Ingegneria del software

Elaborato progetto finale

Autori:
Lorenzo Gasparini
Andrea Pellizzari

Indice

Prefazione

0.1 Introduzione

Tale documento si pone lo scopo di raccogliere specifiche, documentazione, nonchè l'integrazione data dalla sintesi delle attività svolte durante la fase di progettazione e sviluppo del progetto conclusivo del modulo "Ingegneria del software" per l'anno accademico 2024-2025, tenuto presso l'*Università degli studi di Verona* dal professor Carlo Combi. Il progetto è stato redatto interamente da un gruppo di due studenti frequentanti il corso di studi in *Informatica* presso la stessa. Si elencano i componenti del gruppo e le rispettive matricole:

- Lorenzo Gasparini - VR500420;
- Andrea Pellizzari - VR502128;

Il documento, è da intendersi come un lavoro di approfondimento/relazione basato sui concetti più largamente trattati nel corso di "*Ingegneria del software*". Quanto formulato di seguito non è stato sottoposto a revisione o supervisione alcuna da parte dei professori responsabili della didattica del corso stesso. La traccia scelta per lo svolgimento della prova di esame è "Esercizio 3 - Progettazione di un sistema di telemedicina di un servizio clinico per la gestione di pazienti diabetici".

0.2 Bibliografia

I principi teorici utilizzati nella fase di redazione del documento, ed in generale nella fase di strutturazione del progetto finale, fanno riferimento alla bibliografia del corso stesso, segue un elenco sommario dei libri di testo che la compongono. Si rimanda alla sezione "*Riferimenti bibliografici*" per eventuali approfondimenti.

- [Som07] "Ingegneria del software", I. Sommerville;
- [Som21] "Introduzione all'ingegneria del software moderna", I. Sommerville;
- [Lar21] "Applicare UML e i pattern: analisi e progettazione orientata agli oggetti", C. Larman;
- [AA09] "Object-oriented systems analysis and design", Noushin Ashrafi;

Inoltre verranno utilizzate, come fondamento per alcuni argomenti della trattazione, dispense di varia natura e di differenti autori (opportunamente citate di seguito).

- [Pog15d] "Introduzione a UML", F. Poggi;
- [Pog15c] "Il diagramma dei casi d'uso", F. Poggi;
- [Com] "Sistemi orientati agli oggetti, Concetti di base", C. Combi;
- "Schede CRC e diagramma delle classi", F. Poggi;
- [Pog15a] "Diagrammi di sequenza", F. Poggi;
- [Pog15b] "I diagrammi di attività e stato", F. Poggi;
- Tutorialspoint Design patterns : [Tuth], [Tuta], [Tutb], [Tutd], [Tute], [Tutf], [Tutg], [Tutc];

0.3 Repository

La totalità dei materiali prodotti durante l'implementazione del progetto (ne fanno parte il codice sorgente e la documentazione stessa) sono pubblicamente condivisi mediante repository Git, visitabile seguendo il link: `https://github.com/lorenzogasparini/ProgettoIngegneriaSw`.

[1] Nota: Il volume *Ingegneria del software*, [Som07], è il libro di testo scelto per la didattica del corso stesso, *Ingegneria del software*.

Introduzione

1.1 Specifiche di progetto e raccolta dei requisiti

Il sistema si pone l'obiettivo di gestire un servizio clinico di telemedicina per pazienti diabetici (*diabete mellito di tipo 2*). Esso risulta essere una malattia metabolica cronica caratterizzata da alti livelli di glucosio nel sangue. Risultano quindi utenti del sistema i **pazienti** ed i **medici** (diabetologi - appartenenti ad un comune istituto di medicina -) che tengono in cura gli stessi. Il sistema dovrà permettere l'interazione di medici e pazienti rispetto ai dettagli che competono la gestione della patologia stessa. Il sistema sarà coordinato da utenti individuati come responsabili del servizio di telemedicina, essi inseriscono e gestiscono successivamente i dati iniziali di pazienti e medici necessari per l'autenticazione, verranno indicati nel seguito come **admin**. Ogni medico può analizzare ed aggiornare in qualsiasi momento i dati di ogni paziente autenticato nel sistema (tali dati includono la anagrafica del paziente stesso ed inoltre una breve anamnesi dello stesso, inoltre tali modifiche sono analizzabili e gestite dal responsabile del servizio (admin). Ogni utente che necessiterà di interagire con il sistema dovrà poter autenticarsi. La creazione effettiva del sistema compete direttamente agli utenti amministratori.

La funzionalità di vertice, su cui si basa il sistema, come anticipato precedentemente, è quella di gestione dei dettagli inerenti la cura delle patologie del paziente, legate al suo storico del diabete di tipo 2.

Di fatto, una delle funzionalità di base fornite dal sistema è quella di gestione delle *rilevazioni del* ***livello di glicemia nel sangue***, da effettuare potenzialmente più volte al giorno (la specifica definisce preferibilmente prima e dopo ogni pasto). Il paziente autenticato può quindi inserire la rilevazione del livello di glicemia ottenuto ed il medico potrà successivamente visualizzarle mediante opportuna interfaccia.

Il diabetologo, una volta autenticato, potrà specificare e gestire le ***terapie previste*** per i pazienti del sistema (da lui presi in cura oppure pazienti generici presi in cura da altri medici), dove per gestione si intende la supervisione della ***corretta assunzione di farmaci*** e di ***eventuali sintomi avvertiti*** dal paziente durante un arbitrario istante temporale, che chiaramente potrà interagire con il sistema per visualizzare ed inserire tali informazioni. Per ogni terapia dovrà essere possibile specificare il farmaco, il numero di assunzioni giornaliere, quantità di farmaco per assunzione ed eventuali indicazioni (ad es. dopo i pasti, lontano dai pasti, e così via). Si assume il seguente DETTAGLIO IMPLEMENTATIVO: al paziente può essere assegnato un solo farmaco per ogni terapia ad egli assegnata.

Il sistema dovrà effettuare la *verifica della* ***sicurezza e correttezza dei livelli di glicemia*** e del *rispetto delle* ***terapie interne*** ad esso assegnate da un medico interno (dettaglio implementativo: si assumono come terapie interne anche farmaci non strettamente legati alla gestione del diabete di tipo 2 ma anche patologie in comoribdità dello stesso, ciò implica quindi la gestione della coerenza della terapia prescritta rispetto alla patologia diagnosticata) e talvolta inviare una notifica nel caso in cui vi siano situazioni non in linea con quelle corrette. Ulteriore DETTAGLIO IMPLEMENTATIVO dedotto delle specifiche fornite è il seguente: il sistema gestisce le terapie sulla base del fatto che data una terapia, vi sia certamente una patologia (sia essa legata al diabete di tipo 2 o una patologia pregressa e/o comorbidità).

1.2 Specifica e ingegnerizzazione dei requisiti

Ribadendo le principali attività cardine di ogni processo di sviluppo software (che, dal punto di vista dell'ingegneria del software, è un insieme di attività che impiego per ottenere un dato risultato finale con l'evoluzione temporale), (nota che tipicamente si tratta di attività atomiche ma tuttavia a loro volta possono suddividersi in micro-attività) che risultano essere: **specifica** (definisce cosa si intende ottenere dal sistema), **progettazione e implementazione** (definisce l'organizzazione e

l'implementazione del sistema), **validazione** (fase di controllo nella quale si verifica se il prodotto ottiene quanto richiesto dall'utilizzatore), **evoluzione** (consiste nella modifica del sistema dovuta alla evoluzione dei bisogni dell'utilizzatore) procederemo nella documentazione delle sopra-citate fasi come descritto di seguito. Di fatto ci occuperemo nel presente capitolo e fino al capitolo sesto di questo lavoro di documentare le prime due componenti del processo software applicate al nostro sistema software. Mentre tratteremo le ultime due nel capitolo settimo.

Sulla base di quanto appena detto, la fase di specifica dei requisiti ricopre un ruolo di cruciale importanza, al punto che da essa è nata una ulteriore branca dello studio ingegneristico, detta appunto "ingegneria dei requisiti del processo". Una attenta analisi dei principi dell'ingegneria dei requisiti del processo ci permette di ottenere una prima distinzione tra i requisiti di un sistema software: requisiti funzionali e requisiti strutturali. Intraprendiamo la fase di specifica e ingegnerizzazione sulla base dei principi teorici appena riportati.

1.2.1 Ingegnerizzazione dei requisiti

Dalla definizione di *requisito funzionale* come requisito strettamente legato a funzionalità e alla specifica dei requisiti dell'utente possiamo individuare gli stessi nell'ottica del sistema software appena formulato nella specifica. I requisiti funzionali del nostro sistema sono riassumibili con le seguenti proposizioni: Autenticazione di un generico utente (la cui utenza esiste già nel sistema), inserimento - da parte del personale autorizzato - di nuove utenze per il sistema, interazione con terapie, patologie e rilevazioni da parte di utente medico e paziente (nonchè amministratore, con uno scopo di coordinazione), verifica della correttezza dei parametri dati da assunzioni di terapie, rilevazioni (di farmaci, sintomi e livelli di glicemia) da parte del sistema, contatto del medico curante da parte dell'utente, visualizzazione dei log per medici e amministratori.

Dalla definizione di *requisito strutturale* come requisito tipicamente tecnico alla quale il sistema software si deve attenere durante il suo funzionamento possiamo ancora una volta individuare gli stessi nell'ottica del sistema software appena formulato nella specifica. I requisiti strutturali del nostro sistema sono: interfaccia funzionale ed intuitiva, con opportuna diversificazione della interfaccia per gli utenti con maggiore abilità e varietà di interazione (medico e amministratore), creazione del sistema di collegamento con servizio di mailing predefinito per il sistema al fine di predisporre il contatto tra gli utenti del sistema, opportune performance rispetto all'interazione con le pagine e con la base di dati (con le opportune interfacce di interazione con il dato), nonchè opportune performance, in un ottica generale (per l'utilizzatore) dell'intero sistema software.

1.3 Notazione UML: attori e casi d'uso

Sulla base delle specifiche di sistema e dei requisiti ad esse associati stabiliamo gli **attori** (della notazione UML associata al nostro sistema), che sono dati, per definizione [Som07], da quelle entità che necessiteranno di *utilizzare il sistema*, *avviare* lo stesso, ed *interagirvi*. Segue un elenco degli attori del sistema: **Diabetologo** (alias *medico diabetologo*): utente autenticato correttamente cui spetta il compito di stabilire operazioni per la gestione della cura dei pazienti del sistema, **Paziente**: utente autenticato a cui risulta diagnosticato il diabete di tipo 2 e per cui si prevede la gestione della patologia stessa mediante servizio di telemedicina, **Administrator** (alias responsabile del servizio): utente cui spetta la gestione del sistema di telemedicina (come stabilito nelle specifiche).

Sulla base di quanto definito nelle specifiche di progetto possiamo, inoltre, elencare di seguito i principali casi d'uso del sistema software in corso di studio. Dalla definizione di caso d'uso, come l'*"insieme delle funzioni che ci si aspetta di fornire ad un attore"*, *"insieme degli eventi esterni ed interni che producono effetti sul sistema"*, otteniamo il seguente elenco: **Autenticazione**, **Gestione terapia e patologia paziente**, **Visualizzazione e modifica dei dati paziente**, **Visualizzazione rilevazioni pazienti** (rilevazione livello glicemia, rilevazione assunzione farmaci e sintomi avvertiti), **Gestione delle rilevazioni del paziente**, **Verifica della correttezza delle assunzioni**, **Contatto del diabetologo**, **Creazione dell'account**, **Visualizzazione dei log**.

Nei successivi capitoli è proposta la disamina approfondita della struttura del sistema software fondata sulle entità appena introdotte e basata su documentazione in notazione UML.

Casi d'uso e specifica UML

In questa sezione intraprendiamo la disamina della specifica fornita dal **diagramma dei casi d'uso** del sistema di telemedicina. Essa comprenderà una prima fase di descrizione del diagramma dei casi d'uso nella sua interezza, nello specifico tratteremo nello specifico gli elementi che insieme ad attori e casi d'uso ci permettono di definire il sistema in maniera coerente con le regole sintattiche e semantiche definite dal modello UML. Gli elementi di base che analizzeremo sono: il sistema, le tipologie di associazione utilizzate per legare le entità, dipendenza tra entità, seguita da una fase di analisi della **specifica dei casi d'uso**, ovvero delle *precondizioni*: condizioni che devono essere vere prima che il caso d'uso possa eseguire, *sequenza degli eventi*: i passi che compongono il caso d'uso, *post condizioni*: condizioni che valgono alla terminazione del caso d'uso.

2.1 Diagramma dei casi d'uso

Forniamo di seguito la rappresentazione grafica del diagramma dei casi d'uso del sistema. Si notino le scelte implementative legate alla notazione UML precedentemente definite nella sezione di Notazione UML al capitolo precedente.

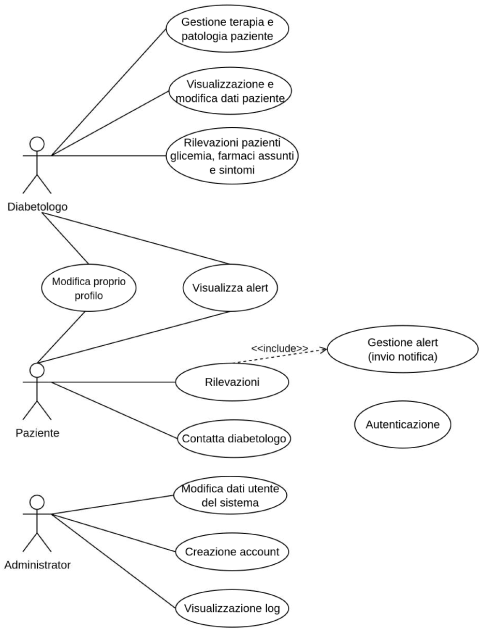

Figura 1: Diagramma dei casi d'uso del sistema software.

2.2 Specifica dei casi d'uso

A questo punto passiamo alla disamina approfondita dei casi d'uso mediante la specifica dei casi d'uso UML. Seguirà una analisi suddivisa nelle seguenti quattro sezioni: casi d'uso inerenti rispettivamente a attore diabetologo, attore paziente ed attore amministratore e di un attore generico utente del sistema, a partire da una sommaria e riassuntiva descrizione fino alla rappresentazione integrale mediante precondizioni, sequenza degli eventi (ed eventuali sequenze alternative, se presenti) e post condizioni.

2.2.1 Casi d'uso dell'attore *Diabetologo*

Seguono i casi d'uso dell'attore diabetologo, si rimanda alla sezione del primo capitolo: *Notazione UML: attori e casi d'uso*, per una più approfondita descrizione.

2.2.1.1 UC1: Gestione terapia e patologia paziente

Il diabetologo, una volta autenticato, dovrà poter specificare e gestire le terapie previste per i pazienti da lui presi in cura, dove per gestione si intende la supervisione della corretta assunzione di farmaci e di eventuali sintomi avvertiti dal paziente durante un generico istante temporale.

Caso d'uso: Gestione terapia e patologia paziente	
Id	UC1
Attori	Diabetologo
Precondizioni	Il medico è stato assegnato al paziente. Il medico si è autenticato.
Sequenza degli eventi	1. Il medico entra nella pagina con l'elenco delle terapie per i suoi pazienti. 2. Se un paziente non ha già una terapia assegnata per una determinata patologia: (a) il medico stila una terapia atta a risolvere la patologia diagnosticata al paziente. 3. Il medico visualizza e/o modifica la terapia correlata al paziente.
Post condizioni	Il medico ha visualizzato e/o creato/modificato correttamente la terapia assegnata al paziente, che può iniziare o cambiare la cura.

Tabella 1: Specifica del caso d'uso *Gestione terapia e patologia paziente*.

2.2.1.2 UC2: Visualizzazione e modifica dati paziente

Ogni medico può analizzare ed aggiornare in qualsiasi momento i dati di ogni paziente autenticato nel sistema (tali dati includono la anagrafica del paziente stesso ed inoltre una breve anamnesi dello stesso.

Caso d'uso: Visualizzazione e modifica dati paziente	
Id	UC2
Attori	Diabetologo
Precondizioni	Il medico è stato assegnato al paziente. Il medico si è autenticato.
Sequenza degli eventi	1. Il medico è in grado di visualizzare i dati anagrafici del paziente. 2. Ha la possibilità di modificare questi dati in caso di necessità.
Post condizioni	I dati sono stati visualizzati e/o aggiornati.

Tabella 2: Specifica del caso d'uso *Visualizzazione e modifica dati paziente.*

2.2.1.3 UC3: Rilevazioni pazienti glicemia, farmaci assunti e sintomi

Il paziente autenticato può quindi inserire la rilevazione del livello di glicemia ottenuto ed il medico potrà successivamente visualizzarle.

Caso d'uso: Rilevazioni pazienti glicemia, farmaci assunti e sintomi	
Id	UC3
Attori	Diabetologo
Precondizioni	Il medico si è autenticato.
Sequenza degli eventi	1. Il paziente utilizza correttamente e con frequenza il sistema, di conseguenza è monitorato rispetto alla sua condizione clinica. (a) Se qualche parametro - in riferimento a rilevazione della glicemia - non risulta compreso negli intervalli di sicurezza, allora il sistema invia una notifica al medico. (b) Se qualche parametro - in riferimento ai farmaci assunti - non risulta compreso nelle soglie di sicurezza, allora il sistema invia una notifica al medico stesso, nello specifico: se un paziente non assume la terapia prescritta per 3 o più giorni viene notificato ciò al medico curante. 2. Il medico è in grado di prendere contromisure rispetto alla terapia prescritta al paziente in cura.
Post condizioni	Lo stato clinico del paziente viene correttamente monitorato dal medico e dal paziente stesso.

Tabella 3: Specifica del caso d'uso *Rilevazioni pazienti glicemia, farmaci assunti e sintomi.*

2.2.2 Casi d'uso dell'attore *Paziente*

Seguono i casi d'uso dell'attore paziente, si rimanda alla sezione del primo capitolo: *Notazione UML: attori e casi d'uso*, per una più approfondita descrizione.

2.2.2.1 UC4: Rilevazioni

Il paziente autenticato può quindi inserire la rilevazione del livello di glicemia ottenuto.

Caso d'uso: Rilevazioni	
Id	UC4
Attori	Paziente
Precondizioni	Il paziente necessita di inserire le sue rilevazioni nel sistema. Il paziente si è autenticato.
Sequenza degli eventi	1. Il paziente può inserire a seconda delle necessità i dati relativi a: (a) Livelli di glicemia prima e dopo i pasti principali; (b) Assunzione di farmaci prescritti e non; (c) Eventuali sintomi correlati o meno al diabete;
Post condizioni	Il paziente ha interagito con il sistema notificando il suo stato clinico attuale.

Tabella 4: Specifica del caso d'uso *Rilevazioni*.

2.2.2.2 UC5: Verifica Correttezza assunzioni (invio notifica)

Il sistema dovrà effettuare la verifica della sicurezza e correttezza dei livelli di glicemia e del rispetto delle terapie interne ad esso assegnate da un medico interno (dettaglio implementativo: si assumono come terapie interne anche farmaci non strettamente legati alla gestione del diabete di tipo 2 ma anche patologie in comorbidità dello stesso, ciò implica quindi la gestione della coerenza della terapia prescritta rispetto alla patologia diagnosticata) e talvolta inviare una notifica nel caso in cui vi siano situazioni non in linea con quelle corrette.

Caso d'uso: Verifica Correttezza assunzioni (invio notifica)	
Id	UC5
Attori	Paziente, Diabetologo.
Precondizioni	Il paziente interagisce memorizzando rilevazioni rispetto ai valori di glicemia nel sangue e terapie assunte.
Sequenza degli eventi	1. Il paziente effettua ed inserisce una nuova rilevazione o assunzione di farmaci (derivanti da terapie prescritte). (a) Se qualche parametro - in riferimento a rilevazione della glicemia - non risulta compreso negli intervalli di sicurezza, allora il sistema invia una notifica al paziente. (b) Se qualche parametro - in riferimento ai farmaci assunti o sintomi avvertiti - non risulta compreso nelle soglie di sicurezza, allora il sistema invia una notifica al medico stesso, nello specifico: se un paziente non assume la terapia prescritta per 3 o più giorni viene notificato ciò al paziente.
Post condizioni	Vengono inviati gli alert al paziente per segnalare eventuali anomalie sull'assunzione di farmaci o sui livelli di glicemia.

Tabella 5: Specifica del caso d'uso *Verifica Correttezza assunzioni (invio notifica)*.

2.2.2.3 UC6: Contatta diabetologo

Per ogni paziente è specificato un medico di riferimento, al quale il paziente può inviare email per richieste e domande varie.

Caso d'uso: Contatta diabetologo	
Id	UC6
Attori	Paziente
Precondizioni	Il paziente necessita di conoscere informazioni riguardanti il medico curante e/o verificare informazioni riguardanti il suo stato clinico. Il paziente si è autenticato.
Sequenza degli eventi	1. Il paziente effettua una richiesta al sistema riguardante le informazioni di contatto del medico curante. 2. Il sistema fornisce al paziente le informazioni richieste. 3. Il paziente comunica autonomamente con il diabetologo mediante sistema di comunicazione esterno.
Post condizioni	Il paziente ha interagito con il medico mediante sistema di comunicazione esterno.

Tabella 6: Specifica del caso d'uso *Contatta diabetologo*.

2.2.3 Casi d'uso dell'attore *Amministratore*

Seguono i casi d'uso dell'attore amministratore, si rimanda alla sezione del primo capitolo: *Notazione UML: attori e casi d'uso*, per una più approfondita descrizione.

2.2.3.1 UC7: Creazione account

Il sistema è coordinato da utenti individuati come responsabili del servizio di telemedicina, essi inseriscono e gestiscono successivamente i dati iniziali di pazienti e medici necessari per l'autenticazione.

Caso d'uso: Creazione account	
Id	UC7
Attori	Amministratore
Precondizioni	L'amministratore si è autenticato.
Sequenza degli eventi	1. I nuovi diabetologi o i nuovi pazienti richiedono la creazione di un account ad un amministratore di sistema. 2. Questo crea la nuova utenza fornendo le credenziali al diretto interessato.
Post condizioni	Gli utenti del sistema sono in grado di autenticarsi e di poter accedere all'applicativo.

Tabella 7: Specifica del caso d'uso *Creazione account*.

2.2.3.2 UC8: Visualizzazione log

Ogni medico può analizzare ed aggiornare in qualsiasi momento i dati di ogni paziente autenticato nel sistema (tali dati includono la anagrafica del paziente stesso ed inoltre una breve anamnesi dello stesso, inoltre tali modifiche sono analizzabili e gestite dal responsabile del servizio (admin).

Caso d'uso: Visualizzazione log	
Id	UC8
Attori	Amministratore
Precondizioni	L'amministratore è correttamente autenticato.
Sequenza degli eventi	1. Gli amministratori accedendo alla propria area riservata sono in grado di visualizzare la sequenza delle operazioni eseguite dai vari medici sui pazienti.
Post condizioni	Si è in grado di comprendere quale medico ha eseguito determinate operazioni.

Tabella 8: Specifica del caso d'uso *Visualizzazione log.*

2.2.3.3 UC9: Modifica dati utente del sistema

L'amministratore può gestire le utenze della piattaforma, mediante una opportuna interfaccia, all'evenienza deve poter effettuare la modifica di informazioni sull'utenza, e talvolta effettuare la cancellazione della stessa.

Caso d'uso: Modifica dati utente del sistema	
Id	UC9
Attori	Amministratore
Precondizioni	L'amministratore è correttamente autenticato.
Sequenza degli eventi	1. L'amministratore seleziona l'utente da modificare. 2. L'amministratore più modificare le informazioni dell'utenza oppure cancellare definitivamente l'utenza stessa.
Post condizioni	L'utente è stato correttamente modificato/eliminato.

Tabella 9: Specifica del caso d'uso *Modifica dati utente del sistema.*

2.2.4 Casi d'uso dell'attore *Utente generico*

Sia pazienti, diabetologi che amministratori di sistema devono poter autenticarsi nel sistema, dove la creazione dell'utenza (come detto precedentemente) spetta direttamente agli amministratori.

2.2.4.1 UC10: Autenticazione

Qualsiasi use case precedentemente citato prevede una autenticazione avvenuta con successo, tuttavia l'utente non deve autenticarsi ad ogni operazione che esegue, ma il sistema memorizza le sue credenziali.

Caso d'uso: Autenticazione	
Id	UC10
Attori	Diabetologo, Amministratore, Paziente
Precondizioni	L'utenza deve essere stata creata da un amministratore di rete.
Sequenza degli eventi	1. L'utente inserisce username. 2. L'utente inserisce la password. 3. In caso di login eseguito con successo accede alla propria area riservata con la possibilità di eseguire tutte le operazioni a lui consentite.
Post condizioni	L'utente può utilizzare il sistema.

Tabella 10: Specifica del caso d'uso *Autenticazione*.

2.2.4.2 UC11: Modifica proprio profilo

Caso d'uso: Modifica proprio profilo	
Id	UC11
Attori	Paziente, Diabetologo
Precondizioni	L'utente è correttamente autenticato.
Sequenza degli eventi	1. L'utente seleziona se modificare password o immagine profilo.
Post condizioni	L'utente è stato correttamente modificato.

Tabella 11: Specifica del caso d'uso *Modifica proprio profilo*.

2.2.4.3 UC12: Visualizza alert

Gli utenti del sistema devono poter ricevere opportuni alert da parte del sistema, nell'evenienza che si presenti una delle situazioni definite nella specifica. Se l'utente autenticato è un medico esso potrà visualizzare gli alert relativi a tutti i pazienti (nella visualizzazione standard) oppure degli alert specifici sulla base di alcuni filtri. Se, invece, l'utente autenticato è un paziente allora esso potrà visualizzare solamente gli alert che lo riguardano, inoltre, potrà eventualmente assumere in questa sede i farmaci mancanti per la giornata (sulla base delle sue terapie).

Caso d'uso: Visualizza alert	
Id	UC12
Attori	Paziente, Diabetologo
Precondizioni	L'utente è correttamente autenticato.
Sequenza degli eventi	1. L'utente visualizza i suoi alert non letti. (a) Se l'utente loggato è un paziente allora visualizza i farmaci da assumere nel giorno. (b) Se l'utente loggato è un diabetologo allora visualizza gli alert per i valori di glicemia e farmaci non assunti (da almeno 3 giorni), con la possibilità di filtrare opportunamente le informazioni.
Post condizioni	Gli alert vengono visualizzati correttamente.

Tabella 12: Specifica del caso d'uso *Visualizza alert*.

Diagrammi di sequenza ed interazione

In questa sezione intraprendiamo la disamina della formalizzazione UML delle relazioni vigenti tra le entità del sistema (basate sui casi d'uso ampiamente descritti nei capitolo precedenti), mediante i **diagrammi di interazione** e **diagrammi di sequenza** associati al sistema. Prima di incominciare con la descrizione delle relazioni esistenti tra le entità del sistema di telemedicina in corso di studio, anche in legame ai casi d'uso precedentemente definiti, è necessario definire la natura degli elementi di cui si studiano le interazioni.

Di fatto, nella notazione UML, un diagramma di interazione è un concetto che astrae qualsiasi rappresentazione diagrammatica che descriva un'interazione tra elementi (oggetti, componenti, sistemi) interni al sistema in corso di progettazione. Questo concetto include vari tipi di diagrammi, tra cui: diagrammi di sequenza, diagrammi di comunicazione (detti anche "collaboration diagram"), diagrammi di temporizzazione, diagrammi di overview dell'interazione. Il diagramma di sequenza è quindi un tipo specifico di diagramma di interazione che si concentra sull'ordine temporale dei messaggi tra gli oggetti. Mostra come gli oggetti o componenti comunicano tra loro nel tempo, evidenziando quando i messaggi vengono inviati e ricevuti. Di fatto utilizzeremo indistintamente questi strumenti - fornito dal formalismo UML - e i principi che li riguardano nel seguito, ma in questa fase introduttiva si preferisce porre un attenzione particolare al fatto per cui i diagrammi di sequenza verranno utilizzati per formalizzare concretamente le relazioni vigenti nel sistema di telemedicina tra uno o più attori del sistema e il sistema stesso mediate attività che compongono, e sono quindi direttamente associabili, con i casi d'uso del sistema stesso, mentre utilizzeremo il più astratto e generico strumento UML del diagramma di interazione per sottolineare dei dettagli - implementativi - non strettamente o direttamente deducibili dalla sola documentazione già fornita (risulteranno quindi essere uno strumento di supporto alla integralità della documentazione delle relazioni vigenti tra le entità del sistema). Di seguito forniamo nell'ordine: diagrammi si sequenza e successivamente diagrammi si interazione integrativi.

3.1 Diagrammi di sequenza

Seguono i diagrammi di sequenza derivati dai casi d'uso definiti al capitolo precedente, e gli attori definiti nel primo capitolo, si rimanda ad essi per una più approfondita descrizione.

3.1.1 Caso d'uso: Autenticazione

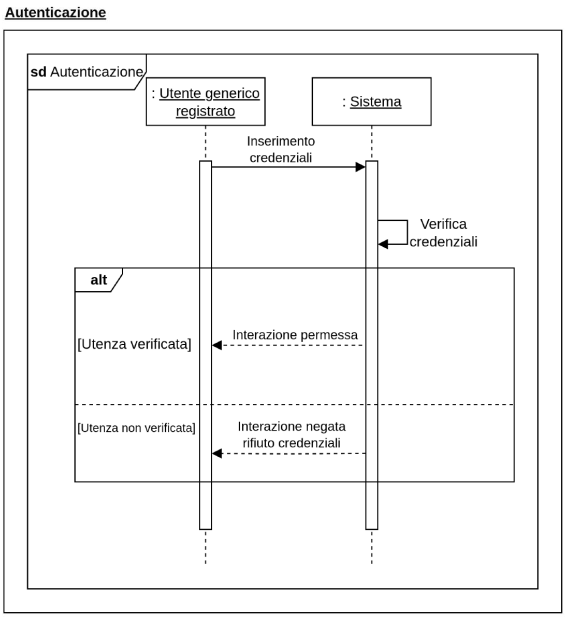

Figura 2: Diagramma di sequenza del caso d'uso *Autenticazione*.

3.1.2 Caso d'uso: Visualizzazione e modifica dati paziente

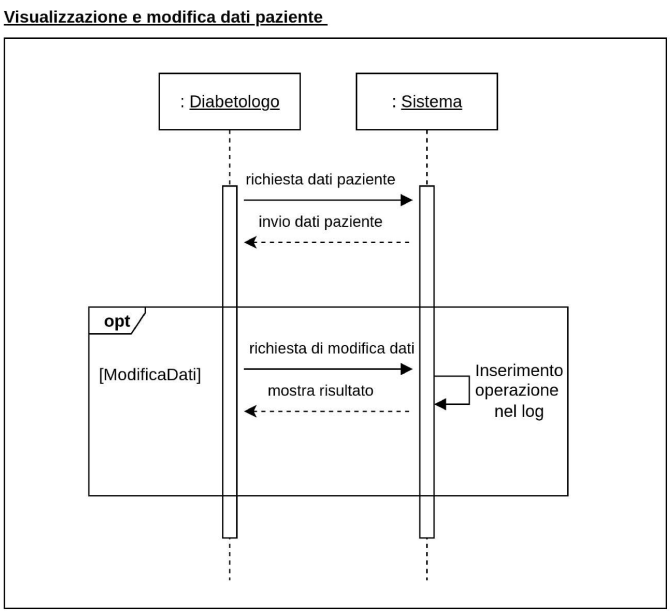

Figura 3: Diagramma di sequenza del caso d'uso *Visualizzazione e modifica dati paziente*.

3.1.3 Caso d'uso: Gestione terapie e patologie paziente

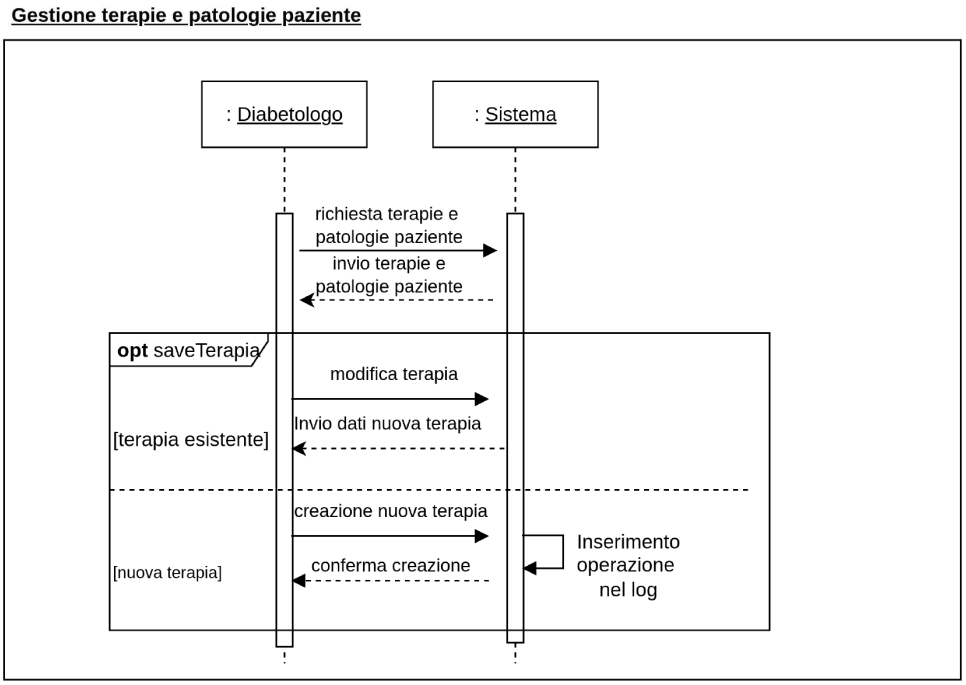

Figura 4: Diagramma di sequenza del caso d'uso *Gestione terapie e patologie paziente*.

3.1.4 Caso d'uso: Rilevazioni pazienti glicemia, farmaci assunti e sintomi

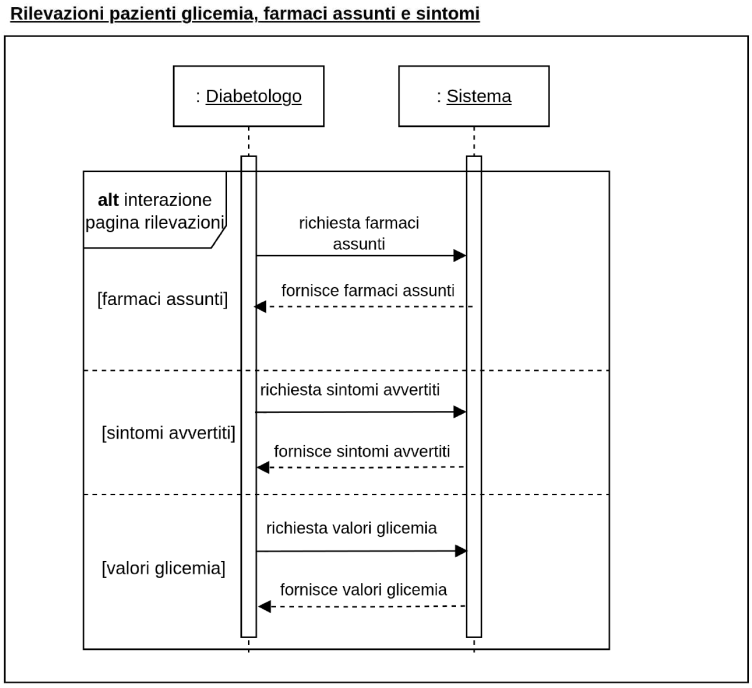

Figura 5: Diagramma di sequenza del caso d'uso *Rilevazioni pazienti glicemia, farmaci assunti e sintomi*.

3.1.5 Caso d'uso: Rilevazione

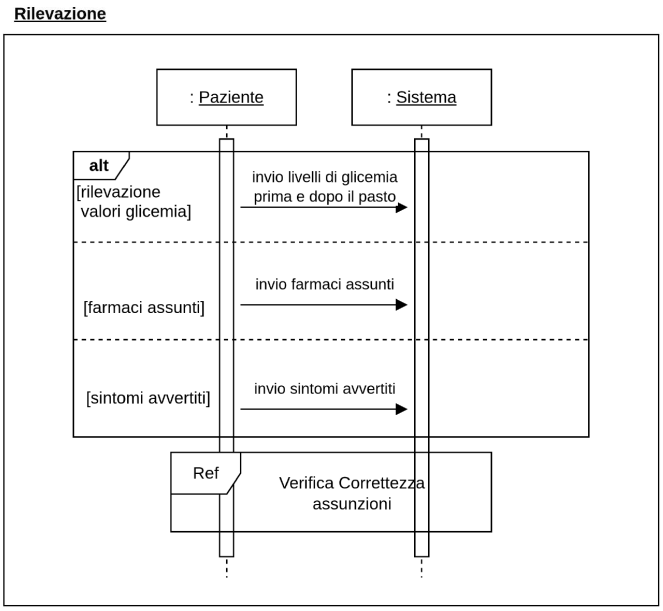

Figura 6: Diagramma di sequenza del caso d'uso *Rilevazione*.

3.1.6 Caso d'uso: Contatta medico

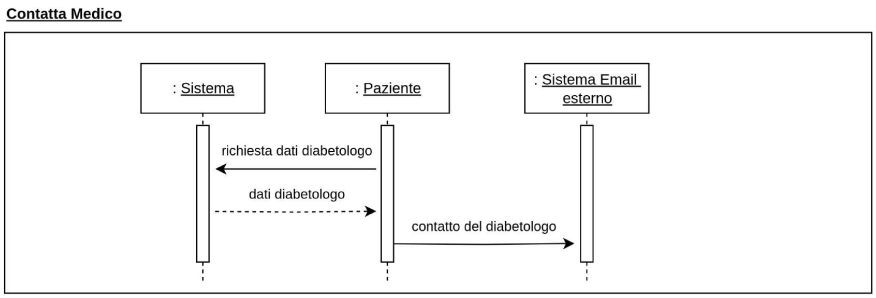

Figura 7: Diagramma di sequenza del caso d'uso *Contatta medico*.

3.1.7 Caso d'uso: Gestione alert (invio notifica)

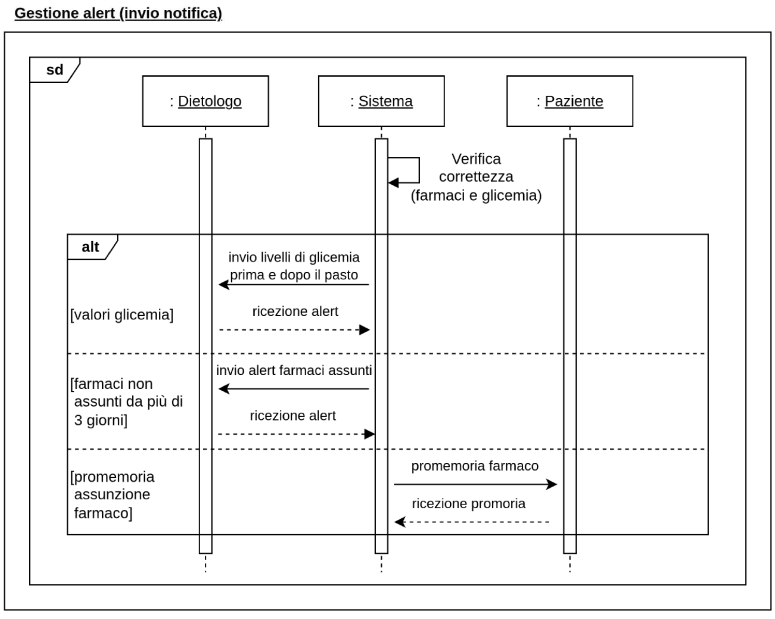

Figura 8: Diagramma di sequenza del caso d'uso *Verifica correttezza assunzioni (invio notifica)*.

3.2 Diagrammi di interazione

3.2.1 Caso d'uso: Creazione account

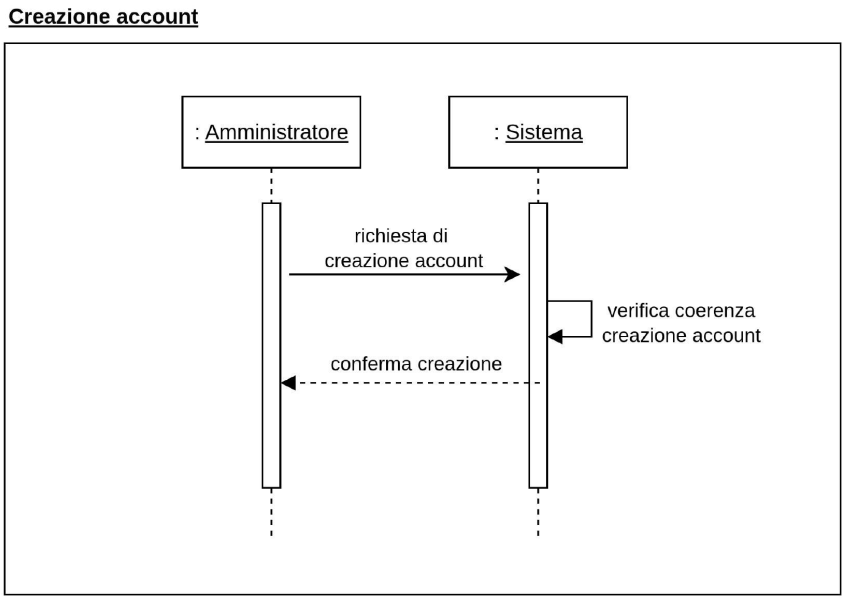

Figura 9: Diagramma di interazione integrativo del caso d'uso *Creazione account*.

3.2.2 Caso d'uso: Visualizzazione log

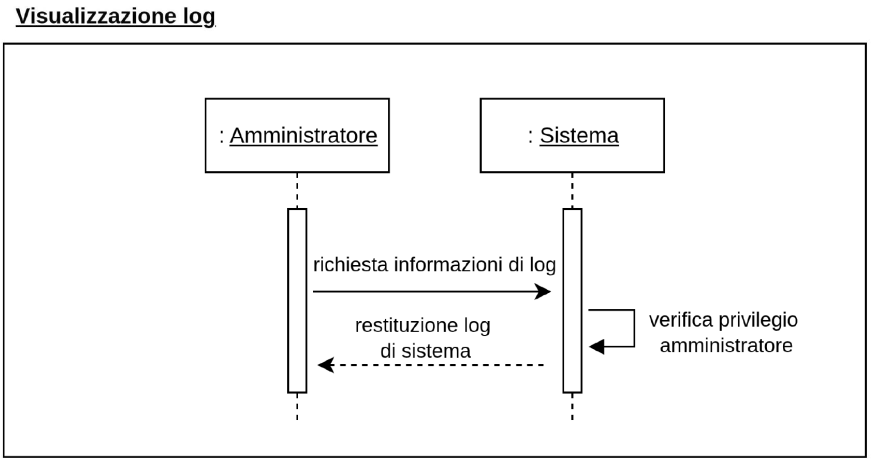

Figura 10: Diagramma di interazione integrativo del caso d'uso *Visualizzazione log*.

Diagrammi di attività

In questa sezione forniamo l'analisi del diagramma delle attività e dei dettagli implementativi - in legame ai fondamenti teorici della notazione UML - che hanno portato alle scelte che riflettono il funzionamento del sistema nell'evoluzione temporale dei servizi forniti. I principi generali del diagramma UML delle attività definiscono un diagramma i cui scopi sono quelli di modellare il flusso di un caso d'uso (analisi), modellare il funzionamento di un'operazione di classe (progettazione), modellare un algoritmo (progettazione).

Lo scopo ultimo dei diagrammi di attività forniti di seguito sarà quindi quello di descrivere un flusso di azioni che realizzano un certo comportamento specifico, dove l'enfasi non è sullo scambio di messaggi (altri diagrammi UML si occupano direttamente di ciò) ma sui blocchi di comportamento.

4.1 Diagramma di attività relativo all'attore utente generico: Autenticazione

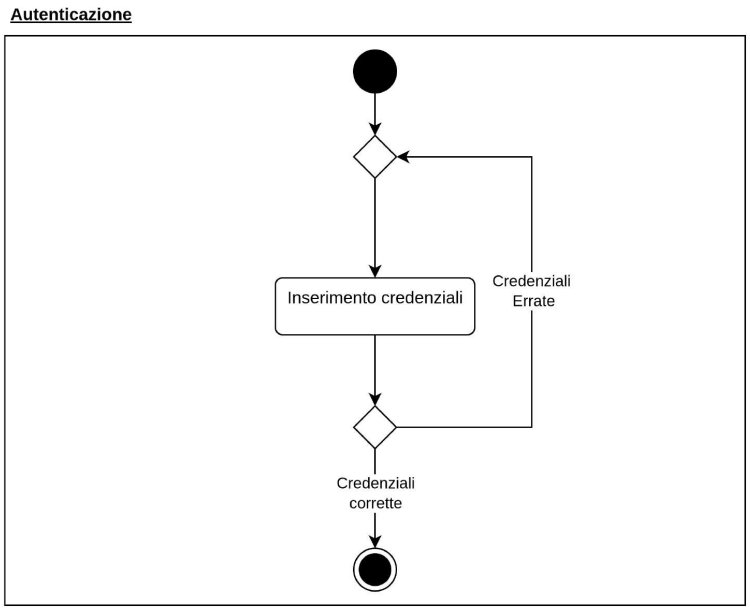

Figura 11: Diagramma di attività relativo all'attore utente generico: *Autenticazione*.

4.2 Diagramma di attività relativo all'attore medico

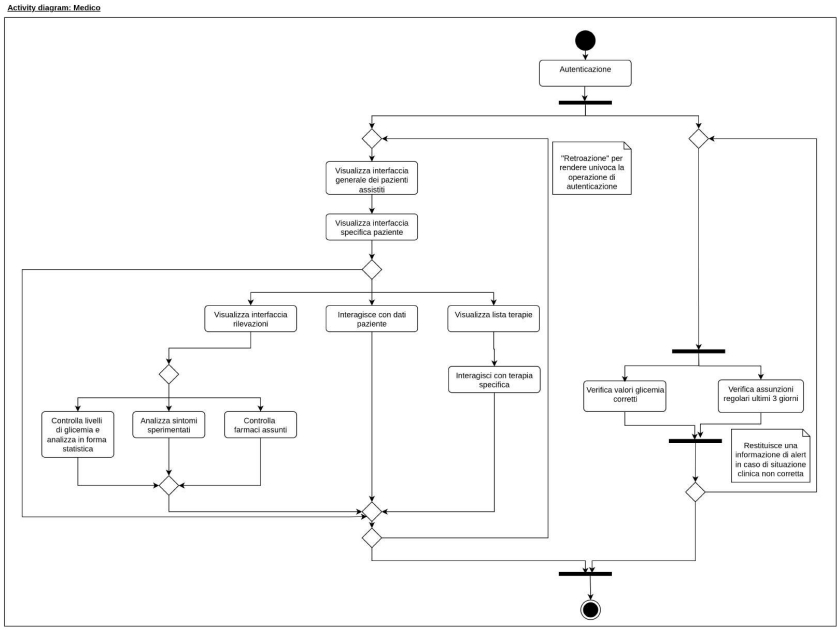

Figura 12: Diagramma di attività relativo all'attore *medico*.

4.3 Diagramma di attività relativo all'attore paziente

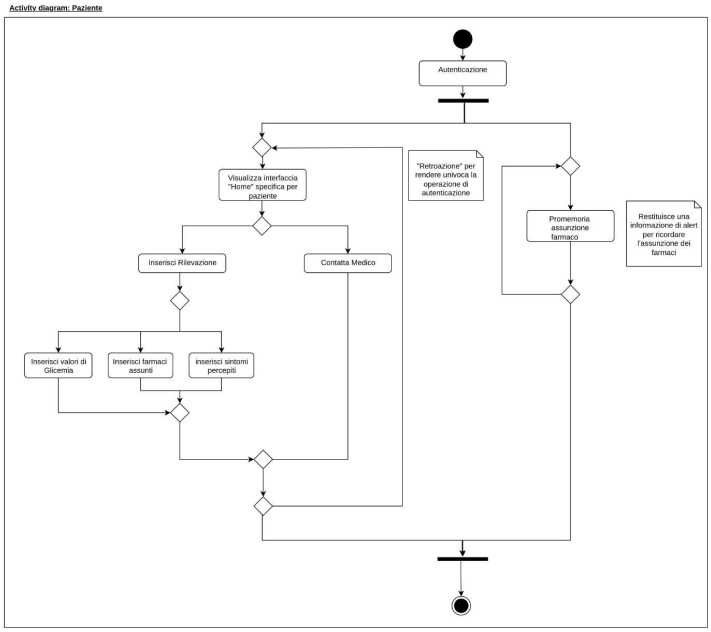

Figura 13: Diagramma di attività relativo all'attore *paziente*.

Diagramma delle classi

A questo punto, mediante *linguaggio di progettazione* UML, siamo in grado di adattare il modello di analisi (definito dalla documentazione precedente) affinché esso diventi implementabile, grazie al **diagramma delle classi**. Una volta definiti i diagrammi di analisi (i diagrammi di analisi sono più astratti di quelli di progettazione) siamo in grado di estrarre un insieme di classi di analisi dalla specifica del problema. Saremo, inoltre, in grado di definire quali attributi e quali operazioni dovremo fornire mediante il nostro sistema orientato agli oggetti.

Ogni classe definita in fase di progettazione verrà riprodotta nel diagramma mediante una rappresentazione che permetterà di dedurre il rispettivo *nome e l'eventuale stereotipo*, gli *attributi* e le *operazioni*. Saranno presenti inoltre gli ornamenti adeguati, come le tipologie di *visibilità*. Inoltre verranno definite le *relazioni* (*generalizzazione*, *realizzazione*, *associazione*, *dipendenza*, *aggregazione*, *composizione*) presenti tra le classi del sistema, dove necessario.

Verranno forniti i diagrammi nel seguente ordine: diagramma delle classi delle entità definite dalla specifica e delle classi strettamente correlata ad esse (o al loro comportamento), successivamente verrà esposto un diagramma delle classi ausiliario che illustra le classi che implementano dei concetti o delle entità della specifica non strettamente tramutate in entità UML, infine forniamo un diagramma delle classi funzionale agli oggetti definiti nella modellazione stratificata svolta mediante *pattern MVC*, nello specifico si mira ad esporre le dinamiche che definiscono le relazioni interne tra gli strumenti software (classi java e file fxml) che compongono *controller* e *view*.

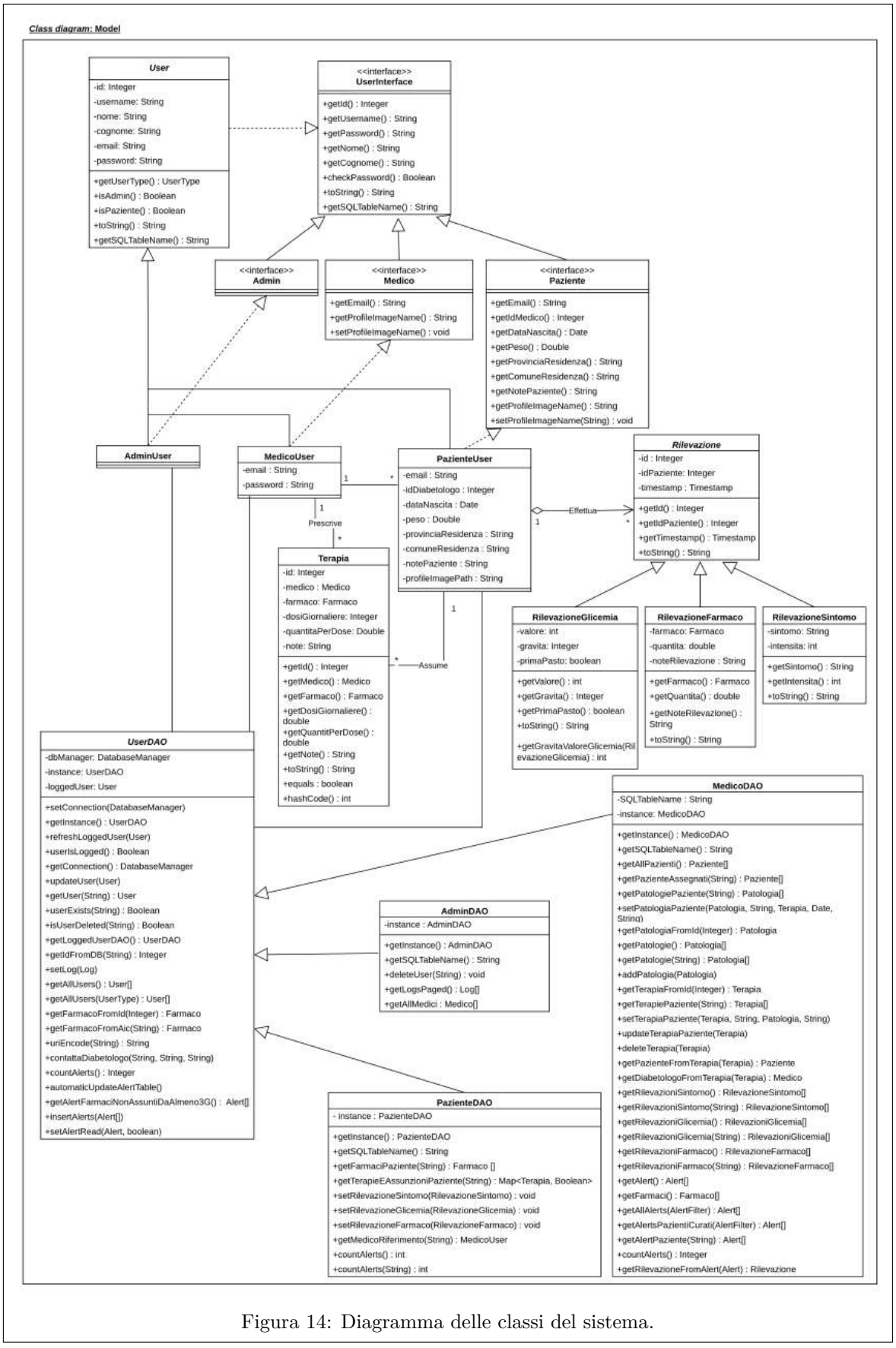

Figura 14: Diagramma delle classi del sistema.

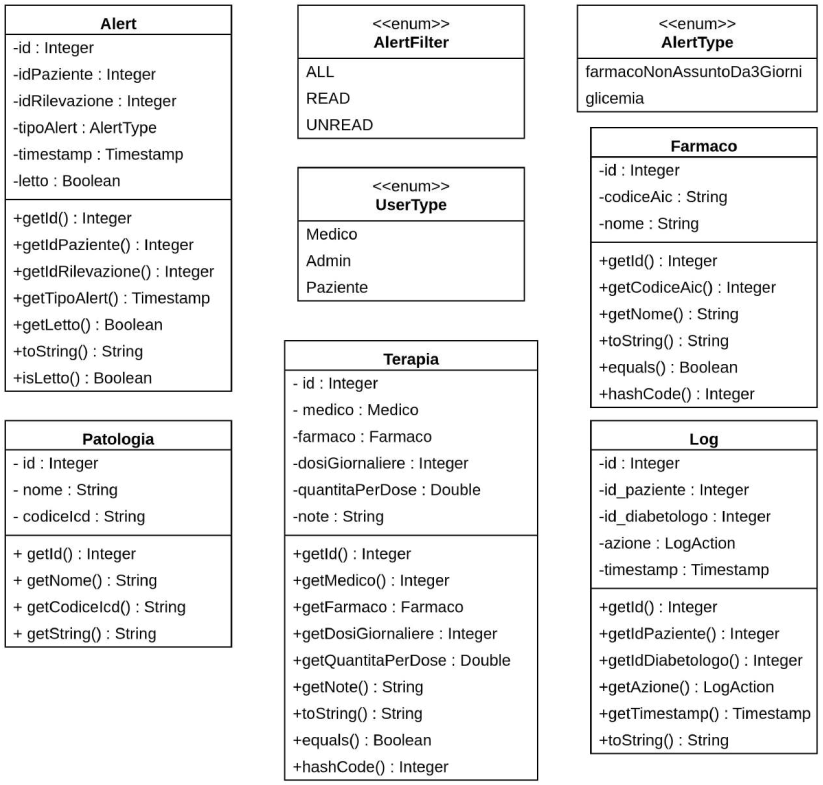

Figura 15: Diagramma delle classi del sistema, contenute nel package *Utils*.

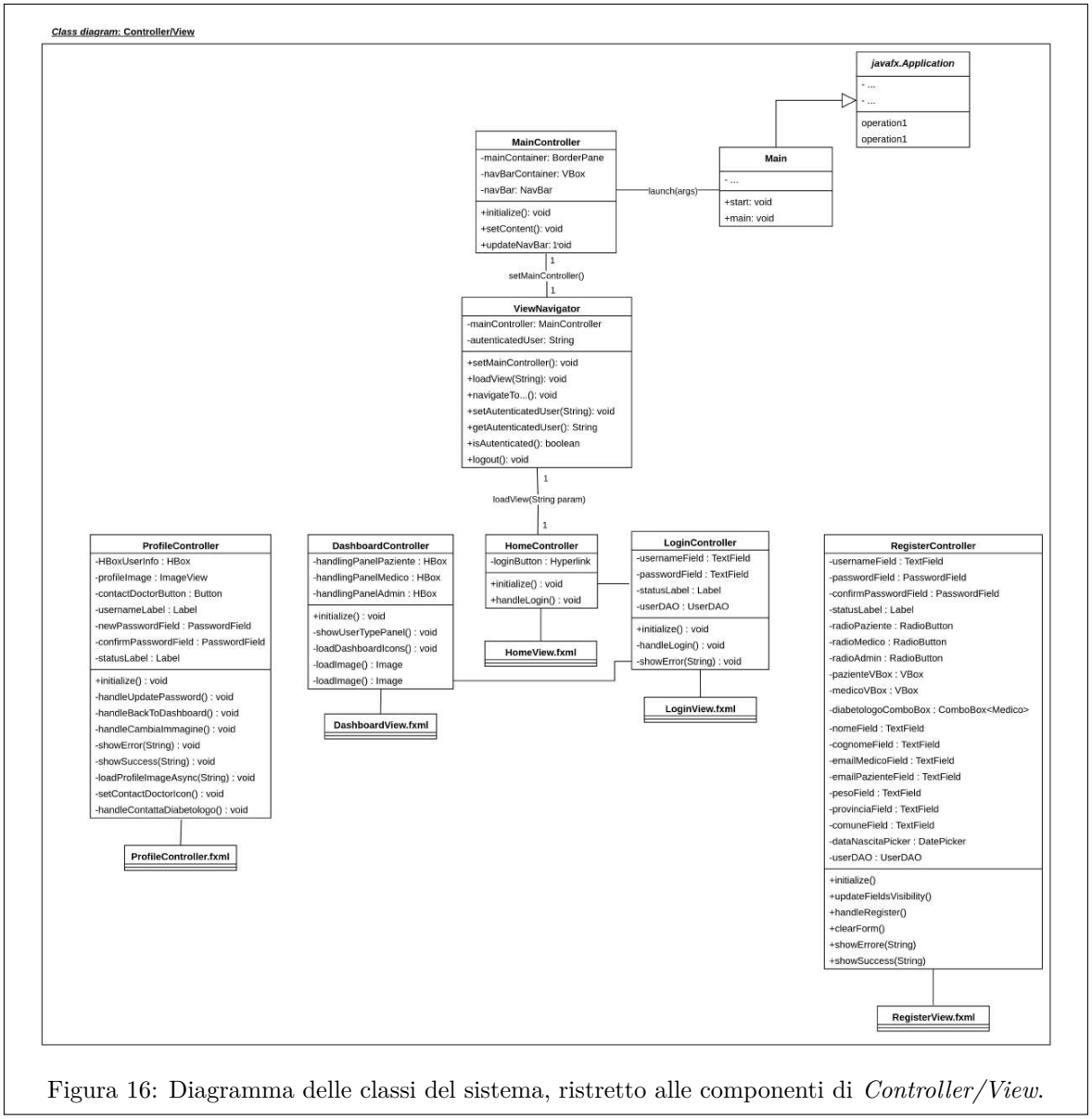

Figura 16: Diagramma delle classi del sistema, ristretto alle componenti di *Controller/View*.

Processo di progettazione e sviluppo del sistema software

In questa sezione forniamo l'integrazione data dalla sintesi delle attività svolte durante la fase di progettazione e sviluppo del progetto. Descriveremo scelte progettuali giustificando le stesse mediante i principi teorici alla base dell'ingegneria del software e dei **pattern architetturali** e di **design** dei sistemi software da noi in corso di studio.

Verranno presentate, inoltre, delle sezioni di descrizione del **processo di sviluppo software** adottato. Quanto appena descritto vedrà la sua attuazione più integrale con l'activity diagram di fine capitolo.

6.1 Descrizione del prodotto software

Data la classificazione delle tipologie di prodotto software fornita dall'ingegneria del software possiamo affermare che il sistema in corso di studio sia un sistema ***stand-alone*** *specifico* (quindi non generalizzato) con lo scopo di predisporre le *transazioni* con un sistema di base di dati tra tre tipologie di utenti differenti, il cui utilizzo e interfacciamento al sistema è differente. L'architettura del sistema software, perciò, contiene componenti di più modelli di architettura differenti, elenchiamo di seguito ognuno di essi, specificando le caratteristiche che hanno portato alla loro attuazione.

6.1.1 Architettura di applicazione

Sistemi di elaborazione delle transazioni: si tratta di sistemi il cui scopo è elaborare le richieste di informazioni degli utenti da un database o le richieste di aggiornamento di dati contenuti in una base di dati. Possiamo riscontrare le caratteristiche appena indicate, in legame al nostro sistema, dal momento che sia utenti medici che utenti pazienti e amministratori necessitano di interagire con dati non volatili e da conservare al termine della sessione di interazione con il sistema. Il dettaglio implementativo dato dall'implementazione di una base di dati relazionale SQL si è reso necessario per la quantità di informazioni per cui è richiesta la memorizzazione, ed inoltre per la omogeneità delle stesse che favorisce una organizzazione data da una base di dati relazionale. Le componenti che, per definizione data della notazione UML, costituiscono un sistema di elaborazione delle transizioni sono riscontrabili nella rappresentazione grafica di seguito.

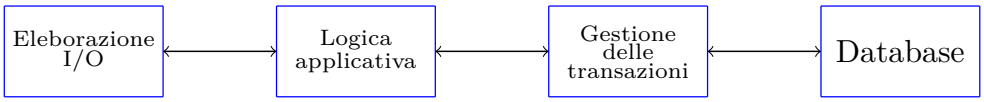

Figura 17: Rappresentazione grafica delle componenti del sistema date dalla definizione di "*sistemi di elaborazione delle transazioni*".

Sistema stratificato: si tratta di un sistema la cui struttura delle funzionalità può essere organizzata a strati. Si tratta comunque di un sottoinsieme dei sistemi basati su transazioni, poiché l'interazione con questi sistemi comporta generalmente transazioni di database. Di fatto, l'implementazione di una tale architettura implica la definizione di almeno i seguenti quattro componenti: *interfaccia utente*, *comunicazioni con gli utenti*, *reperimento delle informazioni*, *database di sistema*. La implementazione di questo modello di architettura si rende necessaria dal momento che tutti gli utenti dispongono di una interazione con il database ed inoltre essi necessitano di una

interfaccia adeguata alla tipologia di utente e di interazione con il dato. L'organizzazione astratta degli strati, appena formulata, per l'architettura, può essere riscontrata nella implementazione del pattern DAO (*Data Access Object*), la cui trattazione sarà approfondita di seguito. Forniamo di seguito la rappresentazione grafica dei livelli dell'architettura e dei relativi componenti.

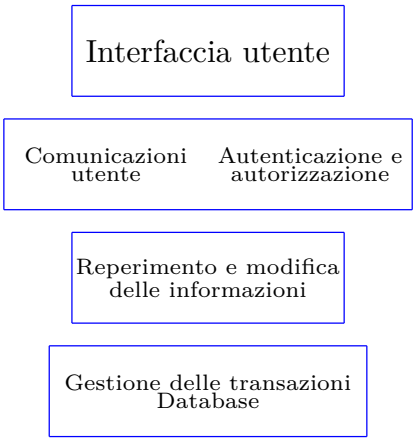

Figura 18: Rappresentazione grafica componenti date dalla definizione di "*sistema stratificato*".

6.2 Progettazione architetturale

Quanto descritto nelle sottosezione (ovvero i modelli di architettura utilizzati) ci permette ora di analizzare la architettura vera e propria del sistema e come si è giunti ad essa mediante la fase di progettazione. Da quanto definito è emersa la necessità di definire (in legame, in primis, al modello di architettura dei *sistemi di elaborazione delle transizioni* e successivamente quello del *sistema stratificato*), un opportuno sistema di ***base di dati***, un insieme di componenti di ***Gestione delle transazioni*** oltre alla *logica applicativa* e *elaborazione I/O*, ***interfaccia utente***, ***Comunicazioni utente***, ***Autenticazione e autorizzazione***, ***reperimento e modifica delle informazioni***, ***Gestione delle transazioni***. Analizziamo come le componenti sono state inserite nel sistema software nella fase di progettazione e di sviluppo.

6.2.1 Base di dati

Il sistema di base di dati rappresenta il livello fondante sia del modello di architettura di sistemi di elaborazione delle transazioni che di quello dei sistemi stratificati, perciò verrà trattato per primo, dal momento che su di esso si fondano gli altri livelli.

Dal momento che il sistema di base di dati che si intende inserire nel sistema software deve rispettare tutti i requisiti strutturali definiti, al fine di ottenere un sistema software che integralmente rispetti gli stessi, una volta analizzate le possibili implementazioni a nostra disposizione si è optato (in fase di progettazione e di sviluppo) per l'attuazione di una base di dati di tipo relazionale (Structured Query Language), mediante l'utilizzo di un DBMS **SQLite**[1]. SQLite, [SQL], è una libreria in linguaggio C che implementa un database engine SQL piccolo, veloce, autonomo, affidabile. La base di dati è stata creata mediante la suite software fornita dall'organizzazione di SQLite stessa. Essa permette la creazione di un file di formato *.DB* interrogabile mediante opportune librerie java **JDBC sqlite**, correttamente implementate proprio in questa prima fase di progettazione e sviluppo. Di fatto, trattandosi di un database relazionale è risultato necessario, nella fase di progettazione, attenersi ai principi fondanti della branca Relazionale delle basi di dati. A tale fine si è proceduto con un attenta *analisi dei requisiti* (che nella pratica si è tramutata in un

[1]SQLite uno tra i motori di database più utilizzato al mondo. SQLite è integrato in tutti i telefoni cellulari e nella maggior parte dei computer e viene fornito in un'infinità di altre applicazioni utilizzate quotidianamente.

adattamento dei requisiti già specificati per l'intero sistema software), *analisi del mondo chiuso* di riferimento, *definizione del modello E/R* associato (Entity-Relationship Model). Forniamo di seguito il diagramma del modello E/R associato al database del nostro sistema software.

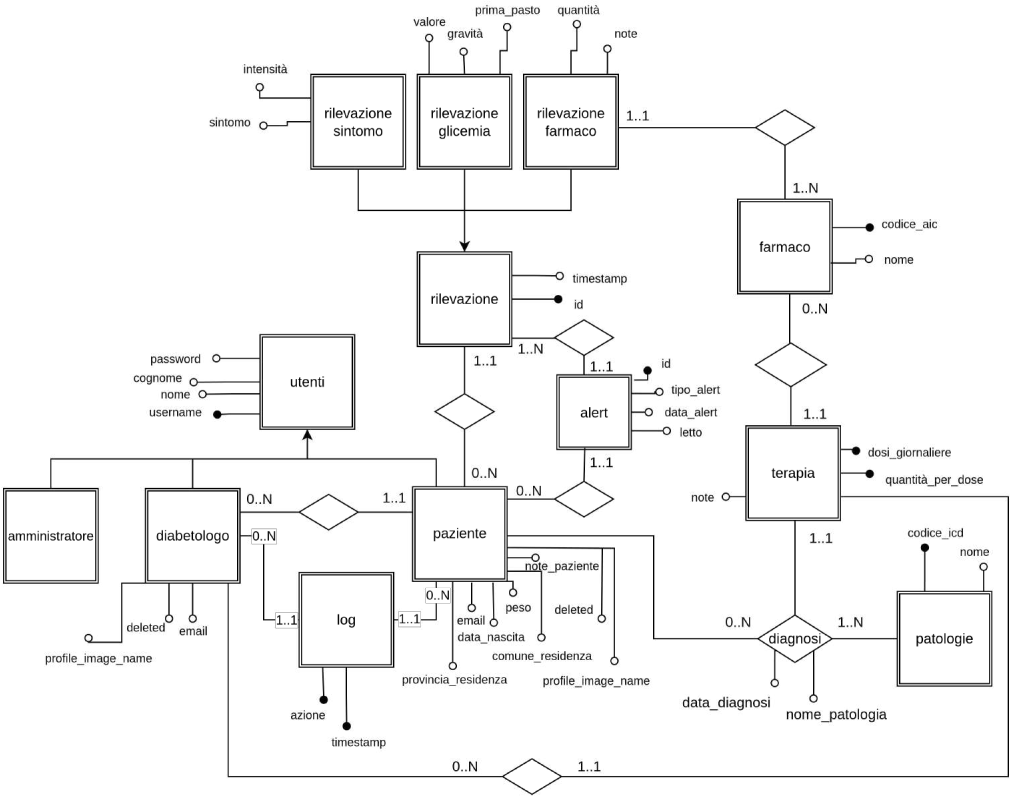

Figura 19: Diagramma E/R relativo alla base di dati del sistema.

Una volta definito il diagramma del modello E/R del database del nostro sistema si è proceduto alla ristrutturazione del database in formato di tabella di database relazionale, che hanno culminato con la definizione delle **tabelle** del database stesso. Forniamo di seguito la rappresentazione grafica del risultato ottenuto dalla ristrutturazione del modello E/R.

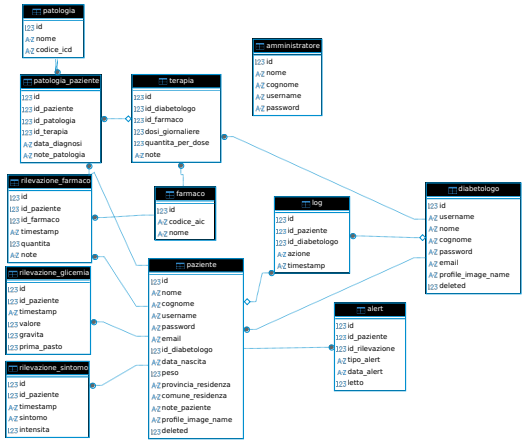

Figura 20: Rappresentazione grafica del risultato ottenuto dalla ristrutturazione del modello E/R.

La fase di progettazione della base di dati del sistema si è conclusa con la popolazione della stessa e la creazione delle query di principale rilevanza per il futuro utilizzo in fase di sviluppo. Con le azioni appena descritte si è giunti quindi alla realizzazione delle componenti di gestione delle transizioni del database, reperimento e modifica delle informazioni e gestione delle transizioni. La fase successiva è stata quella di progettazione della logica applicativa che rendesse fruibile la architettura appena progettata, segue una descrizione di questa fase. Suddividiamo la trattazione dei concetti in essa considerati in una prima sezione descrivente, mediante un più alto livello di astrazione dell'architettura del sistema, l'organizzazione generale di un sistema software, mediante i **pattern architetturali** ed una successiva sezione descriventi i dettagli implementativi e le scelte progettuali per l'attuazione dei **design pattern**.

6.2.2 Pattern architetturali

6.2.2.1 Pattern *MVC*

Da quanto è emerso durante l'analisi dell'architettura del sistema software è stato presto chiaro che l'ottica verso la quale era necessario optato nella fase di progettazione della stessa sarebbe dovuta essere quella di definire stratificazioni in più livelli di astrazioni della progettazione che potessero interagire tra di loro. Il pattern architetturale (che rappresenta, inoltre, l'unico effettivo pattern implementato) mediante il quale è stato possibile definire una prima stratificazione nel sistema software è il **pattern MVC** (***Model View Controller***). La motivazione alla base della scelta della stratificazione (per maggiori informazioni analizzare quanto descritto nella sezione di descrizione del prodotto software e di progettazione architetturale) a livello di pattern architetturale fornita dal pattern bene si adattano con le condizioni e le funzionalità offerte dal pattern MVC.

Il pattern Model-View-Controller è un pattern architetturale, esso è basato sulla suddivisione del sistema software in tre componenti: **Model**, **View** e **Controller** che assolvono a compiti differenti ma complementari rispetto al sistema software dove agiscono. Descriviamo esplicitamente le componenti di tale pattern:

- **Model**: gestisce l'organizzazione dell'insieme dei dati necessari per il funzionamento del sistema software.
- **View**: gestisce la presentazione del sistema software e dell'interfaccia verso l'utente.
- **Controller**: gestisce come avviene l'interazione tra le componenti interne al sistema sulla base delle operazioni svolte.

Questo pattern è utilizzato maggiormente nell'eventualità in cui i dati possano essere rappresentati in maniera differente sulla base di differenti tipologie di utente e interazione con il sistema, e quindi si debbano suddividere la rappresentazione dei dati rispetto alla strutturazione dei dati vera e propria, notiamo che questo presupposto/requisito è ben presente nelle specifiche del nostro sistema software (dal momento che alcune interfacce utente possono presentare punti di condivisione nelle funzionalità tra gli utenti del sistema, perciò è necessaria una gestione oculata delle funzionalità ben attuabile mediante la stratificazione offerta dal pattern MVC), perciò si è deciso di procedere con l'implementazione del pattern stesso. Il modello teorico di pattern MVC da noi adottato è descritto dalla rappresentazione grafica fornita di seguito.

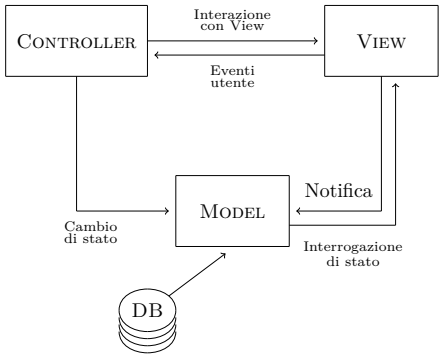

Figura 21: Rappresentazione grafica dell'organizzazione del *pattern MVC* all'interno del sistema software.

Nella pratica, il pattern MVC è stato implementato mediante gli strumenti forniti da linguaggi Object oriented e nello specifico Java, con una organizzazione in *package* come la seguente:

```
.../controller/
    AlertsHandlingController.java
    RilevazioniFarmaciHandlingController.java
    ContattaUtenteController.java
    RilevazioniGlicemiaHandlingController.java
    DashboardController.java
    RilevazioniHandlingController.java
    GestioneUtentiController.java
    RilevazioniSintomiHandlingController.java
    HomeController.java
    StatsController.java
    LoginController.java
    TerapieController.java
    MainController.java
    TerapieHandlingController.java
    PazientiController.java
    UserHandlingController.java
    ProfileController.java
    VisualizzazioneLogController.java
    RegisterController.java
.../model/
    /Admin/
        AdminDAO.java
        Admin.java
        AdminUser.java
    /Medico/
        Medico.java
        MedicoDAO.java
        MedicoUser.java
    /Paziente/
        Paziente.java
        PazienteDAO.java
        PazienteUser.java
    /Utils/
        AlertFilter.java
        LogAction.java
        RilevazioneGlicemia.java
```

```
        Alert.java
        Log.java
        Rilevazione.java
        AlertType.java
        Patologia.java
        RilevazioneSintomo.java
        Farmaco.java
        RilevazioneFarmaco.java
        Terapia.java
    DatabaseManager.java
    User.java
    UserDAO.java
    UserInterface.java
    UserType.java
    UserTypeNotFoundException.java
.../view/
    /components/
        NavBar.java
    ViewNavigator.java
.../fxml/
    AlertsHandlingView.fxml
    RilevazioniFarmaciHandlingView.fxml
    ContattaUtenteView.fxml
    RilevazioniGlicemiaHandlingView.fxml
    DashboardView.fxml
    RilevazioniHandlingView.fxml
    GestioneUtentiView.fxml
    RilevazioniSintomiHandlingView.fxml
    HomeView.fxml
    StatsView.fxml
    LoginView.fxml
    TerapieHandlingView.fxml
    MainView.fxml
    TerapieView.fxml
    PazientiView.fxml
    UserHandlingView.fxml
    ProfileView.fxml
    VisualizzazioneLogView.fxml
    RegisterView.fxml
```

Listing 1: Organizzazione dei package del sistema software implementante il pattern MVC.

6.2.3 Design pattern

La logica applicativa del sistema è stata implementata sulla base delle entità orientate agli *oggetti* definite nel capitolo dedicato al diagramma delle classi. Di fatto tali classi contengono la logica applicativa, ma inoltre la gestione delle transazioni e la elaborazione dell'I/O, nonchè l'interfacciamento con l'utente, il reperimento e modifica delle informazioni e la comunicazione del sistema con l'utente. Di fatto risulta complesso descrivere integralmente il sistema dal punto implementativo, forniremo quindi una diffusa descrizione dei concetti organizzativi e di progettazione sulla quale si è fondato lo sviluppo del sistema e quindi il codice sorgente stesso. Definiremo, a tal proposito, i **pattern architetturali** implementati nella fase di sviluppo del sistema software. Tuttavia al termine del capitolo scenderemo nello specifico con una analisi di alcuni dettagli/scelte implementative, includendo alcune componenti di codice java di interesse per la comprensione di come il sistema sia in grado di rispettare requisiti strutturali tanto quanto i requisiti funzionali.

6.2.3.1 Pattern *Singleton*

Il pattern ***singleton*** è pattern creazionale, ovvero un pattern che permette di risolvere problemi legati alla corretta instanza degli oggetti. Risponde al bisogno di avere una classe che effettua l'instanza di un solo oggetto e non permette di istanziare più oggetti per quella classe. Dal punto di vista teorico tale pattern si implementa nel paradigma Object-Oriented mediante l'imposizione delle seguenti condizioni: il costruttore deve essere privato, altrimenti più oggetti potrebbero chiamare quel costruttore, il metodo *getInstance()* restituisce l'unico oggetto che esiste per una classe, tipicamente denominato *instance*, esso sarà di tipo concorde all'oggetto su cui si implementerà il pattern. Nella variante *lazy* la variabile *instance* (che caratterizza il pattern) non viene instanziata in una fase iniziale ma solamente successivamente. Nel contesto del nostro sistema tale pattern si rende necessario dal momento che gli oggetti di logica applicativa, che effettuano il *reperimento e modifica delle informazioni* nonchè la gestione delle transazioni con il database, necessitano di essere univoche per rispettare i requisiti strutturali.

6.2.3.2 Pattern *Observer*

Il pattern ***observer*** è un pattern comportamentale che ci permette di definire un comportamento più complesso che coinvolge più oggetti (non c'è creazione). In questo pattern disponiamo di un *oggetto osservato* di cui alcuni *oggetti osservatori* vogliono sapere le informazioni sul suo cambio di stato; gli osservatori devono poter comunicare all'osservato di essere osservatori dello stesso *subject()*. Nella pratica tale pattern è maggiormente implementato nella situazione in cui un oggetto osservato cambia di stato per logica applicativa e tale cambiamento deve essere notificato agli osservatori utilizzando un metodo in possesso degli osservatori, come ciò che avviene tipicamente nei metodi *listener* di Java per componenti grafiche di varia natura. Il pattern Observer viene inoltre utilizzato frequentemente nelle situazioni in cui vi è una relazione uno-a-molti tra gli oggetti, ad esempio se un oggetto viene modificato, i suoi oggetti dipendenti devono essere notificati automaticamente. L'implementazione di tale pattern si rende necessaria dal momento che si necessita di una dinamica di osservazione a partire da alcune classi presenti nel *controller* rispetto ad altri oggetti interni osservati; ciò viene implementato sopratutto il legame al presupposto appena citato nella compagine delle componenti JavaFX presenti nella parte di View-Controller del pattern MVC. Di fatto le componenti JavaFX che costituiscono l'oggetto osservato sono di tipologia più svariata, ma principalmente possono rappresentare: *menù dropdown*, *bottoni*, *tabelle*.

```
1  import javafx.collections.FXCollections;
2  import javafx.collections.ObservableList;
3
4  public class ListaEsempio {
5      public static void main(String[] args) {
6          ObservableList<String> lista =
7          FXCollections.observableArrayList();
8
9          lista.addListener((
10         javafx.collections.ListChangeListener<String>) change -> {
11             while (change.next()) {
12                 if (change.wasAdded()) {
13                     System.out.println("Elementi aggiunti: "
14                     + change.getAddedSubList());
15                 }
16             }
17         });
18
19         lista.add("Mario");
20         lista.add("Giulia");
21     }
22 }
```

Listing 2: Implementazione del pattern observer nel sistema software.

6.2.3.3 Pattern *DAO*

Il design pattern **DAO** (*Data Access Object*) è un pattern architetturale che isola la logica di accesso ai dati (ad esempio, l'accesso a database, file, web API) dal resto della logica applicativa. Esso suddivide la struttura del sistema software, implementato mediante paradigma *object-oriented*, di modellazione dell'accesso al dato (individuabile, all'interno del nostro sistema, nella componente di *model* data dal pattern MVC) in tre componenti:

- Interfaccia di oggetti dato: definisce le operazioni standard sugli oggetti di tipo model.
- Classe concreta di accesso al dato: tale classe implementa l'interfaccia sopra-citata. Essa è responsabile di ottenere i dati dalla sorgente.
- Oggetti del modello o oggetti valore: si tratta di semplici oggetti *POJO* (*(Plain Old Java Object*), forniti di metodi *getter* e *setter* per gestire i ottenuti dalla classe concreta di accesso al dato.

Di fatto, grazie a questo design pattern avremo un insieme di classi che gestiscono collezioni di oggetti accessibili mediante opportuni metodi *getter* e *setter*, al fine di raggiungere in maniera definitiva la stratificazione del sistema software precedentemente anticipata. Forniamo ora la struttura delle componenti software che implementano il pattern DAO.

```
.../model/
    /Admin/
        AdminDAO.java
        Admin.java
        AdminUser.java
    /Medico/
        Medico.java
        MedicoDAO.java
        MedicoUser.java
    /Paziente/
        Paziente.java
        PazienteDAO.java
        PazienteUser.java
    /Utils/
        AlertFilter.java
        LogAction.java
        RilevazioneGlicemia.java
        Alert.java
        Log.java
        Rilevazione.java
        AlertType.java
        Patologia.java
        RilevazioneSintomo.java
        Farmaco.java
        RilevazioneFarmaco.java
        Terapia.java
    DatabaseManager.java
    User.java
    UserDAO.java
    UserInterface.java
    UserType.java
    UserTypeNotFoundException.java
```

Listing 3: Organizzazione delle componenti software del sistema software implementanti il pattern DAO.

6.2.4 Descrizione di dettagli implementativi

In questa sezione, come anticipato precedentemente, realizziamo una dettagliata disamina di alcuni dei principali dettagli implementativi che ci hanno permesso di rispettare requisiti strutturali (ma talvolta anche funzionali) imposti nella specifica. A tale scopo riportiamo di seguito i principali obiettivi dati dagli stessi: **interfaccia funzionale ed intuitiva**, con opportuna **diversificazione della interfaccia** (sulla base del loro ruolo), creazione del sistema di collegamento con servizio di mailing predefinito per il sistema al fine di predisporre il **contatto tra gli utenti del sistema**, **opportune performance** rispetto all'interazione con le pagine e con la base di dati (con le opportune interfacce di interazione con il dato), nonchè **opportune performance**, in un ottica generale (per l'utilizzatore) dell'**intero sistema software**. Allo scopo di descrivere i dettagli che ci hanno permesso di soddisfare tali obiettivi, analizziamo i metodi utilizzati in sede di sviluppo per aumentare le performance e la funzionalità della **UI** (User Interface) e del sistema software in generale. Nota che queste tecniche si pongono il fine ultimo di ottenere gli obiettivi appena citati in un ottica di caricamento dei dati necessari per la visualizzazione mediante UI e di velocizzazione della computazione svolta nella logica applicativa. Suddividiamo in sotto-sezioni le componenti software che ci hanno permesso di giungere agli obiettivi appena citati.

6.2.4.1 Componenti per aumento delle performance

Le operazioni realizzate con JavaFX se non gestite diversamente sono costituite da un unico thread principale che esegue le operazioni sull'UI chiamato 'JavaFX Application Thread'. Questo però può portare rallentamenti indesiderati nel caso di blocchi pesanti di codice in quanto l'UI sarà bloccata fino al termine delle operazioni rendendo l'esperienza di utilizzo dell'app poco reattiva e lenta negli input ed output.
Per poter risolvere questo problema nel codice sono state utilizzate le seguenti tecniche:

- **JavaFX Task**: è una classe speciale che lancia un thread in background per la gestione di operazioni particolarmente lente che rallenterebbero il thread principale come ad esempio operazioni di accesso al DB
- **Platform.runLater():** viene utilizzato da thread secondari per aggiornare l'UI in modo sicuro. Infatti richiamando questa funzione l'aggiornamento della UI viene messo in coda ed eseguito in seguito al completamento delle operazioni del thread

```
1  private void setup() {
2      //  Effettuo operazioni pesanti di caricamenti dal DB
3      evitando freezing della UI
4      Task<Void> loadDataTask = new Task<>() {
5          @Override
6          protected Void call() throws Exception {
7
8              //  Ottengo i dati dal DB
9              MedicoDAO medicoDAO = MedicoDAO.getInstance();
10             RilevazioneFarmaco[] rilevazioniFarmaci =
11             medicoDAO.getRilevazioniFarmaco(
12             ViewNavigator.getAuthenticatedUsername());
13             RilevazioneGlicemia[] rilevazioniGlicemia =
14             medicoDAO.getRilevazioniGlicemia(
15             ViewNavigator.getAuthenticatedUsername());
16             RilevazioneSintomo[] rilevazioniSintomi =
17             medicoDAO.getRilevazioniSintomo(
18             ViewNavigator.getAuthenticatedUsername());
19
20             ObservableList<RilevazioneFarmaco> rilFarmaci =
21             FXCollections.observableArrayList(rilevazioniFarmaci);
22             ObservableList<RilevazioneGlicemia> rilGlicemia =
23             FXCollections.observableArrayList(rilevazioniGlicemia);
```

```
24              ObservableList<RilevazioneSintomo> rilSintomi =
25              FXCollections.observableArrayList(rilevazioniSintomi);
26
27
28              Platform.runLater(() -> {
29                  setTimestampFormat();
30                  setupButtonsIcon();
31                  setupTimeSpinners();
32
33                  //  Operazioni di visualizzazione dati sulla UI
34              });
35
36              return null;
37          }
38      };
39
40      new Thread(loadDataTask).start(); // run the background task
41  }
```

Listing 4: metodo setup() di RilevazioniHandlingController

6.2.4.2 Gestione della funzione Timeline per gli alert

Per poter gestire due funzionalità del progetto che verranno spiegate di seguito si è scelto di utilizzare la componente **Timeline di JavaFX**; quest'ultima consente di eseguire azioni ripetute nel tempo ogni x secondi passati come parametro. Le funzionalità implementate grazie a questo meccanismo sono: **Aggiornamento tabella Alert**, **Monitoraggio degli alert in arrivo**.

6.2.4.3 Aggiornamento tabella Alert

Questa funzionalità è gestita quasi interamente nel Main alla chiamata del metodo *startAlertLockWatcher(updateIntervalSec, retryIntervalSec)*. Quest'ultimo ha come parametri due interi che rappresentano i secondi che il sistema aspetterà prima di riprovare ad eseguire rispettivamente le seguenti operazioni:

- **Aggiornamento vero e proprio della tabella Alert**: un'istanza attiva dell'applicazione si occupa di impostare una Timeline che ogni *updateIntervalSec* richiama la funzione userDAO.automaticUpdateAlertTable(). Questa funzione implementata nel DAO individua tutti gli alert che devono essere segnalati ai diabetologi e successivamente li inserisce nella tabella Alert
- **Ottenere il lock**: per garantire una maggiore sicurezza e migliori prestazioni è stato implementato un *meccanismo di lock* grazie al quale una sola istanza dell'applicazione si occupa di aggiornare la tabella Alert, le altre ogni *retryIntervalSec* verificano se il lock è occupato ed in caso negativo una di esse lo acquisisce tramite la funzione *acquireLockAlertHandler()* continuando poi l'aggiornamento regolare della tabella Alert come descritto nel punto precedente.

6.2.4.4 Monitoraggio degli alert in arrivo

Per poter notificare gli utenti della presenza di nuovi alert è stato inserito un bottone "Alert" sulla navbar che si colora di rosso nel caso in cui ci siano degli alert non gestiti dall'utente.
Il controllo avviene anche in questo caso ogni intervallo di tempo regolare tramite la funzione *startAlertPolling()* presente nella classe *NavBar*. Questa procedura ogni 2 secondi verifica in base al tipo di utente loggato la presenza di notifiche che sono accessibili cliccando sul pulsante "Alert" presente nella Navbar.
Solo una volta che l'utente loggato ha gestito tutti gli alert il pulsante sulla Navbar tornerà del suo colore originale.

6.3 Processo di sviluppo software

A questo punto della trattazione risulta utile definire l'ultima componente rimasta del metodo di sviluppo software ingegnerizzato, ovvero il **processo di sviluppo software**. La progettazione e sviluppo del software professionale (basato quindi sull'ingegnerizzazione del software) è dato dalla seguente uguaglianza, che bene riassume la relazione esistente tra metodo di progettazione e sviluppo e linguaggio di modellazione e processo software:

$$\text{Metodo} = \text{Linguaggio} + \text{Processo}$$

Il *linguaggio di modellazione* è la notazione per esprimere le caratteristiche del progetto. Al fine di comprendere al meglio la natura del linguaggio di modellazione, citiamo il fatto che UML per primo risulta disporre di estensioni per la caratterizzazione di elementi di object-oriented, al fine di avvicinarsi al linguaggio, ed inoltre ai processi di sviluppo, di tipo Object Oriented, inoltre che l'intera notazione e paradigma UML è orientato agli oggetti e alle loro proprietà. Per questo, come anticipato precedentemente, tale modello è stato il favorito per il nostro sistema software. Il termine modello indica, modelli standardizzati, ovvero un insieme di strumenti di progettazione e modellazione (principalmente diagrammatici) utile a documentare gli aspetti caratteristici delle componenti del software che si sta sviluppando.

Di fatto, un modello è un'astrazione che cattura le proprietà salienti della realtà che si desidera rappresentare (nel nostro caso il nostro sistema software), ed inoltre idealizza una realtà complessa, individuandone i tratti importanti e separandoli dai dettagli, facilitandone la comprensione. Astraendo il concetto di modello al concetto di linguaggio di modellazione è possibile avere una visione più ampia di quanto svolto da tali entità. Un linguaggio di modellazione fornisce le primitive a cui ricondurre la realtà in esame, e nel nostro caso il linguaggio fa riferimento a quanto formulato dal modello UML per il paradigma Object Oriented. Permette di esprimere le entità che compongono un sistema complesso, le loro caratteristiche e le relazioni che le collegano. Nell'ambito di un progetto, il linguaggio di modellazione è normalmente distinto dal processo di sviluppo.

Il processo di sviluppo software è l'insieme delle attività il cui fine ultimo è lo sviluppo del sistema software. Vi sono diverse tipologie di modelli di processo ma tutti coninvolgono le seguenti attività:

- *Specifica*: definisce cosa si intende ottenere dal sistema;
- *Progettazione e implementazione*: definisce l'organizzazione e l'implementazione del sistema;
- *Validazione*: fase di controllo nella quale si verifica se il prodotto ottiene quanto richiesto dall'utilizzatore;
- *Evoluzione*: consiste nella modifica del sistema dovuta alla evoluzione dei bisogni dell'utilizzatore;

Quando si descrivono e discutono i processi, di solito si descrivono le attività che fanno parte di questi processi, come ad esempio la specificazione di modelli di dati, progettazione dell'interfaccia utente, ecc. e l'ordine di queste attività. Effettuiamo una panoramica del processo di sviluppo software da noi adottato per il sistema in corso di studio.

6.3.1 Descrizione del processo di sviluppo software

Il processo di sviluppo software da noi attuato nella implementazione del sistema software è descritto di seguito. Disponiamo di *macro-attività* ***plan-driven*** e *micro-attività* ***agili***. Dal momento che i processi (o sotto attività di processi) "plan-driven" (anche detti *guidati da una pianificazione*) dispongono di attività pianificate a priori e il progresso è correlato al suddetto piano, esse bene si sono prestate per la definizione delle attività principali del processo, ovvero: *fase di specifica*, *fase di progettazione e implementazione*, *fase di test*. Mentre, dal momento che i processi (o sotto-attività di processi) di sviluppo software di tipo agile dispongono di pianificazione dinamica ed evoluzione in istanti di tempo successivi a quello iniziale, permettendo una maggiore flessibilità rispetto al cambiamento dei requisiti e delle richieste del cliente, essi ben si prestavano a rappresentare le fasi interne ad ognuna delle attività plan-driven appena elencate, in quanto predisponevano un elevato carico di variabili e di elementi da definire incrementalmente.

6.3.2 Modello di processo software

Il modello adottato per le macro-attività plan-driven è il **modello a cascata**. Di fatto, il modello a cascata è un modello plan-driven nella quale le attività sono poste idealmente in sequenza come una cascata (dove le attività di sviluppo sono analoghe all'acqua che prima di tornare alla fonte - inizio dello sviluppo software - deve raggiungere valle - ovvero la fine dello sviluppo software -). In essa vi sono fasi separate e distinte di specifica e sviluppo. Forniamo di seguito una rappresentazione grafica di tale modello:

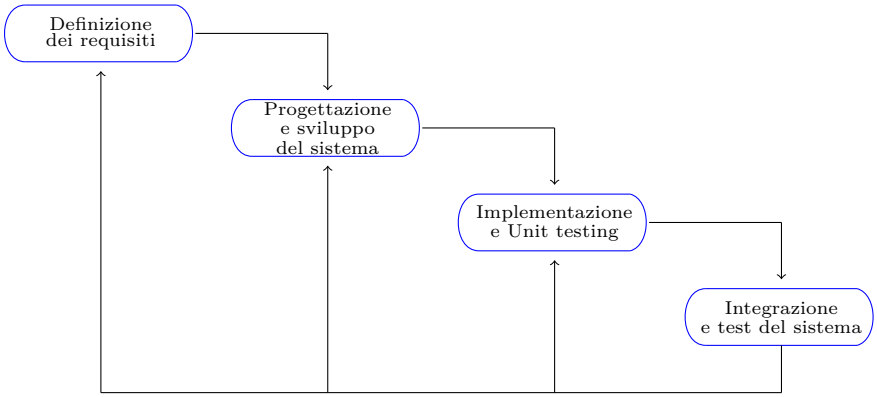

Figura 22: Rappresentazione grafica delle componenti del modello "*a cascata*".

Il modello rappresentato suggerisce che le fasi che lo compongono siano a tutti gli effetti separate; di fatto vi è retroazione tra le fasi solamente quando si giunge alla fine del processo di sviluppo software.

Questo modello risulta appropriato quando i requisiti sono ben noti e le modifiche sono limitate durante il processo di progettazione. Di fatto, l'inflessibilità della partizione delle attività del progetto in attività distinte rende difficile l'adattamento all'evoluzione delle necessità dell'utente del sistema. È risultato presto chiaro che questo modello ben si adattasse alle nostre esigenze come team di sviluppo, dal momento che le specifiche a noi fornite mediante la consegna non sarebbero evolute, e che esso potesse quindi costituire lo "scheletro" del nostro processo di sviluppo software.

Il modello adottato per le attività (interne alle macro-attività plan-driven appena descritte) è il **modello a sviluppo incrementale**. Tale modello di processo software dispone di 3 fasi di sviluppo che interagiscono ed eseguono parallelamente. Di fatto, possono essere implementati in maniera plan-driven o agile.

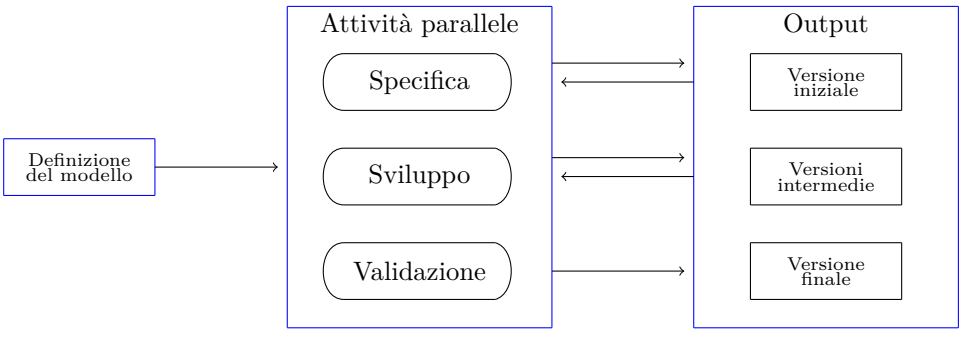

Figura 23: Rappresentazione grafica del modello "*incrementale*".

I benefici dello sviluppo incrementale sono dati da costi di gestione ed accomodamento dei cambiamenti ridotti, e da sviluppo e consegna del software con tempistiche più rapide e favorevoli.

Tali caratteristiche risultano ideali specialmente in fase di progettazione e sviluppo di componenti software, perciò tale modello è stato scelto per le micro-attività interne a quelle macro, plan-driven.

6.3.3 Consegna ed evoluzione del software

A questo punto risulta utile discutere la dinamica della consegna del sistema software. Di fatto, seppur implementando attività agili per le micro-attività interne a quelle macro, plan-driven, la consegna del sistema software avviene per - definizione della consegna del progetto -, in maniera **totale**, ovvero si prevede il *rilascio* dell'intero prodotto software in una sola consegna, ovvero un unità atomica di software utilizzabile dall'utente finale.

6.4 Documentazione processo sviluppo

La seguente sezione descrive il processo seguito per lanalisi, la progettazione e lo sviluppo del software realizzato. Il flusso di lavoro è stato modellato attraverso un diagramma di attività UML, che consente di rappresentare in modo chiaro e strutturato le fasi principali del progetto.
Il processo inizia con la consegna delle specifiche e prosegue con lelicitazione e lanalisi dei requisiti, da cui derivano i diagrammi dei casi duso, di sequenza e di attività. Tali diagrammi costituiscono la base per la progettazione architetturale del sistema.
Successivamente si passa alla progettazione software dettagliata, rappresentata dal diagramma delle classi, e allo sviluppo del codice. Le versioni prodotte vengono sottoposte a test di versione e a test di unità, con la possibilità di ritornare alle fasi precedenti in caso di errori o nuove esigenze.
Infine, il processo si conclude con lintegrazione della documentazione e il rilascio della versione finale del software che verrà infine esposto.

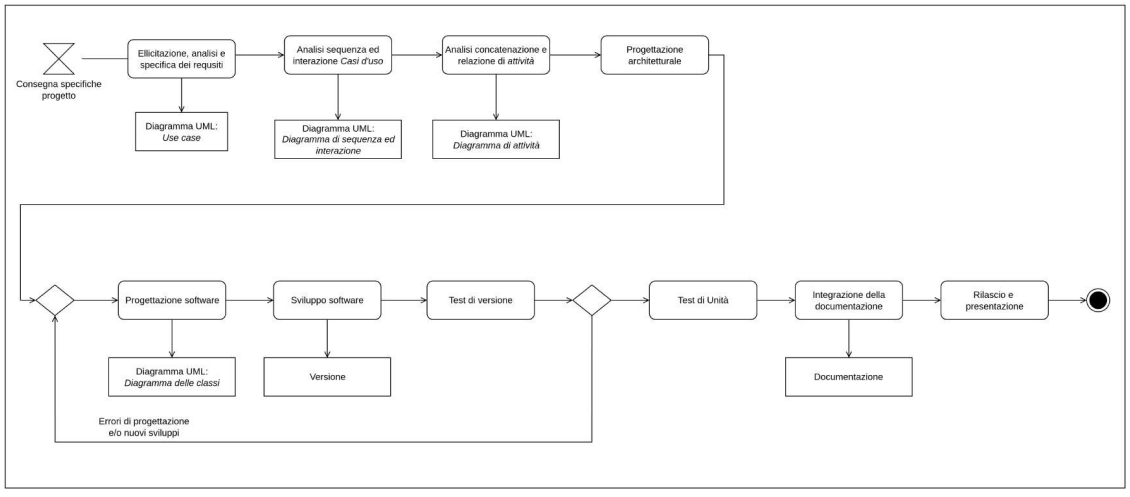

Figura 24: Raffigurazione grafica del diagramma delle attività rappresentante il processo di sviluppo.

Fase di Test

7.1 Unit Test - testing automatico con JUnit

Prima di eseguire i successivi tipi di test, è stato sviluppato e condotto un set completo di **unit test** automatici utilizzando il framework `JUnit 5`.
Tali test hanno lo scopo di verificare il corretto funzionamento delle principali classi di accesso ai dati (*DAO*) che compongono il modello dell'applicazione.

Isolamento tramite database di test

Per garantire l'affidabilità e la ripetibilità dei test, ogni esecuzione viene effettuata su un **database SQLite isolato e temporaneo**, generato dinamicamente a partire da uno **schema SQL pulito**. Questo approccio offre i seguenti vantaggi:

- evita di contaminare il database principale in uso dall'applicazione;
- garantisce che ogni test inizi da uno stato iniziale noto e controllato;
- consente di ripetere i test infinite volte in maniera consistente;
- permette l'individuazione più semplice di errori in quanto lo schema è sempre lo stesso

Il database viene inizializzato all'interno della classe `SQLiteTestDatabase`, che:

- cancella il file di database test esistente (se presente);
- esegue lo schema SQL passato come parametro;
- fornisce un oggetto `DatabaseManager` associato alla connessione.

Struttura dei test

I test sono organizzati in classi corrispondenti ai vari DAO presenti nel progetto:

- `AdminDAOTest`
- `MedicoDAOTest`
- `PazienteDAOTest`
- `UserDAOTest`

Ogni test segue il pattern `@BeforeAll` per la preparazione e `@AfterAll` per il teardown, dove viene chiusa la connessione e cancellato il file del database test.

Casi coperti

Di seguito un elenco rappresentativo dei test implementati:

- **Creazione, aggiornamento e cancellazione di utenti (Admin, Medico, Paziente)** tramite `UserDAO`;
- **Inserimento, aggiornamento e eliminazione di Terapie** con `MedicoDAO`, inclusa l'associazione con patologie e pazienti;
- **Inserimento di rilevazioni** (glicemia, sintomi, farmaci) e loro validazione nel database con `PazienteDAO`;
- **Recupero e verifica di alert**, sia per i pazienti che per i medici, e aggiornamento automatico della tabella `Alert`;

Esempio di test significativo

Ad esempio, il metodo `testSetTerapiaPazienteInsertAndUpdate()` verifica:

1. la corretta creazione di una terapia associata a un paziente e una patologia richiamando `medicoDAO.setTerapiaPaziente(...)`;
2. il recupero e verifica della presenza della terapia nel database tramite `medicoDAO.getTerapiePaziente(username)`
3. la modifica della stessa terapia e il controllo del nuovo stato aggiornato richiamando `medicoDAO.updateTerapiaPaziente(newTerapia)` e successivamente `medicoDAO.getTerapiePaziente(username)`

Conclusioni

L'infrastruttura di testing automatizzata consente di mantenere elevata l'affidabilità del codice, prevenire regressioni e garantire che le operazioni sui dati siano coerenti. L'uso di un database isolato è stato fondamentale per ottenere risultati stabili e indipendenti dallo stato dell'applicazione in esecuzione. I test JUnit costituiscono la base della verifica funzionale interna prima di ogni ciclo di sviluppo o refactoring.

7.2 Test da parte di utenti esperti

Nel corso del progetto, è stata svolta un'attività di collaudo da parte di un team di sviluppo esterno composto da **Fozzato Davide** e **Rebonato Mattia**, al fine di identificare eventuali malfunzionamenti o casi d'uso non correttamente gestiti dall'applicazione.
Di seguito sono riportati alcuni scenari significativi rilevati durante l'utilizzo dell'applicativo.

7.2.1 Scenario 1: Click su riga vuota di una tabella

Contesto: L'utente visualizza una tabella con associato un trigger sul click di riga.
Azione: L'utente clicca accidentalmente su una riga vuota.
Comportamento osservato: Viene generata un'eccezione `NullPointerException`, in quanto l'elemento selezionato è nullo.
Intervento effettuato: È stato inserito un controllo di nullità per evitare l'esecuzione del trigger se la riga selezionata è vuota.

7.2.2 Scenario 2: Inserimento rilevazione farmaco senza note

Contesto: L'utente vuole inserire una rilevazione di un farmaco.
Azione: L'utente lascia vuoto il campo *note*.
Comportamento osservato: L'inserimento viene rifiutato, in quanto il campo note è obbligatorio.
Osservazione del team: Il campo note dovrebbe essere facoltativo, poiché l'utente potrebbe non avere nulla da segnalare.
Intervento effettuato: Il campo note è stato reso opzionale nel controllo di validazione.

7.2.3 Scenario 3: Inserimento di testo in campi numerici

Contesto: L'applicazione presenta campi per l'inserimento di valori numerici (es. dosi giornaliere, quantità per dose).
Azione: L'utente inserisce caratteri testuali in un campo numerico.
Comportamento osservato: L'applicazione accetta il valore o genera un errore a runtime.
Intervento effettuato: Sono stati applicati filtri ai campi di input tramite `TextFormatter`, per permettere solo l'inserimento di numeri.

Queste attività di testing sono state fondamentali per individuare edge case non precedentemente considerati, e hanno contribuito al miglioramento della stabilità dell'applicazione prima del rilascio.

7.3 Release Testing - test utente esterno

Dopo i test condotti dal team di sviluppo esterno, è stato svolto un test con un **utente non esperto** con l'obiettivo di valutare l'usabilità dell'interfaccia e la chiarezza delle funzionalità offerte dall'applicazione. Questo tipo di verifica permette di evidenziare problematiche legate all'esperienza utente che potrebbero non emergere durante i test funzionali o di unità.
Il test è stato svolto in modalità esplorativa: l'utente ha utilizzato liberamente le funzionalità principali dell'applicazione e ha fornito feedback spontanei durante e dopo le interazioni. Di seguito sono riportati i principali scenari riscontrati, con le azioni correttive implementate dal team di sviluppo.

Scenario 1: Mancato reset dei campi nella sezione "Contatta utente"

Comportamento osservato: durante il cambio del destinatario nella sezione *Contatta utente*, l'utente ha notato che i campi relativi a oggetto e corpo del messaggio rimanevano popolati con i valori precedenti, creando confusione.
Soluzione adottata: al cambio utente:

- i campi oggetto e corpo vengono automaticamente sbiancati quando si clicca sul bottone "Contatta" o quanso si cambia utente
- il pannello viene temporaneamente chiuso per comunicare che l'operazione è andata a buon fine; viene poi lanciata l'applicazione email predefinita del sistema operativo.

Scenario 2: Inserimento di unità di misura in campi numerici

Comportamento osservato: l'utente, durante l'inserimento di una nuova terapia, ha provato a digitare nel campo `quantità per dose` la stringa `"5 mg"` e nel campo `dosi giornaliere` la stringa `"2 al giorno"`.
Problemi identificati:

- I `TextField` accettavano input testuali, provocando errori non gestiti o inconsistenza nei dati.
- L'unità di misura non era indicata nella label del campo, inducendo l'utente a includerla manualmente.

Soluzione adottata:

- È stato applicato un `TextFormatter` per accettare solo input numerici (interi o decimali).
- Le label dei campi sono state aggiornate per includere le unità di misura, ad esempio: `Quantità per dose [mg]`.

Scenario 3: Mancanza di conferma visiva per operazioni di inserimento e modifica

Comportamento osservato: dopo aver effettuato un inserimento o una modifica (es. terapia), l'utente non riceveva alcuna conferma esplicita dell'avvenuto successo dell'operazione.
Rischio percepito: incertezza sull'effettivo salvataggio delle informazioni, con possibilità di ripetere involontariamente l'operazione.
Soluzione adottata:

- In caso di successo, viene mostrato un `Alert` di conferma (finestra modale).
- In caso di errore, viene mostrata una `Label` con messaggio di errore ben visibile e stilizzata in rosso.

Questo comportamento è stato esteso anche all'inserimento delle rilevazioni (`Farmaco`, `Glicemia`, `Sintomo`).

Scenario 4: Bug nella registrazione utente da parte dell'admin

Comportamento osservato: durante l'inserimento di un nuovo utente da parte dell'amministratore, se non veniva selezionato il tipo utente (Paziente, Medico o Admin), l'app mostrava comunque un messaggio "Utente inserito con successo", sebbene nessuna riga fosse effettivamente salvata nel database.
Gravità: alto rischio di malintesi e inconsistenza tra interfaccia e stato reale dell'applicazione.
Soluzione adottata:

- È stato introdotto un controllo obbligatorio sulla selezione del tipo di utente.
- Se il tipo non è selezionato, viene mostrato un messaggio di errore tramite una `Label` rossa.

Conclusione:
Questo test ha evidenziato l'importanza del feedback visivo e dell'usabilità dell'interfaccia. Sebbene non si tratti ancora di *release test* formali, gli scenari emersi hanno permesso di migliorare sensibilmente l'esperienza d'uso, con interventi mirati su controlli, feedback e gestione degli errori. Il feedback dell'utente non esperto si è rivelato cruciale per garantire un'applicazione più intuitiva e robusta evidenziando aspetti di più difficile individuazione per noi programmatori.

Riferimenti bibliografici

[Som07] I. Sommerville. *Ingegneria del software*. Paravia/Bruno Mondadori, 2007. ISBN: 9788871923543. URL: https://books.google.it/books?id=h-hCKFMbqNMC.

[AA09] N. Ashrafi e H. Ashrafi. *Object-oriented Systems Analysis and Design*. Pearson Education, 2009. ISBN: 9780131354791. URL: https://books.google.it/books?id=PlpDPAAACAAJ.

[Pog15a] Francesco Poggi. "Diagrammi di sequenza". In: (2015). A.A. 2015-2016.

[Pog15b] Francesco Poggi. "I diagrammi di attività e stato". In: (2015). Dal materiale di Angelo di Iorio e Gian Piero Favini, A.A. 2015-2016.

[Pog15c] Francesco Poggi. "Il diagramma dei casi d uso". In: (2015). Dal materiale del Prof. Ciancarini, Dott. Favini e Di Iorio, Dott.ssa Zuppiroli, A.A. 2015-2016.

[Pog15d] Francesco Poggi. "Introduzione a UML". In: (2015). Dal materiale del Prof. Ciancarini, Dott. Favini e Di Iorio, Dott.ssa Zuppiroli, A.A. 2015-2016.

[Lar21] C. Larman. *Applicare UML e i pattern: analisi e progettazione orientata agli oggetti*. Pearson, 2021. ISBN: 9788891924193. URL: https://books.google.it/books?id=uVjszgEACAAJ.

[Som21] I. Sommerville. *Introduzione all'ingegneria del software*. Pearson, 2021. ISBN: 9788891915276. URL: https://www.hoepli.it/libro/ingegneria-del-software-ediz-mylab-con-etext-con-aggiornamento-online/9788891902245.html.

[Com] Carlo Combi. "Sistemi orientati agli oggetti, Concetti di base". In: (). Computer Science Department, University of Verona, Italy.

[SQL] SQLite. *Sqlite - Website*. URL: https://www.sqlite.org/ (visitato il giorno 08/07/2025).

[Tuta] Tutorialspoint. *Tutorialspoint Design patterns - Facade pattern*. URL: http://www.tutorialspoint.com/design_pattern/facade_pattern.htm (visitato il giorno 01/04/2019).

[Tutb] Tutorialspoint. *Tutorialspoint Design patterns - Factory pattern*. URL: http://www.tutorialspoint.com/design_pattern/factory_pattern.htm (visitato il giorno 01/04/2019).

[Tutc] Tutorialspoint. *Tutorialspoint Design patterns - Iterator pattern*. URL: http://www.tutorialspoint.com/design_pattern/iterator_pattern.htm (visitato il giorno 01/04/2019).

[Tutd] Tutorialspoint. *Tutorialspoint Design patterns - Observer pattern*. URL: http://www.tutorialspoint.com/design_pattern/observer_pattern.htm (visitato il giorno 01/04/2019).

[Tute] Tutorialspoint. *Tutorialspoint Design patterns - Proxy pattern*. URL: http://www.tutorialspoint.com/design_pattern/proxy_pattern.htm (visitato il giorno 01/04/2019).

[Tutf] Tutorialspoint. *Tutorialspoint Design patterns - Singleton pattern*. URL: http://www.tutorialspoint.com/design_pattern/singleton_pattern.htm (visitato il giorno 01/04/2019).

[Tutg] Tutorialspoint. *Tutorialspoint Design patterns - Template pattern*. URL: http://www.tutorialspoint.com/design_pattern/template_pattern.htm (visitato il giorno 01/04/2019).

[Tuth] Tutorialspoint. *Tutorialspoint Design patterns- Abstract factory pattern*. URL: http://www.tutorialspoint.com/design_pattern/abstract_factory_pattern.htm (visitato il giorno 01/04/2019).