UNIVERSITÀ DEGLI STUDI DI MESSINA

FACOLTÀ DI INGEGNERIA

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

Tesina di Ingegneria del Software

LABANALISI

Tesina di:
Giuseppe Gerace

Docente:
Prof. Antonio Puliafito

ANNO ACCADEMICO 2010 – 2011

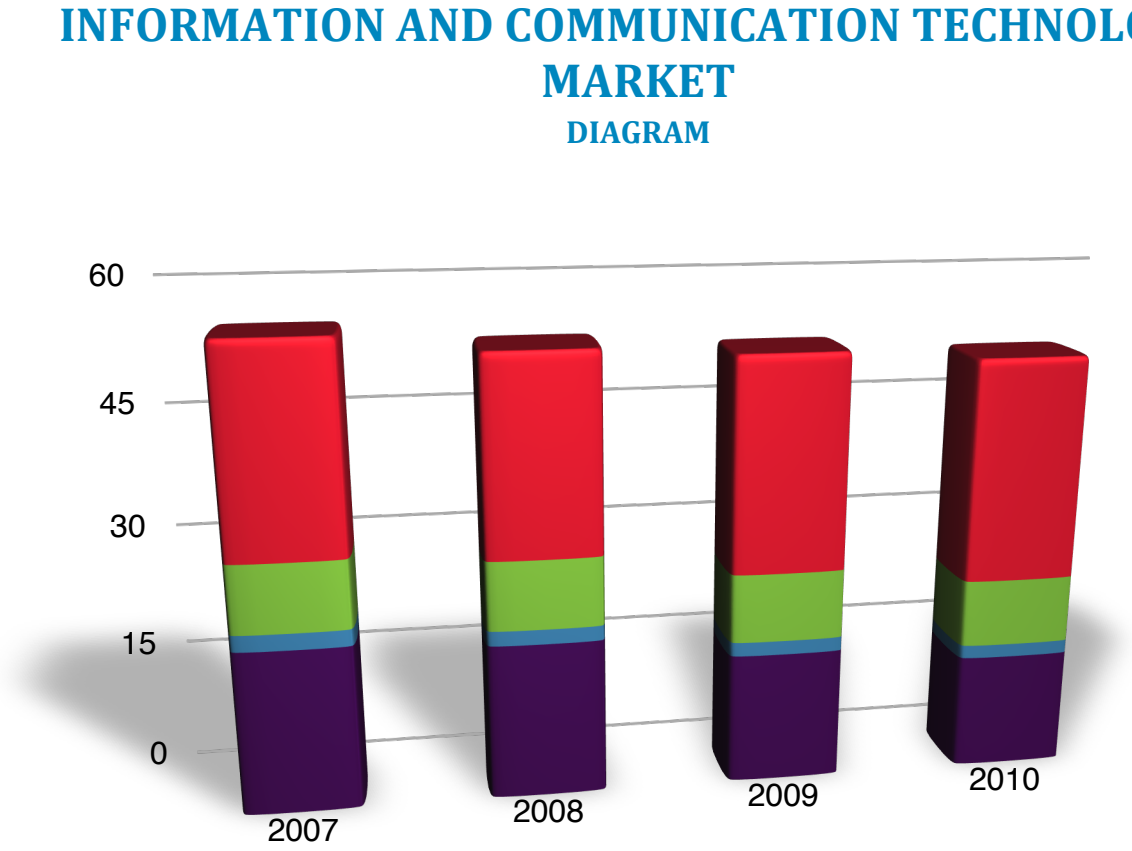
INDICE

Capitolo 1

Introduzione all'ingegneria del software

1.1 L'importanza del software

Oggi il software rappresenta la tecnologia più importante a livello mondiale. Negli anni '50, nessuno avrebbe potuto prevedere che il software sarebbe diventato una tecnologia indispensabile per il commercio, la scienza e l'ingegneria; che il software sarebbe stato il "trigger" che avrebbe dato origine alla rivoluzione dei personal computer. Qualunque economia di un paese sviluppato ha una dipendenza fortissima dal software. Grazie ad esso, è possibile controllare una infinità numerabile di sistemi e, conseguentemente, offrire dei servizi all'utente finale. Ciò giustifica gli investimenti fatti sul software.



Esiste una componente servizi che è significativa, una componente software, una di assistenza e una di hardware. Si può notare come l'hardware, rispetto a software e servizi, è una componente limitata. Il valore aggiunto risiede nella parte software.

In generale, con il termine software si intende una serie di programmi, dati, procedure, regole per l'utilizzo e relativa documentazione. Sviluppare un software non vuol dire, quindi, sviluppare un programma; un "software" è un sistema complesso costituito da tanti programmi che collaborano tra di loro. Oggigiorno, nessun sistema complesso viene scritto da zero; esistono dei "componenti", open source e non, già pronti al "riuso". La parola chiave nell'Ingegneria del Software è "riutilizzo del codice". Esempi di componenti sono i database, i componenti che fanno reporting, quelli che permettono di avere interfaccia mobile e interfaccia web standard, etc. La sfida consiste, quindi, non tanto nello scrivere un codice, ma piuttosto nel sapere scegliere adeguatamente i componenti che andranno a caratterizzare un sistema di base. In generale, scrivere un sistema software composto da più componenti è più difficile che scrivere un semplice programma: a parte la complessità di più componenti, deve esserci la base di verifica. Infatti, programmare vuol dire produrre un codice efficace e al tempo stesso efficiente; bisogna inoltre tenere in considerazione i problemi di comunicazione tra membri di un team che lavorano ad un determinato progetto. Esistono svariate tipologie di software:

- stand alone (word processor, browser)
- embedded (ABS, smartphone)
- process support (product procession, business procession)
- safety critical (aerospace, military, medical)
- mission critical (banking, logistics)
- other (games)

Con il termine stand alone ci si riferisce ad un'unica applicazione che funziona da sola. Esiste una quantità enorme di codice definito "embedded", annidato all'interno di elettronica di consumo. Un'altra tipologia di software è quella definita "safety critical", ovvero software che ha ricadute in termini di sicurezza, al contrario del software "mission critical" che ha, invece, conseguenze di tipo economico.

Queste varie tipologie di software sono diffuse dappertutto: nell'elettronica di consumo, a livello industriale, nei pc desktop e in tutti i servizi commerciali.

In generale, quindi, abbiamo dei grossi vantaggi in termini di funzionalità e di flessibilità di comportamento; basti pensare ad uno smartphone touchscreen, il cui software ha permesso di riconfigurare la classifica interfaccia utente fisica. All'aumentare delle funzionalità, aumenta la complessità del codice e conseguentemente la possibilità di bug: questo è l'aspetto negativo. Il grosso problema odierno riguarda la scrittura di codice robusto, scalabile e riutilizzabile. Esempi di progetti celebri, falliti a causa di errori di programmazione sono:

- Ariane 5 (fallimento del primo lancio del vettore commerciale ESA)
- Patriot (una caserma colpita per un difetto nel sistema di guida)
- Therac 25 (il software ha provocato la perdita di vite umane)

- Mars Climate Orbiter & Mars Polar Lander (difetti nel software hanno causato il fallimento delle missioni).

Per poter garantire la sicurezza del sistema ed evitare scenari come quelli sopra descritti, il software tende ad avere dei costi altissimi. Buona parte dei costi del software sono dovuti alla rassicurazione di qualità e alla manutenzione. Solitamente un software utile viene utilizzato per tanti anni e deve essere, quindi, in grado di adattarsi alle esigenze che cambiano (es.: software gestione sistemi bancari).

Un elemento fondamentale del codice è la qualità. Bisogna cercare di produrre codice robusto, efficace ed efficiente, prima che questo venga introdotto sul mercato. Ad esempio, correggere un software che non rispetta i requisiti iniziali del cliente comporta dei costi esponenziali dovuti al ritiro del software in questione, alla rimodulazione dei requisiti iniziali, alla riscrittura del codice e alla fase finale di test.

Risulta, quindi, di fondamentale importanza l'analisi dei requisiti, che si suddividono in due categorie: requisiti funzionali e non funzionali. I requisiti funzionali indicano la funzione svolta da un determinato sistema (es.: sommare due numeri interi); i requisiti non funzionali specificano le altre proprietà che deve soddisfare un sistema software (es.: usabilità, affidabilità, precisione del risultato). Si capisce come la verifica dei requisiti non funzionali non sia immediata.

Di seguito, vengono illustrate le caratteristiche tra uno sviluppo ingegneristico del software e quella che è la programmazione di un singolo programmatore:

Software Engineering	Solo programming
software grandi dimensioni	software piccole dimensioni
team / teams	singola persona
requisiti definiti dal cliente all'atto del contratto	requisiti definiti dal programmatore
molti componenti	pochi componenti
molti cambiamenti, vita lunga	pochi cambiamenti, vita corta
costi elevati	costi ridotti

Per essere un valido ingegnere del software non basta, quindi, solo saper programmare; mentre un programmatore sviluppa un programma da solo lavorando su specifiche sconosciute, un ingegnere del software è un elemento integrante di un team, il quale, dopo aver identificato i requisiti e sviluppato le specifiche, realizza componenti che saranno combinati con altri, sviluppati e manutenuti da altri.

1.2 Breve storia dell'ingegneria del software

Il termine ingegneria del software risale agli anni '60, quando programmi più complessi e commerciali iniziano ad essere sviluppati da parte di team di esperti.

Si assiste, quindi, a una trasformazione del software da prodotto artigianale a prodotto industriale. Un ingegnere del software deve essere un buon programmatore, un esperto di algoritmi e strutture dati con una buona conoscenza di uno o più linguaggi di programmazione. Immaginando la produzione di software come un processo industriale che coinvolge diversi gruppi di persone, un ingegnere del software deve anche conoscere diverse modalità di progettazione, essere capace di tradurre requisiti generici in specifiche ben precise e in grado di comunicare con l'utente finale in un linguaggio a quest'ultimo comprensibile.

L'ingegneria del software è, comunque, una disciplina ancora in via di sviluppo; non esistono tuttora degli standard per le specifiche di progetti di software.

A differenza dell'ingegneria tradizionale, che si basa su teorie matematiche e metodi consolidati ed in cui si seguono determinati standard di progettazione, l'ingegneria del software punta maggiormente sull'esperienza piuttosto che su strumenti matematici non ancora ben sviluppati.

La conoscenza del dominio, cioè dell'ambiente esterno in cui il sistema software dovrà funzionare, è fondamentale da parte dell'ingegnere del software: per progettare un sistema di controllo di navigazione, un ingegnere dovrà capire come funziona una nave! Software basati su una conoscenza errata del dominio possono causare veri e propri disastri.

Di seguito, viene illustrata una breve panoramica delle prime tre decadi della storia del software engineering:

1950s:

- i computer iniziano ad essere utilizzati in maniera estensiva per applicazioni di tipo business

1960s:

- viene immesso sul mercato il primo prodotto software
- IBM annuncia il suo "unbundling" nel Giugno 1969

1970s:

- i prodotti software iniziano ad essere acquistati dagli "utenti normali"
- l'industria del software cresce rapidamente nonostante la mancanza di finanziamenti
- nascono le prime "software house"

1.3 Peculiarità di un sistema software

Il termine sistema è utilizzato in svariati contesti; l'idea di fondo è che un sistema è qualcosa di più semplice dell'insieme dei componenti che lo caratterizzano. Volendo dare una definizione funzionale, potremmo dire che un sistema è una collezione significativa di componenti interrelati che lavorano assieme per realizzare un determinato obiettivo. I sistemi che ci interessano sono quelli tecnico-informatici, che comprendono componenti hardware e software, ma non procedure e processi.

Alcune proprietà riguardano un sistema nella sua interezza: queste proprietà vengono definite **proprietà complessive**. Altre possono essere derivate direttamente da simili proprietà dei sottosistemi. Come nel caso dei requisiti, anche in questo caso distinguiamo tra *proprietà complessive funzionali*, che appaiono quando tutte le parti di un sistema lavorano assieme per raggiungere un obiettivo, e *proprietà complessive non funzionali*, che si riferiscono al comportamento del sistema nel suo ambiente operativo. Esempi di proprietà complessive sono la sicurezza, l'affidabilità, la protezione, la robustezza, la produttività e l'usabilità.

L'affidabilità è una delle caratteristiche più importanti di un prodotto software: un prodotto affidabile non dovrebbe arrecare danni fisici o economici in caso di guasto del sistema stesso; l'affidabilità riguarda, quindi, esclusivamente la probabilità che si verifichino. L'affidabilità è un requisito essenziale nei sistemi critici, in cui un malfunzionamento del sistema può arrecare grave danno a persone o cose. Con la definizione **sistemi critici** ci riferiamo a sistemi tecnici o socio-tecnici da cui dipendono persone o aziende. La mancata fornitura di servizi da parte di questi ultimi può comportare seri problemi e importanti perdite. Normalmente se per un prodotto si richiede un'alta affidabilità, avrà un costo di sviluppo maggiore.

Non è semplice stabilire se un sistema software è affidabile; si tratta quindi di sviluppare alcuni criteri in base ai quali giudicare il funzionamento del sistema. In genere, non si potranno esaminare tutti i possibili comportamenti di un sistema, ma è importante identificare quelli che sono ritenuti vitali per un suo funzionamento affidabile.

L'affidabilità dipende:

- dalla correttezza delle fasi iniziali di sviluppo del prodotto software; se queste fasi sono svolte correttamente sarà più probabile avere un prodotto affidabile;

- dal passaggio tra specifica e implementazione, che deve essere fatto in modo da preservare la correttezza;

- ogni componente che partecipa alla realizzazione del sistema deve essere affidabile.

Il software è robusto quando si comporta in maniera accettabile anche in corrispondenza di situazioni non specificate nei requisiti: procede nel suo lavoro in maniera accettabile anche in presenza di Hw-failures, failures del sistema operativo, input non previsti, failures del software di interfaccia.

Si definisce, quindi, robusto un prodotto che, anche se usato in condizioni di "stress", si comporta in modo corretto. La robustezza è una caratteristica che, per essere soddisfatta, implica accortezze particolari durante la costruzione del prodotto. Questo significa inevitabilmente un costo di realizzazione maggiore. Il software è sicuro quando protegge l'accesso ad informazioni private e protette, impedendo accessi non autorizzati, sia di natura involontaria, che di natura dolosa. È una categoria particolare di software robusto.

La produttività valuta le prestazioni inerenti al processo di produzione del software. È una qualità molto difficile da misurare perché n! di linee di codice (unica caratteristica di cui si dà una unità di misura) su unità di tempo.

Di tutte queste caratteristiche, alcune sono qualità interne al prodotto, cioè sono in relazione con lo sviluppatore e solo questi ne può usufruire (ad es.: la riusabilità); altre sono qualità esterne, cioè sono in relazione stretta con l'utilizzatore (es.: usabilità); alcune caratteristiche, infine, sono di entrambe i tipi.

Dalle caratteristiche appena analizzate, si possono ottenere delle linee guida e dei vincoli per lo sviluppo di un prodotto software. Saranno rispettati soltanto alcuni di questi vincoli, poiché in generale non è possibile o economicamente vantaggioso soddisfare tutte le caratteristiche esposte.

Un prodotto software è usabile se un utente non esperto riesce a capire come utilizzarlo. È una caratteristica più "esterna" della comprensibilità, perché è legata all'interfaccia utente, a chi deve usare il sistema e a quante conoscenze egli ha del sistema stesso, ed esprime il grado di interfacciabilità con l'utente (es: mouse, icone, help-on-line, ecc.).

Capitolo 2

Il prodotto software: modelli e requisiti

2.1 Il processo software

Un processo software è un insieme di passi che devono essere seguiti nella realizzazione di un prodotto software. È uno degli strati fondamentali dell'ingegneria del software in quanto la qualità del prodotto finale dipende molto dall'adeguatezza del processo.

Figura 2.1: Caratterizzazione di un processo software

Si può, quindi, ritenere un processo software come l'insieme delle attività necessarie per realizzare software di alta qualità.

Il processo di realizzazione del software consente uno sviluppo organico entro i tempi previsti dai requisiti. I metodi comprendono un'ampia gamma di attività: analisi dei requisiti, progettazione, testing, assistenza post-vendita.

Gli strumenti possono includere notazioni come UML, che consente la descrizione del dominio applicativo, o possono includere ambienti CASE (*Computer Aided Software Engineering*) specifici come Rational Rose.

Indipendentemente dalle dimensioni e dalla complessità di un progetto software, esistono una serie di attività strutturali di base che sono applicabili a tutti i progetti software e che possono essere personalizzate in base alle specifiche esigenze. Le principali sono: comunicazione, pianificazione, modellazione, costruzione e deployment.

L'attività di comunicazione prevede la raccolta e l'analisi dei requisiti che derivano da una serie di "confronti" con il committente di un progetto.

Nell'attività di pianificazione si determinano le risorse necessarie, i probabili rischi, le operazioni da svolgere e i prodotti da realizzare.

L'attività di costruzione prevede la generazione del codice e della relativa documentazione annessa.

Alla fine del processo di realizzazione del software, quest'ultimo verrà consegnato al cliente che potrà, quindi, valutare il prodotto e fornire utili indicazione allo sviluppatore per eventuali miglioramenti.

Queste attività, che caratterizzano la struttura di un processo, possono essere personalizzate prendendo in considerazione le esigenze specifiche del progetto stesso e le caratteristiche del team di sviluppo.

La struttura precedentemente descritta è affiancata da una serie di attività parallele che prevedono, ad esempio, la gestione dei rischi, la valutazione della qualità del software, la misura, la gestione della configurazione software.

Per poter soddisfare al meglio le richieste di progetto, esistono alcuni modelli che facilitano la comprensione dei requisiti software tra cliente e sviluppatore.

Ogni modello di processo software, che ne è una rappresentazione "ad alto livello", rappresenta un processo da un particolare punto di vista. Questi modelli generici possono essere utilizzati come strutture base per dar vita a processi di ingegneria del software più specifici.

Il ciclo di vita ingloba tutte le attività necessarie per sviluppare un sistema nella sua interezza: al suo interno vi è, dunque, il ciclo di sviluppo che prevede attività, passi e fasi per realizzare un sistema.

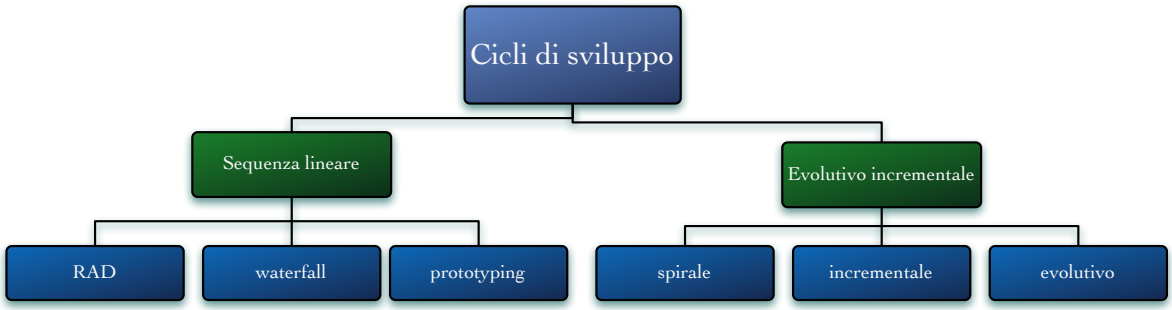


Figura 2.2: Ciclo di sviluppo di un sistema

Tra i vari modelli di processo esistenti, il più famoso è quello a cascata.

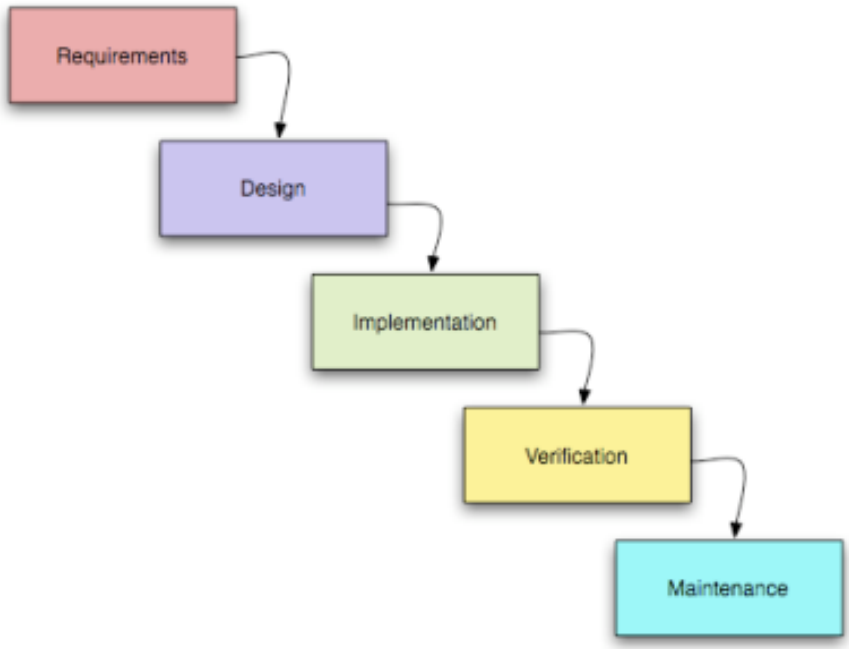


Figura 2.3: Modello a cascata

Esso descrive la sequenza di passi che caratterizza il processo di produzione del software. Viene chiamato a cascata in quanto ogni fase genera un output che rappresenta l'input della fase successiva. Lo schema soprastante rappresenta un tipico modello a cascata.

Generalmente la fase di raccolta dei requisiti è preceduta dallo *studio di fattibilità*. In questa fase il progettista interagisce con il committente, al fine di capire gli obiettivi di quest'ultimo e può così formalizzare il problema in un linguaggio rigoroso: è indispensabile stabilire tutti i probabili scenari di utilizzo del software come anche analizzare i costi di realizzazione.

La fase successiva prevede la raccolta e l'analisi dei requisiti. Attraverso la discussione con i clienti, in questa fase vengono stabiliti i servizi che dovrà avere il sistema. È importante affrontare questa fase con eccessiva scrupolosità per evitare di avere ripercussioni sulle fasi successive del processo e quindi sul sistema in toto.

Il termine requisito viene usato con una doppia accezione. Ci si può riferire sia ad una formulazione astratta di un servizio che un sistema dovrebbe fornire, che ad una definizione dettagliata di una funzionalità del sistema. Quando parliamo di requisiti utente ci riferiamo al primo caso; quando parliamo di requisiti di sistema al secondo.

I requisiti utente vengono formulati in linguaggio naturale e molto spesso sono corredati da diagrammi; sono inoltre definiti i vincoli sotto cui deve operare il sistema.

I requisiti di sistema definiscono le stesse cose dei requisiti utente, ma con un livello di dettaglio maggiore.

I requisiti vengono raggruppati in quattro categorie fondamentali: *funzionali, non funzionali, tecnologici e inversi.*

I requisiti funzionali descrivono le funzioni che dovrà svolgere il sistema. Si parte da una descrizione molto generale dei requisiti che man mano vengono "raffinati" per rispecchiare le attività che l'utente del sistema può realmente svolgere.

I requisiti non funzionali riguardano le caratteristiche generali che il sistema dovrà possedere, come ad esempio prestazioni e affidabilità.

I requisiti tecnologici stabiliscono le tecnologie che verranno utilizzate per realizzare il progetto.

I requisiti inversi riguardano la sicurezza: specificano operazioni che il sistema non dovrà mai permettere di effettuare.

Nella fase successiva, ovvero quella riguardante la progettazione, i requisiti vengono suddivisi tra sistema hardware e sistema software; viene inoltre fissata l'architettura generale del sistema. È consigliato suddividere il progetto in moduli di dimensioni minori, sottolineando e analizzando le relazioni fra di essi. Al termine di questa fase viene prodotta la documentazione contenente tutte le informazioni necessarie all'esecuzione del progetto; per questo scopo, generalmente si ricorre ad un linguaggio di progettazione come l'UML.

Si passa quindi alla fase di implementazione, in cui i vari moduli dell'architettura diventano un insieme di programmi scritti in un opportuno linguaggio di programmazione. Questa fase comprende anche il test delle unità, che serve a verificare che ognuna di queste rispetti le specifiche. L'attività di test rappresenta una fase estremamente critica.

I moduli scritti e testati nella fase di implementazione vengono integrati nella fase di *integrazione:* i singoli programmi vengono integrati e testati come un sistema completo per verificare che i requisiti del software vengano soddisfatti. Seguendo la filosofia del modello a cascata, alla fine di questa fase, il progetto dovrà essere conforme a tutti i requisiti iniziali.

La fase finale è quella di manutenzione. Questa fase prevede l'installazione e la messa in funzione del sistema, la correzione degli errori non scoperti nelle prime fasi del ciclo vitale e

l'incrementazione dei servizi del sistema nel caso in cui nascano nuove esigenze. Le ricerche dimostrano che questa è la fase più lunga ed economicamente più onerosa.

Ogni fase del modello a cascata produce la documentazione relativa. Il più grande problema di questo modello è rappresentato dalla rigida suddivisione del progetto in fasi distinte, approccio che contrasta con eventuali futuri cambiamenti dei requisiti da parte del cliente.

Il modello a cascata segue il principio della *consegna incrementale*, ovvero le specifiche, la progettazione e l'implementazione del software sono suddivisi in una successione di incrementi sviluppati uno per volta.

Un modello di ciclo di vita alternativo a quello a cascata è il *modello a spirale.*

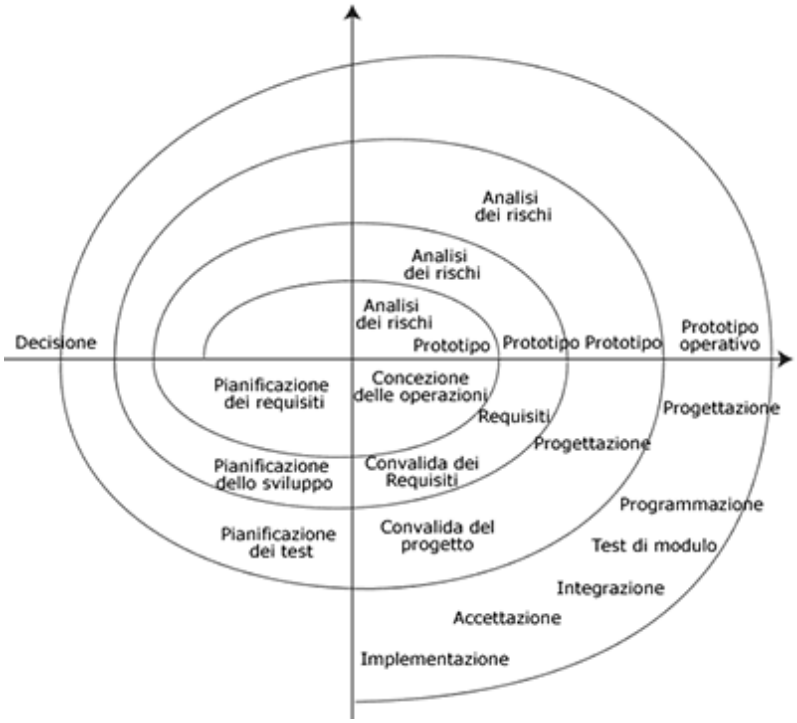


Figura 2.4: Modello a spirale

Il nome deriva dal fatto che un generico processo software viene descritto come una spirale: ogni giro della spirale corrisponde ad una fase del processo. Ogni giro della spirale è diviso in quattro parti.

1. *Analisi:* in questa fase vengono definiti gli obiettivi del progetto. Lo scenario e gli obiettivi vengono inizialmente stabiliti dal committente, per poi essere delineati nel dettaglio dal progettista.
2. *Progetto:* si cerca di valutare gli eventuali rischi del progetto e per ciascuno di essi si prendono precauzioni. Dopo questa fase iniziale verrà definita l'infrastruttura hardware e software.

3. *Sviluppo:* questa fase prevede la scrittura del codice e la verifica del prodotto software.
4. *Collaudo:* si verificano i risultati delle fasi precedentemente attraversate e si pianifica l'iterazione successiva

Il modello a spirale è molto attento alla questione "robustezza". Ad ogni iterazione viene effettuata una "valutazione del rischio". Con "rischio" intendiamo una situazione anomala. Un ciclo della spirale inizia con la valutazione dei requisiti e con l'elaborazione degli obiettivi; vengono inoltre elencati dei modi alternativi per raggiungerli. Ogni alternativa è valutata rispetto ad ogni obiettivo e vengono evidenziate le cause dei rischi. Successivamente, attraverso un'attività di raccolta di informazioni, vengono eliminati i suddetti rischi. Di conseguenza, ad ogni iterazione, requisiti di robustezza vengono trasformati in requisiti di correttezza.

La definizione di correttezza assume che siano disponibili le specifiche e che sia possibile determinare in maniera non ambigua se un programma rispetta i requisiti funzionali stabiliti inizialmente.

Un programma è robusto se si comporta "normalmente" anche in circostanze anomale, ovvero non previste durante l'analisi dei requisiti.

Possiamo dire che robustezza e correttezza sono caratteristiche strettamente intrecciate: se un requisito appartiene alle specifiche, il suo soddisfacimento diventa un problema di correttezza; se invece non appartiene, può diventare un problema di robustezza.

La metodologia iterativa propone un modello basato sulle stesse fasi del modello a cascata; la differenza consiste nel fatto che le fasi sono disposte lungo un cerchio, quindi l'ultima innesca un nuovo riciclo.

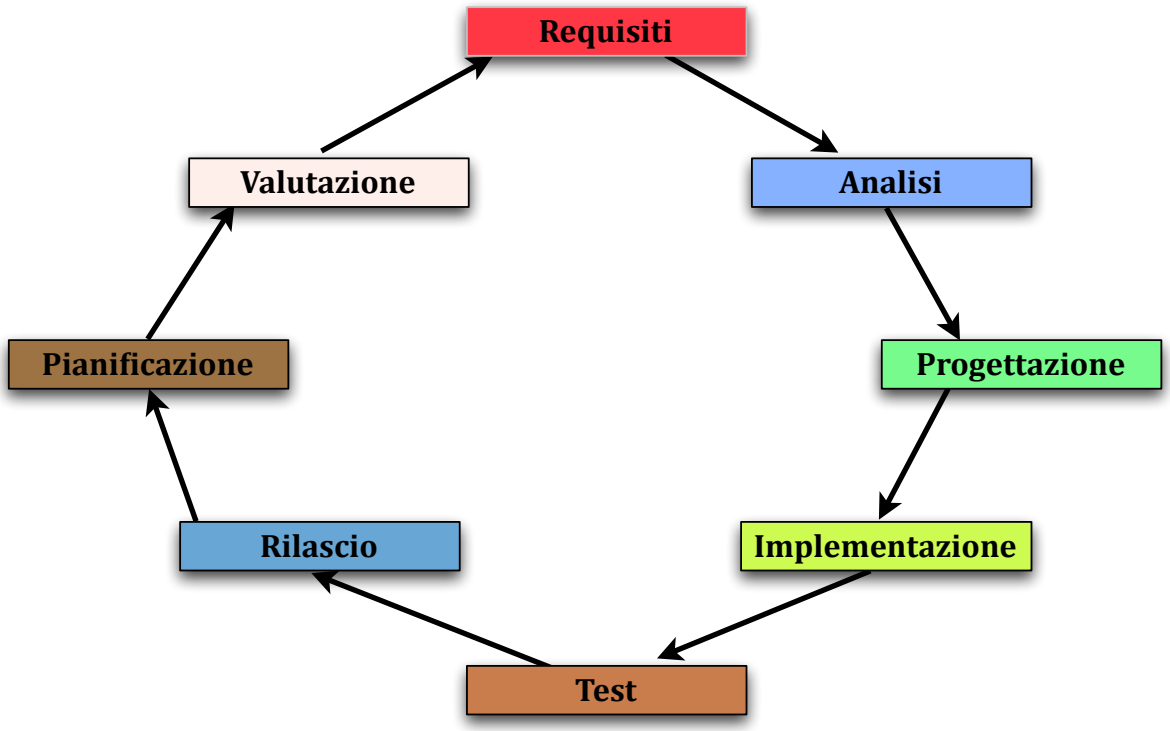


Figura 2.5: Modello iterativo

Il motivo che porta a questa evoluzione rispetto al modello a cascata deriva dal bisogno di avere un maggior controllo sul progetto. Ogni iterazione comprende attività di sviluppo che consentono il rilascio di una versione del sistema con funzionalità ridotte al fine di consentire un'evoluzione incrementale del sistema.

Nel contesto dei modelli di processo iterativi si inquadra Rational Unified Process (RUP).

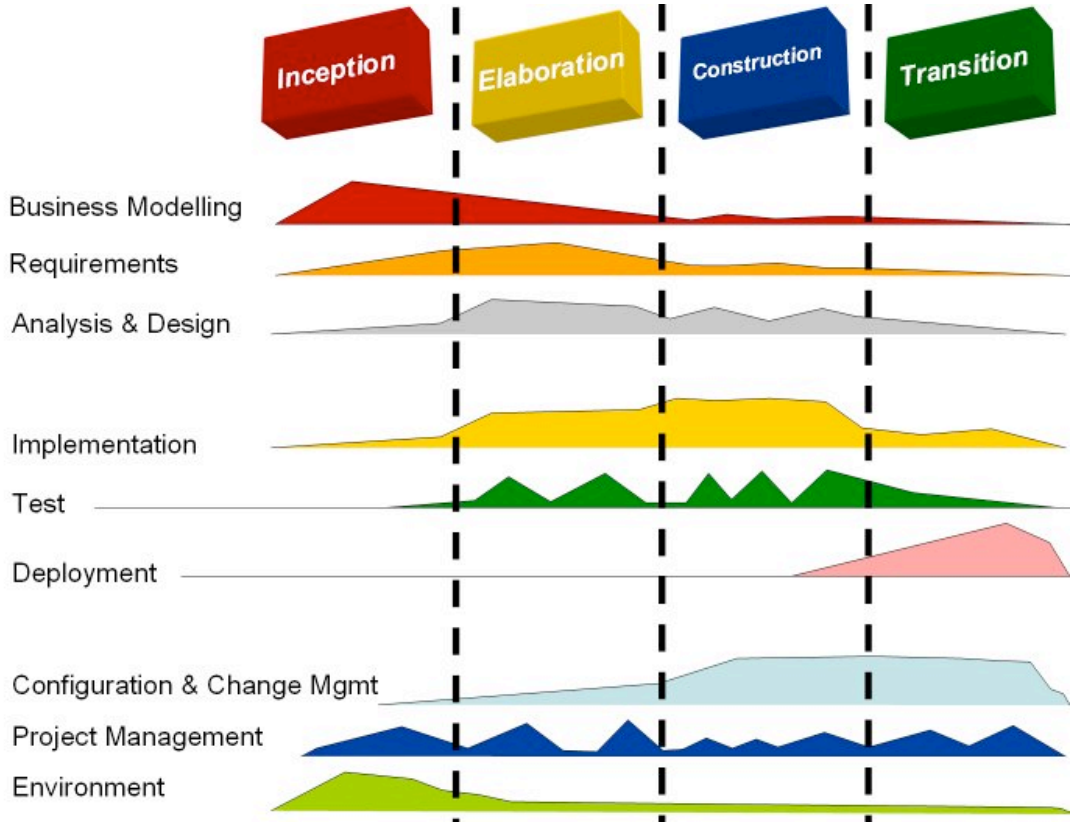


Figura 2.6:Rational Unified Process

È molto utilizzato dalle aziende in quanto definisce un'infrastruttura per la definizione di processi; inoltre rende il processo di produzione del software rigoroso e flessibile. RUP suddivide il progetto del processo software in quattro fasi che sono correlate agli aspetti aziendali.

1. *Avviamento:* si vanno a definire le interazioni tra "attori" (persone e sistemi esterni) e sistema. Si definiscono inoltre gli obiettivi del progetto e si effettua una stima dei costi.

2. *Elaborazione:* si determina il dominio del problema, si definiscono le caratteristiche funzionali, strutturali e architetturali del sistema e si valutano i rischi chiave. Alla fine di questa fase si dovrebbe avere un diagramma dei casi d'uso.
3. *Costruzione:* esegue il processo di implementazione e test del sistema; varie parti del sistema vengono sviluppate parallelamente per poi essere integrate. Al termine di questa fase si dovrebbe avere un sistema software pienamente funzionante, con relativa documentazione annessa.
4. *Transizione:* il sistema viene consegnato agli utenti finali. Questa fase, che viene generalmente ignorata dalla maggior parte dei modelli di processi software, è un'attività costosa e a volte problematica. Prevede le fasi di marketing, installazione, configurazione, supporto e mantenimento. Al completamento di questa fase, si dovrebbe avere un sistema software completo di documentazione pienamente funzionante.

L'iterazione favorisce il riuso dei componenti o, in particolare, di parti di codice. Alla fine di ogni iterazione è possibile effettuare una correzione degli errori, verificare lo stato del progetto e cambiare l'organizzazione del lavoro, rendendo il codice più robusto e migliorando il processo nella sua totalità; in questo modo inoltre si semplifica notevolmente la fase di test.

2.2 UML: un linguaggio per rappresentare il software

UML è una notazione per la modellazione. Può essere utilizzato per rappresentare in maniera astratta non solo sistemi software e hardware, ma qualunque insieme di oggetti, sia informatici che del mondo reale. È uno strumento molto utile non solo per gli ingegneri del software, ma anche per i business analyst.

Creare un modello di un sistema è il primo passo per poterlo poi implementare correttamente; il risultato della suddetta attività di modellazione sarà un insieme di diagrammi che rappresenteranno varie sfaccettature dell'entità da modellare. È molto importante verificare la correttezza del procedimento seguito. Esistono due livelli di correttezza: sintattica e semantica. La correttezza sintattica riguarda un corretto utilizzo della sintassi di UML. Con la correttezza semantica intendiamo verificare che il modello descriva correttamente ciò che si voleva inizialmente rappresentare.

Si ricorre ad un modello per rappresentare un'entità in modo semplificato "ad alto livello". Esistono vari tipi di modelli: ogni modello viene ottimizzato per un determinato fine.

Oggigiorno, la notazione UML rappresenta lo standard de facto nel mondo dell'ingegneria del software.

La scelta di un linguaggio di modellazione piuttosto che un altro dipende dal tipo di realtà che si intende modellare. Ogni linguaggio è caratterizzato da un insieme di parole chiave che rappresentano i concetti principali del dominio che si vuole andare a modellare. È quindi normale come alcuni linguaggi non risultano adatti a rappresentare aspetti di interesse di

determinate entità. Ad esempio, non è possibile utilizzare i simboli di una carta stradale per modellare una canzone.

Un aspetto fondamentale da tenere presente nella scelta di un linguaggio di modellazione è la facilità d'uso: facilità nel generare modelli, nel leggere e nel comprendere un modello sono le parole chiave. Infatti, il risultato dell'attività di modellazione è rappresentato da una serie di diagrammi che derivano da un "confronto" tra cliente e progettista. Una misura della facilità d'uso di un modello è rappresentata dai vantaggi che si possono trarre da un suo utilizzo. La formalità di un linguaggio di modellazione è fondamentale per le attività di analisi e di elaborazione: generalmente questi linguaggi vengono corredati di un sistema di supporto informatico che consenta la produzione e la valutazione di modelli in maniera abbastanza semplificata. Avendo precedentemente parlato di processi di modellazione, è importante sottolineare come questi ultimi non debbano essere necessariamente correlati con i suddetti linguaggi di modellazione. Ad esempio, riferendoci al mondo dei sistemi informatici, un linguaggio può essere utilizzato tanto per modellare il dominio di un sistema esistente, quanto per il design di un sistema ancora da produrre.

Il principale punto di forza di UML è l'essere un insieme di notazioni visuali object-oriented, progettate per descrivere classi, oggetti e relativi "comportamenti". La scelta di adottare una famiglia di notazioni consente di descrivere un'entità da modellare da più punti di vista. È possibile, infatti, avere diagrammi che descrivono la struttura statica del sistema, quella dinamica, il comportamento delle classi e così via. È importante che le varie viste del sistema siano tra loro consistenti. L'altro punto di forza è rappresentato dal metamodello di UML. Quest'ultimo definisce una notazione e, appunto, un meta-modello. La notazione è costituita dagli elementi grafici che caratterizzano i diagrammi. Un metamodello è un diagramma, solitamente delle classi, che definisce i concetti del linguaggio in questione: serve per esprimere un modello. Serve per aumentare il rigore formale del linguaggio mantenendo al tempo stesso la facilità d'uso. Un metametamodello definisce il linguaggio necessario per esprimere metamodelli. UML fa parte di un'architettura standardizzata per la modellazione di OMG (Object Management Group) chiamata MOF (Meta-Object Facility): un meta-metamodello che serve a definire UML. MOF ha 4 livelli: M0, M1, M2 e M3. Ogni livello è un'istanza di un elemento del livello superiore.

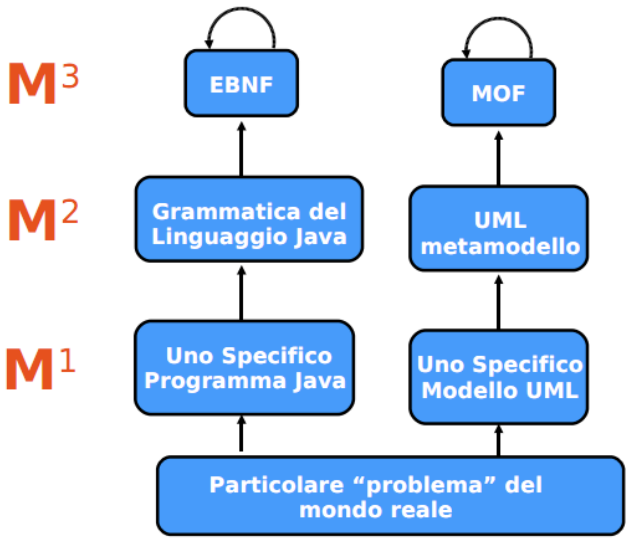


Figura 2.7: Livelli dell'architettura Meta-Object Facility

I linguaggi di modellazione, alla stessa stregua dei linguaggi di programmazione, sono specificati da grammatiche definite come metamodelli.

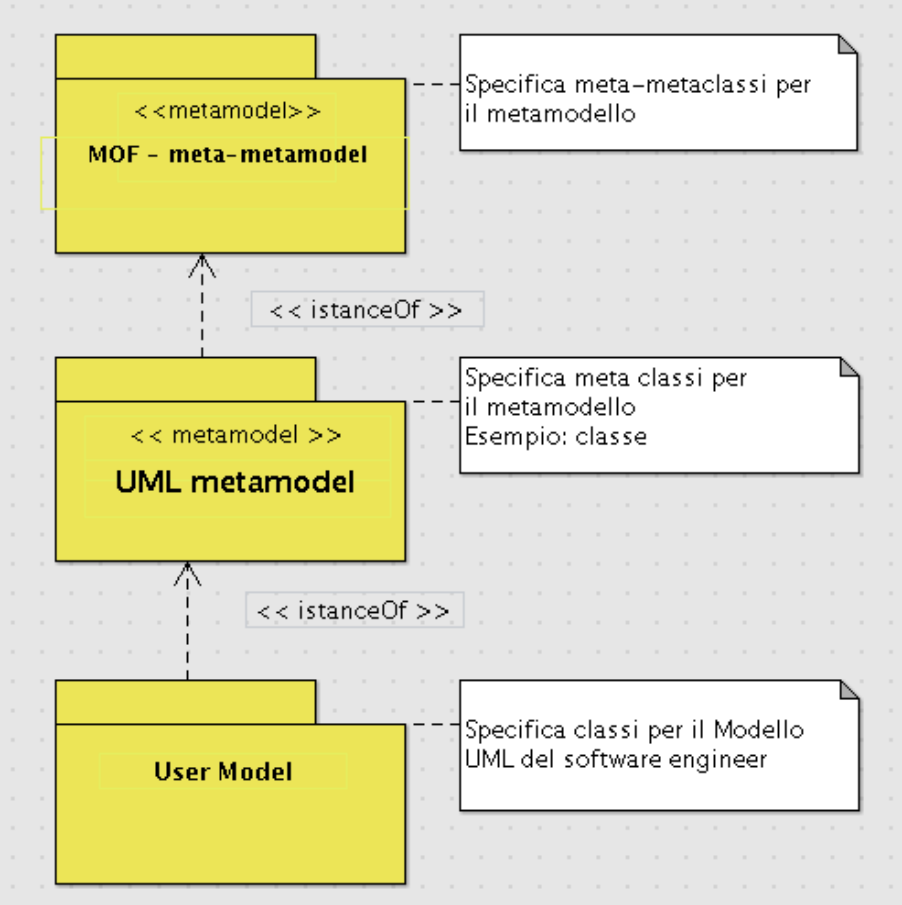


Figura 2.7.1: Dettagli dell'architettura Meta-Object Facility

L'utilizzo del substrato fornito dal metamodello consente di manipolare automaticamente i modelli prodotti: questo è uno dei principali motivi di successo di UML. Inoltre il metamodello si serve di una rappresentazione standard nel linguaggio XML per l'interscambio di informazioni tra i diversi strumenti UML. La scelta di incorporare in UML notazioni provenienti dal mondo "object-oriented" è risultata strategica per la rapida diffusione di questo linguaggio.

Un altro motivo di successo di UML è rappresentato dalla sua adattabilità. Sin dalle versioni iniziali è possibile estendere la notazione di base per renderla più usabile.

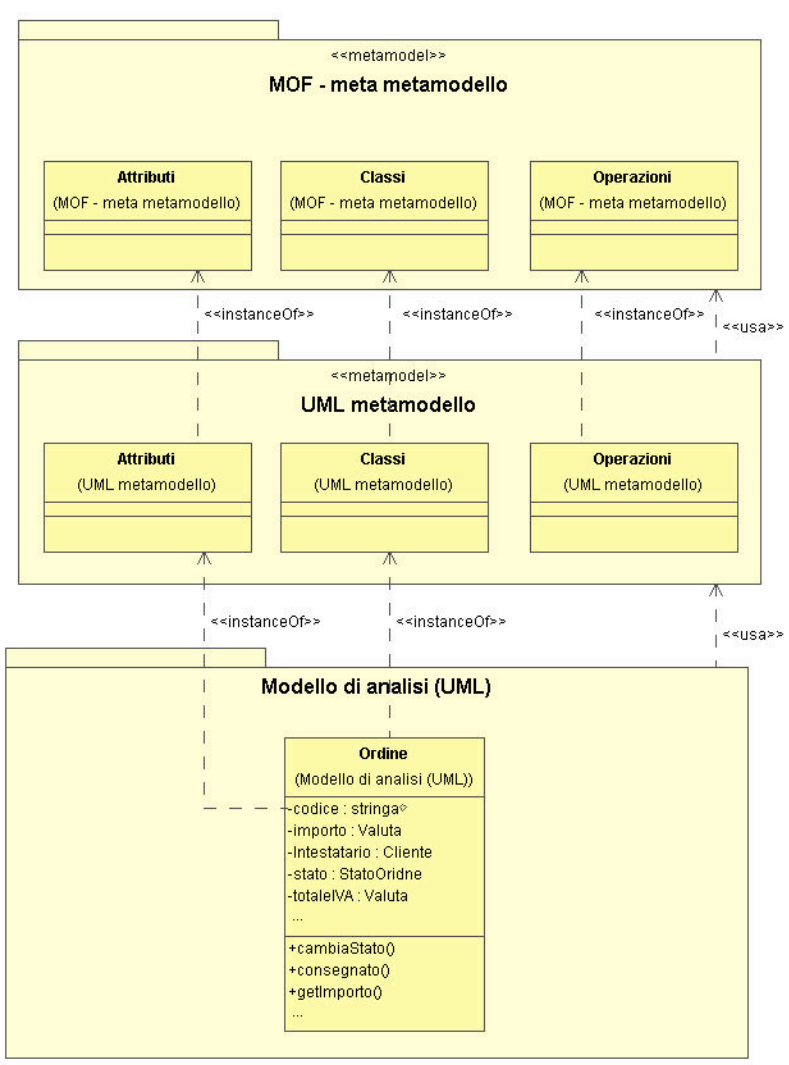


Figura 2.7.2: Dettagli dell'architettura Meta-Object Facility

Capitolo 3

LabAnalisi

3.1 Introduzione

Nella seguente tesina si è scelto di descrivere, mediante le tecniche dell'ingegneria del software, il funzionamento di un laboratorio analisi.

3.2 Requisiti funzionali

Il sistema prevede l'interazione di quattro attori: amministratore, medico, segretario e paziente. La registrazione di un paziente all'interno del database viene effettuata da un segretario previa compilazione di un semplice form: durante la registrazione, oltre ai dati anagrafici, viene anche richiesto l'indirizzo e-mail. La registrazione di un medico viene effettuata dall'amministratore del sistema, che è unico. La de-registrazione di tutti gli utenti è anch'essa a cura dell'amministratore. Un utente registrato può, in maniera automatica e senza intervento alcuno da parte dell'amministratore, recuperare e modificare la password. Ogni paziente all'interno del database è identificato attraverso una scheda, che può essere creata e modificata sia da un segretario che da un medico; quest'ultimo ha inoltre la possibilità di allegare documenti. La cancellazione di documenti e/o allegati può essere effettuata solamente dal proprietario degli stessi o dall'amministratore del sistema. È stata prevista una cronologia delle revisioni che consente di visualizzare tutte le modifiche effettuate su una scheda e sui relativi allegati. Ogni paziente può ricevere, sull'indirizzo e-mail scelto al momento della registrazione, notifiche relative alla disponibilità dei risultati delle analisi.

3.3 Requisiti non funzionali

Ogni utente all'interno del sistema è identificato da una scheda. Il sistema consente la gestione concorrente di circa un migliaio di schede. Il sistema è sempre attivo.

3.4 Use case diagram

3.4.1 Introduzione

Un *use case* descrive l'interazione tra un utente e il sistema; serve per catturare il comportamento esterno del sistema da sviluppare, utilizzando un approccio di tipo "black-box". Questo tipo di costrutto è particolarmente cruciale nella fase di analisi di sviluppo del sistema: rappresenta un ottimo strumento per stimolare gli utenti ad esprimere, da un punto di vista pratico piuttosto che tecnico, le funzionalità che il sistema dovrà avere.

Le interviste con gli utenti portano alla definizione di use case ad alto livello e degli attori del sistema. L'interazione tra use case e attori viene definita per mezzo di una sequenza di messaggi scambiati tra gli attori e il sistema. Un use case diagram descrive i requisiti funzionali che il sistema dovrà fornire ai suoi utenti. In UML, gli attori sono rappresentati attraverso l'icona di un omino stilizzato, un use case mediante un'icona ellissoidale e un nome che caratterizza l'interazione stessa. Il nome di un use case deve comunicare ad un lettore profano del diagramma le funzionalità da esso rappresentate.

Le relazioni usate nei diagrammi successivi sono quelle di *inclusione* e di *estensione*. In una relazione di inclusione («include»), rappresentata da una linea tratteggiata con indicazione dello stereotipo include, un caso d'uso (quello alla base della freccia) include esplicitamente il comportamento di un altro caso d'uso (quello alla punta).

In una relazione di estensione («extend»), rappresentata da una linea tratteggiata con indicazione dello stereotipo extend, un caso d'uso base (quello alla base della freccia) include implicitamente il comportamento di un altro caso d'uso (quello alla punta) in uno o più punti specificati, chiamati punti di estensione. Questa relazione viene generalmente utilizzata per isolare, in un use case, comportamenti che si verificano solamente in determinate circostanze.

Di seguito viene illustrata una visione generale dello use case diagram:

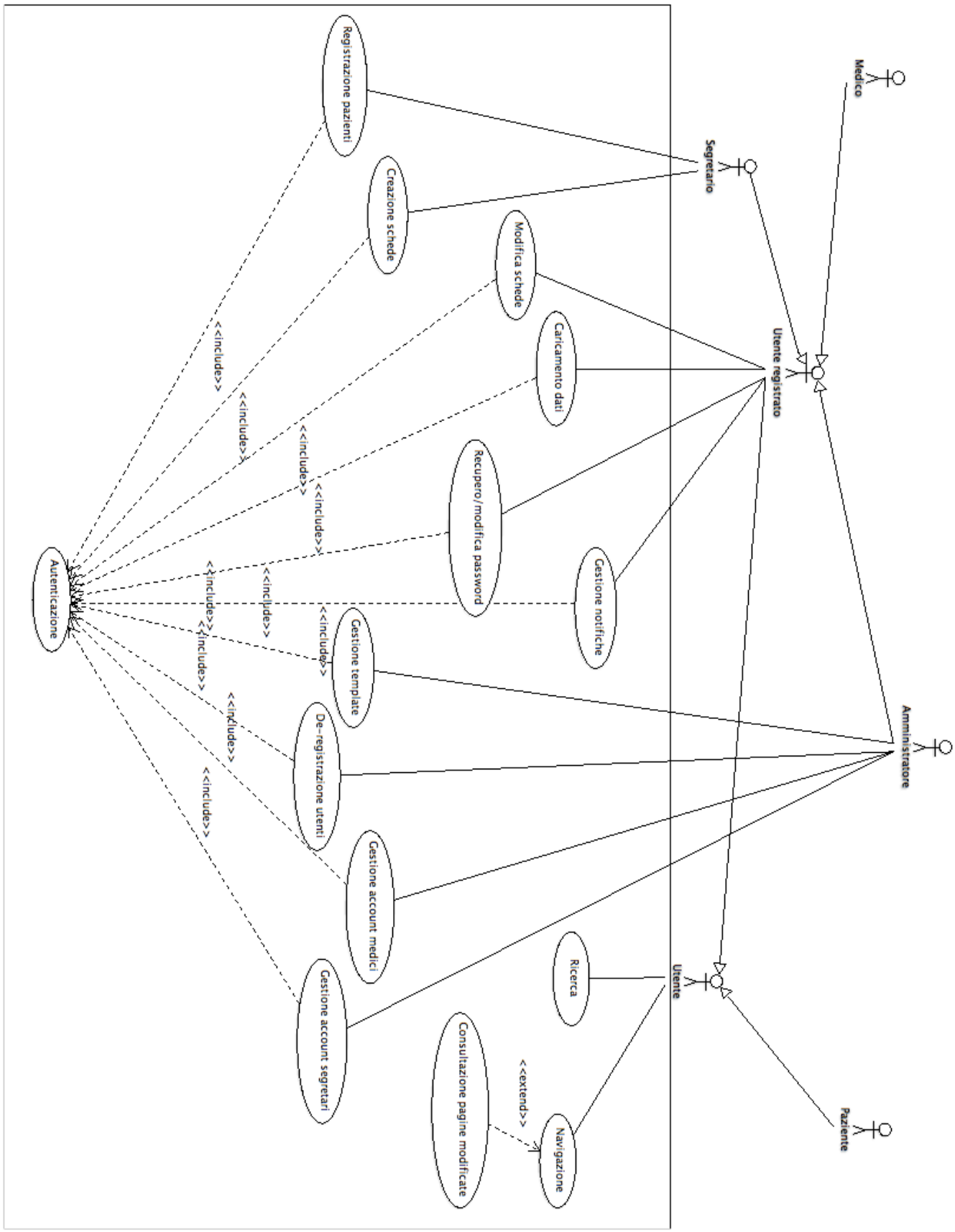


Figura 3.1: Use Case Diagram

3.5 Descrizione dei casi d'uso

Vengono descritti di seguito, in accordo con la specifica dei requisiti, i principali casi d'uso del sistema in questione. Per analizzare il sistema, si è seguito un approccio di tipo top-down: ciascun caso d'uso verrà in seguito indicato nel dettaglio attraverso l'utilizzo di diagrammi delle attività, che ben si prestano alla rappresentazione di sistemi caratterizzati da un elevato numero di diagrammi di flusso.

3.5.1 Introduzione

Gli attori del sistema sono quattro: **Paziente, Medico, Segretario, Amministratore**. Un Paziente rappresenta inizialmente un utente esterno al laboratorio. La registrazione è a cura del Segretario. Medico e Segretario sono due tipi particolari di utenti registrati. Un Segretario ha, inoltre, il compito di schedare di tutti i pazienti che entrano nel laboratorio. Un Amministratore è anch'esso un particolare tipo di utente registrato: ha permessi di lettura e scrittura su qualunque risorsa del sistema.

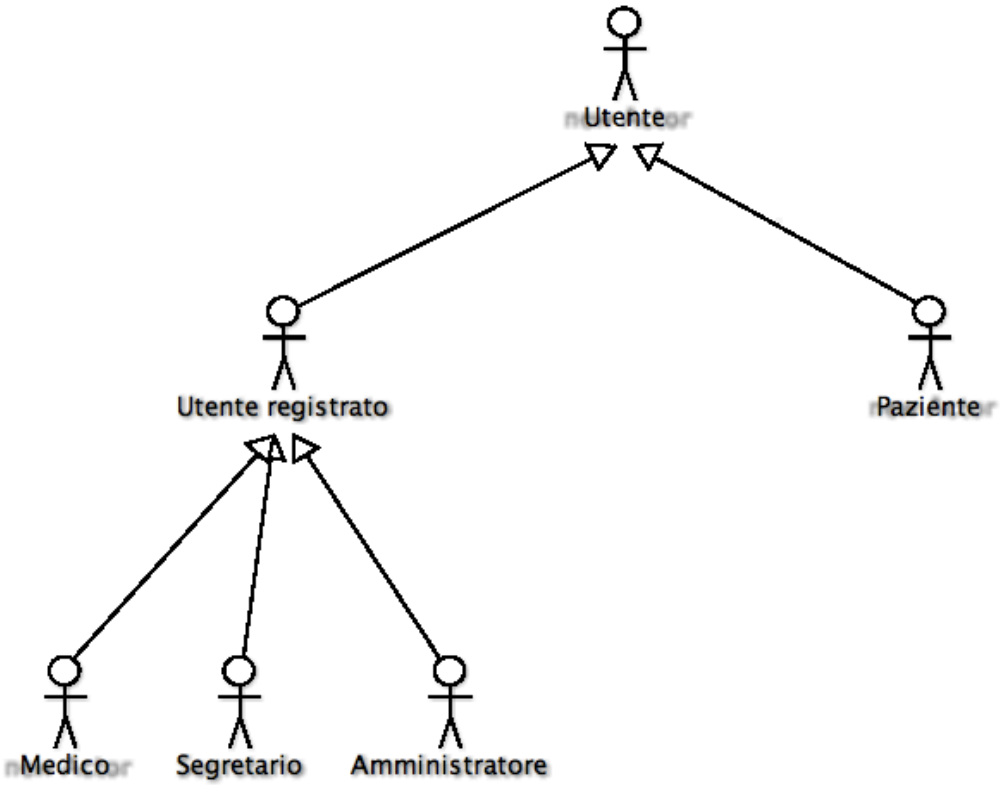


Figura 3.2: Gerarchia degli attori

3.5.2 Autenticazione

In questa fase, un utente registrato può autenticarsi al sistema. In base alle credenziali fornite, il sistema consentirà l'accesso a determinate funzionalità. Lo scenario descritto prevede la presentazione all'utente, da parte del sistema, di una schermata con un form, attraverso il quale l'utente potrà inserire la "username" e la "password". Dopo aver verificato la validità dei dati immessi dall'utente, il sistema autorizzerà l'utente ad accedere ai contenuti del sistema; in caso contrario, apparirà una schermata con un messaggio di errore e l'utente verrà reindirizzato alla pagina di login.

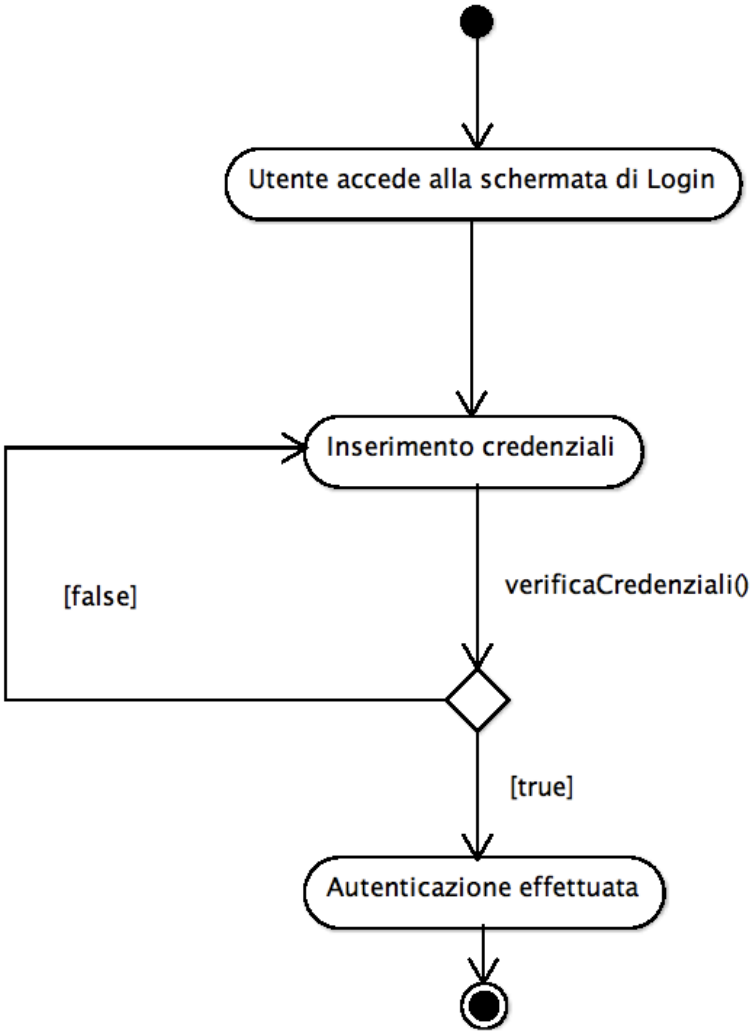


Figura 3.3: Autenticazione

3.5.3 Registrazione e creazione scheda

Gli attori coinvolti in questo scenario sono il Segretario e il Paziente. Il sistema deve consentire la registrazione di pazienti e la conseguente creazione di una scheda previo compilazione di un form da parte di un segretario, il quale inserirà le credenziali fornite dal paziente. Risulta ovvio come questa fase sia preliminare rispetto a quella di autenticazione. Il sistema presenta all'utente una pagina con un form contenente i campi: "Nome", "Cognome", "Età", "CodiceFiscale", "Indirizzo", "Sesso" e "Telefono". Dopo aver verificato la correttezza sintattica e semantica dei dati inseriti, il sistema aggiungerà un record al database con i dati forniti dal paziente; in caso contrario, apparirà una schermata con un messaggio di errore e il segretario verrà reindirizzato alla pagina di login.

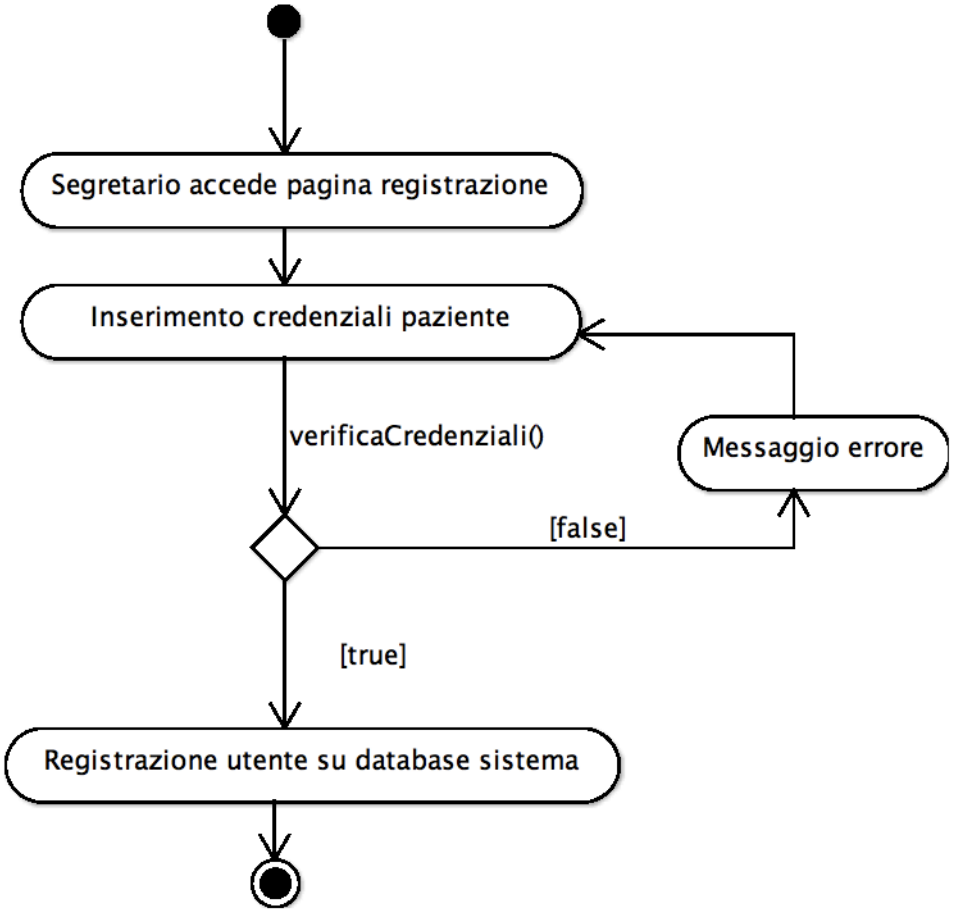


Figura 3.4 Fase di registrazione

3.5.4 De-registrazione

In questa fase, viene permessa la rimozione di un utente registrato dal sistema; la de-registrazione è a cura dell'amministratore del sistema. Con questa azione, verrà eliminato l'account annesso all'utente de-registrato.

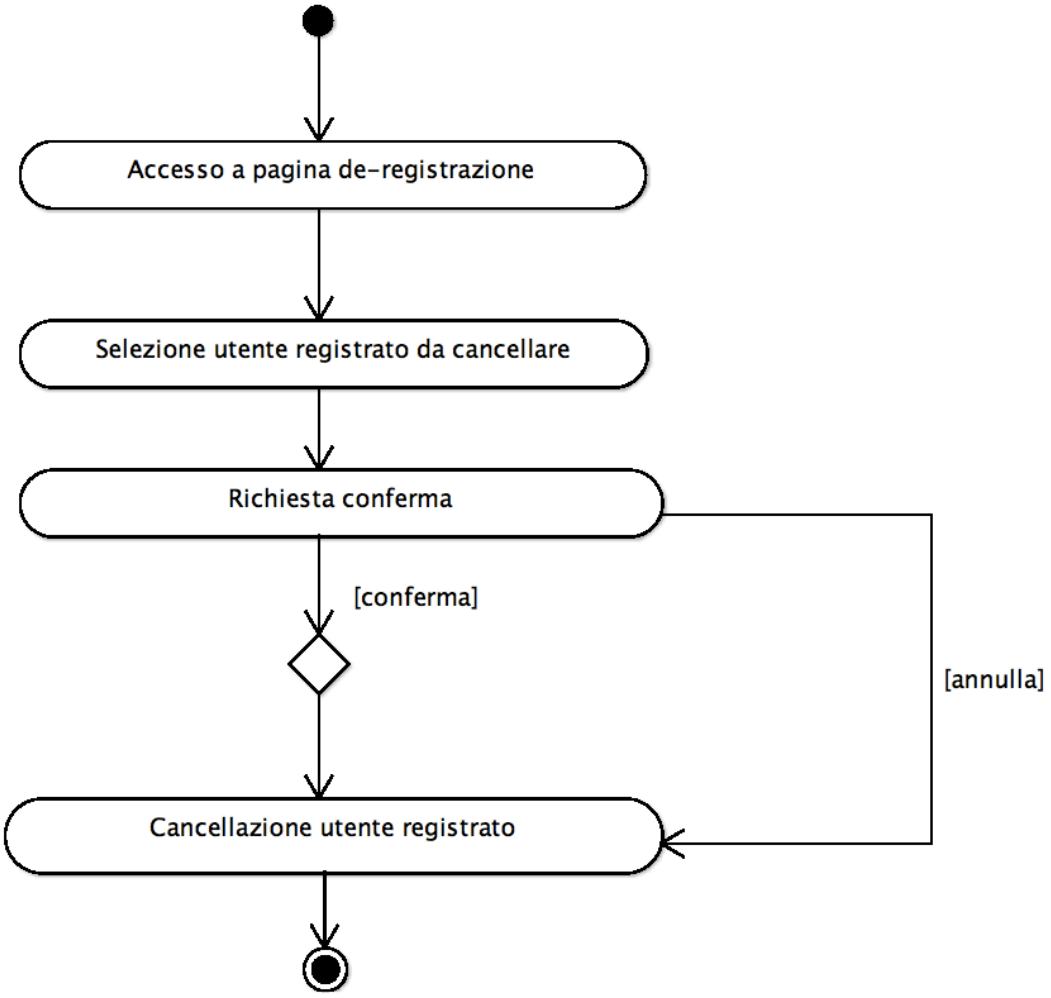


Figura 3.5: De-registrazione

3.5.5 Recupero password

Nel caso in cui un utente registrato smarrisca la propria password, il sistema deve essere nelle condizioni di potergliela restituire. Attraverso un'apposita pagina, l'utente potrà immettere l'indirizzo e-mail utilizzato in fase di registrazione. In caso di validità del dato inserito, il sistema spedirà una e-mail all'indirizzo immesso contenente una password temporanea; in caso contrario, apparirà una schermata con un messaggio di errore e l'utente verrà reindirizzato alla pagina che consente di effettuare il recupero della password.

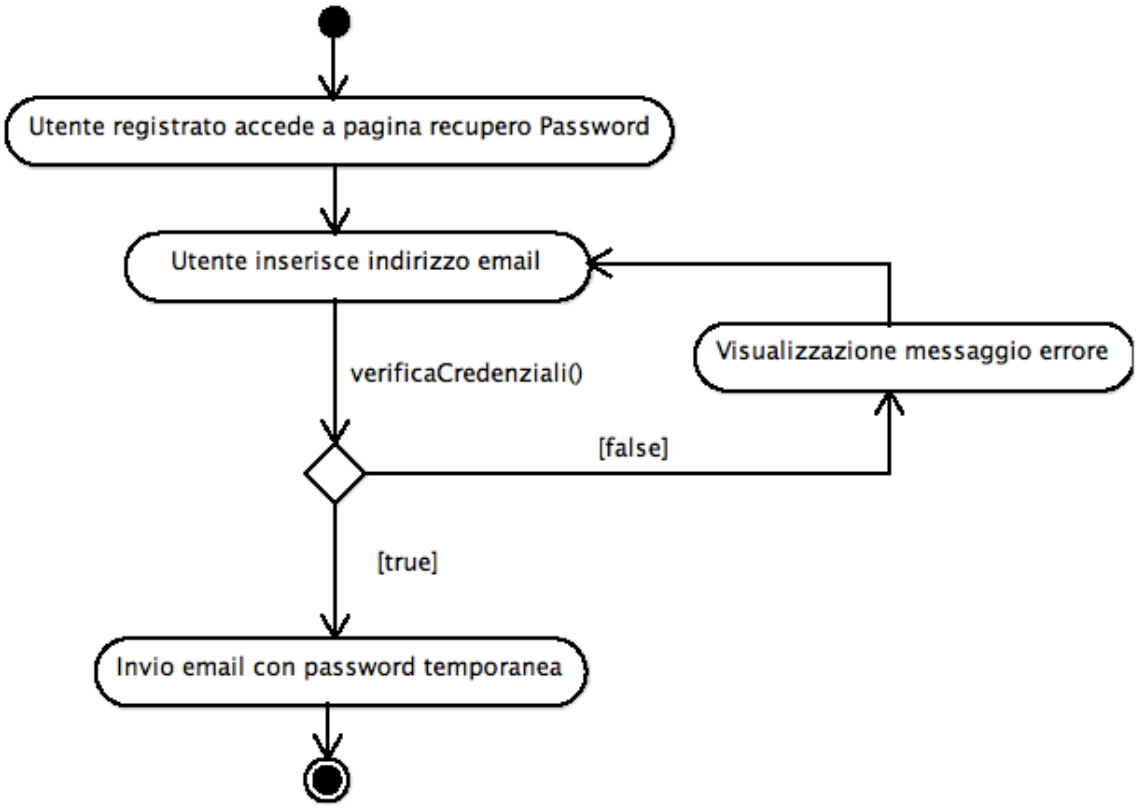


Figura 3.6: Fase di recupero password

3.5.6 Modifica password

Un utente registrato deve essere in grado di modificare la propria password di accesso al sistema. Attraverso un'apposita finestra, nella quale sarà presente un form da compilare, l'utente potrà immettere la nuova password. Il sistema, dopo aver verificato la validità dei dati immessi dall'utente, procede alla modifica della password; in caso contrario, apparirà una schermata con un messaggio di errore e l'utente verrà reindirizzato alla pagina che consente di effettuare il recupero della password.

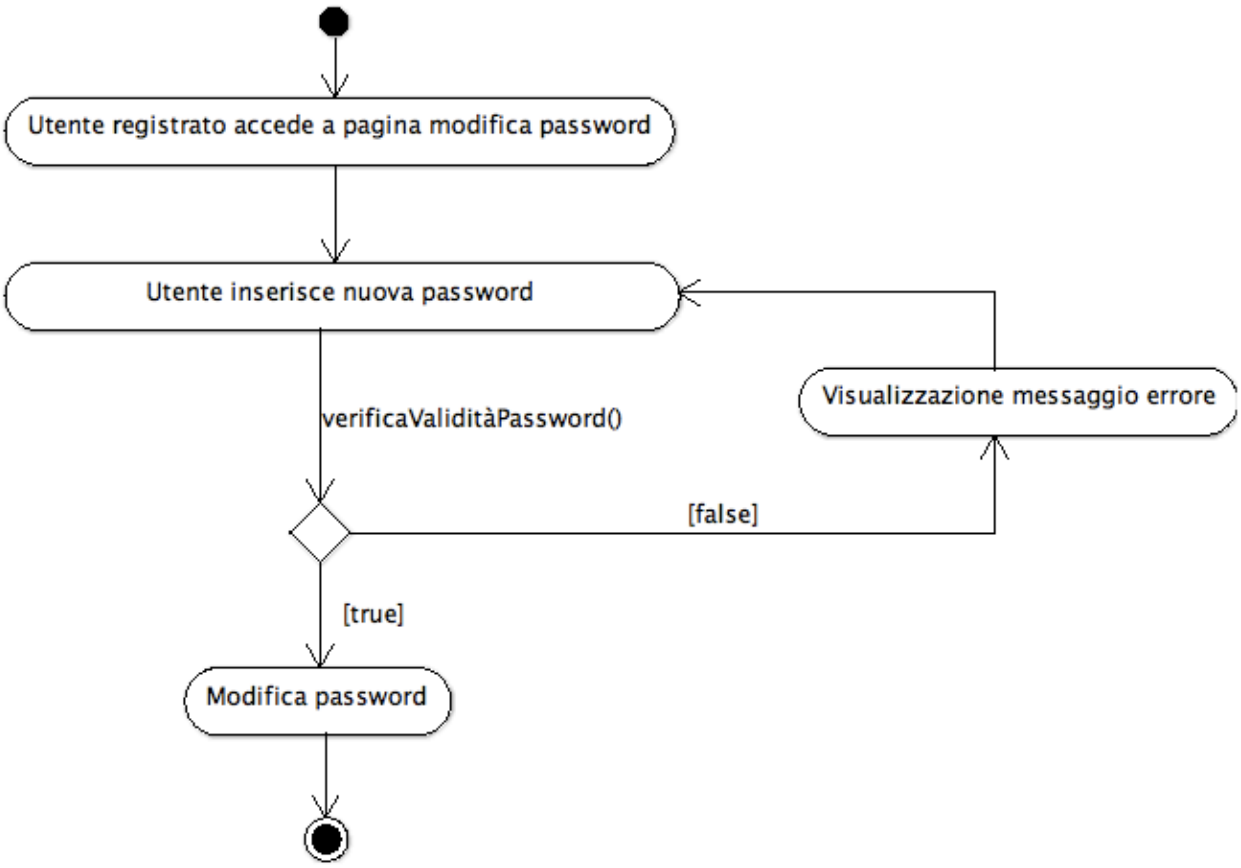


Figura 3.7: Modifica password

3.5.7 Modifica scheda

La modifica di una scheda, la quale riguarda esclusivamente un paziente, può essere effettuata sia da un medico che da un segretario. Il primo può aggiungere commenti alla scheda, il secondo può modificare dati anagrafici e allegare documenti. A differenza di un segretario e di un medico che hanno accesso in scrittura ad una scheda, un paziente ha esclusivamente accesso in lettura. Il sistema, dopo aver verificato che l'utente abbia diritto di accesso in scrittura alla scheda che vuole modificare e dopo aver verificato che la risorsa non sia occupata, blocca la risorsa attraverso un meccanismo di mutua esclusione e mostra la schermata di modifica all'utente. Dopo aver apportato le modifiche, l'utente salva la scheda e il sistema rilascia il lock precedentemente impostato sulla risorsa.

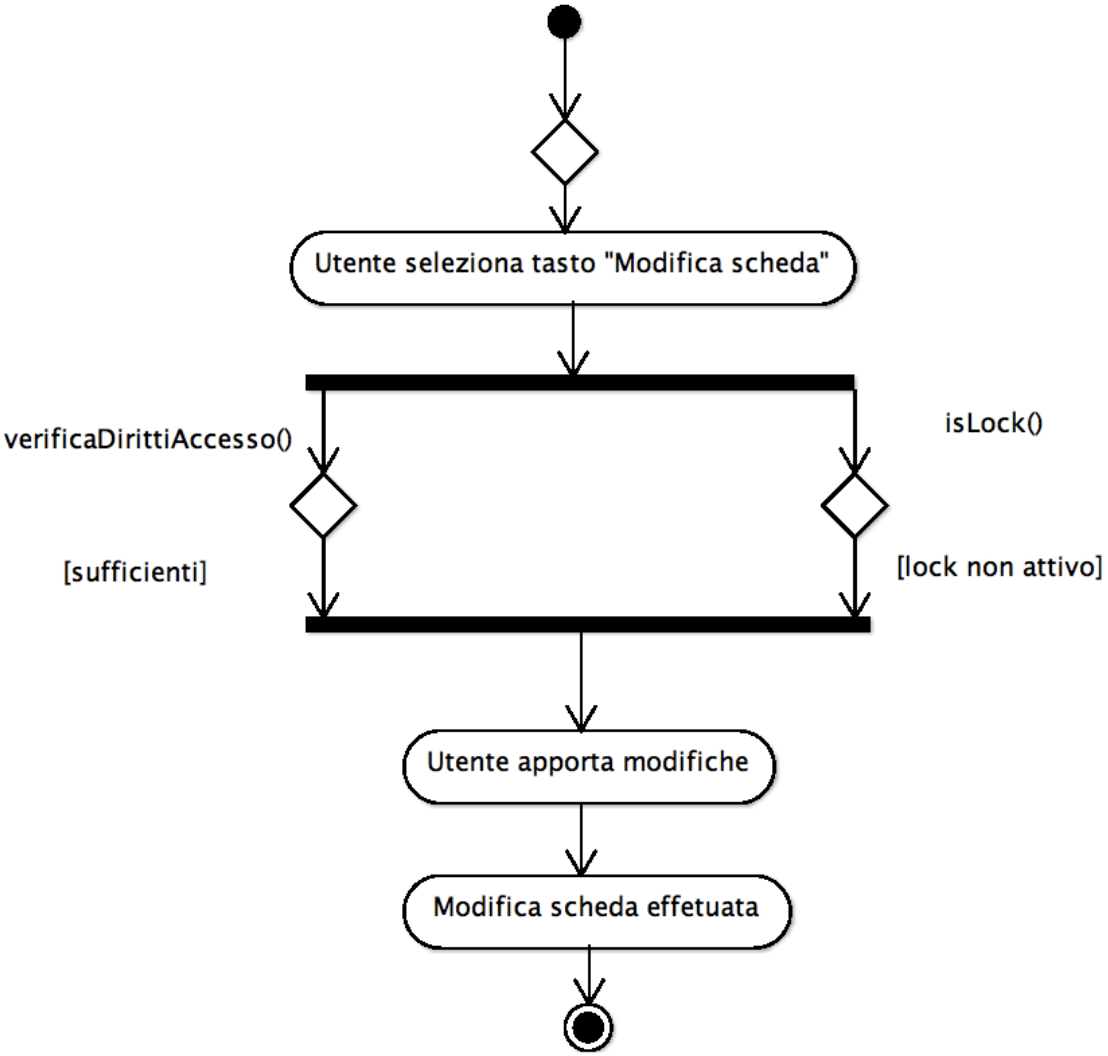


Figura 3.8: Modifica scheda

3.5.8 Registrazione medici e segretari

La creazione dell'account di un medico o di un segretario è esclusiva prerogativa dell'amministratore del sistema. Il sistema mostrerà una pagina contenente un form con i seguenti campi:

- un campo per il nome e il cognome del medico / segretario
- un campo per l'indirizzo e-mail del medico / segretario

In seguito all'immissione dei dati, il sistema verificherà che l'utente inserito non sia già presente in archivio e che l'indirizzo e-mail sia valido; in caso affermativo, il sistema registrerà l'utente e spedirà una mail di avvenuta registrazione con una password temporanea. In caso contrario, apparirà una schermata con un messaggio di errore e l'amministratore verrà reindirizzato alla pagina di registrazione.

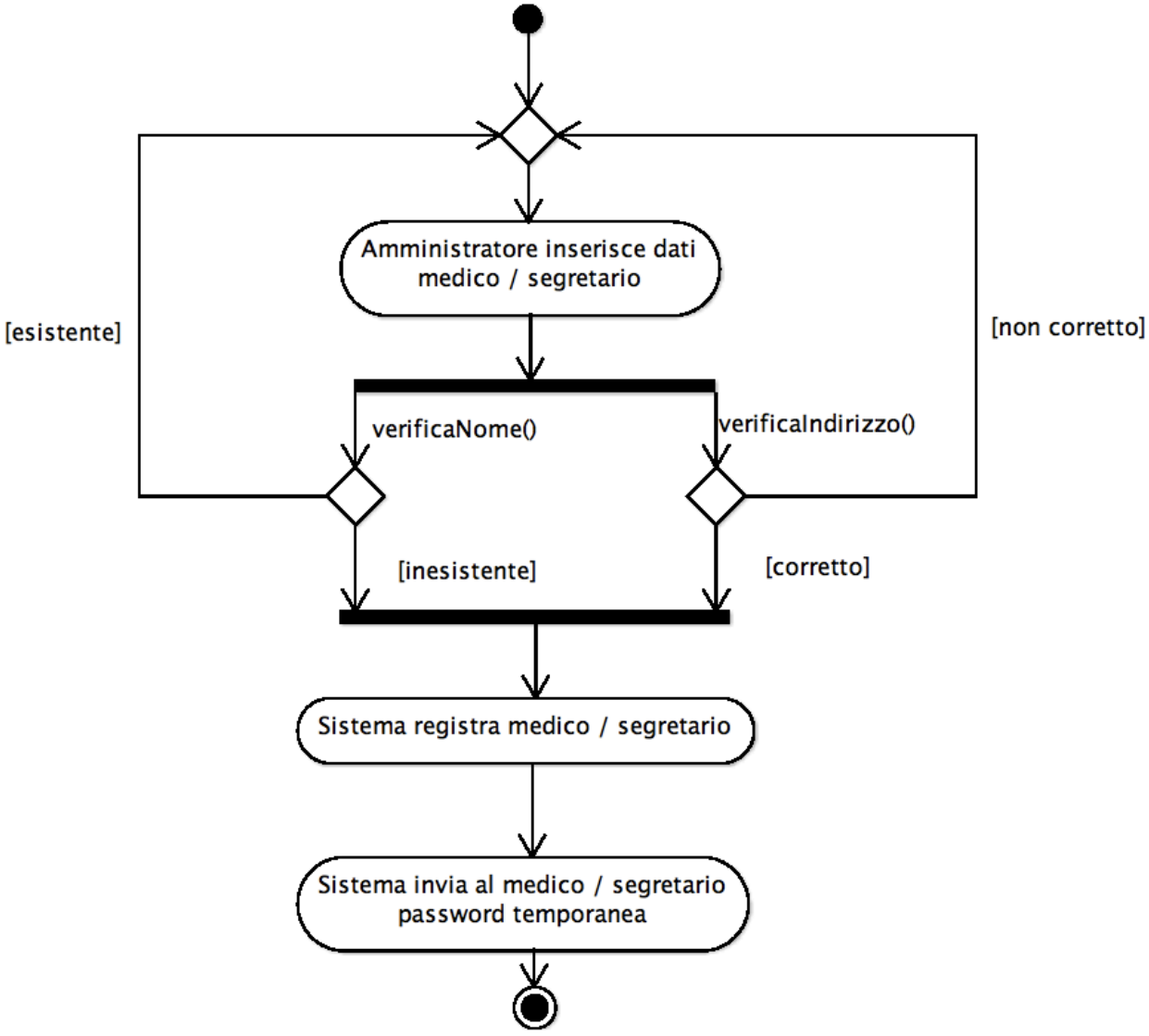


Figura 3.9: Registrazione di un medico / segretario

3.5.9 Upload di un file

Un utente registrato può effettuare allegare un file ad una scheda già presente in archivio. Generalmente questo compito viene affidato ad un segretario. L'utente seleziona il file che vuole allegare: il sistema allega il file alla scheda e avverte l'utente con un messaggio di buon fine.

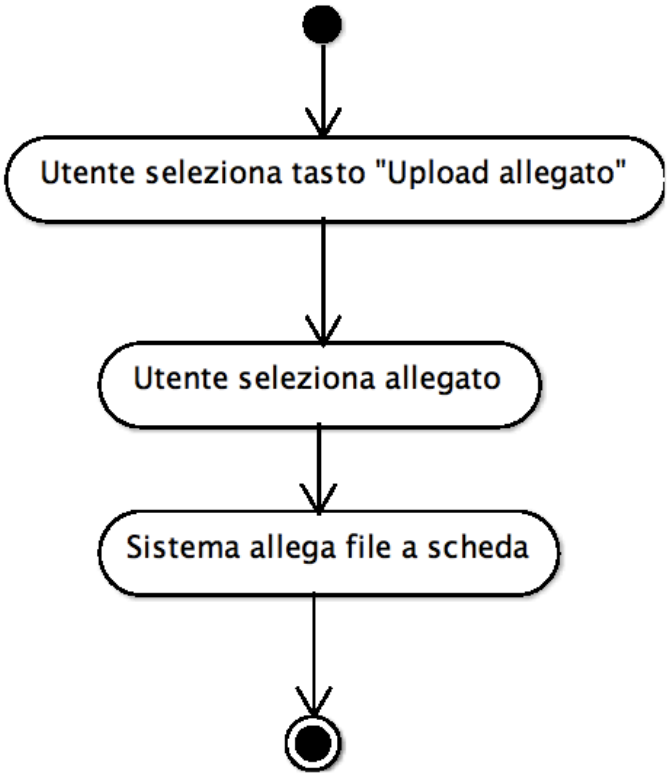


Figura 3.10: Upload di un file

3.5.10 Impostazione notifiche

Con questa funzionalità, un paziente può scegliere tra le opzioni “Attiva notifica via email” o “Nessuna notifica”. Nel primo caso, il paziente riceverà notifiche sull’e-mail quando vengono apportate modifiche alla sezione “Note” o quando viene allegato un file alla propria scheda.

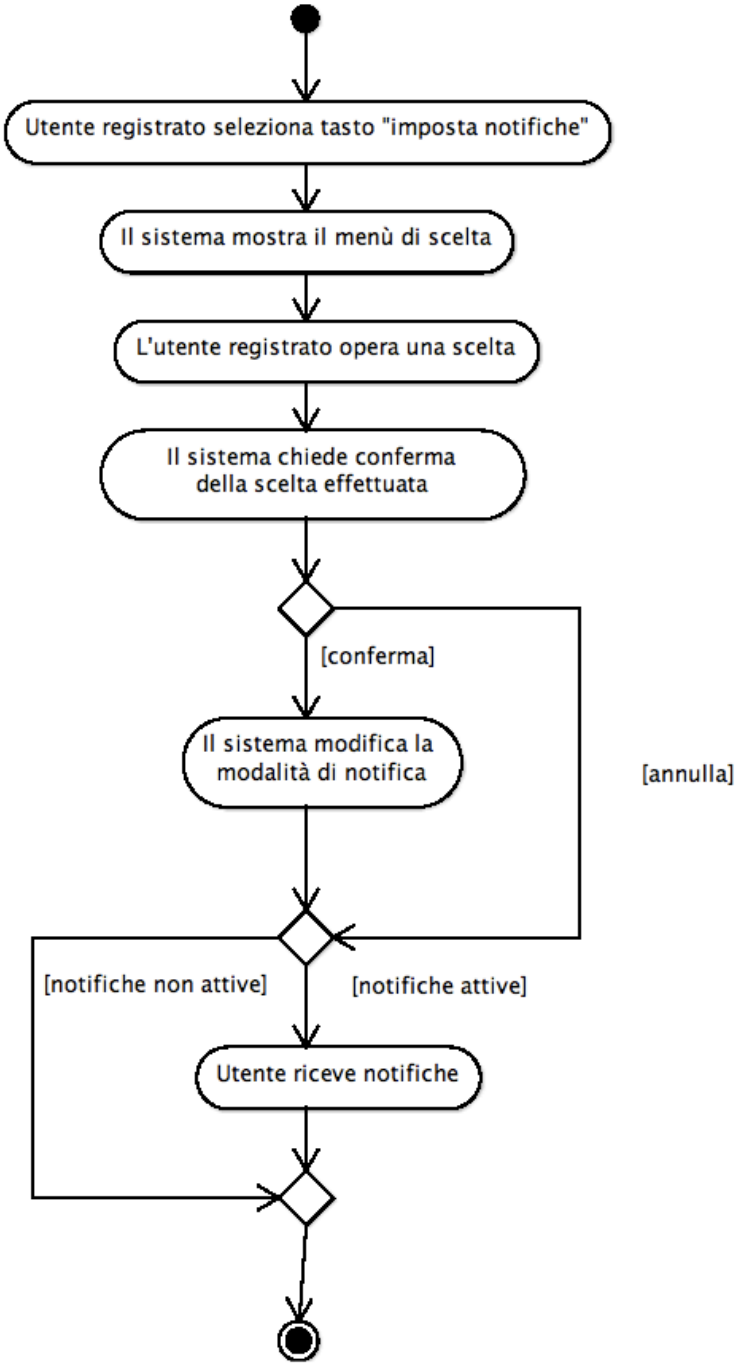


Figura 3.11: Impostazione notifiche

3.5.11 Commento alle schede

Un utente registrato, generalmente un medico, deve poter aggiungere dei commenti alla sezione “Note” della scheda di un paziente. Dopo aver inserito un commento, un utente può decidere se cancellare o salvare il commento poco prima inserito; il sistema registra la scelta dell’utente e agisce di conseguenza. Se un utente prova a cancellare un commento di cui non è autore, il sistema mostra una pagina con un messaggio di errore.

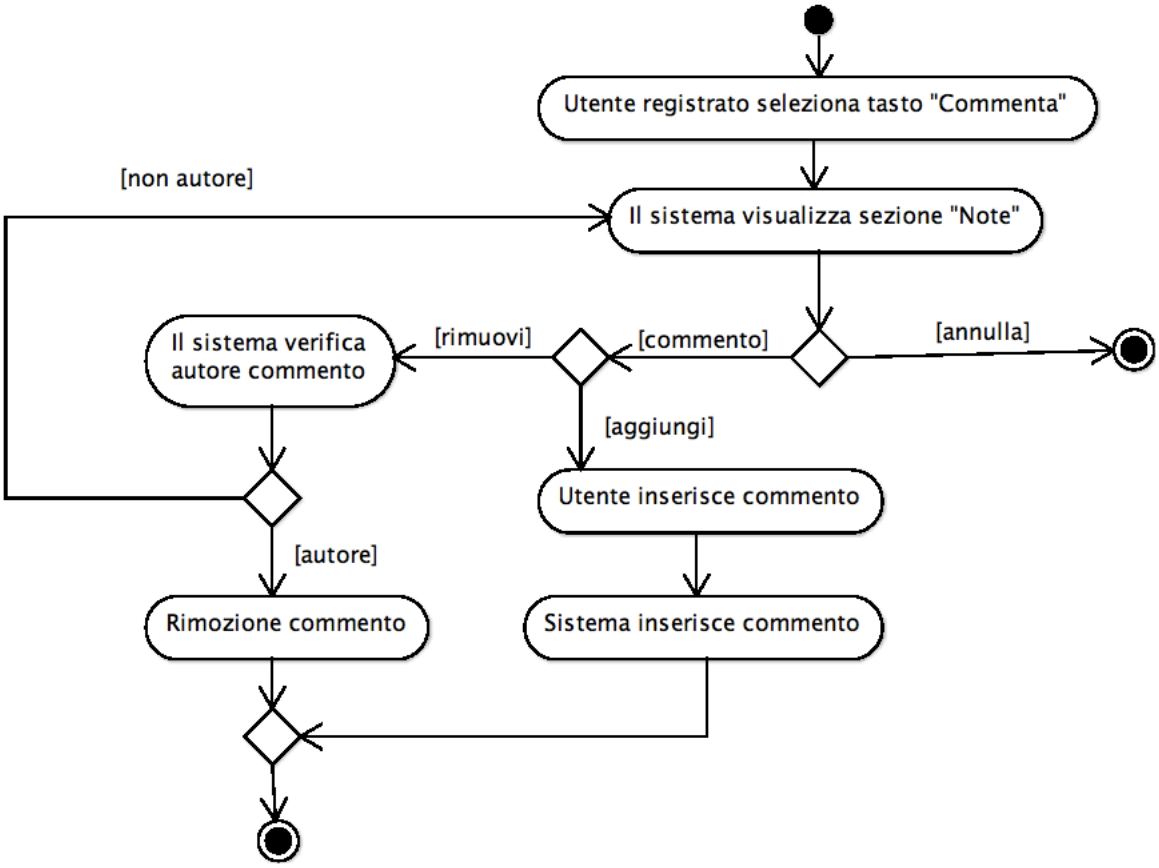


Figura 3.12: Commenta schede

3.6 Class diagram

3.6.1 Introduzione

Un *class diagram* descrive le classi e le interfacce che verranno usate per tipare gli oggetti del sistema; rappresenta il diagramma maggiormente usato dagli autori. I Class Diagram rappresentano l'evoluzione Object-Oriented dei diagrammi Entity-Relationship ben noti nel mondo dei database. Lo scopo di questo diagramma è quello di definire la visione statica del sistema. Come suggerisce il nome, un class diagram è composto da classi (di oggetti) e della relazioni che intercorrono tra queste ultime.

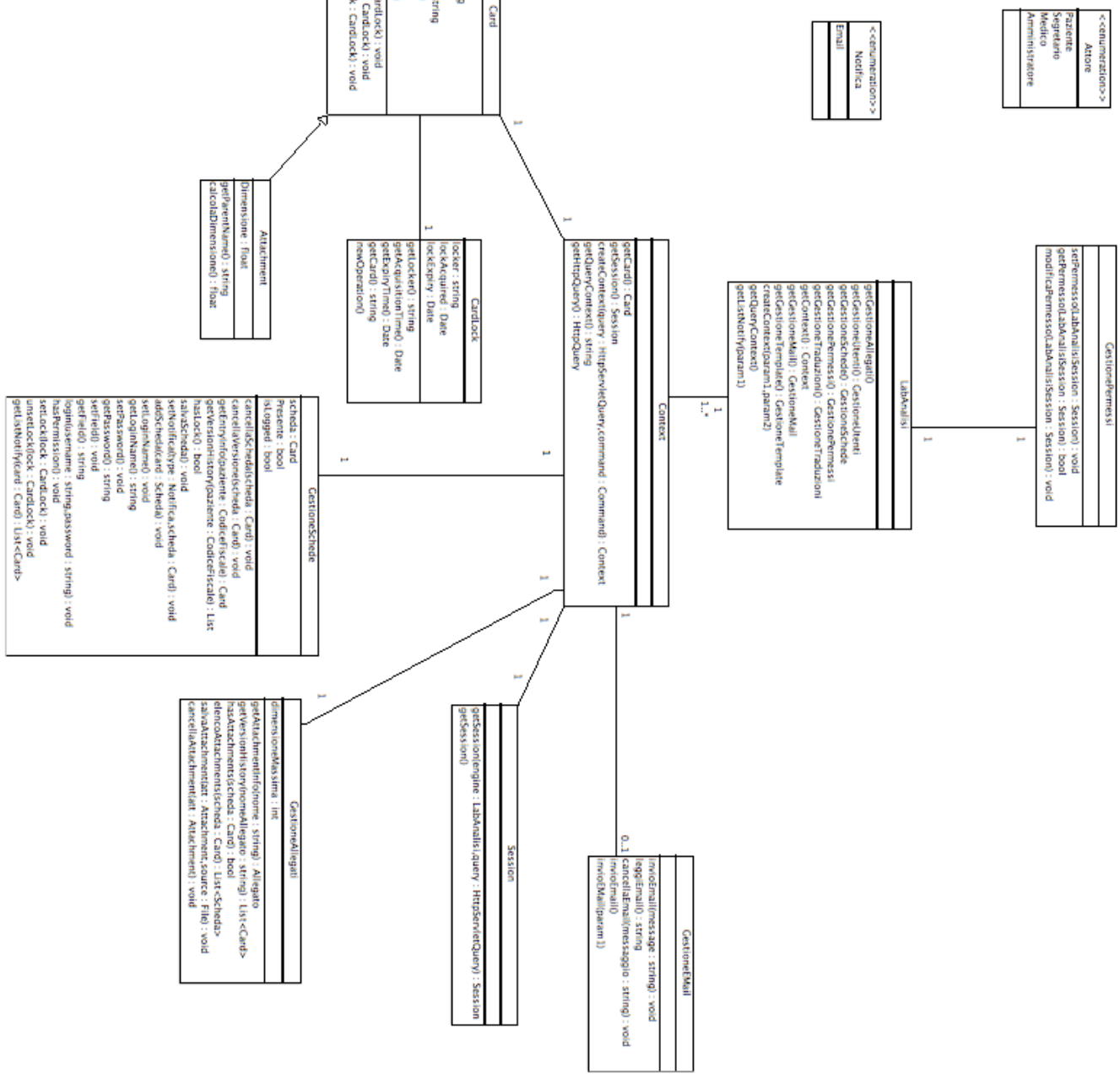


Figura 3.13: Class Diagram

3.7 Sequence diagram

3.7.1 Introduzione

Un *sequence diagram* descrive uno scenario d'uso, mettendo in evidenza la sequenza delle azioni intraprese e le relazioni che intercorrono tra le varie entità del sistema analizzato. Gli elementi fondamentali di un sequence diagram sono: le *lifeline* e i *messaggi.* Una lifeline, che viene rappresentata tramite un riquadro che riporta un identificativo dell'elemento stesso, descrive uno "scorcio di vita" di un elemento del modello. I messaggi vengono rappresentati come delle frecce che collegano la lifeline del mittente con quella del destinatario. Il contenuto di un messaggio è indicato dall'etichetta presente sulla freccia.

3.7.2 Autenticazione

Viene di seguito illustrato lo scenario di autenticazione da parte di un utente registrato; come precedentemente mostrato nel diagramma delle classi, questo scenario va "incluso" in tutti quegli scenari che richiedono l'autenticazione.

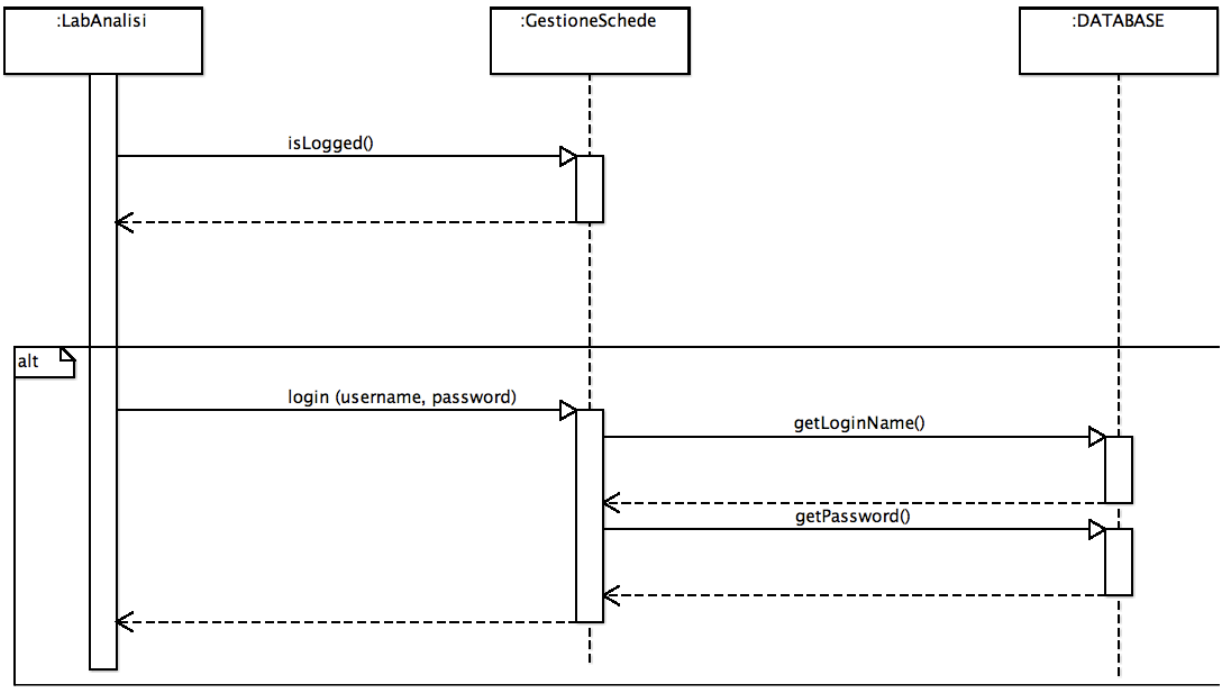


Figura 3.14: Sequence diagram per "Autenticazione"

3.7.3 Registrazione

Un utente non registrato può registrarsi accedendo alla pagina register.jsp. Come effetto di questa azione, viene creato un Context opportuno per mezzo del metodo createContext(). Successivamente viene associato un oggetto Context con la chiamata getSession() e vengono estrapolati i dati dalla richiesta HTTP effettuata dall'utente per mezzo del metodo getHttpQuery().

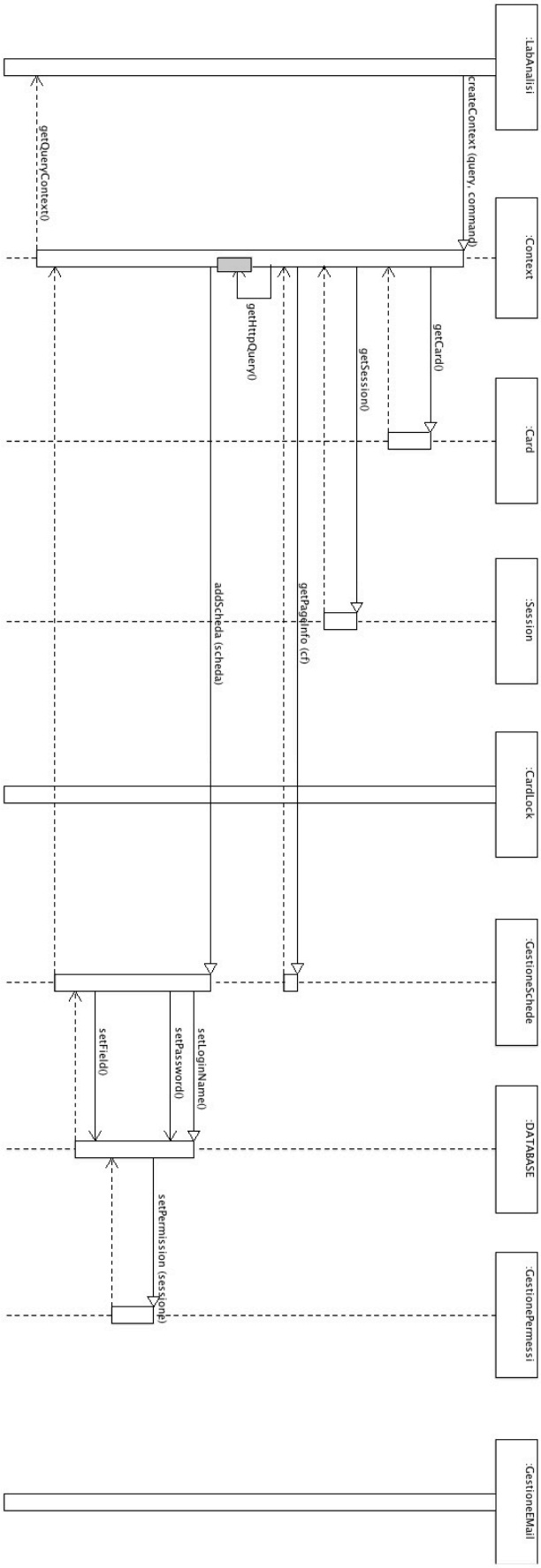


Figura 3.13: Sequence Diagram per "Registrazione"

3.7.4 De-registrazione

Durante questa fase, il sistema interagisce con il database per l'eliminazione dell'utente registrato.

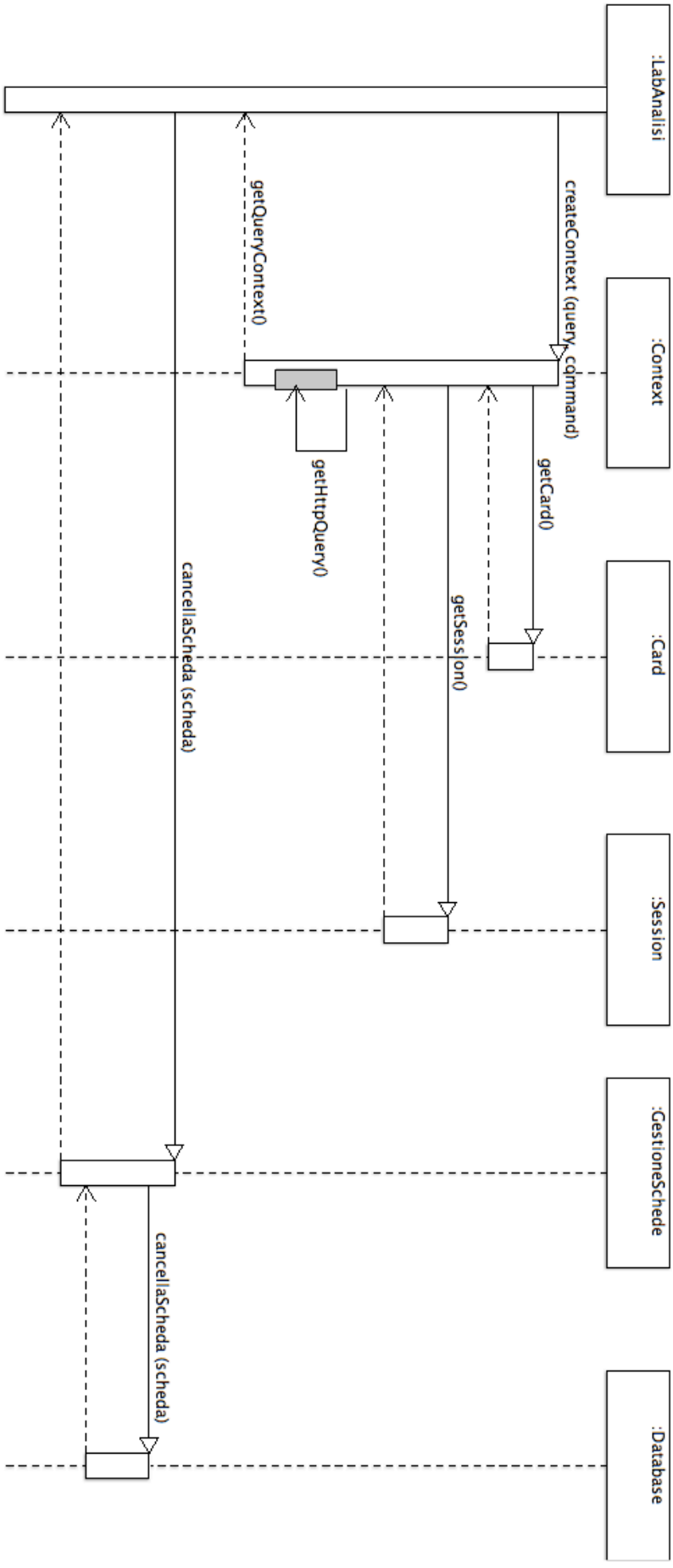


Figura 3.14: Sequence Diagram per "De-registrazione"

3.7.5 Ripristino password

In questa fase, il sistema genera una nuova password, la salva nel database degli utenti utilizzando il metodo setPassword() e, infine, spedisce una e-mail all'utente registrato per notificargli l'avvenuta creazione della nuova password. Il caso d'uso "Modifica password" è simile al caso d'uso in questione e, pertanto, viene omesso.

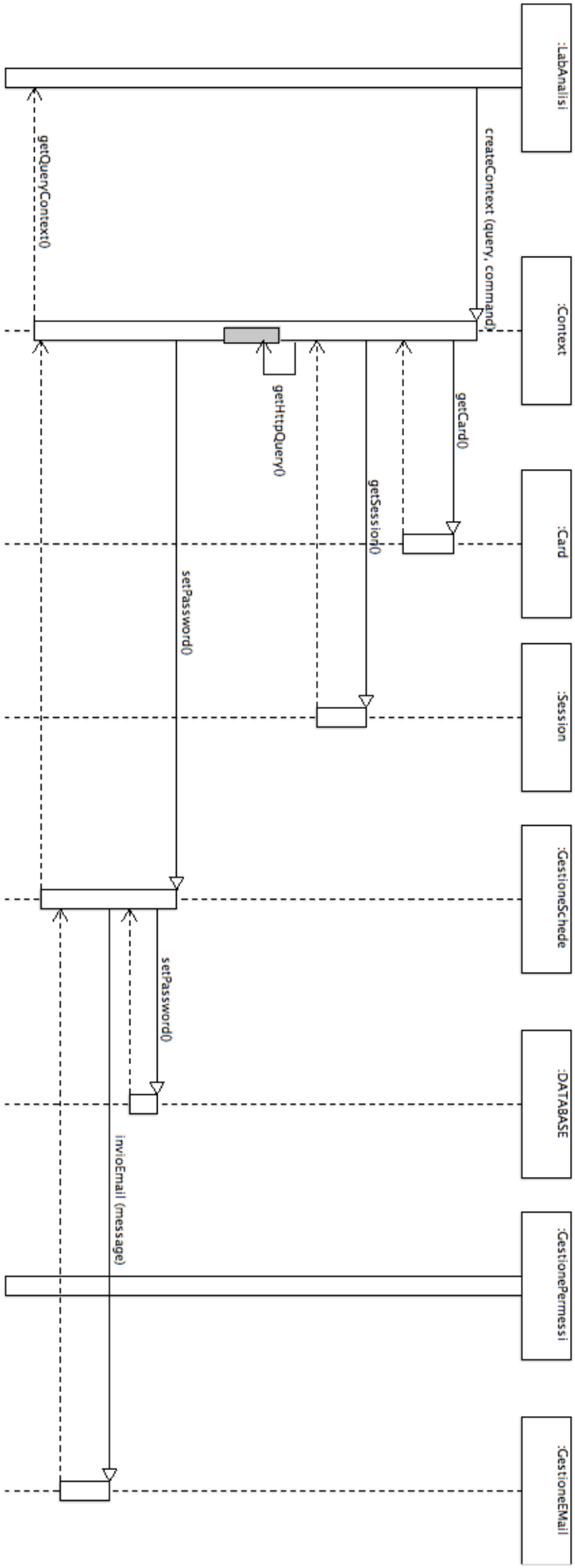


Figura 3.15: Sequence Diagram per "Ripristino password"

3.7.6 Creazione scheda

Dopo esser stata compilata con tutti i dati forniti dal paziente, la scheda viene salvata su filesystem attraverso il metodo salvaScheda().

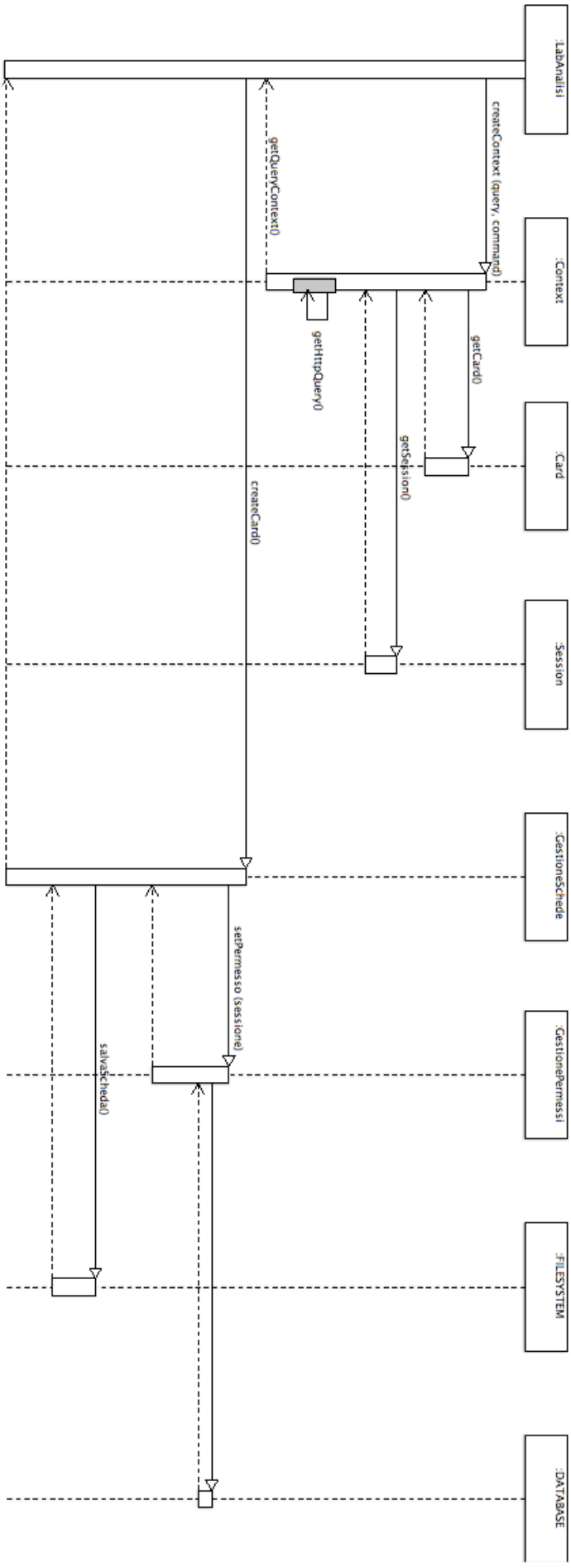


Figura 3.16: Sequence Diagram per "Creazione scheda"

3.7.7 Modifica scheda

La modifica di una scheda inizia con la chiamata della funzione modificaScheda(). Il sistema, invocando il metodo hasPermission(), verifica che l'utente sia in possesso dei permessi necessari per poter effettuare una modifica. In caso di successo, si effettua il lock della scheda attraverso un meccanismo di mutua esclusione.

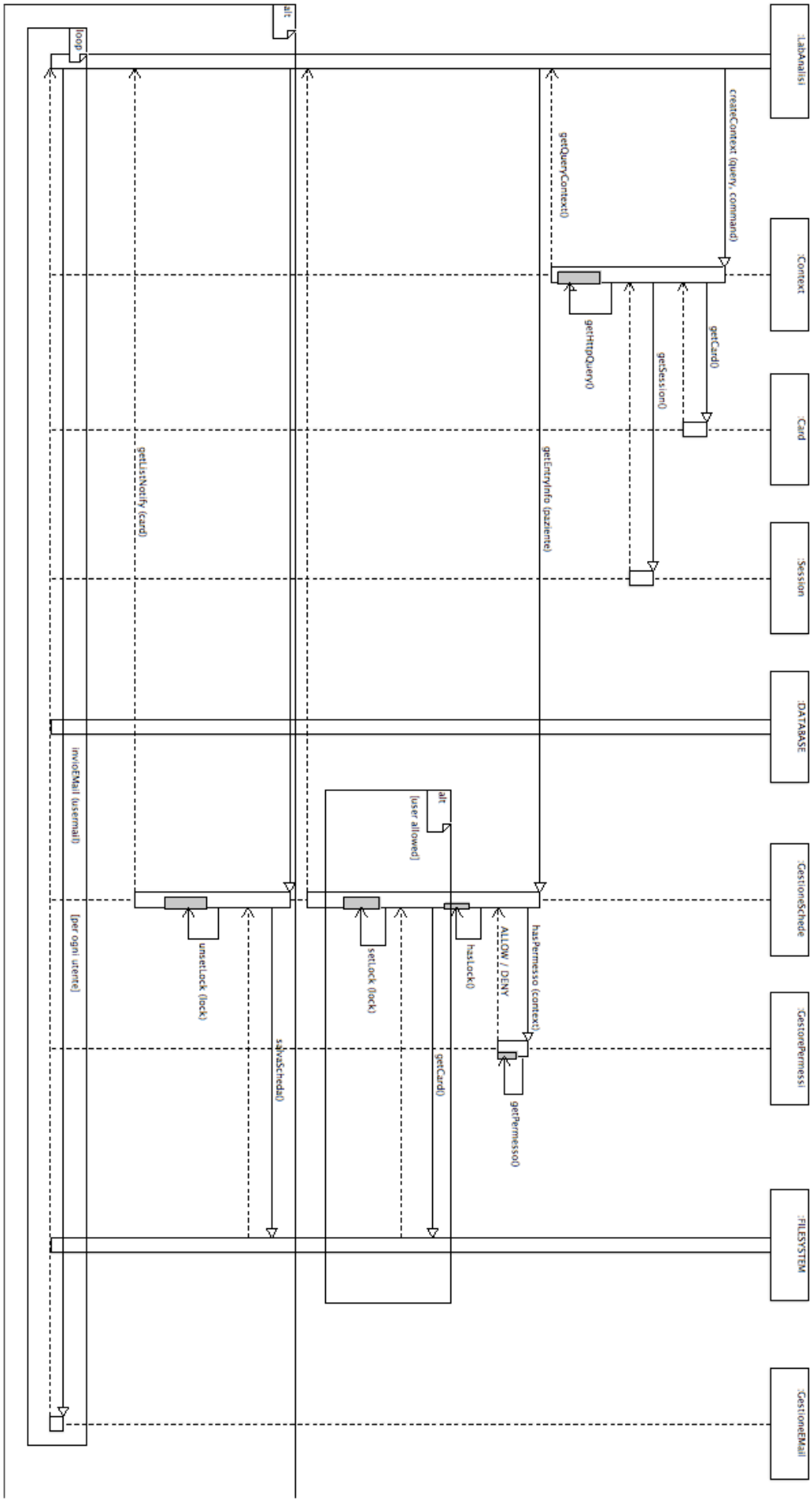


Figura 3.17: Sequence Diagram per "Modifica scheda"

3.8 Deployment diagram

Un *deployment diagram* serve a stabilire la distribuzione dei componenti di un sistema sui nodi della rete fisica, indicando per ciascun nodo le caratteristiche hardware e software e le sue connessioni con altri nodi.

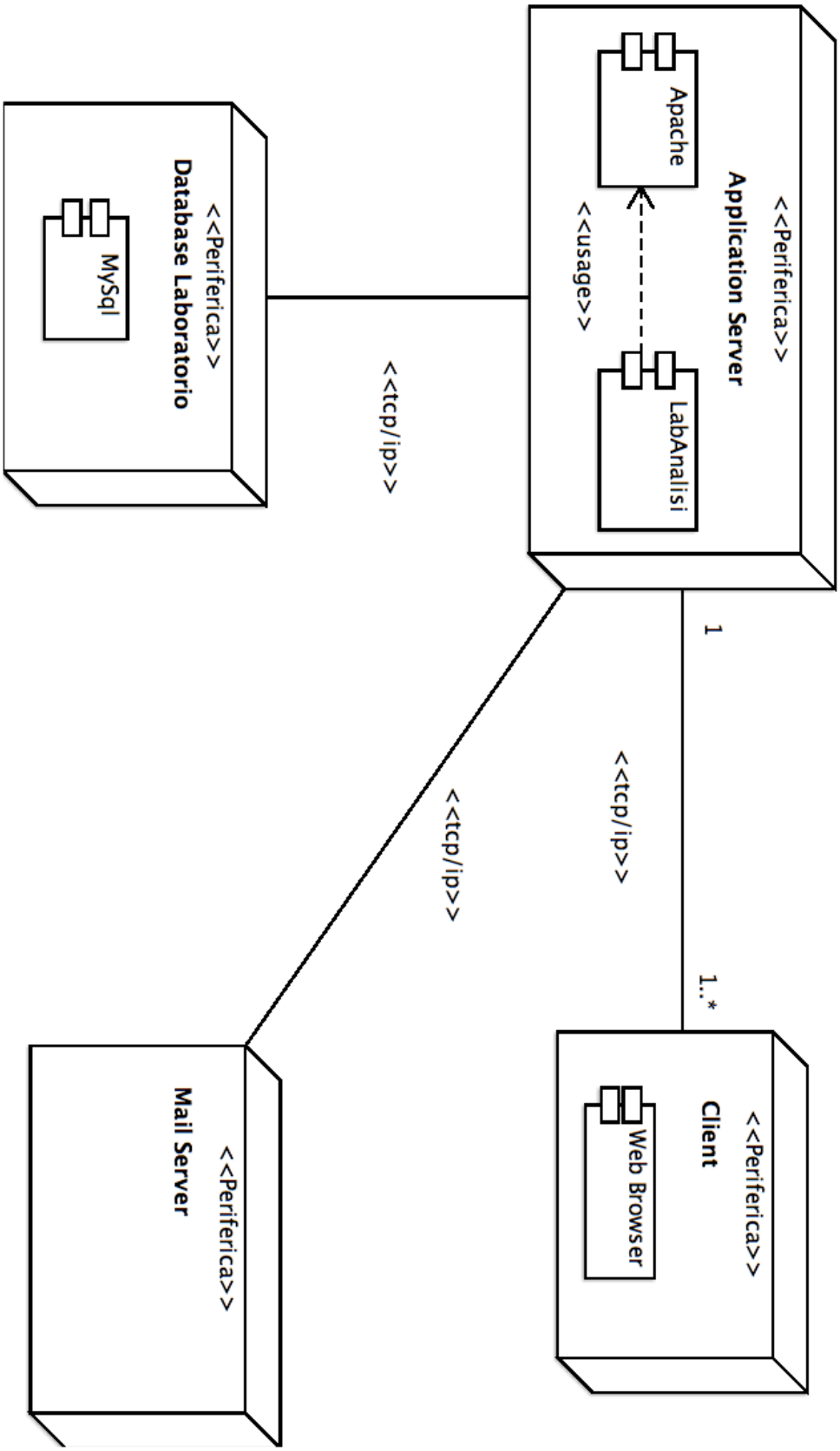


Figura 3.18: Deployment Diagram