ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA
DIPARTIMENTO DI
INGEGNERIA DELL'ENERGIA ELETTRICA E
DELL'INFORMAZIONE
"*GUGLIELMO MARCONI*"

Corso di Laurea *in* INGEGNERIA BIOMEDICA

TITOLO DELL'ELABORATO

Smart Hospital Assistant: studio ed implementazione di un assistente vocale ospedaliero tramite tecnologie semantiche

Elaborato
in
CALCOLATORI ELETTRONICI

Relatore:
Prof. Luca ROFFIA

Presentato da:
Gianluca DI TUCCIO

Correlatore:
Elisa RIFORGIATO

Anno Accademico 2020/2021

Abstract

In questo elaborato è stata proposta l'implementazione di un assistente vocale ospedaliero, denominato *Mario*, fondato su tecnologie semantiche. La sua realizzazione è stata possibile grazie alla creazione di un'ontologia e all'implementazione dell'assistente vocale basato sul linguaggio Python.
Per l'inserimento dei dati diagnostici è stato realizzato un software usato sia per la registrazione della parte anagrafica di operatori sanitari e pazienti, sia per l'inserimento o la modifica dei dati di una cartella clinica elettronica. Verranno discussi i vantaggi di queste tecnologie, tra cui l'interoperabilità tra sistemi diversi ed il poter realizzare *queries* personalizzate *ad hoc* per l'assistente vocale.

Indice

Introduzione

Spesso i software in ambito ospedaliero risultano essere molto complessi e macchinosi, ricorrendo all'utilizzo di database che presentano una struttura rigida e difficile da modificare. Una soluzione possibile è l'implementazione di una architettura a tre livelli, ma tale scelta spesso non risolve il problema in caso di continui aggiornamenti riguardanti le norme e i dati dei pazienti, soprattutto in ambito ospedaliero. In questo elaborato si propone, come soluzione, l'utilizzo di software che si appoggino ad ontologie e ad architetture in grado di farsi carico delle richieste di *queries* ed *uploads* con l'utilizzo di un tramite (*broker*), come proposto dall'architettura *SEPA*, utilizzante il linguaggio *SPARQL*.

Inoltre, grazie a queste tecnologie è possibile dimostrare la realizzazione di un assistente vocale in ambito ospedaliero. *Mario*, l'assistente vocale proposto in questo elaborato, è stato pensato anche per venire incontro alle esigenze dei pazienti che spesso cercano le informazioni sui dottori all'interno degli ospedale (come posizione e orari di visita). E' possibile, inoltre, poter implementare l'assistente vocale con un pacchetto di lingua straniera per aiutare anche le persone straniere.

Questo assistente rappresenta una possibile soluzione economica per fornire aiuto extra a pazienti, medici e segretari, seppur non possa sostituire totalmente il lavoro di una persona fisica.

In questo elaborato si è voluto realizzare un assistente vocale che non vincoli il personale medico a dover conoscere l'uso dei *form filling* e del programma ospedaliero. Per far ciò è stato realizzato un assistente che sia in grado di realizzare dei *form filling ad hoc* in base alle richieste di un medico. Ad esempio, un operatore sanitario potrà chiedere all'assistente vocale di poter cercare le cartelle cliniche di un determinato paziente, le cartelle cliniche create o modificate da un determinato medico, cercare le cartelle in base al loro *ID* o richiedere statistiche con l'uso di *ICD9-CM*, tutto in un unico

programma, un unico assistente vocale che sia in grado di fornire tali informazioni.
Mario, potrà essere implementato all'interno di un *personal computer*, all'interno di un *arduino* con microfono, schermo LCD eventualmente dotato di *touch screen resistiv*o, di una connessione ad *internet*, che rappresenta una soluzione molto più economica, o di uno smartphone. Inoltre, tale sistema potrebbe essere adattato a realtà diverse di vita quotidiana (al di fuori dell'ambito ospedaliero) o ai dispositivi *IoT*.
Infine, i file implementati durante questo elaborato sono stati caricati nella repository personale di *GitHub [1]* e liberamente accessibili.

1. Ontologie

"*L'ontologia, una delle branche fondamentali della filosofia, è lo studio dell'essere in quanto tale, nonché delle sue categorie fondamentali*" *[2]*

Il termine ontologia venne coniato soltanto agli inizi del XVII secolo da Jacob Lorhard e si imporrà definitivamente dal 1729 con il trattato Philosophia prima di Christian Wolff.

1.1 Ontologie nel mondo informatico e Web Semantico

"L'Ontologia è una esplicita specificazione di una concettualizzazione" *[3]*

Questa definizione rappresenta una delle tante definizioni associate al termine ontologia in ambito informatico, mostrando un significato diverso da quello filosofico. Altri brani esprimono l'ontologia informatica come un insieme di tecnologie per la *modellazione* della conoscenza, da cui si può ricavare il termine *Modello Ontologico*. Tale modello permette di definire le relazioni e la semantica delle entità di un dominio basata su alcuni oggetti fondamentali:

- *classi*, un insieme di oggetti
- *attributi*, rappresentano delle caratteristiche che gli oggetti (tra cui le *classi*) possono condividere. Essi sono composti da un valore e da un tipo (*literal*, *dateTime*, *integer*)
- *relazioni*, modo in cui gli oggetti possono essere relazionate tra loro;
- *individui*, rappresentano le istanze di un modello, analogamente con quanto avviene con i diagrammi ER per *SQL*.

"Ho un sogno per il web [...] in cui i computer diventino capaci di analizzare tutti i dati: contenuti, link, transazioni tra persone e macchine"
[1998, Berners-Lee] *[4]*

Le ontologie rappresentano nel mondo odierno un aspetto fondamentale nell'architettura del *Web Semantico*, termine coniato da Tim Berners-Lee, ovvero un insieme di servizi e strutture in grado di interpretare il significato dei contenuti presenti sul Web *[5]*. Tale estensione del Web associa ad ogni pagina, immagine, testo delle informazioni o metadati, fornendo così un contesto semantico che lo rendono più facile da interrogare attraverso motori di ricerca e automi.

Viene dunque introdotta l'architettura del Web Semantico, denominata Semantic Web Tower.

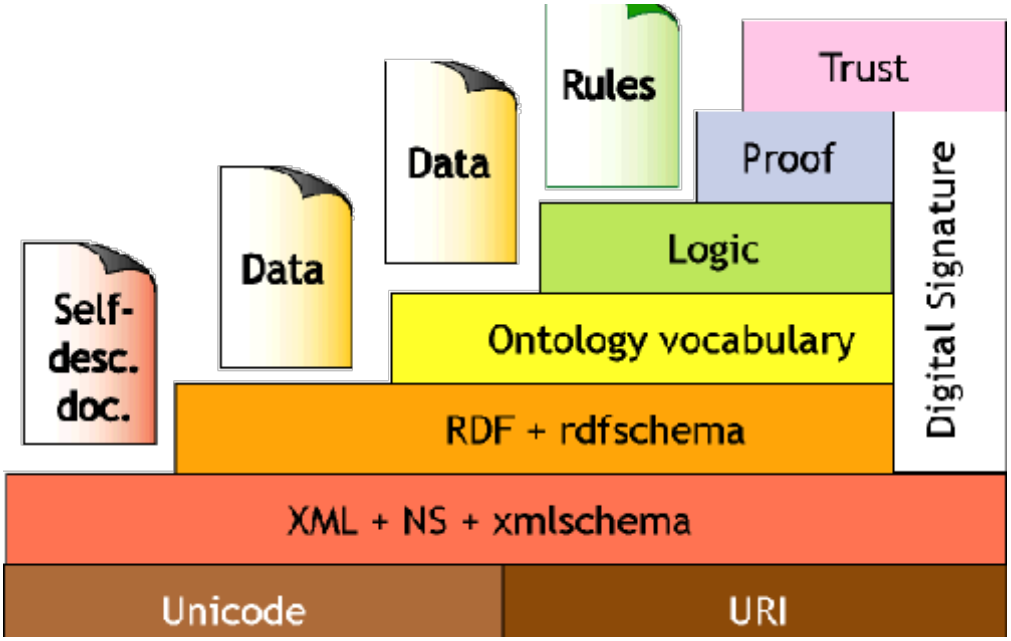


Figura 1.1: Architettura a *livelli* del *Web Semantico* (fonte: https://www.websemantico.org/articoli/approcciwebsemantico.php)

Come proposto dalla *figura 1.1*, la base è caratterizzata da *XML*[1], un linguaggio che ha come l'obiettivo quello di diventare universale all'interno della rete. Tuttavia, il *Web Semantico* si differenzia proprio da *XML* per i

[1] eXtensible Markup Language

livelli superiori, tra cui *RDF*[2]. Quest'ultimo rappresenta un linguaggio di base che permette di esprimere una informazione attraverso l'uso di triple: soggetto, predicato, oggetto, per poter così codificare informazioni sempre più complesse e provenienti da diverse parti del web.
L'*RDF* descrive delle risorse in cui ogni risorsa è identificata da un *URI*[3].
A seguire si trova *OWL*[4], gestito dal *W3C*[5], che rappresenta un linguaggio per la descrizione di modelli ontologici, attribuendo una semantica ad un modello *RDF*. Tali *OWL* possono essere gestiti tramite l'uso di editor, come *Protégé*, un editor open source organizzato in plug in (*capitolo 2.1*).
In questo elaborato verrà, dunque, proposto un assistente vocale, denominato *Mario*, implementato per il mondo ospedaliero, con l'obiettivo di congiungere i concetti di ontologia e *RDF*, tramite l'uso di *Protégé*, con *Python* e *SEPA*.

[2] Resource Description Framework

[3] Uniform Resource Identifier

[4] Web Ontology Language

[5] World Wide Web Consortium

2. Tecnologie usate e definizioni informatiche

Nei successivi paragrafi sarà approfondito il tipo di tecnologie utilizzate per la realizzazione dell'Ontologia, del programma per l'inserimento dei dati (*capitolo 3*) e lo sviluppo dell'Assistente Vocale (*capitolo 4*).

2.1 Protégé

Per la descrizione e l'implementazione dell'Ontologia è stato utilizzato *Protégé [6]*, uno strumento open source molto potente, basato su Java, che permette la realizzazione di soluzioni knowledge-based in aree distinte come la biomedicina o l'e-commerce. Questo editor è stato sviluppato per consentire sviluppi di ontologie in *OWL* (e sue successive versioni) e in *RDF*. Inoltre, a differenza di altri programmi, presenta una schermata user-friendly e permette di visionare la propria ontologia sotto forma di grafico. Un ulteriore aspetto positivo è la possibilità di installare dei plug-in, come ad esempio OntoGraf *[7]* e Debugger *[8]*. Il primo permette una visualizzazione grafica più intuitiva e precisa rispetto alla visualizzazione grafica già presente nel software, denominata OWLViz *[9]*, la quale non permette di descrive il tipo di interazione tra classi o superclassi di oggetti. Il Debugger, invece, permette di visionare se l'ontologia sia coerente e consistente; quest'ultimo plug-in può essere facilmente scaricato ed installato direttamente dall'applicazione, recandosi sulla voce "Check for plug-in" tramite la finestra "File", selezionando successivamente OntoDebug. Per quanto concerne OntoGraf e OWLViz, entrambi sono presenti all'interno di *Protégé*, potendoli selezionare attraverso la voce "Tab" della finestra denominata "Window". Tuttavia, per poter utilizzare quest'ultimi due pacchetti, è necessaria l'installazione di GraphViz *[10]*.

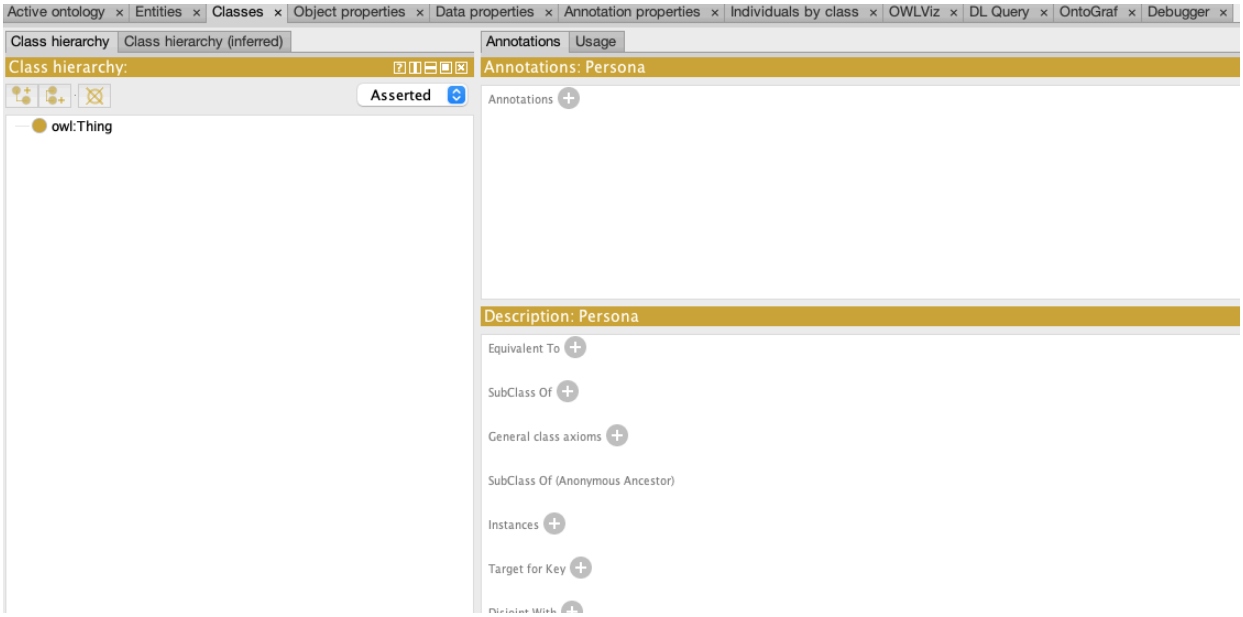


Figura 2.1: esempio della schermata *Classes* di *Protégé*

Durante la creazione *[11]* di un nuovo progetto, verrà richiesto l'inserimento del nome e dell'*URI*. L'ultimo, in particolare, è di grande importanza per la definizione dell'*URI* delle classi e del *namespaces* che serviranno per la stesura del file *JSAP* (*capitolo 2.4* e *capitolo 3.2*) necessario per la creazione dei grafi e l'inserimento dei dati. Va infatti precisato che, tramite la realizzazione dell'ontologia, è possibile istituire dei grafi con gli stessi *URI* definiti nell'ontologia; in ogni grafo, poi, verranno inseriti i relativi dati. Un esempio potrebbe essere dato dalla creazione di un *URI* del progetto denominato <http://semanticweb.org/esempio> e successivamente definire le varie classi tramite l'*URI* appena descritto. Un ulteriore vantaggio di *Protégé*, è l'autogenerazione dell'*URI* durante l'inserimento delle classi. Infatti, nel caso si volesse inserire la classe *Person*, il programma mostrerà l'auto *URI* denominato <http://semanticweb.org/esempio#Person>.
Come descritto nel *capitolo 1*, il software permette l'inserimento di più sottoclassi e l'aggiunta di Data Properties e Object Properties. Le prime rappresentano degli attributi[6] delle classi, come ad esempio l'attributo nome, mentre i secondi rappresento un "predicato", ovvero in grado di collegare un

[6] intrinseci ed estrinseci

soggetto ad un oggetto. Tramite *Protégé*, inoltre, è possibile definire alcuni vincoli e assiomi per queste ultime categorie *[12]*. Tra le varie proprietà spiccano soprattutto *Symmetric* (se un'istanza X è correlata alla istanza Y, allora anche Y è correlata ad X) e *Transitive* (se un'istanza di X è correlata a Y e a sua volta quest'ultima è correlata a Z, allora X è correlato a Z). In *Protégé*, la realizzazione delle Data Properties avviene all'interno della relativa sezione. Dopo aver definito il nome della proprietà è necessario specificare il dominio ed il range. Il primo determina la classe a cui fa riferimento, mentre il secondo determina il *tipo* dell'attributo (come ad esempio *rdfs:Literal*[7]). Analogo procedimento per l'inserimento di un Object Property, in cui il dominio sarà sempre dato dalla classe che rappresenta il soggetto della relazione (soggetto - predicato - oggetto), mentre il range sarà costituito dall'oggetto della relazione.
Ulteriori considerazioni importanti sono la definizione dell'*inverse of* di un Object Properties, il quale permette di definire le relazioni biunivoche tra due classi, e *disjoint with*, ove cui è possibile definire la "separazione totale" di una classe con un'altra classe, come ad esempio può avvenire tra Persona e Reparto[8], le quali non presentano alcuna informazione[9] in comune.
In conclusione, *Protégé* permette l'inserimento delle istanze, denominate *individuals*. Analogamente a quanto avveniva con i diagrammi ER, è possibile inserire delle istanze che rappresentano le singole informazioni di una classe. Un esempio è rappresentato dalla classe Persona e dal suo attributo *fiscalCode* (codice fiscale). Le istanze della classe Persona sono rappresentate dalle persone stesse, come ad esempio *"Mario_Rossi"* e *"Gianluca_Rossi"* che rappresentano due istanze di Persona. Ad ogni istanza è possibile associare un valore specifico per ogni attributo della classe ("*MRORSS…*" è il valore di *fiscalCode* per la Persona "*Mario Rossi*"). Un ulteriore esempio è mostrato dalla classe Reparto (*capitolo 3.1*) in cui ad ogni istanza corrisponde un nome del reparto, come *"CARDIOLOGIA" e "OCULISTICA*".

---

[7] *Protégé* offre anche la possibilità di creare un proprio tipo, non presente nei valori di default

[8] rappresentano delle classi di una ontologia in ambito ospedaliero

[9] nessuna proprietà ed istanza in comune

2.2 SPARQL

SPARQL è un linguaggio legato al paradigma del *Web Semantico* e fu sviluppato nel 2008 da parte di *W3C*. Il suo principale utilizzo è per l'interrogazione di *RDF*, basato sul concetto di triple: soggetto, predicato ed oggetto. Un esempio di tripla potrebbe essere data da *Persona* (soggetto) *è_nata* (predicato) in una *città* (oggetto). Inoltre, dalla conoscenza del soggetto e del predicato, è possibile risalire alla conoscenza dell'oggetto. In ogni modo, il soggetto potrebbe essere composto, per esempio, da un *URI* o da un *blank node*. Solitamente, nelle ontologie, un soggetto viene indicato con il termine "classe", come mostrato dalla classe *Person* presente all'interno di *Protégé*.
Tale linguaggio presenta molte similitudini con *SQL*. Tuttavia, verrà trattata solo la sintassi necessaria per lo sviluppo di una intelligenza artificiale basata su triple ed ontologie.
Le triple possono essere espresse in diverse modalità, come ad esempio *turtle [13]* o *RDF/XML [14]*.
In *turtle* le triple vengono rappresentate secondo la seguente dicitura: *soggetto - nome_namespace:predicato - oggetto*, in cui il nome del *namespace* può essere definito all'interno dell'ontologia, come descritto nel *capitolo 2.1*. Gli oggetti e soggetti potrebbero essere identificati anche tramite *URI*. In questo caso, l'*URI* può essere semplificato proprio con l'utilizzo dei *namespaces*. Se ad esempio il soggetto *Patient* è rappresentato da <http://semanticweb.org/esempio#Patient>, esso potrà essere riscritto come es:Patient, definendo precedentemente il namespace come "es" : <http://semanticweb.org/

```
@prefix dc: <http://purl.org/dc/terms/> .
@prefix frbr: <http://purl.org/vocab/frbr/core#> .

<http://books.example.com/works/45U8QJGZSQKDH8N> a frbr:Work ;
     dc:creator "Wil Wheaton"@en ;
     dc:title "Just a Geek"@en ;
     frbr:realization <http://books.example.com/products/9780596007683.BOOK>,
         <http://books.example.com/products/9780596802189.EBOOK> .

<http://books.example.com/products/9780596007683.BOOK> a frbr:Expression ;
     dc:type <http://books.example.com/product-types/BOOK> .

<http://books.example.com/products/9780596802189.EBOOK> a frbr:Expression ;
     dc:type <http://books.example.com/product-types/EBOOK> .
```

Figura 2.2: esempio di utilizzo di prefissi (o *namespace* nel caso di *JSAP*) e relative triple descritte in *turtle* (fonte: https://www.w3.org/TR/turtle/)

esempio#>. Anche grazie al file *JSAP* (*capitolo 2.4*) sarà possibile l'utilizzo dei *namespaces*. L'oggetto, invece, potrebbe essere anche rappresentato da un letterale, indicato con l'utilizzo dei doppi apici.
Come mostrato dalla *figura 2.2*, è importante conoscere anche ulteriori parole chiave della sintassi:

- "***a***" , corrisponde ad una semplificazione di *rdf:type*, in cui *rdf* corrisponde al prefisso http://www.w3.org/1999/02/22-rdf-syntax-ns#;
- "***.***" **,** utilizzato per separare le diverse triple (talvolta viene omesso nell'ultima tripla);
- "***;***" **,** permette di omettere il soggetto se il soggetto è lo stesso della tripla precedente (esempio: *?user rdf:type schema:Person . ?user schema:name ?userName --> ?user rdf:type schema:Person ; schema:name ?userName*);
- "***_:***" , è utilizzato per i *blank node*;
- "***?***" , viene posto prima di una stringa per indicare che quest'ultima sia una variabile.

Passando a *SPARQL*, esso permette l'interrogazione delle triple utilizzate nelle ontologie. Analogamente a *SQL*, una query *SPARQL* presenta:
- ***SELECT***, per selezionare cosa bisogna visualizzare;
- ***WHERE***, dove si trovano tali valori da visualizzare (inoltre, è incluso anche la sintassi ***GRAPH***, per indicare in quale grafo è necessario cercare i valori).
Quanto visto si applica in modo analogo anche per ***INSERT***, ***INSERT DATA*** e ***DELETE***, come proposto dalle *figure 2.4* e *figura 2.5*.

```
SELECT ?diagn (count(?diagn) as ?last)
WHERE {GRAPH <http://unibo.it/ontology/SmartHospitalAssistant/Ward_Section>
{?g rdf:type sha:Ward_Section ; sha:diagn ?diagn }} GROUP BY ?diagn
```

Figura 2.3: esempio di utilizzo di *SPARQL*; vengono visualizzati i nomi delle diagnosi (rappresentato da *?diagn*) insieme a quante volte si presenta ogni diagnosi (count(*?diagn*), in particolar modo permette di contare quante volte appare una determinata diagnosi) ed infine un ***GROUP BY*** per raggruppare i risultati in base al loro nome

```
INSERT DATA
{graph <http://unibo.it/ontology/SmartHospitalAssistant/Patient>
{_:XYZ rdf:type schema:Patient ; sha:fiscalCode ?fiscalCode}}
```

Figura 2.4: inserimento di un nuovo Paziente e del suo codice fiscale nel grafo dei Pazienti attraverso l'uso del *blank node*. Tuttavia, è necessario specificare anche quale valore deve essere inserito, attraverso l'uso del *force binding* (verrà approfondito nel *capitolo 2.4* e *capitolo 3.2*)

```
INSERT {graph <http://unibo.it/ontology/SmartHospitalAssistant/Patient>
{?h rdf:type schema:Patient ; sha:givenName ?givenName ; sha:fiscalCode ?fiscalCode}}
WHERE {graph <http://unibo.it/ontology/SmartHospitalAssistant/Patient>
{?h sha:fiscalCode ?fiscalCode ; rdf:type schema:Patient}}
```

Figura 2.5: inserimento del nome di una Persona in base al suo codice fiscale

2.3 SEPA (broker)

Il *SEPA* (SPARQL Event Processing Architecture) *[18]* rappresenta una architettura *publish-subscribe* designata per l'interoperabilità tra sistemi[10], supportata anche dal protocollo *TLS* (Transport Layer Security) del livello quattro del modello ISO/OSI per garantire una maggior sicurezza. Tale architettura è basata su dei meccanismi di rilevamento e di notifica delle modifiche dei dati ed è stata progettata per *endpoint*[11] *SPARQL* (e dunque

[10] *sistema*: qualsiasi dispositivo che sia software, applicazione, sensore, etc, ma sempre atti al concetto di *queries* ed *updates* utilizzando il protocollo SPARQL 1.1

[11] con il termine endpoint ci si riferisce a qualsiasi dispositivo in grado di connettersi ad internet

conforme al protocollo *SPARQL* 1.1[12]). Inoltre, l'architettura introduce il protocollo *SPARQL* 1.1 *Secure Event*[13] *[15]* e *SPARQL* 1.1 *Subscribe Language*[14] *[16]* .

L'architettura permette la semplificazione delle queries ed updates dei dati grazie all'utilizzo di un broker, un oggetto in grado di prendere in carico tali richieste *[17] [18] [19]*.
Si possono individuare alcuni elementi principali di questa architettura, come:
- produttore, un produttore di triple da inviare tramite update al *SEPA*[15]. Un esempio di produttore potrebbe risiedere in un sensore termico che, una volta effettuato la misura, formatta ed inoltra l'informazione al *SEPA*;
- consumatore, ovvero colui che si occupa solo della raccolta dei dati provenienti dal SEPA. Può effettuare una sottoscrizione ai dati di suo interesse per essere sempre aggiornato;
- aggregatore, colui che, ricevuta una notifica di un nuovo dato, è in grado di eseguire una update.
Nel *capitolo 2.4* viene trattato un file di configurazione denominato JSAP per descrivere un'applicazione che utilizza il SEPA come architettura e che, in concomitanza con la Dashboard, costituisce uno strumento per agevolare l'implementazione del modello da scrivere.

[12] rappresenta il protocollo definito per il linguaggio SPARQL

[13] un protocollo che permette comunicazioni protette

[14] un protocollo per trasmettere ed esprimere le richieste e notifiche di sottoscrizioni *[19]*

[15] e dunque non rappresenta un oggetto in grado interrogare una struttura

2.4 JSAP

JSAP [20] (JSON SPARQL Application Profile) è uno strumento che permette la descrizione di un'applicazione basata sull'architettura del *SEPA*, utilizzando il formato JSON *[21]*. Il suo ruolo è quello di accumulare gli *updates* e le *queries*, definire alcuni parametri usati dal *SEPA* e permette di contenere dei prefissi. Per quanto riguarda la sua struttura, vi è:
- una parte introduttiva riguardante alcuni parametri di connessione, i *namespaces* e il grafo di default;
- una parte riservata alle *uptades*, facenti riferimento all'inserimento o eliminazione dei dati;
- una ultima parte denominata *queries* nella quale è possibile interrogare il database.

Per quanto concerne i parametri, i termini *host*, *sparql11protocol* e *sparql11seprotocol* dovranno sempre essere presenti all'interno del file *JSAP*, e saranno approfonditi ulteriormente nel *capitolo 3.2*.
La struttura delle *queries* e updates permette di effettuare delle ricerche o inserimenti all'interno dei grafi utilizzando proprio *SPARQL*. Dopo aver specificato il nome della query, è possibile definire il tipo di linguaggio che sarà usato. In questo caso "*sparql*" seguito successivamente dal carattere "*:*" che permette di scrivere la stringa della query o update in formato *SPARQL*.
Un ulteriore aspetto è rappresentato dal *force binding*, uno strumento in grado di specificare quali valori di default possano essere assunti dalle variabili. Per esempio, specificando il *type* (es. "*literal*") e *value* (es. "cardiologia") della variabile reparto, è possibile inserire questi valori all'interno dei grafi. Infine, nel caso in cui non si utilizzasse un grafo di default e gli inserimenti avvengano in grafi diversi, è importante specificare tramite i tag (<...>) in quale grafo saranno inseriti o letti i valori.
Come descritto nel *[22]*, dopo aver installato ed avviato *blazegraph* e l'*engine*[16] del *SEPA broker*, è possibile caricare il file JSAP attraverso l'avvio della Dashboard, esattamente come spiegato nel paragrafo su GitHub *[22]*

[16] entrambi rappresentano degli strumenti presenti all'interno del *SEPA Bins [22]*; il primo rappresenta un database a grafo necessario per la realizzazione dei grafi, mentre l'*engine* rappresenta il *SEPA*

```
{
  "host":"...",
  "oauth":{},
  "sparql11protocol":{},
  "sparql11seprotocol":{},
  "namespaces":{},
  "extended":{},
  "graphs":{
    "default-graph-uri":"...",
    "named-graph-uri":"...",
    "using-graph-uri":"...",
    "using-named-graph-uri":"..."
  },
  "updates":{
    "UPDATE_IDENTIFIER_1":{},
    "...":{},
    "UPDATE_IDENTIFIER_N":{}
  },
  "queries":{
    "QUERY_IDENTIFIER_1":{},
    "...":{},
    "QUERY_IDENTIFIER_N":{}
  }
}
```

Figura 2.6: esempio di struttura del *JSAP*, con la suddivisioni tra parametri, *updates* e *queries* (fonte: http://mml.arces.unibo.it/TR/jsap.html)

```
'REMOVE":{
  "sparql":"DELETE {?message ?p ?o} WHERE {?message rdf:type schema:Message ; ?p ?o}'
  "forcedBindings":{
    "message":{
      "type":"uri",
      "value":"chat:ThisIsAMessage"
    }
  }
},
```

Figura 2.7: esempio di updates all'interno del file *chat.jsap*. Viene mostrato il nome dell'updates (***REMOVE***), il tipo di linguaggio ("*sparql*") e la sua relativa stringa (in questo caso un ***DELETE***). Successivamente viene mostrato un esempio di utilizzo di *forceBindings* con il nome della variabile ("*message*", la stessa usata nella stringa di *SPARQL*), il tipo ("*uri*") ed infine il suo valore ("*chat:ThisIsAMessage*") (fonte: https://github.com/arces-wot/SEPABins)

denominato "*Play with SEPA*". Quest'ultimo rappresenta un ottimo strumento per testare il file *JSAP* e per verificarne l'integrità. Inoltre, è presente, sempre su *GitHub [22]*, un file di prova denominato *chat.jsap* che permette di comprendere l'utilizzo del JSAP e dei suoi parametri.
Come anticipato precedentemente, nel *capitolo 3.2* verrà proposto il file *JSAP* utilizzato per questo elaborato.

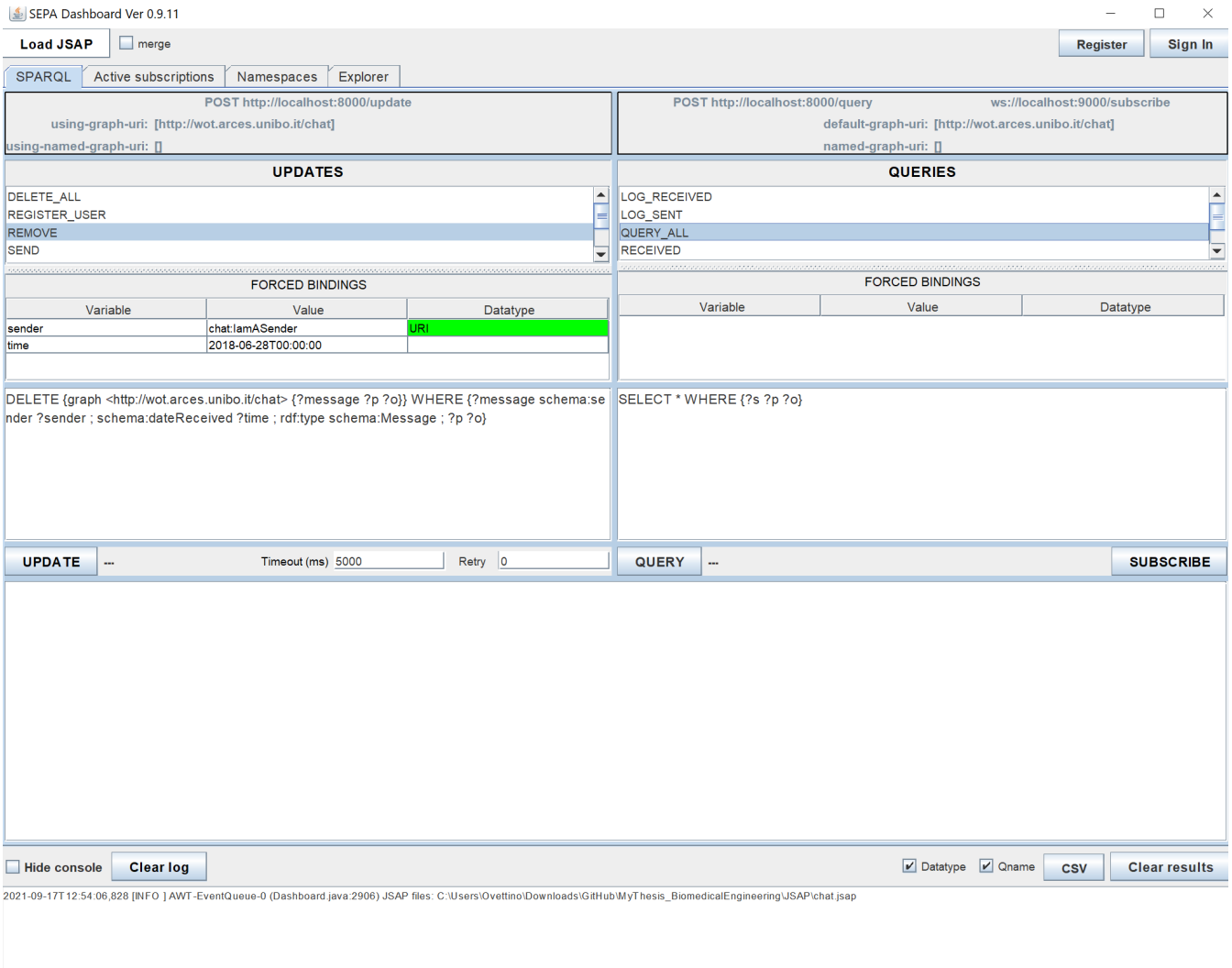


Figura 2.8: schermata dello strumento Dashboard dopo il caricamento del file *chat.jsap*. Come si può notare nella sezione *UPDATES*, è possibile selezionare il tipo di updates (in questo caso ***REMOVE***) e poter visionare il codice *SPARQL* per la cancellazione del dato e della sezione *FORCE BINDING*, la quale permette di modificare i valori di default che sono stati descritti nella *figura 2.7*

2.5 Python

Per l'elaborato proposto è stato scelto l'utilizzo di *Python [23]*. Rappresenta un linguaggio all'avanguardia, molto più semplice e versatile rispetto ad altri linguaggi in circolazione. Venne creato nel 1991 da Guido Van Rossum ed è in uso attualmente per oltre il 50% tra tutti i linguaggi globali secondo gli ultimi dati del 2019 *[24]*. *Python* permette il caricamento e la creazione di svariati pacchetti denominati *packages*, utilizzati per semplificare il lavoro dei programmatori. Questi pacchetti possono essere facilmente scaricati dal Web o direttamente dalla console (o terminale su *macOS*), utilizzando la stringa "python -m pip install" seguito dal nome del pacchetto. Tale stringa deve essere inserita nello stesso percorso (*path*) in cui è stato installato *Python*. Tuttavia, si consiglia l'utilizzo della versione 3.9.4 di *Python* e 21.2.4 per *pip*, per evitare alcuni errori e *bug* presenti nelle versioni successive per macchine virtuali o per *macOS* (inoltre, il numero della versione può essere visionato digitando "*python --version*" e "*pip --version*"). Tramite il link personale di *GitHub [1]* è presente un PDF per l'installazione di una macchina virtuale e di *Python*.
Per poter usufruire del linguaggio *Python* è stato utilizzato *Atom [25]*, un programma gratuito che permette lo sviluppo di applicazioni in svariati linguaggi ed è in grado di accoppiarsi perfettamente con *GitHub*.
In questo elaborato sono stati utilizzati diversi *packages*, come:

- *SpeechRecognition [26]*, necessario per la traduzione e la comprensione di una conversazione orale;
- PlaySound *[27]*, per la riproduzione di file sonori (nel caso in esame, tale pacchetto è stato utilizzato per riprodurre proposizioni); tuttavia, si consiglia l'utilizzo della versione 1.2.2 per problemi legati al microfono sulle macchine virtuali, utilizzando la stringa *playsound==1.2.2;*
- codicefiscale *[28]*, per la costruzione di un codice fiscale;
- tkinter *[29]*, un pacchetto che permette la realizzazione di *form filling* e applicazioni;
- gTTS *[30]*, per l'utilizzo di google translate;
- icd9cms *[31]*, per l'utilizzo dello Standard *ICD9-CM*, utilizzato per la codifica delle diagnosi mediche;
- wikipedia *[32]*, per l'utilizzo di wikipedia attraverso *Python*.

3. Implementazione dell'ontologia e del Software per l'inserimento dei dati

In questo capitolo verrà esposto l'implementazione dell'ontologia e del software per l'inserimento dei dati in ambito sanitario, propedeutico per la realizzazione dell'Assistente Vocale "*Mario*", un assistente vocale per venire incontro alle esigenze degli utenti. Come anticipato dal *capitolo 2.1* e *capitolo 2.5*, per la realizzazione dell' ontologia verrà utilizzato *Protégé*, mentre per il software verrà utilizzato il linguaggio *Python* con l'ausilio dei *form filling [33]*. Il software per gli inserimenti dei dati ospedalieri non rispecchia esattamente la realtà di come vengono gestiti i dati dei pazienti e delle cartelle cliniche, tuttavia, permette di mostrare la facilità e l'interoperabilità del software con l'utilizzo di grafi e dell'ontologia[17]. Una delle principali difficoltà che si presenta nell'implementazione di software ospedalieri è l'interoperabilità dei vari programmi con il database. Spesso accade che un programma, realizzato in un secondo momento, non sia in grado di compiere degli inserimenti o query verso un database (nel caso di architettura a due livelli[18]).

Questo problema potrebbe costringere l'operatore sanitario a dover immettere più volte i dati all'interno dei diversi programmi. Un software che si appoggia a strutture interoperabili è in grado di superare questi limiti. Infatti, un database ha una struttura ben definita e rigida e non permette di variare in che modo i dati possono essere inseriti nel database. Come ad

---

[17] seguendo sempre le linee guida della programmazione e dei *form filling*

[18] rappresenta una struttura fondata su *Viste Utent*i e *Server + Database* realizzati in un unico livello. Tuttavia, il problema si può espandere anche nell'utilizzo di una architettura a tre livelli, poiché una modifica del database implicherebbe comunque una modifica agli altri due livelli *[34]*

esempio nel caso di inserimento dei dati di un paziente ignoto o l'inserimento di nuove informazioni. Nel primo caso è necessario implementare nel software un'opzione per cui un operatore sanitario può non inserire tutti i dati nel programma e far sì che il database possa accettare anche valori *Null*, che poi saranno modificati attraverso un *update*. Il secondo tema, spesso, non è realizzabile, poiché costringerebbe il personale a modificare in primo luogo la struttura del database[19] che in cascata recherebbe ulteriori problematiche ai restanti livelli dell'architettura. Tali temi possono essere facilmente risolti con l'utilizzo delle ontologie e dei relativi grafi, i quali permettono l'inserimento delle informazioni (anche in un secondo momento) utilizzando una struttura non rigida.

3.1 Implementazione dell'Ontologia

Per la realizzazione dell'Assistente Vocale è stato necessario come primo step implementare un'ontologia che potesse descrivere in buona parte il mondo ospedaliero con le sue figure professionali, analogamente a come avviene con diagrammi ER per la realizzazione del database *SQL*.
Nel mondo ospedaliero spiccano tantissime figure, tra cui gli operatori sanitari e i pazienti. Ed è proprio qui che inizia la creazione dell' ontologia. Nel caso in esame, si può affermare che tutte queste figure possono essere racchiuse all' interno di un'unica classe (un unico "gruppo") denominata *Person* (Persona). Dunque, una persona può a sua volta suddividersi in *Patient* (Paziente), *Health Worker* (Operatore Sanitario) o *Other* (per esempio tecnico, magazziniere, ecc). In linea generale una classe può essere descritta da ulteriori sottoclassi (così avere una maggior specificità con l'evento che si vuole descrivere) e può essere correlata ad un'ulteriore

[19] il database rappresenta il terzo livello in una architettura a tre livelli

classe, non per forza della stessa famiglia, come ad esempio il collegamento tra la classe Patient ed una classe Cartella Clinica, che non

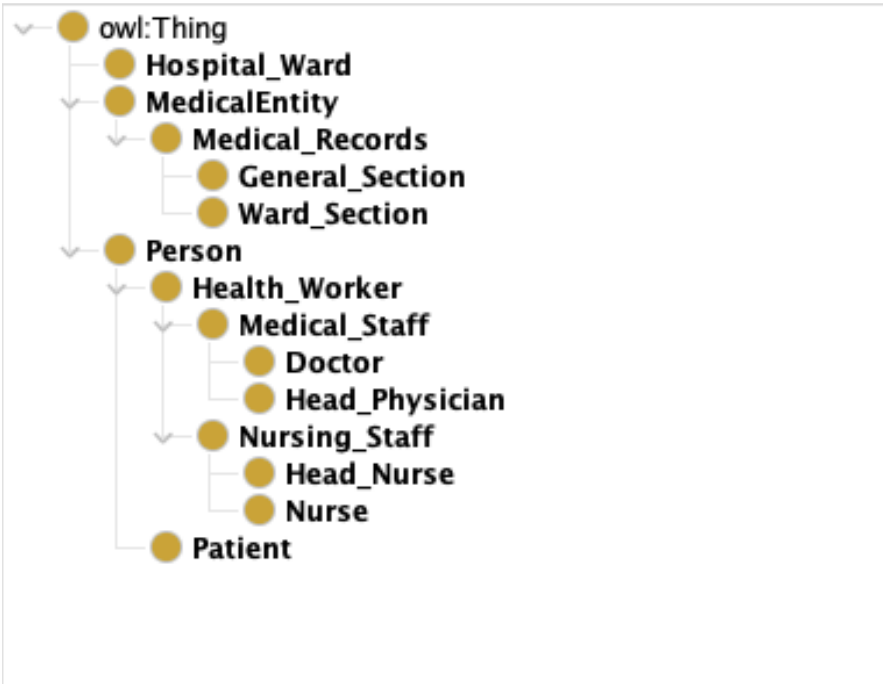


Figura 3.1: ontologia di questo elaborato, con la descrizione dei soli operatori sanitari, pazienti, reparti e cartella clinica

rientra nella famiglia delle *Person*. Sulla falsa riga di queste definizioni, è possibile suddividere la classe *Health Worker* in ulteriori sottoclassi, come *Medical Staff* (Medici e Primari) e *Nursing Staff* (Infermieri e Caposala). A loro volta possono essere suddivise rispettivamente in *Doctor* (Medico) e *Head Physician* (Primario), e *Nurse* (infermiere) e *Head Nurse* (Caposala). Come si può notare, tali classi sono state suddivise fino a raggiungere il maggior dettaglio possibile restando in linea con il progetto dell'elaborato. Ulteriori suddivisioni potrebbero essere effettuate anche per Patient e Other come proposto dalla *figura 3.2*.

Oltre alle figure professionali, ai fini del progetto è stato necessario descrivere gli "oggetti". Esempi sono rappresentati dal reparto e dalla cartella clinica, rispettivamente *Health Ward* e *Electronic Medical Record*.

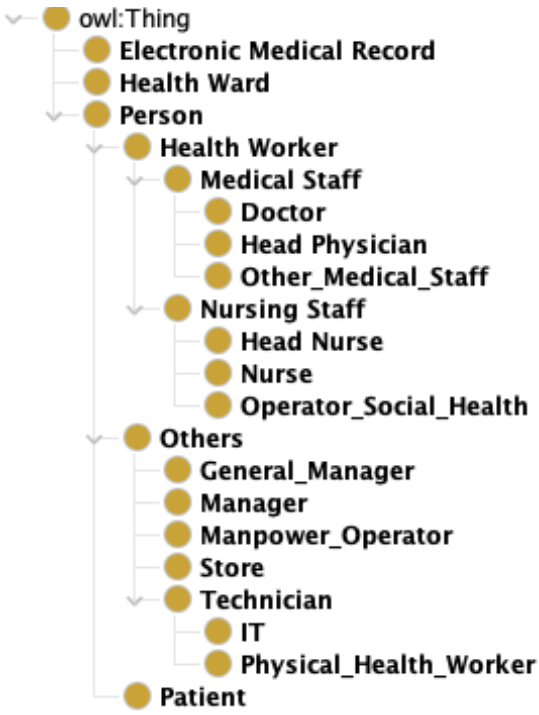


Figura 3.2: esempio di ontologia per una migliore rappresentazione delle figure presenti all'interno del mondo ospedaliero

Queste classi, e le loro relative sottoclassi, sono state poste in "parallelo"[20] alla classe Person. Tutte le classi fanno riferimento ad un'unica classe globale, *Thing*.

Gli utenti possono anche realizzare un'ontologia basandosi sulle classi fornite da schema.org *[35]*, una comunità che ha come scopo quello di realizzare uno schema prestabilito fondato su classi fondamentali[21] per strutture utilizzate per Internet, e-mail, pagine web ed in altre svariate strutture. Uno dei principali vantaggi dell'utilizzo delle ontologie, è quello di poter condividere la propria ontologia e poter modificare liberamente l'ontologia realizzata da altri utenti secondo le proprie esigenze. Un chiaro esempio è l'ontologia ***SNOMED*** *[36]*, un dizionario utilizzato per la

[20] con il termine parallelo si vuole intendere che le classi Person, Health Ward ed Electronic Medical Record sono tra di loro poste allo stesso livello

[21] classi fondamentali: Persona, Cosa, Evento Sportivo, Evento Medico, etc. La quasi totalità delle attività può essere descritta attraverso semplici classi definite da schema.org

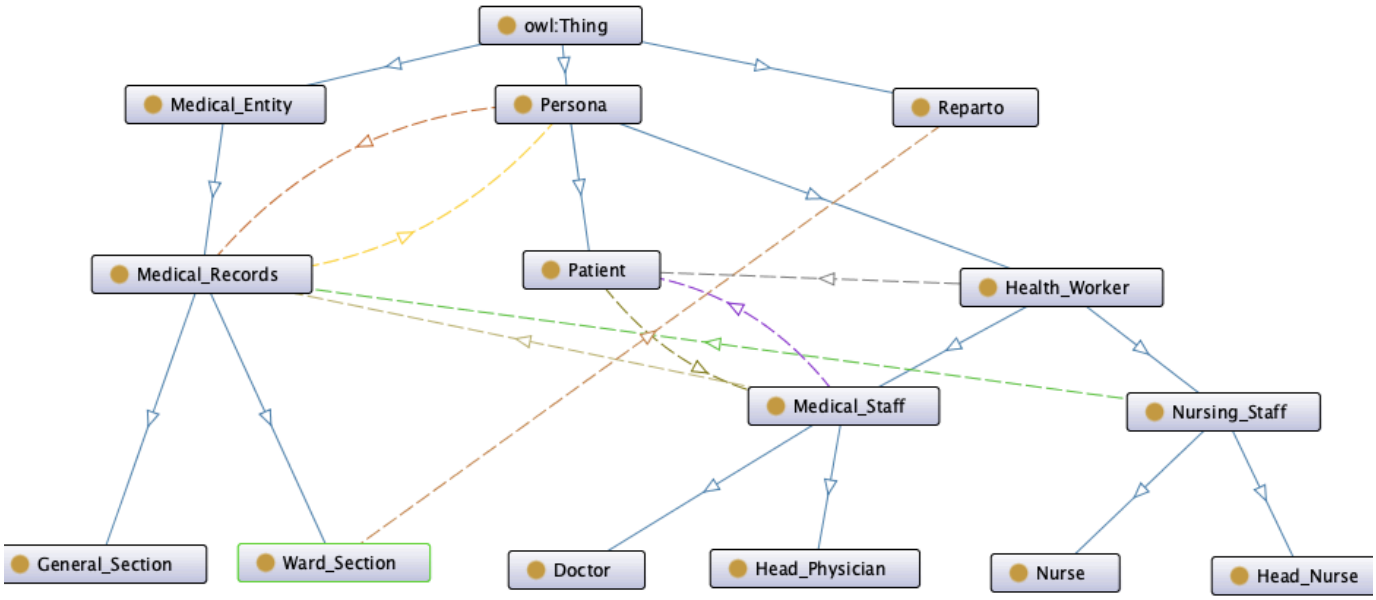


Figura 3.3: ontologia per descrivere le principali figure in ambito ospedaliero e i relativi reparti e cartelle cliniche. Inoltre sono rappresentati i vari collegamenti tra le diverse classi, come ad esempio Patient con una relazione biunivoca verso Medical_Staff attraverso un Object Property

codifica dei termini medici. Seguendo schema.org, è stato scelto di posizionare la cartella clinica (*Medical Records*) all'interno della classe *Medical Entity [37]* di schema.org, che permette così l'accesso a delle Data Properties già esistenti definite per Medical Entity come medicinali, stato di salute, trattamento, allergie, etc.

Grazie a questo schema è stato possibile semplificare il lavoro e realizzare il file *JSAP* (*capitolo 3.2*), molto intuitivo ed internazionale.

La cartella clinica, uno strumento molto complesso da gestire e che differisce, per leggi e norme, da paese a paese; si è dunque cercato di rendere tale strumento il più internazionale e semplice possibile, cosicché sia possibile modificare questa cartella in ulteriori stati e regioni. Come descritto precedentemente, la cartella clinica è stata connessa con una relazione biunivoca alla classe Patient tramite un Object Property, per poter

così descrivere la parte anagrafica della stessa attraverso la parte anagrafica del paziente.

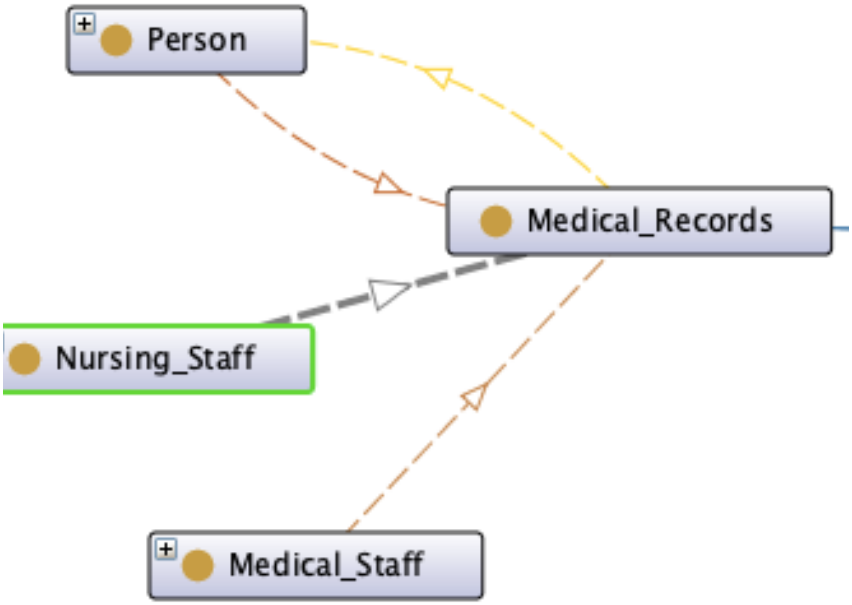


Figura 3.4: collegamento *anagrafico* tra la cartella clinica ed il paziente e tra operatori sanitari e la cartella clinica. Da notare il differente collegamento tra nursing staff e medical records (un infermiere non può modificare una cartella clinica) e tra medical staff e medical records (può sia modificare che creare una cartella clinica)

Inoltre, come riportato in diversi documenti *[38] [39]*, la cartella clinica può essere suddivisa, oltre che nella parte anagrafica, in una sezione comune ed in una specifica. La prima, denominata *General Section*, permette di descrivere le informazioni generali di un paziente, come gruppo sanguigno, allergie note, patologie in atto e terapia continuativa, mentre la seconda, denominata *Ward Section*, permette di descrivere le informazioni di un evento ospedaliero, come causa del ricovero, farmaci assunti, reparto ed infine una lettera di dimissione.

Come si può facilmente evincere, un paziente potrà avere al massimo un *General Record* (dunque un Object Property biunivoco con cardinalità 1-1) e lo stesso paziente potrà avere più *Ward Section* (con un analogo Object Property biunivoco e cardinalità 1-N). Tale discorso è estendibile anche alla figura del medico e del primario, i quali potranno aver compilato più di una cartella clinica. Risulta, quindi, fondamentale per la realizzazione di questo progetto creare un collegamento anche tra *Medical Staff* e il *Medical Record*, poiché sia il software che l'assistente vocale *Mario* (*capitolo 4*) devono poter conoscere l'ipotetico collegamento tra Medical Staff e cartella clinica per la creazione e modifica di quest'ultima. Tuttavia, quest'ultima relazione risulta essere totalmente diversa dalla relazione *Nursing Staff - Medical Record* poiché, coloro che sono racchiusi nella classe del *Nursing Staff*, non possono in alcun modo modificare una cartella clinica, bensì solo visualizzarla, cercando così di emulare ciò che avviene nella realtà. Tali relazioni (Object Properties) sono state denominate, rispettivamente, con *readWriteMedicalWard* e *readOnlyMedicalWard*.

Nella realtà la caposala ha la responsabilità della conservazione della cartella clinica durante il periodo del ricovero, ma in questo elaborato non è stato considerato l'aspetto delle responsabilità della conservazione di una cartella clinica, poiché questa opzione non è stata inserita all'interno del software.

In aggiunta, come fatto per le figure all'interno dell'ambiente ospedaliero, è possibile ampliare non solo le classi ma anche le Data Properties (distretto anatomico, riassunto paziente, gravidanza). Come descritto nel *capitolo introduttivo*, è possibile ampliare questa ontologia liberamente tramite la repository personale di *GitHub [1]*.

La sezione anagrafica può essere realizzata in diverso modo. Nella realtà la sezione anagrafica è direttamente collegata con il Server Nazionale Anagrafico, dal quale è possibile risalire a tutti i dati personali anagrafici *[40]*.

Per quanto concerne le Data Properties utilizzate, oltre a quelle descritte precedentemente, si è cercato di suddividere in base al loro utilizzo: le Data

Figura 3.5: esempio di Data Properties facenti riferimento alla classe Person

Properties relative alla parte anagrafica sono state inserite all'interno della classe Person[22], mentre quelle più specifiche direttamente nella classe desiderata. Alla classe *Person*, ad esempio, sono state attribuite le Data Properties *birthDate*, *email*, *givenName*, *familyName* ed ulteriori proprietà descritte nella *figura 3.5*. La classe *Medical Staff* può avere attributi come studio, orario di ricevimento, numero di telefono lavorativo, mentre per la cartella clinica può avere *drug*, *status* e *patinetResume*[23].
Importanti per le ontologie sono anche i *commenti* e le *Label* che permettono rispettivamente di spiegare ed aiutare gli utenti nel comprendere una determinata classe, mentre le *Label* permettono di denominare le classi scegliendo anche una determinata lingua. Ad esempio, è possibile richiedere all'ontologia di mostrare le sole denominazioni italiane o inglesi, se presenti, mediante queries *SPARQL*.

[22] gli operatori sanitari e pazienti hanno gli stessi attributi relativi alla parta anagrafica, come codice fiscale, nome, cognome, data di nascita, etc.

[23] alcune proprietà sono le stesse utilizzate nella struttura di schema.org

Per concludere, le *individuals* (istanze) hanno un aspetto molto importante per verificare l'integrità e la caratterizzazione della medesima ontologia[24]. Tuttavia, queste ultime, che rappresentano le istanze delle classi, possono essere aggiunte automaticamente con l'utilizzo di un qualsiasi software che sia in grado di utilizzare i file *JSAP*, come spiegato nel *capitolo 3.2* e *capitolo 3.3*.

3.2 Implementazione del file JSAP

Per poter proseguire lo sviluppo del software in linguaggio *Python*, è prima di tutto necessario realizzare un file di estensione *JSAP*, seguendo la medesima struttura definita nel *capitolo 2.4*, necessario per l'interrogazione di grafi e per l'inserimento dei dati.
La definizione dei parametri può differire in base alla posizione dell'*host* e all'utilizzo di una connessione sicura verso il *SEPA*. Per questo elaborato è stata utilizzata una connessione non sicura (come mostrato dalla *figura 3.6*), con l'utilizzo del protocollo *http* al posto di *https* e dal non aver concesso l'autorizzazione al client tramite la stringa *oauth*). La scelta dell'host, invece, si basa sulla posizione in cui è in esecuzione il *SEPA broker*; in questo caso la scelta è ricaduta su *localhost*, poiché tale broker sarà in esecuzione su uno stesso dispositivo. Tuttavia, in una situazione reale, il *SEPA broker* dovrà essere in esecuzione su un *Server* e, dunque, tali parametri saranno sostituiti con nome e posizione del *Server*. Per concludere, i *namespaces* utilizzati sono gli stessi usati per descrivere ed abbreviare *schema.org* e la sintassi *rdf*. In aggiunta è stato introdotto un

[24] le individuals rappresentano l'elemento più specifico con cui si può rappresentare la realtà desiderata tramite ontologie

```
{
  "oauth":{
    "enable":true,
    "register":"https://localhost:8443/oauth/register",
    "tokenRequest":"https://localhost:8443/oauth/token"
  }
}
{
  "sparql11protocol":{
    "protocol":"https",
    "port":8443,
    "query":{
      "path":"/secure/query",
      "method":"POST",
      "format":"JSON"
    },
    "update":{
      "path":"/secure/update",
      "method":"POST",
      "format":"JSON"
    }
  }
}
```

Figura 3.6: esempio di parametri per abilitare la sicurezza e l'autorizzazione del *Client* (fonte figura: http://mml.arces.unibo.it/TR/jsap.html)

namespaces denominato *sha*, che permette di semplificare l'*URI* <http://unibo.it/ontology/SmartHospitalAssistant#>.

Tuttavia, per le *updates* sono state realizzate delle modifiche rispetto a quanto detto nel *capitolo 3.1*, poiché l'inserimento dei dati[25] all'interno dei grafi avviene tramite l'utilizzo di blank node come proposto nella *figura 3.13*.

Per quanto concerne la realizzazione di *queries* ed *updates*, esse verranno trattate nel *capitolo 3.3*.

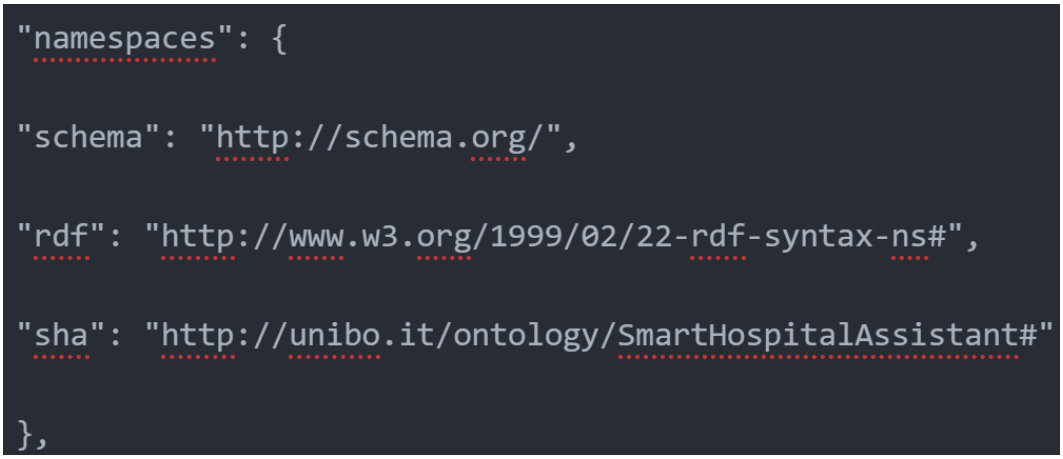
[25] il problema nasce utilizzando Python con il file JSAP per l'inserimento dei dati, in particolar modo con l'uso di *force binding* (*capitolo 3.3*)

```
"host": "localhost",
"oauth": {
    "enable": false
},
"sparql11protocol": {
    "protocol": "http",
    "port": 8000,
    "query": {
        "path": "/query",
        "method": "POST",
        "format": "JSON"
    },
    "update": {
        "path": "/update",
        "method": "POST",
        "format": "JSON"
    }
```

Figura 3.7.1: definizione di host ed i parametri per *SPARQL 11 Protocol* (come mostrato nel *capitolo 2.4*)

```
"sparql11seprotocol": {
    "protocol": "ws",
    "reconnect" : true,
    "availableProtocols": {
        "ws": {
            "port": 9000,
            "path": "/subscribe"
        },
        "wss": {
            "port": 9443,
            "path": "/secure/subscribe
        }
    }
},
```

Figura 3.7.2: definizione dei parametri per *SPARQL 11 SE Protocol* (come mostrato nel *capitolo 2.4*)



```
"namespaces": {

"schema": "http://schema.org/",

"rdf": "http://www.w3.org/1999/02/22-rdf-syntax-ns#",

"sha": "http://unibo.it/ontology/SmartHospitalAssistant#"

},
```

Figura 3.8: *namespaces* utilizzati per l'elaborato

3.3 Implementazione del software per l'aggiunta delle informazioni in ambito sanitario

Per poter usufruire correttamente dell'assistente vocale, è necessario inserire i dati sanitari all'interno dell'ontologia. Benché i dati possano essere inseriti all'interno di *Protégé* tramite l'utilizzo di *individuals* in fase di progettazione, è risultato più efficiente realizzare un *form filling* che permetta una compilazione guidata dei dati. Questa interfaccia rappresenta il modo più frequente con cui sono inseriti dati e informazioni in ambito informatico e sanitario. Per la sua realizzazione è stato scelto di suddividere il software in due parti: una parte utilizzata da un tecnico o dirigente per l'inserimento dei dati personali di un operatore sanitario, e una parte rivolta agli operatori sanitari per l'aggiunta e la modifica dei dati dei pazienti e delle cartelle cliniche. Tuttavia, tale software sarà privo di una fase di *log-in* per tecnici, dirigenti od operatori sanitari data l'impossibilità, per la privacy, di comunicare con un database ospedaliero per ottenere le credenziali. Quest'ultima fase potrebbe essere realizzata con l'utilizzo delle stesse credenziali utilizzate in una struttura sanitaria (per esempio la firma elettronica semplice, formata da un *username* e da una *password*).

Figura 3.9: esempio di schermata di *log-in* da parte di un operatore sanitario

3.3.1 Inserimento di un nuovo Reparto

Per procedere con l'inserimento dei dati di un operatore sanitario è necessario l'inserimento delle informazioni di un nuovo reparto. In questo caso è stato attuato l'inserimento del nome di un reparto e di un relativo *ID* incrementale[26] (che non verrà mostrato all'utente). Tali informazioni potrebbero essere ampliate, per esempio, con l'aggiunta della posizione del reparto all'interno della struttura sanitaria.
Per tutti gli inserimenti nei *form filling* successivi è stato realizzato un controllo accurato sull'input dell'utente. Errori come "nome già esistente" o "caratteri non validi" saranno rilevati dal programma il quale, grazie all'utilizzo di una *Message Box*, provvederà ad avvisare l'utente.

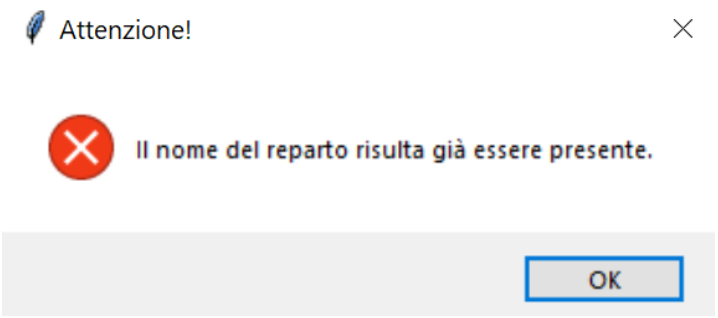


Figura 3.10: *Message Box* di errore. Permette di avvisare l'utente in caso di errato inserimento dei dati

Il rilevamento di un reparto già esistente avviene per mezzo di una query presente nel file con estensione *JSAP*, in cui vengono restituiti tutti i nomi dei reparti già esistenti. E' importante, dunque, aver un ottimo connubio tra Python e lo stesso file *JSAP* (come mostrato dalla gestione delle visualizzazioni dei dati finali). Infatti, quando si invoca una query, essa restituirà un dizionario (in Python: *dict*) in cui sono presenti anche alcune informazioni come il nome delle proprietà e alcuni parametri di connessione,

[26] un ID è necessario per identificare un oggetto, in questo i nomi dei reparti. Il termine incrementale indica che, ad ogni inserimento, l'ID verrà incrementato automaticamente di una unità

che non sono importanti per la visualizzazione finale dei dati. I valori relativi alle informazioni necessarie vengono sempre preceduti dalla stringa *value*. Una possibile soluzione potrebbe essere quella di eliminare tutto quello che non è preceduto dal termine *value* e successivamente rimpiazzare lo stesso *value* con un carattere nullo, come per esempio una stringa vuota. Per far ciò, in questo progetto, è stato scelto di utilizzare semplicemente la proprietà già presente per le stringhe: *replace(… , …)* , in cui il primo parametro rappresenta la stringa da sostituire, mentre il secondo rappresenta la nuova stringa sostituita. Di seguito, con l'utilizzo di uno *split(…)*, è stato possibile creare una lista in cui ogni elemento della lista sia costituito proprio dai risultati della query. Inoltre, il codice per tale visualizzazione, è stato realizzato tramite la creazione di un nuovo file *Python*, denominato su *GitHub [1]* ***query_sparql.py***. Questo file può essere importato e utilizzato in qualunque programma, così da permettere un grande risparmio di righe di codice per la visualizzazione dei risultati della query. Gli unici parametri da utilizzare per tale oggetto saranno il nome della query nel file *JSAP* (in questo caso *QUERY_WARD*, come mostrato nelle immagini successive) e un *path*, ovvero il percorso per poter rintracciare lo stesso file *JSAP*.

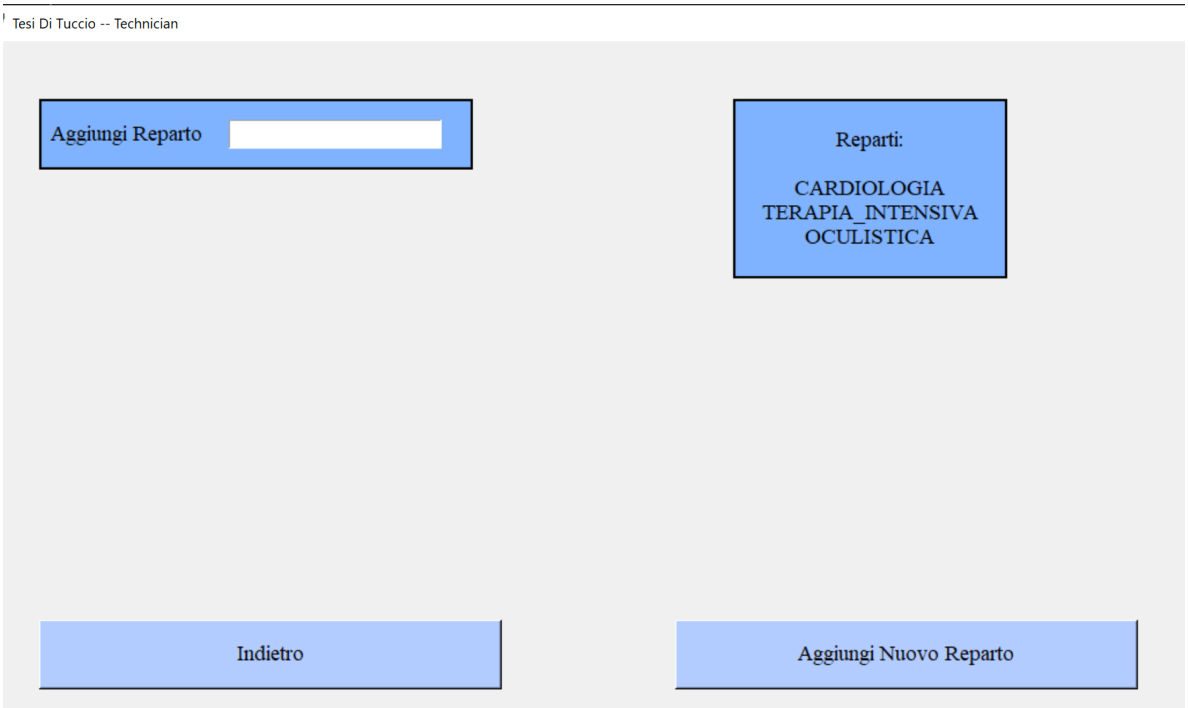


Figura 3.11: *form filling* per l'inserimento di un nuovo reparto. A destra vengono mostrati i reparti già presenti ed in basso sono presenti dei *button* per l'aggiunta del nuovo reparto e per poter tornare alla schermata iniziale

```python
def connessione(sparql_query, path):
    # Open the jsap and connection
    mySAP = open(path,"r")
    sap = SAPObject(json.load(mySAP))
    sc = SEPA(sapObject=sap)

    # querying
    result = sc.query(sparql_query)
    str1 = json.dumps(result) # convert dict into string

    # the follow lines are used for extrapolating only the correct values of the result
    str1 = str1.replace(" ","").replace('"value":"',"&&&").replace("{","").replace("}","").replace(",","\n")
    text_file = open("Output.txt", "w")
    text_file.write(str1)
    text_file.close()
    with open("output.txt","r") as fin:
        with open("input.txt","w") as fout:
            for line in fin:
                if line.startswith('&&&'):
                    fout.write(line)
                else:
                    fout.write("")
    with open("input.txt","r") as file:
        query_result = file.read().replace("&&&","").replace("[","").replace("]","").replace('"',"")
    os.remove("input.txt")
    os.remove("output.txt")

    # return the string (It's a string vector with only the values)
    return query_result
```

Figura 3.12: codice utilizzato per la connessione al file *JSAP* con l'utilizzo del *SEPA broker* e la successiva restituzione dei valori finali, privi di tutti i caratteri non necessari per la loro visualizzazione. Codesto passaggio è stato realizzato con l'utilizzo di due file in contemporanea e l'utilizzo della funzione replace. Il programma cerca la posizione del termine *value* e lo sostituisce con dei caratteri predefiniti ("&&&") e successivamente scarta tutto quello che non sia composto da "&&&"

```
"INSERT_WARD": {
"sparql": "INSERT DATA {graph <http://unibo.it/ontology/SmartHospitalAssistant/Hospital_Ward>
{_:WARD rdf:type sha:Hospital_Ward ; sha:wardName ?wardName}}",
"forcedBindings": {
"wardName": {
"type": "literal",
"value": "Cardiologia"
}
}
},
```

Figura 3.13: esempio di *update* presente nel file *JSAP*. Essa permette l'inserimento del nome di un nuovo reparto nel grafo degli *Hospital_Ward*

```
"QUERY_WARD": {
"sparql": "SELECT ?ward
WHERE {GRAPH <http://unibo.it/ontology/SmartHospitalAssistant/Hospital_Ward>
{?hospital_ward rdf:type sha:Hospital_Ward ; sha:wardName ?ward}}"
}
```

Figura 3.14: query *QUERY_WARD* del file *JSAP* che permette la visualizzazione di tutte le istanze di *wardName*, ovverosia tutti i nomi dei reparti presenti nel grafo *Hospital_ward*

Figura 3.15: esempio di risultati della query *QUERY_WARD* mostrati tramite l'utilizzo della *Dashboard* del *SEPA* (*capitolo 2.4*)

Dopo aver verificato che uno specifico reparto non sia già presente, si passa all'inserimento degli stessi input all'interno dei "grafi". Ogni grafo rappresenta esattamente il nome della classe dell'ontologia. Per esempio la classe *sha:Patient* sarà descritta da un grafo denominato esattamente *sha:Patient*. Il termine *sha*, come già anticipato, rappresenta una semplificazione del namespace che risulta essere <http://unibo.it/ontology/SmartHospitalAssistant#>. In questo caso il codice *SPARQL* non sarà più costituito da una query, bensì da un *update* e più precisamente da un *INSERT*. Come si può evincere facilmente, è necessario prima di tutto capire in quale grafo vanno inseriti i dati (*GRAPH*), quali dati vanno inseriti (per esempio *wardName*, che rappresenta il data property che descrive il nome del reparto) e a quali dati fanno riferimento (*WHERE*). Un ulteriore aspetto da considerare è il *force binding*, ovvero i dati di default che saranno inseriti. Tramite il *force binding* è possibile inserire i dati veri e propri all'interno dei grafi. In *figura*

3.13 si può notare che vi è una differenza tra la data property *wardName* e il dato "CARDIOLOGIA", che rappresenta l'istanza di *wardName*. Tramite il *force binding*, dunque, si possono inserire le istanze.
In questo caso risulta già presente nell'API del *SEPA [41]* una funzione (*update*) che permette l'inserimento delle istanze tramite *force binding* conoscendo il nome dell'*updates* presente all'interno del *JSAP*.
Anche questa funzione è stata implementata all'interno del file *query_sparql.py* presente su *GitHub* e denominata con *insert_one*.
Infine, è possibile ottenere un risultato qualitativamente migliore personalizzando il *form filling* in base al reparto e distribuendo ad ognuno i diversi tipi di *form filling*: alcune informazioni potrebbero non essere idonee per alcune tipologie di reparto.

```
def insert_one(sparql_query, path, force_binding):
    mySAP = open(path,"r")
    sap = SAPObject(json.load(mySAP))
    sc = SEPA(sapObject=sap)
    sc.update(sparql_query,
        forcedBindings=force_binding)
```

Figura 3.16: funzione Python per il caricamento del file *JSAP* tramite il suo percorso (*path*), l'inserimento dei dati tramite il nome dell'*updates* (*sparql_query*) e il *force binding* (*force_binding*)

```
ward_name = input_reparto.get().upper().replace(" ", "_")
force = {"wardName": ward_name}
query_sparql.insert_one("INSERT_WARD", path, force)
```

Figura 3.17: esempio di utilizzo della funzione *insert_one* (figura 3.16) tramite l'oggetto *query_sparql*, in cui *input_reparto* rappresenta la text box utilizzata per l'immissione del nome del reparto

3.3.2 Inserimento dei dati di un Operatore Sanitario

L'inserimento dei dati di un Operatore Sanitario può avvenire in modo similare a quanto esposto nel capitolo precedente, ma con una gestione maggiore dei dati in input. Il *form filling* è stato creato utilizzando lo stesso codice e *text box* proposte per l'aggiunta di un nuovo reparto. Anche in questo caso è utile prevedere gli eventuali errori che un utente può commettere durante l'inserimento dei dati. Nei campi anagrafici obbligatori (come codice fiscale, nome, cognome, data di nascita) il programma avvisa l'utente in caso di dati errati o mancanti. Per Nome e Cognome, inoltre, è stata prevista la sostituzione delle spaziature con l'underscore ("_"), per evitare errori durante l'invio dei dati tra sistemi diversi. Infatti, accade spesso che le spaziature o i ritorni a capo vengano codificate in modo diverso, compromettendo la visualizzazione finale dei dati. Per quanto riguarda la data di nascita è stato inserito un controllo più accurato: non potrà essere inserita una data di nascita inferiore a 18 anni dal momento in cui verrà compilato il campo. Tale aspetto

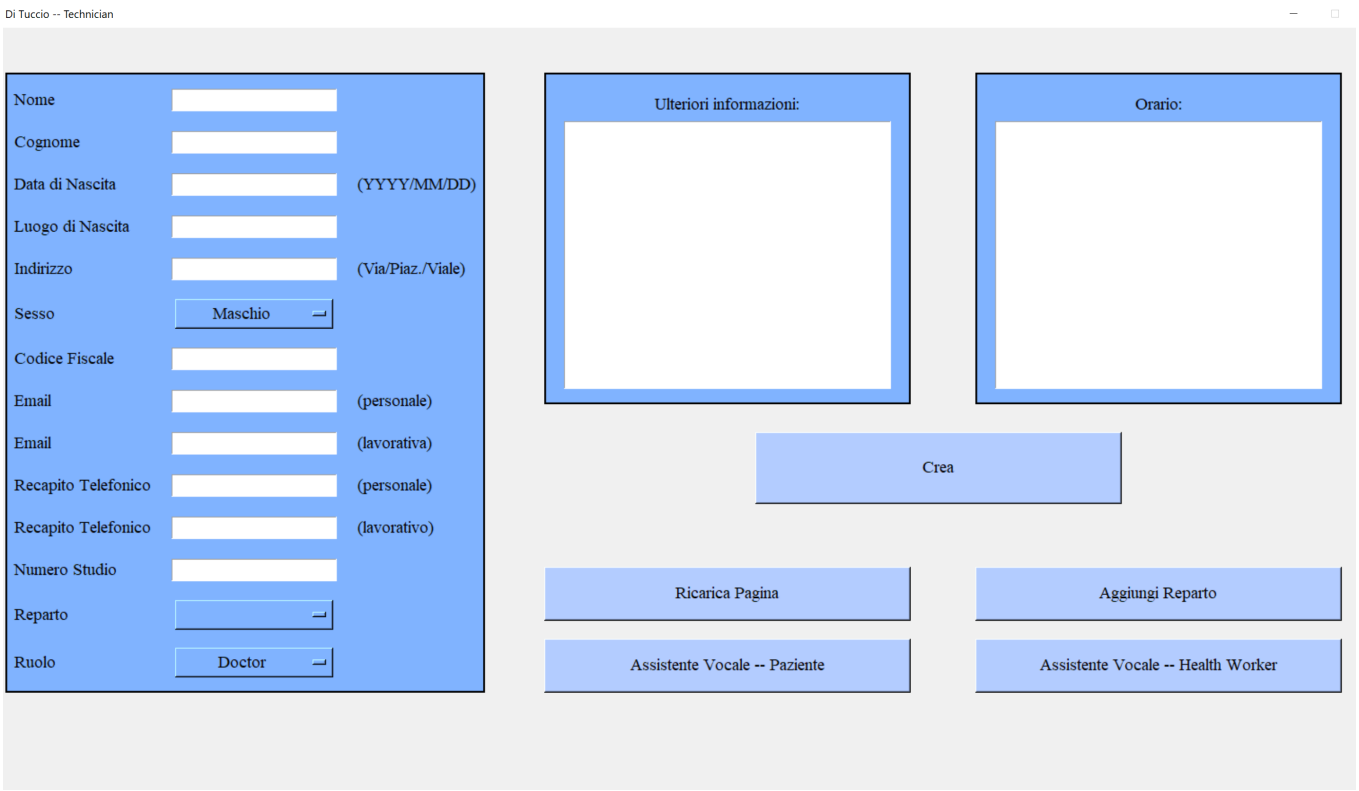


Figura 3.18: schermata per l'aggiunta delle informazioni relative ad un Medico o Primario.

è stato realizzato con la funzione *today() [42]* offerta dal *package datetime* ed è mostrato nella *figura 3.19*.

```
isValidDate = True
try:
    year,month,day = input_eta.get().split('/') # it gets the current day
    if len(year) != 4:
        isValidDate = False
    a = datetime.datetime(int(year),int(month),int(day))
    b = datetime.date.today()
    if (b - a.date()).days == 0 or (b - a.date()).days < 0 or (b - a.date()).days < 365 * 18:
        isValidDate = False
except ValueError:
    isValidDate = False
```

Figura 3.19: controllo sull'input della data di nascita. "isValidDate" rappresenta una variabile booleana per bloccare l'inserimento dei dati nel caso di inserimento errato della data. E' stato utilizzata la locuzione *try - except*, per così gestire qualsiasi tipo di errore (indicato in ambito informatico con *exception*)

Il luogo di nascita e l'indirizzo possono essere, eventualmente, valutati tramite un dizionario *JSON*, contenente tutte le città mondiali e tutti gli indirizzi. Questi file sono facilmente reperibili tramite l'utilizzo del web *[43]*. In questo elaborato si è deciso di non inserire tale funzione per non rallentare il software.

Il controllo del codice fiscale avviene tramite il package realizzato da Caccamo e denominato *codicefiscale*, descritto nel *capitolo 2.5*.

Nel caso in cui un codice fiscale non risulti valido (tramite l'utilizzo di *codicefiscale.is_valid(valore_del_codice_fiscale)* il programma avvertirà l'utente sempre tramite l'utilizzo di una *Message Box*.

Per i campi delle email (lavorativa e personale, *figura 3.20*) è stato effettuato un controllo su ogni carattere presente nella mail *[44]*.

Inoltre, per sesso, ruolo e reparto è stato scelto di utilizzare un menù a tendina *[45]* (ovvero una *Combo Box, figura 3.21*), così da facilitare l'inserimento dei dati. Un particolare riguardo è necessario porre alla *Combo Box* di reparto. E' stata realizzata in due fasi: una prima fase necessaria per determinare quali siano i reparti presenti all'interno del grafo dei reparti e una seconda fase per costruire la *Combo Box* tramite i risultati della query. Per la realizzazione della prima fase è necessario scrivere il codice da inserire nel file *JSAP*

```
regex = r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b'
re.fullmatch(regex, input_email.get())
```

Figura 3.20: controllo dell'email. *input_email* rappresenta la *text box* per l'inserimento della mail, mentre *regex* contiene i caratteri e l'ordine corretto per la costruzione di una email

```
ward = query_sparql.connessione("QUERY_WARD", path)
var_ward = StringVar()
h_ward = ward.replace("\n","|").split("|")
var_ward.set(h_ward[0]) # default --> first value
```

Figura 3.21: utilizzo della funzione *connessione()* del pacchetto *query_sparql* descritto nel *capitolo 3.3.1*. Segue la realizzazione della *Combo Box* con i risultati ricevuti. La query nel file *JSAP* rappresenta la stessa query utilizzata nella *figura 3.13*

utilizzando il linguaggio *SPARQL* per le query, come mostrato dalla *figura 3.13*.

I restanti campi non presentano un controllo e pertanto potranno essere modificati successivamente.

Infine, è necessario realizzare uno *switch* che consenta un passaggio tra le diverse schermate. La sua realizzazione è stata possibile anche grazie l'utilizzo della programmazione ad oggetti. A tal riguardo è stata creata una classe contenente i *form filling*. Una volta avviato il programma, verrà creata un oggetto avente come type esattamente la classe creato in precedenza (nel linguaggio C#, per esempio, è possibile creare una classe denominata Persona avente i suoi attributi e i suoi metodi, privati e pubblici, cosicché sia possibile utilizzarla come *type* di un oggetto: *Persona nome_oggetto = new Persona()*, analogamente a quanto avviene con *int nome_variabile = 1*). Si deve, inoltre,

garantire anche un passaggio di parametri tra i vari oggetti. Un esempio di *switch* è illustrato nella pagina web di *Geeksforgeeks [46]*.

```
"INSERT_MEDICAL_STAFF":{
"sparql": "INSERT
{graph <http://unibo.it/ontology/SmartHospitalAssistant/Medical_Staff>
{?h rdf:type sha:Medical_Staff ; sha:givenName ?givenName ;
sha:familyName ?familyName ; sha:birthPlace ?birthPlace ; sha:fiscalCode ?fiscalCode}}
WHERE {graph <http://unibo.it/ontology/SmartHospitalAssistant/Medical_Staff>
{?h sha:fiscalCode ?fiscalCode ; rdf:type sha:Medical_Staff}}",
"forcedBindings": {
"fiscalCode": {
"type": "literal",
"value": "XXX"
},
"givenName": {
"type": "literal",
"value": "GIANLUCA"
},
"familyName": {
"type": "literal",
"value": "DI_TUCCIO"
},
"birthPlace": {
"type": "literal",
"value": "CESENA"
}
}
},
```

Figura 3.22: esempio di inserimento dei dati di un *medical staff* all'interno del relativo grafo; i dati verranno inseriti solo dopo aver verificato ed inserito il codice fiscale. In questa situazione sono stati inseriti i dati relativi a: nome, cognome e luogo di nascita

```
force = {"fiscalCode": input_cod_fis.get().upper().replace(" ","_"),
    "givenName": input_nome.get().upper().replace(" ","_"),
    "familyName": input_cognome.get().upper().replace(" ","_"),
    "birthPlace": input_luogo_nascita.get().upper()}
query_sparql.insert_one("INSERT_MEDICAL_STAFF", path, force)
```

Figura 3.23: esempio di inserimento dl nome, cognome e luogo di nascita di un *medical staff* utilizzando l'oggetto *query_sparql* definito precedentemente e l'*updates* del file *JSAP* proposto nella *figura 3.21*

3.3.3 Inserimento dei dati di un Paziente e di una Cartella Clinica

Analogamente con quanto accaduto con l'inserimento dei dati di un operatore sanitario, il software svolgerà dei controlli sui dati inseriti di un Paziente e del relativo codice fiscale. Inoltre, è obbligatorio inserire i dati di quest'ultimo prima di proseguire con la creazione di una cartella clinica: questo passaggio è stato realizzato con la verifica del codice fiscale all'interno del grafo denominato *Patient*. Tuttavia, il software creato non sarà in grado di distinguere pazienti aventi lo stesso codice fiscale. Una soluzione potrebbe essere quella dell'utilizzo di un ID incrementale o dell'utilizzo di un database nazionale anagrafico.
Inoltre, per poter sempre garantire un riepilogo dei dati inseriti con l'utilizzo di una *Message Box*.

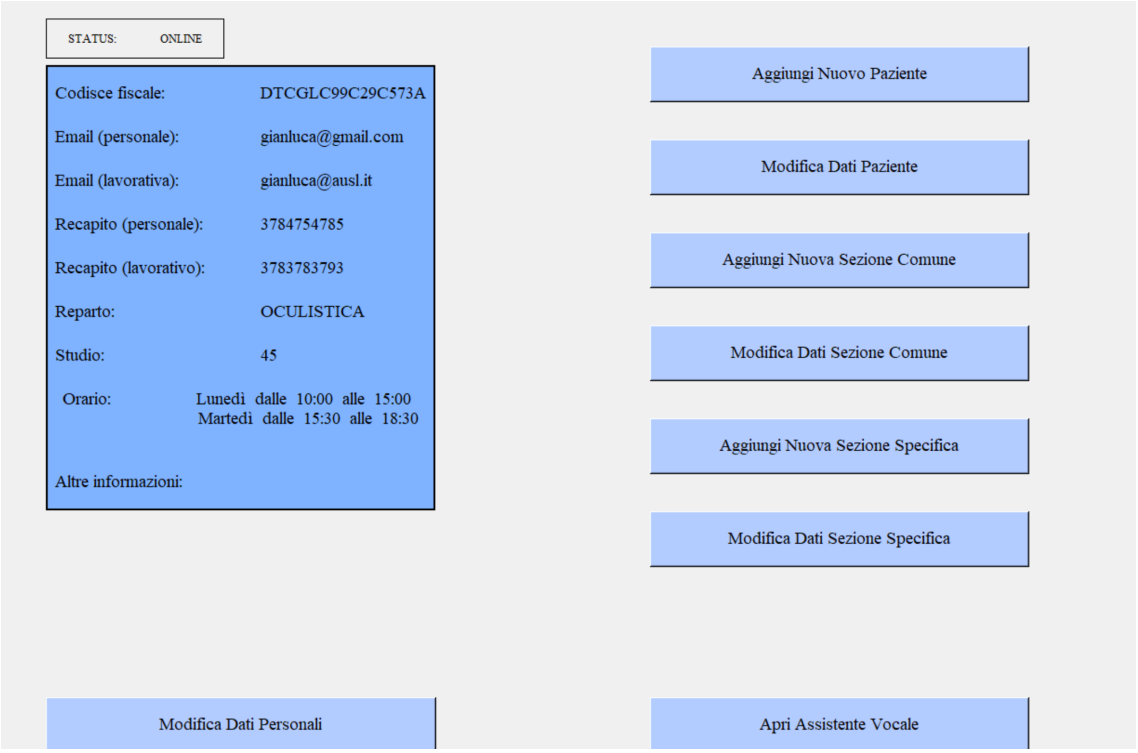


Figura 3.24: schermata del software realizzata per medici e primari. A sinistra è presente una *label* con le proprie informazioni, a destra una serie di pulsanti che permetterà la creazione e modifica dei dati di un paziente o di una cartella clinica

Questo software può essere modificato per le esigenze dei singoli reparti. Ad esempio, per il reparto di oculistica è possibile realizzare un *form filling* di una cartella clinica con solo i dati relativi ad un evento oculistico.
La cartella clinica, invece, è stata suddivisa in due sezioni (come proposto nel

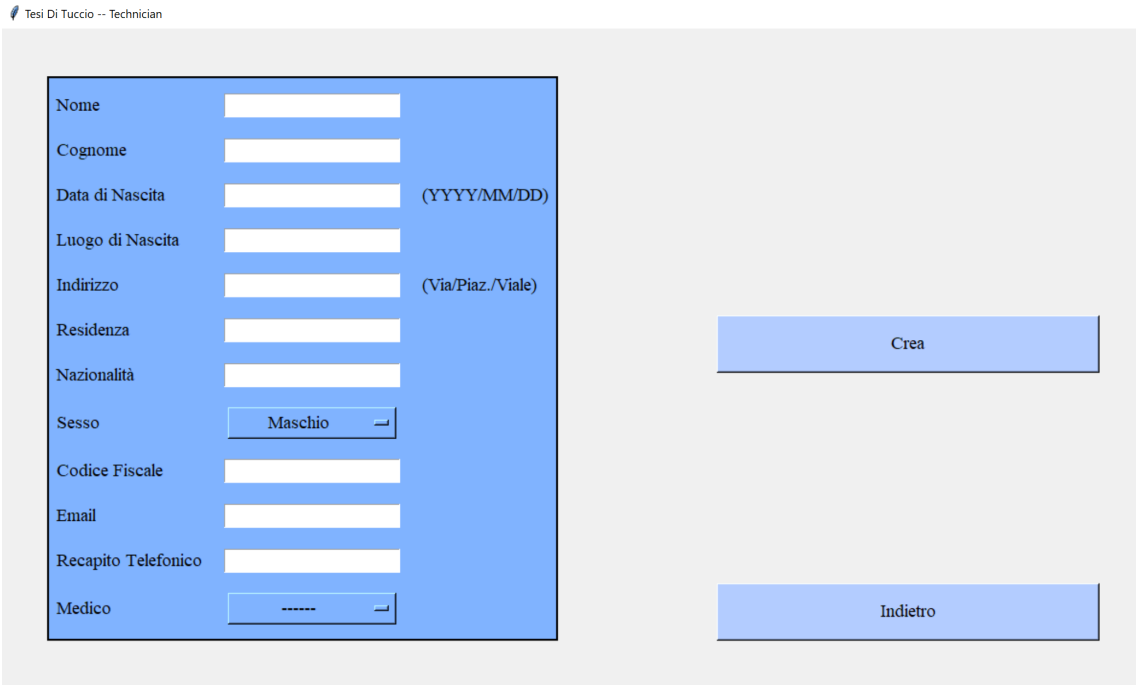


Figura 3.25: schermata per l'inserimento dei dati relativi ad un Paziente. La selezione del medico del Paziente avviene tramite l'utilizzo di una *Combo Box*, avente come risultato i nomi e cognomi dei medici presenti all'interno del grafo *medical staff*. Inoltre, è possibile non selezionare alcun medico e poter modificare tale scelta successivamente

capitolo 3.1): *General Section* e *Ward Section*, che in ambito normativo vengono denominati rispettivamente *Sezione Socio Sanitaria Comune* e *Sezione Socio Sanitaria Specifica*.
L'utente dovrà necessariamente creare la sezione Generale di un paziente prima di poter creare la sezione relativa ad un evento ospedaliero (*Ward Section*); tale situazione verrà controllata dal software e l'utente verrà di seguito avvisato.
Sarà possibile realizzare una ed una sola sezione generale per il paziente, che poi potrà essere modificata. Tale sezione riguarderà l'inserimento del gruppo sanguigno tramite una *Combo Box* e l'inserimento di ulteriori dati, tramite *text*

box, come patologie in atto, terapia continuativa, allergie note e riassunto paziente. Va considerato che tali dati rappresentano solo una piccola parte della realtà.
La *Ward Section*, invece, è realizzata sempre sulla base dell'inserimento del codice fiscale di un paziente. Qui saranno presenti i dati relativi alla diagnosi (in formato *ICD9-CM*), esami effettuati, piano di cura e lettera di dimissione. Per quanto concerne la sezione relativa alla diagnosi, essa è stata realizzata con il pacchetto *Python* denominato *icd9cms*, il quale permette di controllare se il codice inserito risulta valido ed a quale malattia sia associato. Tale scelta

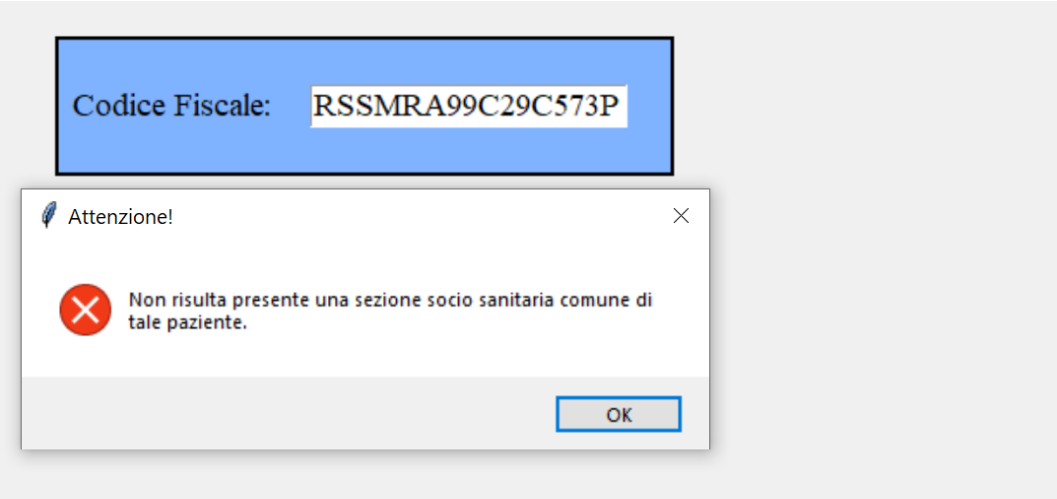


Figura 3.26: creazione di una *Ward Section* della cartella clinica di RSSMRA99C29C573P senza aver ancora inserito i dati relativi alla *General Section*; il software bloccherà la creazione di tale cartella avvisando l'utente

può essere estesa anche ai successivi campi con l'utilizzo dello *SNOMED*.
Un ultimo aspetto di grande importanza è l'ora e la data esatta della creazione e modifica di una cartella clinica. Infatti, il software deve prevedere all'interno dei grafi della cartella clinica l'inserimento della data e ora di creazione, della modifica e dell'utente che ha realizzato l'ultima modifica. E' altresì importante prevedere che data e ora non vengano estrapolate dal personal computer da cui vengono inviati i dati, poiché tale data o ora potrebbe risultare non veritiera. Due possibili soluzioni sono l'utilizzo di una funzione *BIND(now())* in *SPARQL* per ricavare la data e l'ora del Server su cui è in esecuzione il *SEPA*,

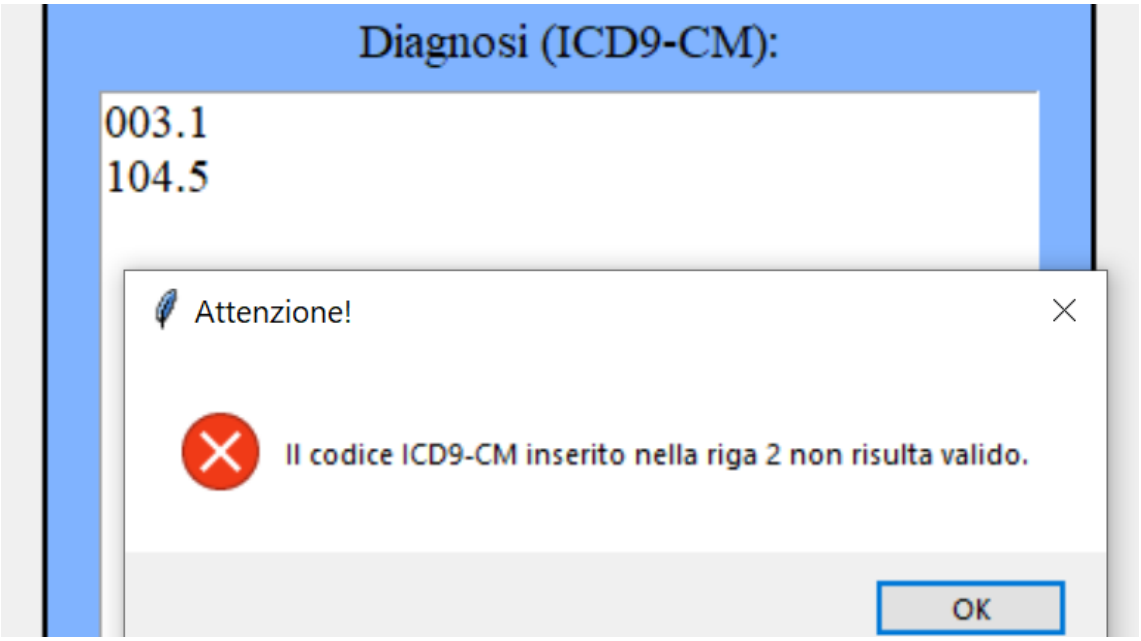


Figura 3.27: inserimento di un codice *ICD9-CM* non valido e relativo messaggio di errore

oppure l'utilizzo di un *Server SNTP* con relativo protocollo NTP (Network Time Protocol) per l'ottenimento della data e ora esatta certificata [39].
Sarà possibile creare anche più di una sezione specifica, facente però riferimento alla sola sezione comune del paziente in questione.
La creazione della cartella clinica, sia comune che specifica, avviene per mezzo di un *ID* incrementale, realizzato in automatico dal Server. Dunque, la

```
diagnosi = txt_diagn.get("1.0", "end-1c").replace(" ","").replace(";","").split("\n")
for i in range(len(diagnosi)):
    check = icd9_package.search_diagn(diagnosi[i])
    if check == None:
        Mbox("Attenzione!", "Il codice ICD9-CM inserito nella riga " + str(i+1) + " non risulta valido.")
        return
for i in range(len(diagnosi)):
    force = {"fiscalCode": input_cod_fis.get().upper().replace(" ","_"),
    "diagn": str(diagnosi[i]),
    "idWSection": id}
    query_sparql.insert_one("INSERT_DIAGNOSI", path, force)
```

Figura 3.28: codice per il controllo delle diagnosi inserite con eventuale *Message Box* di errore; dopodiché, verranno inserite le diagnosi all'interno della stessa *Ward Section* tramite l'utilizzo dell'*ID* di quest'ultima e del codice fiscale.

sezione comune avrà un suo *ID*, la quale potrà essere individuata anche utilizzando il solo codice fiscale, mentre la sezione specifica sarà individuata tramite la coppia *ID* (diverso dall'*ID* della sezione comune) e codice fiscale, facendo però sempre riferimento ad un'unica sezione comune.

3.3.4 Modifica dei dati inseriti

La modifica dei dati avviene semplicemente con l'utilizzo di ***DELETE*** e di un nuovo ***INSERT*** con l'utilizzo del linguaggio *SPARQL*. E' tuttavia importante non eliminare in alcun modo alcuni dati di base che non subiranno la modifica nel corso del tempo, come codice fiscale, nome e cognome. L'utente che userà tale software inserisce il codice fiscale della persona della quale è necessario modificare i dati e, successivamente, il software provvederà al caricamento di quest'ultimi, lasciando bloccati i campi relativi al codice fiscale, nome, cognome, luogo di nascita e data di nascita.

```
input_nome.configure(state="normal") # abilitazione della text box
input_nome.delete(0, "end") # eliminazione del contenuto della text box
input_nome.configure(state="disabled") # disabilitazione della text box
input_nome.insert(END, text[2]) # inserimento di un dato nella text box
input_cod_fis.cget("state") # ottenimento dello stato di una text box
```

Figura 3.29: alcuni comandi fondamentali per la gestione delle *text box* e del loro relativo utilizzo

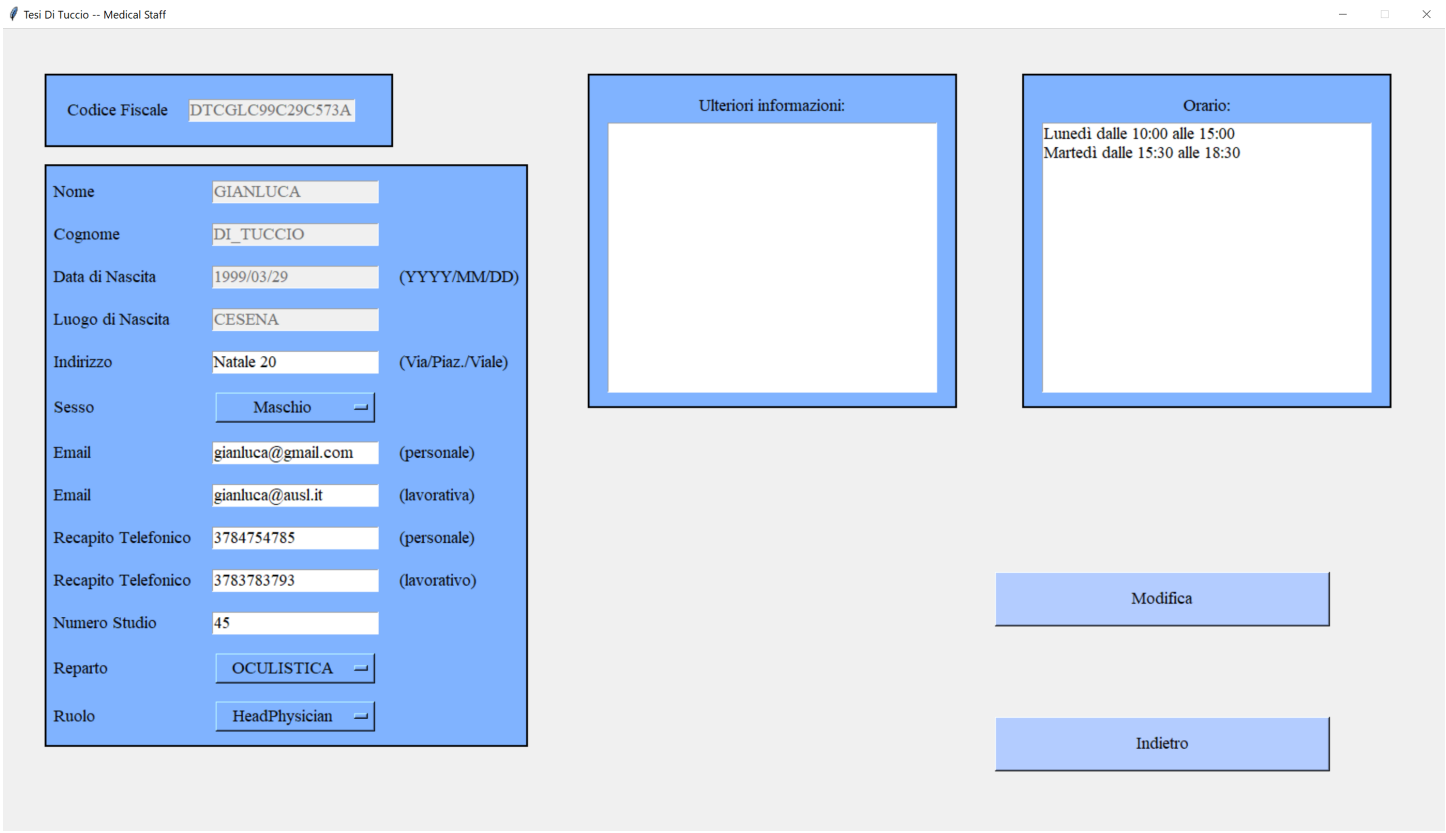


Figura 3.30: schermata per la modifica dei dati di un medico o primario. Alcuni campi risultano bloccati e immodificabili.

4. *Mario*: realizzazione dell'Assistente Vocale

Dopo aver popolato i grafi dell'ontologia anche grazie all'utilizzo del *JSAP*, è possibile proseguire con l'implementazione dell'Assistente Vocale *always on*[27]. Quest'ultimo è stato realizzato sulla base di query e riconoscimento vocale. Grazie ad alcuni pacchetti di *Python*, è possibile comprendere ciò che viene affermato attraverso l'uso del microfono, per poi realizzare una query personalizzata in base alla domanda che l'utente pone a quest'ultimo. Questo sistema servirà per la visualizzazione dei dati senza la necessità di comprendere a fondo i vari programmi per la visualizzazione dei dati, e con la possibilità di comunicare con *Mario* in qualsiasi forma. Il sistema sarà diviso in due programmi, uno realizzato a misura per i pazienti che cercano le informazioni di un determinato medico, mentre il secondo realizzato per operatori sanitari che richiedono la visualizzazione di dati statistici o di dati relativi al paziente o alla cartella clinica.

4.1 Assistente Vocale - Lato Paziente

Mario permetterà agli utenti di poter chiedere informazioni di un medico come numero di telefono, email, orario, reparto o studio. Inoltre, sarà garantita anche la possibilità di cercare un termine medico sul Web attraverso l'uso di Wikipedia. Questo sistema non potrà in alcun modo sostituire l'operato di un segretario, ma rappresenta una alternativa molto economica per venire incontro alle esigenze degli utenti e dei pazienti internazionali con l'utilizzo del traduttore. Nel mondo ospedaliero capita molto spesso che un utente si

[27] con il termine *always on* si vuole intendere un assistente vocale sempre attivo, senza la necessità di premere un pulsante per attivarlo

trovi costretto a richiedere informazioni a medici o infermieri del reparto per cercare un determinato medico.
I primi parametri da gestire sono il nome per invocare l'assistente vocale, la parola per chiuderlo e il valore del microfono, necessario per isolare i disturbi esterni[28]. È stato scelto il nome *Mario*, poiché rappresenta un nome comprensibile dalla quasi totalità delle lingue straniere, mentre per chiudere il programma è stato scelto semplicemente "*chiudi*"; tuttavia, questo parametro può essere modificato molto semplicemente in base alle esigenze di un ambiente ospedaliero. Un secondo step è l'implementazione del dizionario o stringa per i termini comuni e dell'interfaccia dell'assistente vocale. In questo elaborato è stato deciso di utilizzare un vettore di tipo *string* contenente le principali parole che un utente potrebbe nominare: orario, posizione di un medico, reparto ed altri, che rappresentano le *key words*. E' necessario che *Mario* recepisca la frase in ingresso e verifichi quale di queste *key words* è contenuta all'interno della frase, avviando poi la risposta desiderata.
Tale passaggio poteva essere altresì realizzato con l'utilizzo di un *dict*, ma è stato scelto di utilizzare un vettore di stringhe per non rallentare

```
def who(name, surname, answer):
    if key_word[0] in answer or key_word[1] in answer or key_word[12]
        orario(name, surname)
    if key_word[2] in answer or key_word[3] in answer or key_word[4]
        studio(name, surname)
    if key_word[5] in answer or key_word[6] in answer or key_word[7]
        telefono(name, surname)
    if key_word[8] in answer or key_word[17] in answer:
        email(name,surname)
    if key_word[9] in answer:
        reparto(name,surname)
```

Figura 4.1: esempio della sintassi *in* per verificare se una *key word* (un vettore di stringhe di default) sia presente all'interno della frase; segue la successiva operazione di riposta all'utente

[28] il valore dipende fortemente dal tipo di disturbo presente all'interno di un reparto e può variare da 50, quasi nessun disturbo, a 1000, valore per isolare una conversazione a basso volume, a oltre 2000)

eccessivamente il software. È stata prevista una frase di default ("*Mi dispiace, non ho capito*") per tutte quelle espressioni che non contengono una *key word*. Per l'interfaccia è stata scelta l'immagine di un'anatra in formato *ASCII* in due versioni: una dormiente, *sleep mode*[29], ed una in attività denominata *active mode*[30].
Dopodiché, è possibile passare all'implementazione dell'assistente (con i pacchetti definiti nel *capitolo 2.5*). L'implementazione è avvenuta in due step, uno per l'invocazione dell'assistente vocale tramite il nominativo *Mario*, ed una per lo svolgimento della conversazione tra l'assistente vocale e l'utente.

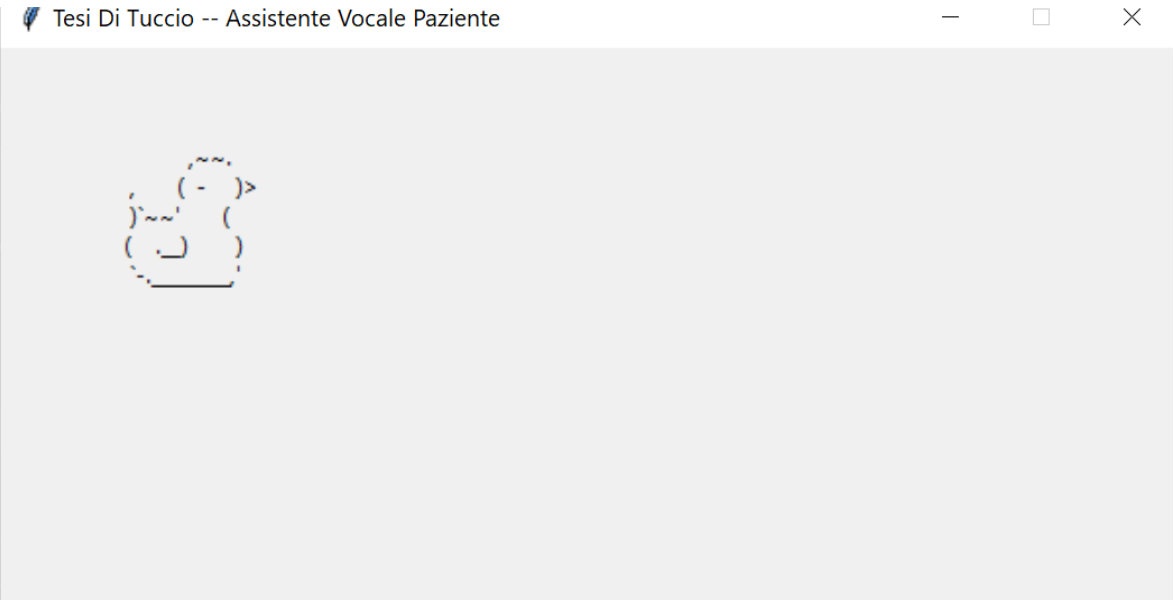


Figura 4.2: schermata dell'assistente vocale in *sleep mode*

4.1.1 Primo step: invocazione dell'assistente vocale

Per questo primo passaggio l'assistente dovrà essere in grado di mettersi in ascolto per la ricerca delle sole due parole chiave: nome dell'assistente e

[29] l'assistente vocale non è stato invocato

[30] l'assistente vocale è stato invocato

parola d'ordine per la chiusura; in tutte le altre situazioni, esso non risponderà e non sarà così in grado di avviare una conversazione. Ovviamente la privacy può essere assicurata, poiché ogni singola conversazione verrà poi eliminata e, in caso di malintenzionato, sarà possibile attuare delle soluzioni via software.
Nel caso in cui venga pronunciata la parola d'ordine per la chiusura del software, l'assistente provvederà a chiudersi con l'utilizzo di *sys.exit()*, mentre la sua invocazione determinerà l'avvio dello step due. Come nel *capitolo 3*, è stata realizzata una funzione su un oggetto denominato *use_microphone* che potrà essere semplicemente usato nelle fasi successive o in programmi futuri. Questa funzione si metterà in ascolto per un determinato periodo di tempo o in alternativa restare in ascolto fino a che l'utente non finisca la conversazione, come avviene nello step due. Dopodiché, provvederà ad interpretare la frase con l'utilizzo di Google Translate (e dunque sarà necessario garantire una connessione ad internet). Infine, restituirà l'interpretazione della frase, o in tutti gli altri casi la stringa "*Error*". Inoltre, per garantire un *always-on*, il programma controllerà ogni 1.2 secondi se verranno pronunciate i termini necessari per l'avvio dello step due o per la chiusura del software.
Questo passaggio può prevedere anche l'utilizzo di una lingua straniera; infatti, basterà utilizzare come parametro il valore "*en-EN*" per l'inglese o "*fr-FR*" per il francese. Una possibile soluzione potrebbe essere quella di utilizzare dei tasti per la lingua per impostare il valore del parametro precedente, oppure tramite l'utilizzo di Google Translate: nel caso in cui venisse pronunciata il nome di una lingua (non per forza in italiano), l'assistente provvederà a cambiare la lingua.

```python
def understand(valore_microfono, parA, parB, lang, pause):
    riconoscitore = speech_recognition.Recognizer()
    riconoscitore.energy_threshold = valore_microfono
    riconoscitore.dynamic_energy_threshold = False
    riconoscitore.pause_threshold = pause
    with speech_recognition.Microphone() as source:
        riconoscitore.adjust_for_ambient_noise(source)
        kkk = riconoscitore.listen(source, parA, parB)
    try:
        text = riconoscitore.recognize_google(kkk, language=lang)
        return(text)
    except Exception:
        return("Error")
```

Figura 4.3: funzione per interpretare e tradurre la conversazione dell'utente. Utilizza il pacchetto di Google Translate. In caso di non successo, restituirà la stringa "*Error*"

```python
def start():
    text = use_microphone.understand(valore_microfono, None, 1.5, "it-IT", 0.5)
    if text == "Errore":
        return
    else:
        if text.lower() == nome_assistente:
            talking_duck()
            response()
        elif text.lower() == nome_chiusura:
            vocal_label.configure(text="Chiusura in corso...")
            ws.update()
            audio_bot("chiusura in corso")
            sys.exit()
```

Figura 4.4: utilizzo della funzione *understand* dichiarata nella *figura 4.3*. Un ulteriore aspetto importante è l'utilizzo della funzione *lower()* per via del problema di *case sensitive*

```
def audio_bot(cosa_dire):
    tts = gTTS(text=cosa_dire, lang='it')
    nome_file_mp3 = "prova.mp3"
    tts.save(nome_file_mp3)
    playsound.playsound(nome_file_mp3)
    os.remove(nome_file_mp3)
```

Figura 4.5: funzione per l'utilizzo degli *speakers* (o ulteriore dispositivo di output) per la risposta all'utente. E' importante garantire che tale assistente vocale abbia i permessi dell'amministratore sul Sistema Operativo per poter riprodurre il file e cancellarlo

4.1.2 Secondo step: conversazione con l'assistente vocale

In caso di invocazione dell'assistente, si avvierà una fase di *response* per garantire la risposta all'utente.
La conversazione potrebbe proseguire in tre differenti modi: l'assistente è in grado di rispondere, l'assistente non è in grado di rispondere ed infine l'assistente non ha rilevato alcuna conversazione. Nell'ultimo caso, il software ritornerà alla prima fase, in attesa di una futura invocazione o chiusura.
Nel caso in cui l'assistente vocale non comprenda la frase o nel caso in cui non sia contenuta una *key word*, l'assistente avverte l'utente e lo invita a chiedere "*Cosa sai fare?*". Tale locuzione permetterà all'assistente di mostrare quali attività esso sarà in grado di compiere.
I restanti casi permetteranno all'assistente di rispondere all'utente. Poiché la frase quasi certamente riguarderà un determinato medico, è importante che in essa vi siano contenuti i nomi e cognomi corretti. Dunque, l'assistente vocale eseguirà una query (sempre con l'appoggio del *SEPA* e del *JSAP*) per determinare tutti i nomi e cognomi dei medici presenti nel grafo dei *medical staff*. In caso di presenza di medici con lo stesso nome e cognome, si potrebbe implementare una funzione che permetta all'assistente vocale di informare l'utente fornendo in aggiunta il nome del reparto di ciascuno, per aiutare il

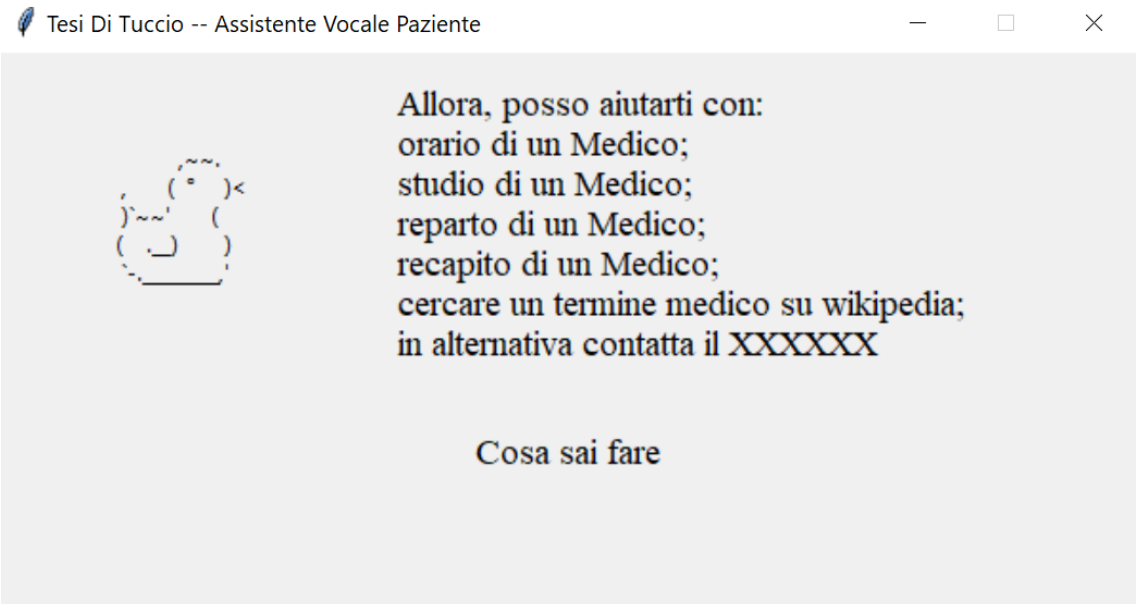


Figura 4.6: assistente vocale in *active mode* (risposta alla domanda "cosa sai fare"). A destra è mostrata una *label* con il messaggio dell'assistente vocale; in basso la frase esclamata dall'utente. In questa schermata la stringa *XXXXXX* potrebbe essere sostituita con il recapito telefonico dell'ospedale o con un altro contatto

paziente ad individuare il medico corretto. Nel caso in cui l'assistente individuasse il corretto medico, passerebbe poi alla comprensione della frase, come descritto nella *figura 4.1*. Tuttavia, ora è necessario poter ricercare le giuste informazioni per un determinato medico, con l'utilizzo del termine *FILTER (?nome_cognome, "nome_da_cercare")*. Essendo però una richiesta personalizzata per via della presenza del "*nome_da_cercare*", risulta impossibile poter preparare una query nel file *JSAP*. Dunque, il software provvederà alla creazione di un nuovo file *JSAP* contenente la query personalizzata. Per far ciò è necessario un pacchetto *Python* denominato *create_sparql* per la generazione dei parametri del file *JSAP* e successivamente un *overwrite* sullo stesso file per la scrittura della query personalizzata. Alla fine, tale file verrà rimosso.

```
def creating(path_new, stringa):
    text_file = open(path_new + "\\temp1.jsap", "w")
    text_file.write(pass_sparql())
    text_file.close()
    with open(path_new + "\\temp1.jsap","r") as fin:
        with open(path_new + "\\temp2.jsap","w") as fout:
            for line in fin:
                if line.startswith('&&&&&'):
                    fout.write(stringa)
                else:
                    fout.write(line)
    os.remove(path_new + "\\temp1.jsap")
    stringa = query_sparql.connessione("QUERY", path_new + "\\temp2.jsap")
    os.remove(path_new + "\\temp2.jsap")
    return stringa
```

Figura 4.7: funzione creating dell'oggetto create_sparql per la creazione dei parametri di connessione (descritto nella funzione pass_sparql) e sostituzione di una stringa di default ("&&&&&&") con la query personalizzata

```
SELECT ?reparto WHERE
{GRAPH <http://unibo.it/ontology/SmartHospitalAssistant/Medical_Staff>
{?h rdf:type sha:Medical_Staff ; sha:wardName ?reparto ;
sha:givenName ?givenName ; sha:familyName ?familyName .
FILTER (regex (?givenName, "nome_da_cercare")) .
FILTER (regex (?familyName, "cognome_da_cercare")) }}
```

Figura 4.8: codice SPARQL per cercare il reparto di un medico conoscendo il suo nome e cognome

```
def reparto(name, surname):
    stringa = query_reparto
    name_query = '"QUERY"'
    stringa = name_query + ': { "sparql": ' + stringa + '}'
    stringa = create_sparql.creating(path_new, stringa)
    stringa = stringa.replace("%%", "ì").replace("&&"," ").replace("&"," ").replace("_"," ")
    vocal_label.configure(text="Il reparto è:  " + stringa)
    ws.update()
    audio_bot("Il reparto è:  " + stringa)
```

Figura 4.9: esempio di utilizzo della funzione creating e del codice SPARQL proposto nella figura 4.8

4.2 Assistente Vocale - Lato Medico

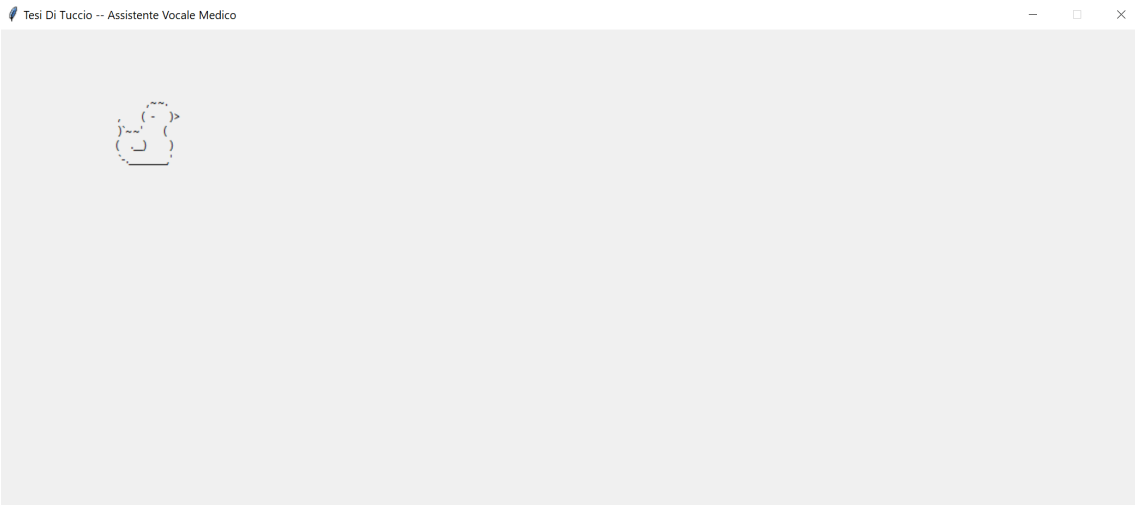


Figura 4.10: schermata principale dell'assistente *Mario* per medici e primari

L'assistente vocale per Medici rappresenta un'ulteriore alternativa all'utilizzo di assistenti in ambito sanitario. La sua implementazione risulta quasi identica a quella per la realizzazione dell'assistente per Pazienti. Tuttavia, è necessario considerare anche l'aspetto della privacy. Molte informazioni strettamente correlate ai pazienti non potranno essere divulgate attraverso una conversazione orale. Per garantirne la privacy, l'assistente provvederà solo a mostrare a video le informazioni personali, senza divulgarle in alcun modo. Inoltre, non potrà essere effettuata alcuna modifica ai dati utilizzando l'assistente vocale, sia perché non è in grado di riconoscere chi effettua la modifica e sia per l'aspetto della privacy. Sarà, inoltre, possibile accedere all'assistente solo dopo aver eseguito l'accesso con username e password o riconoscimento facciale.
L'assistente vocale è in grado di adattarsi alle esigenze del personale medico mediante *form filling ad hoc*. In questo modo il medico non deve conoscere la struttura dei *form filling* di tutti i software ospedalieri a priori.

Il medico può interfacciarsi con l'assistente vocale in qualsiasi modo e i compiti che l'assistente potrà svolgere sono:
- cercare un codice *ICD9-CM* o richiedere statistiche riguardanti tali codici;
- richiedere le cartelle cliniche di un determinato giorno di un reparto;
- richiedere le proprie cartelle cliniche compilate;
- cercare una cartella clinica in base al suo *ID*;
- cercare una cartella clinica con il codice fiscale di un paziente [31].

Inoltre, è prevista anche una funzione che permetterà di visionare le sezioni comuni o specifiche delle cartelle cliniche, semplicemente utilizzando il termine "comune" o "specifico" all'interno di una frase. Il medico potrà rivolgersi all'assistente per esempio con "ciao, mostrami tutte le cartelle cliniche comuni del 27 luglio 2021 del reparto di oculistica".

4.2.1 Ricerca di una cartella clinica con il codice fiscale del Paziente o del personale medico

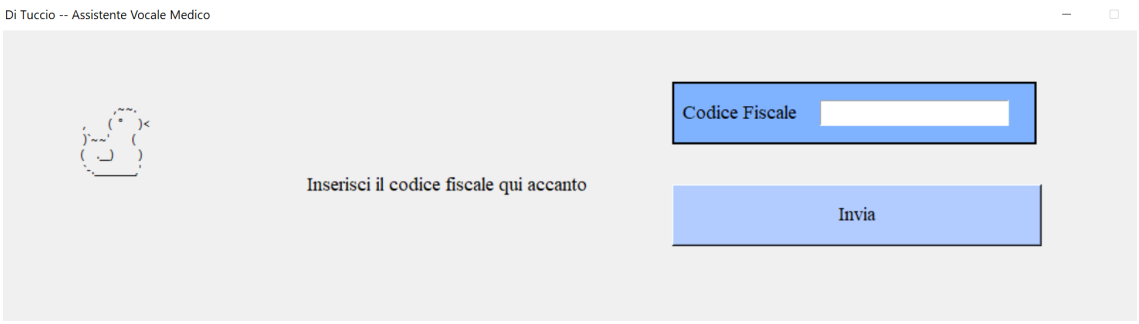


Figura 4.11: esempio di creazione del *form filling* per la ricerca di una cartella clinica di un paziente attraverso il codice fiscale di quest'ultimo

Uno dei principali vantaggi che può offrire un assistente vocale è la capacità di potersi interfacciare in qualsiasi modo con l'operatore sanitario, senza la necessità che quest'ultimo conosca a fondo l'utilizzo dei software per la visualizzazione dei dati.

[31] si ricorda che tale software non sarà in grado di distinguere persone aventi lo stesso codice fiscale

Tramite l'assistente *Mario* sarà possibile chiedere di visualizzare la sezione socio sanitaria comune di un paziente, con le sue relative sezioni socio sanitarie specifiche. Dopo l'inizio della conversazione, l'assistente vocale mostrerà una *text box*, in cui inserire il codice fiscale del paziente (*figura 4.11*), e verranno mostrati a video tutti i risultati inerenti al paziente selezionato (se esso risulta presente all'interno del grafo dei pazienti). Un procedimento analogo è stato realizzato per la visualizzazione delle cartelle cliniche compilate e modificate da un determinato medico o primario. Tuttavia questa non rappresenta una buona soluzione dal punto di vista della privacy, perché permetterebbe a qualsiasi medico di poter visionare le cartelle cliniche compilate da un altro medico. Una possibile soluzione è la possibilità di utilizzare il codice fiscale ricavato tramite la procedura di *log in* (*capitolo 3.3*). In questo elaborato non è stata implementata una funzione di *log in* tramite l'utilizzo di una firma elettronica semplice, dunque il software, attualmente, non è in grado di rispettare nessun requisito per la privacy. Un'ulteriore opzione che potrebbe essere implementata è l'inserimento di un account specifico per un operatore sanitario in grado di poter visionare le cartelle cliniche di qualsiasi medico attraverso la selezione da una *Combo Box*, mentre le restanti figure ospedaliere potrebbero visionare le sole cartelle cliniche private. Sarebbe preferibile adottare questa opzione per evitare che un operatore sanitario possa visionare la cartella clinica di un paziente che non ha in cura, non rispettando così la privacy di quest'ultimo.

```
var = IntVar()
btn_invia = Button(ws, width = 15, text = 'Invia',
    command=lambda: var.set(1), font=f, bg=background_window,
    activebackground="#3399ff")
btn_invia.place(height=60, width=360, x = 700, y = 150)
btn_invia.wait_variable(var)
```

Figura 4.12: utilizzo di *wait_variable* per mettere in *stand by* il programma fino a che il medico o primario non prema sul pulsante denominato *btn_invia*

```
scroll = Scrollbar(frame2)
scroll.pack(side="right", fill="y")

listbox = Listbox(frame2, yscrollcommand=scroll.set)
listbox.pack(side="right", fill="both")
for i in range(len(stringa)):
    listbox.insert("end", stringa[i])
```

Figura 4.13: codice per la creazione di una *list box* per la visualizzazione finale dei dati e di una *scroll bar* (barra di scorrimento) per poter scorrere i risultati all'interno della *list box*

Rispetto al *capitolo 4.1*, è stata introdotta un'ulteriore sintassi come mostrato dalla *figura 4.12* e dalla figura 4.13, mentre le *queries* rappresentano le stesse discusse nel *capitolo 3*.

4.2.2 ICD9-CM: ricerca e statistiche

Grazie all'utilizzo del pacchetto *icd9cms* (*capitolo 2.5* e *capitolo 3.3.3*) è possibile implementare la ricerca del nome della diagnosi di un determinato codice o, per esempio, poter visionare quale sia stata la malattia più diagnosticata in un determinato giorno a fini statistici. Ulteriori statistiche che potrebbero essere implementate sono:
- codice più diffuso in un arco temporale (realizzabile con una funzione ***FILTER*** di *SPARQL*);
- codice più diffuso in una determinata regione (se il software viene espanso al territorio nazionale);
- visionare il numero di volte in cui è stata diagnosticata una malattia (realizzabile con una funzione ***COUNT*** di *SPARQL*).
Tale funzione è stata implementata con la sintassi *in* (figura 4.14) presente in *Python*, avviando la ricerca statistica solo se il medico o primario pronuncerà i

termini "statistica" o “statistiche” con il relativo codice, mentre la ricerca del nome di un codice avverrà analogamente a prima, con il termine “codice”. L'assistente sarà in grado di distinguere la *key word* ed il codice *ICD9-CM* dal resto della frase.

```
if "statistiche" in text:
    for y in text:
        if y.isnumeric():
            if icd9_package.search_diagn(y) != None:
                ## codice ##
```

Figura 4.14: esempio di utilizzo della sintassi in per la ricerca di un termine ("statistiche") all'interno di una frase (*text*) e della ricerca di un codice *ICD9-CM*

4.2.3 Ricerca di una cartella clinica tramite ID e reparto di un determinato giorno

Un’ulteriore opzione presente all’interno dell’assistente *Mario* è la possibilità di ricercare una determinata cartella clinica in base all’*ID* di quest’ultima.
E’ importante inoltre notare la presenza di una priorità di termini tecnici all’interno del programma: la priorità risulta decrescente dalle *key word* paziente e medico, passando per “statistiche” e *ICD9-CM* fino alle *key word* per la ricerca di una cartella clinica tramite *ID*.
In questo caso sarà necessario pronunciare le *key word* “comune” o “specifico” seguito dal codice della cartella. L’assistente avviserà l’utente nel caso in cui non venga specificato una delle due *key word* precedenti o nel caso in cui l’*ID* non risulti valido o presente nel grafo della cartella clinica comune o specifica.

Con l'utilizzo della sintassi *in* è possibile, inoltre, visionare il codice *ID* e il codice fiscale delle sezioni socio sanitarie comuni e specifiche di un determinato reparto in una determinata giornata.

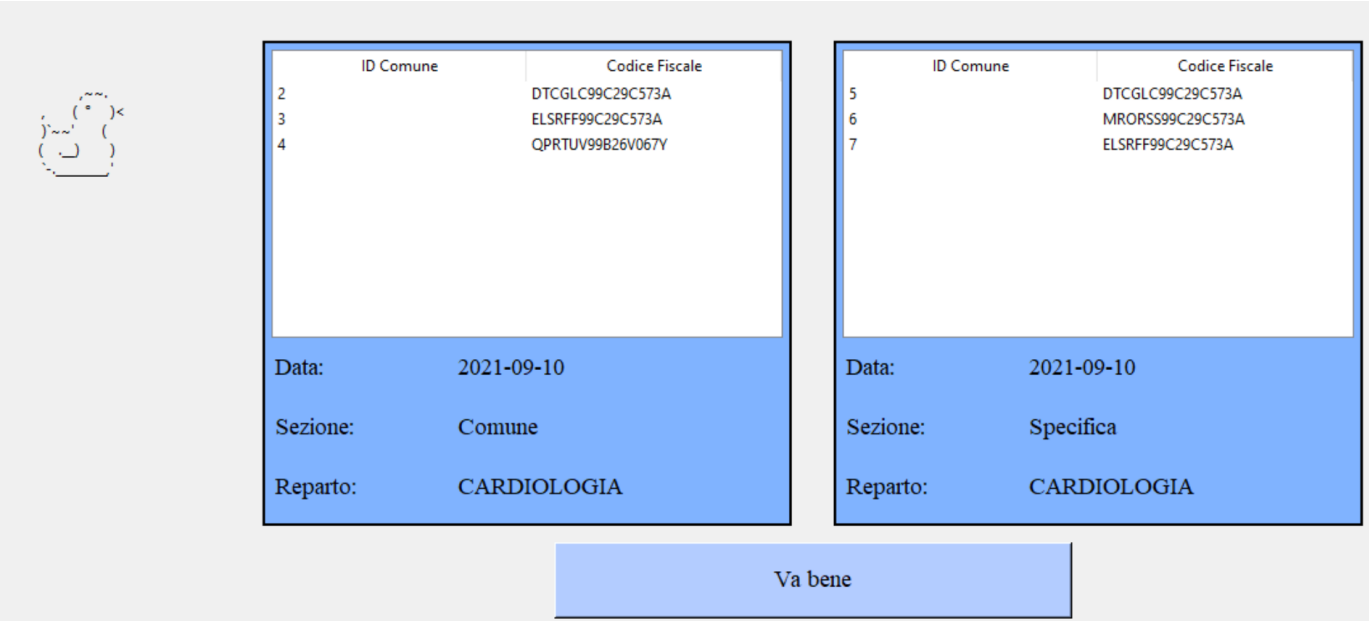


Figura 4.15: schermata per la visualizzazione dell'*ID* e del relativo codice fiscale delle cartelle cliniche (comuni e specifiche) create il "2021-09-10" del reparto di "*CARDIOLOGIA*"

Conclusioni

Gli assistenti vocali rappresentano una buona risorsa se accoppiata correttamente con le applicazioni già in uso in ambito ospedaliero, mentre le ontologie sono una valida strategia all'avanguardia rispetto agli attuali database basati *SQL*. Quest'ultimi presentano una struttura rigida e spesso difficile da modificare, mentre le ontologie, accoppiate con l'architettura *SEPA*, permettono le creazioni di grafi che possono rappresentare accuratamente la realtà e, soprattutto, sono una risorsa libera e usabile da chiunque.
Un ulteriore vantaggio nell'uso dell'assistente vocale è poter dare supporto alle persone che non sono in grado di utilizzare il Web, analogamente al cercare una determinata diagnosi o cura attraverso l'uso di Wikipedia Orale. Così strutturati, assistenti come *Mario* possono essere adattati alle varie tecnologie sotto forma di applicazione gratuita da poter installare sul proprio smartphone o tramite l'utilizzo di un arduino come avviene per i *Totem* presenti in ospedale. Possibili risvolti futuri potrebbero essere applicativi in grado di combinare il fascicolo sanitario elettronico attraverso l'uso con un assistente vocale (in coabitazione con l'intelligenza artificiale), associato ad un riconoscimento facciale, come avviene con le tecnologie attuali di Apple e Samsung, che permettono l'utilizzo dello SPID personale attraverso un riconoscimento facciale.
Inoltre, gli assistenti vocali potrebbero essere accoppiati con l'utilizzo di avatar, i quali permetterebbero di aver un miglior interfacciamento con l'utente.
In conclusione, gli assistenti vocali rappresentano una soluzione economica e volatile di enorme supporto in tutti gli ambiti professionali, garantendo l'inclusività a stranieri e a persone con gravi patologie, come ipovedenti o non vedenti, tramite la configurazione di una tastiera *braille*.

Bibliografia

[1] GitHub personale: https://github.com/DitucSpa/MyThesis_BiomedicalEngineering

[2] Wikipedia, definizione filosofica Ontologia: https://it.wikipedia.org/wiki/Ontologia

[3] Wikipedia, definizione informatica Ontologia: https://it.wikipedia.org/wiki/Ontologia_(informatica)

[4] Citazione Berners Lee: https://www.w3.org/People/Berners-Lee/ShortHistory.html

[5] Definizione Ontologia Treccani: https://www.treccani.it/enciclopedia/ontologia/

[6] Stanford Center for Biomedical Informatics Research, Novembre 1999, https://protege.stanford.edu

[7] Falconer, Aprile 2010, OntoGraf: https://protegewiki.stanford.edu/wiki/OntoGraf

[8] Schekotihin, Agosto 2019, Debugger: https://protegewiki.stanford.edu/wiki/OntoDebug

[9] Horridge, Marzo 2010, OWLViz: https://protegewiki.stanford.edu/wiki/OWLViz

[10] Fariña, Welter, GraphViz: http://www.graphviz.org

[11] Papacchini, Protégé Tutorial: https://cgi.csc.liv.ac.uk/~frank/teaching/comp08/protege_tutorial.pdf

[12] Documento Protégé per gli Object Property Characteristics: protegeproject.github.io/protege/views/object-property-characteristics/

[13] RDF e Turtle, W3C, https://www.w3.org/TR/turtle/

[14] Triple in RDF, https://www.iro.umontreal.ca/~lapalme/ForestInsteadOfTheTrees/HTML/ch07s01.html

[15] Aguzzi, Antoniazzi, Roffia, Viola, Ottobre 2018, SPARQL 1.1 Secure Event Protocol: http://mml.arces.unibo.it/TR/sparql11-se-protocol.html

[16] Aguzzi, Antoniazzi, Roffia, Viola, Ottobre 2018, SPARQL 1.1 Subscribe Language: http://mml.arces.unibo.it/TR/sparql11-subscribe.html

[17] Aguzzi, Antoniazzi, Roffia, Viola, Bisacchi, Bellettini, GitHub SEPA: https://github.com/arces-wot/SEPA

[18] Aguzzi, Antoniazzi, Roffia, Viola, Ottobre 2018, Aguzzi, Antoniazzi, Roffia, Viola, Ottobre 2018, SEPA: http://mml.arces.unibo.it/TR/sepa.html

[19] Ferrari, 2021, Tesi "SPARQL Event Processing Architecture: analisi e ottimizzazione delle prestazioni", https://amslaurea.unibo.it/22391/

[20] Aguzzi, Roffia, Antoniazzi, Viola, Ottobre 2018, http://mml.arces.unibo.it/TR/jsap.html

[21] Introduzione a JSON, https://www.json.org/json-it.html

[22] Roffia, Viola, Bellettini, Aguzzi, SEPABins: https://github.com/arces-wot/SEPABins

[23] Python, sito ufficiale: https://www.python.it/

[24] Python, percentuale di utilizzo: https://www.python.it/blog/2019/3/14/ambiti-di-utilizzo-di-python/

[25] Atom, sito ufficiale: https://atom.io/

[26] SpeechRecognition, download: https://pypi.org/project/SpeechRecognition/

[27] PlaySound, download: https://pypi.org/project/playsound/

[28] Codice Fiscale, realizzato da Fabio Caccamo, download: https://github.com/fabiocaccamo/python-codicefiscale

[29] Tkinter, download: https://docs.python.org/3/library/tkinter.html

[30] gTTS, download: https://pypi.org/project/gTTS/

[31] ICD9-CM, download: https://github.com/sirrice/icd9

[32] Wikipedia, download: https://pypi.org/project/wikipedia/

[33] Iacobelli C., Progettazione tecnologie in movimento, Pioltello, Juvenilia Scuola [2018]

[34] Baldino E. - Iacobelli C., Internetworking Sistemi e Reti, s.l., Juvenilia Scuola [2018]

[35] W3C, Aprile 2015, https://schema.org

[36] National Center for Biomedical Ontology (NCBO), https://bioportal.bioontology.org/ontologies/SNOMEDCT

[37] W3C, Aprile 2015, classe “Medical Entity”, https://schema.org/MedicalEntity

[38] Regione Emilia Romagna, Linee guida tecniche per l'acquisizione, l'adeguamento e l'implementazione della Cartella clinica elettronica, Luglio 2016, https://salute.regione.emilia-romagna.it/assistenza-ospedaliera/file-cci/linee-guida-tecniche-per-lacquisizione-ladeguamento-e-limplementazione-della-cartella-clinica-elettronica-luglio-2016/view

[39] Rosotti, Informatica Medica, McGraw Hill, s.l. [2018]

[40] AGID, progetto ANPR, https://www.agid.gov.it/it/piattaforme/anagrafe-nazionale-popolazione-residente

[41] Antoniazzi, Viola, Bellettini, Aguzzi, Roffia, Maggio 2020, GitHub: https://github.com/arces-wot/SEPA-python3-APIs

[42] Datetime by Python, https://docs.python.org/3/library/datetime.html

[43] "Lutangar", GitHub, https://github.com/lutangar/cities.json

[44] Geeksforgeeks, https://www.geeksforgeeks.org/check-if-email-address-valid-or-not-in-python/

[45] Python Tutorial, Tkinter Combo Box, https://www.pythontutorial.net/tkinter/tkinter-combobox/

[46] Geeksforgeeks, Tkinter Application to Switch Between Different Page Frame, https://www.geeksforgeeks.org/tkinter-application-to-switch-between-different-page-frames/

Indice delle figure

Ringraziamenti

Non sono una persona che spesso fa dei ringraziamenti, ma dopo aver ottenuto questa laurea in questo modo è doveroso fare dei ringraziamenti:

Innanzitutto ringrazio il Prof. Luca Roffia per avermi dato la possibilità di svolgere questa tesi e per avermi fatto conoscere le ontologie e il SEPA. Sono rimasto molto colpito da lei, dalla sua felicità, dalla sua gran voglia di insegnare e di essere stato sempre gentile e disponibile con me.

Ringrazio Elisa, per avermi aiutato a capire e fare le ontologie. Non solo. Grazie immensamente per avermi aiutato nella stesura di questo difficile elaborato, non lo dimenticherò mai.

Grazie Sara, per avermi sempre aiutato, per avermi sempre incoraggiato a non arrendermi mai nel mio percorso universitario e soprattutto grazie per essere stata sempre al mio fianco, nonostante io sia stato poco presente.

Grazie Lorenzo, Mattia e Ion per questi tre anni di università, per avermi aiutato con lo studio e grazie per le tante belle chiacchierate che abbiamo fatto, siete veramente delle persone fantastiche.

Ed infine, ma non per ultimo, ringrazio i miei genitori per avermi sempre aiutato ed incoraggiato a continuare. Grazie.