Javox: A Toolkit for Building Speech-Enabled Applications

Michael S. Fulkerson and **Alan W. Biermann**
Department of Computer Science
Duke University
Durham, North Carolina 27708, USA
{msf, awb}@cs.duke.edu

Abstract

JAVOX provides a mechanism for the development of spoken-language systems from existing desktop applications. We present an architecture that allows existing Java[1] programs to be speech-enabled with no source-code modification, through the use of reflection and automatic modification to the application's compiled code. The grammars used in JAVOX are based on the Java Speech Grammar Format (JSGF); JAVOX grammars have an additional semantic component based on our JAVOX Scripting Language (JSL). JAVOX has been successfully demonstrated on real-world applications.

1 Overview

JAVOX is an implemented set of tools that allows software developers to *speech-enable* existing applications. The process requires no changes to the program's source code: Speech capacity is *plugged-in* to the existing code by modifying the compiled program as it loads. JAVOX is intended to provide similar functionality to that usually associated with menus and mouse actions in graphical user interfaces (GUIs). It is completely programmable – developers can provide a speech interface to whatever functionality they desire. JAVOX has been successfully demonstrated with several GUI-based applications.

Previous systems to assist in the development of spoken-language systems (SLSs) have focused on building stand-alone, customized applications, such as (Sutton et al., 1996) and (Pargellis et al., 1999). The goal of the JAVOX toolkit is to speech-enable traditional desktop applications – this is similar to the goals of the MELISSA project (Schmidt et al., 1998). It is intended to both speed the development of SLSs and to localize the speech-specific code within the application. JAVOX allows developers to add speech interfaces to applications at the end of the development process; SLSs no longer need to be built from the ground up.

We will briefly present an overview of how JAVOX works, including its major modules. First, we will examine TRANSLATOR, the implemented JAVOX natural language processing (NLP) component; its role is to translate from natural language utterances to the JAVOX Scripting Language (JSL). Next, we will discuss JSL in conjunction with a discussion of EXECUTER, the interface between JAVOX and the application. We will explain the JAVOX infrastructure and its current implementation in Java. In conclusion, we will discuss the current state of the project and where it is going.

[1]Java and Java Speech are registered trademarks of Sun Microsystems, Inc.

2 Basic Operation

JAVOX can be used as the sole location of NLP for an application; the application is written as a non-speech-enabled program and JAVOX adds the speech capability. The current implementation is written in Java and works with Java programs. The linkage between the application program and JAVOX is created by modifying – at load time – all constructors in the application to *register* new objects with JAVOX. For this reason, the application's source code does not need any modification to enable JAVOX. A thorough discussion of this technique is presented in Section 4. The schematic in Figure 1 shows a high-level overview of the JAVOX architecture.

Issuing a voice command begins with a user utterance, which the speech recognizer processes and passes to the NLP component, TRANSLATOR. We are using the IBM implementation of Sun's Java Speech application program interface (API) (Sun Microsystems, Inc., 1998) in conjunction with IBM's VIAVOICE. The job of TRANSLATOR – or a different module conforming to its API – is to translate the utterance into a form that represents the corresponding program actions. The current implementation of TRANSLATOR uses a context-free grammar, with each rule carrying an optional JSL fragment. A typical bottom-up parser processes utterances and a complete JSL program results. The resulting JSL is forwarded to EXECUTER, where the JSL code is executed. For example, in a hypothetical banking application, the utterance *add $100 to the account* might be translated into the JSL command:

```
myBalance = myBalance + 100;
```

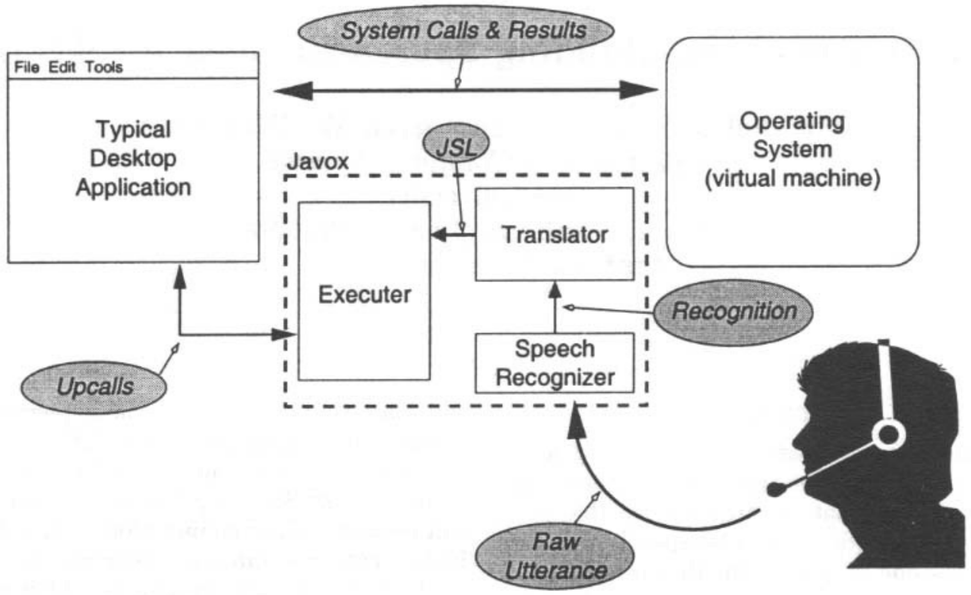


Figure 1: Schematic of the JAVOX architecture.

The job of EXECUTER – or a different module that conforms to EXECUTER's API – is to execute and monitor upcalls into the running application. The upcalls are the actual functions that would be made by the appropriate mouse clicks or menu selections had the user not used speech. For this reason, we are currently concentrating our efforts on event-driven programs, the class of most GUI applications. Their structure is usually amenable to this approach. Our implementation of EXECUTER performs the upcalls by interpreting and executing JSL, though the technology could be used with systems other than JSL. In the banking example, EXECUTER would identify the `myBalance` variable and increment it by $100.

The main JAVOX components, TRANSLATOR and EXECUTER, are written to flexible APIs. Developers may choose to use their own custom components instead of these two. Those who want a different NLP scheme can implement a different version of TRANSLATOR and – as long as it outputs JSL – still use EXECUTER. Conversely, those who want a different scripting system can replace JSL and still use TRANSLATOR and even EXECUTER's low-level infrastructure.

3 Javox Grammars

The JAVOX infrastructure is not tied to any particular NLP method; in fact, the JAVOX grammar system is the second NLP implementation we have used. It is presented here because it is straightforward, easy to implement, and surprisingly powerful. JAVOX grammars are based on Sun's Java Speech Grammar Format (JSGF) (Sun Microsystems, Inc., 1998). JSGF is a rule-based, speech-recognition grammar, designed to specify acceptable input to a recognizer. In JAVOX grammars, each JSGF rule may be augmented with a fragment of JAVOX Scripting Language code – we refer to JAVOX grammars as *scriptable* grammars. The result of parsing an utterance with a JAVOX grammar is a complete piece of JSL code, which is then interpreted to perform the action specified by the user.

The process of speech-enabling an application in JAVOX consists of writing a grammar that contains the language to be used and the corresponding actions to be performed. Building on top of JSGF means – in many cases – only one file is needed to contain all application-specific information. JSL-specific code is automatically stripped from the grammar at runtime, leaving an ordinary JSGF grammar. This JSGF grammar is sent to a Java-Speech-compliant recognizer as its input grammar. In the current Java implementation, each Java source file (`Foo.java`) can have an associated JAVOX grammar file (`Foo.gram`) that contains all the information needed to *speak* to the application. Encapsulating all natural language information in one file also means that porting the application to different languages is far easier than in most SLSs.

3.1 Scriptable Grammars

Since JSGF grammars are primarily speech-recognition grammars, they lack the ability to encode semantic information. They only possess a limited *tag* mechanism. Tags allow the recognizer to output a canonical representation of the utterance instead of the recognition verbatim. For example,

```
public <ACTION> = move [the] <PART> <DIR>;
public <PART> = eyes;
public <PART> = ( cap | hat );
public <DIR> = up;
public <DIR> = down;
```

Grammar 1: A JSGF fragment from the Mr. Potato Head domain.

the tag rm may be the output from both *delete the file* and *remove it*.

Tags are not implemented in JAVOX grammars; instead, we augment the rules of JSGF with fragments of a scripting language, which contains much richer semantic information than is possible with tags. TRANSLATOR receives the raw utterance from the recognizer and translates it into the appropriate semantic representation. JAVOX grammars do not mandate the syntax of the additional semantic portion. Though JSL is presented here, TRANSLATOR has been used to form Prolog predicates and Visual Basic fragments.

JSGF rules can be explicitly made *public* or are implicitly *private*. Public rules can be imported by other grammars and can serve as the result of a recognition; a private rule can be used in a recognition, but cannot be the sole result. The five rules in Grammar 1 are from a JSGF-only grammar fragment from the Mr. Potato Head[2] domain (discussed later). Grammar 1 allows eight sentences, such as *move the eyes up*, *move the eyes down*, *move the cap up*, *move the cap down*, and *move cap up*. Rule names are valid Java identifiers enclosed within angle brackets; the left-hand side (LHS) is everything to the left of the equality sign and the right-hand side (RHS) is everything to the right. JAVOX grammars include the standard constructs available in JSGF, these include:

Imports Any grammar file can be imported into other grammar files, though only public rules are exported. This allows for the creation of grammar libraries. When using JSL, Java classes can also be imported.

Comments Grammars can be documented using Java comments: single-line comments (//) and delimited ones (/* until */).

Parenthesis Precedence can be modified with parentheses.

Alternatives A vertical bar (|) can be used to separate alternative elements, as in the <PART> rule of Grammar 1.

Optionals Optional elements are enclosed within brackets ([and]), such as the in Grammar 1's <ACTION> rule.

[2]Mr. Potato Head is a registered trademark of Hasbro, Inc.

Kleene Star Operator A postfix Kleene star (*) operator can be used to indicate that the preceding element may occur zero or more times.

Plus Operator A similar operator to indicate that an element may appear one or more times.

A grammar's rules may be organized however the developer wishes. Some may choose to have one rule per utterance, while others may divide rules to the parts-of-speech level or group them by semantic value. In practice, we tend to write rules grouped by semantic value for nouns and verbs and at the parts-of-speech level for function words. Grammar 2 shows the Mr. Potato Head grammar augmented with JSL fragments.

The semantic component of each rule is separated from the RHS by a colon and delimited with a brace and colon ({: until :}). Using Grammar 2, the parse and translation for *Move the cap up* is shown in Figure 2.

Each rule may have either *one* semantic fragment or *any number* of named fields. A single fragment is sufficient when there is a one-to-one correlation between a lexical item and its representation in the program. Occasionally, a single lexical item may require several components to adequately express its meaning within a program. In Grammar 2, there is a one-to-one correlation between the direction of movement and the `slideUp` and `slideDown` functions in the <DIR> rules. These functions can also written as a single `slide` function, with the direction of the movement given by two parametric variables (`cos` and `sin`). In this situation, the direction rule (<DIR_NF>) needs to be expressed with two values, each known as a named field. The word *up* may be represented by the named fields `cos` and `sin`, with the values 0 and 1 respectively.

Another issue in JSL – which does not arise in the syntax-only JSGF – is the need to uniquely identify multiple sub-rules of the same type, when they occur in the same rule. For example, in a geometry grammar, two <POINT>s may be needed in a rule to declare a <LINE>, as in:

```
public <LINE> = make a line from
<POINT> to <POINT> :  ...
```

Uniquely numbering the sub-rules eliminates the ambiguity as to which <POINT> is which. Numbering

```
public <ACTION> = move [the] <PART> <DIR> : {: <PART>.<DIR>(); :};
public <PART> = eyes : {: Canvas.eyesObj :};
public <PART> = ( cap | hat ): {: Canvas.capObj :};
public <DIR> = up : {: slideUp :};
public <DIR> = down : {: slideDown :};
public <ACTION_NF> = slide [the] <PART> <DIR> : {: <PART>.slide(<DIR:cos>,<DIR:sin>); :};
public <DIR_NF> = up : cos {: 0 :}
                  sin {: 1 :};
public <DIR_NF> = down : cos {: 0 :}
                    sin {: -1 :};
```

Grammar 2: A JAVOX grammar fragment for the Mr. Potato Head domain.

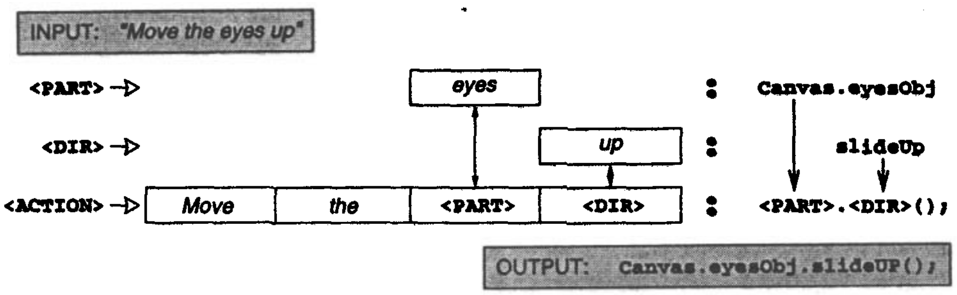


Figure 2: The JAVOX translation process – NL to JSL – for *Move the cap up.*

can be used in both the RHS and the semantic portion of a rule; numbering is not allowed in the LHS of a rule. Syntactically, sub-rules are numbered with a series of single quotes[3]:

```
public <LINE> = make a line from
<POINT'> to <POINT''> :  ...
```

3.2 Javox Scripting Language (JSL)

The JAVOX Scripting Language (JSL) is a standalone programming language, developed for use with the JAVOX infrastructure. JSL can be used to manipulate a running Java program and can be thought of as an application-independent macro language. The EXECUTER module interprets JSL and performs the specified actions. The specifics of JSL are not important to understanding JAVOX; for this reason, only a brief summary is presented here.

JSL can read of modify the contents of an object's fields (data members) and can execute methods (member functions) on objects. Unlike Java, JSL is loosely-typed: Type checking is not done until a given method is executed. JSL has its own variables, which can hold objects from the host application; a JSL variable can store an object of any type and no casting is required. JSL supports Java's primitive types, Java's reference types (objects), and Lisp-like lists. Though JSL does support Java's primitive types, they are converted into their reference-type equivalent. For example, an `integer` is stored as a `java.lang.Integer` and is converted back to an `integer` when needed.

JSL has the standard control flow mechanisms found in most conventional programming languages, including `if-else`, `for` and `while` loops. With the exception of the evaluation of their boolean expressions, these constructs follow the syntax and behavior of their Java counterparts. Java requires that `if-else` conditions and loop termination criteria be a boolean value. JSL conditionals are more flexible; in addition to booleans, it evaluates non-empty strings as true, empty strings as false, non-zero values as true, zero as false, non-`null` objects as true, and `null` as false.

In addition to Java's control flow mechanisms, JSL also supports `foreach` loops, similar to those found in Perl. These loops iterate over both JSL lists and members of `java.util.List`, executing the associated code block on each item. JSL lists are often constructed by recursive rules in order to handle conjunctions, as seen in Section 5.

4 Infrastructure

The JAVOX infrastructure has been designed to completely separate NLP code from the application's code. The application still can be run without JAVOX, as a typical, non-speech-enabled program – it is only speech-enabled when run with JAVOX.

[3]This representation is motivated by the grammars of (Hipp, 1992).

From the application's perspective, JAVOX operates at the *systems-level* and sits between the application and the operating system (virtual machine), as shown in Figure 1. TRANSLATOR interfaces with the speech recognizer and performs all necessary NLP. EXECUTER interfaces directly with the application and performs upcalls into the running program.

Java has two key features that make it an ideal test platform for our experimental implementation: reflection and a redefineable loading scheme. Reflection provides a running program the ability to inspect itself, sometimes called *introspection.* Objects can determine their parent classes; every class is itself an object in Java (an instance of `java.lang.Class`). Methods, fields, constructors, and all class attributes can be obtained from a `Class` object. So, given an object, reflection can determine its class; given a class, reflection can find its methods and fields. JAVOX uses reflection to (1) map from the JSL-textual representation of an object to the actual instance in the running program; (2) find the appropriate `java.lang.reflect.Methods` for an object/method-name combination; and (3) actually invoke the method, once all of its arguments are known.

Reflection is very helpful in examining the application program's structure; however, prior to using reflection, EXECUTER needs access to the objects in the running program. To obtain pointers to the objects, JAVOX uses JOIE, a load-time transformation tool (Cohen et al., 1998). JOIE allows us to modify each application class as it is loaded into the virtual machine. The JAVOX transform adds code to every constructor in the application that registers the new object with `Executer`. Conceptually, the following line is added to every constructor:

```
Executer.register(this).
```

This modification is done as the class is loaded, the compiled copy – on disk – is not changed. This allows the program to still be run without JAVOX, as a non-speech application. EXECUTER can – once it has the registered objects – use reflection to obtain everything else it needs to perform the actions specified by the JSL.

5 Example

Our longest running test application has been a Mr. Potato Head program; that allows users to manipulates a graphical representation of the classic children's toy. Its operations include those typically found in drawing programs, to include moving, recoloring and hiding various pieces of Mr. Potato Head. Grammar 3 shows a portion of application's grammar needed to process the utterance *Move the eyes and glasses up.* The result of parsing this utterance is shown in Figure 3.

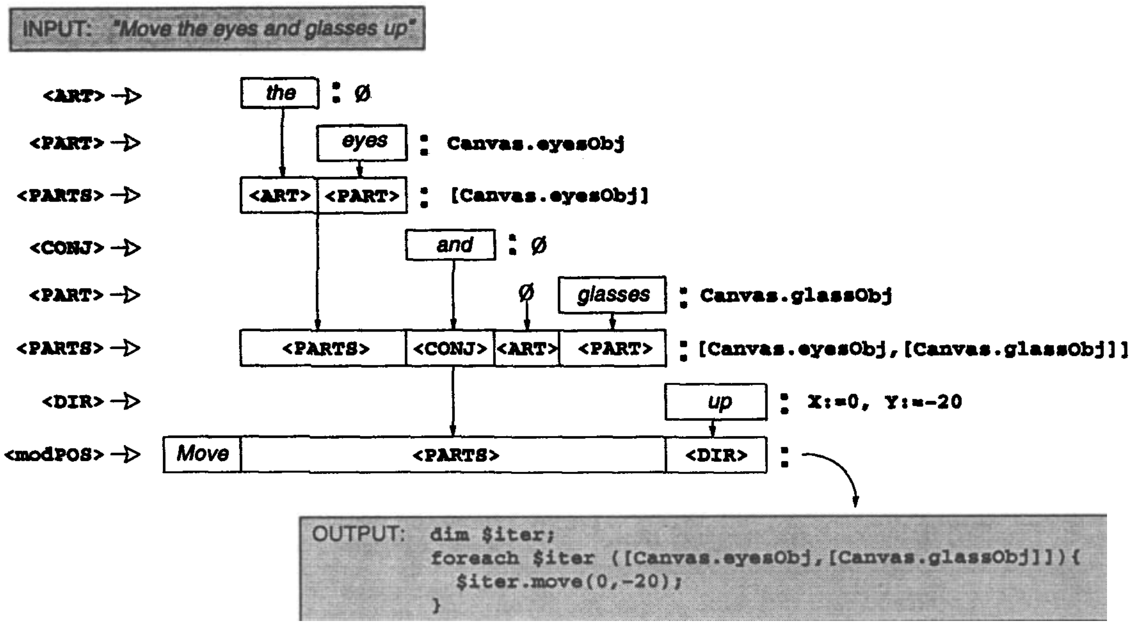
Once TRANSLATOR has processed an utterance, it forwards the resulting JSL fragment to EXECUTER. Figure 4 provides a reduced class diagram for the Mr. Potato Head application; the arrows correspond to the first iteration in the following trace. The following steps are performed as the JSL fragment from Figure 3 is interpreted:

1. A new variable – local to EXECUTER – named `$iter` is created. Any previously-declared variable with the same name is destroyed.
2. The `foreach` loop starts by initializing the loop variable to the first item in the list: `Canvas.eyesObj`. This object's name consists of two parts; the steps to locate the actual instance in the application are:
   (a) The first part of the name, `Canvas`, is mapped to the only instance of the `Canvas` class in the context of this application. JAVOX has a reference to the instance because it *registered* with EXECUTER when it was created, thanks to a JOIE transformation.
   (b) The second part of the name, `eyesObj`, is found through reflection. Every instance of `Canvas` has a field named `eyesObj` of type `BodyPart`. This field is the `eyesObj` for which we are looking.
3. Once `eyesObj` is located, the appropriate method must be found. We determine – through reflection – that there are two methods in the `BodyPart` class with the name `move`, as seen in Figure 4.
4. We next examine the two arguments and determine them to be both integers. Had the arguments been objects, fields, or other method calls, this entire procedure would be done recursively on each.
5. We examine each possible method and determine that we need the one with two integer arguments, not the one taking a single `Point` argument.
6. Now that we have the object, the method, and the arguments, the upcall is made and the method is executed in the application. The result is that Mr. Potato Head's eyes move up on the screen.
7. This process is repeated for `glassObj` and the loop terminates.

After this process, both the eyes and glasses have moved up 20 units and `Executer` waits for additional input. The application continues to accept mouse and keyboard commands, just as it would without speech.

```
public <modPOS> = move <PARTS> <DIR> : {:
            dim $iter;
            foreach $iter (<PARTS>)
              $iter.move(<DIR:X>,<DIR:Y>);
        :};
public <PARTS> = [<ART>] <PART> : {: [<PART>] :};
public <PARTS> = <PARTS> [<CONJ>] [<ART>] <PART> : {: [<PARTS> , <PART>] :};
public <DIR> = up : X {: 0 :} : Y {: -20 :};
public <DIR> = left : X {: -20 :} : Y {: 0 :};
public <ART> = (the | a | an);
public <CONJ> = ( and | plus );
public <PART> = eyes        : {: Canvas.eyesObj :};
public <PART> = glasses     : {: Canvas.glassObj :};
```

Grammar 3: A detailed JAVOX grammar for the Mr. Potato Head domain.


INPUT: "Move the eyes and glasses up"
<ART> -▷
<PART> -▷
<PARTS> -▷
<CONJ> -▷
<PART> -▷
<PARTS> -▷
<DIR> -▷
<modPOS> -▷
the : Ø
eyes : Canvas.eyesObj
<ART> <PART> : [Canvas.eyesObj]
and : Ø
Ø
glasses : Canvas.glassObj
<PARTS> <CONJ> <ART> <PART> : [Canvas.eyesObj,[Canvas.glassObj]]
up : X:=0, Y:=-20
Move <PARTS> <DIR> :
OUTPUT:

```
dim $iter;
foreach $iter ([Canvas.eyesObj,[Canvas.glassObj]]){
  $iter.move(0,-20);
}
```

Figure 3: The translation process for the utterance *Move the eyes and glasses up.*

6 Discussion and Future Work

In practice, building a JAVOX-based, speech interface – for limited-functionality applications – is straightforward and reasonably quick. To date, we have used three diverse applications as our test platforms. Speech-enabling the last of these, an image manipulation program, took little more than one person-day. Though these applications have been small; we are beginning to explore JAVOX's scalability to larger applications. We are also developing a library of JAVOX grammars for use with the standard Java classes. This resource will shorten development times even more; especially compared to building a SLS from the ground up.

One of the existing challenges is to work with applications consisting entirely of dynamic objects, those that cannot be identified at load time. Some typical dynamic-object applications are drawing programs or presentation software; in both cases, the user creates the interesting objects during runtime. We have implemented a system in JSL which allows objects to be filtered based on an attribute, such as color in the utterance: *Move the blue square.*

In situations where there is a one-to-one correlation between a lexical item in the grammar and an object in the program, it is often the case that the lexical item is very similar to the element's identifier. It is quite often the same word or a direct synonym. Since JAVOX is primarily performing upcalls based on existing functions within the program, it also can be predicted what type of objects will cooccur in utterances. In the Mr. Potato Head appli-

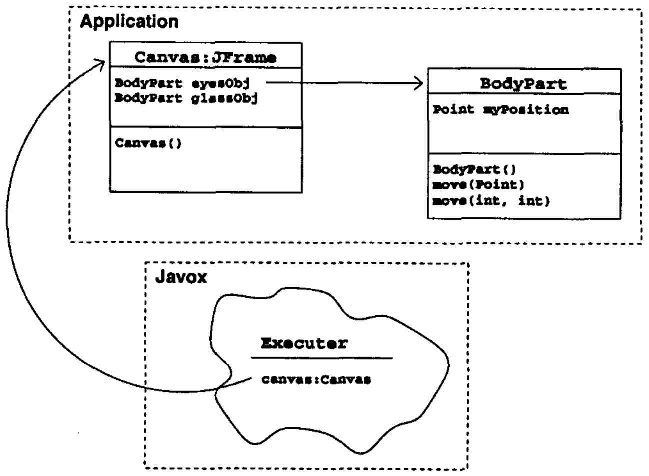


Figure 4: A simplified class diagram for the Mr. Potato Head application.

cation, we can assume that objects representing a `Point` or `integers` will occur when the user speaks of moving a `BodyPart`. We are developing a system to exploit these characteristics to automatically generate JAVOX grammars from an application's compiled code. The automatically-generated grammars are intended to serve as a starting point for developers – though they may certainly require some hand crafting. Our current, grammar-generation tool assumes the program is written with Java's standard naming conventions. It is imaginable that additional data sources – such as a sample corpus – will allow us to more accurately generate grammars for an application. Though in its infancy, we believe this approach holds vast potential for SLS development.

7 Conclusion

JAVOX provides a fast and flexible method to add a speech-interface to existing Java applications. The application program requires no source-code modification: The JAVOX infrastructure provides all NLP capabilities. We have implemented a grammar and scripting system that is straightforward enough that inexperienced developers and those unfamiliar with NLP can learn it quickly. We have demonstrated the technology on several programs and are commencing work on more ambitious applications. The current implementation of JAVOX is available for download at:

`http://www.cs.duke.edu/~msf/javox`

8 Acknowledgments

This work has been partially supported by the Defense Advanced Research Projects Agency under contract F30602-99-C-0060.

References

Geoff A. Cohen, Jeffrey S. Chase, and David L. Kaminsky. 1998. Automatic program transformation with JOIE. In *USENIX Annual Technical Conference (NO98)*, New Orleans, LA.

D. Richard Hipp. 1992. *A New Technique for Parsing Ill-formed Spoken Natural-language Dialog*. Ph.D. thesis, Duke University.

Andrew Pargellis, Jeff Kuo, and Chin-Hui Lee. 1999. Automatic dialogue generator creates user defined applications. In *6th European Conference on Speech Communication and Technology*, volume 3, pages 1175—1178, Budapest, Hungary.

Paul Schmidt, Sibylle Rieder, Axel Theofilidis, Marius Groenendijk, Peter Phelan, Henrik Schulz, Thierry Declerck, and Andrew Brenenkamp. 1998. Natural language access to software applications. In *Proceedings of COLING-ACL-98*, pages 1193–1197, Montreal, Quebec.

Sun Microsystems, Inc. 1998. Java speech API specification 1.0.

Stephen Sutton, David G. Novick, Ronald A. Cole, Pieter Vermeulen, Jacques de Villiers, Johan Schalkwyk, and Mark Fanty. 1996. Building 10,000 spoken-dialogue systems. In *Proceedings of the International Conference on Spoken Language Processing (ICSLP)*, pages 709—712, Philadelphia, PA.