applicazioni fondamentali incorporate nel sistema. Infatti la versione base di Android non presenta solamente il sistema operativo ma anche applicazioni base di frequente utilizzo come un browser, un client di posta elettronica, un'agenda per la gestione dei contatti etc. A esse possono essere aggiunti altri programmi non appartenenti alla distribuzione ufficiale. La filosofia della OHA riguardo lo sviluppo di nuove applicazioni può essere riassunto con la frase: *"All applications are created equal"*. Ovvero: non c'è differenza fra i prodotti ufficiali e quelli da sviluppare in modo indipendente. Tutti si basano sulle stesse librerie e hanno pari privilegi nell'accesso alle risorse messe a disposizione dal sistema operativo.

4.3 Basi delle applicazioni Android

4.3.1 Gestione delle applicazioni (dal sistema)

La applicazioni Android sono scritte in linguaggio Java. Gli strumenti di *Android SDK* compilano il codice, compresi i dati e le risorse ad esso as-

sociati, in un unico file di archivio Android con estensione *.apk*. Tutto il materiale in un file apk è considerato come un'applicazione ed è il file che i dispositivi android usano per installare l'applicazione.
Una volta installata nel dispositivo, ogni applicazione vive nel suo spazio protetto conforme a quanto segue:

- Android è un sistema operativo multi-utente in cui ogni applicazione appartiene a un utente differente.
- Di default il sistema assegna ad ogni applicazione un unico identificativo (*Linux user Id*) conosciuto e utilizzato solo dal sistema stesso. Assegna i permessi ai file delle applicazioni in modo che solo le applicazioni che hanno l'user ID associato ai file possano accedervi.
- Ogni processo ha la sua macchina virtuale (*Dalvik VM*), di conseguenza il codice di ogni applicazione viene eseguito in modo isolato dal codice di altre applicazioni.
- Di default ogni applicazione viene eseguita nel suo proprio processo. Android avvia il processo quando uno dei componenti dell'applicazione deve essere eseguito, e lo termina quando il processo non è più richiesto o quando deve liberare la memoria per fare spazio ad altre applicazioni.
  Con le caratteristiche elencate sopra, il sistema Android implementa il principio di *"least privilege"* cioè privilegio minimo nel senso che di default ogni applicazione ha accesso solo ai componenti di cui esso necessita per la sua esecuzione e niente altro. Questo crea un ambiente sicuro e protetto in cui le applicazioni non possono accedere a parti del sistema di cui non hanno necessità.
  Tuttavia ci sono modi per un'applicazione di condividere dei dati con altre e anche modi per accedere ai servizi del sistema.
- È possibile fare in modo che due applicazioni usino lo stesso *Linux User Id*, nel quel caso una avrà la possibilità di accedere ai file dell'altra e viceversa. In questo caso può anche essere possibile ottimizzare le risorse del sistema facendo girare le applicazioni con lo stesso *Linux User Id* nello stesso processo e con la stessa macchina virtuale.

- Un'applicazione può richiedere il permesso per accedere ai dati del dispositivo, come i contatti dell'utente, sms, memoria rimovibile (SD card), Bluetooth, etc. Tutti questi permessi devono essere dati dall'utente durante l'installazione dell'applicazione.

4.3.2 Componenti di un'applicazione

Un'applicazione Android può essere composta da uno o più componenti principali. Ogni componente rappresenta un punto diverso attraverso il quale il sistema può accedere all'applicazione. Non tutti i componenti sono punti di ingresso effettivo per l'utente e alcuni dipendono l'uno dall'altro ma ognuno esiste come un'entità a sè stante e svolge un ruolo specifico. Ognuno rappresenta un blocco unico che contribuisce a definire lo stato generale dell'applicazione.
Esistono quattro tipi diversi di componenti di applicazioni. Ogni tipo ha una funzione distinta e un ciclo di vita distinto che definisce come il componente viene creato e distrutto. Di seguito un elenco di questi componenti.

✓ **Activities**

Ogni singola operazione che l'utente può svolgere è rappresentata da una Activity. La creazione di una nuova azione prevede la definizione di una nuova classe, la quale estende la classe Activity che si trova nel package android.app dell'SDK. La maggior parte delle Activity prevede un'interazione con l'utente. Per esempio un'applicazione che gestisce la posta elettronica può avere una activity che mostra la lista dei nuovi messaggi, un'altra che permette di comporre un nuovo messaggio e un altra per leggere i messaggi. Si rende quindi necessaria la creazione di una interfaccia grafica e la sua associazione alla corrispondente Activity. La *Dalvik Virtual Machine* mantiene in memoria una pila delle attività invocate dall'utente durante l'utilizzo dell'applicazione. Ogni volta che l'utente invoca una nuova schermata (ovvero un'attività) essa viene inserita alla sommità della pila. Sul fondo si troveranno invece le attività meno recenti, che non sono visualizzate attualmente sullo schermo. Questo meccanismo rende molto facile e veloce il browsing fra le differenti schermate. È quindi molto importante

comprendere a fondo il cosiddetto *life-cycle* delle activity, ovvero la gestione degli stati nei quali esse si possono trovare. Tre sono le modalità principali nelle quali un'Activity si può trovare:

- ***Running***: l'activity è visualizzata sullo schermo e l'utente interagisce con essa.

- ***Paused***: l'utente non può più interagire con l'attività ma il suo layout è ancora visibile (ad esempio sopra di esso appare una finestra di dialogo). Le activity in questo stato mantengono tutte le informazioni riguardanti il loro stato e possono essere terminate dalla *DVM* nel caso di scarsità di risorse.

- ***Stopped***: quando l'utente passa da un'Activity a un'altra, la prima è posta in stato di stop. Ciò significa che una nuova activity verrà posta sulla cima della pila. Le attività in stop sono le prime a essere terminate in caso di necessità di ulteriori risorse.

La figura 4.2 mostra i passaggi di stato di una activity. I rettangoli rappresentano le chiamate ai metodi di callback, ognuno dei quali svolge una funzione specifica nell'ambito del ciclo di vita dell'applicazione.
Tutti i metodi possono essere oggetto di override da parte del programmatore, che può quindi adattarli alle proprie esigenze.

- *onCreate(Bundle)*: questo metodo è invocato dalla *DVM* quando l'Activity viene creata per la prima volta ed è necessario alla sua inizializzazione. Le operazioni fondamentali che vengono svolte in questo metodo riguardano la definizione dell'interfaccia grafica attraverso un file *xml*[3], l'associazione di ogni elemento dell'interfaccia alla corrispondente azione desiderata e l'inizializzazione delle strutture dati utilizzate quali liste e basi di dati. L'oggetto di tipo *Bundle* passato come parametro è necessario al passaggio di informazioni riguardanti lo stato dell'applicazione.

[3]Extensible Markup Language, è un metalinguaggio di markup, ovvero un linguaggio marcatore che definisce un meccanismo sintattico che consente di estendere o controllare il significato di altri linguaggi marcatori

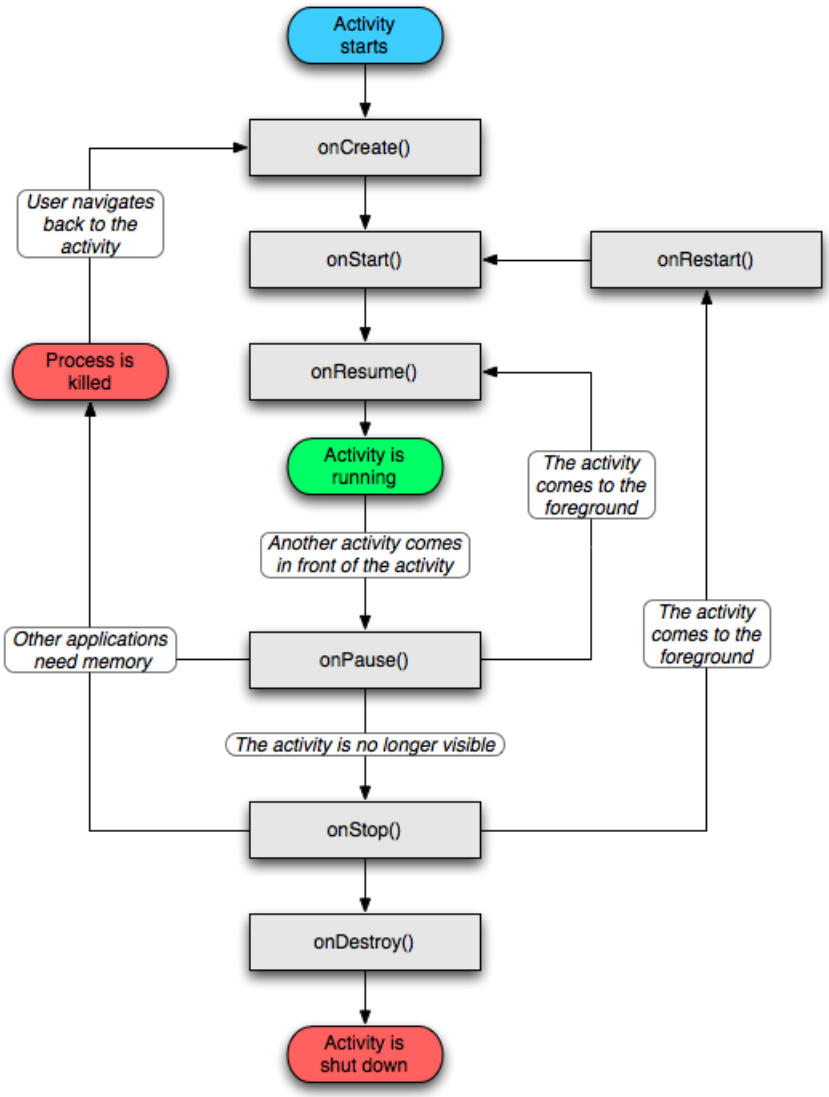


Figura 4.2: Ciclo di vita di una Activity

- *onPause()*: ogni volta che l'utente lascia un'Activity la *DVM* invoca questo metodo nel quale il programmatore dovrà aver specificato operazioni riguardanti il salvataggio dello stato raggiunto dall'applicazione.

- *onStart()*: quando l'activity è visualizzata sullo schermo del dispositivo viene invocato questo metodo.

- *onRestart()*: metodo chiamato ogni volta che l'activity diventa visibile all'utente dopo che è stata posta nello stato *paused*. Subito dopo la terminazione del metodo viene invocato *onStart()*.

- *onResume()*: il metodo è chiamato subito prima che l'activity sia posta sulla cima della pila e che quindi sia possibile per l'utente interagire con essa.

- *onStop()*: quando un'Activity viene oscurata da un'altra viene invocato questo metodo.

- *onDestroy()*: il metodo è lanciato subito prima della distruzione finale dell'Activity.

Nello schema si possono quindi individuare 3 cicli:

- ***ciclo di vita***: ricadono in questo insieme tutte le operazioni comprese fra le chiamate ai metodi *onCreate()*, nel quale si allocano tutte le risorse necessarie, e *onDestroy()*, nel quale le risorse sono rilasciate.

- ***ciclo di visibilità***: compreso fra i metodi *onStart()* e *onStop()*. In questo periodo di tempo l'interfaccia dell'actitvity è visibile all'utente ma può non essere possibile l'interazione con essa a causa della presenza di altre activity in primo piano.

- ***ciclo di vita in primo piano***: in questo insieme sono comprese tutte le operazioni fra i metodi *onResume()* ed *onPause()*. L'Activity è qui completamente disponibile all'utente il quale può interagire con essa, ad esempio riempiendo dei form di input.

Sebbene più activity possano concorrere alla formazione di una sola applicazione come nel caso dell'applicazione per gestire la posta elettronica, ognuna è indipendente dalle altre. In quanto tale, un'applicazione diversa può avviare una qualunque di quelle Activity (se l'applicazione lo permette). Per esempio un'applicazione per webcam può lanciare un'attività dell'applicazione di posta elettronica in modo da permettere all'utente di condividere una foto.

✓ **Services**

Un Service è un componente dell'applicazione che viene eseguito in background per un periodo di tempo indefinito, e che non ha quindi interazione diretta con l'utente. È importante notare che il codice contenuto in una classe che estende Service è eseguito nel thread principale dell'applicazione. Nel caso in cui siano eseguite operazioni che utilizzano molto la CPU o la connessione a una rete è quindi consigliabile la creazione di un nuovo thread. Come per le activity anche nei Service sono presenti metodi di *callback* eseguiti in momenti specifici dell'esecuzione. Essi sono ad esempio *onCreate()*, *onDestroy()*, con lo stesso significato di quello specificato per la classe Activity.

✓ **Content Providers**

Gli oggetti di tipo *ContentProvider* hanno la funzione di garantire l'accesso a dati condivisi da più applicazioni. L'effettiva implementazione del salvataggio dei dati in memoria di massa non è specificato ed è lasciata al programmatore. Tutti i *content provider* però devono implementare interfacce predefinite che specificano il modo col quale le query possono essere effettuate e le modalità di presentazione del risultato. I tipi di dati salvati possono essere di vario tipo[4]: contatti telefonici, file audio e video. Ogni *ContentProvider* fornisce un URI (Uniform Resource Identifier) ai client che vogliono utilizzare i suoi dati; è attraverso questa stringa che è possibile effettuare richieste o inserire dati. Ad esempio:

[4]Per consultare l'elenco dei tipi di dato astratti che rappresentano i dati memorizzati in un ContentProvider si veda: http://code.google.com/android/reference/android/provider/package-summary.html

L'URI *content://contacts/people/* permette di ottenere tutti gli elementi della tabella people; invece l'URL (Uniform Resource Locator) *content://contacts/people/10* ritorna la singola persona identificata dal codice 10. L'inserimento di dati invece avviene specificando oltre a un URI anche una mappa che fa corrispondere alle colonne della tabella relazionale i campi della riga da inserire.

✓ **Broadcast Receivers**

Un broadcast receiver è un componente che risponde alle notifiche che arrivano in modalità broadcast, per esempio una notifica del sistema che segnala che lo schermo si è spento o la batteria è scarica, ecc. Le applicazioni possono anch'esse avviare delle notifiche broadcast, per esempio per segnalare ad altre applicazioni la messa a disposizione di certi dati di cui loro necessitavano. I broadcast receiver non hanno un'interfaccia utente ma possono creare delle notifiche nella barra delle notifiche per allertare l'utente quando si verifica un evento broadcast. Un ricevitore broadcast è più comunemente usato come un *"gateway"* per altri componenti ed è destinato a svolgere una quantità di lavoro minima. Per esempio si potrebbe avviare un servizio per eseguire alcune operazioni in risposta ad un particolare evento.
Un aspetto importante di Android è il fatto di poter permettere a un'applicazione di avviare un componente di un'altra applicazione. Per esempio se si vuole che l'utente catturi una foto tramite la fotocamera del suo dispositivo dentro un'applicazione e se esiste già un'altra applicazione che fornisce tale funzionalità, si può avviare direttamente il suo componente (in questo caso una activity) per fare la cattura invece di sviluppare un'altra activity per fare la stessa cosa. Alla fine la foto viene restituita all'utente che non si rende conto che l'acquisizione è stata fatta da un'altra applicazione.
Quando il sistema avvia un componente di un'applicazione, esso avvia un processo per tale applicazione (se non già in esecuzione) e crea le istanze delle classi necessarie per il funzionamento del componente. Per esempio se un'applicazione avvia un'activity dell'applicazione che gestisce la fotocamera per la cattura delle foto, tale activity viene eseguita dentro il

processo dell'applicazione di cui fa parte (in questo caso sarà il processo dell'applicazione che gestisce la fotocamera) e non nel processo dell'applicazione che l'ha avviata. Pertanto a differenza di altre applicazioni su altri sistemi, un'applicazione Android non ha un solo punto d'ingresso (non c'è la funzione *main()* per esempio).
Le applicazioni non possono attivare direttamente il componente di un'altra applicazione ma il sistema lo può fare. Questo è dovuto al fatto che ogni applicazione viene eseguita nel proprio processo con i permessi che impediscono ad altre applicazioni di avere accesso ai suoi file. Per attivare quindi un componente di un'altra applicazione, una richiesta (tramite messaggio) deve essere mandata al sistema specificando l'intenzione di volere avviare quel componente ed esso lo farà.

4.3.3 Attivazione dei componenti

Tre dei quattro tipi di componenti - activity, service, broadcast receiver - sono attivati tramite un messaggio asincrono chiamato Intent (intento in italiano). Gli intent creano (in fase di esecuzione) un accoppiamento tra due componenti che possono essere della stessa applicazione oppure di due applicazioni diverse. Possono essere visti come dei messaggeri che richiedono delle azioni da altri componenti.
Un intent è creato con un oggetto Intent che definisce il messaggio per l'attivazione di uno specifico componente.
Per le activity e i service, un intent definisce l'azione da eseguire (per esempio per vedere o inviare qualcosa) e può specificare l'URI dei dati su cui agire (oltre alle altre cose di cui il componente potrebbe necessitare per avviarsi). Ad esempio un intent può trasmettere una richiesta a un activity per farsi aprire un'immagine o una pagina web. In alcuni casi un activity può essere avviata per ricevere un risultato, nei quali casi l'activity restituisce il risultato usando un intent (per esempio si può avviare un intent per consentire all'utente di selezionare un contatto personale: il risultato sarà restituito tramite un intent contenente l'URI che punta al contatto selezionato).
Per i broadcast receiver l'intent definisce semplicemente il messaggio da

trasmettere in broadcast (per esempio un broadcast per segnalare il livello di batteria basso conterebbe la stringa "batteria scarica"). Un content provider invece non può essere attivato tramite un intent. Può essere attivato piuttosto quando è interrogato da un *ContentResolver*. Quest'ultimo gestisce tutte le transazioni direttamente con il content provider in modo che il componente che ha originato le transazioni con il content provider non comunichi direttamente con esso (content provider) ma piuttosto tramite metodi del *ContentResolver*. Questo lascia un strato di astrazione tra il fornitore di contenuti e il componente richiedente (per motivi di sicurezza).
Ci sono metodi diversi per attivare ogni tipo di componente.

- Si può avviare una activity in due modi diversi. Essi si identificano nei due metodi *startActivity(Intent)* e *startActivityOnResult(Intent,int)*. Il primo permette il semplice lancio di un'Activity identificata dall'oggetto di tipo Intent. Il secondo metodo invece prevede che la classe invocata ritorni un risultato; esso sarà disponibile facendo override, nel chiamante, del metodo *onActivityResult(int,int,Result)* che verrà invocato alla terminazione dell'attività chiamata.

- Vi sono due modi per avviare un service:

  - All'invocazione del metodo *Context.startService()* una nuova istanza di un service è creata (con il metodo *onCreate()*) e lanciata (con il metodo *onStart(Intent,int)*). L'esecuzione prosegue fino alla chiamata del metodo *Context.stopService()*.

  - Se il metodo *Context.onBind()* è invocato viene instaurata una connessione persistente a uno specifico servizio che, se non esiste già, può venir creato. In questo caso il metodo *onStart()* non è chiamato. Il chiamante otterrà così un identificativo del Service, cosa che gli permetterà di effettuare richieste.
    Un Service può essere al tempo stesso avviato con il metodo *onStart()* e oggetto di connessioni. Esso potrà essere terminato solo quando non ci saranno più connessioni attive e il metodo *Context.stopService()* non verrà chiamato.

- Un broadcast può essere iniziato passando un Intent ai metodi come *sendBroadcast()*, *sendOrderedBroadcast()*, oppure *sendStickyBroadcast()*.
- Per effettuare una query su un content provider si chiama *query()* del *ContentResolver*.

4.4 Il Manifest file

Affinché il sistema Android possa avviare un componente, deve essere informato della sua esistenza leggendo il file *AndroidManifest.xml* dell'applicazione (il file di "Manifesto"). Ogni applicazione deve dichiarare tutti i suoi componenti in quel file che deve essere alla radice della cartella del progetto.
Oltre a dichiarare i componenti dell'applicazione, il file di manifesto svolge ad esempio i seguenti compiti:

- Identifica tutti i permessi utente richiesti dall'applicazione come ad esempio l'accesso a Internet, o l'accesso in lettura ai contatti dell'utente, ecc.
- Dichiara il livello minimo dell'API su cui l'applicazione si basa.
- Dichiara i componenti hardware e software utilizzati dall'applicazione come ad esempio la fotocamera, il servizio Bluetooth, etc.
- Dichiara le librerie API che devono essere *linkate* (oppure associate) all'applicazione (sono librerie che non appartengono a quelle del *framework Android*) come, ad esempio, le librerie *Google Maps*.

4.4.1 Dichiarazione dei componenti

Il compito principale del file di manifesto è quello di informare il sistema dei componenti dell'applicazione. Nel Listing4.1 si vede come un file di manifesto può dichiarare un'activity.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ... >
  <application android:icon="@drawable/app_icon.png" ... >
      <activity android:name="com.example.project.ExampleActivity"
```

```
                android:label="@string/example_label" ... >
        </activity>
        ...
    </application>
</manifest>
```

Listing 4.1: Il Manifest file

Nell'elemento `<application>`, l'attributo `android:icon` punta sulla risorsa (immagine in questo caso) che fa da logo all'applicazione. Nell'elemento `<activity>` l'attributo `android:name` specifica il nome completo della classe che implementa l'activity e l'attributo `android:label` punta alla stringa utilizzata come nome dell'activity visto dall'utente.
Tutti i componenti dell'applicazione devono essere dichiarati nel modo seguente:

- `<activity>` per le actitvity
- `<service>` per i service
- `<receiver>` per i broadcast receiver
- `<provider>` per i content provider

Le activity, i service, e i content provider inclusi nel codice e non dichiarati nel file di manifesto non sono visibili dal sistema e di conseguenza non possono mai essere eseguiti. Tuttavia, i broadcast receiver possono essere dichiarati nel file di manifesto o creati dinamicamente nel codice (come oggetti `BroadcastReceiver`) e registrati al sistema chiamando il metodo `registerReceiver()`.

4.4.2 Dichiarazione delle capacità dei componenti

Come detto precedentemente nel paragrafo relativo all'attivazione dei componenti (paragrafo 4.3.3), si usa un intent per avviare le activity, i service e i broadcast receiver. Questo può essere fatto nominando esplicitamente il componente che si vuole avviare (usando il nome della classe del componente) nell'intent. Tuttavia il vero potere degli intenti risiede nel concetto di *"azioni di intenti"* (oppure intenti impliciti). Con le azioni di intenti è

sufficiente descrivere il tipo di azione che si desidera eseguire (e facoltativamente, anche i dati su cui si desidera eseguire l'azione) e si lascia il sistema trovare il componente sul dispositivo in grado di eseguire tali azioni e avviare esso. Se vi sono più componenti in grado di eseguire un'azione descritta, allora l'utente deve scegliere quello da usare.

4.4.3 Dichiarazione dei requisiti per il corretto funzionamento dell'applicazione

Ci sono una varietà di dispositivi che supportano il sistema Android ma non offrono tutti le stesse caratteristiche e funzionalità. Al fine di evitare che l'applicazione venga installata su un dispositivo che non ha tutte le caratteristiche necessarie al suo corretto funzionamento, è importante definire chiaramente il profilo del dispositivo che può supportare l'applicazione nel file di manifesto. La maggior parte di queste dichiarazioni hanno valore informativo e il sistema non le legge, ma i servizi esterni come ad esempio l'Android Market le leggono in modo da filtrare certi dispositivi quando l'utente cerca delle applicazioni compatibili con il dispositivo in uso.
Ad esempio, se un'applicazione richiede una fotocamera e usa le API introdotte in *Android 2.1* (API level 7), si dovrebbero dichiarare questi requisiti nel file di manifesto. In questo modo i dispositivi che non dispongono di una fotocamera e una versione di Android superiore o uguale a 2.1 non potranno installare l'applicazione tramite *Android Market*.
Tuttavia è possibile dichiarare che l'applicazione richiede una fotocamera, ma non obbligatoriamente. In tal caso l'applicazione deve eseguire un controllo in fase di esecuzione per determinare se il dispositivo è dotato di una fotocamera e disattivare le funzioni che utilizzano la fotocamera se non è disponibile.

4.5 Risorse di un'applicazione

Un'applicazione Android è composta dal codice sorgente e altre risorse che sono separate dal codice come ad esempio le immagini, i file audio, e altre relative all'aspetto visuale dell'applicazione. Ad esempio si dovrebbero

definire le animazioni, i menu, gli stili, i colori, i layout delle interfacce utente delle activity quando esse sono implementate usando i file XML. Questo rende più facile aggiornare varie caratteristiche dell'applicazione senza modificare il codice sorgente e fornire un insieme di risorse alternative consente di ottimizzare l'applicazione per una varietà di configurazioni dei dispositivi (come, ad esempio, la possibilità di usare più lingue, varie grandezze di schermi, ecc.).
Per ogni risorsa inserita in un progetto Android, gli strumenti dell'SDK definiscono un unico ID utilizzabile per referenziare la risorsa all'interno del codice sorgente dell'applicazione o all'interno di altre risorse definite in XML. Per esempio se un'applicazione contiene un file immagine di nome `logo.png` (inserita nella directory `res/drawable/`), gli strumenti dell'SDK generano un unico ID denominato `R.drawable.logo` che può essere utilizzato all'interno del codice per fare riferimento all'immagine.
Uno degli aspetti importanti della separazione tra il codice sorgente e le risorse è la possibilità di fornire risorse alternative per configurazioni diverse di dispositivi. Per esempio, definendo l'interfaccia utente tramite file XML, è possibile tradurre le stringhe in altre lingue e salvarle in file diversi (ad esempio `res/values-fr/` per i valori di stringhe in francese e `res/values-it/` per valori di stringhe in italiano) in modo da adattare l'interfaccia dell'applicazione alla lingua impostata sul dispositivo. Un altro esempio è la possibilità di creare diversi layout per le activity a seconda dell'orientamento e la grandezza dello schermo del dispositivo. Si potrebbe voler avere, per esempio, un layout con i bottoni allineati in verticale quando lo schermo ha un orientamento portrait (verticale), e volere il contrario quando lo schermo ha un orientamento landscape (orizzontale). Avendo definito due layout e in modo appropriato per ogni tipo di orientamento dello schermo, il sistema cambierà automaticamente il layout in funzione dell'orientamento del dispositivo.

Capitolo 5

Fase di progettazione

In questa parte verrà presentata la fase di progettazione dell'applicazione mobile senza entrare nei dettagli implementativi, rimandati al capitolo 6 che tratta della realizzazione.

5.1 Interfacce utente - Organizzazione della scheda in views

A riguardo delle dimensioni del tipo di dispositivo mobile di cui siamo interessati e visto le dimensioni della scheda del paziente, l'idea di suddividere la scheda in più views (o interfacce utente) è stata una soluzione inevitabile. Per fare ciò, si è pensato prima di suddividere il periodo di ricovero di un paziente in più fasi secondo le azioni eseguite dagli infermieri. Successivamente, tutti i parametri registrabili in ogni fase sono stati raggruppati.

5.1.1 Organizzazione del periodo di ricovero del paziente in fasi

Basandosi sui dati registrabili in determinati momenti del ricovero di un paziente, si è suddiviso il suo periodo di ricovero nelle seguenti fasi:

- *Fase 1 (arrivo del paziente)*: in questa fase, alcuni parametri possono già essere inseriti nella scheda del paziente al suo arrivo in RR. Questi parametri possono essere eventualmente suddivisi in parametri fissi

(cognome, nome, data di nascita, reparto, intervento, anestesista, anestesia, asa, numero di drenaggi) e parametri relativi al monitoraggio all'arrivo (ora arrivo, SpO2, pressione arteriosa, frequenza cardiaca, temperatura).

- *Fase 2 (permanenza del paziente in RR)*: durante questa fase, possono venire inseriti nella scheda gli scores (o recovery scores), i rilevamenti di drenaggi, i dati informativi (allergie, note e osservazioni), i provvedimenti terapeutici.

- *Fase 3 (dimissione del paziente)*: i dati che devono essere inseriti in questa fase sono i parametri relativi al monitoraggio di dimissione (stessi parametri di quelli relativi al monitoraggio all'arrivo), le informazioni sull'esito ed eventuale reparto in cui viene trasferito, le informazioni sul responsabile della dimissione e dell'infermiere che si è occupato del paziente durante il suo ricovero in RR.

Nella prima fase è possibile inserire solo i parametri appartenenti ad essa. Nella seconda e terza fase invece, è anche possibile tornare a registrare i parametri o dati che si potevano inserire nella fase precedente.

5.1.2 Le differenti views e il loro contenuto

Dopo aver suddiviso il periodo di ricovero del paziente in RR in più fasi, è stato possibile identificare un'organizzazione della scheda del paziente in più views in modo da ottimizzare la raccolta dei dati. Questa è la suddivisione:

- *view 1*: questa view dovrà contenere un formulario con i campi corrispondenti ai parametri registrabili nella fase 1. I campi potranno essere eventualmente suddivisi in due blocchi per separare i campi fissi dai campi corrispondenti ai parametri monitorati all'arrivo in RR.

- *view 2*: per l'inserimento degli scores e i rilevamenti dei drenaggi del paziente. Questa view dovrà contenere due liste che mostreranno gli scores già registrati e i rilevamenti dei drenaggi segnati successivamente. Di conseguenza l'inserimento dei nuovi scores e dei nuovi

valori dei drenaggi sarà possibile attraverso altre due views. Queste altre due saranno accessibili solo a partire dalla view 2 principale.

- *view 3*: questa view dovrà contenere un formulario per l'inserimento dei dati registrabili nella fase 2 tranne gli scores e i rilevamenti dei drenaggi. I campi corrispondenti ai dati da registrare in questo formulario potranno essere eventualmente suddivisi in 3 blocchi: uno per le allergie, un altro per i provvedimenti terapeutici e l'ultimo per le note e osservazioni.

- *view 4*: questa sarà costituita da un formulario contenente i campi organizzati in modo da poter permettere l'inserimento dei dati e dei parametri registrabili nella fase 3. I campi di questo formulario potranno essere eventualmente separati in due blocchi: uno per i parametri monitorati alla dimissione, e l'altro per le informazioni sull'esito e sullo staff che si è occupato del paziente durante il ricovero in RR.

5.1.3 Specificità sull'inserimento dei valori dei drenaggi

La view per l'inserimento dei rilevamenti dei valori dei drenaggi non poteva essere pensata come le altre in quanto il numero di campi presenti nel formulario che essa contiene dipende dal numero di drenaggi del paziente. Per tale view si è pensato quindi a una interfaccia dinamica che viene costruita ogni volta che si vuole fare un nuovo inserimento e che si basa appunto sul numero dei drenaggi affettivamente presente sul paziente in questione.

5.1.4 Specificità sull'inserimento degli scores

Ugualmente al caso dell'inserimento dei valori dei drenaggi, la view per l'inserimento dei nuovi scores non poteva essere pensata come le altre, soprattutto per il fatto che il numero di volte in cui vengono inseriti gli scores è variabile e dipende dall'andamento del paziente durante il ricovero. Si è quindi pensato a un inserimento dinamico in cui ogni volta che si vuole inserire degli scores, viene costruito un formulario che permette di inserire

una colonna intera della tabella degli scores che si può vedere sulla scheda cartacea.

5.2 Gestione della scala usata per i recovery scores

La parte della scheda relativa ai recovery scores è risultata la più complessa da gestire. La prima difficoltà incontrata è dovuta al fatto che la scala a punteggi usata doveva poter variare. In letteratura esiste più di una scala a punteggi; i medici dell'Ospedale dell'Angelo utilizzavano una scala progettata da loro, però volevano avere la possibilità di cambiare questa scala in caso di bisogno. Infatti durante la realizzazione di questo progetto, l'hanno cambiata una volta.
La soluzione trovata per rispondere a questo problema è stato di formalizzare la scheda a punteggi con un file XML. Questo permette di avere un file con una struttura fissa anche se il contenuto potrà eventualmente variare.

5.2.1 Struttura della scala a punteggi

Come si può vedere nell'esempio della scala di Aldrete di figura 2.1 presentata nel capitolo 2, una scala è composta da elementi (o parametri) principali che sono oggetto dei punteggi, e da altri elementi che sono associati a ogni parametro principale. Un parametro principale è in effetti un parametro dello stato del paziente (Attività, Respirazione, Coscienza, ecc.) che viene monitorato facendo uso dei punteggi. Per ogni parametro principale sono definiti i possibili stati rappresentativi e a ogni stato rappresentativo è associato un punteggio. Il punteggio di un parametro è quindi il punteggio associato allo stato che descrive meglio le condizioni del paziente nel momento in cui esso viene valutato. Di seguito viene illustrato un esempio di parametro che si può vedere nella scala di Aldrete:
Parametro principale: *Attività*; che serve a valutare l'attività fisica del paziente.
Stati rappresentativi:

- *Capace di muovere 4 estremità volontariamente o a comando*; al quale è stato associato il punteggio 2.

- *Capace di muovere 2 estremità volontariamente o a comando*; al quale è stato associato il punteggio 1.
- *Incapace di muovere le estremità volontariamente o a comando*; al quale è stato associato il punteggio 0.

5.2.2 Struttura XML della scala a punteggi

Si è scelta una struttura XML comprensibile e molto facile da modificare. L'elemento root del file, chiamato `RecoveryScoreFields`, contiene gli attributi `versione` e `Name` che permettono di specificare rispettivamente la versione e il nome della scala a punteggi. Esso contiene inoltre degli elementi chiamati `item` che possono rappresentare i parametri principali oppure i loro stati rappresentativi.
Un `item` di parametro principale ha i seguenti attributi:

- `type`: ha sempre valore *"parentField"*;
- `label`: rappresenta la sua stringa (o nome testuale) identificava;
- `id`: permette di identificare in modo univoco ogni parametro principale con un numero intero positivo.

Un `item` che rappresenta lo stato rappresentativo di un parametro principale ha i seguenti attributi:

- `type`: ha sempre valore *"field"*;
- `label`: stringa che descrive lo stato;
- `value`: rappresenta il punteggio associato allo stato in questione.

Tutti gli attributi degli item devono essere non vuoti.
Per l'esempio considerato nel paragrafo 5.2.1 precedente, si avrà il risultato riportato nella figura 5.1:

La figura 5.2 presenta la struttura di un esempio di file XML per una scala a punteggi contenente tre parametri principali. Nel caso specifico si tratta di una parte della scala di White e Song [8], che è una delle più diffuse in letteratura.

```
<item type="parentField" label="Attività" id="1"/>
<item type="field" label="Capace di muovere 4 estremità volontariamente o a comando" value="2" />
<item type="field" label="Capace di muovere 2 estremità volontariamente o a comando" value="1" />
<item type="field" label="Incapace di muovere le estremità volontariamente o a comando" value="0" />
```

Figura 5.1: Rappresentazione in XML del parametro "Attività" della scala di Aldrete

```
<?xml version="1.0" encoding="utf-8"?>
<recoveryScoreFields version="1" Name="Scala di White e Song">
    <item type="parentField" label="Livello di coscienza" id="1"/>
    <item type="field" label="Sveglio e orientato" value="2" />
    <item type="field" label="Risvegliabile con un minimo stimolo" value="1" />
    <item type="field" label="Responsivo solo alla stimolazione tattile" value="0" />

    <item type="parentField" label="Attività fisica" id="2"/>
    <item type="field" label="Muove tutte le estremità a comando" value="2" />
    <item type="field" label="Debolezza nel muovere le estremità" value="1" />
    <item type="field" label="Incapace di muovere volontariamente le estremità" value="0" />

    <item type="parentField"  label="Stabilità emodinamica" id="3"/>
    <item type="field" label="PA ± 15% dei valori preoperatori" value="2" />
    <item type="field" label="PA ± 30% dei valori preoperatori" value="1" />
    <item type="field" label="PA > 30% dei valori preoperatori" value="0" />
</recoveryScoreFields>
```

Figura 5.2: Parte della scala di White e Song strutturato in XML

In generale un parametro principale ha sempre tre stati rappresentativi, ma nella struttura del file non c'è questo vincolo. Negli esempi presentati si vede che i punteggi associati agli stati sono sempre 2 (quando lo stato è buono), 1 (per lo stato medio) e 0 (per uno stato non buono), ma in realtà questi valori non sono fissi. L'unica condizione da rispettare è che il primo stato deve sempre avere il valore più grande e questo ordine deve essere rispettato per tutti gli stati.

5.3 Progettazione della base di dati

Come già detto nel capitolo 4, il sistema Android incorpora SQLite, un efficiente *Database Management System (DBMS)* basato su *Structured Query Language (SQL*[1]*)* utilizzabile da tutte le applicazioni. Ogni applicazione può

[1]È un linguaggio di interrogazione per database progettato per leggere, modificare e gestire dati memorizzati in un sistema basato sul modello relazionale, per creare e modificare gli schemi di database, per creare e gestire strumenti di controllo ed accesso ai dati.

quindi creare e utilizzare una propria base di dati privata.
Il database è risulto fondamentale per la fase di raccolta dei dati. Infatti, durante il ricovero dei pazienti tutti le informazioni sono salvate in esso e vengono trasferiti verso il server solo alla dimissione del paziente. Avendo la necessità di monitorare più pazienti contemporaneamente con lo stesso dispositivo, la base di dati deve essere in grado di memorizzare le informazioni di più pazienti. Essa è stata realizzata soprattutto allo scopo di contenere in modo persistente i dati del paziente solo durante il suo ricovero in RR. Alla dimissione i dati sono completamente cancellati dal database, anche se è possibile mantenerli in memoria per un breve periodo di tempo. In realtà il dispositivo non ha abbastanza spazio per contenere i dati e lasciando numerosi dati nel database c'è anche il rischio di peggiorare le prestazioni della base di dati.

5.3.1 Modello ER del database

Dalla suddivisione dei parametri della scheda descritta nel paragrafo 3.2.2 e dall'organizzazione della scheda in views del paragrafo 5.1.2 è stato possibile realizzare il modello ER di figura 5.3 le cui entità e relazioni verranno ora brevemente descritte.

Entità

- *paziente*: rappresenta il singolo paziente in RR.
- *monitoraggio_arrivo*: rappresenta il pacchetto di dati costituto dai parametri monitorati all'arrivo del singolo paziente in RR.
- *monitoraggio_dimissione*: rappresenta il pacchetto di dati costituito dai parametri monitorati nella fase di dimissione del paziente.
- *recovery_scores*: rappresenta un singolo score del paziente (ossia una colonna degli scores) registrato in un momento preciso.

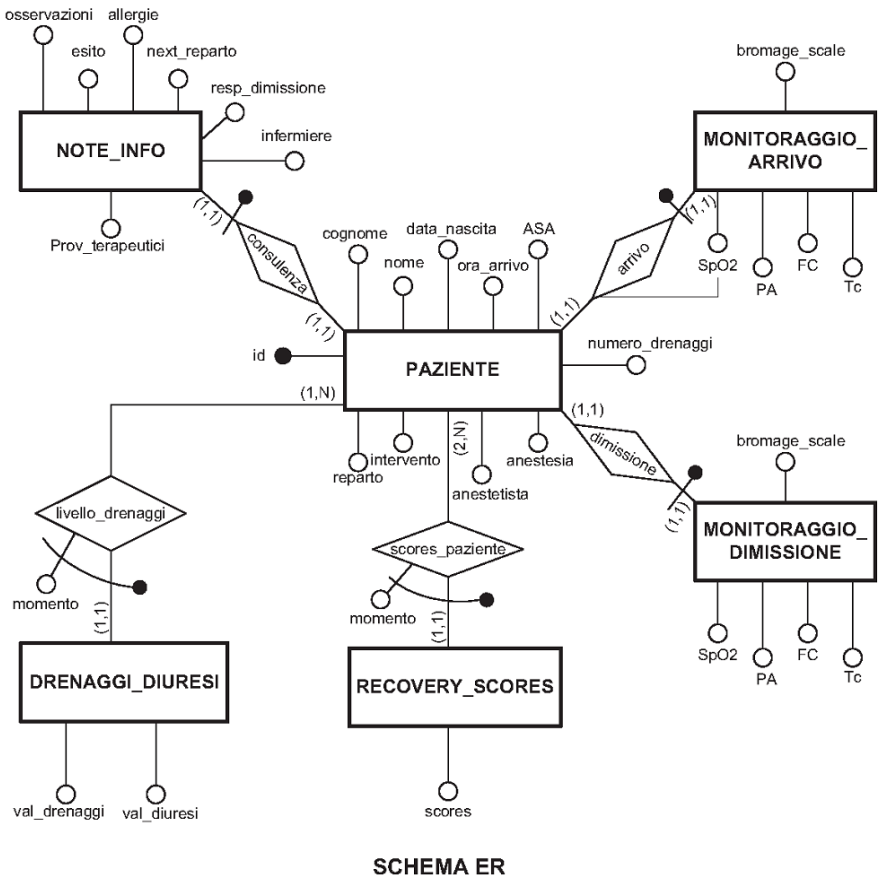


Figura 5.3: Modello ER del database

- *note_info*: rappresenta il pacchetto di dati informativi (allergie, provvedimenti terapeutici, responsabile dimissione, ecc.) registrabili per un singolo paziente.

- *drenaggi_diuresi*: rappresenta una singola registrazione dei valori dei drenaggi e della diuresi del paziente effettuato in un momento preciso.

Associazioni

- *arrivo*: relazione tra un paziente e un monitoraggio d'arrivo, in quanto ci devono essere una serie di parametri da monitorare all'arrivo del

paziente in RR.

- *dimissione*: relazione tra un paziente e un monitoraggio di dimissione, in quanto per ogni paziente ci devono essere una serie di parametri da registrare nel momento delle dimissioni dalla RR.
- *scores_paziente*: relazione tra un paziente e i suoi scores registrati durante il suo ricovero in RR.
- *livello_drenaggi*: relazione tra un paziente e i rilevamenti dei valori dei suoi drenaggi effettuati durante il suo ricovero in RR.
- *consulenza*: relazione tra un paziente e tutte le altre informazioni registrabili sul paziente durante il suo ricovero in RR, come per esempio i provvedimenti terapeutici, le allergie, ecc.

5.3.2 Dallo schema ER al modello relazionale

In questa fase non si sono prese molte decisioni, poiché il modello relazionale traduce senza grandi modifiche lo schema ER. Le uniche scelte rilevanti effettuate sono le seguenti:

- Fusione delle entità MONITORAGGIO_ARRIVO e MONITORAGGIO_DIMISSIONE in un'unica entità MONITORAGGIO aggiungendogli un nuovo attributo per differenziare il monitoraggio all'arrivo da quello effettuato alla dimissione del paziente.
- Aggiunta di una nuova entità STATO_COMPILAZIONE per controllare lo stato di compilazione della scheda e tenere un'informazione sullo stato dell'ultimo score del paziente.
- Sostituzione delle relazioni uno a uno oppure uno a molti tra l'entità PAZIENTE e le entità MONITORAGGIO, NOTE_INFO, RECOVERY_SCORES, DRENAGGI_DIURESI, STATO_COMPILAZIONE, aggiungendo a quest'ultimi un attributo id_paziente per identificare il paziente a cui i dati da loro contenuti si riferiscono.
- Aggiunta di un nuovo attributo all'entità RECOVERY_SCORES per poter caratterizzare lo stato dello score.

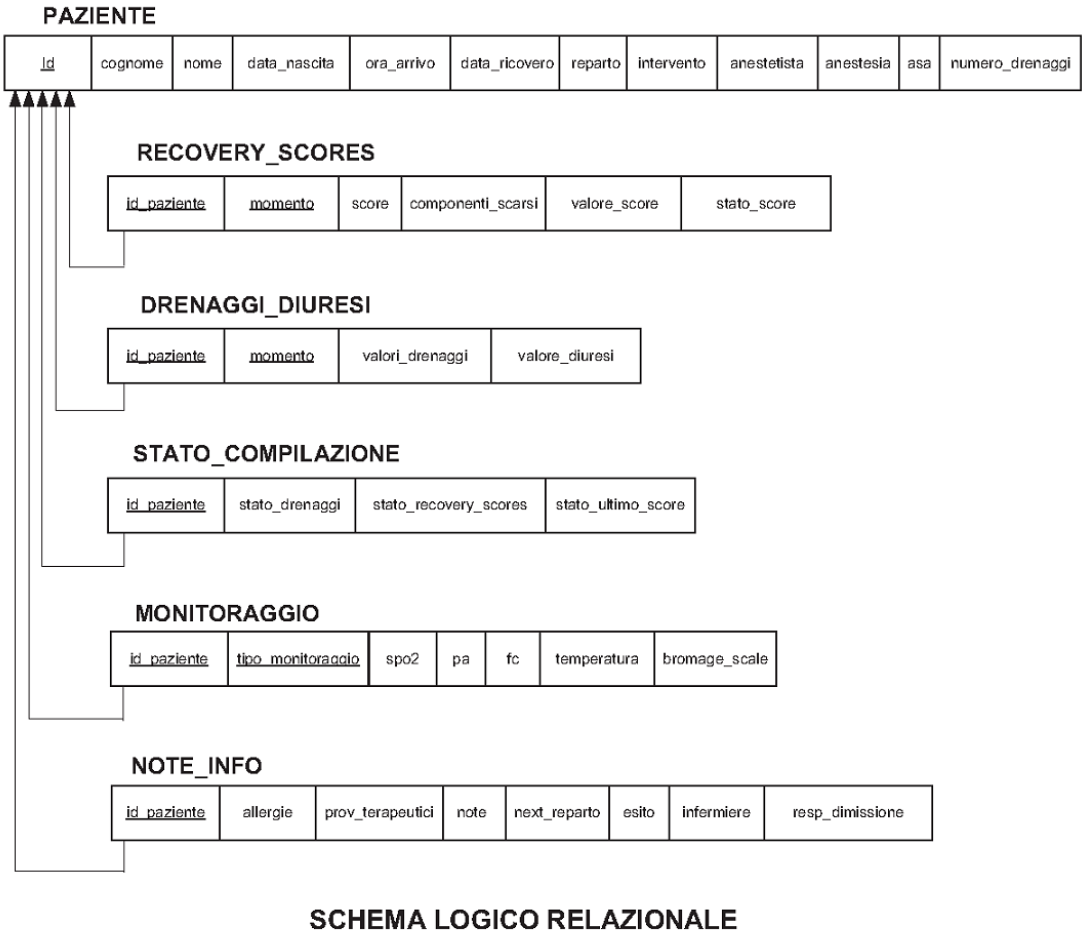


Figura 5.4: Modello relazionale del database

5.3.3 Modello relazionale

Come detto nel paragrafo precedente, il passaggio dallo schema ER al modello relazionale non ha portato molte modifiche. Le relazioni sono quindi le seguenti:

- paziente
- monitoraggio
- recovery_scores
- note_info
- stato_compilazione
- drenaggi_diuresi

Lo schema relazionale è riportato nella figura 5.4.

5.3.4 Dal modello relazionale allo schema fisico

In questo paragrafo sono riportate le modifiche fatte al momento della traduzione in codice *sql*. Esse riguardano fondamentalmente scelte volte a semplificare il codice sql e rendere più immediate le query. Nonostante l'insieme degli attributi definiti come chiave per le tabelle, si è deciso di utilizzare come chiave primaria un singolo attributo di tipo intero, chiamato semplicemente "_id". Questa scelta è dovuta soprattutto al fatto che per il corretto funzionamento del database in Android ogni tabella deve avere un campo "_id" per identificare univocamente una tupla.

5.4 Schemi risultanti dalla progettazione

Alla fine di questa fase di progettazione, si può finalmente avere un'idea generale dell'architettura del sistema presentata nella figura 5.5, e l'organizzazione generale delle views che compongono l'interfaccia grafica, presentata nella figura 5.6.

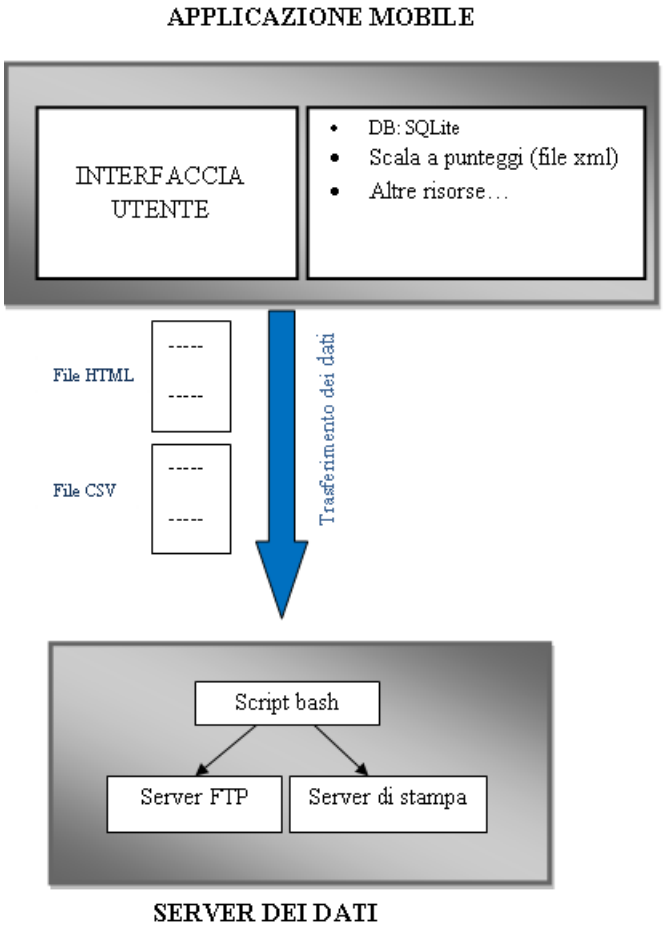


Figura 5.5: Architettura generale del sistema

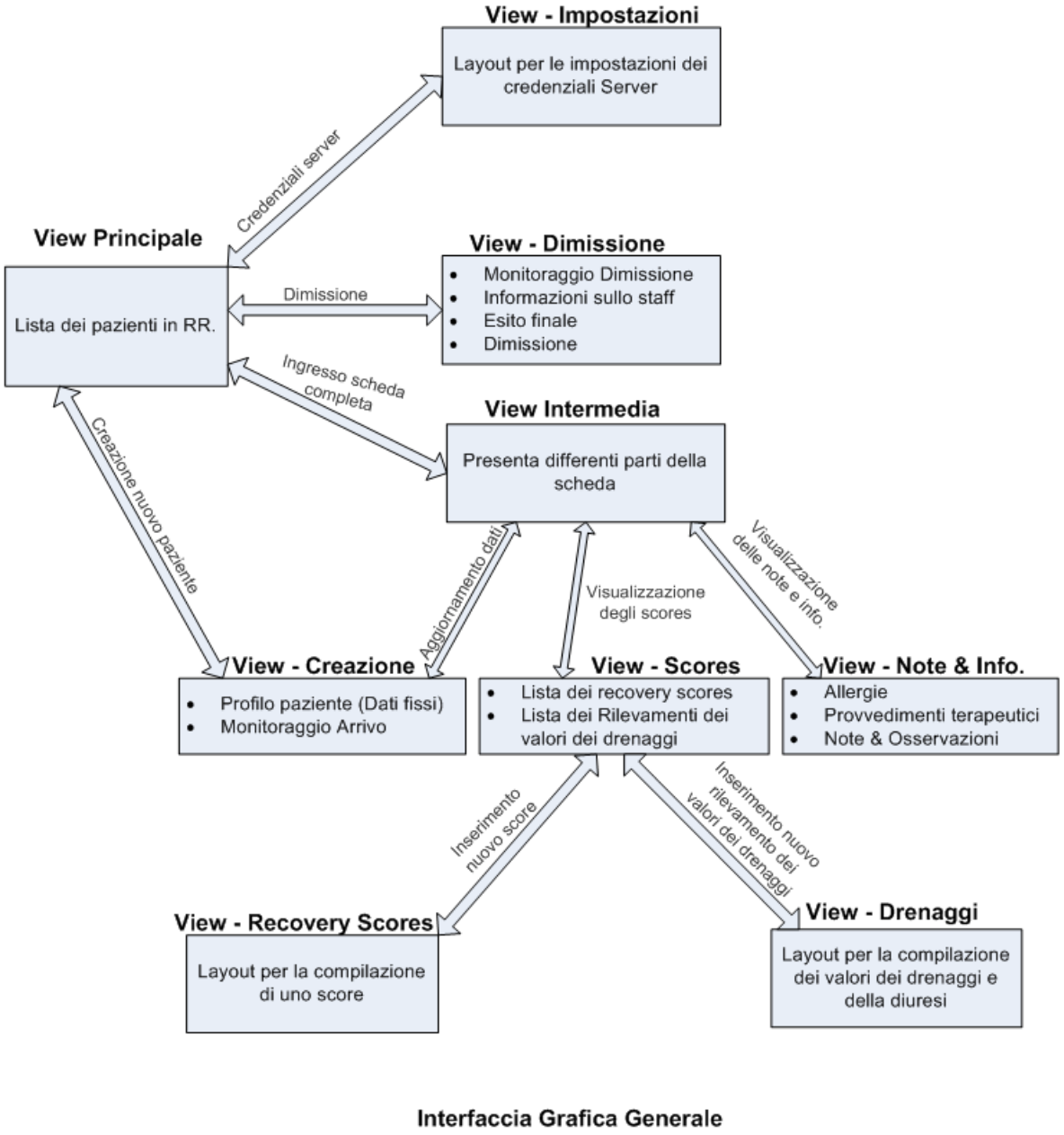


Figura 5.6: Organizzazione delle view dell'interfaccia grafica

Capitolo 6

Realizzazione

In questo capitolo sarà descritta nel dettaglio l'implementazione dell'applicazione mobile per la raccolta dei dati dei pazienti. Si inizierà dall'implementazione del database e successivamente si presenterà la realizzazione delle differenti interfacce utente.

6.1 Database

Come già detto nel capitolo precedente, la base di dati è risultata una parte fondamentale dell'applicazione. Le librerie Android mettono a disposizione una serie di classi per l'implementazione di una base di dati *SQLite* [9]. In questa sezione sarà quindi trattata la questione dell'implementazione con le API Android della base di dati la cui progettazione è stata presentata nel capitolo precedente.

6.1.1 Schema generale

La base di dati è stata implementata usando due classi principali, una chiamata `DbAdapter`, che rappresenta l'oggetto base di dati, e l'altra chiamata `DbKeys` è usata per costruire le costanti rappresentative dei nomi dei campi usati dalle classi della base di dati `DbAdapter`. Questa strategia di costruire le chiavi dei nomi dei campi in un'altra classe, permette di avere più chiarezza nella struttura della base di dati e soprattutto consente di evitare eventuali sbagli dovuti a errori di battitura dei nomi dei campi.

6.1.2 La classe DbAdapter

Questa classe rappresenta l'oggetto base di dati. Tutte le altre classi dell'applicazione che vogliono utilizzare la base di dati dovranno creare un'istanza di tale classe tramite la quale avranno poi la possibilità di effettuare delle query verso il database.
Essa contiene una classe interna chiamata `DatabaseHelper` che estende la classe `SQLiteOpenHelper` fornita dalle librerie Android per agevolare la creazione della base di dati e le sue tabelle. Inoltre, contiene una serie di metodi che sono query usate poi dalle altre classi per la consultazione e/o modifica del contenuto della base di dati.

DatabaseHelper

Questa classe contiene i seguenti metodi:

- `onCreate`: che permette di creare le tabelle fisiche della base di dati.
- `OnUpgrade`: che definisce l'azione da compiere quando si cambia la versione del database. In questo caso si è scelto di cancellare tutto il contenuto della database e ricreare di nuovo le sue tabelle.

DbAdapter

Come detto prima, questa classe ne contiene una interna chiamata `DatabaseHelper` e i seguenti metodi:

- `open`: si appoggia su un'istanza della classe `DatabaseHelper` per aprire o creare (se ancora inesistente) il database dell'applicazione in lettura e scrittura.
- `close`: chiude il database dell'applicazione.
- `createPatient`: permette di inserire un nuovo paziente nella tabella Paziente del database.
- `updatePatient`: permette di aggiornare i dati di una tupla Paziente.

- `deletePatient`: cancella tutti i dati relativi a un paziente specifico del database. Il paziente è identificato con il suo attributo id della tabella Paziente.
- `fetchAllPatients`: recupera i dati essenziali (identificativo, nome, cognome, tempo già trascorso in RR, stato ultimo score, ecc.) di tutti i pazienti presenti in database (cioè tutti i pazienti attualmente presenti in RR).
- `createNotesInfo`: inserisce una nuova tupla nella tabella note_info (entità note_info specificata nel paragrafo 5.3.1) che contiene le altre informazioni su un paziente.
- `deleteNotesInfo`: cancella le informazioni inserite con il metodo `createNotesInfo`
- `updateNotesInfo`: aggiorna una tupla della tabella note_info con le informazioni relative alle allergie del paziente e i provvedimenti terapeutici che gli vengono somministrati.
- `updateLastInfo`: aggiorna una tupla della tabella note_info con le informazioni relative allo stato finale del paziente e quelle relative allo staff che si è occupato di lui in RR.
- `fetchNotesInfo`: recupera una tupla della tabella note_info.
- `fetchLastInfo`: recupera la parte di una tupla della tabella note_info che contiene le informazioni sull'esito finale del paziente e lo staff che si è occupato di lui in RR.
- `insertScore`: inserisce un nuovo score per un paziente specifico.
- `deleteScores`: cancella dal database tutti gli scores che riguardano un paziente.
- `fetchScore`: recupera l'informazione sull'ultimo score inserito per un paziente.
- `fetchAllScores`: recupera tutte le informazioni su tutti gli scores registrati per un paziente.

- `fetchAllReducedScores`: recupera informazioni solo in parte (solo l'identificativo e l'istante in cui essi sono stati registrati), di tutti gli scores registrati per un paziente.

- `insertMonitoring`: inserisce una nuova tupla nella tabella monitoring (entità `Monitoraggio` specificata nella fase di progettazione). Durante le dimissioni un paziente dovrebbe avere due tuple in questa tabella, una relativa al monitoraggio effettuato al suo arrivo e l'altra per il monitoraggio effettuato poco prima delle dimissioni.

- `deleteMonitoring`: cancella tutti i dati del paziente relativi ai monitoraggi effettuati all'arrivo e in dimissione.

- `updateMonitoring`: aggiorna i dati relativi al monitoraggio effettuato all'arrivo e quello effettuato in dimissione su un paziente.

- `fetchmonitoring`: recupera i dati relativi al monitoraggio effettuato all'arrivo e quello effettuato in dimissione su un paziente.

- `insertDrainageValues`: inserisce un nuovo rilevamento dei valori dei drenaggi di un paziente.

- `deleteDrainageValues`: cancella tutti i rilevamenti dei valori dei drenaggi di un paziente.

- `fetchAllDrainageValues`: recupera le informazioni su tutti i rilevamenti dei valori dei drenaggi registrati per un paziente.

- `fetchAllReducedDrainageValues`: recupera le informazioni solo in parte (solo l'identificativo e l'istante in cui essi sono stati registrati), di tutti i rilevamenti di drenaggi registrati per un paziente.

- `createFillingState`: inserisce una nuova tupla nella tabella filling_state (entità stato_compilazione specificata nella fase di progettazione), che permette di tenere informazioni relative allo stato di compilazione della scheda del paziente e lo stato del suo ultimo score registrato.

- `deleteFillingState`: cancella quanto inserito con `createFillingState`.
- `updateFillingState`: aggiorna una tupla della tabella filling_state.
- `fetchFillingState`: recupera una tupla della tabella filling_state relativa a uno specifico paziente.

Tutti i metodi elencati sopra sono quindi utilizzati da altre classi per accedere al database e potervi leggere e modificare i dati relativi ai pazienti presenti in RR.
In questa sezione si sono solamente elencati i metodi, senza però descrivere la loro implementazione, cosa che faremo più avanti.

6.2 Activity principale dell'applicazione

L'Activity principale dell'applicazione contiene la prima finestra che l'utente vede all'avvio. Si è pensato di realizzare quest' intefaccia in modo che l'utente possa trovarvi direttamente le informazioni significative dei pazienti che si trovano in RR. Infatti, essa mostra la lista dei pazienti presenti in RR e per ogni paziente dà le seguenti informazioni:

- cognome e nome;
- tempo trascorso in RR;
- stato dell'ultimo score registrato (informazione data usando un codice a colori per qualificare lo stato degli scores).

Oltre alle caratteristiche citate sopra, questa view consente di accedere alle schede dei singoli pazienti e contiene un Menu che permette di inserire dei nuovi pazienti in RR, di gestire le impostazioni generali dell'applicazione e di cambiare la scala a punteggi in uso.

6.2.1 Creazione interfaccia utente

L'interfaccia grafica di questa view è costituita essenzialmente da una lista per presentare le informazioni sui pazienti presenti in RR, ragione per

cui per implementarla si è fatto uso di una classe che estende la classe `ListActivity` fornita nelle API Android. Infatti, come spiegato nel capitolo sull'introduzione al sistema Android, la creazione di un'interfaccia utente si fa estendendo la classe `Activity`. In questo caso si è esteso la classe `ListActivity` che è una sotto-classe di `Activity` implementata e fornita nelle API allo scopo di agevolare la creazione delle interfacce grafiche che presentano solo una lista di oggetti.
Il Listing 6.1 presenta il file XML di layout che si è usato per specificare i componenti dell'interfaccia. Esso è costituito da un `LinearLayout` che contiene al suo interno due elementi di View: una `ListVew` per visualizzare gli elementi della lista quando essa ha delle informazioni e una `TextView` visualizzata al posto della `ListView` quando la lista non contiene elementi. Quest'ultima fa vedere un messaggio per indicare che non ci sono elementi da visualizzare nella lista (vedere figura 6.1.a).

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:background="#8E90BC"
  android:scrollbars="vertical">

  <ListView
    android:id="@android:id/list"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
  />
  <TextView
    android:id="@android:id/empty"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:text="@string/no_patients"
    style="@style/fieldSectionText"
   />
</LinearLayout>
```

Listing 6.1: Layout view principale

6.2.2 Popolazione dell'interfaccia utente

Allo scopo di ottimizzare la gestione degli elementi della lista, si è fatto uso della classe `PatientCursorAdapter`, che estende la classe `Cursor-`

`Adapter` della libreria. Infatti per popolare la lista, una query viene eseguita tramite il metodo `fetchAllPatients` (della classe `DbAdapter` descritta precedentemente) che restituisce i dati dentro un oggetto `Cursor` il quale viene passato come parametro per la creazione dell'oggetto `PatientCursorAdapter` che si occupa di popolare la lista.
Un oggetto `Cursor` può essere visto come un contenitore di dati, strutturato come una tabella costituita da delle righe. Esso fornisce dei metodi che permettono di esplorare le sue righe andando avanti e/o indietro su di esse (*move(int offset)*, *moveToNext()*, *moveToPrevious()*, *moveToLast()*, ecc.) come nel caso di un *iterator Java*, e di accedere a un dato presente in una specifica colonna di una sua determinata riga.
La porzione di codice del Listing 6.2 realizza quello appena descritto:

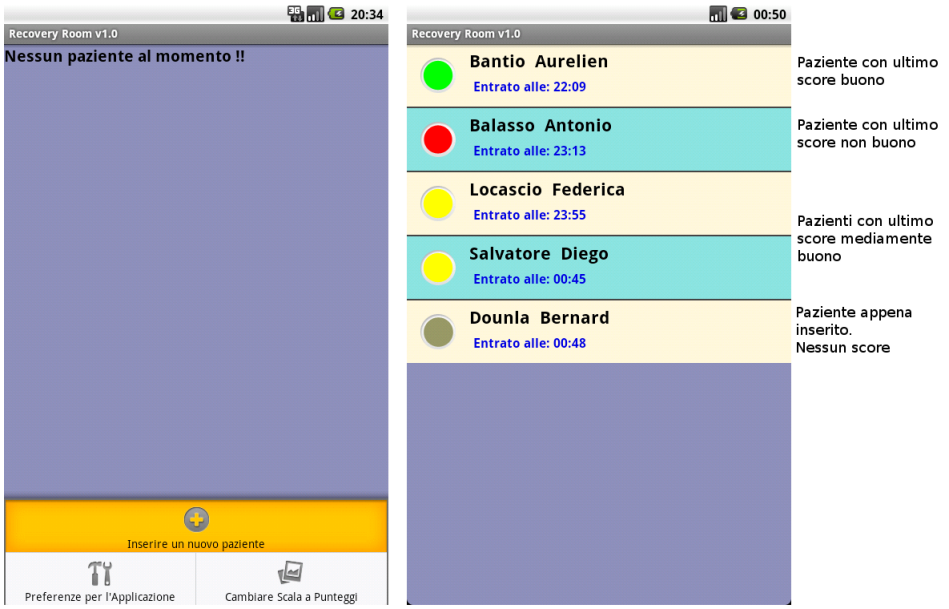
```
private void fillData() {
// Get all patients from the database and populate the ListView
      Cursor patientsCursor = mDbHelper.fetchAllPatients();
      startManagingCursor(patientsCursor);
      patientsCursor.moveToFirst();
      PatientCursorAdapter patients =
                  new PatientCursorAdapter(this, patientsCursor);
      setListAdapter(patients);
}
```

Listing 6.2: Popolazione dei pazienti nella View principale

Breve descrizione della classe *PatientCusorAdapter*

Questa classe è in effetti un `Adapter` che prende in carico il disegno di ogni riga di una `ListView`. Esistono due principali tipi di `Adapter`; quelli basati sulle liste e quelli basati sui `Cursor`. La scelta di utilizzare l'uno o l'altro dipende dalla sorgente dei dati con i quali si vuole riempire la `ListView`. Nel nostro caso i dati recuperati dal database ci vengono restituiti dentro un `Cursor` e per questo abbiamo usato l'`Adapter` basato sui `Cursor`.
I metodi di questa classe sono:

- `binView`: permette di costruire un item della `ListView` associando i dati ai componenti del layout.
- `newView`: permette di disegnare un nuovo item nella `ListView`.



a. View che presenta il suo menu

b. View che presenta 5 pazienti in sala

Figura 6.1: View principale

Il listing 6.3 presenta il layout di un item della ListView che presenta la lista dei pazienti in sala, in cui:

- l'`ImageView` è usato per caratterizzare lo stato dell'ultimo score del paziente: pallino rosso se l'ultimo score non è buono, verde se l'ultimo score è buono, giallo se l'ultimo score è mediamente buono e grigio se non è ancora stato registrato uno score per il paziente (paziente appena inserito).
- Il primo elemento `TextView` (con id text1) è usato per visualizzare il cognome e nome del paziente
- Il secondo elemento `TextView` (con id text2) è usato per visualizzare il tempo trascorso dal paziente in RR.

La figura 6.1.b presenta la view principale con pazienti in RR.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
```

```
    >
    <ImageView
      android:id="@+id/imageState"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_gravity="center_vertical"
      android:padding="15px"
    />
    <RelativeLayout
      android:layout_width="wrap_content"
      android:layout_height="fill_parent"
      android:layout_weight="1"
    >
      <TextView
        android:id="@+id/text1"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="5dip"
        android:layout_marginBottom="5dip"
        android:textStyle="bold"
        android:textSize="22sp"
        android:textColor="#000000"
      />
      <TextView
        android:id="@+id/text2"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_below="@id/text1"
        android:layout_alignLeft="@id/text1"
        android:layout_marginLeft="5dip"
        android:layout_marginBottom="10dip"
        android:textStyle="bold"
        android:textSize="16sp"
        android:textColor="#0000E3"
      />
      </RelativeLayout>
</LinearLayout>
```

Listing 6.3: Layout item della ListVew dei pazienti in sala

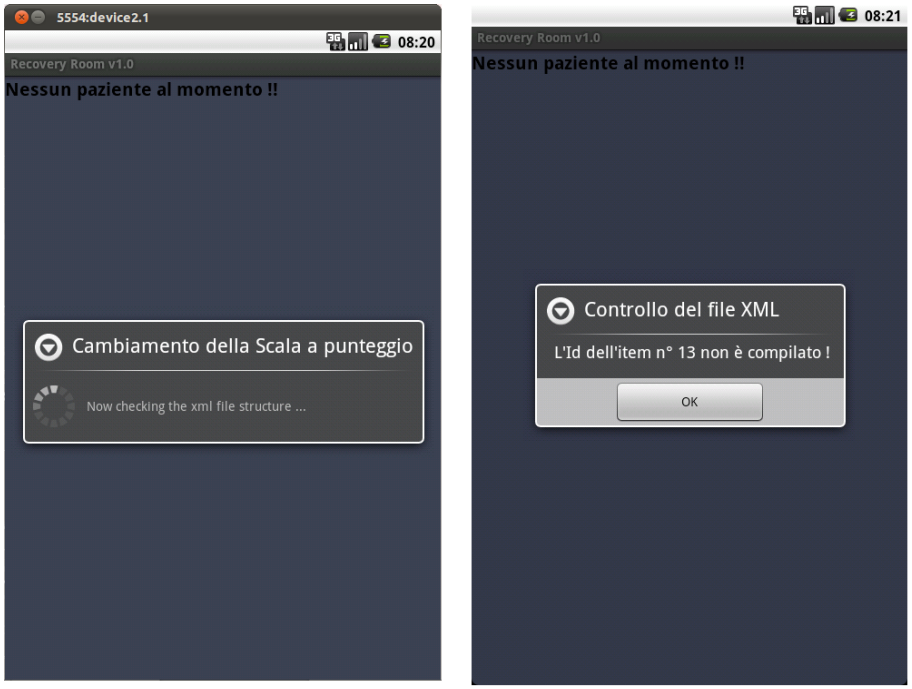
6.2.3 Altri metodi dell'Activity principale

Oltre al metodo `fillDat()` presentato sopra e il metodo `onCreate()`, l'Activity principale ha altri metodi che permettono soprattutto di percorrere e modificare la scheda dei singoli pazienti, come:

- `onCreateOptionsMenu`: usato per disegnare e visualizzare il `Menu` quando richiesto dal sistema. Infatti quest'Activity ha un `Menu` con tre opzioni: la prima per creare un nuovo paziente, la seconda per

gestire le impostazioni dell'applicazione e l'ultima per cambiare la scala a punteggi in uso.

- `onMenuItemSelected`: usato per avviare un'azione in seguito alla scelta di un'opzione del `Menu`.
- `onCreateContextMenu`: usato per disegnare e visualizzare il `Menu` contestuale quando si fa un clic lungo su un elemento della `ListView` che presenta i pazienti in sala. Infatti si è deciso di creare un Menu contestuale per gli item della `ListView` che presenta due opzioni, una per cancellare il paziente dal database e l'altra per dimettere il paziente.
- `onContextItemSelected`: usato per avviare un'azione in seguito alla scelta di un'opzione del `Menu` contestuale definito sugli item della `ListView`.
- `onListItemClick`: usato per entrare nella scheda del paziente in seguito a un clic semplice effettuato dall'utente sull'item della `ListView` che mostra un paziente.
- `checkAppData`: usato per controllare se è la prima volta che l'applicazione viene utilizzata sul dispositivo, nel qual caso vengono settati alcuni parametri dell'applicazione.
- `initializeApp`: usato per settare alcuni parametri principali dell'applicazione quando viene usato per la prima volta.
- `finish`: usato per gestire la chiusura dell'Activity. Oltre a chiamare il metodo `finish` dell'Activity, esso chiude il database.
- `changeRecoveryScoreSheet`: usato per avviare il cambiamento del file xml che serve per specificare la scala a punteggi, e gestire la barra di progressione utilizzata per informare l'utente dell'avanzamento della fase di cambiamento di questa stessa scala.
- `updateProgressDialog`: usato per aggiornare l'informazione mostrata all'utente nella barra di progressione durante la fase di cambiamento della scala a punteggi. Si tratta qui dei messaggi che sono



a. View principale durante il cambiamento della scala a punteggio in uso nella RR

b. Segnalazione di un errore durante il controllo del nuovo file xml della scala a punteggio

Figura 6.2: Cambiamento della scala a punteggio

mandati in modo asincrono alla barra di progressione mentre si procede nella fase di controllo della struttura del file xml che contiene la nuova scala a punteggi che si vuole usare.

Per il cambiamento del file xml che specifica la scala a punteggi, si è realizzata una classe di tipo Thread, interna a quest'activity principale, di cui il metodo run esegue un controllo sulla struttura del nuovo file xml che si vuole utilizzare come nuova scala a punteggi, seguendo la struttura illustrata nel paragrafo 5.2.2 del capitolo di progettazione. Se si riscontra un'errore o un'incoerenza nel nuovo file durante la fase di controllo, il processo si ferma informando l'utente dell'esito tramite una finestra di dialogo realizzata a tale scopo, che permette di specificare chiaramente l'errore riscontrato e il punto in cui si è trovata. In quasto caso si continua a usare la scala precedentemente in uso. La figura 6.2 mostra la view principale durante il cambiamento della scala.

6.3 View per l'inserimento di un nuovo paziente

L'interfaccia grafica usata per la creazione di un nuovo paziente e per la modifica dei dati fissi e i parametri relativi al monitoraggio all'arrivo del paziente è implementata con l'Activity `PatientEdit`. Essa è suddivisa in due componenti principali, uno per l'inserimento dei dati fissi del paziente e l'altro per l'inserimento dei parametri monitorati all'arrivo del paziente in RR. I componenti di questa interfaccia grafica sono stati scelti in modo da minimizzare l'uso della tastiera per la compilazione dei campi. Di seguito verranno descritte le scelte operate per ottimizzare tale aspetto e successivamente sarà presentato come i dati vengono salvati nel database.

6.3.1 Scelte dei componenti di layout per i campi da compilare

Campi fissi (Profilo Paziente)

- *Cognome e Nome*: per questi campi sono stati scelti i componenti di tipo `AutoCompleteTextView` che sono dei campi testuali che si autocompletano in funzione dei primi caratteri digitati dall'utente e dei dati presenti in dei contenitori di stringhe a esso associati. Infatti ogni elemento `AutoCompleteTextView` è associato a un contenitore di stringhe in cui esso recupera le parole che iniziano con i caratteri già digitati dall'utente e glieli propone come possibili scelte.
  Non essendo in grado di riempire i contenitori con tutti i nomi e cognomi esistenti al mondo, si è associato a tali campi un algoritmo di apprendimento dei nomi e cognomi. Sono stati usati due file come contenitori, uno per i cognomi e l'altro per i nomi. Ogni volta che viene inserito un nuovo paziente, si controlla se nome e cognome sono già presenti negli appositi contenitori e, in caso contrario, si aggiungono ai file. Ecco di seguito l'algoritmo implementato a tale scopo.

```
In fase di inserimento di un nuovo paziente
quando il campo Cognome o Nome perde il focus
se il suo contenuto non è presente nell'apposito
contenitore (lastName.txt o firstName.txt)
    setta una variabile al fine di ricordare di
```

```
salvare il suo contenuto nell'apposito conteni-
tore in fase di salvataggio dei dati compilati.
```

- *Reparto, Anestesista*: anche per questi campi si è scelto di usare i componenti `AutoCompleteTextView`. Si è deciso di utilizzare le strutture `String-Array` come contenitori di stringhe per quest'ultimi. Si è quindi presa la lista dei reparti e degli anestesisti per riempire tali contenitori (vedere figura 6.3.a). Infatti uno `String-Array` è un array di stringhe che può essere creato usando il formato XML nelle risorse di stringhe dell'applicazione in modo da poter essere poi usate dall'applicazione stessa.

- *Anestesia*: per questo campo si è scelto di usare un componente di tipo `MultiAutoCompleteTextView`. Questo differisce da `AutoCompleteTextView` per il fatto che consente di inserire più stringhe nello stesso campo, sempre in modalità auto-completamento e separa le stringhe automaticamente con una virgola. La scelta si è portata su tale componente perché c'è in effetti la possibilità che un paziente subisca più anestesie. Come nel caso precedente, si è usato la struttura `Sting-Array` come contenitore, appositamente riempito con una lista di anestesie effettuabili.

- *Intervento*: nella fase di progettazione si è pensato di usare anche un `AutoCompleteTextView` per questo campo, ma l'idea è stata poi abbandonato a causa della numerosità dei casi di interventi. Infatti il gran numero di interventi possibili non rendeva più efficiente la compilazione del campo con l'auto-completamento a causa della molteplicità delle proposte che venivano fornite all'utente.

- *Data di nascita*: per questo campo si è scelto di usare il `Widget DatePicker` associato al `Dialog DatePickerDialog` messi a disposizione nelle API. Questa rappresenta un'interfaccia ben strutturata per la compilazione di una data ed è stata scelta allo scopo di garantire la coerenza della data inserita dall'utente (vedere figura 6.3.b). Infatti è stato messo un bottone accanto a un `TextView` (dentro il quale la data viene inserita dopo la compilazione) sul quale l'utente deve

fare un clic per avere un `Dialog DatePickerDialog` in cui potrà compilare comodamente la data.

- *Numero dei drenaggi*: per questo campo si è scelto di usare uno `Spinner` che può permettere di scegliere comodamente un valore per il numero dei drenaggi. Infatti uno `Spinner` è un `Widget` dell'API che permette di realizzare una lista scorrevole. Ad essa viene associato un contenitore di elementi che userà per riempire questa lista che verrà presentata all'utente in seguito a un clic. Si è scelto di mettere nel contenitore associato ad essa i primi cinque numeri interi visto che difficilmente si arriva ad avere 3 drenaggi su un paziente.

- *ASA*: come per il campo precedente si è usato uno `Spinner` associato a un contenitore che possiede i primi cinque valori interi che sono gli unici valori possibili per il parametro ASA.

Campi relativi al monitoraggio all'arrivo

- *Ora di arrivo*: come nel caso del campo *"Data di nascita"*, si è scelto di usare il `Widget TimePicker` associato al `Dialog TimePicker-Dialog` messi a disposizione nelle API: esso rappresenta un'interfaccia ben strutturata per la compilazione dell'ora. Questo widget è stato scelto allo scopo di garantire la coerenza dell'ora inserita dall'utente (vedere figura 6.3.c). Infatti è stato messo un bottone accanto a `TextView` (dentro il quale l'ora viene inserita dopo la compilazione) sul quale l'utente deve fare un clic per avere un `Dialog TimePickerDialog` in cui potrà compilare comodamente l'ora.

- *SpO2, PA, FC, T°*: per questi campi si è scelto di usare un semplice `TextView`. Si è vincolato il TextView in modo che i caratteri inseribili siano solo numerici, visto che i valori di questi parametri sono di tale tipo.

- *Bromage Scale Arrivo*: si è scelto di usare uno `Spinner` con un contenitore che possiede i primi 4 numeri interi positivi che sono gli unici valori attribuibili a tale campo.

6.3.2 Metodi dell'Activity PatientEdit

Gestione del ciclo di vita

Nella gestione del ciclo di vita di quest'Activity si è avuta una particolare attenzione in modo da evitare la perdita di dati in fase di compilazione, cosa che avrebbe portato a dover riprendere la compilazione dei campi da zero. Di seguito si descrive ciò che realizzano i metodi implementati per la gestione del ciclo di vita di quest'Activity quando vengono chiamati dal sistema.

- *onCreate*: crea l'istanza della base di dati e componenti della view, recupera l'`Id` del paziente dall'`Intent` chiamante e riempie i campi se non si tratta di un nuovo paziente da inserire.
- *onSaveInstanceState*: salva i dati già compilati in database tramite il metodo `saveState()` e salva l'`Id` del paziente in modo da potere ricostruire lo stato attuale della View in caso di riattivazione.
- *onPause*: salva i dati già compilati in database.
- *onResume*: ripopola i campi che erano già stati compilati prima che l'Activity andasse in pausa, usando il metodo `populateFields()`.

Altri metodi

- *populateFields*: se si tratta di un nuovo paziente, esso propone l'ora corrente nel campo *"Ora"*, associa i contenitori dei dati ai campi gestiti con auto-completamento, e attiva l'auto-apprendimento dei nomi. Se si tratta invece di un paziente già esistente, esso recupera tutti i dati dei campi in database e riempie quelli che erano già stati compilati e, successivamente, crea l'associazione tra contenitori di dati e campi che si auto-completano, esclusi i campi *"Cognome"* e *"Nome"* visto che questi ultimi devono per forza essere già stati compilati altrimenti i dati del paziente non sarebbero stati salvati precedentemente.
- *saveState*: se si tratta di un nuovo paziente, salva le informazioni sul paziente in database inserendo anche i nuovi record per la gestione

della scheda e successivamente, se non sono ancora presenti nel file, salva il contenuto dei campi *"Cognome"* e *"Nome"* negli appositi file. Se si tratta invece di un paziente già esistente, aggiorna solo i record in database con lo stato attuale della compilazione.

- *initializeAutoNames*: attiva e gestisce l'auto-apprendimento dei cognomi e nomi. Implementa l'algoritmo presentato precedentemente.
- *initializeAutocompletes*: avvia i processi di auto-completamento dei campi gestiti con `AutoCompleteTextView` o `MultiAutoCompleteTextView`.
- *updateFile*: usato per l'inserimento dei nuovi cognomi o nomi negli appositi file.
- *onCreateDialog*: usato per la gestione della visualizzazione dei `Dialog`. In questo caso ci sono due `Dialog` da gestire, `DataPickerDialog` e `TimePickerDialog`.
- *setBirthDate, setTime, pad*: sono usati per gestire l'inserimento della data di nascita e dell'ora d'arrivo del paziente in seguito alla compilazione assistita dai `Dialog DatePickerDialog` e `TimePickerDialog`.

6.4 View per la gestione degli score e dei rilevamenti dei valori dei drenaggi

Questa sezione tratta della view implementata con l'Activity `PatientScores` che presenta gli scores e i rilevamenti dei valori dei drenaggi effettuati sul paziente. Essa è la porta d'ingresso alle view che permettono di inserire un nuovo rilevamento dei valori dei drenaggi e un nuovo score per il paziente.

6.4.1 Interfaccia grafica

L'interfaccia grafica di questa view contiene essenzialmente due `ListView`: una che elenca tutti gli scores e l'altra che elenca tutti i rilevamenti dei va-

a. Inserimento del nome del paziente

b. Inserimento della data di nascita

c. Inserimento dell'ora di arrivo

d. View compilata interamente

Figura 6.3: View Per l'inserimento di un nuovo paziente

lori dei drenaggi già effettuati sul paziente. Come si può vedere dal Listing 6.4 e nella figura 6.4, si è associato a ognuna delle `ListView` due `TextView`, una che fa da intestazione (`score_list_header`, `drainage_list_header`) e l'altra che permette di visualizzare un messaggio quando non ci sono elementi da visualizzare nella `ListView` (`score_emptylist`, `drainage_emptylist`).

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
      xmlns:android="http://schemas.android.com/apk/res/android"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent" >
      <TextView
              android:id="@+id/score_list_header"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:text="@string/scorelist_header"
              android:layout_marginRight="5dip" />
      <TextView
              android:id="@+id/score_emptylist"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:text="@string/score_emptylist"
              android:layout_below="@id/score_list_header"/>
      <ListView
              android:id="@+id/score_list"
              android:layout_height="wrap_content"
              android:layout_width="wrap_content"
              android:layout_below="@id/score_list_header"/>
      <TextView
              android:id="@+id/drainage_list_header"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:text="@string/drainagelist_header"
              android:layout_alignParentRight="true"
              android:layout_marginRight="5dip" />
      <TextView
              android:id="@+id/drainage_emptylist"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:text="@string/drainage_emptylist"
              android:layout_below="@id/drainage_list_header"
              android:layout_alignLeft="@id/drainage_list_header"/>
      <ListView
              android:id="@+id/drainage_list"
              android:layout_height="wrap_content"
              android:layout_width="wrap_content"
              android:layout_below="@id/drainage_list_header"
              android:layout_alignLeft="@id/drainage_list_header"/>
```

```
</RelativeLayout>
```

Listing 6.4: Layout - view degli scores

ScoreCursorAdapter

Similmente al caso della `ListView` dei pazienti presenti in RR presentata nel paragrafo 6.2.1, si è usato anche per queste `ListView` un `Adapter` (`ScoreCursorAdapter`) per gestire la visualizzazione degli elementi recuperati dal database.
Essa ha gli stessi metodi della classe `PatientCursorAdapter` ma il layout definito per ogni elemento della `ListView` è diverso. Nel Listing 6.5 viene presentato il layout per ogni elemento (item) delle `ListView`. Come ci si può vedere, ogni elemento della `ListView` è costituito da due `TextView` e un separatore. Il primo elemento `TextView` serve a dare un nome all'elemento della `ListView` (si è scelto di dare dei nomi del tipo *Score 1*, *Score 2*, ecc. oppure *Rilevamento 1*, *Rilevamento 2*, ecc.) e il secondo elemento `TextView` serve per indicare il momento in cui lo score o il rilevamento dei valori dei drenaggi è stato inserito dall'arrivo del paziente in RR.

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content">
  <TextView
      android:id="@+id/scoreText1"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_marginLeft="5dip"
      android:layout_marginRight="5dip"
      android:layout_marginBottom="5dip"
      android:textSize="18sp"
      android:textStyle="bold" />
  <TextView
      android:id="@+id/scoreText2"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_below="@id/scoreText1"
      android:layout_alignLeft="@id/scoreText1"
      android:layout_marginRight="5dip"
      android:layout_marginBottom="5dip"
      android:textStyle="bold" />
```

```
    <View
        android:layout_below="@id/scoreText2"
        android:layout_width="wrap_content"
        android:layout_alignRight="@id/scoreText2"
        android:background="#8E90BC"
        android:layout_height="2px" />
</RelativeLayout>
```

Listing 6.5: Layout di un item della ListView degli scores

6.4.2 Metodi rilevanti dell'Activity PatientScores

La gestione del ciclo di vita di quest'Activity non è stata molto rilevante per cui in questa parte ci soffermeremo solo sui seguenti metodi:

- *fillData*: usato per recuperare gli scores e i rilevamenti dei valori dei drenaggi registrati sul paziente in database e conseguentemente fornirli all'`Adapter` che si fa carico di presentarli nelle apposite `ListView`.

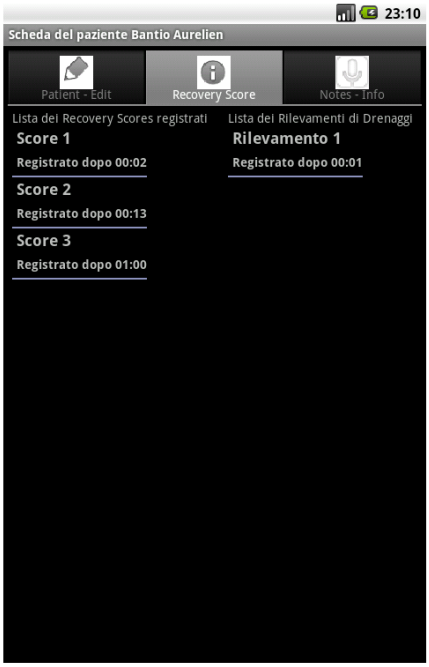
- *onCreateOptionsMenu*: usato per disegnare e visualizzare il `Menu` quando richiesto dal sistema. Infatti quest'Activity ha un `Menu` con tre opzioni: una per inserire un nuovo score, un'altra per inserire un nuovo rilevamento dei valori dei drenaggi e l'ultima per generare e mostrare all'utente, all'interno di una `WebView`, il riassunto di tutti gli scores e valori dei drenaggi rilevati sul paziente.

- *onMenuItemSelected*: usato per avviare un'azione in seguito alla scelta di un'opzione del `Menu`.

- *createScore*: questo metodo serve per avviare l'Activity `EditScores` che permette di inserire un nuovo score per il paziente. Infatti, di seguito alla scelta dell'opzione *"Crea nuovo Score"*, viene creato un Intent nella quale si mettono l'`Id` del paziente e il numero di scores già registrati sul sul paziente e successivamente tramite l'`Intent` creato si avvia l'Activity `EditScores`. Si può vedere il pezzo di codice di questo metodo nel Listing 6.6.

- *createDrainagesCheckUp*: questo metodo serve per avviare l'Activity `EditDrainagesCheckUp` utilizzato per inserire un nuovo rileva-

a. View che presenta il menu e le liste vuote degli score e dei rilevamenti dei valori dei drenaggi



b. View che presenta gli score e i rilevamenti dei valori dei drenaggi già registrati

c. Visualizzazione dello stato degli score (parte 1)

d. Visualizzazione dello stato degli score (parte 2)

Figura 6.4: View Per la gestione degli score e rilevamenti dei valori dei drenaggi

mento dei valori dei drenaggi per il paziente. I passi eseguiti sono gli stessi del metodo precedente.

- *onActivityResult*: questo metodo viene chiamato quando una tra le due Activity `EditScores` e `EditDrainagesCheckUp` termina. Infatti queste Activity essendo invocate in modo asincrono dall'Activity `PatientScores`, al termine della loro esecuzione devono restituire un risultato all'Activity invocante (in questo caso `PatientScores`), che viene recuperato tramite il metodo `onActivityResult`. In questo caso, quando si finisce di editare un nuovo score o un nuovo rilevamento dei valori dei drenaggi, si ridisegnano semplicemente le `ListView`, aggiornandole alla nuova situazione dei dati che si trovano nel database.

- *getScoresState*: questo metodo permette di costruire una stringa strutturata in HTML che rappresenta il riassunto dell'evoluzione dello stato degli scores e dei valori dei drenaggi del paziente dall'arrivo in RR. Questa rappresentazione è costruita immediatamente dai dati recuperati in database. Essa è composta da due tabelle: una che presenta la scheda degli scores e l'altra che presenta la scheda dei valori dei drenaggi rilevati.

- *showScoresState*: questo metodo permette di costruire e presentare all'utente un `Dialog` contenente una `WebView` dentro la quale viene caricata la stringa creata col metodo precedente (vedere figura 6.4).

```
private void createScore() {
      Intent i = new Intent(this, EditScores.class);
      Bundle bundle = new Bundle();
      bundle.putLong(DbKeys.PATIENTID, patientId);
      bundle.putInt(DbKeys.NUMSCORES, numScores);
      i.putExtras(bundle);
      startActivityForResult(i, SCORE_CREATE);
}
```

Listing 6.6: Avvio view per inserire un nuovo score

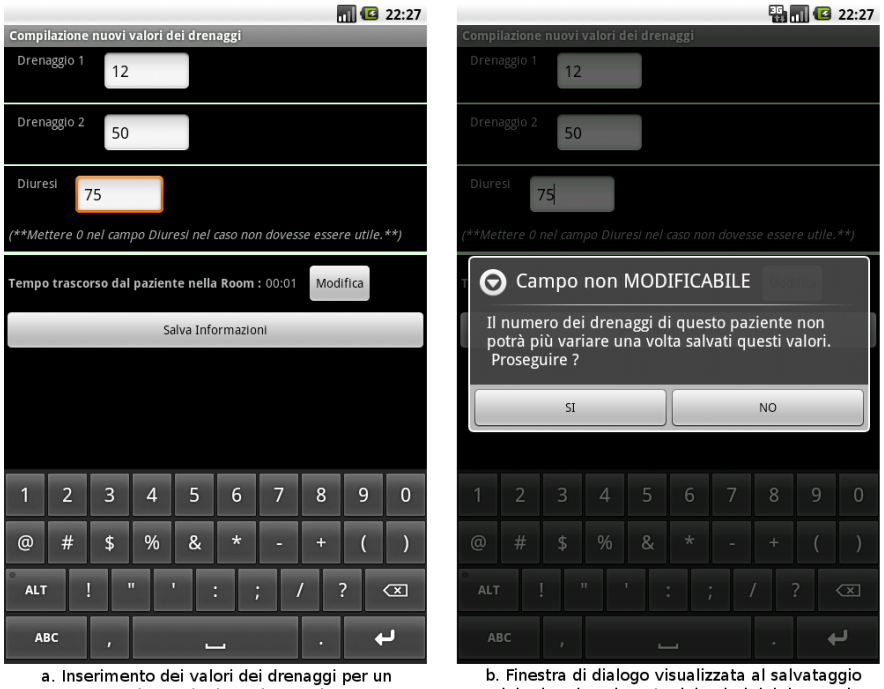
6.5 View per la compilazione di un nuovo rilevamento dei valori dei drenaggi

In questa parte verrà presentata l'Activity `EditDrainagesCheckUp` che è necessaria per la compilazione dei nuovi rilevamenti dei valori dei drenaggi. Si insisterà sulla costruzione della sua interfaccia grafica e successivamente si presenteranno i metodi rilevanti concentrandosi su quelli usati per salvare i dati nel database.

6.5.1 Interfaccia grafica

Diversamente dalle interfacce delle view presentate finora, l'interfaccia grafica di questa view non è stata costruita usando l'XML per specificare i componenti del layout, ma si è usato piuttosto una costruzione dinamica. Questa scelta è giustifica dal fatto che, visto che il numero dei drenaggi non è uguale per tutti i pazienti, non si poteva costruire un layout statico utilizzabile per inserire i valori dei drenaggi per tutti.
Un'altra soluzione meno elegante poteva essere quella di costruire un layout statico con un numero di campi elevato (ad esempio 6) per la compilazione dei valori dei drenaggi, supponendo che i pazienti non arrivino mai a un tale numero di drenaggi; in fase di compilazione l'utente avrebbe compilato solo un numero di campi corrispondente al numero dei drenaggi effettivamente presenti sul paziente. Questa soluzione è stata scartata perché è meno intuitiva per l'utente.
Per realizzare l'interfaccia, si è creata una nuova classe chiamata `GuiDrainField` per rappresentare i campi dei drenaggi da compilare. Si tratta graficamente di una barra che contiene due elementi di view al suo interno: una `TextView` per indicare il nome del campo (*Drenaggio 1*, *Drenaggio 2*, ecc.) accanto al quale c'è una `EditText` in cui l'utente deve compilare il valore rilevato per il drenaggio corrispondente. Ecco di seguito i passi eseguiti per la generazione dinamica dell'interfaccia grafica di questa view.

1. Per ogni drenaggio, inserire un'istanza di GuiDrainField nella view e aggiungere in seguito all'oggetto GuiDrainField appena inserito un separatore (istanza della classe GuiSeparator). I nomi dei campi so-



a. Inserimento dei valori dei drenaggi per un paziente che ha 2 drenaggi

b. Finestra di dialogo visualizzata al salvataggio del primo inserimento dei valori dei drenaggi

Figura 6.5: View Per l'inserimento dei valori dei drenaggi

no incrementati ogni volta, cioè i campi avranno nome Drenaggio 1, Drenaggio 2, ecc.

2. Aggiungere nella view un altro oggetto GuiDrainField, questa volta con "Diuresi" come nome del campo. Quest'ultimo sarà il campo in cui compilare il valore della diuresi del paziente.

3. Aggiungere un separatore con spessore più grande per separare i campi dei drenaggi e diuresi da quello che viene dopo.

4. Aggiungere una barra per l'inserimento del tempo trascorso dal paziente in RR.

5. Aggiungere un bottone con il quale l'utente deve salvare i dati compilati.

La figura 6.5.a presenta la view mentre si compilano i valori dei drenaggi per un paziente che ne ha due.

6.5.2 Metodi rilevanti dell'Activity EditDrainagesCheckUp

I metodi descritti di seguito sono quelli che sono stati rilevanti nell'Activity al fine della compilazione dei valori dei drenaggi e del loro salvataggio in database.

- *initializeForm*: in questo metodo si crea l'interfaccia grafica della view seguendo i passi elencati precedentemente. Dopo aver costruito i campi, si presenta il tempo trascorso dal paziente nella RR all'utente calcolando la differenza tra data di arrivo e data corrente; quel tempo proposto è modificabile tramite gli stessi strumenti utilizzati per compilare l'ora d'arrivo nell'Activity `PatientEdit`.

- *saveState*: con questo metodo si recuperano i valori inseriti nei campi per salvarli nel database e si termina l'Activity ritornando all'Activity `PatientScores` che è l'unica in grado di avviare l'Activity EditDrainagesCheckUp.
  I valori dei drenaggi sono salvati nel database in un unico campo testuale della tabella *"drainages"*. Non si poteva immaginare di salvare il valore di ogni drenaggio in un unico campo in quanto non tutti i pazienti hanno un numero fisso di drenaggi. Come si può vedere nel Listing 6.7, si costruisce il valore del campo facendo una concatenazione dei valori di tutti i drenaggi, separando gli uni degli altri con il carattere **"|"**.

- *checkState*: metodo usato per controllare che tutti i campi della view siano stati compilati. Nel caso non dovessero essere tutti completi, non si potrà provvedere al salvataggio dei dati compilati in database.

- *generateError*: metodo usato per creare un `Dialog` in caso di errore di compilazione da parte dell'utente. Lo scopo del `Dialog` è di descrivere chiaramente la causa del problema all'utente.

- *onCreateDialog*: usato per avviare la visualizzazione dei Dialog di quest'Activity. Ci sono due Dialog da gestire in questo caso; una è quella descritta sopra che viene visualizzata in caso di errore di compilazione dei campi e l'altra è un Dialog che avvisa l'utente che una volta

salvato il primo rilevamento dei valori dei drenaggi, non potrà più modificarlo (vedere figura 6.5.b).

```
private void saveState() {
      String timeInfo = mTimeDisplay.getText().toString();
      StringBuilder drainageValues = new StringBuilder();
      for(int i = 0; i < drainNumber; i++) {
              drainageValues.append(fields[i].getValue()).append("|");
      }

      diuresisValue = diuresisField.getValue();
      mDbHelper.open();
      mDbHelper.insertDrainageValues(patientId,
                  drainageValues.toString(), diuresisValue, timeInfo);
      // Update the filling state to remember that there are at least
      //two drainCheck's rows inserted for this patient.
      if(numDrainCheck == 1) mDbHelper.updateFillingState(patientId,
                                                  Option.DRAINAGE, 1);
      mDbHelper.close();

      setResult(RESULT_OK);
      finish();
}
```

Listing 6.7: Salvataggio dei valori dei drenaggi e diuresi

6.6 View per l'inserimento di un nuovo recovery score

In questa parte verrà presentata l'Activity `EditScores` che serve per la compilazione degli scores di un paziente. Si insisterà sulla costruzione della sua interfaccia grafica e successivamente ci si concentrerà sui metodi usati per salvare i dati in database.

6.6.1 Interfaccia grafica

Come per la view presentata precedentemente, l'interfaccia grafica di questa view non è stata implementa usando l'XML: si è scelta un'interfaccia che viene creata dinamicamente con il codice. Questa scelta è giustificata dal fatto che la scala a punteggi utilizzata per specificare gli scores non è fissa come già detto.
Prima di entrare nei dettagli implementativi, è opportuno ricordare come è

fatta la scala a punteggi o scheda dei recovery score: essa contiene fondamentalmente due tipi di elementi, che nel paragrafo 5.2.1 abbiamo chiamato rispettivamente *"parametri principali"* e *"stati rappresentativi"*. Ogni parametro principale ha degli stati rappresentativi che caratterizzano i suoi possibili stati; dare uno score a un paziente significa scegliere, per ogni parametro principale della scheda, quello stato rappresentativo che meglio lo rappresenta al momento dell'attribuzione dello score.
Gli obiettivi da perseguire nella costruzione dell'interfaccia grafica di questa view erano quindi due: si doveva trovare un modo per presentare in maniera leggibile la scala a punteggi all'utente nello schermo a disposizione, e si doveva fornire all'utente un modo di compilazione degli scores il più semplice e veloce possibile. Tali obiettivi sono stati raggiunti con la creazione delle seguenti classi.

GuiOption

Questa non è altro che una classe derivata della classe `RadioButton` presente nelle API Android. Essa rappresenta in effetti un bottone radio che possiede un valore (numero intero associato a uno stato rappresentativo di un parametro principale) e un testo rappresentativo (descrizione associata a un parametro rappresentativo). Si capisce che le istanze di questa classe sono usate per rappresentare gli stati rappresentativi dei parametri principali.
Oltre al suo costruttore, la classe contiene due metodi che restituiscono rispettivamente il valore e la stringa descrittiva associati al parametro rappresentativo.

GuiParentField

Questa classe è una classe derivata di `LinearLayout` presente nelle API Android. Essa è costituita da due parti principali: la prima parte è anch'essa un `LinearLayout` che contiene principalmente un' immagine e un testo che rappresenta la descrizione di un parametro principale; la seconda parte è ugualmente un `LinearLayout` che contiene un `RadioGroup` costituito da un insieme di `RadioButton` (istanze di `GuiOption`). Si è fatto

in modo che la seconda parte cambi lo stato da visibile a invisibile o da invisibile a visibile in seguito a un'azione di clic dell'utente sulla prima parte, e che cambi anche da visibile a invisibile di seguito alla scelta di un'opzione (selezione di uno tra i bottoni radio) tra quelle presenti nella seconda parte. L'immagine della prima parte ha due stati diversi in funzione della visibilità o no della seconda parte. Tutto questo permette di simulare l'effetto conosciuto dell'albero espandibile e collassabile.
Si è dunque usata questa classe per rappresentare un parametro principale e i suoi stati rappresentativi.
I metodi rilevanti di questa classe sono i seguenti:

- *isFilled*: usato per controllare se uno stato rappresentativo è stato scelto per il parametro principale. In pratica, verifica se è stato scelto un elemento nel gruppo dei bottoni radio.

- *toString*: costruisce una rappresentazione testuale e strutturata dello stato del parametro quando è già compilato. In pratica restituisce una stringa che è la concatenazione dell'`Id` del parametro principale, del carattere **","**, della posizione dello stato rappresentativo scelto tra tutti i possibili casi, e del carattere **"|"**. Per esempio la stringa **"1,0|"** è la rappresentazione dello stato del parametro principale che ha id 1 e di cui lo stato rappresentativo scelto è il primo (posizione 0).

- *updateValue*: in seguito alla scelta di uno stato rappresentativo o al cambiamento dello stato rappresentativo del parametro principale, questo metodo aggiorna il valore del parametro principale al valore associato allo stato rappresentativo corrente. Esso cambia anche il background della prima parte della view in base allo stato rappresentativo corrente. Infatti si è scelto di usare quattro colori di background per la prima parte. La tabella 6.1 mostra i colori di background usati per la prima parte in funzione dello stato del parametro.

- *getScore*: restituisce il valore intero associato allo stato rappresentativo corrente del parametro principale.

- *notifyChangeValue*: usato per informare la view master (che contiene tutta la scheda) del cambiamento dello stato rappresentativo di un pa-

rametro principale. La *view master* è la view nella quale sono inseriti le istanze della classe `GuiParentField` al fine di costituire l'intera scheda degli scores. Essa contiene il valore generale dello score che è la somma di tutti i valori associati ai parametri principali.

- *collapseChildren*: usata per rendere invisibile la seconda parte della view, cioè quella che contiene gli stati rappresentativi.
- *expandChildren*: usata per rendere visibile la seconda parte della view.
- *getState*: usata per verificare se si è scelto uno stato rappresentativo al parametro principale.
- *isOk*: usata per verificare se il parametro principale ha uno stato buono, cioè se il suo stato rappresentativo è il primo dei possibili stati.

Colore	**Descrizione**	**Stato rappresentativo scelto**
Verde	Stato buono	primo stato rappresentativo
Giallo	Stato medio	diverso dal primo e dall'ultimo
Rosso	Stato non buono	ultimo stato rappresentativo
Nero	Stato non definito	nessuna scelta al momento

Tabella 6.1: Colori di background associati agli scores

GuiMasterView

Questa classe è una sotto-classe di *LinearLayout*. Essa è stata progettata per rappresentare la scheda degli scores e contiene tutti i parametri principali della scala a punteggi; di conseguenza è costituita dalle istanze della classe `GuiParentField` che rappresentano tali parametri principali insieme ai loro stati rappresentativi. Oltre agli oggetti `GuiParentField`, essa contiene una barra per lo score totale che viene aggiornata ogni volta che lo stato rappresentativo di un parametro principale viene cambiato dall'utente.
I metodi rilevanti di questa classe sono i seguenti:

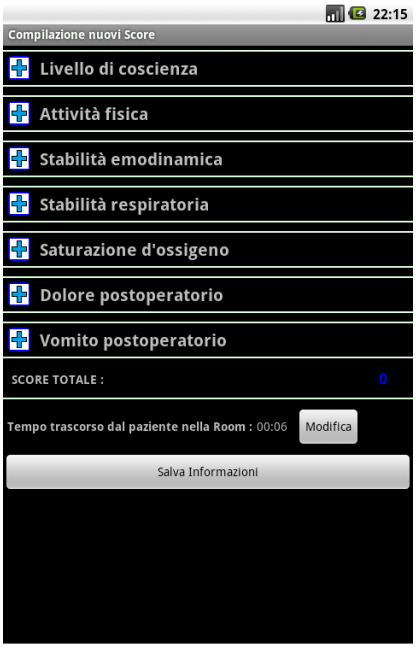
- *addField*: permette di aggiungere un parametro principale alla scheda (ciò significa aggiungere un'istanza di *GuiParentField* alla view).

- *getTotalScore*: restituisce lo score totale corrente dell'intera scheda.
- *updateTotalScore*: aggiorna lo score totale in seguito a un cambiamento dello stato rappresentativo di un parametro principale visto che cambiando tale stato il nuovo valore associato al parametro principale diventa quello associato al suo stato rappresentativo corrente.
- *getFields*: restituisce la lista di tutti i parametri principali contenuti nella scheda (cioè la lista delle istanze di `GuiParentField` che sono stati aggiunti alla view).
- *addScoreBar*: aggiunge la barra dello score totale alla view.

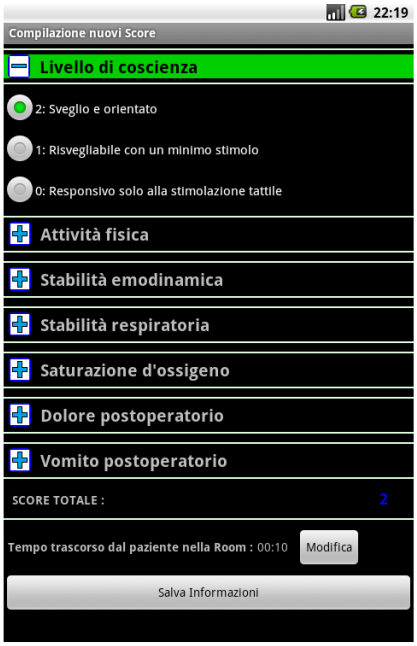
I passi per la costruzione di questa interfaccia grafica sono presentati di seguito.

1. Creare la view master, istanza della classe GuiMasterView per rappresentare l'intera scheda degli scores.
2. Per ogni parametro principale
    - Costruire il RadioGroup di cui opzioni sono le istanze di GuiOption che rappresentano gli stati rappresentativi del parametro principale in questione.
    - Costruire un'istanza della classe GuiParentField che contiene il RadioGroup costruito precedentemente. Questo per rappresentare un parametro principale e i suoi stati rappresentativi.
      Nel caso in cui non sia il primo score compilato per il paziente, si precompila questo parametro principale se era già buono nello score precedente.
    - Aggiungere l'istanza di GuiParentField appena costruita alla view master.
3. Aggiungere una barra per l'inserimento del tempo trascorso dal paziente in RR.
4. Aggiungere un bottone con il quale l'utente deve salvare i dati compilati.

La figura 6.6 presenta qualche screenshot di questa view.

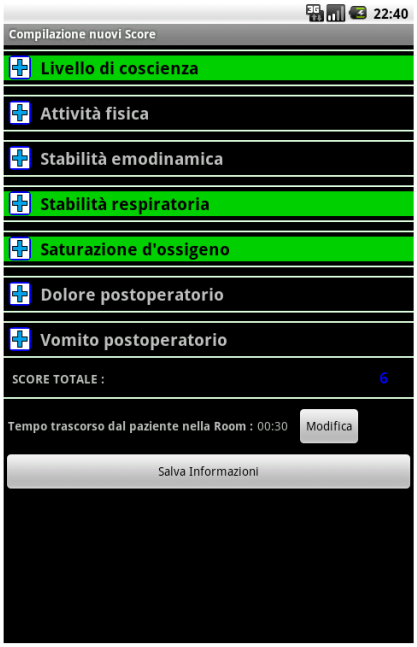


a. Scheda di un nuovo score non ancora compilato



b. Livello di coscienza compilato per il nuovo score

c. Nuovo score interamente compilato



d. Compilazione di un nuovo score partendo dalle informazioni dell'ultimo score del paziente

Figura 6.6: View Per l'inserimento di un nuovo score

6.6.2 Metodi rilevanti dell'Activity EditScores

I metodi descritti di seguito sono quelli che sono stati rilevanti nell'Activity `EditScores` al fine della compilazione degli scores e al loro salvataggio in database:

- *displayForm*: permette di costruire l'interfaccia grafica seguendo i passi descritti precedentemente. Le informazioni sulla scala a punteggi sono recuperate facendo un parsing del file XML che la rappresenta. Il Listing 6.8 presenta il codice con il quale si recuperano quelle informazioni.

- *buildMap*: usato quando lo score che si sta compilando non è il primo per il paziente. Permette di costruire la struttura dello score precedente a quest'ultimo partendo dalle informazioni su tale score recuperate nel database, al fine di proporre una scheda all'utente con alcuni parametri principali (quelli che erano buoni nell'ultimo score registrato) essagià precompilati.

- *isScarce*: permette di controllare se un parametro principale non era buono nell'ultimo score registrato sul paziente. In tal caso questo parametro non verrà precompilato nella nuova scheda.

- *setTime*: permette di calcolare e proporre all'utente il tempo già trascorso dal paziente in RR.

- *onCreateDialog*: permette di avviare la visualizzazione di una tra le finestre di dialogo di questa view, che in realtà ne contiene due, una per aiutare l'utente a compilare il tempo trascorso e l'altra per segnalare eventuali errori di compilazione.

- *saveState*: permette di costruire una struttura dell'intera scheda compilata e successivamente di salvarla in database. Si è scelto di salvare la struttura dello score dentro un unico campo della tabella *"recovery_scores"*; questo è dovuto sempre al fatto che la scala a punteggi non è fissa. In pratica si costruiscono le strutturate testuali di tutti i parametri principali, come spiegato precedentemente nella classe

`GuiParentField` e si concatenano; successivamente si salva il risultato di quella concatenazione come score nella tabella. Viene anche salvato lo stato rappresentativo dell'intero score. La tabella 6.2 mostra i possibili stati rappresentativi attribuibili a un intero score del paziente. Il Listing 6.9 presenta il codice di questo metodo.

Stato rappresentativo	**Descrizione**
BUONO	Tutti i parametri principali hanno stato buono
MEDIO	Nessun parametro principale ha stato inaccettabile ma non tutti hanno stato buono
NON BUONO	Almeno un parametro principale ha stato non buono

Tabella 6.2: Stati rappresentativi di un intero score

```
[...]
InputStream file = getAssets().open("recoveryScoreFields.xml",
                                              Context.MODE_PRIVATE);
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder db = factory.newDocumentBuilder();
Document dom = db.parse(file);
Element  root = dom.getDocumentElement();
NodeList fields = root.getElementsByTagName("item");
if (fields.getLength() < 1) {
  //Log.i(TAG, "---The file is empty !!");
  return false;
}

int i = 0;
if(numScores < 1) {
    while(i < fields.getLength()) {
    Node fieldNode = fields.item(i);
    NamedNodeMap attr = fieldNode.getAttributes();
    if(attr.getNamedItem("type").getNodeValue()
                                .equalsIgnoreCase("parentField")) {
        String label = attr.getNamedItem("label").getNodeValue();
        int id = Integer.parseInt(attr.getNamedItem("id").getNodeValue());
        Vector<String> fieldsLabels = new Vector<String>();
        Vector<Integer> fieldsValues = new Vector<Integer>();
        i++; // Increment i.
        while(fields.item(i).getAttributes().getNamedItem("type")
                             .getNodeValue().equalsIgnoreCase("field")) {
            Node chilFieldNode = fields.item(i);
            NamedNodeMap childAttr = chilFieldNode.getAttributes();
            fieldsLabels.add(childAttr.getNamedItem("label").getNodeValue());
            fieldsValues.add(new Integer(childAttr.getNamedItem("value")
```

```
                                                                .getNodeValue()));

            if(++i == fields.getLength()) break;
          }
          master.addField(new GuiParentField(this, fieldsLabels,
                                          fieldsValues, label, id));
        }
      }

  } else {
   while(i < fields.getLength()) {
   Node fieldNode = fields.item(i);
   NamedNodeMap attr = fieldNode.getAttributes();
   if(attr.getNamedItem("type").getNodeValue()
                                    .equalsIgnoreCase("parentField")) {
       String label = attr.getNamedItem("label").getNodeValue();
       int id = Integer.parseInt(attr.getNamedItem("id").getNodeValue());
       Vector<String> fieldsLabels = new Vector<String>();
       Vector<Integer> fieldsValues = new Vector<Integer>();
       i++; // Increment i.
       while(fields.item(i).getAttributes().getNamedItem("type")
                            .getNodeValue().equalsIgnoreCase("field")) {
           Node chilFieldNode = fields.item(i);
           NamedNodeMap childAttr = chilFieldNode.getAttributes();
           fieldsLabels.add(childAttr.getNamedItem("label").getNodeValue());
           fieldsValues.add(new Integer(childAttr.getNamedItem("value")
                                                    .getNodeValue()));
           if(++i == fields.getLength()) break;
       }
       if(isScarce(""+id)) master.addField(new GuiParentField(this,
                              fieldsLabels,fieldsValues, label, id));
       else master.addField(new GuiParentField(this, fieldsLabels,
                                       fieldsValues, label, id, 0));
     }
   }
  }
master.addScoreBar();
[...]
```

Listing 6.8: Parsing xml - costruzione view degli scores

```
private boolean saveState() {
  StringBuilder fullScore = new StringBuilder();
  StringBuilder scarceFields = new StringBuilder();
  String timeInfo = mTimeDisplay.getText().toString();
  ArrayList<GuiParentField> mGuiFields = master.getFields();
  Iterator<GuiParentField> iter = mGuiFields.iterator();
  boolean good = true;
  boolean bad = false;
  while(iter.hasNext()) {
```

```
        GuiParentField parentField = iter.next();
        State state = parentField.getState();
        if(state != State.UNDEFINED) {
            if(good && state != State.GOOD) good =false;
            if(state == State.BAD && !bad) bad = true;
            fullScore.append(parentField.toString());
            if(state != State.GOOD)
                scarceFields.append(parentField.getId()).append(",");
          }
          else {
            generateError(parentField.getLabel());
            return false;
          }
    }
    int scoreState;
    if(good) scoreState = 0;
    else scoreState = (bad)? 2 : 1;

    mDbHelper.open();
    mDbHelper.insertScore(patientId, fullScore.toString(),
        scarceFields.toString(), master.getTotalScore(), scoreState, timeInfo);
    if(numScores > 0) {
    // Update the filling state to remember there are at least two score's rows
    //inserted for this patient.
     if(numScores == 1 && previousScoreState != scoreState)
      mDbHelper.updateFillingState(patientId, Option.SCOREANDSTATE, scoreState);
      else {
        if(numScores == 1)
                mDbHelper.updateFillingState(patientId, Option.SCORE, 1);
        if(previousScoreState != scoreState)
            mDbHelper.updateFillingState(patientId, Option.STATE, scoreState);
        }

      } else {
            // Just update the last_score_state
            mDbHelper.updateFillingState(patientId, Option.STATE, scoreState);
      }
      mDbHelper.close();
      return true;
}
```

Listing 6.9: Salvataggio di uno score in database

6.7 View per l'inserimento delle allergie, provvedimenti terapeutici e altre informazioni

In questa parte verrà presentata l'Activity `PatientNotes` che serve per la compilazione delle informazioni sulle allergie, sui provvedimenti terapeutici somministrati al paziente e tutte le altre informazioni aggiuntive e osservazioni sul paziente. Si insisterà sulla costruzione della sua interfaccia grafica e successivamente si presenteranno i suoi metodi rilevanti.

6.7.1 Interfaccia grafica

L'interfaccia grafica di questa view è stata implementata usando l'XML per specificare i componenti del layout. Essa è suddivisa in tre parti: la prima parte serve per l'inserimento delle allergie del paziente, la seconda parte serve per la compilazione dei provvedimenti terapeutici somministrati al paziente e l'ultima serve per potere aggiungere qualsiasi tipo di informazione o osservazione sul paziente. Ogni parte di questa interfaccia è stata progettata diversamente rispetto alle altre. Nel seguito verrà presentata ogni parte separatamente al fine di giustificare meglio le scelte implementative.
La figura 6.7.a presenta l'interfaccia grafica appena descritta.

Inserimento delle allergie

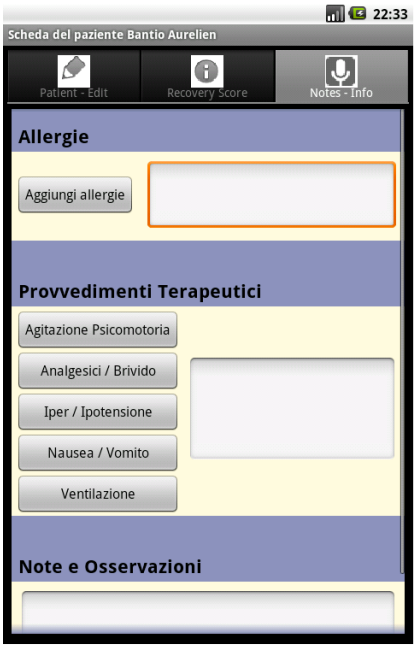
La parte per l'inserimento delle allergie è costituita da un bottone e una `EditText` (campo testuale). Volendo ridurre l'uso della tastiera o addirittura fare in modo che l'utente non abbia bisogna di digitare caratteri in questa parte, si è pensato di creare una finestra di dialogo contente una lista dei tipi di allergie possibili. Quando l'utente fa un clic sul bottone, viene visualizzata la finestra di dialogo in cui egli può selezionare le allergie del paziente. Dopo aver salvato la selezione effettuata, i nomi delle allergie selezionate vengono inseriti automaticamente nella `EditText` corrispondente alle allergie. Si è pensato di realizzare la finestra di dialogo in modo che la lista dei tipi di allergie possa essere cambiata. La finestra di dialogo è stata realizzata con la classe `AllergiesDialog`, che ha i seguenti metodi:

- *onClick*: per gestire la selezione fatta dall'utente. Infatti ogni riga della lista di scelte presenta una `CheckBox` e il nome dell'allergia. Quando l'utente fa un clic su un'allergia (cioè seleziona un'allergia), la `CheckBox` corrispondente all'allergia selezionata viene segnata.
- *getResult*: permette di costruire il risultato della selezione effettuata dall'utente e successivamente inserirla nella `EditText` delle allergie. Il risultato è la concatenazione delle stringhe corrispondenti alle allergie scelte in cui le une sono separate dalle altre con una virgola.

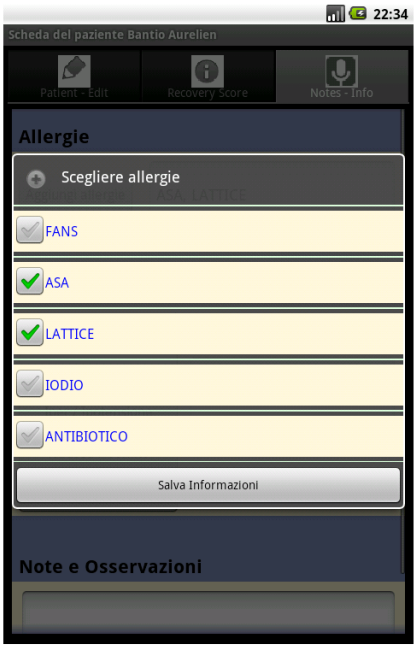
La figura 6.7.b mostra l'inserimento delle allergie.

Compilazione dei provvedimenti terapeutici

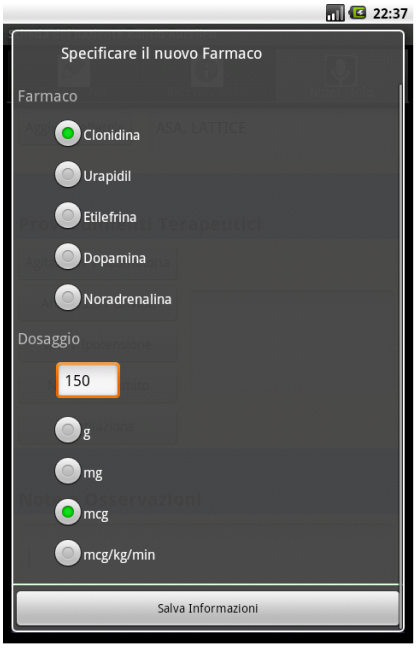
Questa parte è costituita da un gruppo di bottoni e un `EditText`. Sempre al fine di ridurre l'uso della tastiera si è pensato di utilizzare delle finestre di dialogo per gestire questa parte. Nella scheda del paziente, i provvedimenti terapeutici sono generalmente dei farmaci somministrati al paziente per migliorare lo stato di uno dei suoi parametri di salute. Potendo organizzare questi farmaci in gruppi in funzione del trattamento voluto, si è pensato di costruire delle finestre di dialogo per ogni gruppo. La finestra di dialogo contiene i nomi dei farmaci usati all'ospedale per uno specifico gruppo. Visto che il dosaggio di un farmaco può variare, si è anche pensato di inserire un campo per compilare senza grande difficoltà il dosaggio. Queste finestre di dialogo sono state realizzate con la classe `TherProDialog` di cui l' interfaccia grafica è costituita da tre parti: una prima parte che presenta la lista dei farmaci elencati con dei bottoni radio, una `EditText` per compilare la quantità somministrata e l'ultima parte che elenca i diversi tipi di dosaggi (*g*, *mg*, *mcg*, *mcg/kg/min*) tramite un `RadioGroup` (insieme di bottoni radio). L'utente deve quindi fare un clic sul gruppo dei provvedimenti terapeutici di interesse, scegliere il farmaco, riempire un campo per specificare il dosaggio, selezionare il tipo di dosaggio e successivamente salvare quanto inserito. Come risultato, il nome del farmaco e il dosaggio vengono inseriti automaticamente nella `EditText` corrispondente ai provvedimenti terapeutici. I metodi della classe `TherProDialog` hanno sostanzialmente le stesse funzionalità di quelle della classe `AllergiesDialog` descritta



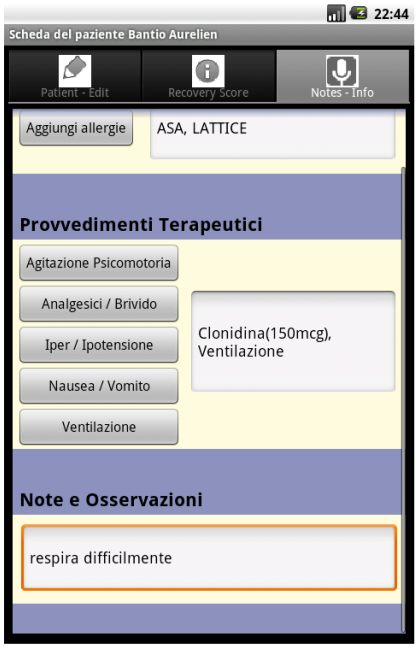
a. View contenente nessuna informazione



b. Compilazione delle allergie



c. Inserimento di un nuovo provvedimento terapeutico relativo all'Iper/Ipotensione



d. View con informazioni inserite in ogni parte

Figura 6.7: View Per l'inserimento delle allergie, provvedimenti terapeutici e altre informazioni

precedentemente.
La figura 6.7.c presenta la finestra di dialogo appena descritta mentre si inserisce un provvedimento terapeutico per un paziente.

Inserimento di altre informazioni e osservazioni

Questa parte è costituita solo da una `EditText` in cui l'utente può scrivere qualsiasi informazione.
La figura 6.7 presenta alcuni screenshot dell'interfaccia di quest'activity.

6.7.2 Metodi rilevanti dell'Activity PatientNotes

I metodi che verranno descritti di seguito sono quelli che sono stati rilevanti nell'Activity `PatientNotes` al fine della compilazione delle allergie, dei provvedimenti terapeutici, delle informazioni varie e al loro salvataggio in database:

- *populateFields*: permette di recuperare i dati del paziente corrispondenti alle allergie, ai provvedimenti terapeutici e ad altre informazioni in database e successivamente di inserirli nelle apposite `EditText`.
- *saveState*: recupera i contenuti delle `EditText` (corrispondenti alle allergie, ai provvedimenti terapeutici e a altre informazioni e osservazioni) e successivamente li salva in database.
- *createDialogs*: usato per creare le finestre di dialogo e associarle agli appositi bottoni su cui gli utenti faranno dei clic per avviare la loro visualizzazione.

6.8 Activity PatientProfile

Quest'Activity è stata usata come contenitore dei tre contesti visuali view per la compilazione dei parametri fissi e quelli relativi al monitoraggio all'arrivo, view per l'inserimento degli scores e dei rilevamenti dei valori dei drenaggi e la view per la compilazione delle allergie, dei provvedimenti terapeutici e altre note. Essa rappresenta in effetti la porta d'ingresso alla scheda di un paziente già presente in RR; si avvia dalla view principale che

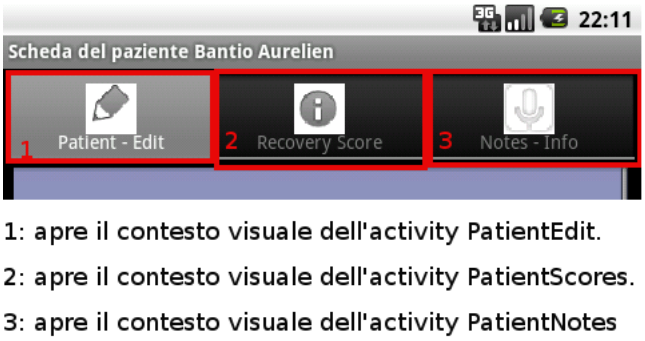


Figura 6.8: View contenente i 3 contesti visuali della scheda generale

presenta i pazienti in RR, in seguito a un clic sul paziente per cui si desidera compilare i dati.
L'idea è di mostrare la scheda del paziente sotto forma di sotto-schede (questa tecnica di visualizzazione si vede nei browser web in cui una finestra può contenere più schede), ognuna delle quali raggiungibile in un solo clic. Infatti questo ha reso più facile la navigazione nella scheda del paziente.
La classe `PatientProfile` estende la classe `TabActivity` che è un'Activity presente nelle API Android realizzata per questi tipi di view che sono contenitori di più contesti visuali. Essa contiene quindi i contesti visuali delle Activity indipendenti `PatientEdit`, `PatientScores` e `PatientNotes` descritte nelle sezioni precedenti.
L'interfaccia grafica di questa view è presentata nella figura 6.8.

6.9 View per la gestione delle impostazioni server

In questa parte verrà presentata l'Activity `Preferences` che serve per fornire all'applicazione le credenziali per l'accesso al server dei dati, utili per la fase di trasferimento.
Essa estende la classe `PreferenceActivity`, un'Activity che è stata realizzata e messa a disposizione nelle API Android al fine di essere estesa da altre Activity per una gestione agevolata delle preferenze di un'applicazione.
Essa non contiene niente di particolare oltre alla sua interfaccia realizzata

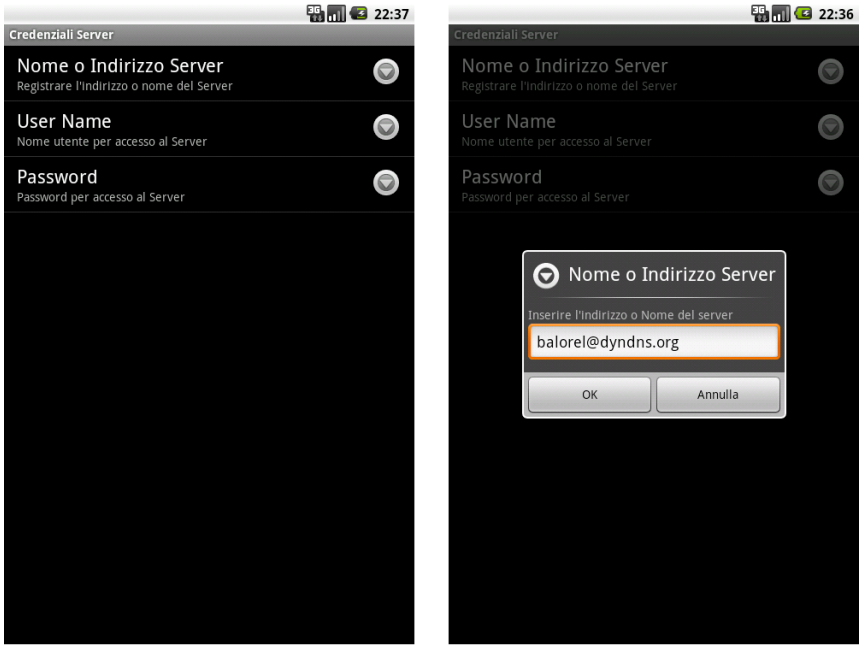
usando l'XML come illustrato nel Listing 6.10. Questo layout presenta infatti un `PreferenceScreen` che contiene tre elementi `EditTextPreference`. Il `PreferenceScreen` rappresenta la view che viene presentata all'utente, l'`EditTextPreference` è un campo testuale che viene presentato nella view solo con una barra contenente il suo titolo e il suo summary; il campo testuale compilabile viene mostrato con l'uso di una finestra di dialogo in seguito a una clic su tale barra (vedere figura 6.9.b). Questa view contiene quindi tre di quest'ultimi elementi, disposti in lista, che permettono di compilare rispettivamente l'indirizzo o il nome del server, il nome utente (username) e la password per accedere a tale server (vedere figura 6.9.a).

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen
      xmlns:android="http://schemas.android.com/apk/res/android">
      <PreferenceScreen
          android:title="@string/serverPreferencesTitle"
          android:summary="@string/serverPreferencesTitle">
          <EditTextPreference
              android:key="serverAddress"
              android:title="@string/serverAddress"
              android:summary="@string/serverAddressSummary"
              android:dialogMessage="@string/serverAddressMessage">
          </EditTextPreference>

          <EditTextPreference
              android:key="serverUsername"
              android:title="@string/serverUsername"
              android:summary="@string/serverUsernameSummary"
              android:dialogMessage="@string/serverUsernameMessage">
              </EditTextPreference>

          <EditTextPreference
              android:key="serverPassword"
              android:title="@string/serverPassword"
              android:summary="@string/serverPasswordSummary"
              android:dialogMessage="@string/serverPasswordMessage"
              android:password="true">
              </EditTextPreference>
      </PreferenceScreen>
</PreferenceScreen>
```

Listing 6.10: Layout della view per le impostazioni dei credenziali Server



a. View per impostare le credenziali server

b. View mentre si compila l'indirizzo del server

Figura 6.9: View per impostare le credenziali del server dei dati

6.10 View per procedere alle dimissioni

In questa parte verrà presentata l'Activity `DismissRobot` che serve per la compilazione dei parametri relativi al monitoraggio di dimissione del paziente, le informazioni sul personale che si è occupato del paziente durante il suo ricovero in RR e l'esito finale del paziente. Successivamente alla compilazione dei dati sopra citati, essa permette di dimettere il paziente dalla RR, trasferendo di conseguenza tutti i suoi dati verso il server dei dati. Ci si concentrerà sulla costruzione della sua interfaccia grafica e successivamente si presenteranno i suoi metodi rilevanti insistendo su quelli relativi al trasferimento dei dati verso il server.

6.10.1 Interfaccia grafica

L'interfaccia grafica di questa view è stata specificata usando unicamente il linguaggio XML. Infatti tutti i suoi componenti sono fissi e non dipendono da un particolare aspetto di un paziente tranne per il fatto che, in funzione

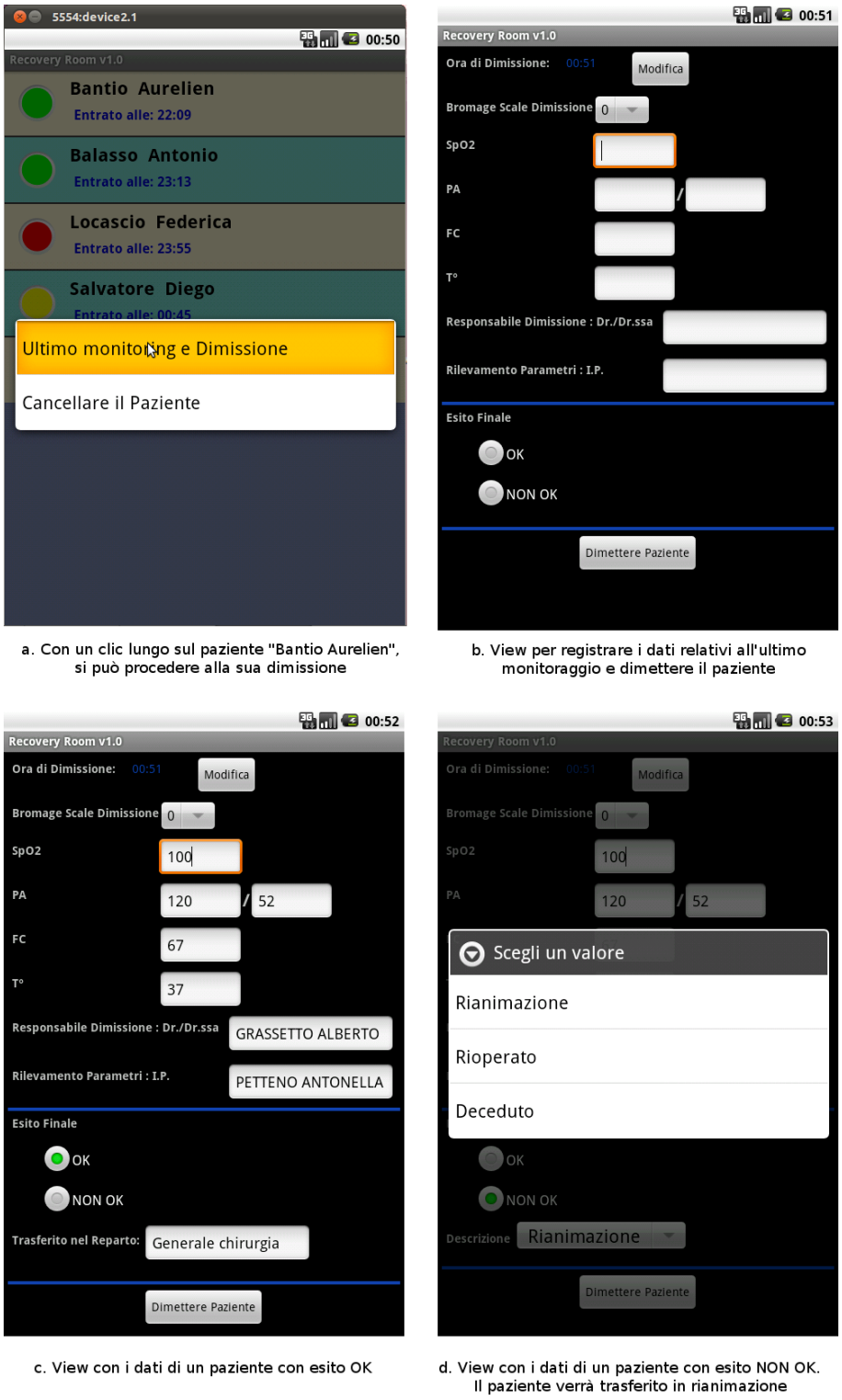
dell'esito finale, un paziente può essere trasferito in un reparto di degenza o per esempio rioperato; di conseguenza un componente del layout verrà reso invisibile al posto di un altro che verrà reso visibile per gestire tale aspetto.
L'interfaccia è composta principalmente da tre gruppi di campi:

- un gruppo che serve alla compilazione dei parametri relativi al monitoraggio di dimissione del paziente. È costituito dagli stessi campi relativi al monitoraggio all'arrivo del paziente in RR presentati nella sezione 6.3.1.

- Un gruppo di due campi testuali che servono per la compilazione degli identificativi del responsabile della dimissione e dell'infermiere professionista che ha seguito il paziente durante il suo ricovero in RR. Per la realizzazione di questi due campi si è scelto di utilizzare delle `AutoCompleteTextView` che sono associate a due file contenitori, che possiedono rispettivamente i nomi dei medici (che hanno il diritto di approvare una dimissione) e i nomi degli infermieri che lavorano nel blocco operatorio. Questa scelta, come tutte le altre fatte durante l'implementazione delle view, è sempre mirata ad agevolare la compilazione dei campi limitando l'uso della tastiera per la digitazione dei caratteri.

- Un gruppo di campi per la compilazione dell'esito del paziente: questo gruppo può presentare componenti diversi in funzione dell'esito del paziente. Contiene un gruppo di due bottoni radio che permettono di scegliere se l'esito del paziente è buono o no. Nel caso l'esito dovesse essere negativo, viene visualizzato un campo `Spinner` contente una lista di tre opzioni (Rianimazione, Rioperato, Deceduto) che rappresentano i casi possibili di un esito negativo. Nel caso di esito positivo, viene presentato un campo di tipo `AutoCompleteTexView` associato a un contenitore che contiene la lista dei reparti di degenza dove i pazienti possono essere mandati all'uscita dalla RR.

La figura 6.10 mostra l'interfaccia appena descritta.



a. Con un clic lungo sul paziente "Bantio Aurelien", si può procedere alla sua dimissione

b. View per registrare i dati relativi all'ultimo monitoraggio e dimettere il paziente

c. View con i dati di un paziente con esito OK

d. View con i dati di un paziente con esito NON OK. Il paziente verrà trasferito in rianimazione

Figura 6.10: View per inserire i dati del monitoraggio in dimissione

6.10.2 Metodi rilevanti dell'Activity DismissRobot

I metodi che verranno descritti di seguito sono quelli che sono stati rilevanti nell'Activity `DismissRobot` al fine della compilazione dei campi citati sopra e al successivo trasferimento dei dati del paziente dal database al server dei dati.

- *displayForm*: usata per creare i componenti dell'interfaccia grafica e precompilare i campi che erano già compilati se non è la prima volta che si entra in tale parte della scheda del paziente. La precompilazione si fa con l'aiuto delle informazioni recuperate in database.

- *dismissPatient*: usato per avviare la dimissione di un paziente e gestire la barra di progressione usata per informare l'utente dell'avanzamento della fase di dimissione.

- *updateProgressDialog*: usato per aggiornare l'informazione mostrata all'utente nella barra di progressione durante la lunga fase di dimissione. Si tratta qui dei messaggi che sono mandati in modo asincrono alla barra di progressione ogni volta che si passa da una sotto-fase a un'altra.

- *onCreateDialog*: usata per creare e visualizzare le finestre di dialogo dell'Activity. Quest'Activity contiene in effetti quattro finestre di dialogo: una per aiutare l'utente a compilare il campo *"ora di dimissione"* in modo consistente (`TimePickerDialog`), una per informare l'utente di un eventuale errore nella compilazione dei campi di quest'Activity (`formAlertDialog`), una per informare l'utente di un eventuale errore di compilazione dei campi dell'intera scheda del paziente (`fillingAlertDialog`) e una per informare l'utente di un eventuale problema di connessione al server dei dati (`connection-AlertDialog`).

- *checkFillingState*: usato per controllare la compilazione dei campi dell'intera scheda del paziente. Ci si aiuta con dei dati contenuti nella tabella *filling_state* del database che viene aggiornata mano a mano che le parti dell'intera scheda del paziente vengono compilati. In effetti

questa tabella tiene una traccia dello stato di compilazione dell'intera scheda e soprattutto dello stato di compilazione dei campi che sono obbligatori al fine della dimissione del paziente dalla RR.

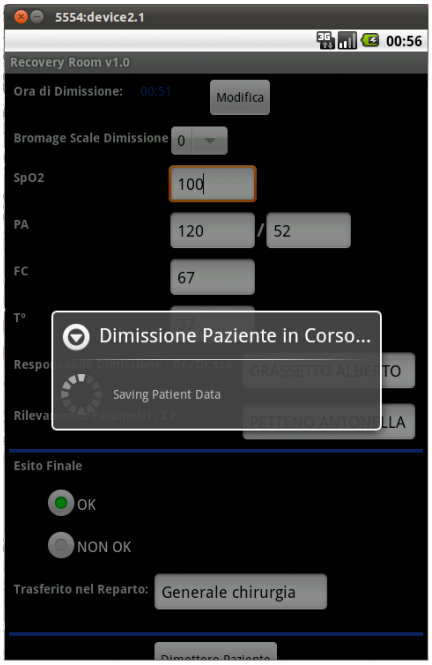
- *saveState*: usato per controllare la corretta compilazione dei campi di quest'Activity e successivamente per il salvataggio dei loro contenuti in database.

- *generateError*: utilizzato per lanciare un messaggio d'errore usando la finestra di dialogo `formAlertDialog` nel caso i campi di quest'Activity non siano stati compilati correttamente . Esso viene chiamato dal metodo `saveState` in caso di non conformità durante la fase di controllo del contenuto dei campi (vedere figura 6.11.b).

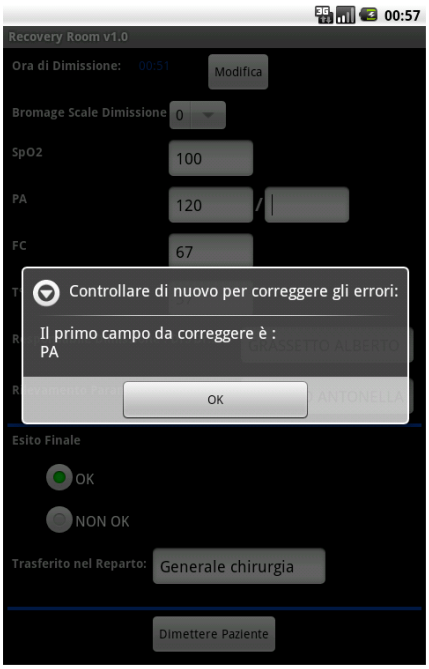
- *generateFillingError*: usato per lanciare un messaggio d'errore usando la finestra di dialogo `fillingAlertDialog` in caso di errata compilazione nei campi dell'intera scheda del paziente. Esso viene chiamato infatti dal metodo `checkFillingState` in caso di non conformità durante la fase di controllo dell'intera scheda del paziente (vedere figura 6.11.c).

- *generateConnectionError*: durante la fase di trasferimento dei dati verso il server di cui si parlerà più avanti, può succedere di avere un errore di connessione. Questo metodo serve quindi per lanciare un messaggio specifico d'errore usando la finestra di dialogo `connectionAlertDialog` nel caso si verificasse un'errore di connessione durante tale fase di trasferimento (vedere figura 6.11.d).

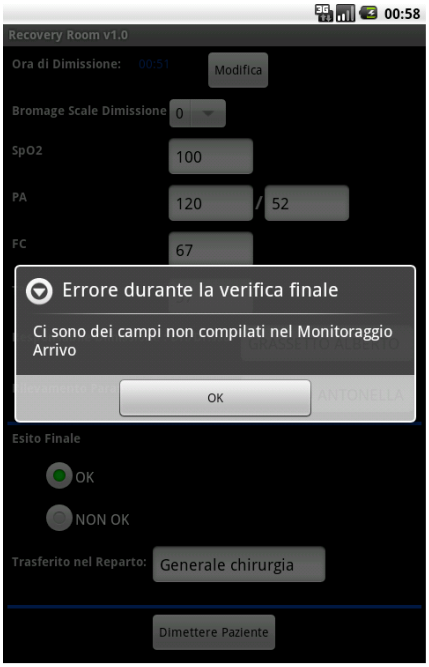
- *buildExternalResultFile*: usato per costruire i file HTML e CSV con i dati del paziente recuperati dal database. In pratica si inizia recuperando tutte le informazioni sul paziente dal database, e successivamente si costruiscono il file HTML strutturato con tutte le parti della scheda del paziente e quello csv strutturato sulla base dell'altro file CSV usato lato server come repository dei dati di tutti i pazienti passati in RR.



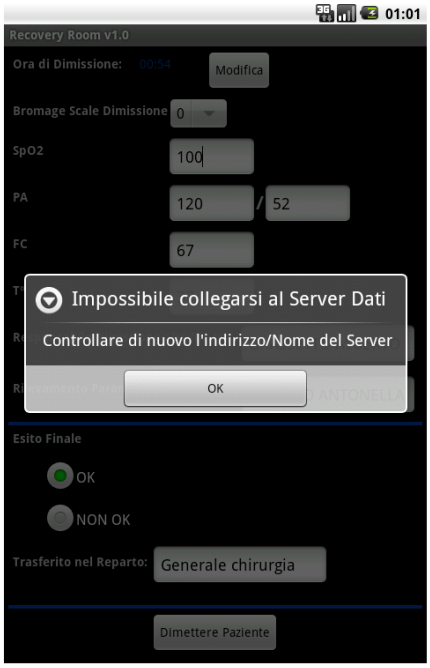
a. Inizializzazione della dimissione di un paziente



b. Finestra di dialogo che indica un errore di compilazione in questa view



c. Finestra di dialogo che indica un errore di compilazione nella parte relativa al monitoraggio all'arrivo



d. Finestra di dialogo che indica un errore di compilazione dell'indirizzo del server

Figura 6.11: Screenshot durante la dimissione di un paziente

- *sentToPrinterServer*: usato per trasferire i file HTML e CSV di cui si è parlato sopra verso il server dei dati, usando il protocollo FTP.

Per il processo di salvataggio dei dati compilati nell'Activity `Patient-Dismisser` e per quello di trasferimento di dati che sono dei processi duraturi, si è utilizzato un `Thread` (classe `Dismisser`) costruito dentro questa stessa classe, di cui il metodo `run` esegue i seguenti passi:

1. Controllare che i campi di questa view siano ben compilati e salvare successivamente i loro contenuti nel data base – metodo saveState.
2. Controllare lo stato di compilazione dei campi dell'intera scheda del paziente – metodo checkFillingState.
3. Costruire i file HTML e CSV contenente i dati del paziente – metodo buildExternalResultFile.
4. Trasferire i file HTML e CSV creati verso il server dei dati – metodo sentToPrinterServer.

Nel caso si verificasse un'errore durante l'esecuzione di un passo, il `Thread` si ferma lanciando un apposito messaggio d'errore tramite i metodi di generazione errori citati precedentemente.

6.10.3 Costruzione dei file riassuntivi e trasferimento dei dati

6.10.3.1 Costruzione del file HTML

Il file da costruire è composto dalle seguenti parti principali contenenti i dati già presenti nella scheda cartacea:

- *Dati fissi del paziente*: Cognome, Nome, Data di nascita, ecc.
- *Monitoraggio Arrivo*: valori dei parametri monitorati all'arrivo del paziente in sala (SpO2, PA, FC, Temperatura, ecc.).
- *Recovery Scores*: tabella riassuntiva degli scores rilevati sul paziente durante il suo ricovero in RR.
- *Rilevamenti dei valori dei drenaggi e della diuresi*: tabella riassuntiva dei valori dei drenaggi del paziente rilevati durante il suo ricovero in RR.

- *Monitoraggio Dimissione*: valori dei parametri monitorati alla dimissione del paziente dalla RR.

- *Altre informazioni*: allergie, provvedimenti terapeutici, altre note e stato finale.

- *Staff completo*: nominativi dei medici (anestesista e responsabile dimissione) e degli infermieri che si sono occupati del paziente durante il suo ricovero in RR.

La costruzione del file è stata realizzata all'interno del metodo `buildExternalResultFile` descritto precedentemente, appoggiandosi ai metodi della classe `FileBuilderTool` creata solo per tale scopo.
Il primo passo è stato il recupero di tutti i dati del paziente dal database. Tale passo viene presentato nel Listing 6.11 in cui `mDbHelper` rappresenta l'oggetto database e gli oggetti di tipo *Cursor* (oggetto illustrato nel paragrafo 6.2.2) `pCursor`, `fmCursor`, `lmCursor`, `sCursor`, `dCursor` e `notesCursor` contengono rispettivamente i dati fissi del paziente, i parametri relativi al monitoraggio all'arrivo, i parametri relativi al monitoraggio in dimissione, i valori degli scores rilevati durante il ricovero del paziente, i rilevamenti dei valori dei drenaggi e i dati relativi alle altre informazioni.

```
private File buildExternalResultFile() {
      [...]

      Cursor pCursor = mDbHelper.fetchPatient(patientId);
      startManagingCursor(pCursor);
      Cursor fmCursor = mDbHelper.fetchMonitoring(patientId, 0);
      startManagingCursor(fmCursor);
      Cursor lmCursor = mDbHelper.fetchMonitoring(patientId, 1);
      startManagingCursor(lmCursor);
      Cursor sCursor = mDbHelper.fetchAllScores(patientId);
      startManagingCursor(sCursor);
      Cursor dCursor = mDbHelper.fetchAllDrainageValues(patientId);
      startManagingCursor(dCursor);
      Cursor notesCursor = mDbHelper.fetchNotesInfo(patientId);
      startManagingCursor(notesCursor);
      notesCursor.moveToFirst();

      [...]
```

```
}
```

Listing 6.11: Recupero dei dati del paziente in database

Per ogni parte si è creata una sezione nel file HTML in cui i dati ad essa relativi sono stati inseriti. Mentre per le sezioni *Dati fissi del paziente*, *Monitoraggio Arrivo*, *Monitoraggio Dimissione*, *Altre informazioni* e *Staff completo* è bastato recuperare i singoli dati dal database e inserirli nel file HTML, le sezioni *Recovery Scores* e *Rilevamenti dei valori* dei drenaggi hanno richiesto più sforzo per la costruzione delle relative tabelle riassuntive. Per queste ultime sezioni si è realizzata la classe `FileBuilderTool` che è una classe che fornisce dei metodi statici che permettono di costruire le tabelle di cui abbiamo bisogno. Nel Listing 6.12 si può vedere come vengono inseriti i dati fissi del paziente nel file HTML dove `out` rappresenta il file HTML e *toHtml* è un metodo che prende come parametri d'ingresso il nome di un parametro e il suo valore e li trascrive in HTML nella forma *"**nome_parametro** : valore_parametro"*.

```
 [...]

 String lname = pCursor.getString(pCursor.getColumnIndex(DbKeys.LNAME));
 String fname = pCursor.getString(pCursor.getColumnIndex(DbKeys.FNAME));

 // replace is used to prevent file's name from containing blank space.
 //that's useful for the script used to make a pdf file.
 String fileName =
       lname.replace('␣', '-') + "_" + fname.replace('␣', '-') + ".html";
try {
 File folder =
   new File(Environment.getExternalStorageDirectory() + "/Recovery_Room_Data");
 boolean success = true;
 if(!folder.exists())
 {
   success = folder.mkdirs();
 }
 if (success && folder.canWrite()) {
   File patientHtmlFile = new File(folder, htmlFile);
   File patientCsvFile = new File(folder, csvFile);

   // ######   html file  #######
   PrintWriter out = new PrintWriter(new OutputStreamWriter(
                     new FileOutputStream(patientHtmlFile),"8859_1"));
   updateProgressDialog(0, "Now␣writing␣the␣patient's␣file␣("+htmlFile+")..");
   out.print(htmlHeader1);
   out.print(String.format(getString(R.string.htmlFileTitle), lname, fname));
```

```
  out.println(htmlHeader2);
out.println("<h1␣align=\"center\">"+getString(R.string.html_header1)+"</h1>");
  out.println(toHtml(getString(R.string.lastName), lname));
  out.println(toHtml(getString(R.string.firstName), fname));
  out.println(toHtml(getString(R.string.birthdate), birthDate));
  out.println(toHtml(getString(R.string.incomingTime), timeInfo));
  out.println("<br/>");
  out.println(toHtml(getString(R.string.unit), unit));
  out.println(toHtml(getString(R.string.intervention), intervention));
  out.println(toHtml(getString(R.string.anesthetist), anesthetist));
  out.println("<br/>");
  out.println(toHtml(getString(R.string.anesthesia), anesthesia));
  out.println(toHtml(getString(R.string.ASA), asa));
  out.println(toHtml(getString(R.string.drainageNumber), drainages_number));
[...]
} catch (IOException e) {
  Log.e(TAG, "Error␣writing␣file:␣" + e.getMessage());
  //updateProgressDialog(2, "Error creating the patient's file !");
  return null;
}
[...]
```

Listing 6.12: Creazione del file HTML e inserimento al suo interno dei dati fissi del paziente

Costruzione della tabella riassuntiva degli scores

Per la costruzione della tabella degli scores, ricordiamo innanzitutto che ogni colonna è salvata in database dentro il campo score della tabella `recovery_scores` nella forma $x_1, y_1|x_2, y_2|...|x_n, y_n|$ dove n rappresenta il numero di campi principali della scala a punteggio in uso, x_i rappresenta l'*id* del campo principale in posizione i e y_i rappresenta l'ordine dello stato rappresentativo (tra tutti gli stati rappresentativi del campo principale y_i) attribuito a y_i. Per esempio il valore **1,0 | 2,1 | 3,0 | 4,0 | 5,0 | 6,1 | 7,1 |** del campo score significa che si hanno 7 parametri principali nella scala a punteggio in uso, al primo campo principale si è attributo il suo primo stato rappresentativo, al secondo campo principale si è attributo il secondo stato rappresentativo, ecc.
Il primo problema incontrato nella costruzione della tabella degli scores in HTML è stato il fatto che serviva avere i dati da inserire nella tabella riga per riga, ma in database i dati erano salvati colonna per colonna. Per superare questo problema, si è dovuto costruire prima una matrice M di

dimensione *m*n* dove *m* rappresenta il numero totale di stati rappresentativi della scala a punteggi utilizzata, e *n* il numero di scores registrati per il paziente. Un elemento M_{ij} contiene un carattere *"X"* se lo stato rappresentativo in posizione *i* è stato scelto nell'*j*-esimo score del paziente, altrimenti non contiene nulla. Si noti che non si tiene conto del parametro principale a cui è associato lo stato rappresentativo. In realtà la matrice viene letta contemporaneamente al *parsing* del file XML che rappresenta la struttura della scala a punteggi utilizzata.
Prima della costruzione della matrice *M*, si sono recuperati gli scores in database e si è costruito un *Array* di *n* (numero di scores registrati per il paziente) *mappe* di stringhe in modo da poter avere per ogni parametro principale lo stato rappresentativo che gli è stato attributo in ogni score registrato. Questo permette di costruire la matrice *M* in un passo di lettura del file XML rappresentando la scala a punteggi in uso.
Avendo costrutto la matrice *M*, si è potuta costruire la tabella degli scores in HTML con una seconda lettura del file XML della scala a punteggi. La tabella contiene due tipi di righe: una con un parametro principale e l'altra che contiene uno stato rappresentativo. La riga con un parametro principale è costituita da una sola colonna che contiene quel parametro principale che si può leggere proseguendo nel parsing del file XML. La riga che contiene uno stato rappresentativo contiene nella prima colonna tale stato rappresentativo e nelle altre colonne contiene gli elementi della riga *i* della matrice *M*, dove i rappresenta la posizione dello stato rappresentativo nelle scala quando si considerano solo gli stati rappresentativi escludendo i parametri principali.
Si può riassumere la costruzione della tabella degli scores in HTML in questo modo:

1. Recupero dei valori degli scores in database: con questo si ha anche il numero di scores che sono stati registrati sul paziente.

2. Costruzione dell'Array di mappe: permette di poter avere per ogni parametro principale, lo stato rappresentativo che gli è stato assegnato in un determinato score registrato. (metodo *getMaps* della classe *FileBuilderTool*).

3. Costruzione della matrice M: permette di poter avere gli elementi della tabella degli scores riga per riga. (metodo *buildMatrix* della classe *FileBuilderTool*).

4. Costruzione vera e propria della tabella degli scores in HTML leggendo contemporaneamente il file XML della scala e la matrice M. (metodo *getHtmlStructureScores* della classe *FileBuilderTool*).

Costruzione della tabella riassuntiva dei rilevamenti dei valori dei drenaggi e della diuresi

Ricordiamo, anche in questo caso, come vengono salvati in database i valori dei drenaggi e della diuresi. Ogni *entry* nella tabella dei drenaggi contiene nel campo drenaggio un valore testuale strutturato nella forma $x_1|x_2|...|x_n|$ dove *n* rappresenta il numero dei drenaggi posseduti dal paziente e x_i il valore dell'*i*-esimo drenaggio; inoltre contiene un campo numerico chiamato diuresi in cui viene salvato il valore della diuresi.
Per la costruzione di questa tabella riassuntiva dei rilevamenti dei valori dei drenaggi e della diuresi, sussisteva lo stesso problema di non avere i dati riga per riga. Si è usata la stessa tecnica illustrata per la costruzione della tabella degli scores, costruendo una matrice *D* di dimensione *n*m* per contenere i dati della tabella; in questo caso *n* rappresenta il numero di drenaggi posseduti dal paziente e *m* il numero di rilevamenti dei drenaggi effettuati sul paziente. I passi per la costruzione della tabella sono i seguenti:

1. Recupero dei dati in database.

2. Costruzione della matrice D analizzando il contenuto dei dati. (metodo getDrainTable della classe FileBuilderTool).

3. Costruzione vera e propria della tabella dei drenaggi in HTML, aggiungendo i valori rilevati per la diuresi come ultima riga della tabella. (metodo *getHtmlStructureDrains* della classe *FileBuilderTool*).

6.10.3.2 Costruzione del file CSV

Questo file si compone di un'unica riga scritta nel formato csv, contenente tutti i dati del paziente. La riga si presenta sulla forma "x_1","x_2",...,"x_n" dove ogni x_i rappresenta un'informazione da prendere in considerazione. L'ordine dei campi x_i è identico a quello del file csv usato lato server per raggruppare i dati di tutti i pazienti.
Come per la costruzione del file HTML, la scrittura delle parti relative agli scores e ai rilevamenti dei valori dei drenaggi è stata problematica per il fatto che il numero di scores registrati e il numero di rilevamenti dei valori dei drenaggi non sono fissi per tutti i pazienti. Si è scelto di mettere quelle informazioni alla fine della riga, lasciando 5 campi per i rilevamenti dei valori dei drenaggi e 10 per gli scores. Questi valori sono stati scelti notando che non è mai successo in precedenza di avere tali numeri di rilevamenti e di scores per un paziente.

6.10.3.3 Trasferimento dei file verso il server

Per questa parte di trasferimento dei dati, si è utilizzata una libreria FTP realizzata dal gruppo *Apache* nel quadro del progetto Apache Commons[1], che mette a disposizione molti componenti Java ri-utilizzabili. Essa permette di stabilire una comunicazione con un server e di trasferirvi dei file utilizzando il protocollo File Transfer Protocol (FTP).
Avendo creato un server FTP sul server dei dati, si è quindi potuto trasferirvi i file costruiti, utilizzando il protocollo FTP e sfruttando la connettività Wi-Fi del dispositivo.

[1] http://commons.apache.org

Capitolo 7

Gestione dei dati lato server

In questa parte verrà presentata la gestione dei dati del paziente lato server. Si inizierà presentando i tools installati nel server e successivamente si presenteranno le elaborazioni effettuate sui file ricevuti dal server.

7.1 Server FTP

La prima operazione svolta a livello server è stata l'installazione di un server FTP per gestire il trasferimento dei file dal dispositivo al server. Si è utilizzato a tale scopo il programma vsftpd[1], che è un demone FTP disponibile sotto licenza GNU GPL[2], installabile sui sistemi UNIX e Linux in particolare. Esso garantisce stabilità, rapidità e sicurezza.
Si è configurato il server in modo che l'accesso sia gestito tramite autenticazione e si è creato di conseguenza un utente denominato *"room"*, che ha la possibilità di fare degli upload nella directory `/home/FTP-shared` creata per contenere i file provenienti dal dispositivo mobile. Dal dispositivo si può quindi trasferire i file nella cartella specificata usando il nome utente *"room"* e la password che si è associata a esso.

[1] Very Secure File Transfer Protocol Daemon; http://vsftpd.beasts.org/
[2] GNU General Public License; http://www.gnu.org/licenses/gpl.html

7.2 Server di Stampa

Il server di stampa si è reso necessario perché si deve stampare la scheda riassuntiva del paziente al momento delle dimissioni e allegarla al resto della sua cartella clinica.
La scelta si è portata sul server CUPS[3], che è un *print spooler* modulare (sistema informatico con la funzione di memorizzare, secondo la logica FIFO, le stampe degli utenti ed inviarle alla stampante appena questa è disponibile) per sistemi operativi di tipo Unix che permette a un computer di funzionare come un efficace server di stampa.
Su questo server vengono installate una o più stampanti. I file dei pazienti vengono mandati a questo server nel formato pdf ogni volta che un paziente viene dimesso; il server poi gestisce le stampe mandandole a una delle sue stampanti disponibili.

7.3 Elaborazioni sui file ricevuti dal server

Per la gestione dei file nel server, si sono create due cartelle e un file *csv* per raggruppare tutti i dati dei pazienti passati in RR. Le due cartelle `/home/FTP-shared` e `/home/Printed_Files` servono rispettivamente per contenere i due file *csv* e *html* ricevuti dal dispositivi ogni volta che un paziente viene dimesso, e i file pdf risultanti dalla conversione in pdf del file *html* del paziente. I file non permangono nella cartella `/home/FTP-shared`: essa serve solo da cartella di transizione dei file che provengono dal dispositivo. Infatti, ogni file trasferito in quella cartella non rimane al suo interno per più di dieci secondi come vedremo più avanti.

7.3.1 Gestione del file HTML

Avendo ricevuto dal dispositivo mobile il file *html* riassuntivo della scheda del paziente, è necessario convertirlo nel formato pdf prima di mandarlo in stampa, altrimenti, non si otterrebbe un documento leggibile ma piuttosto un documento contenente il codice *html*.

[3]Common Unix Printing System; http://www.cups.org/

Allo scopo di convertire il file html in pdf si è usato il programma *HtmlDoc*, un piccolo programma scritto in C++ che permette di convertire le pagine html in pdf o ps. Si è installata quindi l'ultima versione stabile di questo programma (*HTMLDOC v1.8.27.1*), poi si è configurato il programma in modo da avere un layout del file di output che rispettasse i requisiti dei medici, quindi simile alla scheda cartacea (tabella degli scores contenuta interamente nella prima pagina, ecc.).
Lo script nel Listing 7.1 è stato inserito nel Cron della macchina server e viene eseguito ogni minuto. Cron è un *"job scheduler"* incluso nei sistemi operativi Unix che permette agli utenti di schedulare dei job (comandi o script shell) in modo che possano essere eseguiti periodicamente in determinati tempi e date.
Nello script, tutto il contenuto del loop esterno viene eseguito ogni 10 secondi, di conseguenza il file *html* non rimane più di 10 secondi nella cartella `/home/FTP-shared`. Esso viene trasformato in pdf con il comando `htmldoc –webpage -f /home/Printed_Files/$file%.*.pdf /home/FTP-shared;` il file pdf viene mandato nella cartella `/home/Printed_Files` che costituisce la directory di archiviazione delle schede dei pazienti.
Le figure 7.1 e 7.2 presentano rispettivamente un esempio della prima e della seconda pagina della scheda prodotta alla dimissione di un paziente.

```
#!/bin/bash

let "expireTime = $(date +%s) + 58"
while [ $(date +%s) -lt $expireTime ]
do
  for file in `ls /home/FTP-shared/`
  # FTP-shared is the folder in which the device upload patient files.
  do
    case ${file##*.} in
      "html")
              `htmldoc --webpage -f /home/Printed_Files/${file%.*}.pdf
                                                    /home/FTP-shared/$file`
              if [ $? != 0 ]
              # if the exit status of the last command was bad.
              then
                # echo "Error printing ${file%.*}.pdf: $(date)" >>
                                                /tmp/RecoveryRoom_Print_Log.log
                exit 1
              fi
              `lpr /home/Printed_Files/${file%.*}.pdf`
              # delete the elaborated file.
```

```
22                 `rm /home/FTP-shared/$file`
                   # echo "file ${file%.*}.pdf printed Successfully: $(date)"
24                 #                                >> /tmp/RecoveryRoom_Print_Log.log
                   ;;
26
        "csv")
28                 cat /home/FTP-shared/$file >> /home/recovery_room.csv
                   # delete the elaborated file.
30                 `rm /home/FTP-shared/$file`
                   # echo "added successfully data for ${file%.*}
32                 #    in the csv file: $(date)" >> /tmp/RecoveryRoom_Print_Log.log
                                ;;
34      esac
      done
36    sleep 10  # wait 10 seconds.
  done
38  # echo "the script is alive: $(date)" >> /tmp/RecoveryRoom_Print_Log.log
```

Listing 7.1: Script usato per la gestione dei file nel server

7.3.2 Gestione del file csv

Nel server, il file *"recovery_room.csv"* è usato per mettere insieme tutte le informazioni sui pazienti che passano in RR. Esso può essere visto come una tabella di foglio elettronico di cui l'intestazione è costituita rispettivamente dai seguenti campi: *Scala a Punteggi*, *Data Ricovero*, *Ora Arrivo*, *Ora Dimissione*, *Cognome*, *Nome*, *Data Nascita*, *Reparto*, *Intervento*, *Anestesista*, *Anestesia*, *ASA*, *Numero Drenaggi*, *SpO2_arrivo*, *PA_arrivo*, *FC_arrivo*, *T_arrivo*, *Bromage_arrivo*, *SpO2_dimissione*, *PA_dimissione*, *FC_dimissione*, *T_dimissione*, *Bromage_dimissione*, *Allergie*, *Prov. Terapeutici*, *Note & Osservazioni*, *Stato Finale*, *Dopo RR*, *Responsabile Dimissione*, *I.P.*, *Rilevamento_1*, *Diuresi_1*, *Rilevamento_2*, *Diuresi_2*, *Rilevamento_3*, *Diuresi_3*, *Rilevamento_4*, *Diuresi_4*, *Rilevamento_5*, *Diuresi_5*, *Score_1*, *Score_2*, *Score_3*, *Score_4*, *Score_5*, *Score_6*, *Score_7*, *Score_8*, *Score_9*, *Score_10*. Dove il campo *"Scala a Punteggi"* serve per specificare il nome della scala a punteggi usata per segnare gli scores, il campo *"Rilevamento_i"* serve per specificare l'i-esimo rilevamento dei valori dei drenaggi, il campo *"Diuresi_i"* serve per specificare l'i-esimo valore della diuresi, il campo *"Score_i"* serve per specificare l'i-esimo score. Questi ultimi campi possono essere vuoti se non si è arrivato a segnare *"i"* volte i valori dei drenaggi, della diuresi oppure degli scores.
Il file *csv* del paziente che arriva nella cartella `/home/FTP-shared` con-

Informazioni del Paziente Bantio Aurelien

Resoconto Informazioni - Recovery Room

Cognome : Bantio **Nome** : Aurelien **Data di nascita** : 27/2/1985 **Ora di arrivo** : 22:09
Reparto : Generale chirurgia **Intervento** : Inter. 1 **Anestesista** : GRASSETTO ALBERTO
Anestesia : Generale, **ASA** : 0 **Numero di drenaggi** : 2

Monitoraggio Arrivo

SpO2 : 98 **PA** : 130/80 **FC** : 73 **T°** : 38 **Bromage Scale Arrivo** : 0

Recovery Scores

RECOVERY SCORE	00:02	00:13	01:00
Livello di coscienza			
Sveglio e orientato	**X**	**X**	**X**
Risvegliabile con un minimo stimolo			
Responsivo solo alla stimolazione tattile			
Attività fisica			
Muove tutte le estremità a comando			**X**
Debolezza nel muovere le estremità	**X**		
Incapace di muovere volontariamente le estremità		**X**	
Stabilità emodinamica			
PA ± 15% dei valori preoperatori	**X**		**X**
PA ± 30% dei valori preoperatori		**X**	
PA > 30% dei valori preoperatori			
Stabilità respiratoria			
In grado di respirare profondamente	**X**	**X**	**X**
Respiro superficiale (tachipnea) con buona capacità di tossire			
Dispnea con debole capacità di tossire			
Saturazione d'ossigeno			
> 90% in aria ambiente	**X**	**X**	**X**
Richiede ossigeno supplementare (occhiali)			
< 90% con ossigeno supplementare			
Dolore postoperatorio			
Assente o lieve			**X**
Moderato o grave, controllato con analgesici ev		**X**	
Grave persistente	**X**		
Vomito postoperatorio			
Assente o lieve nausea			**X**
Vomito transitorio o conati di vomito	**X**	**X**	
Vomito o nausea moderata/grave persistente			
SCORE TOTALE :	**10**	**9**	**14**

Figura 7.1: Esempio della scheda prodotta al momento delle dimissioni (fronte)

Informazioni del Paziente Bantio Aurelien

Valori dei Drenaggi

DRENAGGI	00:01	01:01
Drenaggio 1	12.0	15.0
Drenaggio 2	50.0	60.0
DIURESI	75	80

Monitoraggio Dimissione

Ora di Dimissione : 23:12
SpO2 : 100 **PA** : 120/52 **FC** : 67 **T°** : 37 **Bromage Scale Arrivo** : 0

Informazioni aggiuntive, Note e Osservazioni

Allergie

ASA, LATTICE

Provvedimenti Terapeutici

Clonidina(150mcg), Ventilazione

Note e Osservazioni

respira difficilmente

Esito Finale

Esito: : OK **Trasferito nel Reparto:** : Generale chirurgia

Staff Completo

Anestetista : Dr./Dr.ssa GRASSETTO ALBERTO
Responsabile Dimissione : Dr./Dr.ssa GRASSETTO ALBERTO
Rilevamento Parametri : I.P. PETTENO ANTONELLA

Figura 7.2: Esempio della scheda prodotta al momento delle dimissioni (retro)

tiene un'unica riga costituita dai valori dei campi citati sopra e messi nello stesso ordine. Con le righe da 28 a 32 dello script del Listing 7.1, la riga del file *csv* del paziente viene messa in fondo al file *"recovery_room.csv"*, e successivamente questo file viene cancellato dalla cartella `/home/FTP-shared`. Anche qui si può notare che il file *csv* del paziente non trascorre più di dieci secondi nella cartella `/home/FTP-shared`.

Capitolo 8

Analisi dei dati e risultati ottenuti

In questo capitolo verrà illustrata la parte di lavoro relativa all'analisi dei dati. Si inizierà presentando le tecniche di classificazione insistendo sulla costruzione degli alberi di decisione; successivamente si presenterà l'implementazione fatta da *John Ross Quinlan* di un algoritmo di costruzione degli alberi di decisione infine si presenteranno i risultati ottenuti utilizzando questa implementazione per l'analisi dei dati dei pazienti.

8.1 Approccio generale alla classificazione

La classificazione è un problema di *data mining*[1] che consiste nella scoperta di un modello o una funzione che mappi oggetti in classi predeterminate. Il modello o la funzione è determinato a partire da un un *training set* di dati preclassificati. Questo problema viene ancora chiamato in letteratura *Supervised Learning*.
L'input di un problema di classificazione è un *training set* T dove $T = \{t_1, t_2, ..., t_n\}$ è un insieme di *n* record. Ogni record t_i è composta da:

[1] È un ambito dell'informatica che tratta della **scoperta automatica** di **un'informazione interessante** (non conosciuta in precedenza, potenzialmente utile, non esplicitamente contenuta nei dati) da grandi moli di dati memorizzati su repository di natura diversa (ad esempio database, data warehouse, ecc.)

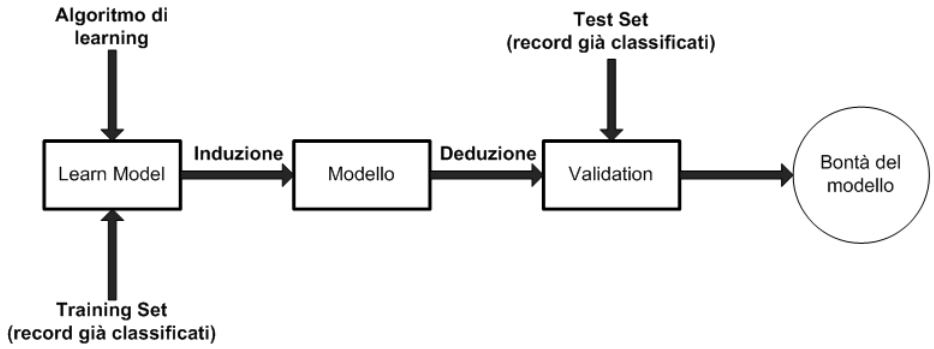


Figura 8.1: Approccio alla Classificazione

- *m* attributi (o feature) $a_{i1}, a_{i2}, \ldots, a_{im}$ dove $(a_{i1}, a_{i2}, \ldots, a_{im})$ è un m-upla $\in D_1 \times D_2 \times ... \times D_m$ cioè ogni a_{ij} appartiene al dominio D_j, 1≤j≤m.
- una classe c_i che appartiene all'insieme $C = \{c_1, c_2, \ldots, c_c\}$ delle classi predefinite.

L'output è una *target function* o *classification model* ***f*** che descrive sinteticamente e accuratamente le classi e che può essere utilizzato per assegnare la classe di un record di cui si conoscono solo le feature. (***f***: $D_1 \times D_2 \times ... \times D_m$ $\rightarrow$ C).
Gli attributi possono appartenere ai seguenti tipi:

- *categorici*: questa gruppo contiene gli attributi nominali a dominio discreto (ad esempio il codice fiscale) e gli attributi ordinali a dominio discreto ma senza ordinamento totale (ad esempio un giudizio sufficiente, buono, ottimo).
- *numerici*: che raggruppa gli attributi discreti (ad esempio l'insieme dei numeri naturali) e gli attributi continui (ad esempio l'insieme dei numeri reali).

L'approccio alla classificazione può essere sintetizzato con il grafico di figura 8.1.

Le metriche per misurare la bontà di un modello di classificazione sono le seguenti:

1. *Confusion matrix*: è un matrice *M* bidimensionale in cui l'elemento m_{ij} rappresenta il numero di record del *test set* di classe c_i che il modello ha associato erroneamente alla classe c_j.

2. $$Accuracy = \frac{\text{Numero di record classificati correttamente}}{|T|}$$

3. $$Error\ rate = \frac{\text{Numero di record classificati non correttamente}}{|T|}$$

Esistono più tipologie di modelli di classificazione. I più noti sono i seguenti:

- alberi di decisione (più utilizzati),
- reti neurali,
- classificatori bayesiani,
- support vector machine,
- classificatori basati su regole.

8.2 Alberi di decisione e algoritmo di *Hunt*

Un albero di decisione per un *training set* T è un albero per il quale valgono le seguenti caratteristiche:

- ogni nodo è associato a un test su un attributo e gli archi verso i suoi figli sono etichettati con i possibili esiti del test;
- ogni nodo foglia *u* è associato a un sottoinsieme di record $T_u \subseteq T$, e
  - date 2 foglie u_1 e u_2, $u_1 \neq u_2$, deve essere $T_{u_1} \cap T_{u_2} = \emptyset$;
  - $\bigcup_u T_u = T$

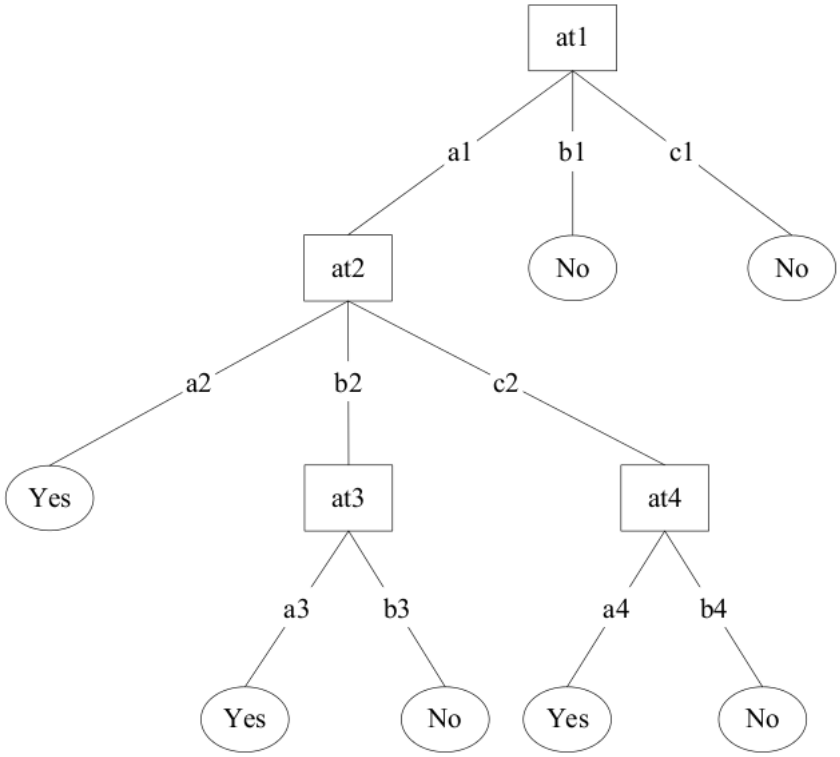


Figura 8.2: Albero di decisione

La foglia u è etichettata con la classe di maggioranza in T_u, e i valori degli attributi di ciascun record di T_u sono coerenti con i risultati dei test incontrati nel camino dalla radice a u.
La figura 8.2 riporta un esempio di un albero di decisione che corrisponde al *training set* della tabella 8.1.

Gli alberi di decisione sono tra i metodi di classificazione più utilizzati perché sono di facile interpretazione, hanno una buona accuratezza per molti tipi di problemi, sono robusti rispetto al rumore e alla ridondanza, e in più sono facili da costruire.

at1	**at2**	**at3**	**at4**	**Classe**
a1	a2	a3	a4	Yes
a1	a2	a3	b4	Yes
a1	b2	a3	a4	Yes
a1	b2	b3	b4	No
a1	c2	a3	a4	Yes
a1	c2	a3	b4	No
b1	b2	b3	b4	No
c1	b2	b3	b4	No

Tabella 8.1: Training Set

Tipi di Test

Come già detto in precedenza, ogni nodo di un albero di decisione è associato a un test su un attributo e gli archi verso i suoi figli sono etichettati con i possibili esiti del test. Il tipo di test effettuato dipende dal tipo di attributo che si ha in un determinato nodo. Si possono avere i seguenti casi:

- in caso di attributo numerico A: $A \leq x$
- in caso di attributo categorico ordinale A: $A \leq x$
- in caso di attributo categorico nominale A: $A \in \psi$ dove ψ è un sottoinsieme dei valori del dominio di A.
  Dominio di A con k valori $\Longrightarrow \frac{1}{2}(2^k - 2) = 2^{k-1} - 1$ possibili test.

Algoritmo di *Hunt* per Decision Tree Induction

Nel metodo di *Hunt*, un albero decisionale è costruito in maniera ricorsiva partizionando i record in sottoinsiemi successivamente più puri, cioè raggruppandoli il più possibile in sottoinsiemi di record appartenenti alla stessa classe. Supponendo di voler costruire un albero di decisione con il *training set* T e avendo l'insieme $C = \{c_1, c_2, \ldots, c_c\}$ di classi predefinite, ci sono tre possibilità:

- T contiene uno o più record, tutti appartenenti alla stessa classe c_i: l'albero di decisione per T è un nodo foglia etichettato con la classe c_i.

- T contiene dei record che appartengono a più di una classe: in questo caso l'idea è di selezionare un attributo su cui eseguire un test per partizionare i record in sottoinsiemi. Un nodo figlio viene creato per ogni esito ottenuto nel test e i record dell'insieme T vengono distribuiti nei nodi figli in funzione del loro valore dell'attributo scelto per effettuare il test. Successivamente l'algoritmo viene ricorsivamente applicato ai nodi figli.

- T non contiene nessun record: l'albero di decisione per T viene considerato come un nodo foglia etichettato con la classe di maggioranza dei record del nodo padre di quest'ultimo.

Algoritmo $TreeGrouth(E, F)$
$T \equiv$ training set
$F \equiv$ insieme delle m feature che caratterizzano ciascun record
input: F *ed* $E \subseteq T$
output: radice di un albero di decisione per E
prima chiamata: $TreeGrouth(T, F)$

if $stopping\_cond(E, F)$ **then**
  crea una foglia LEAF associata a E ed etichettata con la classe di maggioranza di E
  return LEAF
else
  crea un nuovo nodo $root$
  $root\_test\_cond \leftarrow find\_best\_split(E, F)$
  sia $V = \{v : \ v \ possibile \ esito \ di \ root\_test\_cond\}$
  for all $v \in V$ **do**
    $E_v \leftarrow \{e \in E : root\_test\_cond(e) = v\}$
    $child \leftarrow TreeGrouth(E_v, F)$
    aggiungi $child$ come figlio di root
  end for
  return $root$
end if

Ricerca dello split migliore: $find\_best\_split$

Sia E un insieme di record, $E \subseteq T, T : training\ set$
un test su un attributo crea uno split di E (partizione) $E \to E_1, E_2, ..., E_k$,

$$E = \bigcup_{i=1}^{k} E_i, \quad E_i \bigcap E_j = \emptyset, \quad i \neq j$$

.
La bontà di uno split (ovvero la partizione) è legata alla differenza tra la purezza di E e la purezza aggregata di $E_1, E_2, ..., E_k$. Serve quindi una **misura di impurità** di un insieme di record.

Misura di impurità

Sia p_i la frazione dei record di E appartenenti alla classe c_i, $0 \leq i < c$, $p_i \in [0, 1]$ e $\sum_{i=0}^{c-1} p_i = 1$.

- ***Misura di Gini***: $Gini(E) = 1 - \sum_{i=0}^{c-1}(p_i)^2$

$$Gini(E) = 0 \Longleftrightarrow \exists!\, i^*; p_i = \begin{cases} 1 & \text{se } i = i^* \\ 0 & \text{se } i \neq i^* \end{cases} \quad (Massima\ Purezza)$$

$$p_i = \frac{1}{c}\ \forall i \Longrightarrow Gini(E) = 1 - \frac{1}{c} \quad (Minima\ Purezza)$$

- ***Misura di Entropia***: $Entropy(E) = -\sum_{i=0}^{c-1} p_i \log_2 p_i$

$$Entropy(E) = 0 \Longleftrightarrow \exists!\, i^*; p_i = \begin{cases} 1 & \text{se } i = i^* \\ 0 & \text{se } i \neq i^* \end{cases} \quad (Massima\ Purezza)$$

$$p_i = \frac{1}{c}\ \forall i \Longrightarrow Entropy(E) = \log_2 c \quad (Minima\ Purezza)$$

- ***Misura dell'errore di classificazione***: $C_e(E) = 1 - max\left\{p_i : 0 \leq i < c\right\}$

$$C_e = 0 \Longleftrightarrow \exists!\, i^*; p_i = \begin{cases} 1 & \text{se } i = i^* \\ 0 & \text{se } i \neq i^* \end{cases} \quad (Massima\ Purezza)$$

$$p_i = \frac{1}{c}\ \forall i \Longrightarrow C_e(E) = 1 - \frac{1}{c} \quad (Minima\ Purezza)$$

Bontà di uno split: di $E\ in\ E_1, E_2, \cdots, E_k$

Siano $N = |E|$, $N_j = |E_j|$, $1 \leq j \leq k$
$I(\cdot)$, la misura di impurità scelta.

$$\boxed{GAIN = \Delta = I(E) - \sum_{j=1}^{k} \frac{N_j}{N} I(E_j)}$$

Se la misura di impurità utilizzata è la *misura di entropia*, il GAIN Δ è chiamato INFORMATION GAIN.
Trovare il miglior split (funzione *find_best_split*) significa cercare tra tutti gli attributi e tra tutti i possibili split ottenibili con questi attributi quello con GAIN più alto.
Il numero di split ottenibile da un attributo dipende dalla natura dell'attributo stesso. Si sceglie spesso uno split binario. Una casistica degli split binari ottenibili è la seguente:

- *Attributo binario*: 1 solo split possibile.
- *Attributo categorico A nominale con k valori*: $\frac{1}{2}(2^k - 2) = 2^{k-1} - 1$ split possibili; $A \in \psi$ con ψ un sottoinsieme di valori.
- *Attributo ordinale A con k valori* $x_1 < x_2 < \cdots < x_k$: $k-1$ split possibili; $A \leq x_i, 1 \leq i \leq k$.
- *Attributo numerico*: siano $x_1, x_2, \cdots, x_k$; $x_1 < x_2 < \cdots < x_k$, i valori che compaiono nel *training set*. Allora si può prendere $A \leq y_i, \quad 1 \leq i \leq k+1$; con $y_1 < x_1, \quad y_i \in [x_{i-1}, x_i) \quad 1 < i \leq k, \quad y_{k+1} \geq x_k$. Si hanno quindi $k+1$ split possibili

Valutazione della qualità di un classification model

Siano $U \equiv universo\ di\ record$, $E \subseteq U \equiv training\ set$,
$M \equiv modello\ costruito\ a\ partire\ da\ E$.
$\forall t \in U$, definendo $M(t) \equiv classe\ assegnata\ a\ t\ secondo\ il\ modello\ M, t(c) \equiv classe\ vera\ di\ t$, si hanno:

- $si\ ha\ un\ Training\ Error\ se\ per\ t \in E, M(t) \neq t(c)$
- $si\ ha\ un\ errore\ di\ generalizzazione\ se\ per\ t \in U - E, M(t) \neq t(c)$

- $Training\ Error\ Rate := \frac{|\{t \in E : M(t) \neq t(c)\}|}{|E|}$
- $Accuracy\ di\ M$: $probabilità(M(T) = T(c))$ dove T è la variabile aleatoria su U con la stessa distribuzione usata per scegliere E.

Problemi che si possono avere

Classificando si possono avere i seguenti problemi:

- *Underfitting*: avviene quando il modello M non rappresenta bene il *training set* E; si ha in questo caso un elevato *training error rate* e una bassa *accuracy*.
- *Overfitting*: avviene quando il modello M rappresenta troppo accuratamente il *training set* E; questo implica un basso *training error rate* e una bassa o media *accuracy*.
  Questo problema può essere dovuto alla presenza di rumore in E oppure alla bassa cardinalità di E. Per rimediare a esso, si usano due tecniche chiamate *Prepruning* e *Postpruning*.

8.3 Implementazione di Quinlan (C4.5)

Per la costruizione degli alberi di decisione, *John Ross Quinlan* ha implementato l'algoritmo di *Hunt* nel suo software chiamto C4.5. Esso è uno dei più noti in questo ambito di classificazione.
C4.5 usa *Gini* come misura di impurità e per il calcolo dello split migliore, il *GAIN* Δ è aggiustato calcolando come *GAIN*:

$$GAIN = \frac{\Delta}{split\ info} \quad \text{dove} \quad \text{split info} = -\sum_{j=1}^{k} \frac{|E_j|}{|E|} \log_2 \frac{|E_j|}{|E|}; \quad max|E_j| = \frac{|E|}{k}$$

.

Input del programma

La costruzione di un albero di decisione con C4.5 richiede l'uso di due file di input: uno che definisce l'insieme delle classi e attributi del sistema, e uno che contiene il *training set*, cioè l'insieme dei record già classificati con

```
good, bad.

age:          first,second,third,fourth,fifth,sixth
unit:         TOR,ONC,MAXI,PED,PLA,VASC,OCU,ORL,GINE,NCH,ORT,CAR,URO
anesthesia:   GEN,PLE,SUB,PER,LOC,SED,ANA,COM
asa:          continuous
spo2:         continuous
pa_max:       continuous
pa_min:       continuous
fc:           continuous
bromage:      continuous
```

Figura 8.3: Esempio di file di definizione delle classi e attributi

il quale si deve costruire l'albero. Nel caso in cui si è interessati a valutare la qualità dell'albero prodotto, serve un terzo file contenente il *test set*. I file devono avere lo stesso nome ma con estensione diversa e sono strutturati come segue:

- *Definizione delle classi e attributi*: il file che definisce le classi e gli attributi ha estenzione *.names*. La prima riga di questo file contiene le classi separate da una virgola. Le righe successive del file contengono ognuna un attributo del sistema seguito dal carattere *":"* e dal suo tipo. La figura 8.3 presenta un esempio di file di definizione delle classi e degli attributi, che contiene 9 attributi: l'attributo *age* può prendere i valori *first, second, third, fourth, fifth, sixth*; l'attributo *unit* può prendere i valori *TOR, ONC, MAXI, PED, PLA, VASC, OCU, ORL, GINE, NCH, ORT, CAR, URO*; l'attributo *anesthesia* può prendere i valori *GEN, PLE, SUB, PER, LOC, SED, ANA, COM*; e gli altri possono prendere dei valori reali (C4.5 usa il termine "continuous" per indicare il tipo numerico).

- *Training set*: il file che contiene i casi per il *training set* ha estenzione *.data*. Ogni riga di questo file rappresenta un caso. Essa contiene i valori degli attributi scritti nell'ordine in cui sono stati definiti nel file *.names* e separati con virgole (","), e seguiti dalla classe del caso che si sta specificando. La figura 8.4 mostra un esempio di file per un *training set* che si può associare al file di definizione delle classi e degli attributi di figura 8.3. La prima riga specifica un record in cui

```
fifth,ONC,COM,2,100,141,60,66,?,good
third,ONC,GEN,3,95,?,?,111,?,bad
third,GINE,GEN,?,100,121,67,82,0,bad
fourth,PLA,GEN,?,100,115,95,95,?,good
fourth,ONC,GEN,?,?,?,?,106,?,bad
fifth,TOR,GEN,?,99,155,70,86,?,good
third,GINE,GEN,?,100,129,77,118,?,good
fifth,OCU,GEN,?,94,130,75,83,?,good
third,GINE,GEN,?,98,109,70,103,?,bad
third,ORT,?,1,100,140,84,62,?,bad
```

Figura 8.4: Esempio di file contenente il training set

l'attributo *age* ha valore *fifth*, l'attributo *unit* ha valore *ONC*, l'attributo *anesthesia* ha valore *COM*, ecc. Questo record appartiene alla classe *good*. In questa riga si nota che l'attributo *bromage* ha valore *?*: questo sta a significare che il valore dell'attributo *bromage* non è conosciuto oppure non ha senso specificarlo in questo record.

- *Test set*: il file che contiene il *test set* ha estenzione *.test*. La sua struttura è identica a quella del file che contiene il *training set*.

Output del programma

L'ouput di C4.5 fornisce la descrizione dell'albero di decisione prodotto; successivamente fornisce anche una valutazione della qualità dell'albero prodotto sul *training set* e sul *test set*. Quest'ultima valutazione sul *test set* viene fornita nel caso in cui viene esplicitamente richiesta dall'utente, e il file contenente i record del questo *test set* deve essere fornito al programma. La figura 8.5 presenta un esempio di output fornito da C4.5. Si tratta in realtà dell'output ottenuto costruendo un albero di decisione sul livello di coscienza di un paziente, sul quale ritorneremo più avanti nella parte di analisi dei dati dei pazienti. Il file di descrizione delle classi e degli attributi chiamato `paziente_liv_cosc.names` è quello presentato nella figura 8.3, che contiene 9 attributi. Come si può vedere nell'output, il file utilizzato per il *training set* (`paziente_liv_cosc.data`) contiene 20 record e quello usato per il *test set* (`paziente_liv_cosc.test`) ne contiene 9.

```
C4.5 [release 8] decision tree generator    Thu Apr 28 19:53:35 2011
----------------------------------------

   Options:
      File stem <patient_liv_cosc>
      Trees evaluated on unseen cases

Read 20 cases (9 attributes) from patient_liv_cosc.data

Decision Tree:

pa_max <= 115 : bad (4.0/1.0)
pa_max > 115 : good (16.0)


Tree saved


Evaluation on training data (20 items):

       Before Pruning          After Pruning
      ----------------   --------------------------
      Size     Errors   Size     Errors   Estimate

        3    1( 5.0%)     3    1( 5.0%)    (17.6%)   <<

Evaluation on test data (9 items):

       Before Pruning          After Pruning
      ----------------   --------------------------
      Size     Errors   Size     Errors   Estimate

        3    1(11.1%)     3    1(11.1%)    (17.6%)   <<


       (a)  (b)    <-classified as
      ---- ----
        7    1    (a): class good
             1    (b): class bad
```

Figura 8.5: Esempio di output fornito da C4.5

Si vede che il programma si è fermato dopo uno split sull'attributo *pa_max* (pressione arteriosa massima) generando di conseguenza un albero composto da 3 nodi di cui 1 è la radice dell'albero e 2 sono le foglie. I numeri racchiusi tra parentesi accanto alle foglie stanno a indicare il numero di record del *training set* che sono stati associati a ogni foglia e il numero di record, classificati erroneamente dalla foglia stessa. L'albero ottenuto può essere interpretato letteralmente come segue: *se un paziente ha* $pa\_max \leq 115$ *allora il suo stato di coscienza sarà "non buono" e nel caso contrario, esso sarà "buono"*. Il programma fornisce anche un albero semplificato (prodotto dopo la fase di *pruning*) ma in questo caso esso coincide con l'albero originale (prodotto prima della fase di *pruning*).
Dopo l'albero generato, viene fornita una valutazione dell'albero sul *training set*. Si vede che l'albero prodotto, costituito da 3 nodi (una radice e 2 foglie), classifica erroneamente 1 record su un totale di 20 forniti nel *training set*. Lo stesso risultato si ottiene prima e dopo la fase di *pruning* essendo i relativi alberi identici. Il programma predice un *Error Rate* del 17,6%.
La parte del risultato presentata finora non fa uso del file di *test set*: il sistema lo carica a questo punto e fornisce conseguentemente una valutazione dell'albero sul *test set*. L'albero originale (identico all'albero semplificato) classifica erroneamente 1 record su 9 forniti. In questo caso si vede anche che l'*Error Rate* coincide con quello predetto dal sistema. La parte finale dell'output presenta la *confusion matrix* ottenuta con il *test set* sull'albero semplificato (identico in questo caso all'albero originale): ci sono 8 record che appartengono alla classe *"good"* dei quali 7 vengono classificati correttamente come *"good"* mentre 1 è classificato erroneamente come *"bad"*. Similmente, c'è 1 record che appartiene alla classe *"bad"* ed esso viene classificato correttamente come *"bad"*.

8.4 Analisi dei dati dei pazienti con C4.5 e risultati ottenuti

Come anticipato nella sezione 3.2.4, i dati dei pazienti vengono analizzati con l'obiettivo di costruire dei predittori sull'andamento dello stato dei

parametri principali degli score dei pazienti a partire dai parametri registrati al momento dell'ingresso del paziente stesso in RR. In questa analisi i parametri per cui si vogliono costruire dei predittori sono quelli presenti nella scala a punteggio utilizzata attualmente all'Ospedale dell'Angelo di Mestre (vedere figura 2.2). Questa scala è composta dai parametri principali seguenti: *"Livello di coscienza"*, *"Respirazione"*, *"Emodinamica"*, *Dolore*, *PONV*, *Brivido*.
Per ognuno dei parametri, sono state definite due classi: *"good"* (parametro con stato buono) e *"bad"* (parametro con stato non buono). Un parametro di un paziente è considerato *"good"* se tale parametro ha sempre avuto uno score massimo in tutte le valutazioni registrate sul paziente durante il suo ricovero in RR; altrimenti tale parametro viene considerato *"bad"*. Si deve quindi costruire un'albero di decisione per ciascuno dei parametri.

8.4.1 Data set

All'arrivo dei pazienti in RR, sono registrate le seguenti informazioni: Nome, Data di nascita, Reparto di degenza, Intervento, Anestesista, Anestesia, ASA, Allergie, Ora di Arrivo, SpO2, PA, FC, T°, Bromage Scale. Tra queste informazioni, risulta chiaro che il nome del paziente, il nome dell'anestesista che si è occupato di lui e l'ora di arrivo non possono influire sullo stato dei parametri degli score. La temperatura all' arrivo, invece, non è stata considerata come parametro importante perché consultando le schede dei pazienti disponibili ci si è resi conto che tale parmetro non era quasi mai compilato perciò non poteva essere utile per l'analisi dei dati. Discutendo con i medici, ci si è resi conto che anche le allergie non possono influire sullo stato dei parametri degli score. Di conseguenza le seguenti informazioni sono state considerate nella costituzione degli attributi del data set:

1. *Data di Nascita*: da questa informazione si è ideato l'attributo `age` per specificare l'età di un paziente. Questo attributo poteva essere considerato semplicemente come uno di tipo numerico (intero positivo in questo caso) ma con l'aiuto dei medici è stato reso categorico facendo un raggruppamento in classi di età (questo raggruppamento migliora sicuramente i risultati).Sono state delineate queste classi di età:

- Classe 1: meno di 1 anno.
- Classe 2: tra 1 e 3 anni.
- Classe 3: tra 4 e 14 anni (termine dell'età pediatrica).
- Classe 4: tra 15 e 40 anni.
- Classe 5: tra 41 e 60 anni.
- Classe 6: tra 61 e 80 anni.
- Classe 7: più di 80 anni.

Visto che i bambini di meno di 1 anno non passano in RR, l'attributo `age` potrà prendere i valori *first* (per pazienti della classe 2), *second* (per pazienti della classe 3), *third* (classe 4), *fourth* (classe 5), *fifth* (classe 6) e *sixth* (classe 7).

2. *Reparto*: da questa informazione si è ideato l'attributo `unit`. Visto che i reparti dell'ospedale sono in numero ridotto e finito, questo attributo è di tipo categorico nominale e prende i valori *TOR* (per il reparto di chirurgia toracica), *ONC* (per il reparto di chirurgia generale e oncologica), *MAXI* (per la chirurgia maxillofacciale), *PED* (per la chirurgia pediatrica), *PLA* (per la chirurgia plastica), *VASC* (per la chirurgia vascolare), *OCU* (per la chirurgia oculistica), *ORL* (per la chirurgia otorinolaringoiatriaca), *GINE* (per la chirurgia ginecologica), *NCH* (per la neurochirurgia), *ORT* (per la chirurgia ortopedica), *CAR* (per la cardiochirurgia) e *URO* (per la chirurgia urologica).

3. *Anestesia*: da questa informazione si è ideato l'attributo `anesthesia` che, similmente all'attributo `unit` prende i valori *GEN* (per anestesia generale), *PLE* (per anestesia plessica), *SUB* (per anestesia subaracnoidea), *PER* (per anestesia peridurale), *LOC* (per anestesia locale), *SED* (per sedazione), *ANA* (per analgosedazione) e *COM* (per anestesia combinata).

4. *ASA*: da questa informazione si è ideato l'attributo `asa` di tipo categorico che prende un valore intero che va da 1 a 5, che sono i valori che può avere il parametro ASA di un paziente. Si poteva considerare questo parametro come numerico, ma visto che i valori che esso può

```
good, bad.

age:          first, second, third, fourth, fifth, sixth
unit:         TOR,ONC,MAXI,PED,PLA,VASC,OCU,ORL,GINE,NCH,ORT,CAR,URO
anesthesia:   GEN,PLE,SUB,PER,LOC,SED,ANA,COM
asa:          1,2,3,4,5
spo2:         continuous
pa_max:       continuous
pa_min:       continuous
fc:           continuous
bromage:      0,1,2,3
```

Figura 8.6: File di definizione delle classi e attributi per i classificatori

prendere non sono molti, si è preferito considerarlo come categorico; questo facilita sicuramente lo split.

5. *Pressione arteriosa (PA)*: da questa informazione si sono ideati gli attributi `pa_max` e `pa_min`, per specificare rispettivamente la pressione arteriosa massima e quella minima. Questi attributi sono di tipo numerico e possono prendere dei valori compresi tra 0 e 200.

6. *Frequenza cardiaca (FC)*: da questa informazione si è ideato l'attributo `fc` che è di tipo numerico e può prendere un valore compreso tra 65 e 180.

7. *Bromage Scale*: da questa informazione si è ideato l'attributo `bromage` di tipo categorico che può prendere un valore compreso tra 0 e 3. La scelta di considerare questo attributo come categorico viene dalle stesse motivazioni illustrate per l'attributo `asa`.

Il tipo di intervento non è stato incluso tra gli attributi a causa della numerosità dei casi che esistono. Sarebbe sicuramente stato un parametro importante per l'analisi dei dati ma è risultato molto difficile includerlo in quanto non si è riuscito a raggruppare i tipi di interventi in classi di interventi. E inoltre il numero di schede paziente disponibili era troppo ridotto per avere un numero sufficiente di casi per ciascun intervento.
Per ognuno dei classificatori da costruire, il file di definizione delle classi e degli attributi è quello presentato nella figura 8.6.

Per costruire i classificatori, si è riusciti a ottenere un *data set* di 46 schede di pazienti dell'Ospedale di Mestre. Queste schede sono poche per raggiungere gli obiettivi fissati.
La costruzione si è fatta usando due modi: utilizzando prima tutti i 46 record del *data set* come *training set* senza usare un *test set*; e dopo si sono utilizzati 34 record nel *training set* e 12 record nel *test set*. In entrambi i casi, i risultati ottenuti sono stati quasi identici.
I dati statistici del *data set* sono presentati nella tabella 8.2.

	Data Set		**Training Set**		**Test Set**	
	good	*bad*	*good*	*bad*	*good*	*bad*
Dolore	26	20	19	15	7	5
Emodinamica	43	3	32	2	11	1
Livello Coscienza	34	12	26	8	8	4
PONV	45	1	33	1	12	0
Respirazione	37	9	28	6	9	3
Brivido	39	7	30	4	9	3

Tabella 8.2: Statistica del data set usato per i classificatori

8.4.2 Classificatori ottenuti

Livello di coscienza

Dalle figure 8.7 e 8.8, il classificatore predice che: se PA_MAX è maggiore di 120 allora lo stato di coscienza sarà *good*, altrimenti sarà *bad*.
Secondo i medici, questo potrebbe essere una cosa sensata, nel senso che l'ipotensione di per sè può essere causa di alterazioni dello stato di coscienza. Infatti, nel caso di emorragia, un grado elevato di shock compromette (oltre vari altri parametri) anche lo stato di coscienza. Ad esempio, una classe di shock III, in cui il paziente ha perso più del 30% del suo volume ematico, può presentarsi anche con uno stato di ansia/confusione, se ha perso oltre il 40% il paziente può essere anche comatoso. Il problema qui è il 120 di pressione che non rappresenta un cut-off corretto, mentre lo è di più un 90.

```
C4.5 [release 8] decision tree generator     Wed May 18
18:37:12 2011
----------------------------------------

   Options:
      File stem <patient_liv_cosc>

Read 46 cases (9 attributes) from patient_liv_cosc.data

Decision Tree:

pa_max <= 120 :
|   pa_min <= 51 : good (2.1/0.0)
|   pa_min > 51 :
|   |   age = first: bad (0.0)
|   |   age = second: bad (0.2)
|   |   age = third: bad (2.2/0.2)
|   |   age = fifth: bad (2.0)
|   |   age = sixth: bad (0.0)
|   |   age = fourth:
|   |   |   fc <= 75 : bad (3.0)
|   |   |   fc > 75 : good (3.2/1.0)
pa_max > 120 :
|   age = first: good (0.0)
|   age = second: bad (0.7/0.0)
|   age = third: good (8.0/0.0)
|   age = fourth: good (11.1/1.0)
|   age = fifth: good (8.2/0.0)
|   age = sixth:
|   |   unit = TOR: good (0.0)
|   |   unit = ONC: good (0.0)
|   |   unit = MAXI: good (0.0)
|   |   unit = PED: good (0.0)
|   |   unit = PLA: good (0.0)
|   |   unit = VASC: good (0.2)
|   |   unit = OCU: good (0.0)
|   |   unit = ORL: good (0.0)
|   |   unit = GINE: good (0.0)
|   |   unit = NCH: good (1.0)
|   |   unit = ORT: bad (2.0)
|   |   unit = CAR: good (0.0)
|   |   unit = URO: good (2.0)


Simplified Decision Tree:

pa_max > 120 : good (33.2/5.7)
pa_max <= 120 :
|   pa_min <= 51 : good (2.1/1.1)
|   pa_min > 51 : bad (10.7/4.1) <--


Tree saved


Evaluation on training data (46 items):

       Before Pruning           After Pruning
      ----------------   ---------------------------
      Size      Errors   Size      Errors   Estimate

       32    2( 4.3%)      5    6(13.0%)    (23.6%)   <<
```

Figura 8.7: Albero per il livello di coscienza ottenuto senza test set

```
C4.5 [release 8] decision tree generator     Thu May 19 19:12:29
2011
----------------------------------------

   Options:
      File stem <patient_liv_cosc>
      Trees evaluated on unseen cases

Read 34 cases (9 attributes) from patient_liv_cosc.data

Decision Tree:

pa_max <= 120 :
|   age = first: bad (0.0)
|   age = second: bad (0.2)
|   age = third: bad (2.2/0.2)      <--
|   age = fourth: good (3.2/1.0)
|   age = fifth: bad (2.0)     <--
|   age = sixth: bad (0.0)
pa_max > 120 :
|   anesthesia = PLE: good (0.0)
|   anesthesia = SUB: bad (1.0/0.0)
|   anesthesia = PER: good (0.0)
|   anesthesia = LOC: good (0.0)
|   anesthesia = SED: good (0.0)
|   anesthesia = ANA: good (0.0)
|   anesthesia = COM: good (3.1)
|   anesthesia = GEN:
|   |   age = first: good (0.0)
|   |   age = second: bad (0.8/0.0)
|   |   age = third: good (6.9/0.0)
|   |   age = fourth: good (7.1/1.0)
|   |   age = fifth: good (5.2/0.0)
|   |   age = sixth: good (2.1/0.0)


Simplified Decision Tree:

pa_max <= 120 : bad (7.7/3.9)
pa_max > 120 : good (26.3/4.6)


Tree saved


Evaluation on training data (34 items):

       Before Pruning          After Pruning
      ----------------   ---------------------------
      Size     Errors   Size     Errors   Estimate

       23    2( 5.9%)     3    5(14.7%)    (25.1%)   <<

Evaluation on test data (12 items):

       Before Pruning          After Pruning
      ----------------   ---------------------------
      Size     Errors   Size     Errors   Estimate

       23    6(50.0%)     3    3(25.0%)    (25.1%)   <<


       (a)  (b)    <-classified as
      ---- ----
        6    2    (a): class good
        1    3    (b): class bad
```

Figura 8.8: Albero per il livello di coscienza ottenuto con test set

```
C4.5 [release 8] decision tree generator    Wed May 18
18:33:24 2011
----------------------------------------

   Options:
      File stem <patient_emo>

Read 46 cases (9 attributes) from patient_emo.data

Decision Tree:

spo2 > 93 : good (40.9)
spo2 <= 93 :
|   pa_max <= 115 : bad (2.6/0.1)
|   pa_max > 115 : good (2.6/0.5)


Tree saved


Evaluation on training data (46 items):

      Before Pruning          After Pruning
      ----------------  --------------------------
      Size      Errors   Size      Errors   Estimate

        5    0( 0.0%)     5    0( 0.0%)    ( 8.7%)   <<
```

Figura 8.9: Albero per l'emodinamica ottenuto senza test set

Emodinamica

Come illustrate nelle figure 8.9 e 8.10, il classificatore predice che: l'emodinamica sarà NON-OK se la saturazione dell'ossigeno è bassa ($\leq 93\%$). Le cose vanno ancora peggio se inoltre PA_MAX è bassa (≤ 115).
Secondo i medici, la saturazione è influenzata dalla pressione arteriosa, non viceversa: se un paziente è ipoteso la saturazione potrebbe essere bassa.
Il classificatore non è in contrasto con quello che osservano i medici: il classificatore considera più importante la bassa SpO2, e infatti i medici dicono che non necessariamente una bassa pressione implica una bassa saturazione. Il fatto di avere solo 3 record nella classe NON-OK nel *Data Set* non aiuta a catturare bene la realtà clinica.

Dolore

Dai risultati presentati nelle figure 8.11 e 8.12, il classificatore predice che: il paziente avrà dolore se proviene dai reparti di neurochirurgia, di chirurgia

```
C4.5 [release 8] decision tree generator    Thu May 19 19:10:37
2011
----------------------------------------

   Options:
      File stem <patient_emo>
      Trees evaluated on unseen cases

Read 34 cases (9 attributes) from patient_emo.data

Decision Tree:

spo2 <= 92 : bad (2.1/0.1)
spo2 > 92 : good (31.9)


Tree saved


Evaluation on training data (34 items):

      Before Pruning          After Pruning
      ----------------  ---------------------------
      Size      Errors   Size      Errors   Estimate

        3    0( 0.0%)      3    0( 0.0%)    ( 7.1%)   <<

Evaluation on test data (12 items):

      Before Pruning          After Pruning
      ----------------  ---------------------------
      Size      Errors   Size      Errors   Estimate

        3    2(16.7%)      3    2(16.7%)    ( 7.1%)   <<


       (a)  (b)    <-classified as
       ----  ----
        10    1    (a): class good
         1         (b): class bad
```

Figura 8.10: Albero per l'emodinamica ottenuto con test set

generale (con fc > 80 oppure SpO2 ≤ 99), di ginecologia (con SpO2 ≤ 99) o di ortopedia (con PA_Max ≤ 141). In tutti gli altri casi il paziente non avrà dolore.
Secondo i medici, questo è un dato interessante: la tipologia di intervento e la sede hanno una diretta ripercussione sull'incidenza del dolore, questo li potrebbe aiutare a capire in quali interventi non si ottiene un buon controllo del dolore postoperatorio. A prescindere dal tipo di intervento, il dovere dei medici è garantire anche l'analgesia (non solo la sopravvivenza) per cui loro potrebbero capire dove sbagliano e approntare delle modifiche alla loro analgesia intra e postoperatoria.
Il tipo di intervento sarebbe stato sicuramente un parametro rilevante per la costruzione di un albero di decisione per il dolore.

Respirazione

Dai risultati presentati nelle figure 8.13 e 8.14, si può dire che il classificatore riesce solo a delineare il fatto ovvio che un paziente con bassa saturazione d'ossigeno ha problemi respiratori. Servirebbero sicuramente molti più dati per sperare di ottenere un'informazione più interessante. Anche in questo caso il classificatore è riuscito a catturare solo un fenomeno di natura molto generale.

Brivido

Dai risultati presentati nelle figure 8.15 e 8.16, il classificatore predice che: il paziente avrà brivido se PA_MIN > 85 oppure se il paziente ha subito un'anestesia generale e proviene dal reparto di ortopedia.
Secondo i medici, l'ipotensione è fattore di rischio per il brivido, ma non il contrario. Qui i pochi dati a disposizione dicono esattamente il contrario di ciò che avviene nella realtà. Il classificatore comunque riesce a catturare il fatto che il brivido sia più comune al risveglio dell'anestesia generale, che è un fatto noto.

```
C4.5 [release 8] decision tree generator     Wed May 18
18:42:56 2011
--------------------------------------------

   Options:
      File stem <patient_dolore>

Read 46 cases (9 attributes) from patient_dolore.data

Decision Tree:

unit = TOR: good (2.0)
unit = MAXI: good (2.0)
unit = PED: good (0.0)
unit = PLA: good (3.0/1.0)
unit = VASC: good (2.0)
unit = OCU: good (2.0)
unit = NCH: bad (4.0)
unit = CAR: good (0.0)
unit = URO: good (3.0/1.0)
unit = ONC:
|   fc <= 80 : good (4.0)
|   fc > 80 : bad (6.0/1.0)
unit = ORL:
|   spo2 <= 97 : good (3.0)
|   spo2 > 97 : bad (2.0)
unit = GINE:
|   spo2 <= 99 : bad (2.0)
|   spo2 > 99 : good (5.0/1.0)
unit = ORT:
|   pa_max <= 141 : bad (4.0)
|   pa_max > 141 : good (2.0)


Tree saved


Evaluation on training data (46 items):

      Before Pruning           After Pruning
      ----------------   ---------------------------
      Size      Errors   Size      Errors   Estimate

        22    4( 8.7%)     22    4( 8.7%)    (44.2%)   <<
```

Figura 8.11: Albero per il dolore ottenuto senza test set

```
C4.5 [release 8] decision tree generator    Thu May 19 19:07:07
2011
--------------------------------------------

   Options:
      File stem <patient_dolore>
      Trees evaluated on unseen cases

Read 34 cases (9 attributes) from patient_dolore.data

Decision Tree:

unit = TOR: good (2.0)
unit = MAXI: good (1.0)
unit = PED: good (0.0)
unit = PLA: good (1.0)
unit = VASC: good (1.0)
unit = OCU: good (2.0)
unit = ORL: good (3.0/1.0)
unit = NCH: bad (4.0)
unit = CAR: good (0.0)
unit = URO: good (1.0)
unit = ONC:
|   spo2 <= 99 : bad (3.4)
|   spo2 > 99 : good (4.6/0.6)
unit = GINE:
|   spo2 <= 99 : bad (2.0)
|   spo2 > 99 : good (4.0/1.0)
unit = ORT:
|   pa_max <= 141 : bad (3.0)
|   pa_max > 141 : good (2.0)


Simplified Decision Tree:

spo2 <= 99 : bad (18.5/10.0)
spo2 > 99 : good (15.5/6.3)


Tree saved


Evaluation on training data (34 items):

      Before Pruning           After Pruning
      ----------------   ---------------------------
      Size      Errors   Size      Errors   Estimate

       20    3( 8.8%)     3   13(38.2%)    (47.9%)   <<

Evaluation on test data (12 items):

      Before Pruning           After Pruning
      ----------------   ---------------------------
      Size      Errors   Size      Errors   Estimate

       20    4(33.3%)     3    6(50.0%)    (47.9%)   <<


       (a)  (b)    <-classified as
       ---- ----
         2    5    (a): class good
         1    4    (b): class bad
```

Figura 8.12: Albero per il dolore ottenuto con test set

```
C4.5 [release 8] decision tree generator    Thu May 19 18:12:14
2011
--------------------------------------

   Options:
      File stem <patient_resp>

Read 46 cases (9 attributes) from patient_resp.data

Decision Tree:

unit = TOR: good (2.0)
unit = ONC: good (10.0/2.0)
unit = MAXI: good (2.0/1.0)
unit = PED: good (0.0)
unit = PLA: good (3.0)
unit = VASC: good (2.0)
unit = OCU: good (2.0/1.0)
unit = ORL: good (5.0/1.0)
unit = GINE: good (7.0)
unit = NCH: good (4.0)
unit = CAR: good (0.0)
unit = URO: good (3.0)
unit = ORT:
|   spo2 <= 98 : bad (3.0/1.0)
|   spo2 > 98 : good (3.0)


Simplified Decision Tree:
 good (46.0/9.4)


Tree saved


Evaluation on training data (46 items):

       Before Pruning           After Pruning
      ----------------   ---------------------------
      Size      Errors   Size      Errors   Estimate

        16    6(13.0%)     1    7(15.2%)    (20.4%)   <<
```

Figura 8.13: Albero per la respirazione ottenuto senza test set

```
C4.5 [release 8] decision tree generator    Sat May 21 14:55:37 2011
-------------------------------------------

    Options:
        File stem <patient_resp>
        Trees evaluated on unseen cases

Read 34 cases (9 attributes) from patient_resp.data

Decision Tree:

spo2 <= 92 : bad (2.1/0.1)
spo2 > 92 : good (31.9/4.0)


Tree saved


Evaluation on training data (34 items):

        Before Pruning          After Pruning
       ----------------   ---------------------------
       Size      Errors   Size      Errors   Estimate

         3    4(11.8%)      3    4(11.8%)    (20.9%)   <<

Evaluation on test data (12 items):

        Before Pruning          After Pruning
       ----------------   ---------------------------
       Size      Errors   Size      Errors   Estimate

         3    4(33.3%)      3    4(33.3%)    (20.9%)   <<


        (a)  (b)    <-classified as
       ---- ----
         8    1    (a): class good
         3         (b): class bad
```

Figura 8.14: Albero per la respirazione ottenuto con test set

```
C4.5 [release 8] decision tree generator     Wed May 18
18:48:19 2011
--------------------------------------------

   Options:
      File stem <patient_briv>

Read 46 cases (9 attributes) from patient_briv.data

Decision Tree:

unit = TOR: good (2.0)
unit = ONC: good (10.0/1.0)
unit = MAXI: good (2.0)
unit = PED: good (0.0)
unit = PLA: good (3.0/1.0)
unit = VASC: good (2.0)
unit = OCU: good (2.0)
unit = ORL: good (5.0)
unit = GINE: good (7.0/1.0)
unit = NCH: good (4.0)
unit = CAR: good (0.0)
unit = URO: good (3.0/1.0)
unit = ORT:
|   age = first: good (0.0)
|   age = second: good (0.0)
|   age = third: bad (2.0)
|   age = fourth: good (0.0)
|   age = fifth: good (2.0/1.0)
|   age = sixth: good (2.0)


Simplified Decision Tree:
 good (46.0/9.4)


Tree saved


Evaluation on training data (46 items):

       Before Pruning          After Pruning
      ----------------   ---------------------------
      Size      Errors   Size      Errors   Estimate

        20    5(10.9%)      1    7(15.2%)    (20.4%)   <<
```

Figura 8.15: Albero per il brivido ottenuto senza test set

```
C4.5 [release 8] decision tree generator     Thu May 19 19:03:39 2011
----------------------------------------

   Options:
      File stem <patient_briv>
      Trees evaluated on unseen cases

Read 34 cases (9 attributes) from patient_briv.data

Decision Tree:

pa_min > 85 : bad (2.2/0.2)
pa_min <= 85 :
|   anesthesia = PLE: good (0.0)
|   anesthesia = SUB: good (1.0/0.0)
|   anesthesia = PER: good (0.0)
|   anesthesia = LOC: good (0.0)
|   anesthesia = SED: good (0.0)
|   anesthesia = ANA: good (0.0)
|   anesthesia = COM: good (5.2/1.2)
|   anesthesia = GEN:
|   |   unit = TOR: good (2.0)
|   |   unit = ONC: good (4.9)
|   |   unit = MAXI: good (1.9)
|   |   unit = PED: good (0.0)
|   |   unit = PLA: good (0.0)
|   |   unit = VASC: good (2.0)
|   |   unit = OCU: good (2.0)
|   |   unit = ORL: good (3.0)
|   |   unit = GINE: good (5.0)
|   |   unit = NCH: good (3.0)
|   |   unit = ORT: bad (0.8)
|   |   unit = CAR: good (0.0)
|   |   unit = URO: good (1.0)


Simplified Decision Tree:

pa_min <= 85 : good (31.8/3.8)
pa_min > 85 : bad (2.2/1.2)


Tree saved


Evaluation on training data (34 items):

       Before Pruning          After Pruning
      ----------------   ---------------------------
      Size      Errors   Size      Errors   Estimate

       24    1( 2.9%)     3    2( 5.9%)    (14.6%)   <<

Evaluation on test data (12 items):

       Before Pruning          After Pruning
      ----------------   ---------------------------
      Size      Errors   Size      Errors   Estimate

       24    2(16.7%)     3    3(25.0%)    (14.6%)   <<


       (a)  (b)    <-classified as
      ----  ----
        9          (a): class good
        3          (b): class bad
```

Figura 8.16: Albero per il brivido ottenuto con test set

```
C4.5 [release 8] decision tree generator    Wed May 18
18:44:54 2011
----------------------------------------

   Options:
      File stem <patient_ponv>

Read 46 cases (9 attributes) from patient_ponv.data

Decision Tree:
 good (46.0/1.0)


Tree saved


Evaluation on training data (46 items):

       Before Pruning          After Pruning
      ----------------  ---------------------------
      Size      Errors   Size      Errors   Estimate

        1    1( 2.2%)     1    1( 2.2%)    ( 5.6%)   <<
```

Figura 8.17: Albero per il PONV ottenuto senza test set

PONV

Dai risultati presentati nelle figure 8.17 e 8.18, si può concludere che il classificatore non riesce a costruire un albero per il PONV. In realtà, il *Data Set* contiene un solo caso di paziente con PONV *bad*, per cui il classificatore considera il PONV buono per tutti i pazienti.
L'albero costruito con il *Test Set* differenzia anche i casi a seconda del parametro ASA, ma secondo i medici, questo parametro non è rilevante. Servirebbero più casi di pazienti con PONV *bad* per sperare in un risultato significativo.

```
C4.5 [release 8] decision tree generator    Sat May 21 14:14:23 2011
-------------------------------------------

   Options:
      File stem <patient_ponv>
      Trees evaluated on unseen cases

Read 34 cases (9 attributes) from patient_ponv.data

Decision Tree:

asa = 1: good (9.5)
asa = 2: good (16.3/0.0)
asa = 4: good (0.0)
asa = 5: good (0.0)
asa = 3:
|   age = first: good (0.0)
|   age = second: good (0.0)
|   age = third: bad (2.0/1.0)
|   age = fourth: good (0.8)
|   age = fifth: good (3.1)
|   age = sixth: good (2.3)


Simplified Decision Tree:
 good (34.0/2.6)


Tree saved


Evaluation on training data (34 items):

       Before Pruning          After Pruning
      ----------------   ---------------------------
      Size      Errors   Size      Errors   Estimate

        12    0( 0.0%)     1    1( 2.9%)    ( 7.5%)   <<

Evaluation on test data (12 items):

       Before Pruning          After Pruning
      ----------------   ---------------------------
      Size      Errors   Size      Errors   Estimate

        12    0( 0.0%)     1    0( 0.0%)    ( 7.5%)   <<


        (a)  (b)    <-classified as
       ----  ----
         12         (a): class good
                    (b): class bad
```

Figura 8.18: Albero per il PONV ottenuto con test set

Capitolo 9

Conclusioni e sviluppi futuri

9.1 Conclusioni

In questa tesi si è presentata la soluzione progettata e sviluppata per permettere il miglioramento della fase di raccolta dei dati dei pazienti nella *Recovery Room* (o sala di risveglio postoperatorio) dell'Ospedale dell'Angelo di Mestre, e per permettere ai medici di tenere un archivio di questi dati; cosa che non si poteva fare prima di questo lavoro. Successivamente si è presentata anche l'analisi effettuata sui dati; parte che può essere considerata come un'inizio di un lavoro molto più grande che si potrebbe portare avanti per aiutare i medici a prevedere l'andamento dello stato dei pazienti durante il ricovero non appena questi ultimi entrano in sala.
L'applicazione mobile sviluppata per la raccolta dei dati è stata realizzata in modo da rendere l'interfaccia utente la più intuitiva possibile e di semplice utilizzo. Una grande parte della sua progettazione è stata dedicata alla suddivisione dei dati in view compilabili. Queste view sono state organizzate in modo da rispondere alle fasi cliniche applicate nella *Recovery Room* e limitare il numero di accessi al database in fase di salvatagggio delle informazioni inserite in esse: questo per non compromettere la reattività delle view durante la navigazione nelle schede dei pazienti ricoverati.
La scheda cartacea che viene generata e stampata in automatico quando un paziente viene dimesso, è stata strutturata in modo da rispecchiare quella precedentemente in uso nella sala e soprattutto fornire almeno le stesse in-

formazioni che essa forniva.
Con tutto questo, si può dire che la soluzione proposta ha sicuramente portato dei miglioramenti al processo di raccolta precedentemente utilizzato in *Recovery Room*.

9.2 Sviluppi futuri

Per quel che riguarda la soluzione composta dall'applicazione mobile, il server dei dati e la stampante, si può dire che essa è stata pensata per rimanere stabile nel tempo. Non si escludono comunque dei miglioramenti che possono essere portati sull'aspetto grafico dei layout delle view e che possano venire incontro alle eventuali nuove esigenze degli utenti, visto che l'interfaccia generale è stata pensata soprattutto per essere intuitiva e reattiva ed è stata realizzata con lo spirito di essenzialità ed efficacia; non si è lavorato molto sulla sua bellezza grafica.
Per quanto riguarda invece la parte di analisi dei dati, si è visto che si è riusciti a catturare solo fenomeni di natura molto generale. Ci sarebbe ancora molto da fare. Un primo sviluppo sarebbe sicuramente aumentare il *Data Set*; questo sarà più facile da fare una volta che l'applicazione mobile verrà adottata come strumento di raccolta dei dati. Un secondo sviluppo sarebbe riflettere su come introdurre il tipo di intervento tra i parametri dell'analisi. Aggiungere il parametro *intervento* di tipo categorico con più di mille valori nel suo dominio non sarebbe sicuramente una buona soluzione (visto che i tipi di interventi sono tantissimi). Servirebbe magari realizzare un *clustering* sui tipi di interventi in modo da costituire un dominio più ridotto dei valori che quel attributo potrà prendere. Questo richiederebbe sicuramente un significativo contributo dei medici.

Listings

Elenco delle tabelle

Elenco delle figure

Bibliografia

[1] Pedersen T, Møller AM, Hovhannisyan K. *Pulse oximetry for perioperative monitoring*. Cochrane Database Syst Rev 7 ottobre 2009 ;(4):CD002013.

[2] Gruppo di Studio SIAARTI per la Sicurezza in Anestesia e Terapia Intensiva, *Raccomandazioni per l'area di recupero e l'assistenza post-anestesiologica*. Milano (2010).

[3] http://source.android.com/license

[4] http://www.apache.org/licenses/LICENSE-2.0

[5] http://www.gnu.org/licenses/old-licenses/gpl-2.0.html

[6] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers. (1993).

[7] Hunt, Marin, and Stone. *Experiments in Induction*. New York: Academic Press. (1966)

[8] White PF, Song D. *New criteria for fast-tracking after outpatient anesthesia: a comparison with the modified Aldrete's scoring system*. Anesth Analg 1999;88:1069-72.

[9] http://www.sqlite.org/

Ringraziamenti

Vorrei ringraziare il Prof. Carlo FANTOZZI per i suoi consigli e per tutto il tempo dedicato a me durante lo svolgimento del mio progetto di fine studio e la scrittura di questa tesi.
Ringrazio lo staff della Recovery Room dell'Opsedale dell'Angelo di Mestre per il loro contributo durante lo svolgimento del mio progetto, in particolare il mio correlatore l'anestesista Dott. Alberto GRASSETTO.
Ringrazio tutto il personale docente dell'Università di Padova e dell'Ecole Centrale de Lyon che ho incontrato durante questo mio percorso accademico per il loro contributo alla mia formazione.
Ringrazio la mia famiglia che sempre mi è stata accanto, che mi ha insegnato a navigare nelle molte strade della vita ed a non scoraggiarmi, anche se oggi si trova a più chilometri da me.
Ringrazio la famiglia SESSUA oggi in Canada e la famiglia DANCHIO di Padova per i loro consigli e il loro sostegno fin dall'inizio del mio percorso accademico all'Università di Padova.
Ringrazio tutti quelli che mi hanno aiutato nella stesura di questa tesi, in particolare Francesca PELLICCI.
Ringrazio tutti gli amici che mi hanno sopportato e che mi sono sempre stati vicini durante questi ultimi anni, in particolare Christian, Fabrice, Francis, Yannick.
Ringrazio tutti i compagni di classe che mi hanno sopportato e accompagnato in questo percorso non sempre facile fin dal primo anno all'Università di Padova, in particolare Elia, Francesco, Stefano.
Ringrazio infine le ragazze che hanno fatto parte della mia vita nel corso di questi anni per l'affetto che hanno testimoniato nei miei confronti.