Documentazione progetto Ingegneria del Software

Oliviero Nardi, VR406926

Luglio 2018

Contents

Requisiti ed interazioni utente-sistema

Specifiche casi d'uso

Note generali

Il sistema proposto supporta l'utilizzo da parte del personale medico. Il personale è suddiviso in "Medici" e "Farmacologi". Entrambe le categorie hanno a disposizione delle credenziali (pre-fornite dagli amministratori di sistema) con cui possono effettuare l'autenticazione. In caso questa vada a buon fine, il personale verrà reindirizzato alla rispettiva schermata iniziale, a seconda della professione associata alle credenziali.

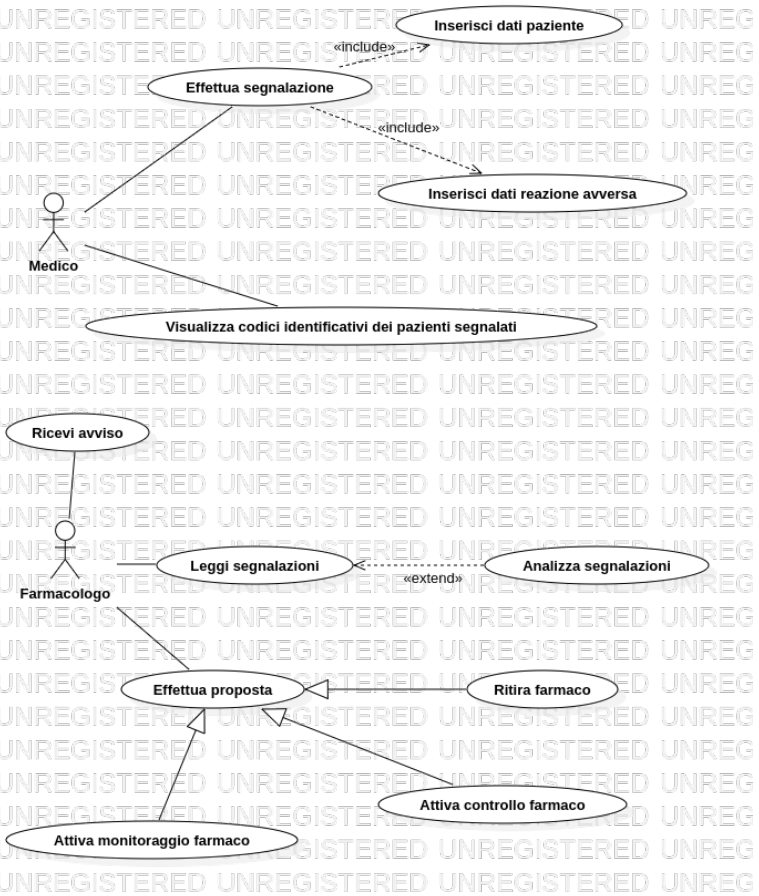

Figure 1: Casi d'uso

Casi d'uso relativi ai Medici

Dopo opportuna autenticazione, il medico viene introdotto ad una interfaccia che permette l'inserimento dei dati delle reazioni avverse e dei pazienti.

Effettuare segnalazioni

I medici devono poter effettuare delle segnalazioni. Per fare questo, è necessario **inserire i dati paziente** ed **inserire i dati della reazione avversa**. La data della segnalazione sarà gestita automaticamente.

Attori: *Medico*
Precondizioni: *Il medico dev'essersi autenticato.*
Passi:

1. *Il medico accede al sistema*
2. *Il medico è introdotto all'interfaccia di base*
3. *Il medico accede all'interfaccia per le segnalazioni*
4.
   (a) *Il medico inserisce i dati relativi al paziente*
   (b) *Il medico seleziona un paziente tra quelli a suo carico*
5.
   (a) *Il medico inserisce una nuova reazione avversa*
   (b) *Il medico seleziona una reazione avversa dalla memoria del sistema*
6. *Il medico inserisce la data della reazione avversa*
7. *Il medico conferma la segnalazione*

Postcondizioni: *la segnalazione è inserita*

In seguito, ulteriori dettagli riguardo all'inserimento dati paziente e reazioni avverse.

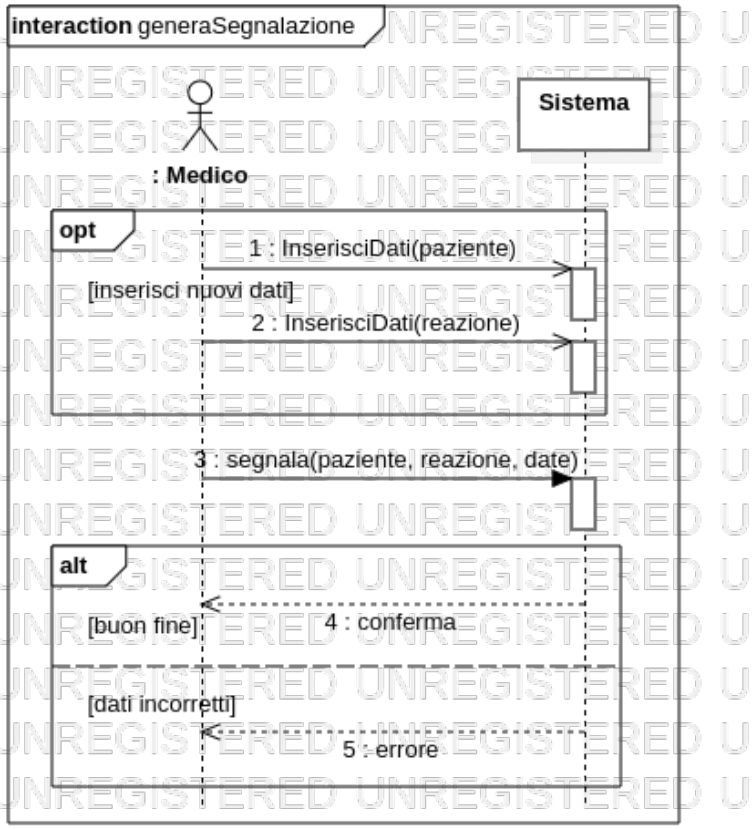

Figure 2: Fai una segnalazione

Inserire dati paziente

In questa fase, vi sono due alternative:

- Compilazione di una nuova scheda paziente, completa di professione, provincia di residenza, fattori di rischio, anno di nascita, terapie farmacologiche in uso;
- Selezione di una scheda utente pre-esistente, **tra quelle a carico del medico in questione.**

L'identificativo univoco del paziente sarà gestito dal sistema.

Inserire dati della reazione avversa

In questa fase, vi sono due alternative:

- Inserimento in sistema di una nuova reazione avversa, immettendo nome *(univoco)*, gravità e descrizione;
- Selezione di una reazione avversa nota, pre-esistente nel sistema.

In ambo i casi, sarà necessario inserire la data della reazione avversa: questa dovrà essere consistente sia con la data della segnalazione sia con la data delle terapie del paziente.

Visualizzare i codici identificativi dei pazienti segnalati

Sarà anche possibile visualizzare una lista di codici identificativi (gestiti automaticamente dal sistema) di pazienti di cui il medico ha effettuato una segnalazione. Per aumentare la **tracciabilità** e l'**affidabilità** del sistema, è stata aggiunta la possibilità di visualizzare alcuni dettagli interessanti riguardo ai singoli pazienti visualizzati.

Attori: *Medico*
Precondizioni: *Il medico dev'essersi autenticato*
Passi:
1. Il medico accede al sistema
2. Il medico è introdotto all'interfaccia di base
3. Il medico accede all'interfaccia per la visualizzazione dei codici identificativi
4. Il medico visualizza i codici identificativi dei pazienti per cui ha effettuato segnalazioni
Postcondizioni: *nessuna*

Attori: *Medico*
Precondizioni: *Il medico deve aver visualizzato la lista di pazienti*
Passi:
1. Il medico seleziona un paziente
2. Il medico seleziona l'opzione "visualizza"
3. Il sistema fornisce le informazioni base riguardo il paziente selezionato
Postcondizioni: *nessuna*

Dal test sul software è emersa anche la possibilità di aggiungere nuove terapie ad un paziente già inserito in sistema. Tale funzionalità è stata aggiunta nel menù di visualizzazione dei pazienti a carico del medico.

Attori: *Medico*
Precondizioni: *Il medico deve aver visualizzato la lista di pazienti*

Passi:

1. *Il medico seleziona un paziente*
2. *Il medico seleziona l'opzione "aggiungi"*
3. *Il medico inserice le informazioni riguardanti la nuova terapia*

Postcondizioni: *La terapia dev'essere stata memorizzata*

Casi d'uso relativi ai Farmacologi

Ricezione avvisi

Il sistema deve fornire un meccanismo di gestione ed invio di avvisi verso i farmacologi. Il sistema può inviare avvisi in tre casi:

1. Avviso settimanale (inviato ogni fine settimana)
2. Quando il numero di segnalazioni raggiunge la soglia di 50
3. Quando un farmaco accumula almeno 10 segnalazioni con gravità superiore a 3 in un anno

Il sistema avverte uno dei tanti farmacologi responsabili. Ciò significa che il primo farmacologo ad autenticarsi riceverà gli avvisi generati e non ancora letti: questo avviene in forma di pop-up e prima di vedere la schermata iniziale. È inoltre possibile, tramite un'opzione nel menù principale, rivedere gli avvisi già letti.

Attori: *Farmacologo*
Precondizioni: *Il farmacologo dev'essersi autenticato, devono esservi degli avvisi nel sistema*
Passi:

1. *Il farmacologo accede al sistema*
2. *Il farmacologo riceve gli avvisi non ancora letti*

Postcondizioni: *Gli avvisi devono essere segnati come già letti*

Attori: *Farmacologo*
Precondizioni: *Il farmacologo dev'essersi autenticato*
Passi:

1. *Il farmacologo accede al sistema*
2. *Il farmacologo visualizza l'interfaccia di base*
3. *Il farmacologo seleziona l'interfaccia per la lettura degli avvisi già letti*

Postcondizioni: *Gli avvisi già letti vengono visualizzati*

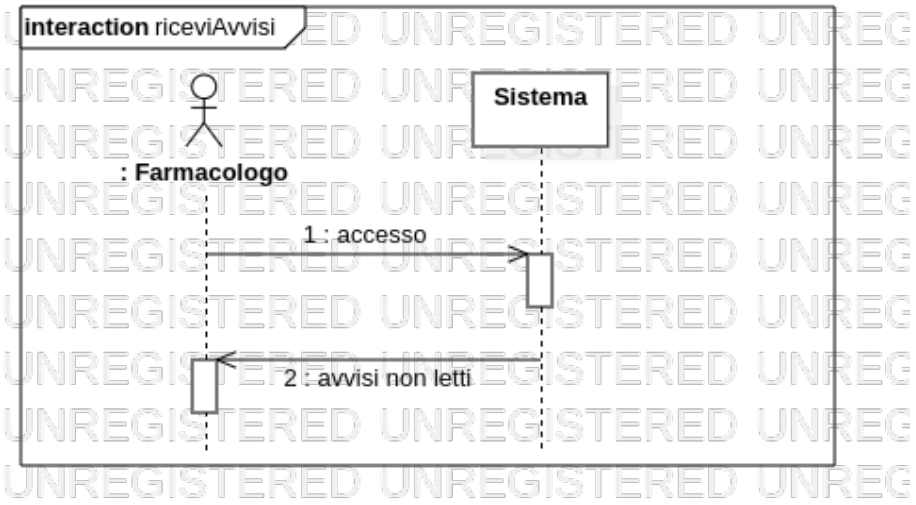

Figure 3: Ricezione avvisi

Leggi segnalazioni

Il farmacologo dopo autenticazione può vedere la lista di segnalazioni presenti in memoria. Può vederne i dettagli ed **effettuare delle analisi.**

Attori: *Farmacologo*
Precondizioni: *Il farmacologo dev'essersi autenticato*
Passi:

1. *Il farmacologo accede al sistema*
2. *Il farmacologo visualizza l'interfaccia di base*
2. *Il farmacologo Il farmacologo accede all'interfaccia per il controllo delle segnalazioni*

Postcondizioni: *Il farmacologo visualizza le segnalazioni. Potrà ora passare alla fase di analisi*

Effettua analisi

Il farmacologo può effettuare alcune analisi di base sulle segnalazioni presenti in memoria. Sono state inserite le seguenti opzioni, selezionabili dall'apposito menù:

- Conta segnalazioni per gravità
- Conta farmaco per gravità
- Conta segnalazioni per farmaco

- Conta segnalazioni gravi (>3) in settimana

Il sistema risponderà con i dati richiesti.

Attori: *Farmacologo*
Precondizioni: *Il farmacologo dev'essersi autenticato e aver visualizzato le segnalazioni*
Passi:

1. *Il farmacologo seleziona l'opzione per le analisi*
2. *Il farmacologo seleziona l'analisi che gli interessa*

Postcondizioni: *Il sistema fornisce la risposta*

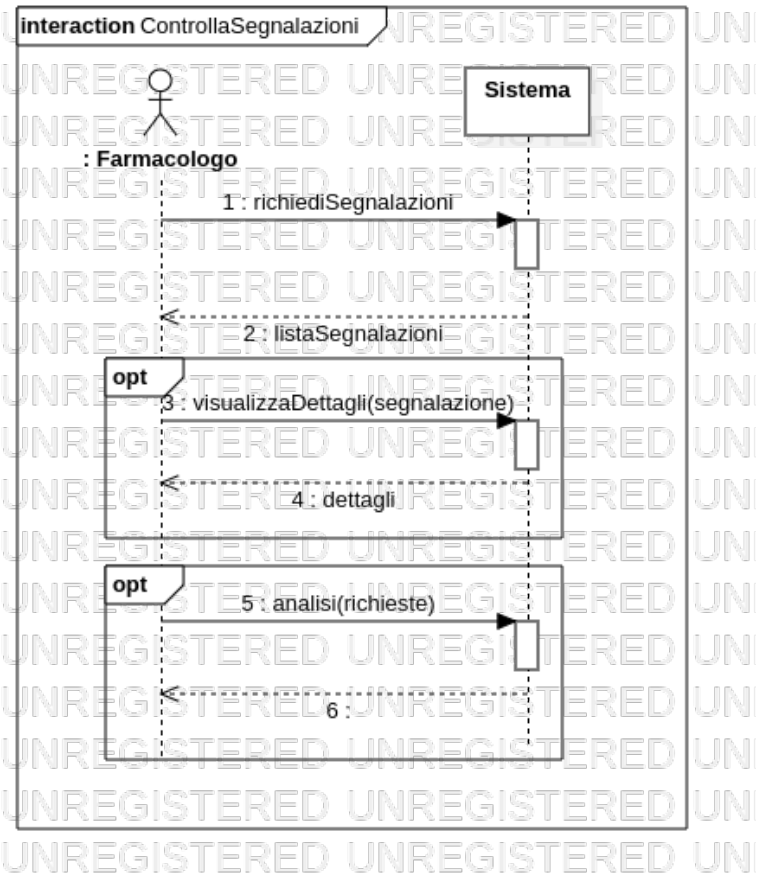

Figure 4: Controllo delle segnalazioni

Effettua proposte

In base alle analisi eseguite e alla lettura delle segnalazioni, il farmacologo può effettuare delle proposte sui farmaci presenti in memoria. Il sistema deve registrare e tracciare queste proposte per ogni farmaco. Le proposte possibili sono:

- Ritirare il farmaco dal commercio immediatamente
- Attivare una fase di controllo per il farmaco
- Mettere il farmaco tra quelli che richiedono un monitoraggio più attento

Attori: *Farmacologo*
Precondizioni: *Il farmacologo dev'essersi autenticato*
Passi:

1. *Il farmacologo accede al sistema*
2. *Il farmacologo accede all'interfaccia per le proposte*
3. *Il farmacologo seleziona il farmaco e le relative proposte*
4. *Il farmacologo dà la conferma*

Postcondizioni: *Le proposte devono essere memorizzate e tracciate dal sistema*

Diagrammi di attività

***Nota**: i seguenti diagrammi catturano una singola attività di un utente rispetto al sistema. Non è stata rappresentata nel diagramma la possibilità di ripetere più volte la stessa operazione, in sequenza, senza chiudere il software. Questo per semplici ragioni di chiarezza e leggerezza; si considerano quindi le singole attività d'interazione.*

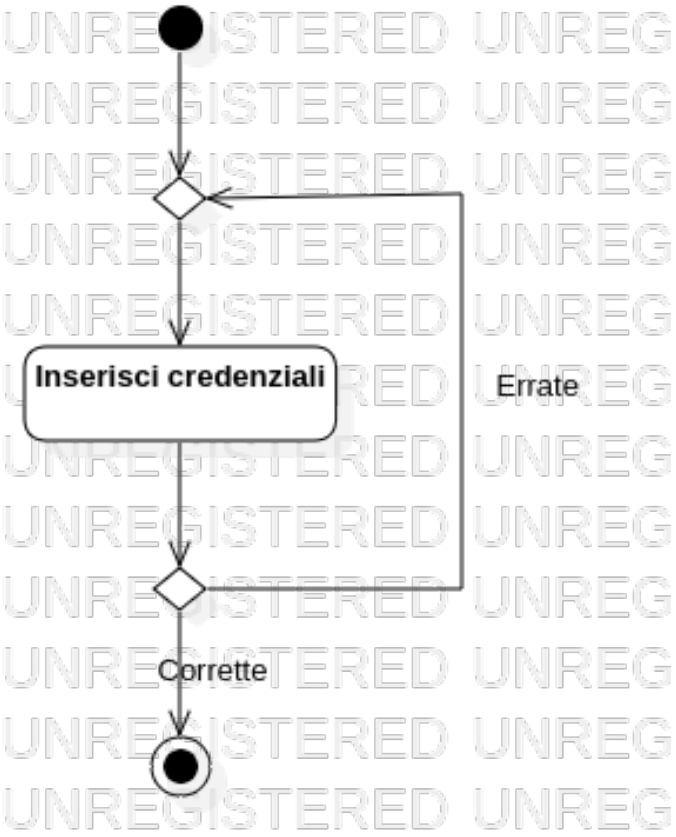

Figure 5: Autenticazione

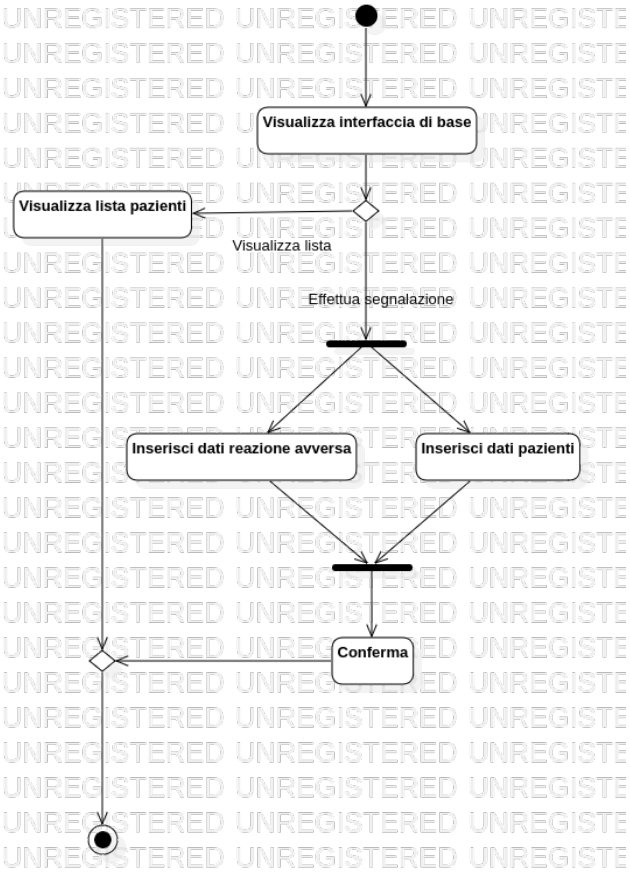

Figure 6: Attività medico

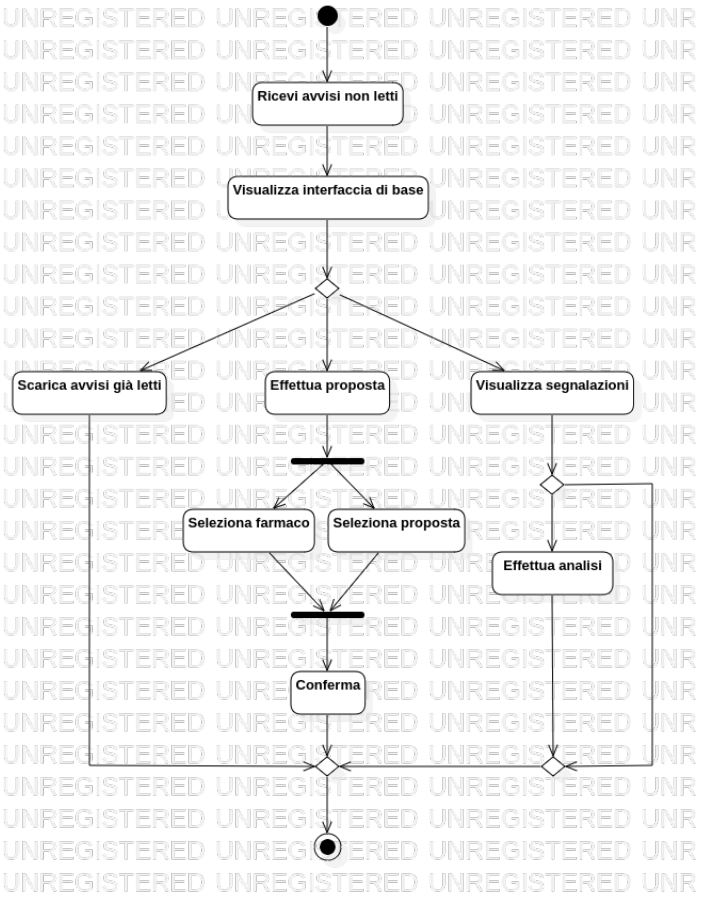

Figure 7: Attività farmacologo

Sviluppo: progetto dell'architettura ed implementazione del sistema

Note sul processo di sviluppo

Il processo di sviluppo è stato essenzialmente di tipo *Agile* ed *Incrementale*. Tuttavia, si è cercato quanto più possibile di mantenere sequenziali (seppure inserite all'interno di un ciclo) le fasi di progettazione, implementazione e validazione. Questo è stato fatto semplicemente per cercare di procedere sempre *prima* progettando e *dopo* implementando, con l'obiettivo di avere sempre un terreno solido e pensato sul quale lavorare. Dopo ogni modifica significativa (*versione*) è stata condotta una breve attività di test. Si noti che queste fasi non sono sempre state lineari e propellenti, ma hanno anche incluso attività di *refactoring* sul materiale già costruito.
Durante ogni ciclo, e parallelamente a queste tre attività, si è condotta ciò che possiamo dire genericamente *documentazione*. In questa attività, si è raccolto il materiale UML generato dalle fasi di test e di progettazione (principalmente *diagrammi di classe*) nel presente documento complessivo, aggiungendo inoltre ulteriori UML descrittivi (i *sequence diagram*, ad esempio, sono stati prodotti dopo aver implementato i metodi relativi).
Prima di cominciare il ciclo principale, si è condotta la fase di analisi dei requisiti, generando i relativi *use-cases* e i *diagrammi di attività*. Anche la (semplice) progettazione architetturale è stata fatta prima di iniziare il lavoro centrale, in modo da assicurarsi di trovarsi per lo meno in una situazione solida da quel punto di vista.
Per quanto riguarda l'implementazione, non sono state fatte grosse divisoni o piani di sviluppo programmatici. Si è seguito l'ordine prioritario di sviluppo andando ad aggiungere ciò che era ritenuto necessario man mano.

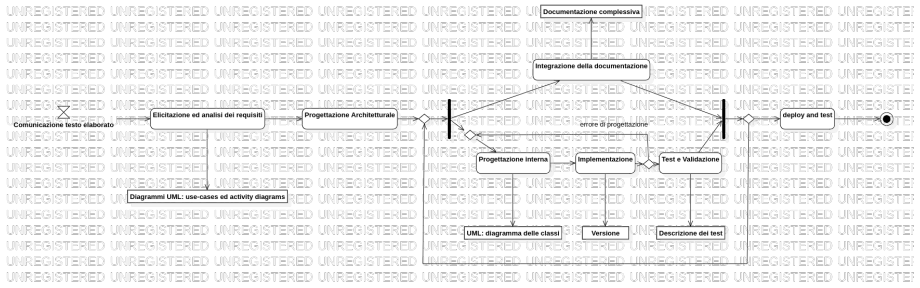

Figure 8: Processo di sviluppo

Progettazione e pattern architetturali usati

Il sistema è stato progettato utilizzando le tecniche di modellazione ad oggetti. Dal punto di vista architetturale, si è scelto di utilizzare il **pattern MVC**. Le ragioni sono semplici: si è pensato che, essendo nativamente implementato tramite le librerie *Swing* (utilizzate per produrre l'interfaccia grafica del progetto), fosse la scelta più comoda e solida.
Ciò ha permesso di separare nettamente e comodamente le tre componenti a livello logico e a livello di sviluppo. Ogni membro è stato inserito in un **package** separato.

- **Modello**: sottoparte del sistema che riguarda i dati e le informazioni memorizzate. Definisce la struttura dell'informazione gestita dal sistema.
- **Vista**: sottoparte del sistema che rappresenta visivamente il modello, e quindi, i dati del sistema. Si è utilizzato il framework Swing per realizzare questa parte.
- **Controllore**: sottoparte del sistema che definisce la logica applicativa, ovvero il comportamento del sistema a fronte degli stimoli esterni. Consiste nei listener che ascoltano la vista.

Le azioni dell'utente saranno catturate dai listener. I listener saranno istruiti a reagire di conseguenza, andando a modificare le informazioni contenute nel Modello e quindi ad aggiornare la vista, che rappresenta il modello.
Non sono state operate ulteriori articolazioni: modello, vista e controllore sono tre blocchi monolitici. Questo perché si è considerato che i vari elementi costitutivi giacessero sullo stesso piano.
Seguono un semplice schema dell'architettura del sistema ed i diagrammi UML delle classi del Modello e della Vista-Controllore.
Le interazioni salienti tra Modello-Vista-Controllore saranno visibili nei diagrammi di sequenza (cfr. *Implementazione*).

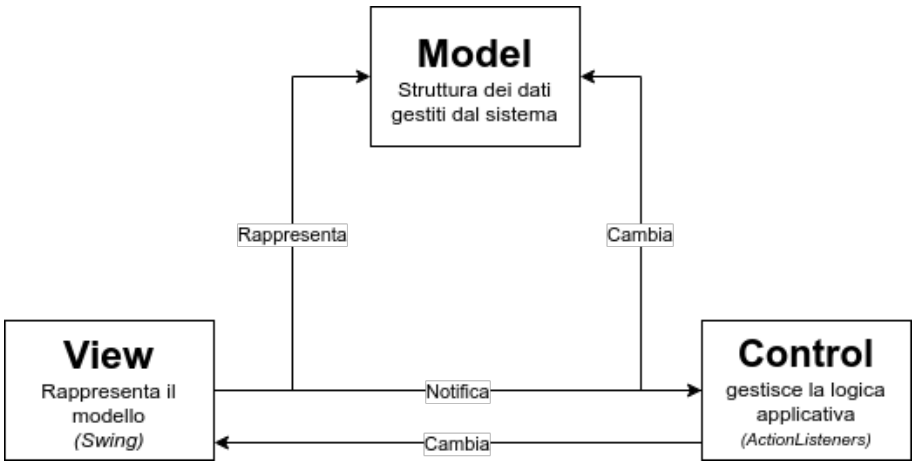

Figure 9: Architettura MVC

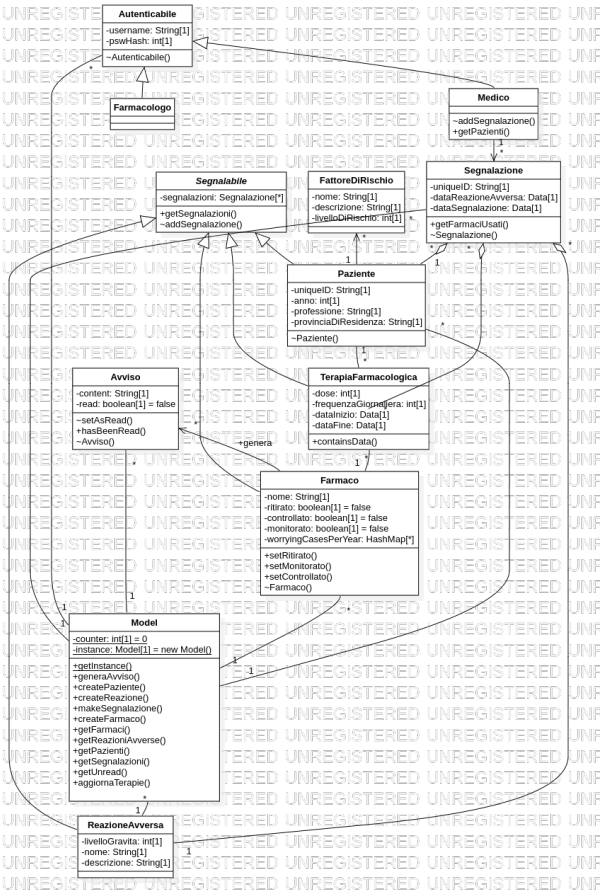

Figure 10: Diagramma delle classi del modello

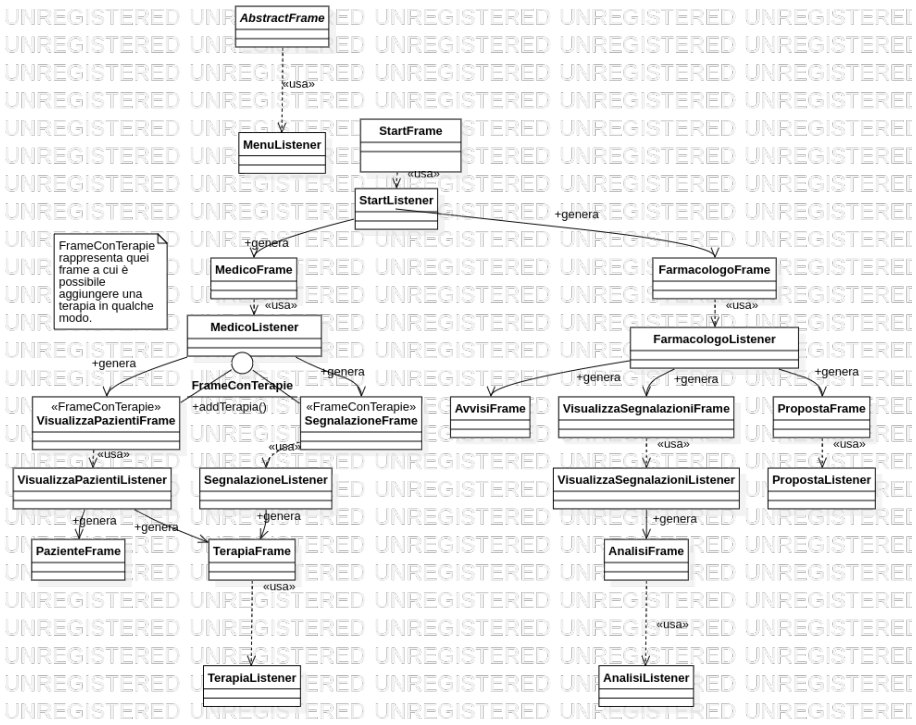

Figure 11: Diagramma delle classi della vista-controllore. *Nota: tutti i frame ereditano da AbstractFrame*

Breve commento sulle interfacce

Per gestire la comunicazione tra Modello, Vista e Controllore, sono state adottate le seguenti scelte.
Per la comunicazione Vista-Controllore, si è banalmente utilizzato il meccanismo basato su *ActionListener*s. L'implementazione è standard, per cui possiamo sorvolare.
Per la comunicazione Modello-Vista e Modello-Controllore, si è utilizzato come punto di accesso alle informazioni la classe *Model*, che immagazzina e fornisce i dati. Per i dettagli riguardo la sua implementazione si rimanda alla sezione successiva; segue un breve riassunto delle comunicazioni Modello-Vista e Modello-Controllore per quanto concerne l'accesso alle collezioni di dati, che non sono altro che i metodi esportati dalla classe *Model*. Per quanto riguarda l'invio dei messaggi ai singoli oggetti (una volta ottenuti dal *Model*) non è stato posto alcun vincolo particolare.

- *getInstace()*: per ottenere l'oggetto univoco *Model*
- *getAutenticabili()*: ottieni la lista di aventi credenziali
- *getFarmaci()*
- *getReazioniAvverse()*
- *getPazienti()*
- *getSegnalazioni()*
- *aggiornaTerapie()*: aggiungi la terapia voluta al paziente
- *getAvvisi()*
- *generaAvviso()*: invia/costruisce un nuovo avviso
- *generateIfWasWeekend()*: genera un avviso *weekend* se è il caso
- *getUnread()*: gli avvisi non letti
- *getFarmacoByName()*: ottieni il farmaco relativo al nome passato
- *createPaziente()*: costruisce un nuovo paziente
- *createReazione()*: costruisce una nuova reazione avversa
- *makeSegnalazione()*: invia/costruisce una nuova segnalazione
- *load()*: carica i dati dal disco
- *save()*: salva i dati su disco

Implementazione e design pattern usati

Generalità

In questa sezione si discute l'implementazione e l'applicazione dei vari pattern.

- Si sorvola sui vari **pattern iterator** ed affini, poiché sono alla base della programmazione in Java.

- Collegandoci brevemente al discorso sull'architettura, si è scelto di utilizzare il **pattern observer** per la comunicazione vista-controllore, come è nativamente suggerito dall'implementazione stessa di *Swing*. Per ulteriori dettagli si rimanda alla documentazione di tali librerie.

- In più punti del codice, ovvero a partire da *JFrame* diversi, si è utilizzato *TerapiaFrame* (con annesso *TerapiaListener*) per raccogliere informazioni riguardo a nuove terapie farmacologiche. Queste informazioni sono state poi passate al *JFrame* chiamante utilizzando un metodo comune (ogni *JFrame* chiamante implementa l'interfaccia *FrameConTerapie*). Tuttavia, ogni *JFrame* tratta in modo diverso la terapia costruita. Per gestire questa variabilità di risposte per lo stesso metodo chiamato, si è utilizzato il **pattern strategy**: il contesto (*TerapiaFrame*) chiamerà un metodo comune (*addTerapia()*) che a seconda della strategia (*FrameConTerapia*) sarà implementato in maniera differente.

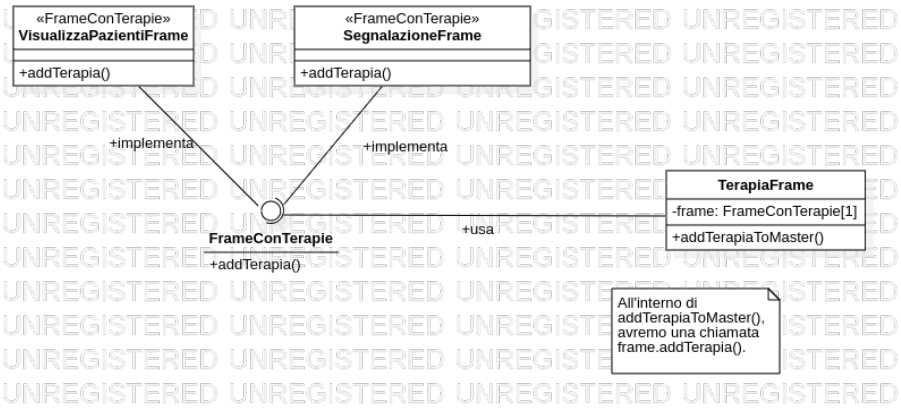

Figure 12: Pattern strategy

Classe *Model*

Dettagli più importanti sono quelli che riguardano *Model*, la classe che gestisce e manipola l'intera mole di dati.
Essa ricopre un ruolo fondamentale nel funzionamento del software. In sostanza, è il punto usato dal controllore e dalla vista per accedere alle informazioni: è colei che immagazzina e fornisce i dati, ed è anche colei attraverso il quale vengono creati gli oggetti principali del modello.
Considerato il fatto che *Model* si troverà a gestire le collezioni di dati, si è visto opportuno permettere la creazione di una singola istanza di questa classe. Per realizzare questo comportamento, si è utilizzato il **pattern singleton**. Per quanto riguarda il ruolo di *Model* come creatore di oggetti, ci si è ispirati fortemente alla filosofia offerta dal **pattern factory**. Si veda l'*appendice* per ultieriori dettagli.
In realtà, oltre alla semplice creazione, tutti i metodi "critici" degli oggetti del modello sono stati messi sotto la responsabilità di *Model*. Infatti, ogni metodo potenzialmente dannoso (in sostanza, quelli che vanno a modificare lo stato del modello) è stato definito con protezione **package** e incapsulato in un metodo più ad alto livello esportato da *Model*. Così facendo, tutte le modifiche critiche del modello passano sotto lo sguardo protettore di questa classe. Il motivo per cui non è stato adottato direttamente il pattern **facade** è il seguente: la complessità del codice è fortemente diminuita se si permette alla vista ed al modello di accedere direttamente alla struttura delle classi del modello. È stato considerato quindi sufficiente incapsulare solamente alcuni metodi, come spiegato sopra.
Per quanto invece riguarda l'immagazzinamento dei dati, l'idea ricalca fortemente l'utilizzo del pattern architetturale **repository**. Seppur non sia stata inserita esplicitamente un'architettura di questo tipo, si è adottata l'idea di fondo, impiegando un sistema centrale di gestione delle moli di dati.
La classe *Model* infatti immagazzina varie collezioni di dati (costituite dagli oggetti principali del dominio d'interesse) ed esporta i metodi per accedervi. In questo senso possiamo vedere *Model* come il punto di immagazinamento ed accesso dei dati da parte della Vista e del Controllore. Si osservi che, notevolmente, *Model* non concede di aggiungere alle proprie collezioni nuovi oggetti esterni. L'unico meccanismo fornito è quello di creazione esplicita di un nuovo oggetto (e conseguente aggiunta), per evitare che oggetti mal-formati vengano aggiunti alla collezione, o che vengano aggiunti duplicati al sistema.
Per qualche piccolo dettaglio visivo riguardo il ruolo di questa classe, si rimanda al *diagramma delle classi* visto in precedenza *(cfr. Progettazione e pattern architetturali usati).*

Diagrammi di sequenza del software implementato

Seguono diagrammi di sequenza che mostrano le dinamiche di alcune interazioni tra classi particolarmente complesse od interessanti. *Sono stati rappresentati soltanto gli scambi salienti.*

Per ulteriori dettagli riguardo al codice, l'implementazione ed il suo funzionamento, si veda il **JavaDoc** nella directory del progetto.

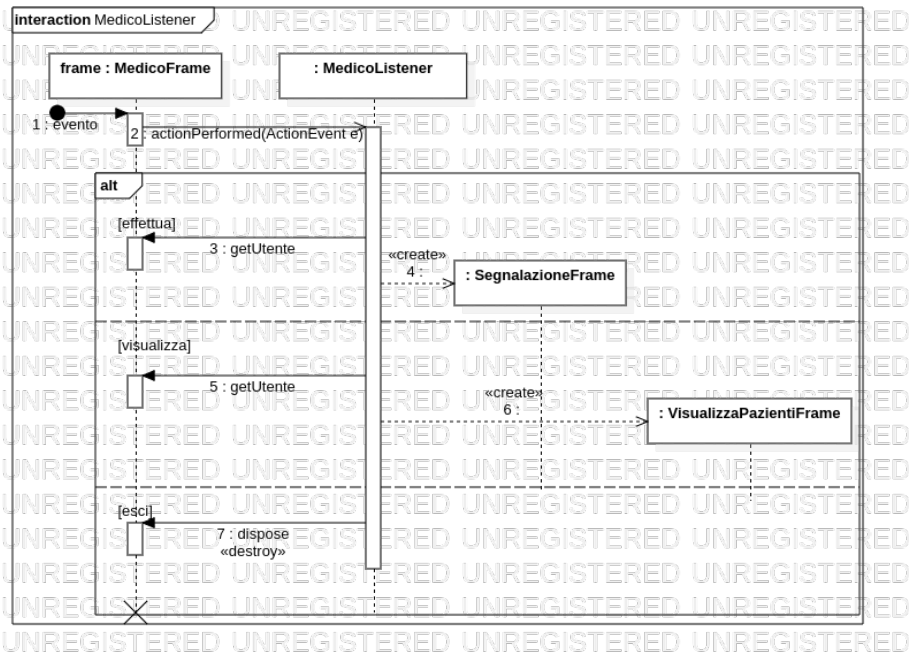

Figure 13: MedicoListener

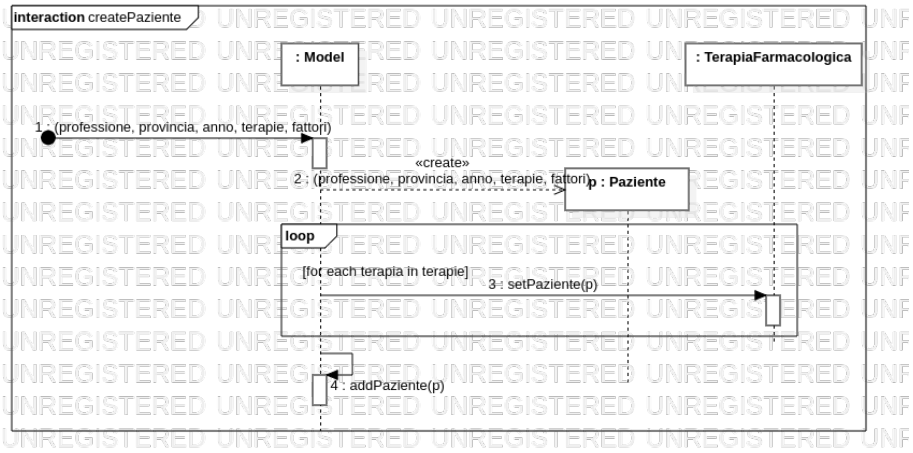

Figure 14: createPaziente

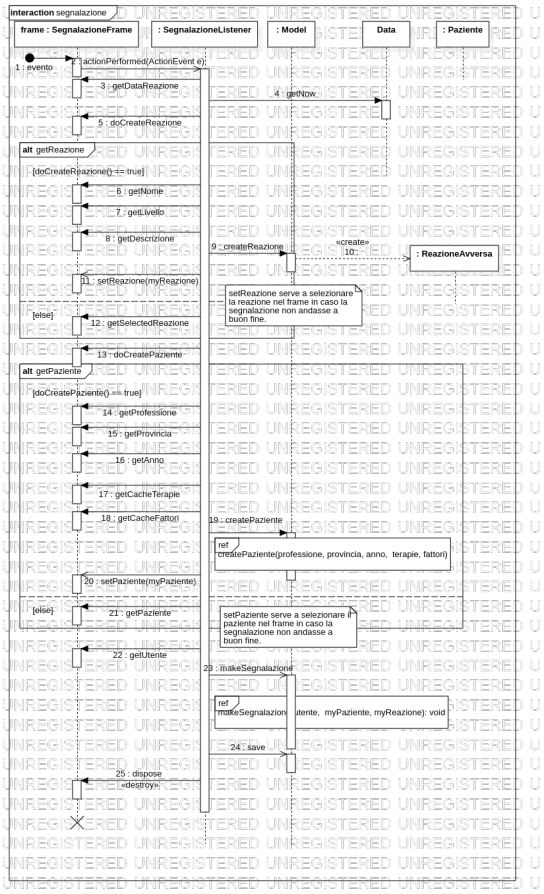

Figure 15: Effettuazione di una segnalazione

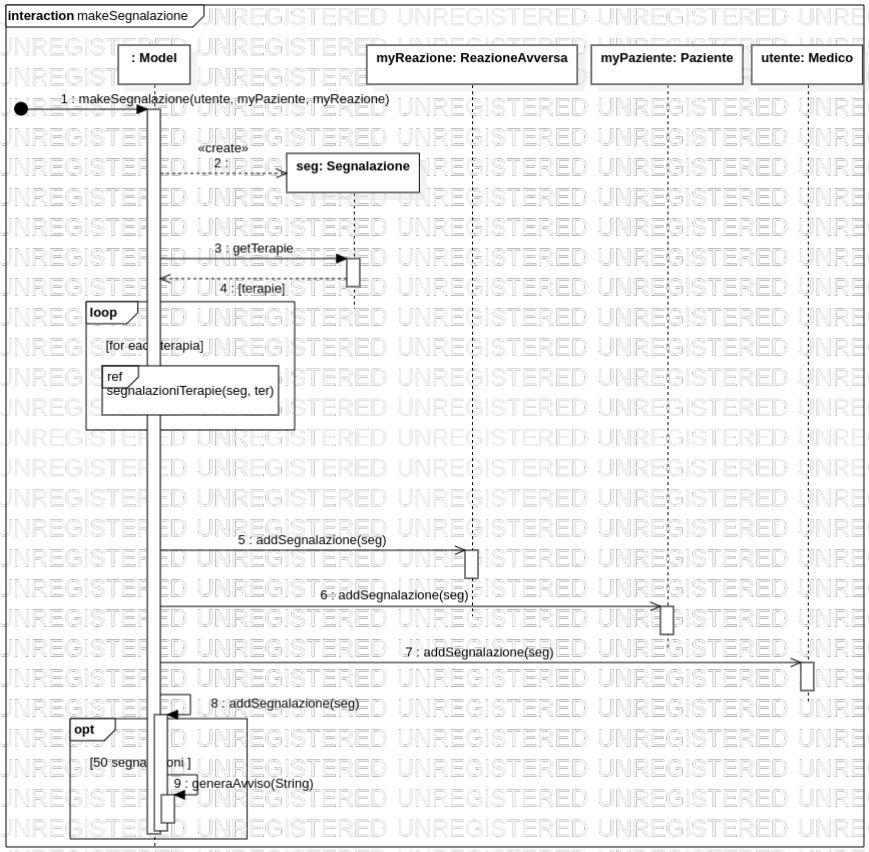

Figure 16: makeSegnalazione

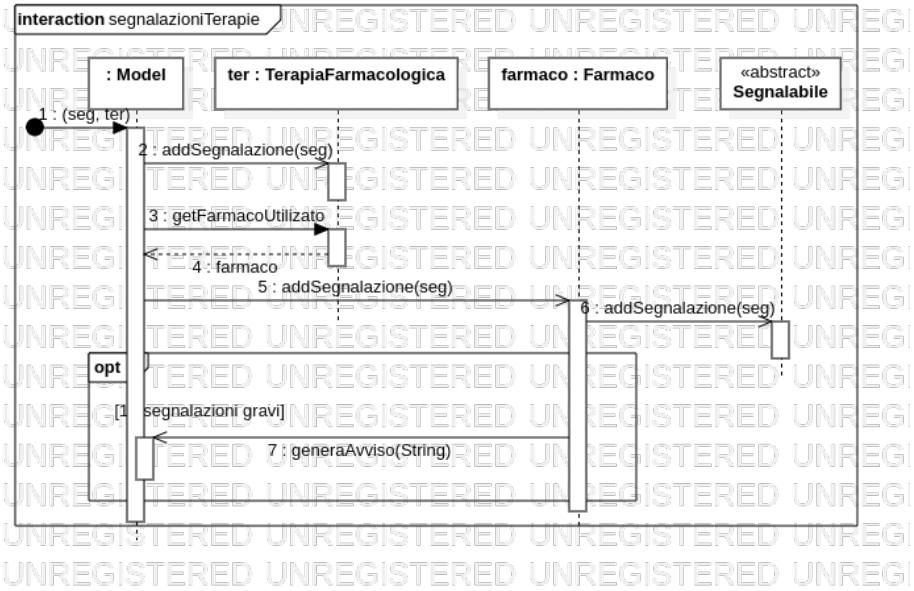

Figure 17: segnalazioniTerapie

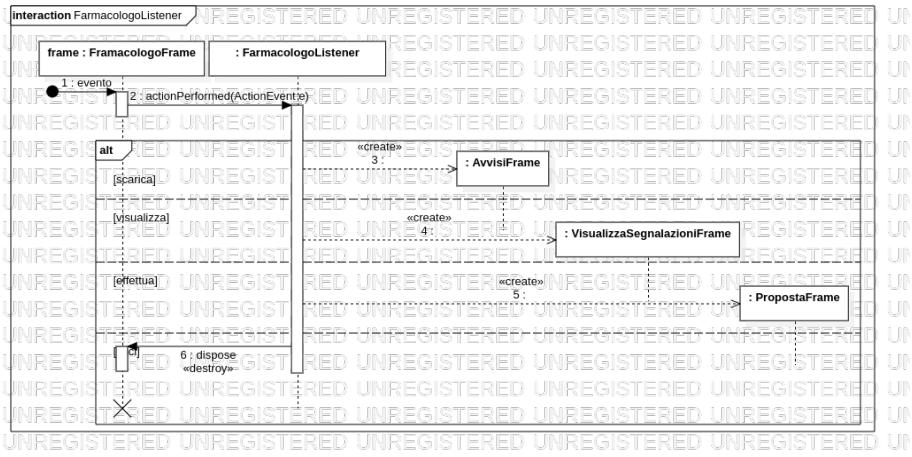

Figure 18: FarmacologoListener

Attività di test e validazione

Per verificare la solidità del software prodotto, si sono svolte le seguenti attività:

1. Ricognizione del documento delle specifiche e confronto con i diagrammi prodotti
2. Verifica della consistenza tra diagrammi e codice prodotto
3. Ispezione del codice, verifica della correttezza dei pattern, ricerca di malepratiche varie
4. Unit test automatizzato mediante JUnit per verificare la correttezza di alcune delle classi più problematiche
5. Test degli sviluppatori sul software
6. Test utente generico sul software

Ispezione codice e documentazione

In questa fase si è semplicemente rivisto il documento delle specifiche e lo si è confrontato con i diagrammi UML prodotti, per verificare la correttezza degli use case, activity diagrams e diagramma delle classi. Una volta finita questa attività si è confrontato il codice (staticamente) ai diagrammi UML, per verificarne la consistenza. Infine si è data una nuova ispezione del codice per cercare infrazioni, malepratiche e cattivi usi dei pattern. Particolare attenzione è stata data al non aver creato autonomamente oggetti delegati alla classe *Model* (factory pattern).

Unit test

In questa fase è stata creata una versione alternativa del codice ai fini del test automatizzato con JUnit. Si sono testate alcune delle classi più problematiche e alcuni delle funzionalità più complesse.

```
import java.util.HashMap;


import Modello.Model;
import Modello.Segnalazione;

public class TestFarmaco extends Segnalabile implements java.io.Serializable{
/**
 *
 */
private static final long serialVersionUID = 1L;
private String name;
boolean ritirato = false, controllato = false, monitorato = false;
private HashMap<Integer, Integer> worryingCasesPerYear = new HashMap<>();
```

```
public TestFarmaco(String name) {
this.name = name;
}

public String getName() {
return name;
}

public void setName(String name) {
this.name = name;
}

public void setRitirato() {
ritirato = true;
}

public void setControllato() {
controllato = true;
}

public void setMonitorato() {
monitorato = true;
}

public int hashCode() {
return name.hashCode();
}

public String toString() {
return name + getStatusString();
}

public String getStatusString() {
if(ritirato)
if(monitorato)
if(controllato)
return " [R, M, C]";
else
return " [R, M]";
else
if(controllato)
return " [R, C]";
else
return " [R]";
else
```

```
if(monitorato)
if(controllato)
return " [M, C]";
else
return " [M]";
else
if(controllato)
return " [C]";
else
return "";
}

public void addSegnalazione(Segnalazione seg) {
super.addSegnalazione(seg);
if(seg.getReazioneAvversa().getLivelloGravita() > 3) {
worryingCasesPerYear.putIfAbsent(seg.getDataReazioneAvversa().getYear(), 0); // se era null
worryingCasesPerYear.put(seg.getDataReazioneAvversa().getYear(), //aumento il counter di 1
worryingCasesPerYear.get(seg.getDataReazioneAvversa().getYear()) + 1
);

if( worryingCasesPerYear.get(seg.getDataReazioneAvversa().getYear()) % 10 == 0 &&
worryingCasesPerYear.get(seg.getDataReazioneAvversa().getYear()) > 0) //
Model.getInstance().generaAvviso("Il TestFarmaco " + this + " ha ricevuto " + worryingCasesP
" casi gravi nel " + seg.getDataReazioneAvversa().getYear() + ".");
}
}

public boolean equals(Object o) {
return o instanceof TestFarmaco && ((TestFarmaco)o).getName().equalsIgnoreCase(name);
}
}
```

Test degli sviluppatori

In questa fase lo sviluppatore ha immesso nel sistema degli input (sia corretti sia errati) per vedere se la reazione del software fosse quella attesa. Alcuni dei test svolti sono:

- **Verifica del corretto funzionamento dell'autenticazione**: i dati errati vengono respinti, ed a fronte dei dati corretti all'utente viene mostrata la schermata iniziale legata alla sua professione.
- **Farmacologo a database vuoto**: si è provato a pigiare ogni pulsante pigiabile in assenza assoluta di dati, per vedere se questo creasse eccezioni od errori.
- **Medico a database vuoto**: si è provato a pigiare ogni pulsante pigiabile

in assenza assoluta di dati, per vedere se questo creasse eccezioni od errori.
- **Inserimento di una segnalazione**. Verifica che tale segnalazione sia poi visibile dai farmacologi e dal medico, ma non da altri medici.
- Verificato che **il medico veda solo i *suoi* pazienti** ma ***tutte* le reazioni in sistema**
- **Verificato il sistema di avvisi** (settimanale, soglia dei 50, soglia dei 10)
- Verifica di vari inserimenti di **input errati, input vuoti, stringhe malformate, numeri negativi**
- Verifica della reazione del software all'inserimento di terapie finite prima di iniziare, o di segnalazioni senza alcuna terapia, terapie prima della nascita di un paziente, o di reazioni non ancora avvenute **(controllo sulle date)**
- Inserimento di reazioni, farmaci o segnalazioni **duplicate**
- Controllato che il meccanismo di segnalazione **inferisca i farmaci corretti** se vi sono più terapie a carico del paziente (ma solo alcune attive durante la reazione avversa)

Test utente generico

Come ultimo scaglione il software è stato sottoposto ad un test da parte di alcuni individui con limitata dimestichezza informatica ed assoluto distacco dallo sviluppo. In questa fase non si è cercato in nessun modo di guidare o strutturare l'esperienza, per non influenzare in alcun modo il risultato; piuttosto si è lasciato che il soggetto navigasse liberamente il sistema. Non è stata data nessuna spiegazione sull'utilizzo del software, se non una generale indicazione dei suoi fini; ci si è limitati a rispondere alle domande, quando sollevate. L'unico scopo del test era quello di rilevare errori invisibili allo sviluppatore; in realtà, gli utenti a cui è stato mostrato il software hanno anche aiutato ad individuare nuove funzionalità per migliorare l'usabilità generale del sistema.

Appendice: creazione di oggetti mediante la classe Model

La classe *Model* riprende la filosofia alla base del **factory pattern**: infatti, funge da punto di creazione per varie classi delicate.
Nonostante sia stato tralasciato l'utilizzo di un'interfaccia comune, si è adottata l'idea di delegare la creazione degli oggetti ad un unico responsabile. La classe *Model*, infatti, crea oggetti di varie classi, come *Paziente*, *ReazioneAvversa*, *Segnalazione*; questo per il semplice motivo che la creazione di questi oggetti è molto delicata, ed è quindi meglio centralizzarla e delegarla ad un unico ente.
La scelta di non usare l'interfaccia comune deriva dal fatto che, sostanzialmente, gli oggetti creati non sono logicamente assimilabili in unica interfaccia e vengono manipolati separatamente dal resto del software. Utilizzare un'interfaccia comune avrebbe aggiunto gradi di complicazione inutili. Chiaramente, potrebbe sembrare che il lasciarsi indietro l'utilizzo dell'interfaccia comune sconfigga il senso stesso dell'ultizzo di questo pattern: a questo punto, in un qualsiasi punto del codice si potrebbe creare una classe tra quelle sopracitate in malo-modo.
La cosa è stata gestita in tre modi: innanzitutto, si è reso i costruttori degli oggetti interessati **package-protected**. Inoltre, si richiede come parametro del costruttore l'oggetto creante (**this**): se non è istanza di *Model*, viene generata un'**eccezione *FactoryException***. Un ulteriore meccanismo di sicurezza in tal senso è fornito dal fatto che non è possibile aggiungere dall'esterno oggetti allo stato di *Model*.
Di seguito la lista di oggetti sottoposti alla creazione tramite Model.

- *Autenticabile* e sottoclassi
- *Paziente*
- *Segnalazione*
- *Farmaco*
- *Avviso*
- *ReazioneAvversa*