# ECHELONBOT REPORT

JASEEM V V

# 1. INTRODUCTION

A chatbot is a type of conversational agent, a computer program designed to simulate an intelligent conversation with one or more human users via auditory or textual methods. Such computer programs can also be referred to as "Artificial Conversational Entities" within some specific contexts. Chatbots may also be referred to as talk bots, chat bots, or chatterboxes.

Much debate has taken place regarding what it may mean to "actually understand". Some claim that the term "to actually understand" is inherently meaningless, because the only available criterion to demonstrate apparent "understanding" is the ability to produce valid responses which superficially indicate an understanding of the conversation being held. The famous Chinese Room argument presented by British philosopher John Searle, maintains that machines in principle may not necessarily require any "understanding" whatsoever to produce meaningful responses.

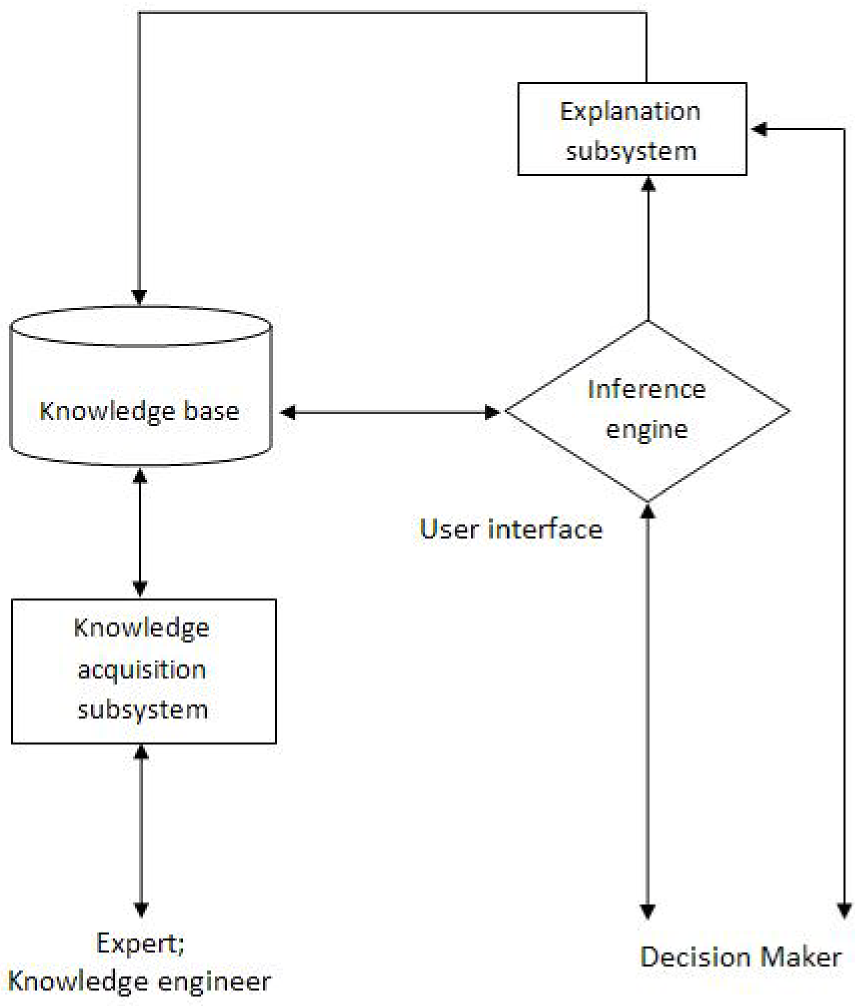
## 2. DESIGN

Natural Language Processing means programming computers to understand language. For example, a bank customer could ask a computer, What is my bank balance? And the computer would respond with the proper amount. The problem with natural language processing is that English is unclear. For example, if a traveler asked a natural-language system, What is the temperature in New York? the computer have to know if the traveler meant the state of New York or New York City. Other English statements, such as We saw the ship with a telescope, can be interpreted in several ways and cause further natural-language processing problems.
A bot can be used as an expert system also. Expert system development is the application of artificial intelligence that is having the greatest impact on the business community.

A bot can be test to know whether it simulates human intelligence by using Turing Test. In Turing Test, a person sits in front of the terminal where he chats with someone on the other end of the system. If he couldn’t distinguish that there other end is a bot or a human and if it’s a bot, then the bot is known to pass the Turing Test.

## General Characteristics of Expert Systems

An expert system is composed of a knowledge base, an inference engine, a knowledge acquisition subsystem, and an explanation subsystem. A schematic diagram of the components of an expert system appears below:



## The Knowledge Base

The knowledge base contains the information and the rules of thumb that the expert systems uses to make decisions. This information should represent high level expertise gained from top experts in the field. In many expert systems, knowledge is represented using rules. An example of a rule is If Mobil Oil stock drops below $150, then buy 1000 shares. Most decisions cannot be based on applying a single rule, how-ever. In the case of buying a Mobil Oil stock, other related questions may have to be asked, such as Do I have enough money? Or Can the money be obtained from selling another stock?

Depending on the nature of the decision the expert system will use different rules. These rules are applied in a different order in different decisions.

## The Inference Engine

The inference engine is the central processing unit of an expert system. The inference engine conducts the dialogue with the user, asking for information and applying it. It uses the knowledge base to draw conclusion in each situation. The structure of the inference engine depends on the nature of the problem and the way the knowledge is represented in the expert system.

## The Knowledge Acquisition and Explanation Subsystem

Most expert systems continue to evolve over time. New rules can be added to the knowledge base by using the knowledge acquisition subsystem. The process of developing an expert system involves building a prototype using a simple problem and continually refining this prototype until the expert system is perfected. As the system matures, new rules may be added and others deleted. The knowledge acquisition subsystem makes these possible.

The explanation subsystem explains the procedures that are being used to reach a decision. In this way the user can keep track of the methods being used to solve the problem and can understand how the decision is reached.

## What is Commonsense Knowledge?

Real expert systems have many facts and rules but even so they contain limited knowledge about quite circumscribed domains.

Much more extensive knowledge will be required for truly versatile, human-level AI systems. And these systems will need to know about many topics that have proved difficult to conceptualize formally. It is perhaps paradoxical that AI scientists find the very subjects that are easy for ten-year-old humans more refractory to AI methods than are subjects that experts must study for years. Physicists are able to describe detailed, exact physical phenomena by wave equations, relativity theory, and other mathematical constructs, but AI researchers still argue about the best way to represent the simple (but very useful) facts that a liquid fills the shape of a cup and will fall out if the cup is turned upside down. The high theoretical characterizations invented by physicists and mathematicians seems easier to formalize than do ideas that, after all, enabled human beings to function pretty well even before Aristotle.

Consider just some of the things that a ten-year-old knows:

If you drop an object it will fall. (Nowadays, a ten-year-old might also know that objects wouldn’t “fall” if dropped in an orbiting satellite).

People don’t exist before they are born.

Fish live in water and will die if taken out.

People buy bread and milk in a grocery store.
People typically sleep at night.

Knowledge of this sort is usually called commonsense knowledge. Typically, knowledge about any subject is spread over a variety of levels – ranging from that possessed by the person-in-the-street to that of the specialist. But the most advanced scientific theories extant around 500 B.C., say, were little more than careful verbal formulations of people's everyday observations. To some the earth was flat, objects fell to the earth because that was "their natural place", and human diseases were caused by a variety of colorful "influences". When knowledge was scarce, little separated everyday commonsense knowledge from advanced scientific knowledge. Commonsense knowledge was (and still is) adequate for many of the things that humans want to do. Scientific knowledge gradually separated itself from commonsense knowledge as people sought more precise description of the world.

## Difficulties in representing commonsense knowledge

What are some of the reasons that it has proved difficult to formalize commonsense knowledge? Probably one reason is its shear bulk. Much expert knowledge can be compartmentalized in such a way that a few hundred or a few thousand facts are sufficient to build useful expert systems. How many will be needed by a system capable of general human-level intelligence? No one knows for sure. Doug Lenat, who is bravely leading an effort to build a large knowledge base of such facts, called CYC, thinks that between one and ten million will be needed.
Perhaps the hardest truth to face, one that AI has been trying to wriggle out of for 34 years, is that there is probably no elegant, effortless way to obtain this immense knowledge base. Rather, the bulk of the effort must (at least initially) be manual entry of assertions after assertions. [Guha & Lenat 1990, p.33]
Another difficulty is that commonsense do not have a well defined frontiers that enable us to get a grip on parts of it independently of the other parts. Conceptualizations of the commonsense world would probably involve many entities, functions, and relations that would crop up diffusely throughout the conceptualization. Thus as we attempt to develop a conceptualization, we wouldn't know whether we had "gotten it right" until we had nearly finished the entire, exceedingly large job.
Another roadblock for formalizing commonsense knowledge is that knowledge about some topic just doesn't seem to be easily captured by declarative sentences. Describing shaped and other physical objects by sentences is difficult. For example, can a human face be described in words so that it can be recognized by another person who hasn't seen it before? How do we use words to capture a tree, a mountain view, a tropical sunset? If something cannot be described in English or some other natural language, there is good

reason to believe that we will not find a conceptualization of it that can be described in logic either.
Related to the difficulty of capturing knowledge in a declarative sentence is the difficulty that many sentences we might use for describing the world are only approximations. In particular, universal sentences (that is, those of the form “All x’s are y’s”) are seldom valid unless they are merely definitions. Various modifications to ordinary logic have been proposed to deal with the fact that much knowledge is approximate.

## Bot Design

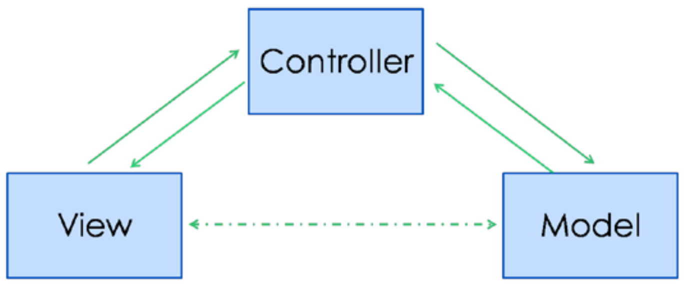
The design of the bot is mainly done with consideration to the Model-View-Controller pattern. Model–View–Controller (MVC) is a software architecture, currently considered an architectural pattern used in software engineering. The pattern isolates "domain logic" (the application logic for the user) from input and presentation (GUI), permitting independent development, testing and maintenance of each.

The **model** is the domain-specific representation of the data upon which the application operates. Many applications use a persistent storage mechanism such as a database to store data. MVC does not specifically mention the data access layer because it is understood to be underneath or encapsulated by the model. Models are not data access objects; however, in very simple apps that have little domain logic there is no real distinction to be made. Also, the Active Record is an accepted design pattern which merges domain logic and data access code - a model which knows how to persist itself.

The **view** renders the model into a form suitable for interaction, typically a user interface element. Multiple views can exist for a single model for different purposes.
The **controller** receives input and initiates a response by making calls on model objects.

The (view) interface consists of an input textarea and and output text area. The user types in the input textarea and the output is displayed in the output textarea.
This application is custom skinned.

MVC Pattern Diagram



## 4.1 Databases

There are several databases for specific purposes. All the databases are named with .db as its file extension. However any custom extensions are allowed.

### CS.db

The main (core) database is the CS.db which stands for common sense database. It contains all the contents that power the brain of the bot. Its saved in the Application Directory itself. It's the directory where the application resides. This database comes along with the application.
Table: cs (id INTEGER, request TEXT, response TEXT)

### Learned.db

This database stores all the information that are taught to the bot by its users. Its saved in the Application Storage Directory. i.e. in XP it's in the C:\Documents and Settings\<user name>\Application Data. This location depends on the OS. This database is created when the user first runs this app.
Table: learned (id INTEGER, request TEXT, response TEXT)

### ShortTermMemory.db

As the name implies, this database acts as short term memory of the bot. It saves the conversations between the user and the bot for every turn with some exceptions. This database is created each time the bot starts. The previous short term database is deleted if any at the creation of new short term memory database. It resides in the Application Storage Directory.
Table: stm (id INTEGER, request TEXT, response TEXT)

### Keyword.db

It's a database that store keywords. This is in addition to the CS.db
Table: keyword (id INTEGER, request TEXT, response TEXT)

### Jokes.db

This is a database that is used to store jokes.
Table: jokes (id INTEGER, joke TEXT)

### Quotes.db

This database contains quotes.
Table: quotes (id INTEGER, quote TEXT)
Jokes.db and Quotes.db are used in interactive mode where bot takes the lead in the conversation.

## 4.2 Logic (Control) Flow

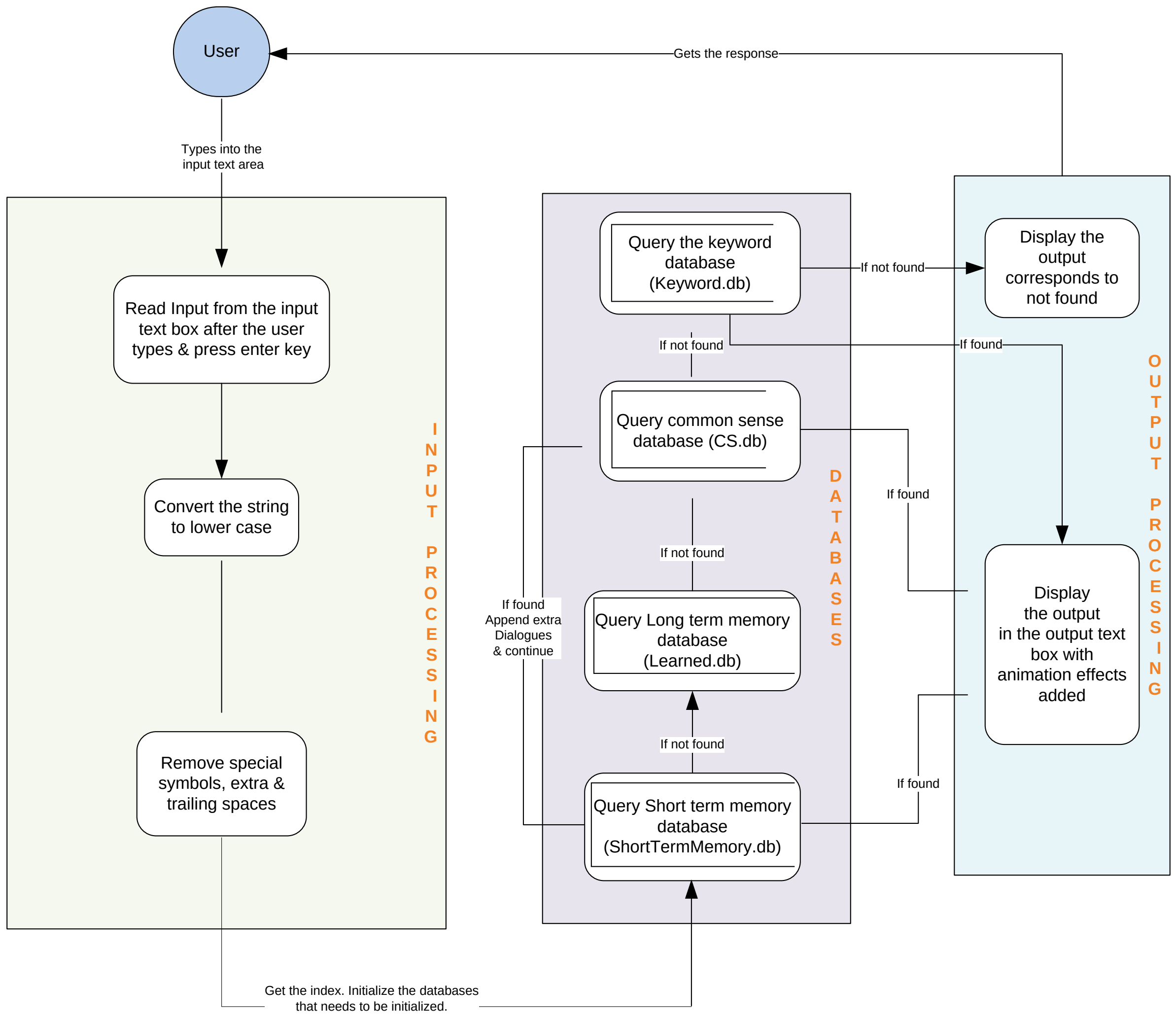
The user types some strings into the input text area, and presses enter key, the entered string is taken and processed. The processing is done in order to remove any unusual, unwanted characters that appear in the entered string. The undesirable characters are multiple spaces, spaces at the beginning of a string as well at the eng, any other symbols. Also the whole input string is converted into lowercases.
The processing is done with the help of regular expressions. The reason for all those is to reduce the redundancy of the data that appears in the database, to make a possible match with the input in order to give a successful reply.
The processed string is considered as the input to the database.

### Normal Mode

The query is first passed to the short term memory. If there is a request field corresponding to the input then the response of the corresponding request is taken. This means the query is repeated. Depending on the frequency of the repetition various dialogues are displayed. If there is not request field that matches with the input then the output returned is null. Then we query the next database which is the Learned.db. The response for which the input matches with the request field in the learned table of the Learned.db database is selected. If there is more than one response then one among them is selected at random and displayed as output.
If there is learned database gives the output as null, it implies that the there is no such data corresponding to the input present In the Learned.db. Then, we query CS.db, which is main database. The same procedure is carried out. If data is not found then the Keyword.db database is queried.
When querying the keyword database the input string is further processed to extract the keywords thus avoiding words like is, was, were, am this, that. If still it fails then the input

is checked whether it is an indication of maintaining state mode (which will be discussed later). If it's not the indication of maintaining state mode then it implies the response isn't found in any of the database. So the bot must now respond with statements like "I don't know", or something similar.
The output is displayed in an animated fashion, which resembles the bot typing.

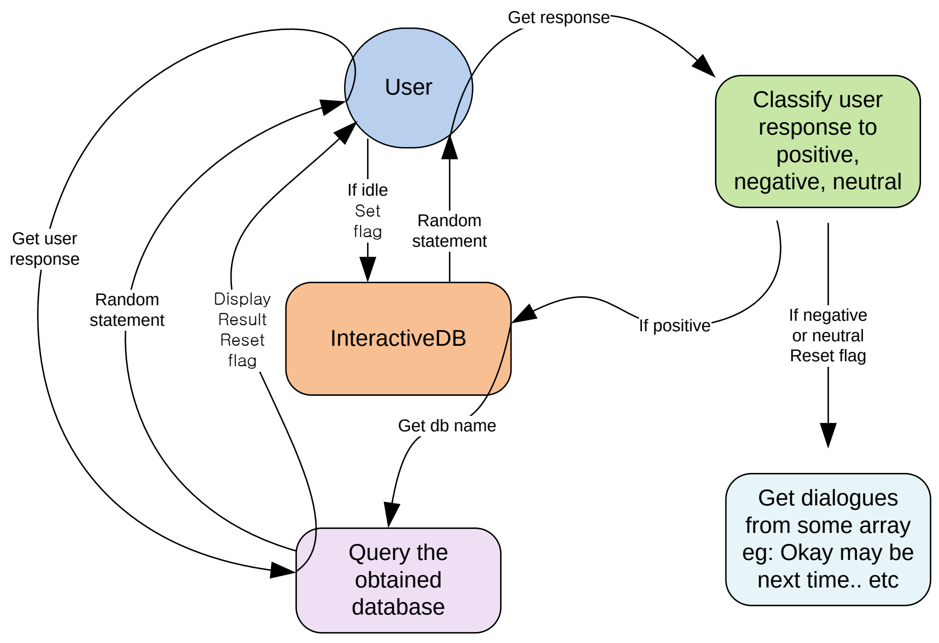
## Normal Mode


User
Gets the response
Types into the input text area
Read Input from the input text box after the user types & press enter key
Convert the string to lower case
Remove special symbols, extra & trailing spaces
INPUT PROCESSING
Query the keyword database (Keyword.db)
If not found
Query common sense database (CS.db)
If not found
Query Long term memory database (Learned.db)
If not found
Query Short term memory database (ShortTermMemory.db)
If found Append extra Dialogues & continue
DATABASES
If not found
If found
If found
If found
Display the output corresponds to not found
Display the output in the output text box with animation effects added
OUTPUT PROCESSING
Get the index. Initialize the databases that needs to be initialized.

## Interactive Mode

That's about the usual case where the user enters the string (query) and the bot replies accordingly. Until now we have the user taking the lead. i.e., first it's expected that the user types in something and the bot replies. More like a one way conversation. So in order to make the bot more interactive, the bot should also take the lead. And when to take the lead is a bit confusing. So it's done in such a manner that if the user goes inactive for a particular period of time, the bot speaks to the user. Here the bot first asks "Are you there?" If the user types anything, that's the indication that the user is present. Now the bot asks the user whether he like to see some jokes or quotes or something similar. The user's response is classified into positive, negative or neutral. If it's positive then correspondingly a joke or a quote is selected at random from the database and displayed. This is the interactive mode.

## Interactive Mode


User
Get response
Classify user response to positive, negative, neutral
If idle Set flag
Random statement
Get user response
Random statement
Display Result Reset flag
InteractiveDB
If positive
If negative or neutral Reset flag
Get db name
Query the obtained database
Get dialogues from some array eg: Okay may be next time.. etc

## Learning Mode

It's actually integrated into the normal query mode. There are three commands that facilitate the bot in learning at runtime. The database can be populated with other independent tools as a part of its learning process. However the bot can be taught on the fly.

The commands are:
$request <statement>
$response <statement>
and
$wrong <statement>

Commands begin with a ‘$’. The $request command is used to enter the request statement and the $response is used to enter the response for the request statement. Its necessary that $response command follows a $request command.

If the user enter a string and bot comes up with some response. Suppose the user doesn’t like that particular response. In such cases he can use $wrong followed by the right statement. So next time onwards the bot will replay as its taught.

## Maintaining State Mode

If the query to all the databases fails, then the bot checks whether the current query is an indication of maintaining state mode. For that the current input is checked for the occurrence of state pattern. Only some of the words like he, him, she, her, it etc are considered.

If state pattern are not found the bot replies as 'I don't know'. Else if state patterns are found then, it enters the maintain state mode.
In this mode the bot takes the previous entered input and queries the database as if it’s the current input and obtains the results. The first word in the first response string of the result is taken as the replacement string. This word is taken and the word matching the state pattern is replaced with this word. Now the query to database is done as usual with the currently modified string. Its output if obtained is displayed or the bot replies with 'I don't know'.

This would be clear with an example.
Let U stand for user and B stand for bot.

**U>** *who is John?*
The response for the corresponding request consists of say 3 responses which are:
*John is one of my programmers.*
*He belongs to my group of authors*
*He is one among who wrote me.*
Of which one will be selected at random and displayed to the user. i.e.,

**B>***He is among one among who wrote me*
which is the output.

Now user asks ->
**U>** *where is he?*

This particular query is state dependent. We can't tell where a person is without knowing which person the 'he' refers to. In usual cases order to know about whom the talk is about, we check the previous query. However, the bot checks previous query with only a depth of one.

Since the current query is not present in any databases, and since there is an occurrence of state pattern, it enters state mode. Here the previous query is taken, which is 'who is John?' and a query is done with the databases as normal and the three outputs will be obtained. Of those the first word of the first output string 'John' is taken as the replacement word. Now the actual current input string is taken the state pattern 'he' is replaced with the word John.

Now the modified string is:
*where is John.*
whose response can be taken from the database, say the response to that is 'John is sleeping'.
This is a little bit tricky. So in order to get the desirable result the data in the database must also be inserted with these considerations in mind.

The output displayed will be
**B>** John is sleeping

Whole process is transparent to the user. In effect the query goes like:
**U>** *who is John?*
**B>***John is one among who wrote me*
**U>** *where is he?*
**B>** John is sleeping

Thus the bot do maintain the state of the conversation.

# 3. IMPLEMENTATION

The runtime environment of choice is Adobe AIR since its aimed as a desktop application with access to SQLite databases. Adobe AIR supports programming in variety of languages though ActionScript is its language of choice. Thus the bot is implemented using ActionScript 3.0 programming language.

As told before this application is designed with Model-View Controller (MVC) design pattern in mind. So the classes are divided accordingly.
This project is hosted at Google Code http://code.google.com/p/echelonbot/. Hence the package used is com.google.code.echelonbot to avoid conflicts.

The class files are:
AvoidList.as
DBController.as
CSDB.as
LearnedDB.as
KeywordDB.as
InteractiveDB.as
ShortTermDB.as
Shared.as
Update.as

The main file is the Echelon.mxml file since it's a flex project. Here, the mxml file implements the view i.e. the interface of the Echelon Bot.
The view is consisting of two text Areas one for input and the other of output. Its a custom skinned application, which looks more like a widget. Cascading Style Sheet (CSS) is used to separate the styles of the application and its components. That makes code more organized and is stored in file Styles.css.
The input text area gets the input, and does the processing. Then its send to the Controller which is the class DBController. Classes end with .as as the extension. The connections to the databases are asynchronous in nature.

**AvoidList** is the class which initializes the file avoid_list_STM.dat which is the text file containing words which when occurred in the input, should avoid populating the short term memory with the current query. This class checks whether the words in the file is present in the current query (input).

**DBController** is the controller. It controls the interaction among various databases (models) and the view. It gets the processed input from the view i.e the Echelon.mxml which is the root. And sets the output to the view. It sets the input to various databases for querying them and gets the queried output.

**CSDB** is the core database of the Echelon bot. It called the Common Sense database. It contains various methods to operate on the CS.db database. It contains only SELECT query since manipulating the CS.db at runtime isn't encouraged.

**LearnedDB** is the database which stores the data (i.e. the request and the response) that the user had taught the bot. It contains various methods to manipulate and operate on Learned.db database. It has methods that consist of CREATE, INSERT and SELECT queries. The statements from the $request, $response, $wrong command are inserted into the Learned.db accordingly.

**ShortTermDB** is used to maintain a short term memory. It's used to detect repeated strings and input history. Not all the input is inserted into this database. Inputs like yes, no etc are exceptions. The inputs that needed to be avoided are determined with the help of AvoidList class. When the output is displayed the input and the displayed output are entered to this database in normal situations.
There is the ability to forget the inputs depending on the number of turns of the conversation. If 12 complete turns occurs then the first 4 rows are deleted. This class contains methods with CREATE, INSERT, SELECT, DELETE queries.

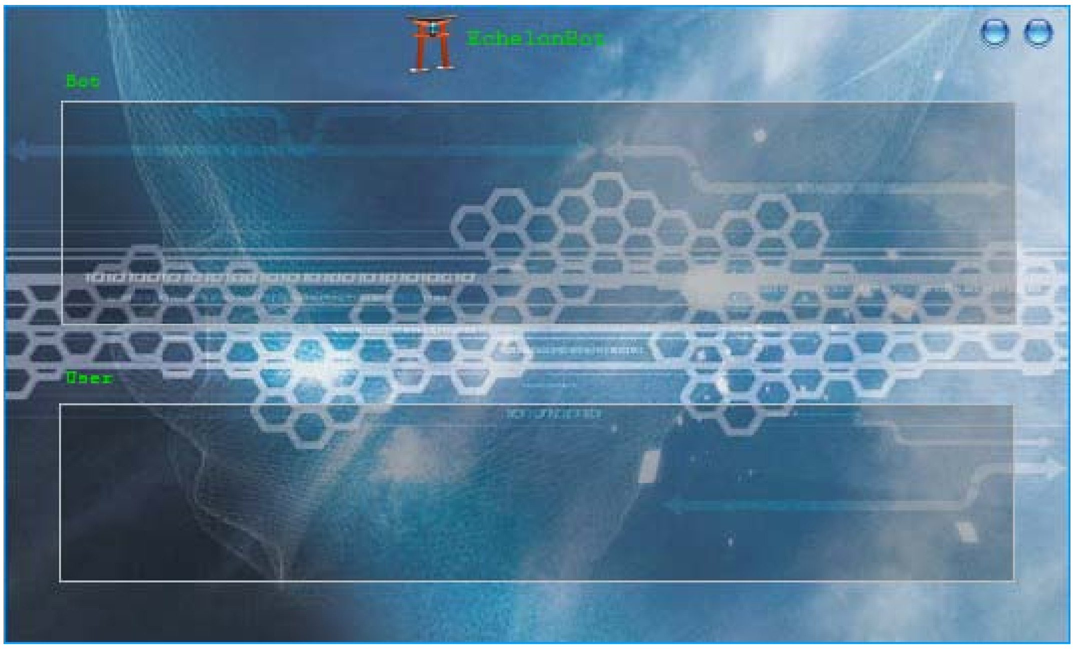
**KeywordDB** contains strings with keywords only. The words like is, was, were, am, this, that are stripped from the input and is queried. This is to increase the possibility of getting the response for the given query. It contains methods with SELECT query.

**InteractiveDB** is used to query the database Quotes.db, Jokes.db. It takes the database name and selects joke or quote from them.

**Shared** is a class containing methods that are used by many other classes. It consists of error handling, random number generation and input classification into positive, negative, neutral.
Most of the classes use getters and setters for getting and setting data.

**Update** is a class which handles automatic updates for this AIR app. It checks each time whether a new version is released using the XML file hosted at the google code. If so it notifies the user that a newer version of the app is available and whether to update. If the user presses yes then the latest version is downloaded and installed automatically. The checking for newer version takes place in the background without user intervention.

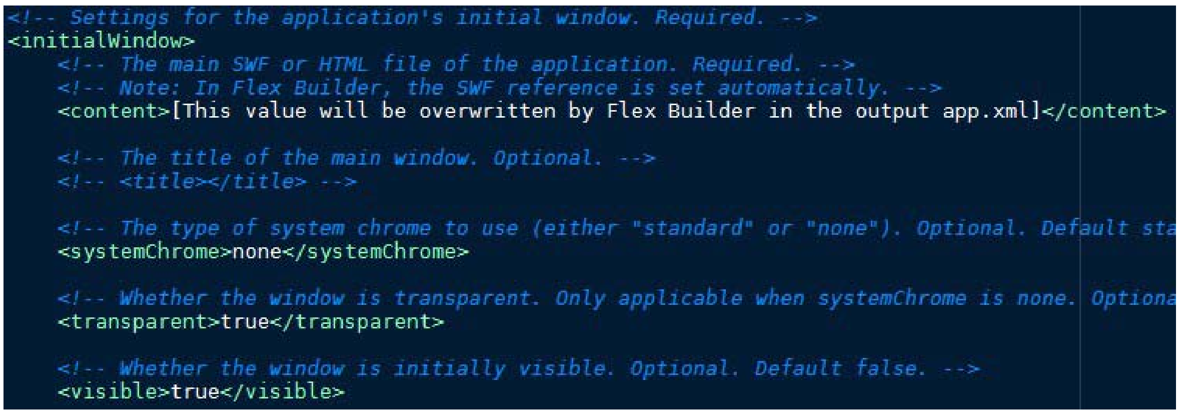
## 5.1 View – Custom Skinned



This is skinned. There is background image given. The minimize and close button are different and there is no system chrome.

So inorder to give it a custom chrome we edit the tag to <mx:Application>. Actually this is the tag for a web application. By removing the <mx:WindowedApplication> tag the properties of a window are lost. i.e, we can't drag as well as no window control buttons like close, minimize etc. (Since for a web application there won't be such control button). So we have to add those functionalities.

### The Application Descriptor File



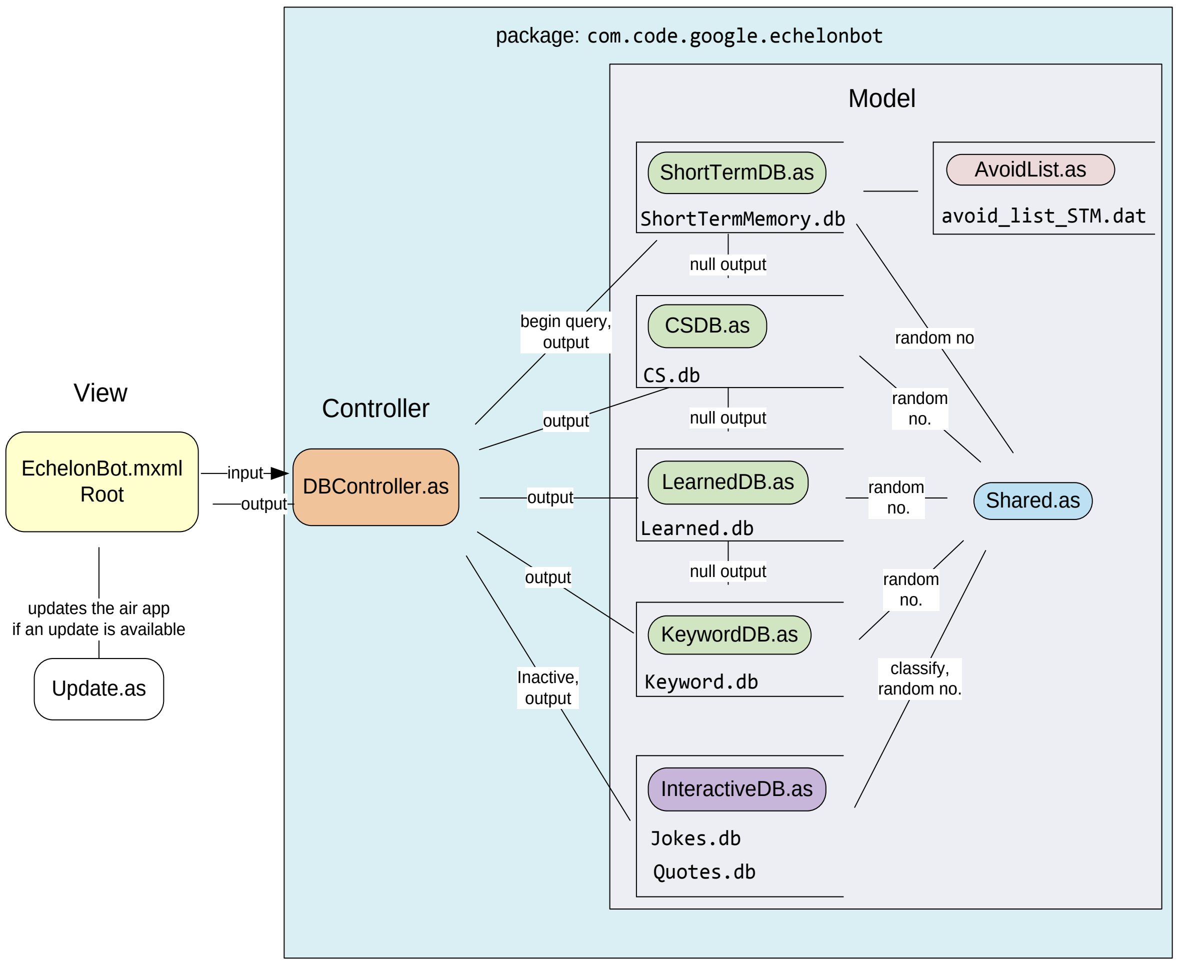
```
<!-- Settings for the application's initial window. Required. -->
<initialWindow>
    <!-- The main SWF or HTML file of the application. Required. -->
    <!-- Note: In Flex Builder, the SWF reference is set automatically. -->
    <content>[This value will be overwritten by Flex Builder in the output app.xml]</content>

    <!-- The title of the main window. Optional. -->
    <!-- <title></title> -->

    <!-- The type of system chrome to use (either "standard" or "none"). Optional. Default sta
    <systemChrome>none</systemChrome>

    <!-- Whether the window is transparent. Only applicable when systemChrome is none. Optiona
    <transparent>true</transparent>

    <!-- Whether the window is initially visible. Optional. Default false. -->
    <visible>true</visible>
```

We also need to change the application descriptor to reflect these. We need to change the systemchrome tag value standard to none, enable transparent and make it visible.

### Custom Chrome Window Controls

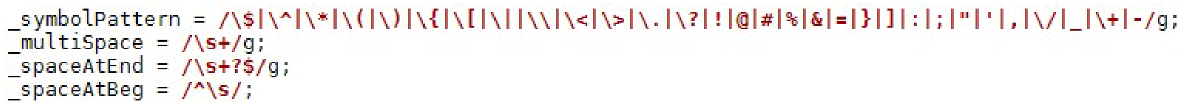
```
private function drag(event:MouseEvent):void
{
    this.stage.nativeWindow.startMove();
}
private function minimize(event:MouseEvent):void
{
    this.stage.nativeWindow.minimize();
}
private function close(event:MouseEvent):void
{
    NativeApplication.nativeApplication.exit();
}
```

## 5.2 Interaction of the Model, View and the Controller



## 5.3 Processing Input

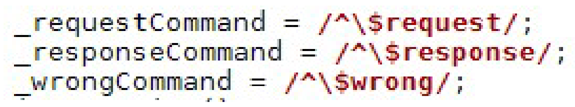
It’s done using Regular Expressions. The pattern is as shown below. Extra spaces between words, at beginning and end are removed. Special characters are also removed.



```
_symbolPattern = /\$|\^|\*|\(|\)|\{|\[|\||\\|\<|\>|\.|\?|!|@|#|%|&|=|}|]|:|;|"|'|,|\/|_|\+|-/g;
_multiSpace = /\s+/g;
_spaceAtEnd = /\s+?$/g;
_spaceAtBeg = /^\s/;
```

## 5.4 Commands for Learning

The Regular Expression patterns for the commands are as shown below:



```
_requestCommand = /^\$request/;
_responseCommand = /^\$response/;
_wrongCommand = /^\$wrong/;
```

## 4. CONCLUSION

Textual bots can simulate AI behavior only to a limited extend.
“A picture speaks a thousand words” is very true here. Since this bot is textual based and uses declarative statements to represent knowledge it hard to represent everything since the details are simply overwhelming.
Humans think in pictures. So in order to effectively represent the intelligence, there must be some mapping between pictures, words, and actions (signals).
I suppose that’s what is done in Singularity University.

## 5. REFERENCES

[1] Programming Adobe® ActionScript® 3.0 for Adobe® Flash® *Adobe Systems Inc. 2008*

[2] Programming Adobe® ActionScript® 3.0 for Adobe® Flex® Adobe *Systems Inc. 2008*

[3] Developing Adobe® AIR® 1.5 Applications with Adobe® Flash® CS4 Adobe *Systems Inc. 2009*

[4] *http://livedocs.adobe.com/flash/9.0/ActionScriptLangRefV3/*

[5] A Two-Stage Bayesian Network for Effective Development of Conversational Agent
-Jin-Hyuk Hong and Sung-Bae Cho

[6] Regular Expressions Cheat Sheet v2

[7] *http://www.a-i.com*

[8] Management Information Systems The Manager's View – Robert Schultheis , Mary Sumner *Tata McGraw Hill*

[9] Artificial Intelligence A new Synthesis - Nils J. Nilson *Elsevier*