

Lab 3 Report

CZ3005: Artificial Intelligence

Sebastian

TS5 / U1820661C

1. **Problem Statement**

This laboratory assignment is about creating a sympathetic doctor system that interacts with a patient. According to the document, the patient can only answer yes or no and the doctor should be able to diagnose the patient while taking into account the patient’s mood and pain level. The system should consider at least 5 pain levels, at least 5 different moods, and must have at least 5 diseases, each with 5 or more symptoms.

1. **Assumptions**

To narrow down the problem scope, the following assumptions are made in the solution.

First, for each disease in the knowledge base, only its major symptoms are listed. The term major here means that the symptom must describe the disease. For example, the disease fever will have the symptom “high temperature” or “intense sweat” but not “sad” since it is not relevant enough.

Second, for both the diseases and the symptoms in the knowledge base, there is no vocabulary conflict. In other words, there should be no more than one word that refers to the same meaning. For instance, if “stomachache” is in the knowledge base then “hurting stomach” will not be.

Finally, a user can only have one pain level but may have more than one different mood. It is also assumed that the user's mood will not be contradictory. For instance, if the patient is calm and angry is inquired, the patient will not answer no.

1. **Solution**

IntelliDoc (Intelligent Doctor) is a software that utilizes JavaFX as its front-end and Prolog as its back-end. The software is designed to be extensible, efficient, and deliver a good user experience. Extensible means that new facts can be added with minimal effort. Efficient means that the system diagnoses the patient as fast as possible. Good user experience ensures that the patient should always be informed that the system is working properly.

1. **Design**
   1. **High-Level Architecture**

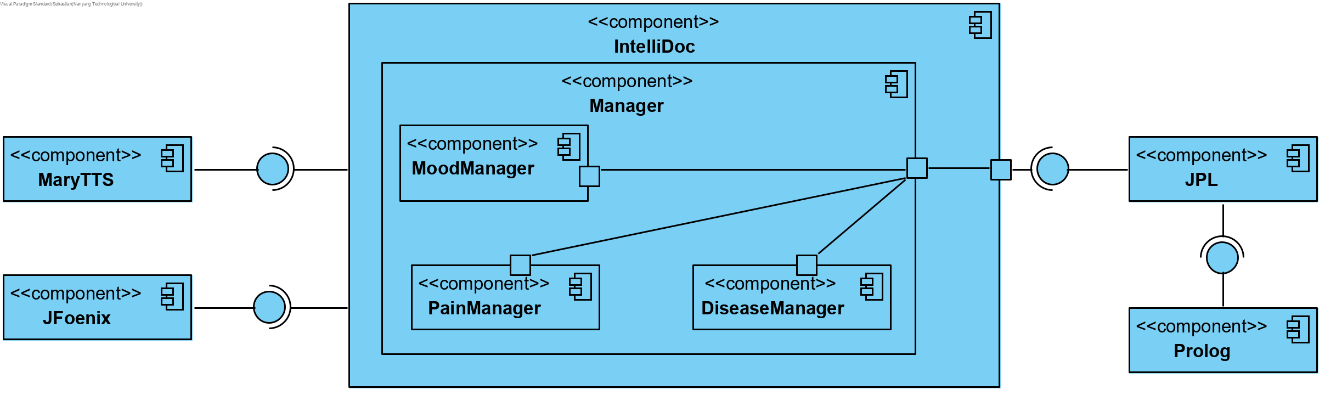


Figure . IntelliDoc's Component Diagram

IntelliDoc’s core feature depends on the JPL library provided by SWI-Prolog. It provides the means of communication between the Java code and the Prolog script. The user interface follows the material design guideline. To support this, the JFoenix library is used. Additionally, MaryTTS library completes the software by providing the text-to-speech feature.

* 1. **Program Flow**

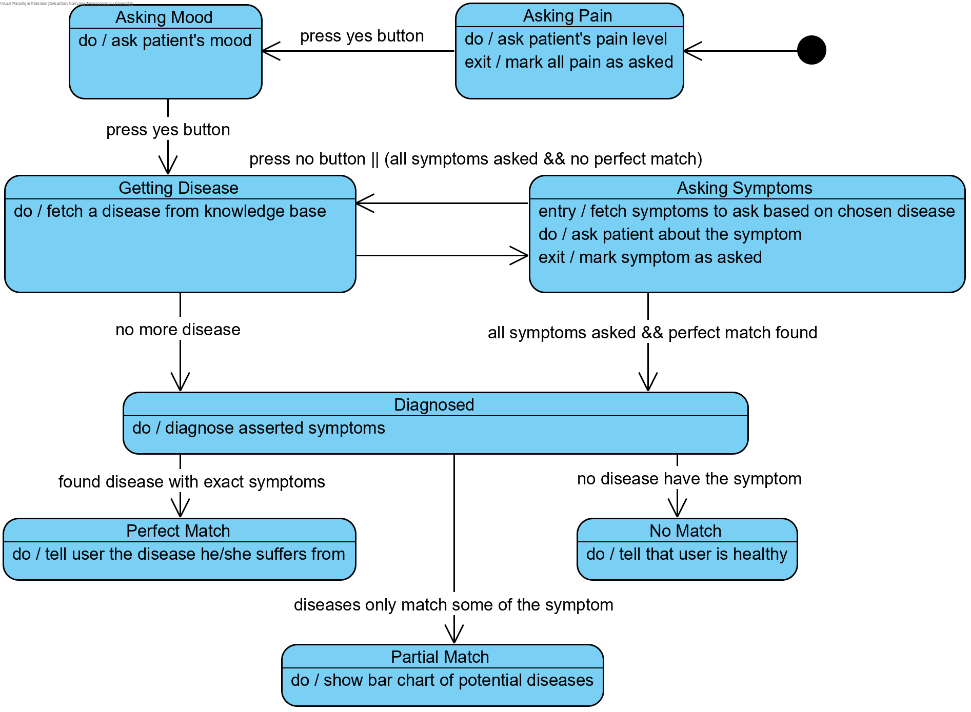


Figure . IntelliDoc's State Machine Diagram

The program starts by inquiring the patient about his or her pain level. Once acknowledged, the mood of the patient is then asked. The program then gets a random order of the disease to ask and fetches the disease one by one. The symptoms of the fetched disease are then asked. If a symptom is not the patient’s symptom, the program will fetch a new valid disease to ask and ask the symptoms of that disease. A disease is valid if each symptom of the disease is either not asked yet or is the patient’s symptom.

The procedure is repeated until the disease is exhausted or a perfect match is found. If a perfect match is found, the program notifies the patient about the disease. Otherwise, a bar chart of potential diseases will be displayed. However, if there are no potential diseases, the program informs the patient that he/she is healthy.

* 1. **User Interface**

|  |  |
| --- | --- |
| Figure 3. Asking Symptom State | Figure 4. Partial Match State |

The program adheres to the material design guideline to improve user experience. The pain level icon and the mood icon together with the list of symptoms ensure the user that the program is working correctly. The user can also choose to enable or disable the text-to-speech feature by modifying the volume slider.

1. **Logic Implementation**

The main logic of the program includes determining gesture, determining questions, diagnosis, and outcome presentation.

* 1. **Determining Gesture**

To determine the doctor’s gesture, the following rule is used.

 Pseudocode 1. Prolog scripts for fetching gesture

To begin with, the program fetches the type of gesture it needs to act, such as a normal gesture or a polite gesture. This is done by the use\_gesture predicate. The rule here states the doctor must use a serious gesture if the pain level is considered as serious pain. Other pain levels will then be considered based on the mood of the patient. After the gesture type is determined, a random gesture is fetched through the predicate random\_member.

* 1. **Determining Question**

Based on the state, the doctor may ask about the pain level, the mood, or the symptom.

* + 1. Determining Pain Level / Mood

To construct the question, the following rules are used.

 Pseudocode 2. Prolog scripts for mood and pain

A question consists of a prefix and a content. The prefix, randomly fetched by fetch\_prefix rule, is concatenated with the content and ended with a question mark. This construct\_question rule handles this construction. The content here refers to either the mood or the pain level to be queried to the patient.

To ask about the mood/pain, the system randomizes the order of the list in the knowledge base. This is performed by fetch\_pains or fetch\_moods. Based on this order, the mood/pain question is built and asked one at a time to the patient.

Finally, the patient’s answer is saved as patient\_mood and patient\_pain.

* + 1. Determining Symptoms

To ask about the symptoms, the program runs two queries simultaneously. One is responsible to pick a disease to be inspected and the other handles the symptom confirmation process.

 Pseudocode 3. Prolog script for choosing a disease

Similar to the previous queries, the program starts by randomizing the order of diseases to ask. During fetching, however, the disease will be validated first by the rule valid\_disease, which will check every symptom by the rule valid\_symptom. If there exists a symptom that has been asked before and is not the patient’s symptom, then it is no longer valid. This is the heuristic used by the program to enhance efficiency. Once the disease has been fetched, the second query executes the following rules.

 Pseudocode 4. Prolog script for asking symptom

To avoid redundancy, the rule fetch\_symptoms will only fetch those symptoms that have not been asked before. The order of fetching is also randomized. After fetch, ask\_symptom will construct the question.

* 1. **Diagnosis**

When the disease library has been exhausted or the symptom list of a disease has been exhausted, the program initiates the diagnosis. The following rules come into play.

 Pseudocode 5. Prolog script for checking the perfect match

The program traverses the list of diseases available and checks if there is a perfect match. The rule count\_match gets the number of matching symptoms while the rule has\_symptoms\_for\_disease determines whether it perfectly matches the patient’s symptoms. If there is no such disease, the program continues by using the following rules.

 Pseudocode 6. Prolog script for partial/no match

The rule count\_symptoms counts how many symptoms match the patient’s symptoms. If there exists a match, the count is saved as a tuple by the build\_tuple rule. The rule build\_estimate depends on the previous rules to build a list of tuples, which is the list of potential diseases, and sort it descending based on the count. The Java component calls the fetch\_estimate rule and draws a bar chart from it. If the result is empty, then the patient is healthy.

* 1. **Outcome Presentation**

To final sentence uttered by the doctor is constructed using the rule outro. The value of Context depends on the result of the diagnosis.

 Pseudocode 7. Prolog script for the outcome

1. **Analysis**

IntelliDoc achieves database extensibility by separating the library for gesture, prefix, mood, pain level, and diseases. Whenever the knowledge base needs to be added, only the specific library is needed to be modified.

The program minimizes the number of questions to be asked by using the heuristic. Diseases that are not valid are skipped. This cuts down a lot of time required if the database is large. However, the trade-off here is the accuracy. The order of disease and the symptom is randomized, and some order may result in a low accuracy unless there is a perfect match.

Suppose the following orders and symptoms are used. Also suppose that the patient’s symptoms are intense sweat, sneeze, and cough.

|  |  |  |  |
| --- | --- | --- | --- |
| Order | Disease | Symptoms (in order of asking) | Validity |
| 1 | Fever | Intense sweat, high temperature, ache | Valid |
| 2 | Cold | High temperature, sneeze, cough | Valid |
| 3 | Asthma | Cough, tight chess, exhausted | Valid |

Now, the symptom “high temperature” for the disease fever is prompted. If the patient answers no, cold will be invalidated although other symptoms have not been asked. The program will come up with the following inaccurate potential diseases.

|  |  |  |  |
| --- | --- | --- | --- |
| Rank | Disease | Matching Symptoms | Match Count |
| 1 | Fever | Intense sweat | 1 |
| 1 | Asthma | Cough | 1 |
| 3 | Cold | - | 0 |

However, recall that it is assumed that only the major symptoms are listed in the database. Therefore, the inaccuracy of the system is still tolerable while maintaining efficiency through the heuristic.

IntelliDoc adheres to the golden rules of interface design by Ben Shneiderman especially full locus of control to the user, informative feedback, and dialog closure. The program responds to the user input rather than instructing the user to do something. The user can enable or disable the text-to-speech feature, and the program responses to the button input. The list of user’s symptoms and the mood and pain icon informs the user that the program acknowledges user inputs. Finally, the user can know that the program has finished when the outcome of the diagnosis is presented to the user.

1. **Conclusion**

IntelliDoc is a sympathetic doctor software that is coded with the Java language and the Prolog language. It is aimed to be flexible to database modification, efficiently diagnose the patient, and provides excellent user experience. The database is flexible due to the loose coupling of the predicates. The program is efficient thanks to the heuristic. The user experience is supported by material design and the fulfillment of the golden rules of design.