2014

# VUI Design

Corey Haskell
*Grand Valley State University*

2014

# VUI Design

## CIS 693

This project examines voice-based user interfaces (VUI) for information systems and the design challenges that they present. The architecture of a voice activated information system is complex and understanding human speech is no small task for a machine. However, the ability to identify the spoken word and produce text is largely a utility today; available freely in open source projects or baked into popular platforms. Thus, software developers find themselves with the means to build into their solutions another dimension of input mechanisms should they be able to wield it. To explore the design challenges that must be overcome once an application has finished listening, this project relies on two experiments to provide insight into how domain and context shape natural language understanding and drive human interactions by a machine. In the process, implications for future systems and potential interactions will be uncovered and discussed.

Corey Haskell
Grand Valley State University
4/15/2014

# Table of Contents

This project examines voice-based user interfaces (VUI) for information systems and the design challenges that they present. Typically, design decisions and challenges should be considered against the domain, contexts, and architecture of an application. Voice User Interfaces are no different. These elements can be relied upon to provide developers with both a set of tools and constraints from which usable applications must emerge.

Human nature adds some interesting twists to these truths, of course. To be voice enabled is to be engaged in a dialog with an unpredictable being who has been conditioned by the human experience to expect a reasonable result from a reasonable request.

## Experiments

To work with voice recognition and study how architecture, domain, and context all come together to impact the design of an application, this project included two experiments. The theme going forward will be to weave examples and lessons learned into the discussion about usable voice activated systems. They are different in terms of graphical footprint as well as the chosen underlying frameworks. To begin using them as a guide, they are introduced below.

### Battleship

This experiment is a virtual take on the classic board game. Two players target each other's fleet calling out coordinates on a grid. Hits and misses are kept track of as each strike is called out until one of the player's ships have all been destroyed.

Battleship provides this project a traditional GUI based application with specific rules and boundaries that users must adhere to. To voice-activate Battleship is to allow the user to carry out the mission with their voice just as they would with an actual board game. This blending of GUI and VUI was chosen to further explore how an application should transition between modes (multimodality).

### Prototype

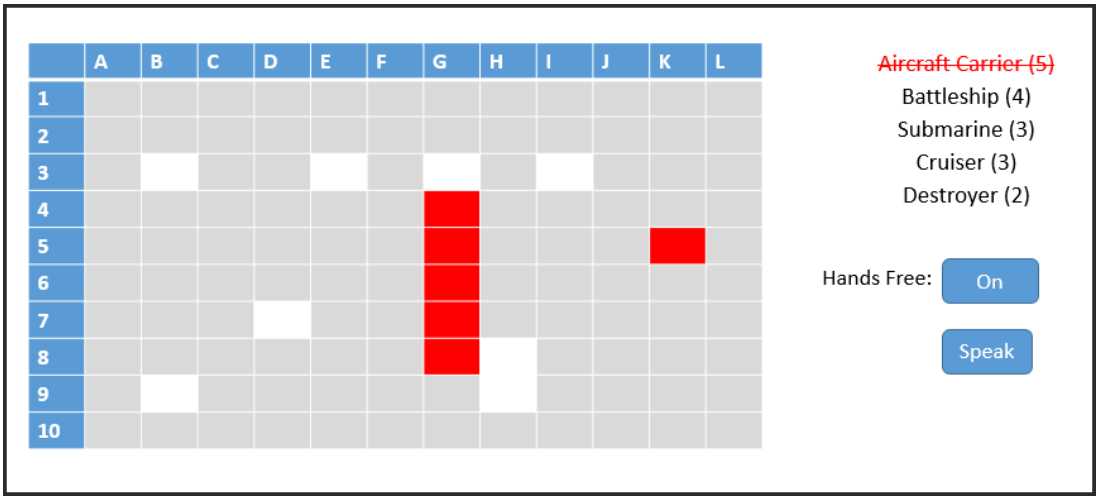


**Figure 1: Battleship Game Board**

## Domain

With Battleship, this project is very finite in regards to what actions it can perform. The rules of the game and the game artifacts makes it possible to model the domain quite specifically:

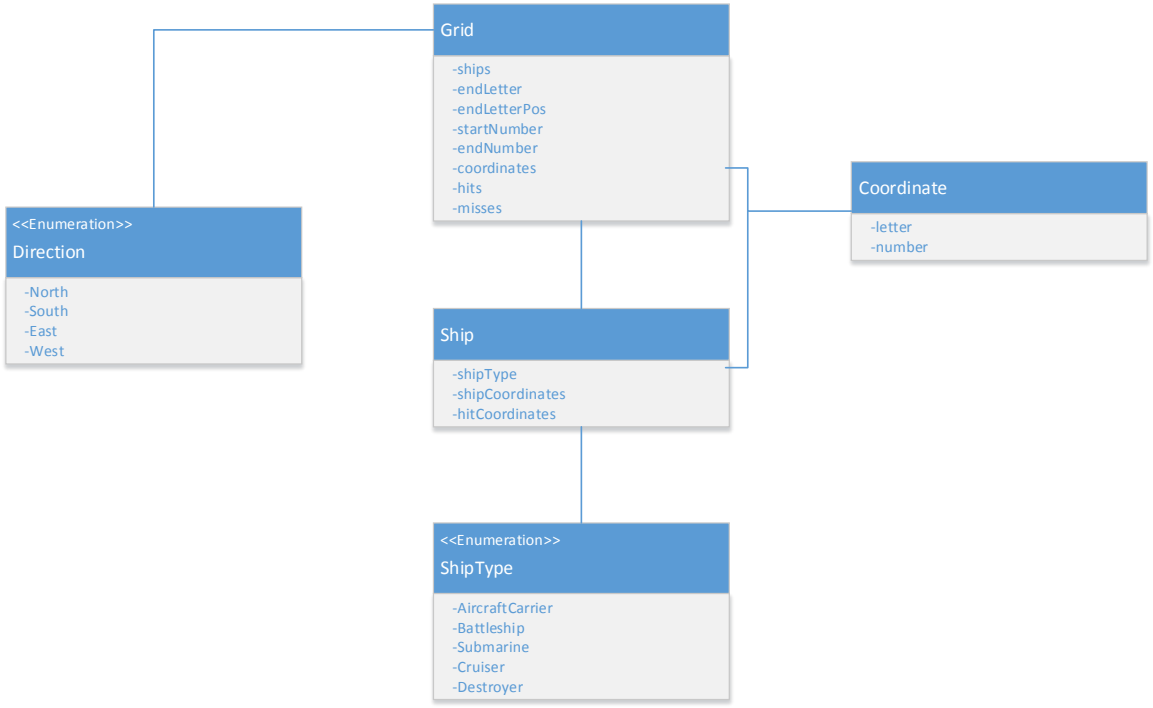


**Figure 2: Battleship Domain**

This graphically represents the following relationships:

1. A Grid has Coordinates
2. A ship takes up coordinates on a Grid
3. The ship must fit on the grid in a given direction (North, South, East, West)
4. A Ship has a type, which is based on how many coordinates it takes up

## Contexts

The contexts that should be considered for our design are:

1. The user is trying to play the game
2. The user is trying to manage the game
3. The user is in hands free mode
4. The user is in tactile mode

When the user is playing the game, it is reasonably safe to assume that they will be delivering coordinates on the grid that they wish to strike. The application will need to deliver back the impact on this strike to the opposing player's grid.

When the user is managing the game, they will be looking to deliver changes to the state of the game to the system. For instance, the user may desire to reset the game grids or exit the application. The application will need to perform the desired change.

## Jarvis

With Jarvis, the goal was to create a virtual assistant who could perform actions on a device without the user having to hunt and peck for applications or provide tactile inputs such as with a mouse or finger. Jarvis was selected as a concept to explore how the lack of a GUI can impact the usability of a voice activated system.

The lack of an application specific GUI was an important area to explore because the automated assistant is quickly moving from science fiction to reality. Whether we are in a blue tooth enabled GM vehicle trying to use hands free calling or talking to Siri on our iPhone or telling our Xbox 1 that we want to watch Breaking Bad on Netflix, we are already talking to someone we can't see. An invisible application to manage other applications is something most developers will be interfacing with very soon if they already aren't.

## Prototype

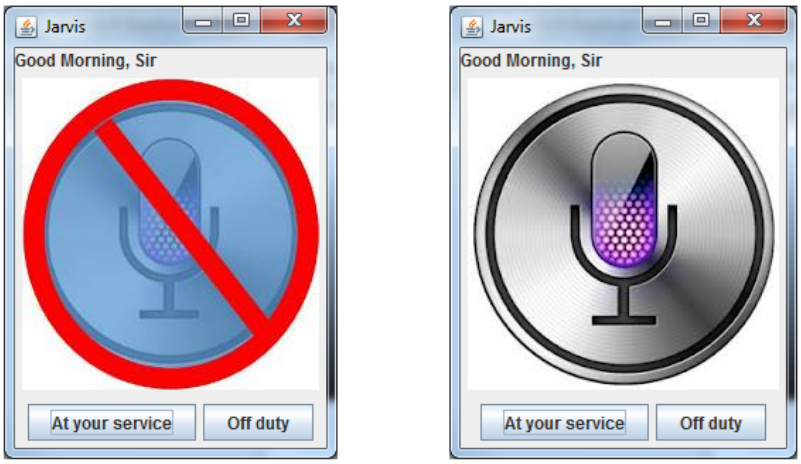


**Figure 3: The UI For Jarvis Both On and Off Duty**

## Domain

Jarvis does not contain an application specific domain in the way that Battleship does. Because Jarvis exists to perform tasks in the environment on which it is installed, the application will not perform actions on a virtualized object living in and created for the Jarvis application. From a design standpoint, this means taking a more abstract approach about what probably exists on a desktop (other applications, files, etc.).

### Contexts

With the limited scope set at opening and closing applications, there are two things that Jarvis should be able to allow the user to do:

1) Open or Close an application
2) Confirm that the user really wants to close an application

## Architecture

### Overview

The architecture of modern voice recognition capable applications can generally be described with five categories or areas of concern: Endpointing, Feature Extraction, Recognition, Natural Language Understanding, and Dialog Management.

Endpoint, Feature Extraction, and Recognition can be thought of as the ears of the application. Together, they tell the application what to listen for and how long to listen for it. These are critical tasks because there is a very low probability of the downstream architecture being effective if background noise skews the result of a spoken utterance or the complete utterance is not obtained.

These first three layers are generally available to developers as utilities embedded in frameworks and platforms; both commercial and open source. Therefore, the focus of this project architecturally is to expand on Natural Language Understanding and Dialog Management. Brief definitions of the other categories will be provided, but the gory details of capturing an utterance are secondary to understanding and action for most developers.

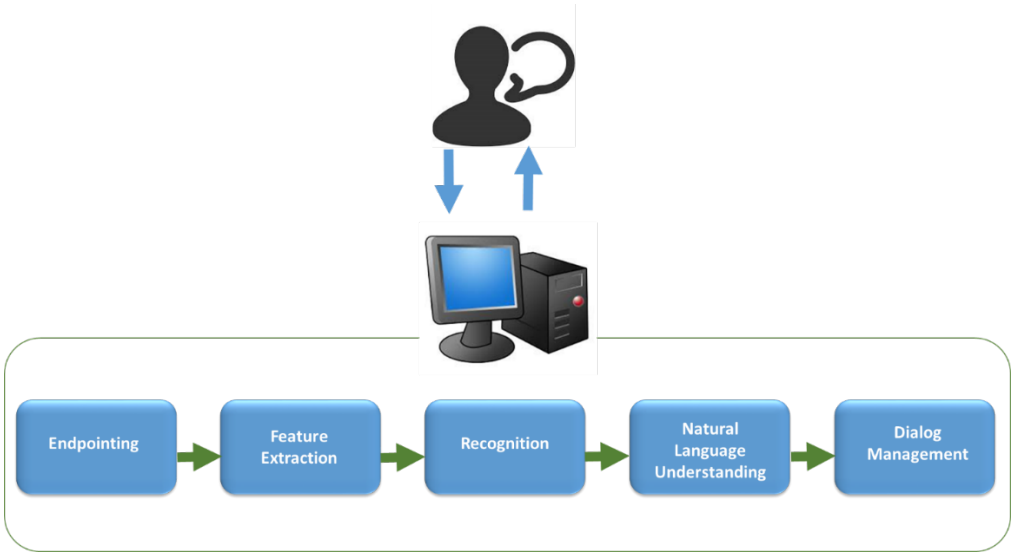


**Figure 4: Overall Architecture**

### Endpointing

Endpointing is the term used to refer to the part of the system that detects the beginning and the end of a user's utterance. Once the system is ready to receive a command, it listens for the callers input. The

endpointer, determines when the waveform (vibrations and frequency) representing the user has paused for an acceptably long enough time to conclude that the user has finished talking.

## Feature Extraction

Once the user's waveform has been captured, a voice activated system will transform the utterance into a list of numbers representing measurable characteristics of speech that are useful for recognition. This type of list is known as a feature vector. Often, systems will create a vector for each small time period in the waveform.

## Recognizer

The sequence of feature vectors are used by a recognizer to determine that words that were spoken by the user. A recognizer uses a model that represents all the possible words a user may say. It takes the incoming feature vectors and analyzes them against the model and outputs the best matching word string. It does this by searching the recognition model; the basic, high level elements of which can be depicted as:

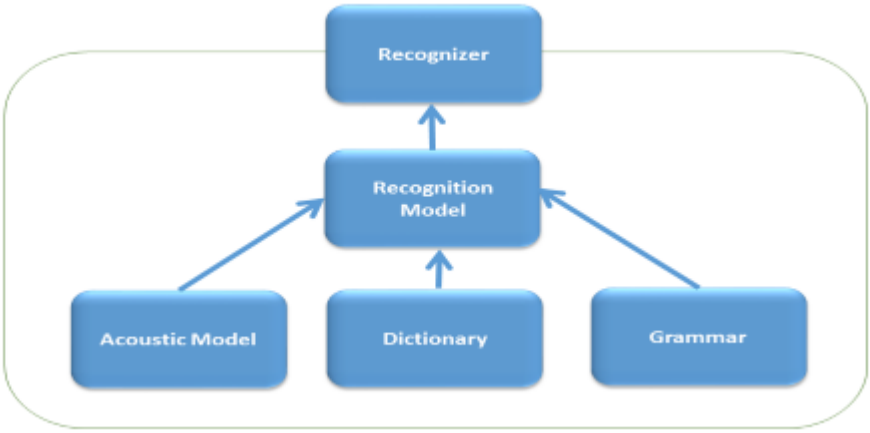


**Figure 5: Typical Recognizer Components (From Cohen, Giangola, and Balough)**

An acoustic model is an internal representation of the pronunciation of possible sounds and the dictionary is a list of words mapped to their pronunciation. The grammar is a definition of all the things that a user can utter to the system and expect an understanding to occur.

There are actually two different types of grammars: a rule based approach and a statistical language model (SLM). The advantage of an SLM is that is allows more freedom to the user with what they are allowed to say. Where the rule based approach defines what can be said explicitly, the SLM calculates the probability of a word occurring in a given context.

## Chosen Frameworks

For each experiment in this project, a recognizer had to be chosen that would provide the services needed to transform the spoken word into a string representation. Furthermore, the recognizer had to be available using a programming language that would run on the device in question.

### Battleship

Battleship was developed for devices running Google's Android operating system; specifically for a tablet. The tablet presents an interesting platform for this project because of its ubiquitous nature translates into diverse operating scenarios. For instance, use in a library will not lend itself well to a verbal interaction while tactile input presents a challenge on the go, such as on a treadmill or while driving. Furthermore, the tablet means that the operating environment may shift during the course of the user experience. The game may start in a quiet area and the user may take the tablet on an errand.

Android itself does not come with the ability to recognize speech. It does come with the next best thing, however; the ability to ask another application to recognize it. In Android development terms, an Activity is a single, focused thing that a user can do. The Activity is the parent of all things that a user can do in an application and for the purposes of this project allows interaction with an app called "Google Voice Search."

When Battleship is deployed to an Android device that has "Google Voice Search," it can use an intent (or message) to ask it to perform an Activity that informs users they can speak. When the user has finished, the dialog disappears and Battleship can receive an array of strings with the recognized speech.

```
private void listen() {
    Intent i = new Intent(RecognizerIntent.ACTION_RECOGNIZE_SPEECH);
    i.putExtra(RecognizerIntent.EXTRA_LANGUAGE_MODEL, RecognizerIntent.LANGUAGE_MODEL_FREE_FORM);
    i.putExtra(RecognizerIntent.EXTRA_CALLING_PACKAGE, this.getPackageName());

    recognizer.startListening(i);
}
```

**Figure 6: Battleship Method For Listening**

```
@Override
public void onResults(Bundle results) {

    ArrayList<String> thingsYouSaid =
            results.getStringArrayList(SpeechRecognizer.RESULTS_RECOGNITION);

    String command = thingsYouSaid.get(0);

    String fixedCommand =  VoiceAutoCorrect.getInstance().fixUtterance(command);

    processSpeech(fixedCommand);

}
```

**Figure 7: Battleship Method For Processing Recognizer Results**

The code above and all logic needed to virtualize the game were written using the Java programming language. Interestingly, direct access to a grammar was not available using this approach. These artifacts were tested using jUnit. Everything was built inside of Eclipse.

## Jarvis

Jarvis was built using the Sphinx4, an open sourced recognizer from created as a joint collaboration between the Sphinx group at Carnegie Mellon University, Sun Microsystems Laboratories, Mitsubishi Electric Research Labs (MERL), and Hewlett Packard (HP), with contributions from the University of Santa Cruz (UCSC) and the Massachusetts Institute of Technology (MIT).

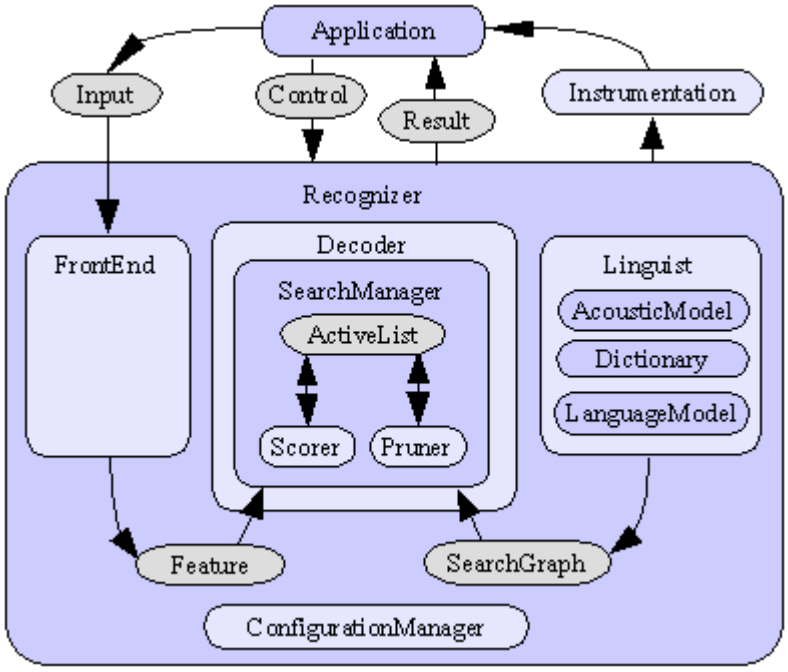


**Figure 8: Sphinx4 Architecture**

Most of the above architecture for Sphinix4 needed to be configured in an XML file. This includes the behavior of the recognizer when it is called. The developer is mostly responsible for defining the grammar and interfacing with the recognizer.

```
// loop the recognition until the programm exits.
do {
    myWindow.changeDisplay("Hello, Sir. How may I help you?  Say a command...");
    System.out.println("Start speaking. Press Ctrl-C to quit.\n");

    Result result = recognizer.recognize();

    if (result != null) {
        String resultText = result.getBestFinalResultNoFiller();

        System.out.println("You said: " + resultText + '\n');
```

**Figure 9: Jarvis working with the Sphinx4 Recognizer**

Sphinx-4 allows developers to leverage the Java platform. And, once again, the application logic necessary for Jarvis to perform its functions as well as the logic to work with the recognizer were written in Java.

# Natural Language Understanding

## Overview

This is the part of the architecture where things start to get interesting for the majority of developers. Up to this point, reliance on a framework or platform has been both possible and desirable. After a recognizer has returned a string representation of a spoken phrase, meaning must be assigned to the words. Practically speaking, this typically means finding a way to pick off only the parts of the phrase that is relevant in a given domain and context. There is no single way to accomplish this, but the type of recognizer used will likely impact the solution.

## Considerations

There are two common issues that can plague recognizers: Ambiguity and Deviation. Developers must take care to be aware of how their recognizer will respond in these scenarios lest they want users to fail at a task for no good reason.

### Ambiguity

Ambiguity provides a special challenge with regards to natural language understanding. For a well-known example, consider the following two sentences:

1) “Wreck a nice beach”
2) “Recognize speech”

Though they look very distinct in print, they can sound remarkably similar when spoken. This problem tends to be compounded in noisy environments or when encountering users who lack a crisp cadence with the language in question (think soft spoken, quick pace, accents, etc.).

The Battleship experiment provides another classic example of the havoc ambiguity can wreak. In the course of game-play, the end user must provide the coordinates where they would like to "fire." The possible values reside on a graph where the alphabetic letters A-L make-up the X-axis while the numbers 1-10 compose the Y-axis. The user is expected to utter the X-Y position, such as "A8."

The first issue in this example is that the recognizer is just as likely to return a word as it is a letter and number combination. User acceptance testing on early prototypes of Battleship showed that results such as "aye-ate" or "eight-eight." The use of the "Google Voice Search" recognizer added the additional wrinkle of the recognizer being tuned as an assistant program along the lines of Siri or the Jarvis experiment. Therefore, something like "F4" will often come back as "App Store."

The second issue that letters of the alphabet can sometimes sound like or rhyme with each other or a number. The "A8" example above often returned "88" or "eight-eight" or "ate-ate." The letters "B" and "D" as well as "A" and "H" were interchanged.

As a result of the Google recognizer being wide open due to the grammar being off limits, the application had to be prepared for any combination of these responses. This meant application logic to massage the utterances into the context of the game. Here, it was a good thing that the domain was a game with specific rules and narrow artifacts (grid, ship, coordinates). Without this limited domain, narrow context, and the GUI to facilitate the flow of the game the amount of application logic needed could have spiraled wildly out of control.

The use of the grammars in the Jarvis experiment provided a degree of protection from ambiguity because the possible phrases never got large enough to include ambiguous paths. Anything that sounded like "Jarvis open word" would match to the appropriate path in the grammar and return [Jarvis], [open], and [word]. Anything out of context, would have returned an unknown, or blank, result.

Designing Jarvis with a different grammar for each context will allow some scaling to occur. Because the program will only match again the task grammar when waiting for a command and will only match against the confirmation grammar when double checking, the fact that "open word" and "yes sir" could be ambiguous doesn't really matter. There is only one path to choose thanks to context!

### Deviation

Deviating from the script is also a point of concern. While it seems simple enough to program an application to perform a requested transaction, human beings have a special knack for breaking the rules of a verbal transaction. This is usually due to our tendency to anthropomorphize. Without sufficient forgiveness or planning, the quirks of cultures can trip up an information system looking to match on too specific inputs.

For example, a southern telephone company was surprised to learn that their automated phone system had a significant failure rate when the system posed the question "Is that correct?" Though the question seems straightforward and unambiguous, the issue was that instead of most callers simply saying "Yes", "No", "Correct", "Incorrect", or some other synonym of the like, they would respond with "Yes, ma'am" or "No, ma'am." The designers of the application had not considered some of the unique elements of southern speech and hadn't programmed the often used "ma'am" into the system (Kotelly).

For the Jarvis experiment, this problem could be solved with the configuration of the grammars. There are two different types of grammars: a rule based approach and a statistical language model (SLM).

Here, Sphnix-4 is an HMM-based recognizer (Hidden Markov Models); which is a type of statistical language model. This is advantageous because optional phrases, such as Ma'am, can be included to give the user more freedom with their speech.

```
public <workProgram> = <ask> <action> <program>;

<ask> = [please | could you ];
<program> = (Word | Excel | out look | Note pad | power point );
<action> = (start | open | stop | close | kill | shut down );
```

**Figure The Application Grammar From Jarvis**

Above, is the grammar used in the Jarvis experiment when the application is expecting the user ask for an action to occur on the desktop. The grammar used by Sphinx-4 is based on the JSpeech Grammar Format (JSGF) which is a platform-independent way of describing one type of grammar. It uses a text based syntax that is understandable by both human developers and computers. The <workprogram> declaration is a rule that will contain up to three parts:

1) Possible interrogative utterances
2) A program to work with
3) The action desired for the program

Notice that the possible utterances for <program> and <action> are surrounded by parenthesis while the <ask> possibilities use square brackets. This is the manner in which JSGF syntax is used to distinguish between required parts of a phrase and optional or possible parts. The name part of the phrase is required at the very beginning so we can be sure that the user is purposefully addressing it. The assumption that the user may show the application the same type of pleasantries as they would an actual human assistant is included next. It is not required to understand user intent in this context, but this known human tendency should not be allowed to cause the program to misunderstand the user.

With Jarvis configured in this manner the recognizer could return the following six string values when asked to start Microsoft Word:

1) Please start word
2) Could you start word
3) Start word
4) Please open word
5) Could you open word
6) Open word

Programmatically, the interrogative utterances can be ignored. Only the action and the program are relevant for understanding. Furthermore, because the grammar ensures it will only be those six possibilities that can be returned, the logic to determine whether or not the action is to open an application can be quite concise:

```
private boolean isOpen(String inVal)
{
    //start | open | stop | close | kill | shut down
    if (inVal == null)
        return false;

    if (inVal.toLowerCase().contains("open") || inVal.toLowerCase().contains("start"))
        return true;

    return false;
}
```

**Figure isOpen method from Jarvis**

Even in this relatively simple example, however, the recognizer is easily fooled. Something as basic as properly identifying word as Microsoft Word would cause a failure given the above configuration of the grammar. While grammar configuration is much cleaner and ultimately more maintainable than source code, this still presents a major design hurdle. How much is enough with regards to optional quirks?

Deviation was less a concern for the Battleship experiment due to the fact that recognizer will return a speech to text string representation of the uttered phrase. As long as the user obeyed the rules of the game, it was possible to programmatically look for one of the possible letters and one of the possible numbers. Likewise if the user abandoned the rules and tried a phrase that was out of bounds for the naming coordinates context, it was easy to disregard the input and move into error handling to ask them to try again.

# Dialog Management

## Overview

With natural language understanding accomplished, the system now needs to act on the user input and respond with a result. This final piece of application architecture is the driver of usability; the brains behind the whole operation.

## Considerations

Because this stage is all about orchestration, domain and context effect dialog management the most. The appropriate medium for responding to a user varies depending on the user's physical environment and the limitations of the device they are using plus the limitations of their own brain. Also relevant are the nature of the response in terms of scale and complexity and recovering from the inevitable failures of the underlying pieces of architecture.

## Multimodality

Multimodality is the understanding that communication can occur through different mediums besides language. In an article titled "The Magical Number Seven, Plus or Minus Two," George Miller argued for a pattern of short term memory in humans that could handle seven items (give or take two items depending on circumstances). The number seven, therefore, is often kept in mind for designers who are considering the number of items to put into things such as menus and dropdowns. For a completely auditory experience, however, such as when an electronic persona reads a sentence to a human, there is evidence that seven, or even five, items can be overwhelming to the human memory. In fact, it

appears that our capacity to remember what is spoken to us works well when the information is presented in groups or clusters of no more than four items (BroadBent ; Wickelgren ). Three is often the safe choice and is thought to be the number that our brain naturally groups pieces of information in.

The ramification of this is that the cognitive load of the human end user has a limited auditory capacity and developers should only consider using auditory responses with Text To Speech engines or with an Automated Persona within contexts that do not require the exchange of a lot of information to the human.

Both experiments conducted in this study have the luxury of the domain including a screen that allows the applications to utilize both graphical and audio inputs and outputs. However, the Battleship experiment does the most eloquent job of demonstrating the power that multimodality can wield. Imagine, briefly if an end user tried to play a game like Battleship as a completely audial experience. While the above grid is very easy to process at first glance, it would be quite tough to remember which coordinates had already been hit or missed and which ships had already been sunk without the visual. For the user, trying to determine where their next strike should occur would be a major task fraught with failure and redundant strikes.

This means that even though the scope of the experiment include playing the game on the go in a hands free manner, the graphical element is still critical to the usability of the application. Indeed, system designers have to consider which mode is appropriate in a given context. A picture tells 1,000 words (probably literally) when the task is to convey the state of a Battleship grid to an end user. However, that doesn't mean the user has use a tactile input mechanism such as a mouse or finger to operate the grid. Nor does the user have to rely solely on sight to receive feedback on whether or not their attempted strikes hit or miss.

The user should have the ability to order a strike and receive this limited, yet important and frequent, feedback in whatever way is best for them in a given context. If the user is in a quiet library? The desirable mechanisms are likely to be graphical and tactile only. If the user is driving down a free-way? The choice will likely include auditory elements such as voice recognition and speech to text to support the graphics.

Battleship provides this flexibility by providing two buttons to allow the user to indicate when they desire to interact with the system in ways beyond the GUI. The "Speak" button will prompt the user for a single utterance; which the system will understand and manage before returning the system to its default tactile mode. The "Hands Free" button can be either "On" or "Off". When it is "On" the game will automatically prompt the user when it is time to speak and they will be able to play the entire game without ever touching the screen if they so desire. The end result is that is that the game can be operated without hands but it still assumes the ability to occasionally glance at the screen, making it no different than a speedometer, GPS, or radio controls in a vehicle.

The default mode of the game is to use tactile inputs and use graphical outputs only. This was done in deference to usability guidelines around respecting the behavior that users will experience in other applications. The out of the box behavior of an Android device, most mobile devices for that matter, is such that verbal and audial interactions must first be turned on through tactile means. This approach simply respects that precedent.

Jarvis also needs to be turned on through tactile means. Even though the application is a running process on the desktop, it will not listen for commands unless the "At your service" button has been clicked.

## Error Handling

When the voice recognition architecture components that occur prior to dialog management break down, the application has to have an ability to recover and get the dialog with the user back on track. To accomplish this, the application has to be ready to react to the different conditions that can occur in the architecture. Voice Recognition accuracy can be described by separating the data into two types: in-grammar and out-of-grammar (*Cohen, Giangola, and Balough*).

In-grammar data falls into one of three categories:

1) **Correct Accept:** The recognizer returned the correct answer.
2) **False Accept:** The recognizer returned the wrong answer.
3) **False Reject:** The recognizer could not find match and gave up.

Out-of-grammar exists in one of two categories:

1) **Correct Reject:** The recognizer correctly rejected the input.
2) **False Accept:** The recognizer returned a value that has to be wrong because the input is not in the grammar.

Of the 5 categories, only one does not need the system to recover: correct accept. In the case of the false rejects, recovery is easier to accomplish because it is clear to the application that something unacceptable has happened and it doesn't have the data it needs to move forward. This can be less obvious when the condition is a false accept (in or out of grammar) because data has been returned and the application will try and move on to natural language understanding.

### *Rejection*

Battleship does not have much of an issue with either reject condition. The recognizer is going to return a variation of the speech that was provided as input. That is, unless no speech was provided and the endpointing mechanism timed out or the user uttered something that wasn't an anticipated language.

The Android API includes an onError method that must be implemented when using the RecognitionListener interface. It accepts an integer value that describes the type of error that the recognizer has experienced. Included in the expected types are a value for speech timeout as well as no match.

```
@Override
public void onError(int error) {
    String mError = "";
    String mStatus = "Error detected";
    switch (error) {
    case SpeechRecognizer.ERROR_NETWORK_TIMEOUT:
        mError = " network timeout";
        break;
    case SpeechRecognizer.ERROR_NETWORK:
        mError = " network" ;
        break;
    case SpeechRecognizer.ERROR_AUDIO:
        mError = " audio";
        break;
    case SpeechRecognizer.ERROR_SERVER:
        mError = " server";
        break;
    case SpeechRecognizer.ERROR_CLIENT:
        mError = " client";
        break;
    case SpeechRecognizer.ERROR_SPEECH_TIMEOUT:
        mError = " speech time out" ;
        break;
    case SpeechRecognizer.ERROR_NO_MATCH:
        mError = " no match" ;
        break;
    case SpeechRecognizer.ERROR_RECOGNIZER_BUSY:
        mError = " recogniser busy" ;
        break;
    case SpeechRecognizer.ERROR_INSUFFICIENT_PERMISSIONS:
        mError = " insufficient permissions" ;
        break;

    }
    Toast.makeText(Board.this, mError, Toast.LENGTH_LONG).show();

}
```

**Figure 10: onError method from Battleship**

This help from the API provides the ability to determine what the recognizer experienced to cause the error condition and allows the developer to react accordingly as demonstrated above. However, not all of the possible rejections are something that can be recovered from. For instance, the insufficient permissions error means that the application developer did not configure their software to tell the Android device that it would be using its microphone and recognizer. Encountering this error would mean that the “Speak” and “Hands Free” buttons are never going to function. For the rest of this topic, let us assume that the rejection is a misunderstanding between the user and the recognizer.

For Battleship, what to do in a rejection scenario depends largely on the interaction context chosen by the user. If the user clicked the “Speak” button, the application should return to a state awaiting further user input. Remember, the “Speak” button is a one-off event. Because that one event failed, it is now best for the user to decide how they wish to proceed; just as it would be had the input resulted in a positive outcome. When “Hands Free” is the way the user has chosen to interact with the system, things get a little more interesting.

After a single error, it is not uncommon for designers to assume that a simple misunderstanding has occurred and a simple do-over is all that is needed to correct the issue. This rapid approach at re-prompting means that the system will repeat an abbreviated version of whatever prompt encountered the failure before escalating the detail in the prompt and offering more help after each failure.

This approach works well for direct prompts such as:

| Event | Prompt |
|---|---|
| Initial Prompt | *"What's your account number?"* |
| First error | *"I'm sorry?"* |
| Second error | *"Sorry, I didn't understand. Please say your 10 digit account number."* |
| Third error | *"Sorry, I still didn't understand. Your 10 digit account number appears on your monthly statement at the top right corner. Please say your account numbers now, or for more information, say Help"* |

**Figure 11: Sample Prompt (From Cohen, Giangola, and Balough)**

It probably also has relevance when the user is trying to manage the Battleship game in the hands free context. If the user is failing to move past recognition after a couple of tries, it would be appropriate to make them aware that they can reset the game, close the game, or continue playing the game. However, this approach could alienate users if utilized during game play. The flow of the game could be jeopardized and the user's intelligence insulted if the rules of the game were explained to them in either a verbal or displayed dialog several times during the course of a game. Due to the nature of the game, of course, that is exactly what could happen because of the sheer volume of strikes the user will have to deliver in order to play. Therefore during game play, a brief message will display indicating the failure and the application will loop through recognition until acceptance occurs or the user turns "Hands Free" "Off." It uses an audio cue, the standard jingle, used on mobile devices to indicate that it is waiting for the user to speak each time the recognizer is listening.

While waiting for the user to provide an application to open or close, Jarvis operates in a similar way. With such a limited scope, it was not a good return on time spent to come up with a more sophisticated prompting system. However, there is opportunity here had the time and scope allowed for it. There will be more on that later. For now the important thing to note is that whatever the prompting strategy happens to be, it still makes sense for Jarvis to go back to listening for a valid response after an error has occurred in the context of working with applications.

The confirmation context that Jarvis offers this project provides an example where simply waiting for acceptance isn't ideal. If the application is waiting for the user to confirm that they would like to close an application, then it can never do the thing that it was created to do: work with applications. Because being in the confirmation context is a result of a request that happened in the main context, the application is no longer in a ready state. It has to finish the task that was requested of it before it can go back to the ready state. What if the misunderstanding between the user and the system is so profound that the user can't recover? To allow the system to go back to its ready state, designers have two choices: they can include a tactile way to kill the request or they can kill the request after a finite number of failures. For Jarvis and confirmation, it is three strikes and the user is out.

### *False Acceptance*

In-grammar false accept scenarios were possible in both Jarvis and Battleship. However, working around the implications of the false accept scenarios were quite different. Because Jarvis was designed to always listen once "At your service" had been clicked, the application had to be ready for the user to give a command at any moment. This caused the recognizer to try and match to a path in the grammar whenever the microphone picked up any conversation and back ground noise; which it often did, causing applications to open and close spontaneously while using the laptop.

To address this, Jarvis first had to be aware when the user was specifically talking to it. Just as the user would use their friend or colleague's name to get their attention before asking a question or starting a conversation, the user would have to single out Jarvis by name.

```
public <workProgram> = <name> <ask> <action> <program>;
<name> = (Jarvis);
<ask> = [please | could you ];
<program> = (Word | Excel | out look | Note pad | power point );
<action> = (start | open | stop | close | kill | shut down );
```

**Figure 12: The Updated Application Grammar From Jarvis**

By updating the grammar, any acceptable path could be required to include the name of the application. Therefore, no correct accept scenarios are possible in the application without addressing Jarvis by name. Now the six possibilities to start Microsoft Word are:

1) Jarvis please start word
2) Jarvis could you start word
3) Jarvis start word
4) Jarvis please open word
5) Jarvis could you open word
6) Jarvis open word

The paradigm shifts with Battleship from a servant application waiting for a command to an application that is facilitating game play in "Hands Free" and "Speak" mode. Using audio and visual cues, the application tells the user when to speak and only listens after the user has been warned.

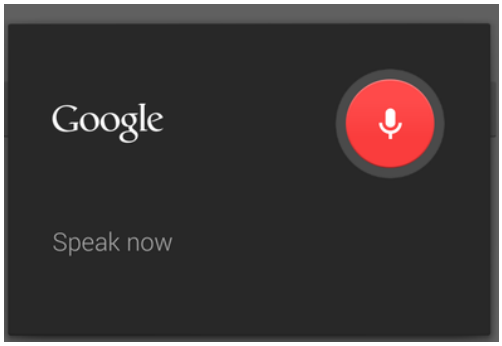


**Figure 13: Android Visual Cue**

Furthermore, the limited domain of Battleship coupled with the vast, wide open grammar that the Google recognizer provided, meant that the efforts to curb the effects of ambiguity and deviation during natural language understanding gave the application a degree of protection from false accepts returning when the user purposefully provided a command. In the context of game play, the application could either understand a valid coordinate or it could not. The only real risk was that it might confuse the coordinates such as "H1" for "A1." Assuming that both were still active game pieces (neither hit nor missed), the end result would be the wrong coordinate acted on. This normally would not be the end of the world, unless the user was embattled in a tight contest with the other player.

Good design principles dictate that when the user is about to perform a risky transaction that will alter the state of the application in way that cannot be undone, then they are asked for a confirmation of the action. In this case, the risk is forfeiture of a turn if the wrong coordinate is struck. It does qualify as a risk, however, the frequency with which the user will provide strikes must again be considered. Asking for confirmation after every strike when the risk is only detrimental a small percent of the time is not practical. The user will gain a good grasp of the accuracy of the recognizer and the natural language understanding through the repetition of voicing the strikes. They also have the ability to strike a coordinate at any time with their finger. As such, this is a good candidate to let the user decide whether or not they want to accept the risk.

By contrast, confirmation should be asked for when the user asks to do something risky that occurs less frequently. For instance, if the user asks Battleship to "Reset" the game they should be asked to confirm the action. As discussed previously, Jarvis seeks confirmation before closing an application. This is because there could be unsaved work, but it also a design choice that had to be made because of the realities of the recognizer used regarding false accept scenarios.

By default, the recognizer will attempt to find a match in the grammar. It has to be configured to look for and recognize out of grammar phrases. Otherwise, it would always return the best match with varying degrees of confidence. In the configuration of the recognizer, developers can tweak how strict it is deciding whether to return a path vs. returning nothing. These tweaks can lead to a significant reduction in the number of false accept scenarios, but they cannot eradicate them totally. There will also be the possibility that the user asks to close something that Jarvis is not programmed to look for and the result will be Jarvis trying to close an application that is programmed to look for; making the confirmation context absolutely necessary.

## What's Next

Though this project was able to cover a great deal of detail around VUI design, time was a limiting factor. In order to produce the necessary deliverables for CIS 693 in a semester, scope had to be established and adhered to.

For Battleship, this limitation meant a fairly crude UI and simple experience from a game play standpoint. The game play has the most compelling upside were time to allow. One thing that remained out of scope for this project was the topic of a persona and how one can enhance the usability of a voice activated system.

Humans tend to prefer dealing with personalities that are identifiable and consistent. Demographics and context often drive what kind of personality we connect with. A single player option for Battleship with different personas running the other game board could have been intriguing. So to would a persona to sort of referee the game and manage game play when the user found themselves in that context. Specifically, text to speech design would have been an interesting topic.

There was also the possibility of keeping track of scores and other competitive measures. It could have been fun to keep a leader board of the players who won with the least amount of strikes. This would have provided the opportunity to work with on device storage on a mobile platform.

With Jarvis, digging in further with regards to context would have been nice. Despite an abstract and uninteresting domain, Jarvis was all about context. Is the user opening or closing an app? Are they trying to search for an answer? Are they trying to use an app? Are they confirming a request? Are they lost? The possibilities seem endless.

Finally, Sphinx4 offers the ability to gauge how confident the recognizer was in the results. This was not easy to set up, but the implications were definitely of value. It would be interesting to see how the results of this project with regards to natural language understanding would shift if results with a low confidence could be discarded.

## Conclusion

While designing applications to process human voice activated commands, it is important to consider what the human is trying to do (context) and what they are trying to do it to (domain). Recognizers will capture the spoken commands and translate them into text as a utility, but they cannot provide the intent behind that text.

Developers still must carefully weigh the considerations for natural language understanding and dialog management discussed in this paper as they create their applications. Battleship and Jarvis provided insights into some of the challenges spawned by these topics and at times offered ways to mitigate them. There were questions that were left unanswered due to the short nature of a semester, but hopefully the way to create a usable voice activated system became a little clearer.

## Bibliography

Adventures in Multimodality. A.I.M. http://muldisc.wordpress.com/

Bernsen and Dybkjaer. Multimodal Usability. Springer.

Broadbent. The magic number seven after fifteen years. In A. Kennedy and A.Wilkes, eds., *Studies in long term memory*. London: Wiley.

Cohen, Giangola, and Balough. Voice User Interface Design. Addision-Wesley

Kotelly. The Art and Business of Speech Recognition. Addision-Wesley

Miller. The Magical Number Seven, plus or minus two: Some limits on our capacity for processing information. Psychological Review 63:81-7

Multimodal Interaction Activity. W3C. http://www.w3.org/2002/mmi/

Sphinx-4 Documentation. http://sourceforge.net/projects/cmusphinx/

Lee. Android Application Development Cookbook. John Wiley and Sons, Inc.

Sheusi. Android Application Development for Java Programmers. Course Technology PTR