國立交通大學

資訊工程研究所

碩士論文

連結語音辨識系統及應用軟體系統之介面語音之設計及製作

The Design and Implementation of an Interfacing Framework for Bridging Speech Recognizers to Application Systems

研究生: 蔣加洛

指導教授: 陳登吉教授

中 華 民 國 九 十 四 年 七 月

連結語音辨識系統及應用軟體系統之介面語音之設計及製作

The Design and Implementation of an Interfacing Framework for Bridging Speech Recognizers to Application Systems

研究生: 蔣加洛　　Student: Jan Karel Ruzicka
指導教授: 陳登吉教授　　Advisor: Dr. Deng-Jyi Chen

國立交通大學
資訊工程研究所
碩士論文

A Thesis

Submitted to Department of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Chiao Tung University

In Partial Fulfillment of the Requirements

For the Degree

Master of Science

In

Computer Science and Information Engineering

July 2005

Hsinchu, Taiwan, Republic of China

中華民國九十四年七月

The Design and Implementation of an Interfacing Framework for Bridging Speech Recognizers to Application Systems

Student: Jan Karel Ruzicka Advisor: Dr. Deng-Jyi Chen

Department of Computer Science and Information Engineering

National Chiao Tung University

ABSTRACT

Current solutions that aim at bridging speech recognizers with applications use an ad hoc approach and lack of a generic and systematic way. Such recognizer's interfacing approaches usually lead to tightly coupled systems where one application is wrapped by a specific recognizer through a low-level programming implementation that makes future modifications very difficult. Also, without supporting mechanisms to abstract group of actions into single reusable macro-level commands to simplify user interaction tasks, intense and time-consuming overheads for end users are created. Applications, especially multimedia oriented ones deal with highly dynamic content, interfacing and keeping track of this kind of content is not yet addressed.

In this thesis research, an attempt to provide an interface framework for bridging speech-recognizers to applications through a generic and systematic approach is proposed to overcome the above challenges and limitations. Specifically, a script language is designed and implemented that allows users to define the interfacing commands between a speech recognizer and application software. These commands are executed on a user-composed visual interfacing environment that sits on top of applications and acts as a reference layer for interaction. With this approach, interaction commands can be dynamically scripted to simplify user interaction and allow more natural speech commanding. Moreover it allows immediate modifications to be made to an application interfacing environment by simply drawing and registering application zones, without the need of relying on low-level programming for changes to take effect. Our approach also allows for the coexistence of multiple application environments, allowing integration of speech recognition to more than one application at once. A prototype interface framework system has been constructed and used to demonstrate the feasibility and applicability of the proposed interface framework.

ACKNOWLEDGEMENTS

I would like to express my greatest gratitude to Dr. Deng-Jyi Chen, who as my advisor and a friend guided me in a good direction through out all the phases of this research and was always committed to spend the great amount of time and effort he did in brainstorming and discussing many areas of this study. I thank him for his patience, dedication, understanding and care.

I also want to express my gratitude to Professor Yih-Ru Wang for his opinions and advice that provided help for this thesis and to Professors Pao-Ta Yu and Chin-Huang Lee.

I would like to thank all my family for always encouraging me to look up and keep moving forward during the good and bad times; and for all their support and care throughout the years, without them my outcome would have not been the same.

I would like to thank all my friends for being there and sharing their moments and living this life on my side.

Finally, this thesis is dedicated to my mother, the best row model I could ever have.

TABLE OF CONTENTS

List of Figures

List of Tables

CHAPTER ONE
Introduction

Graphic user interfaces that utilize recognition control have been playing an important role as an interfacing technology that makes possible the use of application software to people that are not able to interact with computers through traditional input devices. Advances in recognition technology have opened wide possibilities to these type of users, however current ways of interfacing applications with recognizers are time consuming and result in highly coupled applications that lack customization and flexibility, reducing the speech-driven application domain to users that need them.

1.1 Motivation

Interfacing applications with various recognition technologies (such as speech, gesture, and color recognition, to name a few) will impact current methods of interaction in the area of human-machine interfacing technology. Current solutions that aim at bridging speech recognizers with applications use an ad hoc approach and lack of a generic and systematic way. Such a recognizer's interfacing approaches usually lead to tightly coupled systems where one application is wrapped by a specific recognizer through a low-level programming implementation that makes the future modifications very difficult. Also, without supporting mechanisms to abstract group of actions into single reusable macro-level commands to simplify user interaction tasks creates intense and time-consuming overheads for end users. Applications, especially multimedia oriented ones deal with highly dynamic content, interfacing and keeping track of this kind of content is not yet addressed. A generic application-independent, speech-driven interface generator framework that allows the generation of a modifiable visual interfacing environment without the need of dealing with low-level details must be quested.

The above challenges and limitations are taken into consideration for the conduction of this study as this research attempts to provide an interface framework for bridging speech-recognizers to applications through a generic and systematic approach. Specifically, a script language is designed and implemented that allows users to define the interfacing commands between a speech recognizer and application software. These commands are executed on a user-composed visual interfacing environment that sits on top of applications that acts as a reference layer for interaction. With this approach, interaction commands can be dynamically scripted to simplify user interaction and allow more natural speech commanding.

1.2 Current Recognizer Integration Methods

Currently at least two approaches have been used to interface speech recognizers with application software. Bellow we point out the main features of these two approaches.

1.2.1 Wrapping Integration Approach

A wrapping integration approach focuses on a one-to-one model by integrating one recognition engine with a specific application. The integration is done through the recognizer's API and the application's components through a direct and tightly coupled way (Figure 1). The application is in charge of setting up the recognizer's environment, grammar domain, receiving recognition results and interpreting these results to perform the respective internal invocations to execute interactions on its GUI [1]. As it can be foreseen, in Figure 1, the integration results is one application interfaced with one speech recognizer through a interfacing layer that is in charge of directly mapping speech commands into actions on the application's components.

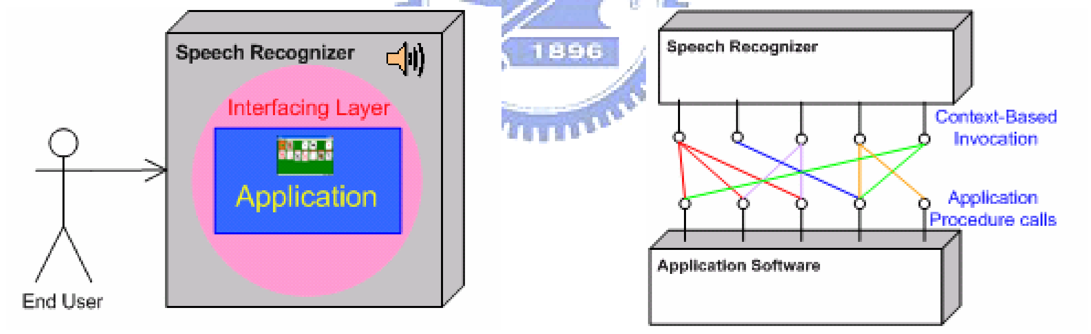


Figure 1. **Wrapping Integration**

Most of speech-driven robots adopt such interfacing approach for its design and implementation. Such is the case of AT&T's Speech-Actuated Manipulator (SAM) [2] that understood spoken commands via telephone and performed the respective actions. Such complex machines must adopt a wrapping integration approach do to the uniqueness that they present in their non standardized internal system that most of the times differ amongst robots. Under such a tightly coupled-system, it is not surprising that any modifications on the low level application software's commands will result in the recoding of the speech interface, leading to the recompilation of the whole system.

1.2.2 OS Integration approach

An OS integration approach focuses on a one-to-many integration by integrating one recognition device to an Operative System's windows environment where applications reside. In a similar way to the wrapping integration approach, the integration is done through the recognizer's API and the bridged system's internal components. This approach adds a reference layer by interfacing applications through the Operating System's API that performs simplistic actions on a focused windows environment where application's GUIs belong. In this way interfacing and interacting directly with the operating system allowing it to respond through interactions with applications of its windows environment. Allowing one speech recognizer to interact with the domain of applications contained in a windows environment at a given time (Figure 2).

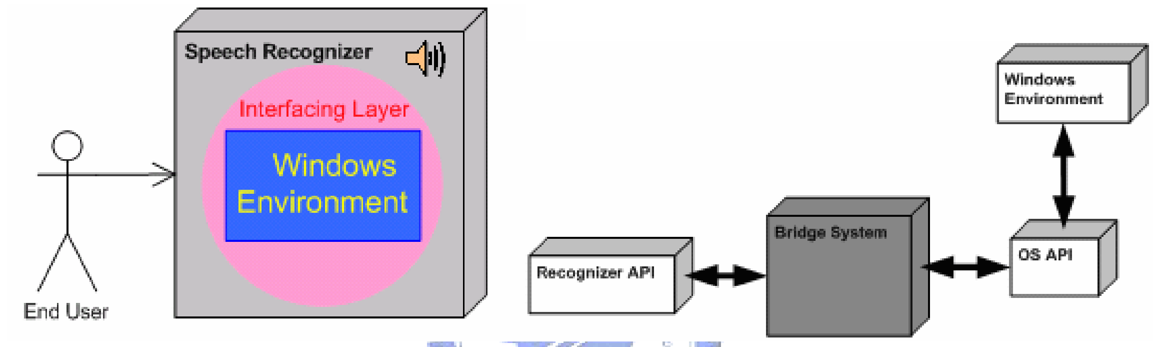


Figure 2. **Windows Environment Integration**

Current Solutions

Three application systems Vspeech 1.0 [3], Voxx 4.0 [4] and IVOS 2.0.1 [5] that utilize an OS integration approach where chosen for discussion to provide a clearer view on how current solutions are designed and what features they provide to users.

Voxx 4.0

Voxx 4.0 [4] is a speech recognition program that incorporates dictation and voice commands for the windows environment. Its main features include:

(1) Window manipulation and menu navigation through voice commands

(2) Document and application opening through simple shortcut words

(3) Custom shortcut creation

To accomplish menu navigation, Voxx invokes OS API parsing functions to retrieve identification of objects present in the windows environment. These Identifiers are used to build the dynamic interaction vocabulary for the speech recognizer. When any recognition occurs, it invokes the OS API functions that perform native actions on the recognized identifiers. Figure 3 shows the shortcut list displayed to the user that is used for viewing the current recognition domain.

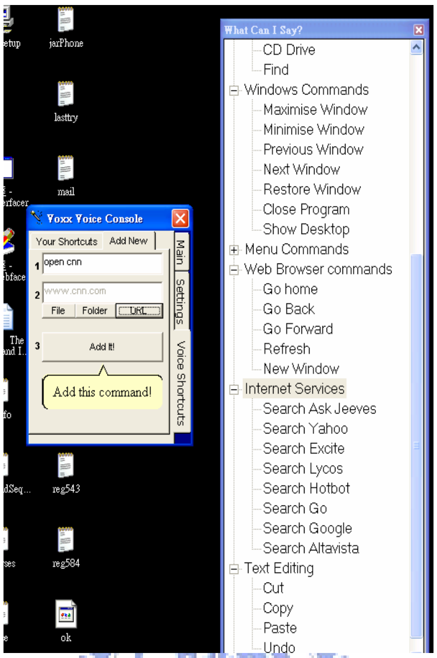


Figure 3. **The Voxx 4.0**

VSpeech 1.0

VSpeech 1.0 [3] is a speech recognition program that incorporates dictation and voice commands in the same fashion as [4]. It differs in the following:

(1) Internet "link" Navigation Support for Microsoft's Internet Explorer

(2) Lacks user-shortcut definition

VSpeech functions just like Voxx, presenting a listing of words that represent the current content that can be spoken to invoke actions on the windows environment (Figure 4). Unlike Voxx, VSpeech adds *URL* links found on focused IE browser to the recognition list, so that they can be accessed by speaking their reference name.

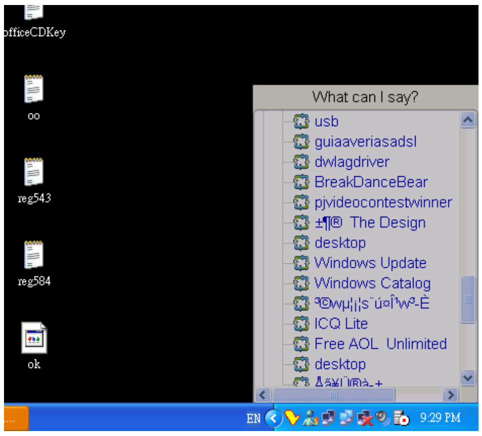


Figure 4. **The VSpeech 1.0**

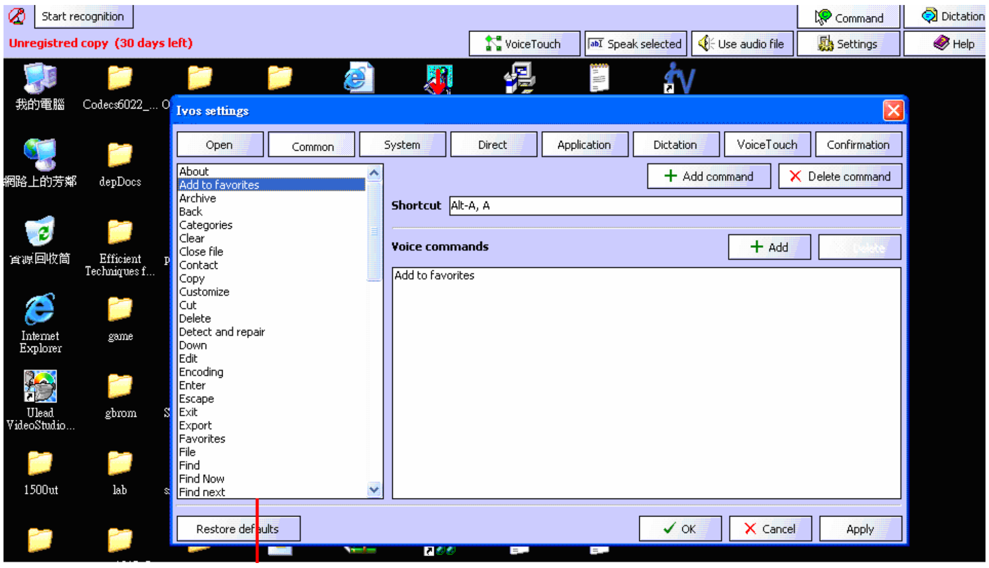
IVOS 2.0.1

IVOS 2.0.1 [5] is also another speech recognition program that incorporates dictation and voice command capabilities. It differs from VSpeech and Voxx in the following ways:

(1) Extends shortcut commands functionality by allowing the user to register synonyms to execute the same actions with different vocabulary

(2) Introduces VoiceTouch technology that enables the computer to learn routines performed by the user as he interacts with the system. This enables routine repetition if needed by the user.

IVOS is a more advanced solution when compared to VSpeech and Voxx, since it introduces a mechanism that simplifies interaction by allowing repetitive tasks to be done by the system. Also it shows signs of a more flexible interaction environment, allowing different vocabulary for referencing the same content. Although it adds extensibility, it does not tackle most of the common limitations that are found in speech-recognition applications. Moreover, the interaction environment imposed to the user compares in a close range of constraint with current solutions even though it attempts to provide a more friendly interfacing mechanism. Figure 5 depicts the outlook of the IVOS interaction environment.



About
Add to favorites
Archive
Back
Categories
Clear
Close file
Contact
Copy
Customize
Cut
Delete
Detect and repair
Down
Edit
Encoding
Enter
Escape
Exit
Export
Favorites
File
Find
Find Now
Find next

Shortcut: Alt-A, A

Voice commands: Add to favorites

Figure 5. **The IVOS 2.0.1**

1.2.3 Challenges and Limitations of Current Recognizer Integration

Limitations suffered by the current integration approaches mostly result from the direct tightly-bind integration of a speech recognizer with either an application or a window's environment. The following challenges and limitations exist in current approaches:

- **Non-Generic:**
  - Current approaches leave no flexibility for future modifications
  - Current approaches lack a generic recognizer interfacing system that can truly coexist with more than one application environment
  - Current approaches lack a graphic interfacing environment that interfaces application's buttons, containers and menu items

- **Complex:**
  - Current approaches focus on recognizer integration through the back-end of applications requiring low-level programming and system design knowledge.
    - When integrating applications, the interfacing process to **bind recognition results to internal application actions** must be redesigned each time
    - Modification of application's interfacing environment require re-compilation of source code

- **Non-Customizable:**
  - Current approaches' tightly coupled system design does not allow the customization of the interaction environment by the user
    - Allows modification of the vocabulary used for speech only
  - Current approaches do not efficiently separate and handle recognition context
- **Inefficient:**
  - Lacks of a post-interfacing mechanism to abstract a group of actions into single commands to minimize user interaction tasks
    - Interaction is based on a "One spoken command yields to execution of one hard-coded Action" basis

1.3 The Proposed Solution

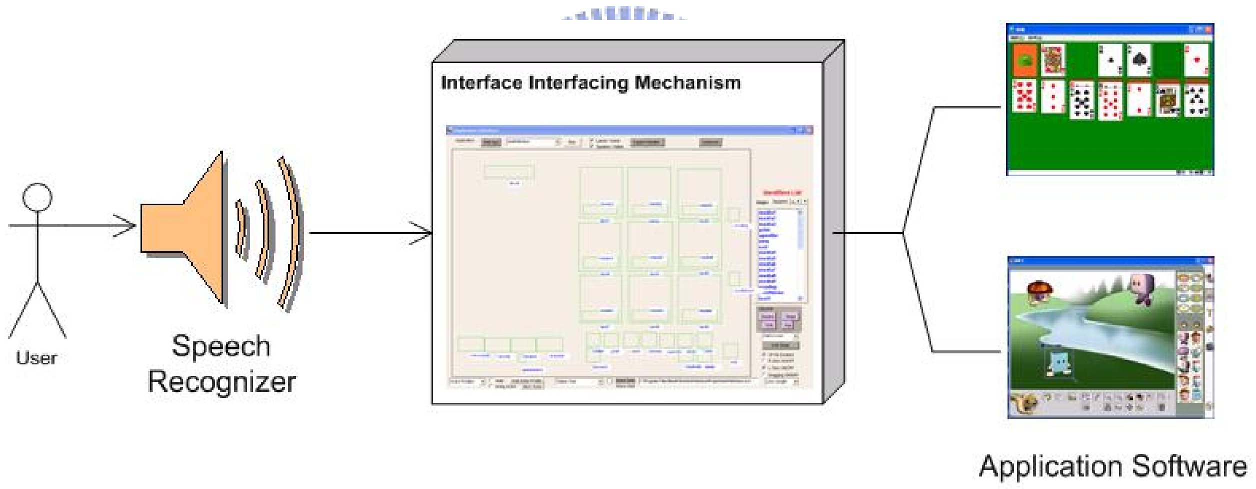


Figure 6. **The Proposed Interfacing Approach**

Our approach consists of an application-independent visual interfacing environment generator to bridge a speech recognizer with applications' front-end (Figure 6). In our approach, to incorporate speech recognition to applications, a user through our system composes a visual interfacing environment by drawing reference zones on top of applications' GUI's interactive areas, without the need of programming the integration. User-generated visual interfacing environments for applications are interacted with by the system as it processes user's requests to perform interaction on the environment's zones that are graphically positioned over interaction objects of applications. Our approach is an improvement of an OS integration approach. The proposed system interacts with target applications by performing invocations to the Operating System's API that then manipulates its input-device and window environments to perform interactions directly on the visual interfacing environment that lays above target applications.

Our approach aims to tackle current challenges and limitations of recognition integration to applications by providing:

Generic Interfacing:

-Fitting more than one application environment

-Allowing simultaneous interfacing content handling for multiple applications resulting in easy application swapping by simply loading the corresponding interfacing profiles that belong to an application.

Flexibility:

-Adopting Front-End custom interfacing through a transparent reference layer

-Developing an interfacing visual environment that allows users to define their specific speech-driven visual environment through the application's front end without doing any low-level programming tasks.

-Allowing visual modification of interfacing content during runtime with out affecting other application's recognition interfaces, and without the need of recompilation of application's source code to make changes take effect.

Efficiency:

-Integrating a language definition that allows the interaction with the visual interfacing environment through spoken commands and that also facilitates composition of macro commands by users to wrap complex and lengthy tasks into single context-free reusable commands, increasing speech recognition efficiency and approximating the way to speak to a more natural one, with out utilizing long and complex sentences to accomplish multiple tasks at once.

Our approach also aims at supporting user interaction with dynamic content of applications during run-time by keeping track of these entities and their different states as the user interacts with them.

1.4 Thesis Organization

The organization of this thesis is divided into two discrete parts. The first part focuses on the challenges present in current approaches and focuses in the foundations that will allow overcoming these challenges. The first chapter is concerned with the importance of the challenges for current recognizer interfacing technologies and how these challenges motivate our study. Chapter Two reviews the technologies involved in this study that are considered for tackling the challenges found in current interfacing solutions.

The second part of the study is concerned about the design and implementation of the proposed system. Chapter Three introduces the aspects of our proposed solution and provides a low-level detailed system architecture design context on how our approach was designed from the bottom up. This Chapter then goes into how the different entities that result from the system architecture design interact whit each other and what interaction steps are taken by these entities to achieve the common goal of setting up a visual interfacing environment and provide a successful interaction with the target application. Chapter Four consists of the definition of the language that is incorporated to our system. Chapter Four un-wraps every aspect of the designed language, its data types, rules, syntax description, and the interpretation steps involved in command processing. This is followed by a detailed analysis and qualitative evaluation of the system in terms of the specific criteria identified for the successful development and deployment of the proposed system in Chapter Five, by setting up and performing common interfacing and interaction scenarios. Chapter Six concludes the paper and offers suggestions for future research.

The reference appendix consists of the listing of previous work referred to and/or referenced in this study.

Appendix I consists of the speech-engine's grammar definition that mirrors that of our designed language.

Appendix II consists of our system's language definition in BNF format.

CHAPTER TWO

Related Technologies Used for the Development of the Interface Interfacing Framework

2.1 Introduction

Creating a successful recognizer interfacing system is dependent on several technologies. These technologies individually belong to different fields of study, however when implemented in a cooperative environment, these technologies merge to contribute towards the vision of Interface Interfacing. This chapter goes into detail about each of the technologies involved in the development of the interface interfacing framework, and gives views of related works for a deeper understanding of each. This chapter serves as the foundation of the overall technological background involved in this study and provides a brief overview of the proposed solution and how it incorporates these technologies.

2.2 "See-Through Interface" Paradigm

In our work we use the "See-Through Interface" paradigm to construct the visual interfacing environment that allows application front-end integration with recognizers through the drawing of reference zones.

The "See-Through Interface" paradigm [6] focus on interfacing tools that appear as a transparent sheet of virtual glass called "Toolglass" between an application and a traditional mouse cursor. These interfaces provide additional views of application objects. The "See-Through" interface provides a new style of interaction that better exploits the user's every day skills. They can be used to reduce steps, cursor motion and errors; moreover they do not require dedicated screen space since they lay on top of the application. These interfaces provide rich context-dependent feedback and the ability to view details and context simultaneously. These widgets [6] can be combined to form operation and viewing macros to simplify use. This paradigm provides mechanisms to draw grids on applications to reference zones that may need this type of guidance, such as drawing panes, or object selection screens. An application may use many views that require more than one "see through interface", for this a managing system is presented to load the corresponding transparent interfaces of each screen, in this way shifting Toolglasses depending on the application content.

The “See-Through Interface” parading is adapted to many research areas. In [7], the authors create an immersive environment that submerges users into a virtual space, effectively transcending the boundary between the real and the virtual world. This virtual 3D world can be manipulated by the user without the need of relaying on traditional input devices such as the mouse and keyboard for interaction. This study adapts a bimanual gesture interpreter and parser that recognizes and translates the user’s arm motions to commands that invoke actions on a “Toolglass” based transparent interface that lays above this 3D environment (Figure 7). The transparent Toolglass interface paradigm is adapted as a gesture interface widget for spatially immerse environments. The user is physically surrounded by this environment as it is projected on walls of a room like structure where the user stands in the middle and uses hand gestures to move the transparent interface to the different locations of the environment to interact and view information of application objects without the need of intermediate hardware such as gloves, 3D-Mouse, or VR headgear. Actions are executed by clicking through one of those wedges, and the action is applied to the object directly behind the Toolglass.

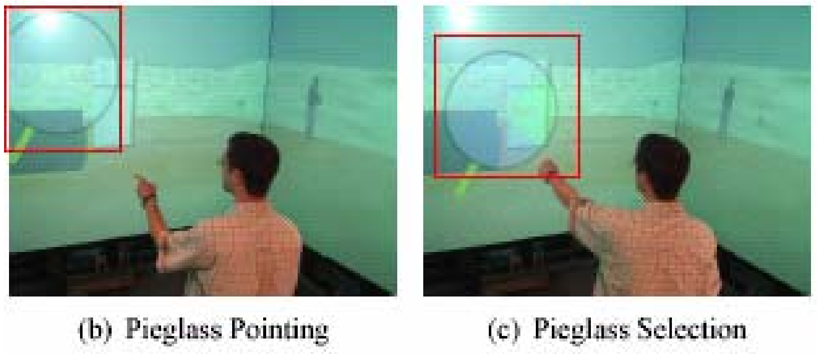


Figure 7. **The 3D Visualization and Manipulation in an Immersive Space**

Another work where the “See-Through Interface” paradigm is adapted is Collaboration Transparency in the DISCIPLE Framework [8]. In this work a framework to share collaboration-transparent single-applications is developed. To share these applications, a conference agent is placed between the application’s GUI and the Windows System (Figure 8). The conference agent intercepts the user input events by adopting a special transparent Toolglass interface to intercept the events destined for the shared application window.

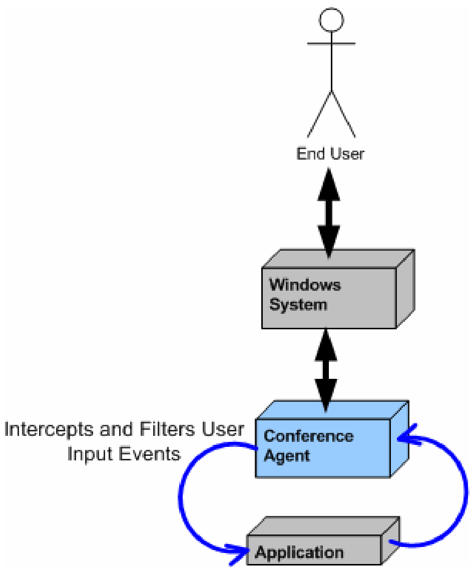


Figure 8. **The Conference Agent Interfacing**

This top-down approach intercepts all the user events (mouse, keyboard, input focus events) using a transparent GUI component without occluding the under-laying applications (Figure 9). Each time an event gets intercepted by the glass-pane, it is dispatched by the agent to the target application object. Such transparent pane is used to filter unwanted invocations to application objects in a collaborative environment when two or more users may be sharing a single application at the same time and such interaction may create conflicts.

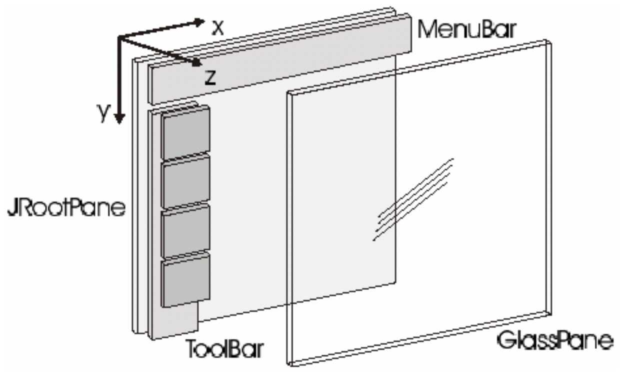


Figure 9. **Transparent Interface**

Futuristic approaches such as Parsimony & Transparency in Ubiquitous Interface Design [9] focus on transparently integrating aspects of the digital world into real life artifacts, by providing ubiquitous interfaces to computation that do not obscure the highly redefined interaction modalities of the host artifact in the physical world. Coexistence of the physical and the digital worlds leads to more learnable interfaces. Here a Toolglass like interface is projected upon real life objects (Figure

10), and it is used to mark the status of the objects during time. A board game is chosen in the study as the physical environment to interface, adding features to the classic game such as game recording and automatic move clock without altering the physical environment. These interfaces, like Toolglass based ones provide different views and information about the interfaced objects attributes when interacting with them physically.

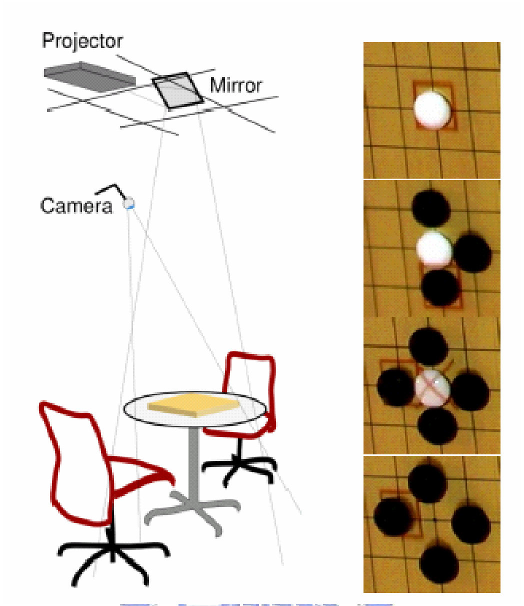


Figure 10. **Interfacing the Physical World with the Digital**

2.3 Script Languages

In our work we apply script languages to enhance interaction efficiency through the definition and use of macro commands that allows abstraction of actions into single context-free reusable commands.

Scripting focuses on connecting diverse pre-existing components to accomplish a new related task [10]. Those languages which are suited to scripting are typically called scripting languages. Script languages are viewed as the "glue" that puts several components together; thus they are widely used for creating and interacting with graphical user interfaces. Scripts are typically stored only in their plain text form (as ASCII) and compiled each time when they are invoked. A scripting language controls the operation of a normally-interactive program, giving it a sequence of work to do all in one batch such as a macro, storing a series of editing commands in a file, and telling an editor to run that "script" as if those commands had been typed interactively. Script languages generally have the following properties:

- Source code is present at run time in production system.
- Use of an interpreter or VirtualMachine is generally required.

- Variables, functions, and methods typically do not require type declarations. There are automated conversions or equivalence between types.
- The ability to generate, load, and interpret source code at run time through an *eval* function.
- Interface to the underlying system components, in order to run other procedures and communicate with them.

In [11], Koong utilizes EDBL (Electronic Book Description Language) script language to hold electronic book projects description files. An interpreter is used along a playback system to present the multimedia effect of authored scripted documents that when processed result in the playing of the final presentation by having the interpreter interpret the description commands found in the script file dynamically. The playback system then executes the associated actions based on the interpretation results of the script instructions.

The language specification of the designed script language for this study is highly influenced by the Java Language Specification [12] to establish its syntactic rules; its design is simple enough to allow programmers to quickly achieve fluency in the language. Unlike Java that is strongly typed by distinguishing between *compile-time errors* and the ones that occur at *run-time*, our language design is based on Just-In-Time compilation by compiling the code as necessary, running it in an interpreted framework [13]. In this way code that is not executed does not get compiled, assimilating our language more to a console command language, but however integrating macro composition functionalities that also allow our language to behave as scripted. Our language approach is very simplistic, being a script language that does not focus on creating objects or maintaining class-like structures but instead interpreting batches of commands.
WinBatch [14], a commercially available high-level macro scripting language takes a similar approach, by designing a language that provides batch automation for Windows systems by allowing users to compose macros that are interpreted utilizing Just-In-Time compilation to automate PC management, business processes, network administration tasks, and overall system use in order to relax the user's interaction overhead. WinBatch accomplishes the above by interfacing directly to the operative system's underlying system components to communicate and run procedures involved in the automation of user's tasks.

2.4 Speech Recognition Engines

Speech Recognition is not a new entity. It first evolved over 30 years ago (Stevens, 1960). This continued growth in technology opened the doors to various applications of Speech Recognition. Speech Recognition not only became a popular medium for use by professionals in the working world, but also an exceptional tool for people with disabilities [15]. Features of Speech Recognition systems have progressively advanced over the last 35 years. Initially, in 1972, dictation and word processing systems were combined to formulate the first Speech Recognition system (Lange, 1993; Meisel, 1993) [15]. At this point, systems could only handle discrete speech dictation where pauses between every word spoken were required for the signal to be processed. Today, it is difficult to find any programs that still use discrete speech; most programs have the capacity to handle continuous speech where the speaker talks naturally, without the need to pause between every word.

Speech applications often use context-free grammars (CFG) to parse the recognizer output and in some instances, to act as the recognizer's language model, Speech recognition engines use CFGs to constrain the user's words to words that it will recognize [16].

CFG is based on grammatical rules that are meant for proper recognition of words. Speech recognition grammar provides the interface of the speech recognizer to the corresponding operations that take place at the target application at the moment of interaction. Complex grammar definition is achieved by organizing rules into hierarchical structures, allowing higher level rules to be composed of references to lower-level rules. Phrases and sub-expressions are represented by separate rules and combined together to form complete sentences. When interpreting a rule decisive selection can be applied to provide a more flexible way of speech and avoiding rules re-definition. Rules restrict the word choices during the recognition process. Applications that are interfaced with speech recognizers listen to context or events that are triggered when a rule is recognized. Depending on the context, the applications takes the corresponding actions, speech recognition engines only handle the recognition job. Phrases spoken use each grammar rule element to determine the recognition path (Figure 11) by traversing the grammatical rule structure.

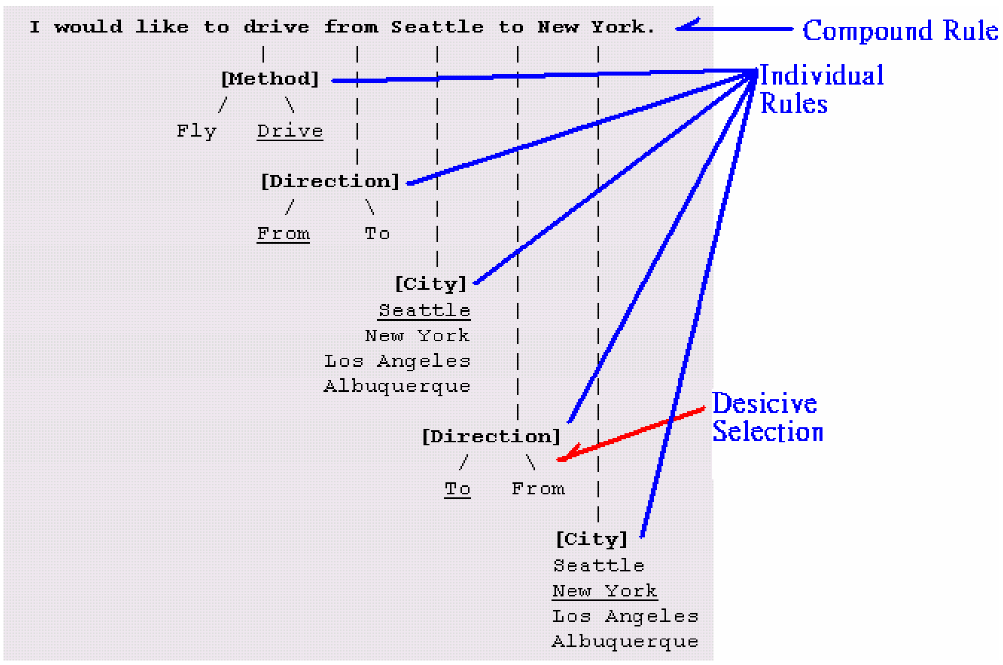


Figure 11. **A Recognition Path Example**

Applications should separate dynamic rule content from static rule content to implement good grammar design and to improve grammar compiler performance. Applications could create a separate rule (isolated in its own grammar) that contained only the static rule content, then the static grammar would contain a rule reference to the dynamic content.

The chosen speech recognizer for this study is Microsoft's Speech-Recognizer V.6.1. [17]. This recognizer contains two different types of speech recognition engines, which are the ISpRecognizer type that is shared amongst several applications by instantiation and the InProc type which is focused for performance hungry applications in which each application has its own ISpRecognizer[16]. In this study the ISPRecognizer type engine is used. Grammar definition for the Microsoft's Speech-Recognizer is CFG rule-based.

A special component of the system was chosen to directly interface the speech recognition engine, this component handles recognized context and distributes it to the rest of the system for further processing to eventually interpret it into actions on the application's interfacing environment. The interfacing component is the only entity in the system that presents a tightly coupled binding with the speech recognition engine as it is integrated into the component through calls of its API to provide a manipulation interface of the recognizer to the rest of the system.

2.5 Conclusion

This chapter presented an overview of the foundation technologies required to base an interface interfacing study on. This chapter provided an insight into the technological needs for the implementation of an environment that tackles the current challenges stated in the previous chapter and how each of these technologies contribute towards the vision of Interface Interfacing. Chapter Three introduces our proposed approach and goes into high detail on the entities that compose it and how they behave with one another to accomplish a common goal of enhanced interfacing with target applications.

CHAPTER THREE
System Architecture Design and Implementation of the Proposed Interface Interfacing Framework

3.1 Introduction

This chapter introduces the proposed system, the Interface Interfacing Framework. The system is discussed in great detail from a high to low level viewpoints to provide a complete view of the system from various perspectives. Focusing on the modules that compose the system and in what way each of them is responsible of providing an overall interfacing of a target application with a recognizer.

3.2 The Interface Interfacing Framework

The proposed system interacts with target applications by performing invocations to the Operating System's API that then manipulates its input-device and window environments to perform interactions directly on the system's "Transparent Interface" that lays above target applications. It adapts the "See-Through Interface" Paradigm to support Front-End custom interfacing of applications, in this way allowing visual modifications to the interfacing environment without the need of recompilation of source code. Our approach integrates a script language that allows real-time interaction through commands with the interfacing environment of an application and also facilitates composition of macro commands by users to wrap complex and lengthy tasks into single context-free reusable commands. The system is designed in a modular way by adapting specialized entities, such as the recognizer interfacing component that integrates a speech recognizer and allows localized integration of future recognizers through modifications on this component while leaving the rest of the system intact. The overall system architecture of the proposed Interface Interfacing Framework is depicted in Figure 12.

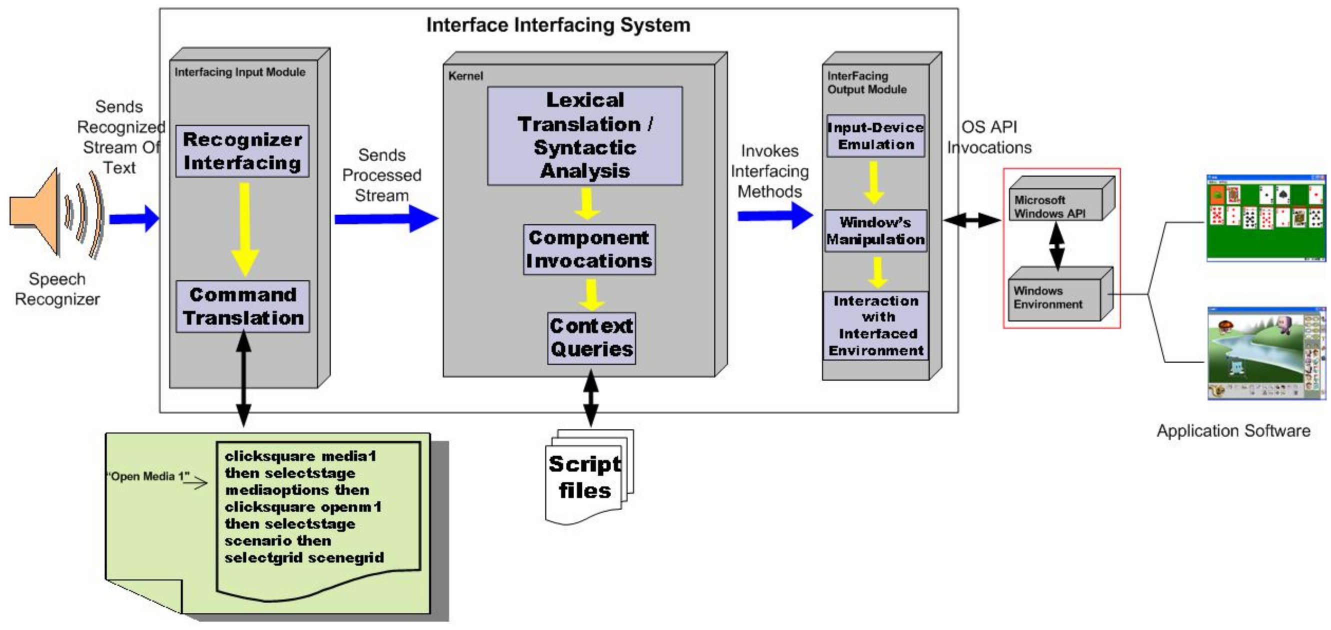

Interface Interfacing System
Sends Recognized Stream Of Text
Speech Recognizer
Interfacing Input Module
Recognizer Interfacing
Command Translation
Sends Processed Stream
Kernel
Lexical Translation / Syntactic Analysis
Component Invocations
Context Queries
Invokes Interfacing Methods
InterFacing Output Module
Input-Device Emulation
Window's Manipulation
Interaction with Interfaced Environment
OS API Invocations
Microsoft Windows API
Windows Environment
Application Software
"Open Media 1"
clicksquare media1 then selectstage mediaoptions then clicksquare openm1 then selectstage scenario then selectgrid scenegrid
Script files

Figure 12. **The proposed Interface Interfacing System**

The interfacing of recognition devices and applications is done through two different interfacing layers that interact directly with the system's kernel (Figure 12). The Interfacing Input Module is in charge of interfacing with recognizers and processing recognized content into a format compatible with the system. The processed stream is sent to the Kernel where the parsing and interpretation of commands take place, here a central invocation mechanisms, delegates invocation requests for the Kernel to the components of the Interfacing Output Module that are the ones that interact directly with the Interfacing Visual Environment to provide interaction with the target application by manipulating the window's environment and emulating input device interactions. To develop our proposed system we took a service-oriented approach by distributing individual services to subsystems of our solution to later proceed with the integration of these, to provide a robust, flexible and modular design. In the following subsections, we elaborate the details of each module inside the proposed Interface Interfacing Framework.

3.2.1 Interfacing Input Module

The main function of the Interfacing Input Module is to interface the system with recognition devices, handle the interfacing, setup and initialization of these, retrieve recognition content and process it by translating it into a format compatible with the system's language commands definition.

3.2.1.1 Interfacing Input Module Components

Textual Command Receiver

In charge of retrieving the content recognized by the Speech Recognizer, including the commands that are spoken by the users, communicating the recognized streams to the *Language Translator*, those are the main task of the Textual Command Receiver. In this research, the Microsoft's Speech-Recognizer V.6.1 [17] was chosen as the target speech recognizer. The interfacing was done through the speech engine's Standard Developing Kit (Microsoft's Speech SDK V.5.1) through API calls on the *Textual Command Receiver*.

Language Translator

The language translator takes care of translating the spoken text to the language that is understood by the internal system. Once the translation is completed, it passes down the translated text to the *Macro-Interpreter* component.

Macro-Composer/Interpreter

The *Macro Composer/Interpreter* is in charge of providing the mechanism of macro command composition by allowing the user to wrap sequences of commands in the system's internal language into single reusable macros. This component interprets macros recognized by the speech recognizer by loading their corresponding code into the system.

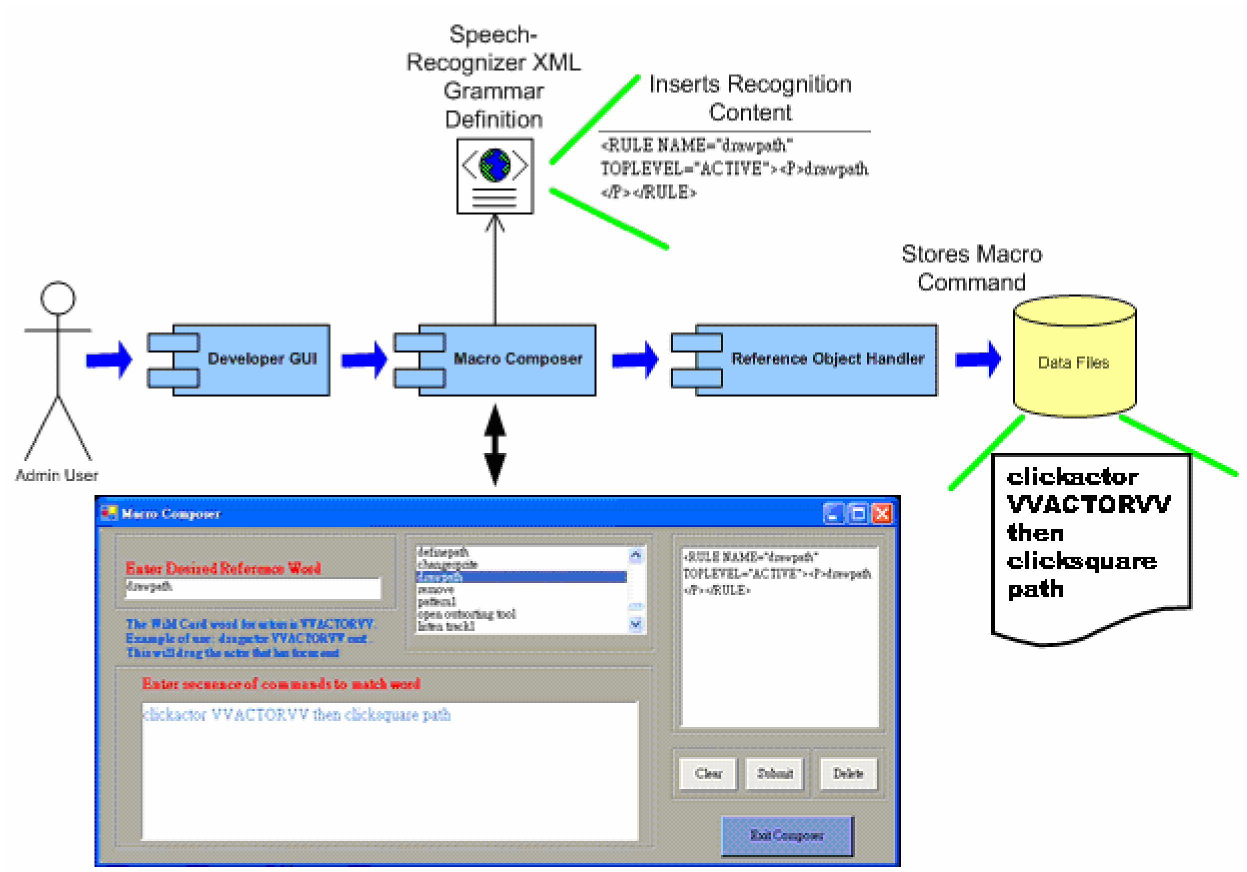
Wildcard-Translator

The *Wildcard-Translator* takes care of replacing wildcards found in macros with the identifiers of currently focused interfacing objects. In this way, it provides a generic and context-free command composition environment.

3.2.1.2 Macro Command Registration and Interpretation

Although interacting with the proposed system by speaking commands according to the syntax provided by the defined language is possible, it requires a high learning curve and overall interaction may be degraded. The system contains a mechanism that allows for the composition of macro commands that are used to perform complex tasks through the invocation of single and reusable commands. The macro commands are defined using the system's internal language and add an extra layer of abstraction but simplifying the user interaction.

The registration of macro commands (Figure 13) takes place at the Developer GUI component where the user composes the macros and assigns them a referenced identifier (“keyword”). During the composition of a macro command a XML structure is dynamically created for speech recognition purposes. When the macro is submitted, the Macro Composer inserts this XML structure into the speech recognition engine’s grammar definition so that a reference to the macro can be successfully recognized when spoken, achieving immediate speech recognition of macros as they are composed. In this way, acting as a black-box mechanism, transparent to the user and avoiding the use of an external XML editor to add recognition of macros to the recognizer.
The macro command itself gets stored to file through the *Reference Object Handler*. When a macro command’s reference keyword is spoken, the macro is loaded from file and executed as a regular set of commands would.


Speech-Recognizer XML Grammar Definition
Inserts Recognition Content
Stores Macro Command
Admin User
Developer GUI
Macro Composer
Reference Object Handler
Data Files
Macro Composer
Enter Desired Reference Word
Enter sequence of commands to match word
Clear
Submit
Delete
Exit Composer

```
<RULE NAME="drawpath"
TOPLEVEL="ACTIVE"><P>drawpath
</P></RULE>
```

drawpath

definepath
changepoints
drawpath
remove
pattern1
open outsourcing tool
listen track1

The Wild Card word for action is VVACTORVV.
Example of use: dragactor VVACTORVV east.
This will drag the actor that has focus east

clickactor VVACTORVV then clicksquare path

clickactor
VVACTORVV
then
clicksquare
path

Figure 13. **Macro Composer/Interpreter**

3.2.1.3 Interfacing Input Module Processes

Whenever the speech recognition engine recognizes spoken phrases, it outputs those phrases as text streams in the spoken language, according to its XML Grammar Definition. The stream of text is then passed down to the Language Translator Component where the first translation takes place. The Language Translator breaks the stream of text into single words, and queries word by word for a corresponding match in the Language Translation Resource to translate them to their corresponding value in the system's language. Once all the phrases are translated to the system's native language, the second translation of the process takes place. The Macro Interpreter receives the stream of text and checks if the stream of text contains keywords that identify macro definitions, it does so by querying the Macro Data Repository for matches. If a match is found, the keyword inside the stream of text gets replaced with the one found. Once a macro is loaded, it is passed down to the Wild Card Translator that checks for the presence of wildcards, replacing any found with the identifier of the interfacing object that currently has focus.

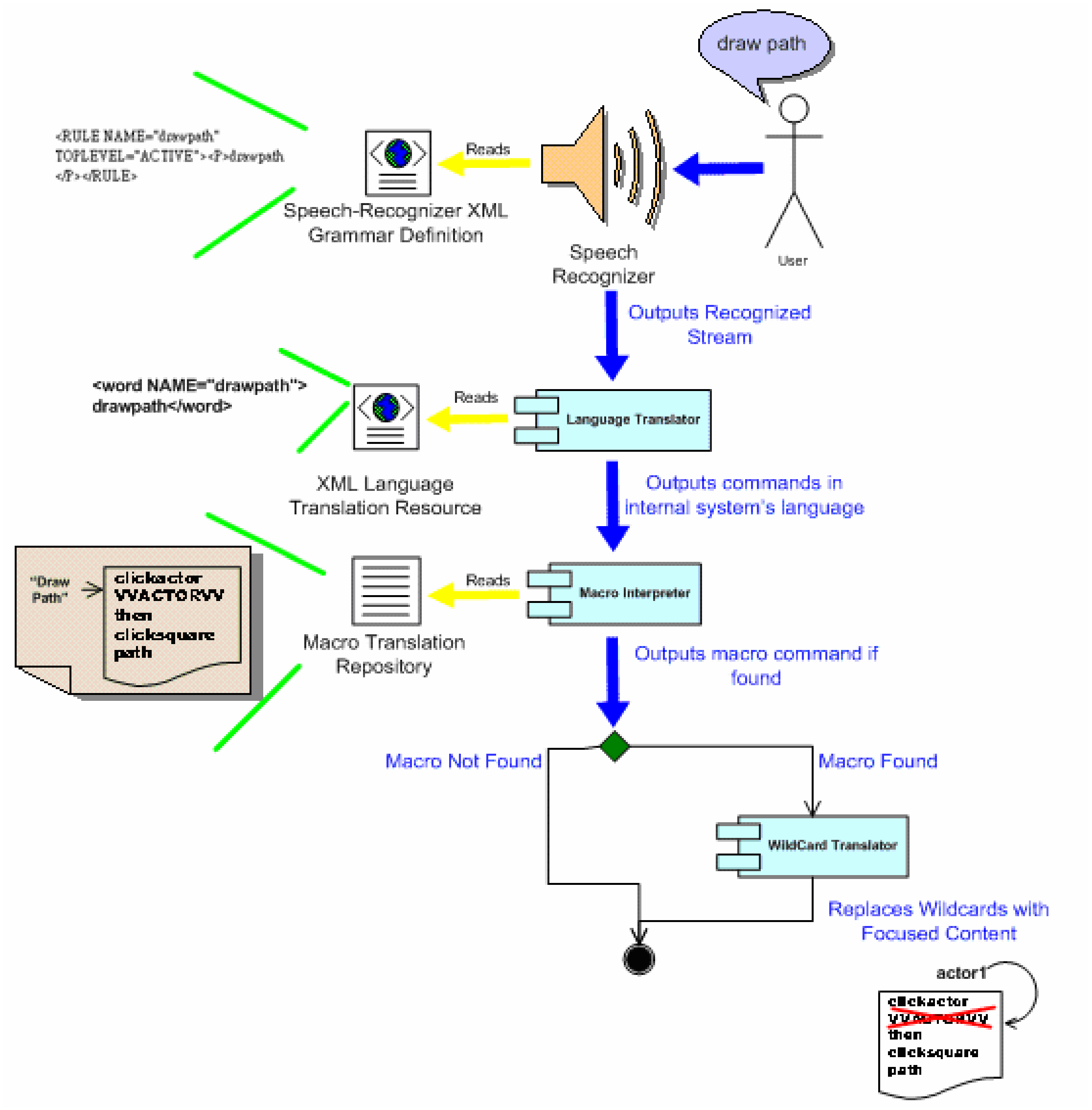

draw path
User
<RULE NAME="drawpath" TOPLEVEL="ACTIVE"><P>drawpath</P></RULE>
Reads
Speech-Recognizer XML Grammar Definition
Speech Recognizer
Outputs Recognized Stream
<word NAME="drawpath"> drawpath</word>
Reads
Language Translator
XML Language Translation Resource
Outputs commands in internal system's language
"Draw Path"
clickactor VVACTORVV then clicksquare path
Reads
Macro Interpreter
Macro Translation Repository
Outputs macro command if found
Macro Not Found
Macro Found
WildCard Translator
Replaces Wildcards with Focused Content
actor1
clickactor ~~VVACTORVV~~ then clicksquare path

Figure 14. **Command Translation Process**

In Figure 14, the macro command "draw path" gets translated into its corresponding system's language format, replacing its wildcards with the actor that currently holds focus.

3.2.2 Kernel Module

The main function of the Kernel Module is to interpret commands into system actions through invocations on components that interact directly with the interfacing environment. The Kernel Module is in charge of delegating and moderating invocation traffic through a centralized component to entities that interact with the "See-Through Interface". The Kernel Module is also in charge of handling the loading, storing, tracking, and activating objects of interfacing content. We design a script language that allows real-time issuing of commands to our system and its rules are used by this module to interpret commands. The language definition is explained in details in **Chapter 4**.

3.2.2.1 Kernel Module Components

Lexical Translator

The Lexical Translator is in charge of receiving a stream of commands from the *Macro Interpreter* and breaking it into token sets, each token set represents a command that is sent to the Syntactic Analyzer for interpretation and validation in a token set per token set basis.

Syntactic Analyzer

It receives token sets from the *Lexical Translator* one set at a time, and processes them by checking their syntactic meaning against the grammatical rules by parsing a syntactic structure and validating any undefined variables found in the tokens. Depending on the parsing path, it produces the target program that consists in invocations through the *Event Driven GUI* on components that interact with the interfacing visual environment.

Interfacing Object Reader/Writer

This component is in charge of storing, retrieving and performing the object activation of the different interfacing objects that are used for building a visual interfacing environment of an application. It is also in charge of handling dynamic interfacing content and providing the tracking mechanism to re-locate them whenever a user interacts with them. This component is subdivided into two parts:

-Application/Actors/Stages/Grids Handler

Handles the loading and storing of objects of type application, actor, actor profile, stage and grid and keeps track of the location of dynamic content such as actors.

-*Square Mapping Mechanism*

Handles square loading and registration by storing each square's graphical information and identifier into individual files under their corresponding stage directory.

Event Driven GUI (*Event Delegating Component***)**

The Event Driven GUI is the centralized component that delegates and moderates invocation traffic that results from the command interpretation process into components that interact with the interfacing environment, in this way acting as a proxy.

3.2.2.2 Kernel Module Interpretation Process

Translated commands that result from the Interfacing Input Module process are sent to the Kernel so that they can be interpreted into a target program (Figure 15) that provides the interaction with the interfacing environment. As the stream of text enters the kernel, the *Lexical Translator* splits the stream of text into token sets. Each token set represents a single command that is fed down to the *Syntactic Analyzer* for interpretation. When the *Syntactic Analyzer* receives a token set, it analyses it token by token and traverses the parsing structure until a match of a valid command with a compatible format is found. Once the parsing is successful, the corresponding target program is executed at the *Event Delegation Component* (Event-Driven GUI) and the later delegates the invocations to the respective system components.

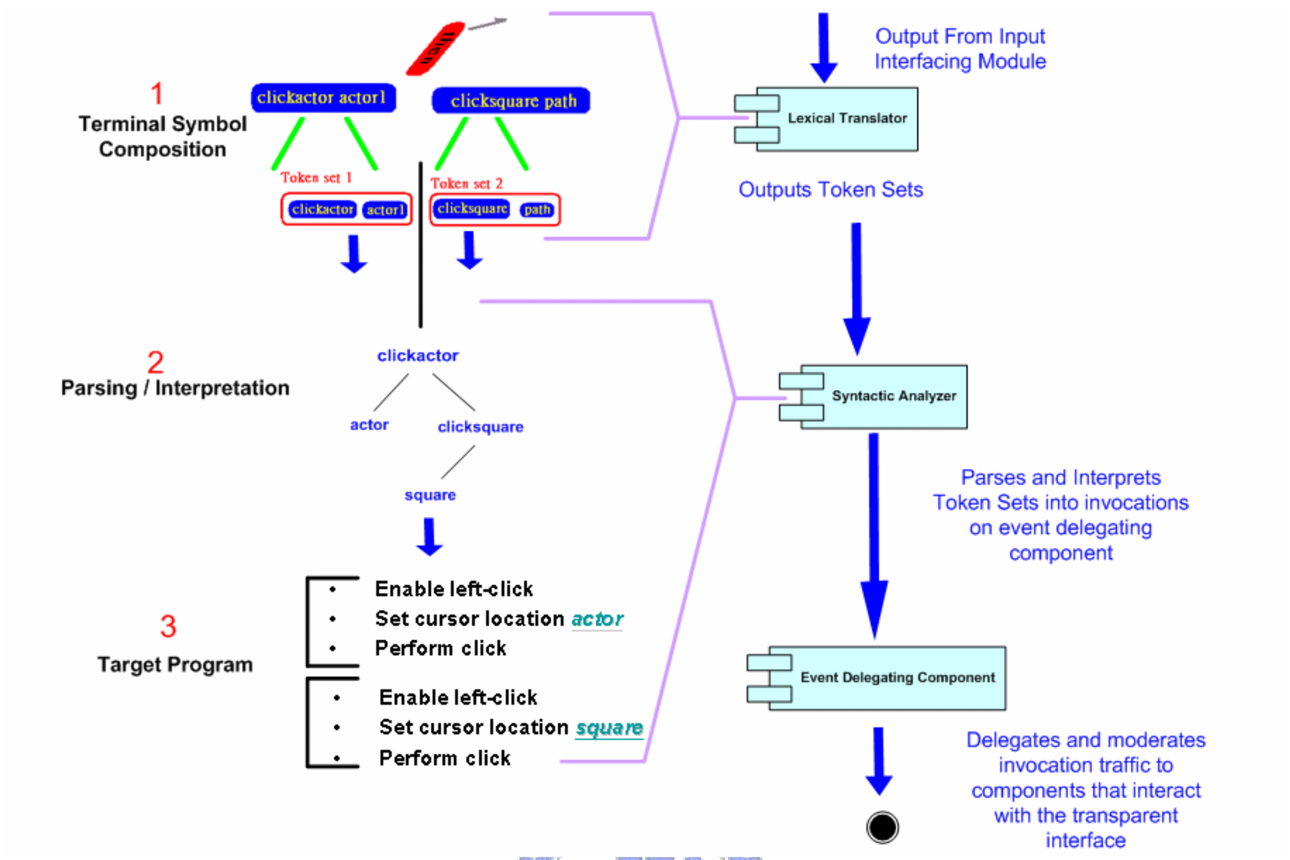

1
Terminal Symbol Composition
clickactor actor1
then
clicksquare path
Token set 1
clickactor
actor1
Token set 2
clicksquare
path
2
Parsing / Interpretation
clickactor
actor
clicksquare
square
3
Target Program
Enable left-click
Set cursor location actor
Perform click
Enable left-click
Set cursor location square
Perform click
Output From Input Interfacing Module
Lexical Translator
Outputs Token Sets
Syntactic Analyzer
Parses and Interprets Token Sets into invocations on event delegating component
Event Delegating Component
Delegates and moderates invocation traffic to components that interact with the transparent interface

Figure 15. **Command Interpretation Process**

In this example diagram, the command resulting from the translation process of the input interfacing module (“*clickactor actor1 then clicksquare path*”) is broken down into two token sets that are interpreted into the target program depicted above.

3.2.2.3 Kernel Module Interfacing Object Handling

The Kernel Module is also in charge of loading, storing, tracking and performing the object activation of interfacing content. To store a graphic object (Figure 16), it involves the drawing of the reference zone at the *Developer GUI*. Once the request is made, the *Square Mapping Mechanism* instantiates the object. If the object is of grid type, the *Grid Composer* component is invoked to automatically compose a grid by creating several squares that later get instantiated and submitted to the storage repository through the *Square Mapping Mechanism*. If the object is of type square, the user specifies its name and other information back in the *Developer GUI* witch then invokes the *Square Mapping Mechanism* once more to submit the graphic object.

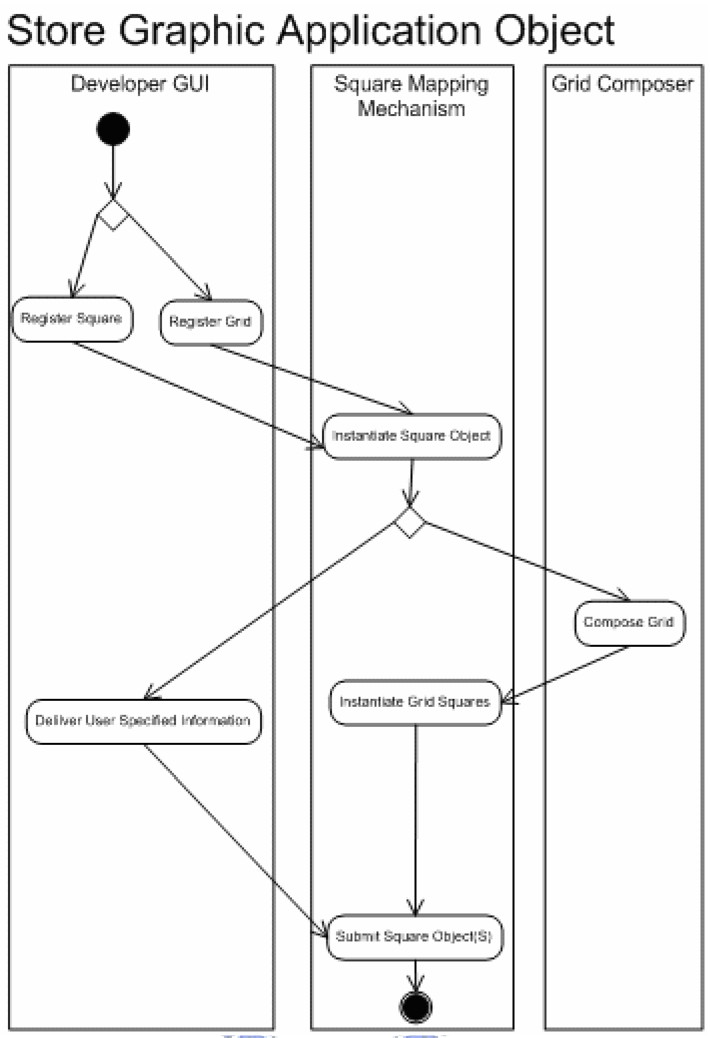


Figure 16. **Storing Graphic Application Objects**

The registration of a non-graphic application object (Figure 17) can be made directly from the *Developer GUI*, by interacting with the controls of the desired object type, triggering the instantiation of the object type in the *Application Handler* component. If the object to be stored is of a type that requires automatic creation (grids, actors, actor profiles) then the system handles the specification of its information, otherwise the user specifies this information at the *Developer GUI* component during the registration stage and requests the submission of the object to finalize the process.

The *Syntactic Analyzer* also interfaces with any of the methods used to store application objects at the *Developer GUI* so that they can be executed through speech.

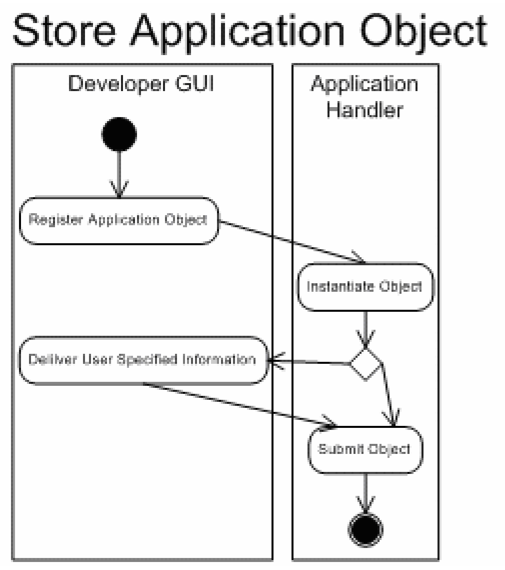


Figure 17. **Storing Non-Graphic Objects**

A request to delete an object (Figure 18) can be made directly from the *Developer GUI*, by interacting with the controls of the desired object type. The *Square Mapping Mechanism* component then takes care of parsing the object storage files and removing any information that corresponds to the selected object. The *Syntactic Analyzer* also interfaces with any of the methods used to delete application objects at the *Developer GUI* so that they can be executed through speech.

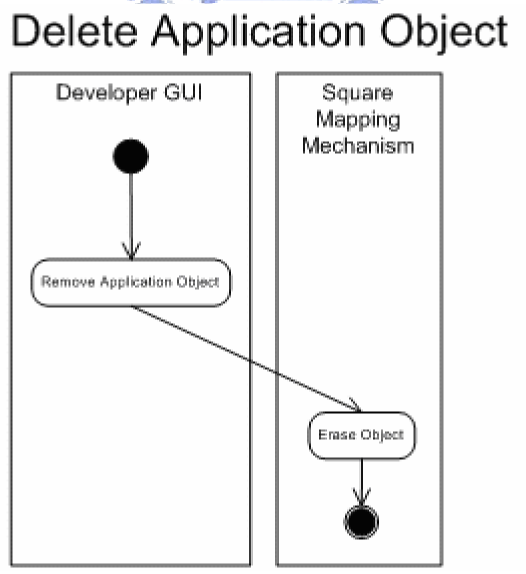


Figure 18. **Deleting Application Object**

Reference interfacing objects such as squares, grids, stages, actor profiles and actors are stored-retrieved and modified dynamically into and from a 4-level hierarchical directory structure (Figure 19).

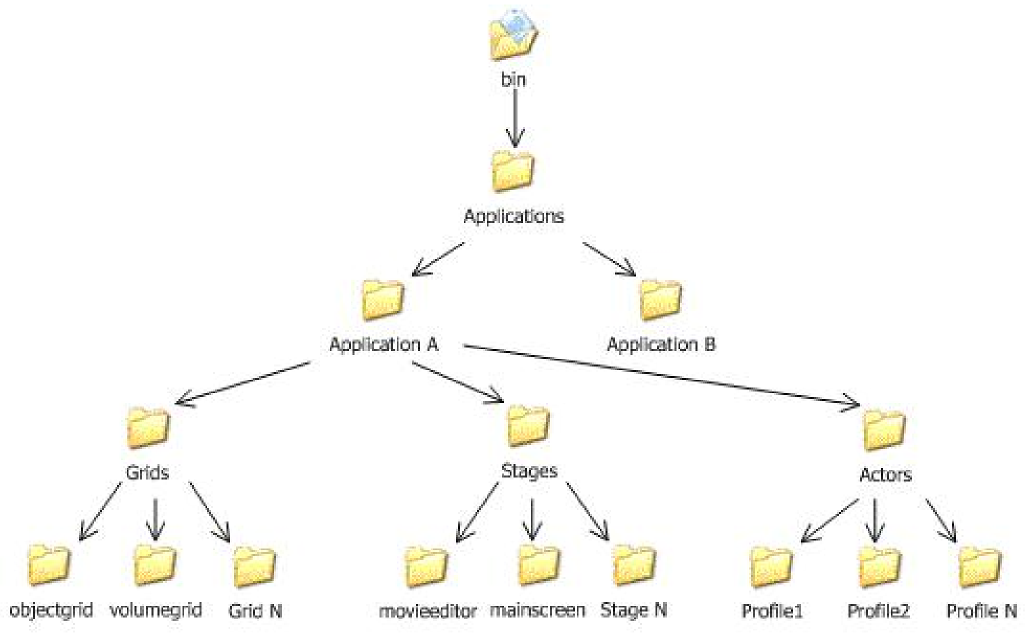


Figure 19. **Interfacing Objects Hierarchical Organization**

3.2.3 Interfacing Output Module

The main function of the Interfacing Output Module is to provide the mechanisms to interact directly with the front-end of the application through the Interfacing Visual Environment by performing input-device emulation and window's environment manipulation. This module adopts the "See-Through Interface" paradigm to provide the tools that are used to compose the Interfacing Visual Environment and macro commands.

3.2.3.1 Interfacing Output Module Components

Developer GUI(Graphic User Interface)

Component targeted at the composition of the customized Event-Driven Visual Interfacing Environment. The "See-Through Interface" Paradigm is adapted to allow users to visually establish reference to buttons, containers or other context inside the target application. Through this component, zones of the target application can be interfaced by drawing referencing zones such as squares and grids on a transparent frame, organizing the context through non-visual referencing by defining Actor Profiles and Stages. A labeling system is developed to visually label each of the registered reference zones at their graphic location with its registered identification name. Figure 20 depicts an application in its natural state, while Figure 21 depicts the application's corresponding visual interfacing environment composed of interfacing objects.

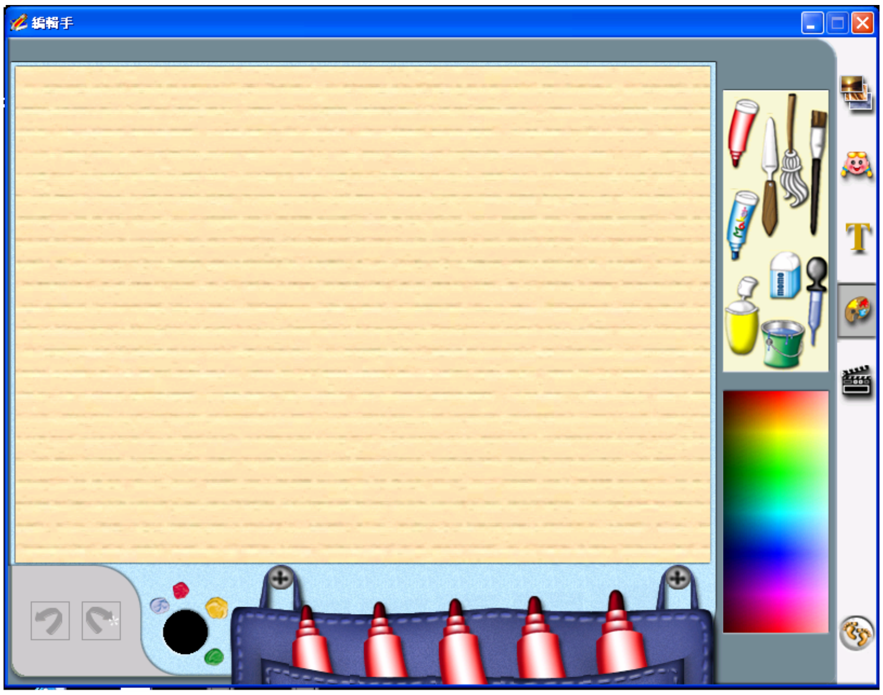
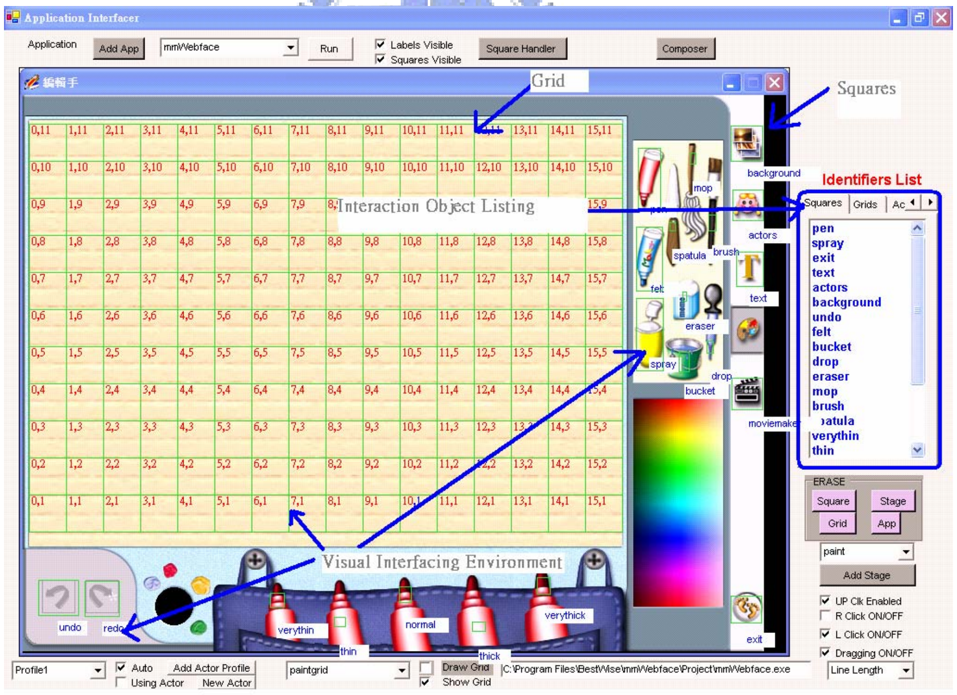


Figure 20. **Unreferenced Application**



☑ Labels Visible
☑ Squares Visible
☑ UP Clk Enabled
☐ R Click ON/OFF
☑ L Click ON/OFF
☑ Dragging ON/OFF
☑ Auto
☐ Using Actor
☐ Draw Grid
☑ Show Grid
C:\Program Files\BestWise\mmWebface\Project\mmWebface.exe

Figure 21. **User Referenced Application**

Grid Composer

The task of this component is to create grids. It creates individual square objects and later returns a collection of them that form the grid, labeling each square with coordinates so that the user can identify each of them individually.

Mouse AI (Input Device Controller)

In Charge of manipulating input devices to perform mouse or keyboard related actions on the Interfacing Visual Environment is the Input Device Controller. This component takes care of emulating the following mouse actions:

-Left_Mouse_Click
-Left_Mouse_Double_Click
-Right_Mouse_Click
-Right_Mouse_Double_Click
-Drag_and_Drop
-Move

Additional features such as moving the mouse cursor in any possible direction and speed, predefined movement-patterns and virtual keyboard implementation are also incorporated in this component.

External App Manager (Window's Environment Handler)

The main task of this component is to manipulate the window's environment by capturing windows, their child and applications inside a container to modify their size and perform any other tasks needed for interacting with the Interfacing Visual Environment. This component interacts directly with the operative system's API to accomplish the above.

3.2.3.2 Interfacing Output Module Processes

Target programs that result from the syntactic analysis are executed through the *Event Delegating Component*. Depending on the command, the requests for each of the involved events is sent to corresponding components that interact directly with the interfacing environment.

These interacting components can also interact with each other by delegating requests amongst them to satisfy a command request. Commands depending on their magnitude could trigger an exponential growth in the number of system-internal calls needed to serve a request or might as well achieve completeness through a single invocation. A more detailed view on invocations is discussed in section **3.6**.

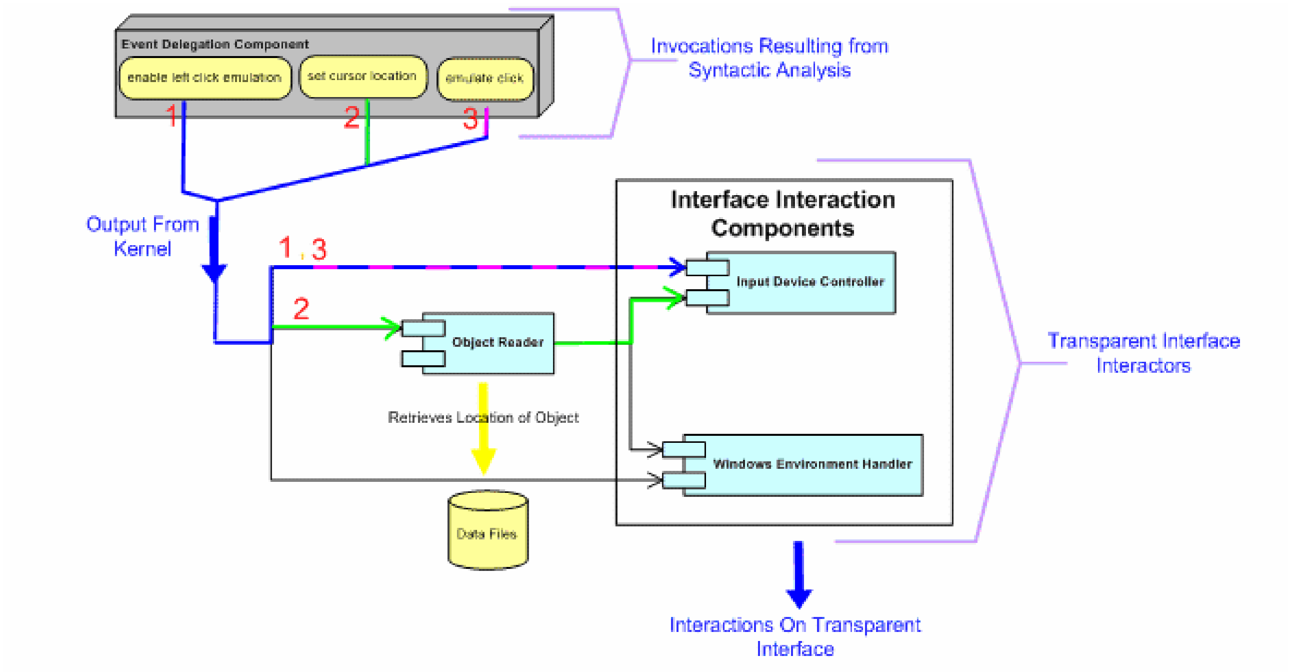

Event Delegation Component
enable left click emulation
set cursor location
emulate click
1
2
3
Invocations Resulting from Syntactic Analysis
Output From Kernel
1 , 3
2
Interface Interaction Components
Input Device Controller
Object Reader
Retrieves Location of Object
Windows Environment Handler
Data Files
Transparent Interface Interactors
Interactions On Transparent Interface

Figure 22(a). **Visual Interfacing Environment Interaction**

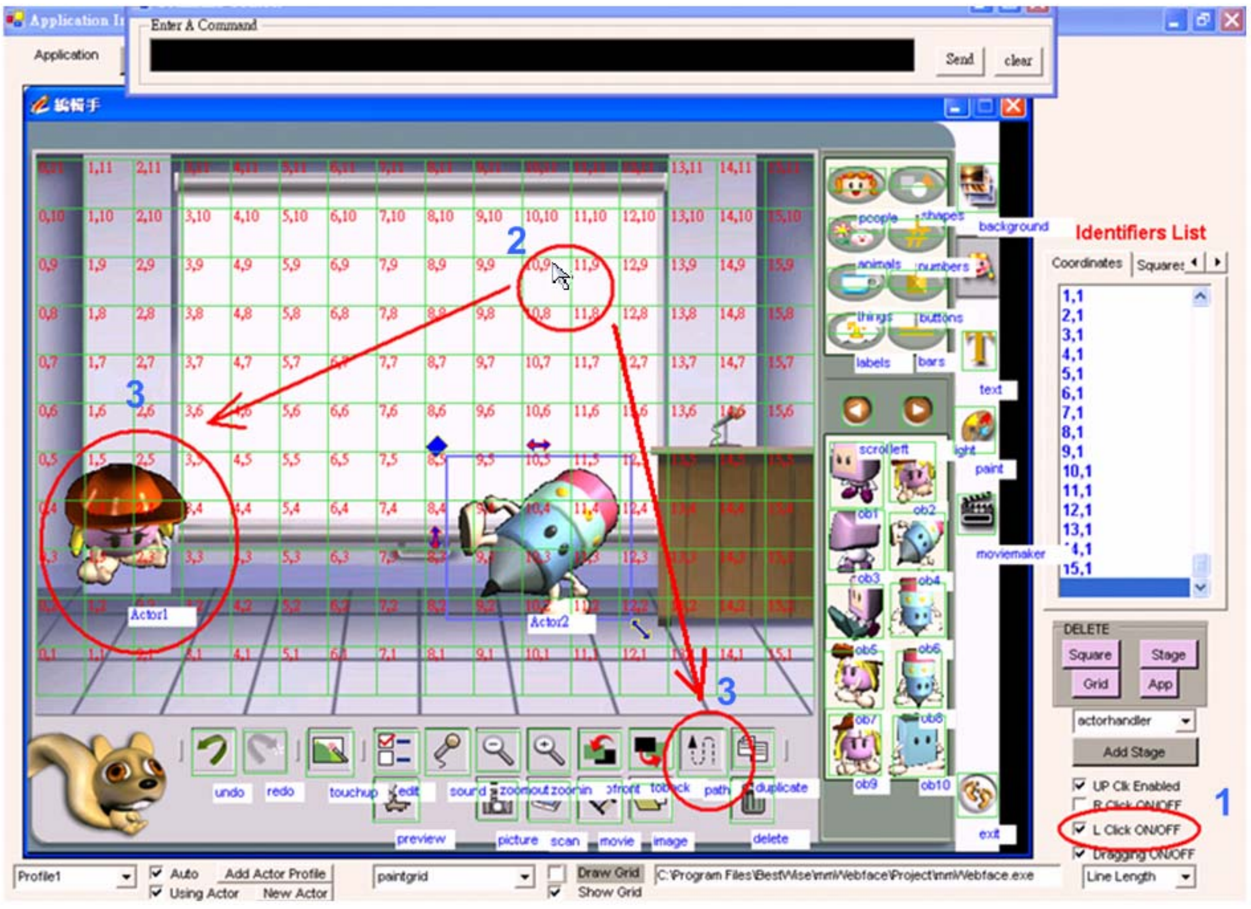


Figure 23(b). **Visual Interfacing Environment Interaction**

In Figures 22, the target program is executed by first enabling left click emulation at the Input Device Controller(1), then the cursor is set to the location of the specified actor that is retrieved by the Object Reader component(2) and lastly a click is performed by the Input Device Controller(3). The same process is repeated for clicking the specified square.

3.3 Design Patterns

Throughout the design of our system, Object-Oriented design patterns [18,19] where taken into consideration to provide a more organized and efficient system interaction. Following are the design patterns that we took into consideration for this study.

3.3.4 Facade Design Pattern

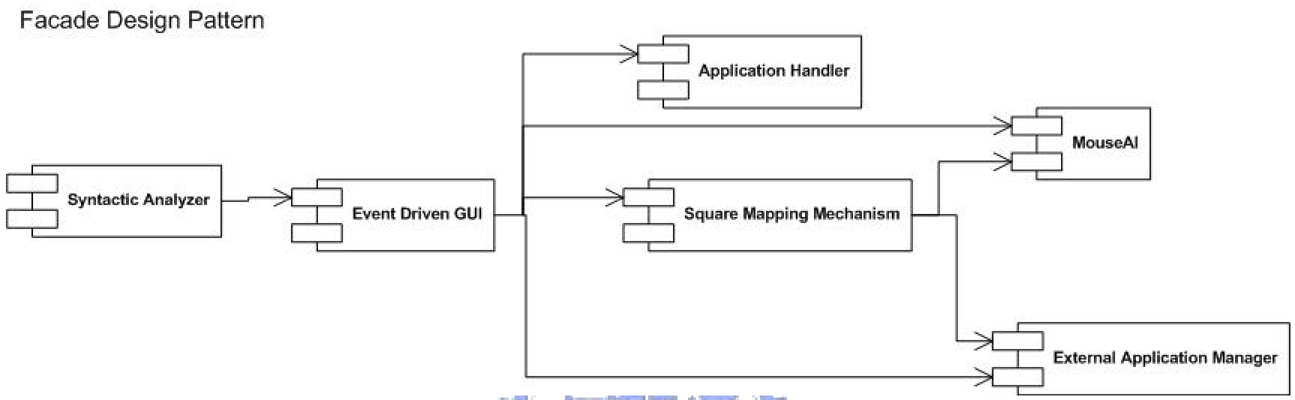


Figure 24. **Facade Design Pattern**

The *facade* pattern (Figure 23) can make the task of accessing a large number of modules much simpler by providing an additional interface layer. When designing good programs, programmers usually attempt to avoid excess coupling between module/classes. Using this pattern helps to simplify much of the interfacing that makes large amounts of coupling complex to use and difficult to understand. This is accomplished by creating a small collection of classes that have a single class that is used to access them, the *facade* [18].

In our system a *facade* approach is used, where the *Event Driven GUI* component is used as this interface layer that acts as the “bridge” to access all of the under-laying modules.
The *Syntactic Analyzer* module acts as the client that requests services from modules through the *Event Driven GUI*. A double facade effect occurs in our system in the sense that the *Square Mapping Mechanism* module itself acts as a facade that delivers requests from the *Event Driven GUI* that is a facade as well.

The primary advantage of using the facade is to make the interfacing between many modules or classes more manageable and organized [19].

3.3.5 Interpreter Design Pattern

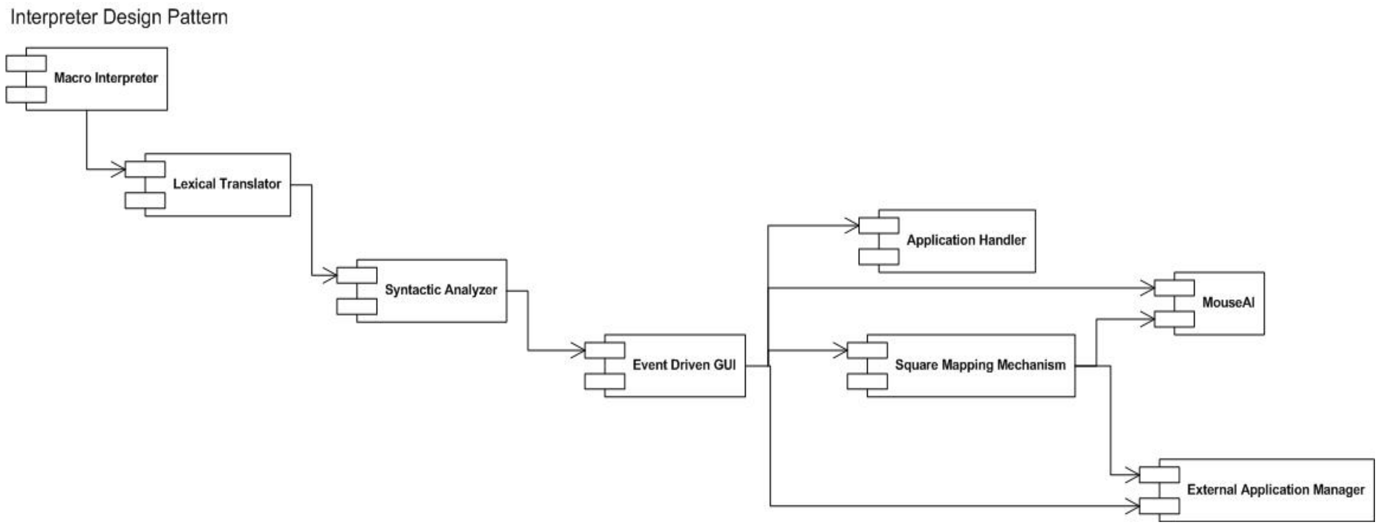


Figure 25. **Interpreter Design Pattern**

The Interpreter Design Pattern (Figure 24) focuses on defining a macro language and syntax, parsing input into objects which perform the correct operations desired [19].

In our system, a language definition exists, and the above design pattern for language interpretation is applied. The Interpreter Design Pattern is present at different levels; the first interpretation is done at the *Macro Interpreter*, to check if the spoken word is a keyword of a macro command, if it is it gets translated to our defined language. The *Lexical Translator* breaks this phrase into token sets and feeds it to the Syntactic Analyzer, acting as the client. The *Syntactic Analyzer* then acts as the Interpreter of the language, by representing phrases according to the defined grammar and performing the set of invocations to procedures through our earlier discussed facade mechanism that reaches a set of worker classes that will take care of performing individual operations of the composed command.

3.3.6 Proxy Design Pattern

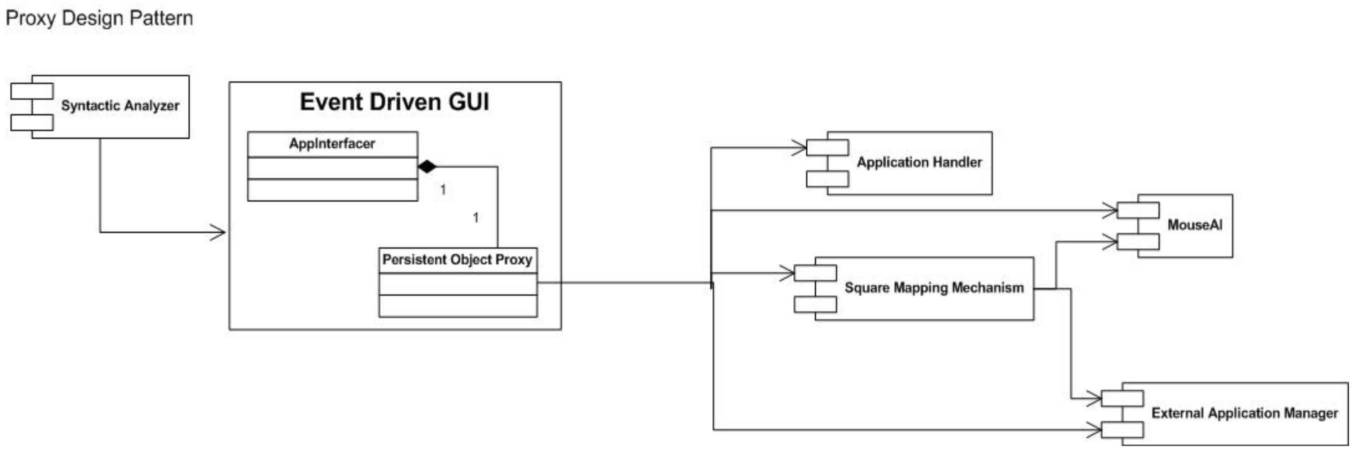


Figure 26. **Proxy Design Pattern**

The proxy design pattern (Figure 25) consist of a main entity (proxy) that controls access to real subjects [18], the real subjects hold methods that are invoked through the proxy to enhance security and handling of objects. The client requests a service destined to the real subjects through the proxy and this one delegates the requests.

The *Syntactic Analyzer* acts as the client while the *Event Driven GUI* component acts as the Proxy mechanism, inside this component the class *AppInterfacer* acts as a proxy for the requests of the client, these requests are then served by a second proxy that corresponds to the *Persistent Object Proxy* class that is the one that makes the final delivery of requests to the real subject. Note that the *Persistent Object Proxy* also plays the role of real subject when related to the class *AppInterfacer* that acts as its proxy, in this way the system utilizes this 2-layered proxy mechanism to deliver requests from the *Syntactic Analyzer*.

3.3.7 Observer Design Pattern

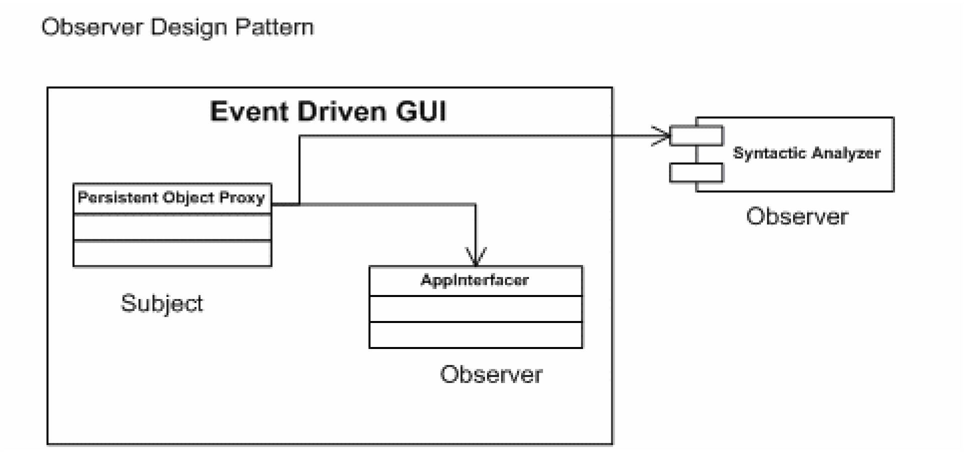


Figure 27. **Observer Design Pattern**

The observer design pattern (Figure 26) is based on defining a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically [19].

The *Persistent Object Proxy* class holds most of the instances for the real subjects of the invocation system. Whenever the state of these entities has been changed, both the class *AppInterfacer* and the class *Syntactic Analyzer* get notified of these new states. In this way the class *Persistent Object Proxy* acts as the Subject while the *AppInterfacer* and the *Syntactic Analyzer* act as the registered Observers.

3.3.8 Factory Design Pattern

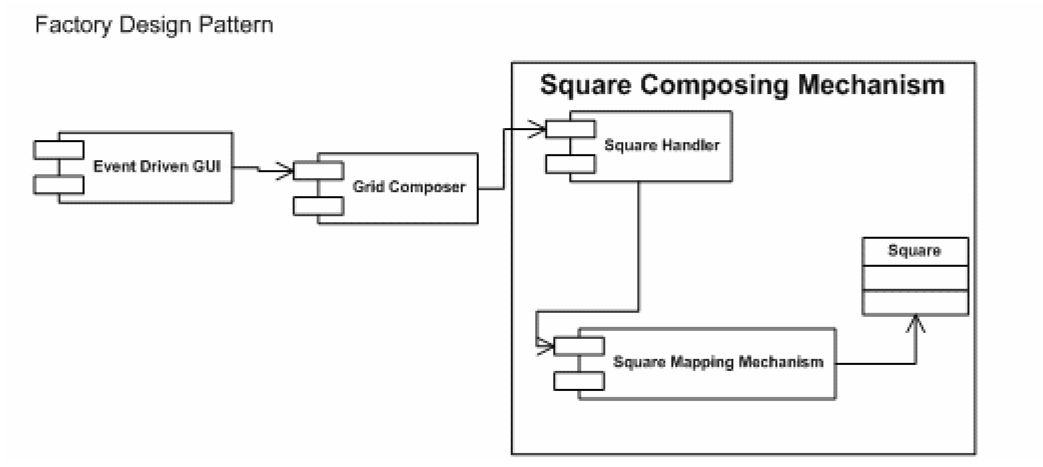


Figure 28. **Factory Design Pattern**

The Factory Design Pattern (Figure 27) consists of an interface for creating an object, letting subclasses decide which class to instantiate [18].

The best example of a factory design applied in our application is that of grid creation, when a grid is created, the *Event Driven GUI* component delegates the request to the *Grid Composer* which holds the factory method that instantiates objects of type square individually and returns a collection of these that form a full grid. The Square Composing Mechanism is used to create each individual square that is placed on the grid, however the *Grid Composer's* factory method is the one in charge of positioning the squares, labeling them and grouping them into a grid, in this way falling under the category of factory, while the grid being the product of the factory method.

3.4 Control Patterns

During the execution of an object-oriented program, the action of invoking a method to execute is known as a control transfer. A method invocation sequence keeps track of all the control transfers occurred during program execution [20].

Most of the Control Patterns [20] that are common in our system fall in the category of Complex Control Patterns since there are log patterns that exist to execute a command in a specified way, this type of log patterns are more abundant in the *Syntactic Analyzer* component do to its strict rules that a command invocation must follow, most of the invocations follow a invocation path from source to sink. Complex Control Patterns are mostly formed of Simple Control Patterns. Perhaps the most abundant Simple Control pattern is the Sequence Pattern. Invocations to methods that fall under this

category are done in sequence, synchronously [20].
A specialized sequence pattern known as the Consecutive Pattern is also present in some types of invocations. The Consecutive Pattern occurs when a sequence of invocations take place to a single method class (The class that holds the invoked method). Perhaps the component that experiences most Consecutive Pattern invocations is the *MouseAI* component that has a high level of interaction. In our language definition we incorporate a loop command, that when invoked it can trigger a Control Pattern behavior known as the Loop pattern. Our loop command loops the preceding command N times (specified by the user); Making invocations for one command to occur N times thus triggering a loop that could be measured as a Loop-N pattern [20].

Compound Control Patterns [20] are also present, especially in the *Syntactic Analyzer* component, where invocation decisions must be made in order to satisfy the languages grammar. Most Compound Control Patterns are in the form of a sequence, where each invocation of the sequence is dependant on a set of rules, in our case we check the syntax of the command and decide what components to invoke in the sequence so that a grammatical rule is followed properly.

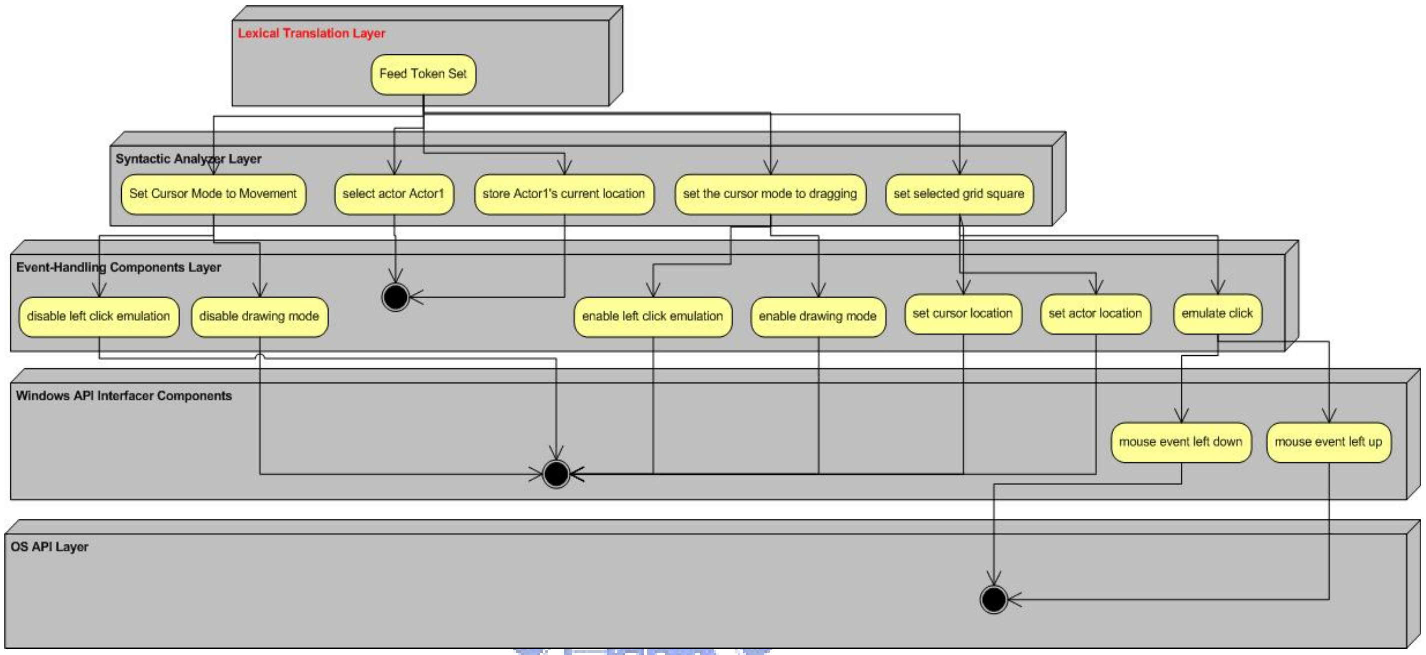
3.5 Layered Invocation Scheme

Command decomposition and processing in the system occurs in a 5-level invocation scheme. The five layers involved in this invocation scheme are the Lexical Translation Layer, Syntactic Analyzer Layer, Event-Handling Layer, API Interfacing Layer and the OS API Layer. The highest layers present invocations with the highest level of abstraction. Invocations at these layers have a longer processing life and do not occur as much as in the lower layers. On the other hand lower layers present invocations with the lowest level of abstraction. Invocations at these layers have a shorter processing life and occur with much more frequency than at the higher invocation layers. The behavior previously described is do to the highly task-specialized component design used that forces the distribution and breaking down of command handling calls to different layers of processing. Lower layers increase in complexity because of the higher number of invocations that are involved. On the contrary, higher layers posses a lower invocation level.

A command's processing life is directly proportional to its nature, since different commands result in different execution paths [20], so the nature of the command directly affects what layers its invocations traverses.

3.5.1 Layered Sequential Invocation Example

The following example is of a valid command utilized in the system. We take an inside look at all the different types of invocations that take place as the command gets processed as it transverses the above described layer structure (Figure 28) in a top-down manner.


Lexical Translation Layer
Feed Token Set
Syntactic Analyzer Layer
Set Cursor Mode to Movement
select actor Actor1
store Actor1's current location
set the cursor mode to dragging
set selected grid square
Event-Handling Components Layer
disable left click emulation
disable drawing mode
enable left click emulation
enable drawing mode
set cursor location
set actor location
emulate click
Windows API Interfacer Components
mouse event left down
mouse event left up
OS API Layer

Recognized command by the speech engine: *dragactor Actor1 to coordinate 5,6*

Figure 29. **Example of Sequential Invocation**

When the command enters the Lexical Translation Layer it gets processed in to a single Token Set that gets sent to the Syntactic Analyzer Layer through a single invocation.
During the parsing at the *Syntactic Analyzer*, the sub-structure belonging to "*dragactor*" is found. This substructure gets further parsed until the first invocation to the Event Handling Layer is found:

"set the cursor to movement"

This Invocation to the Event Handling Layer is then separated in to two invocations executed on the Windows API Interfacer Layer:

"disable left click emulation"

"disable drawing mode"

These two invocations simply set two values inside the Windows API Interfacer Layer to a *Boolean* value and their execution ends in this layer.

Back at the Syntactic Analyzer Layer, the parsing continues and the next sequence of invocations to the Event Handling Layer is found:

I.“select actor Actor1”

II.“store Actor1’s current location”

III.“set the cursor mode to dragging”

Invocation (I) will trigger the Event Handling Layer to simply invoke its internal component that handles actors, this invocation will set the current actor in the system to “Actor1”. No further processing is required thus the next invocation at the Syntactic Analyzer Layer gets processed.

Invocation (II) will trigger the Event Handling Layer to simply invoke its internal component that handles actors, this invocation will store the current actor location in the system for later referencing. No further processing is required thus the next invocation at the Syntactic Analyzer Layer gets processed.

Invocation (III) triggers two invocations that occur in sequence inside the Event Handling Layer:

A. “enable left click emulation”

B. “enable drawing mode”

These two invocations get served at the Windows API Interfacer and simply set two values inside the Windows API Interfacer Layer’s *Mouse AI component* to a Boolean value of “true” in order to enable the dragging mode, ending their execution at this layer.

After this invocation, back in the Syntactic Analyzer Layer, the parsing continues and the next invocation to the Event Handling Layer takes place when a match of the word “To” occurs during the parsing. The corresponding invocation under the next left derivation is:

“set selected grid square”

This invocation to the Event Handling Layer triggers the following invocations in the Windows API Interfacer Layer:

I.”set cursor location”

II.”set actor location”

III.”emulate click”

Invocations (I) and (II) get executed at the Windows API Interfacer's *Mouse AI component*. (I) sets the cursor location to a specified position in this case *"5,6"* thus dragging the actor to that position. (III), on the other hand, invokes the following API calls at the OS API Layer in a sequential order:

"mouse event left down"

"mouse event left up"

These API calls perform a mouse click by first holding the left mouse down and then releasing it. This is an example of an Invocation that traversed all the existing layers from top to bottom in order to achieve completeness.

Back in the Windows API Interfacer Layer invocation (II) gets handled locally without the need of interfacing with other layers, but instead utilizing the local *Actor Handling Component* to set the actor to its new location.

3.6 Conclusion

In this chapter the proposed system is introduced and studied in great detail from a high to low level perspective. The different architectural views are analyzed and the behavioral interactions amongst the entities that compose the system are discussed in detail. Overall this chapter provides the reader with a detailed and comprehensive view of all the features, processes and interactions that occur transparently when the user interacts with the system for interfacing applications with recognizers or interacting with applications. This chapter is structured in a task oriented manner, organizing it by separating the different tasks done by the system and going through each of them in detail to enrich the reader's perception of the overall interfacing environment and its sequential command processing behavior.

The following chapter introduces and discusses the proposed system's internal language definition.

CHAPTER FOUR
Interfacing Script Language Definition

4.1 Introduction

This chapter describes the language definition of the script language used in this system for users to interact with the interface interfacing environment that references applications. This chapter unveils every aspect of the designed language, its data types, rules, syntax description, and the interpretation steps involved in command processing.

4.2 Data types and Syntax

Types limit the values that a variable can hold or that an expression can produce, limit the operations supported on those values, and determine the meaning of the operations. Strong typing helps detect errors at compile time [12, 13]. The Interfacing Script Language separates data types into:

-Identifiers that are used to refer the system's internal variables used to reference the GUIs of the target applications

-Constants that have a binding time that occurs statically prior to compilation thus containing pre-defined values that cannot be reassigned

-Operators that are the key non-terminal symbols used to decide the pattern of expansion of the parsing routes. Providing commands the flexibility that allows their use to deal with multiple interaction situations (Expansion Paths)

-Separators and Terminators, being the elements used by the lexical analysis process as a guide to break down the command input stream by collecting characters into logical groupings known as lexemes[13]

In the following subsections, we elaborate each one of these data types as well as their common use in the language.

4.2.1 Identifiers

In this section variables which can be performed a binding during runtime are introduced. These variables can be changed during the course of execution through dynamic binding by means of assignments.

Square

It is a graphical reference that is introduced by the user to associate locations of static interaction zones. A variable of type square is associated to a location (*integer, integer*) and a graphic rectangle in the system.

Instance limit: from 0 to 999 (excluding instances created in grids)

Coordinate

A graphical square reference used in grids that is auto-generated by the system. The type coordinate is associated to a location (*integer, integer*) and a graphic rectangle in the system.

Instance limit: from 0 to 999

Actor

The actor type is composed of a location (*integer, integer*) and a graphical label (*string*) in the system and is used for the reference to dynamic content of applications. .

Instance limit: from 0 to 999

ActorProfile

An actor profile is the entity that actors are organized by. It is defined by the user prior to actor creation. Actors when created are auto-assigned into their corresponding profiles.

Instance limit: from 0 to 999

App

Represents the name of a target application (*String*) and is introduced by the user.

Instance limit: unlimited (limited by computing power)

Stage

It is used to reference the different GUIs used in an application and is used as the entity to organize and group sets of squares.

Instance limit: from 0 to 999

Grid

The Grid type corresponds to user-defined grids that represent a collection of objects of type square in the system. Grids are used to add reference through localization to GUIs.

Instance limit: from 0 to 999

Number

The number type maps directly to integers and are used to specify coordinates and revolutions of command loops.

Instance limit: unlimited (limited by computing power)

Text

Dictation text (*string*) spoken by the user and is used to emulate typing on the keyboard.

Instance limit: unlimited (limited by computing power)

4.2.2 Constants

In this section variables which have a binding time that occurs prior to runtime are introduced. These variables are constants, with predefined values assigned through static binding that cannot be reassigned during the course of execution.

Direction

Holds predefined constant values of: North, South, East, West, Northeast, Northwest, Southeast, and Southwest. *Direction* type is used in conjunction with the drag and move commands to specify cursor orientation.

Pattern

Hold constant values that are predefined by the system that represents geometric moving patterns. Used for dragging the cursor in a predefined manner by executing a batch of instructions that represents the moving pattern.

Distance

The distance constant holds predefined constant values of: very short, short, normal, long, very long. The distance constant is used for moving and dragging of the cursor, restricting the length of a mouse movement.

Speed

The speed constant holds predefined and constant values of: very slow, slow, fast, and very fast. The speed constant is used when moving or dragging the cursor, restricting the speed of a mouse movement.

clickType

The clickType constant specifies the type of click to perform when the click command is invoked, it holds predefined values of rightclick, doubleclick , and leftclick.

Boolean

Holds the predefined values of true or false and it's used with the commands *showGrid, setDrop, setVisualAide* and *usingActor*.

VisList

Used for referencing the system's visual-aided lists that contain a listing of available stages, grids, squares, actors, applications and actor profiles.

4.2.3 Operators

Operators are the base that our syntactic analysis process utilizes to produce its leftmost derivations as it traces the parse tree structure in a top-down manner (section **4.4.2**). They are the key non-terminal symbols that are used to when deciding pattern of expansion of the parsing routes [13]. Our system makes use of three different operators that are the ***to, by*** and ***loop*** operators, bellow is an overview of them and how they are used in the interfacing script language.

Operator *to*

The "***to***" operator is used in conjunction with action and assignment statements to specify assignment of locations to identifiers.

The use of ***to*** operator is restricted for commands ***move, drag,*** and ***set,*** for the use with right operands of data type ***square, coordinate*** *and* ***distance,*** and for the use with left operands of data type ***actor, squares or coordinates*** **or** ***null***.

Operator *by*

The "***by***" operator is used for assigning modes to its left operand. It is used in conjunction with action statements to specify desired speed, distance, moving pattern, and click type commands.

The use of "***by***" operator is restricted for commands ***move*** and ***drag,*** for the use with left operands of data type ***direction, square, coordinate or actor****, and* for the use with right operands of data type ***distance ,clickType,*** and ***pattern.***

Operator *loop*

The "*loop*" operator allows for looping a command an specific number (*number type*) of times. Its left operand is always a command. Its right operand is the number of times to loop the command, if not stated its default is set to one loop.

The loop operator is restricted for the use with right operands of data type ***number***.

4.2.4 Separators and Terminators

The following elements are used by the lexical analysis process as a guide to break down the input stream by collecting characters into logical groupings known as lexemes [13].

Terminator *then*

The terminator *then* is used for conjunction of statements and delimits one statement from the others.

Syntax

[stmt]→(***then***)→[stmt]→

dragActor <u>*actor x*</u> to <u>*coordinate y*</u> ***then*** *click* <u>*square x*</u>

“dragActor actor1 to 10,4 ***then*** *click rotate”*

Here two action statements (dragActor, click) are delimited by *then*.

Terminator *times*

The terminator **“*times*”** specifies the end of a loop command, and it is placed after the *number* identifier.

Syntax

[stmt]→(***loop***)→(***times***)→

drag direction x by *distance y loop number* ***times***

“drag south by very long then loop 2 times”

Here the drag command gets executed *number* times

Separator *Comma*

The separator **“*comma*”** separates x and y values of a coordinate identifier.

Example

dragActor actor x to *coordinate y*

“dragActor tree to *2,9”*

In the coordinate 2,9 the comma separates the left value(2) which is interpreted as an **x** and the right value(9) which is interpreted as a **y**.

4.2.5 Reserved Words

The following list of words are reserved by this language definition, the use of them as values of identifiers will create conflicts and an exception with its respective error message will be displayed by the system.

send

clearConsole

undoPhrase

$

to

by

loop

Double spacing (will be interpreted as single)

then

times

4.2.6 Input Element Classification

The allowed input elements of the language definition are classified into white space and tokens. Tokens are further classified into the following types:

Identifier

Command Statement

Separator

Operator

4.3 Semantics

This section introduces the semantic meaning of each command statement, rules, and the data type compatibility issues of the language.

4.3.1 General Static Semantics

The type compatibility rules that are used for compile time analysis are generalized and resumed bellow:

I. A token of type command must be followed by a token of type operator or identifier or stand alone only.

***II. A token of type command being used by an operator must comply with the operator rules* (section 4.2.4)**

III. A token of type operator must be followed by a token of type identifier only.

IV. A token of type identifier must be followed by a token of type operator or stand alone only.

***V. A token of type identifier being used by an operator (right or left operand) must comply with the operator rules* (section 4.2.4)**

VI. The last token of a sequence of tokens is always of type command or identifier only.

VII. A token following a valid token of type command must comply with the command type compatibility definition* (section 4.3.2) *for the respective command.

4.3.2 Command Statements

The definition of the different command statements of the language and their type compatibility rules are stated bellow. In the event of breaking any of the rules specified, the system triggers an incompatible type exception with its corresponding warning to the user, halting command execution.

Assignment Commands

Assignment Commands focus on assigning values to system internal identifiers.

setDistance

The *setDistance* command is used to set the distance that the mouse cursor will use during user command-driven movement.

Usage:

setDistance to <u>distance x</u>

"setDistance to <u>short</u>"

The distance gets set to short

The *setDistance* command is only compatible with types **distance** and the operator ***to***.

setDragSpeed

The *setDragSpeed* command is used to set the speed that the mouse cursor will use during user command-driven movement.

Usage:

setDragSpeed *speed x*

"setDragSpeed fast"

The speed of the mouse cursor is set to fast, when the cursor moves through commands it will do so at a faster pace.

The *setDragSpeed* command is only compatible with type **speed**.

addActorProfile

The *addActorProfile* command is used for creating an actor profile. It is a stand-alone command thus it does not interact with any identifiers or operators.

Usage:

addActorProfile

A new actor profile gets created by the system

The *addActorProfile* command can only be used as stand-alone.

eraseProfile

The *eraseProfile* command is used for deleting an existing specified actor profile.

Usage:

eraseProfile *actorprofile x*

"eraseProfile Profile1"

The specified actor profile is erased

The *eraseProfile* command is compatible with the type **actorprofile** only.

addActor

The *addActor* command is used for creating an actor to a specific coordinate or square, it can also be used as a standalone command where the actor's location is chosen automatically by the system.

Usage:

addActor

The actor gets instantiated to system selected location.

addActor to coordinate x

"addActor to 3,5"

The actor gets instantiated to the user specified location

The command *addActor* is compatible with the type **square** and **coordinate**.

eraseActor

The *eraseActor* command is used for deleting an existing specified actor.

Usage:

eraseActor actor x

"eraseActor Actor1"

The specified actor is erased

The *eraseActor* command is compatible with the type **actor** only.

setActor

The *setActor* command is used for fixing an already instantiated actor to a corresponding *coordinate* or *square*.

Usage:

setActor actor x to *coordinate x*

"setActor Actor1 to 10,3"

The actor gets fixed to the coordinate 10,3 of a grid.

The *setActor* command is only compatible with the type **actor, coordinate**, **square** and the operator ***to***.

resetActor

The *resetActor* command is used for re-setting the location of the currently selected actor to its previous location.

Usage:

resetActor

The current actor is fixed to its previous location.

The *resetActor* command is a standalone command.

setDrop

The *setDrop* command is used for disabling the drop of drag commands. Performing a drop if set to true (system's default), and a not performing a drop when set to false.

Usage:

setDrop *boolean x*

"*setDrop* *false*"

Triggers the system to disable the drop event when a drag is executed, thus hanging on the object that is being dragged.

The *setDrop* command is only compatible with the type **Boolean**.

setList

The *setList* command is use to give focus to the system's visual-aid lists, displaying them graphically.

Usage:

setList *VisList x*

"*setList* *actorlist*"

Trigger the system to give focus to the specified list in this way will display it to the user.

The *setList* command is only compatible with the type **VisList**.

setVisualAide

The *setVisualAide* command when set, allows the user to see visual aid content such as labeling of buttons, and grid coordinates. When not set, the system simply displays bounding boxes with out a textual description.

Usage:

setVisualAide *boolean x*

"setVisualAide *true"*

Trigger the system to enable visual aid in the application, will textually identify interfacing objects.

The *setVisualAide* command is only compatible with the type **Boolean**.

showGrid

The *showGrid* command when set, allows the user to hide or show the present displayed grid if any.

Usage:

showGrid *boolean x*

"showGrid *false"*

This will trigger the system to hide the present grid.

The *showGrid* command is only compatible with the type **Boolean**.

usingActor

The *usingActor* command is used for enabling and disabling actor interaction.

Usage:

usingActor *boolean x*

"usingActor *false"*

This will trigger the system to disable all interaction with actors, hiding them from the user.

The *usingActor* command is only compatible with the type **Boolean**.

Action Commands

The command type “action” focuses on interacting with application system’s interfacing content, performing actions that directly affect the target application.

drag

The drag command is used for dragging of the mouse cursor, applied to type **direction**, **pattern, speed** and **distance**. When invoked the respective mouse movement will occur (depending on operand) and simultaneously activating the left click of the mouse, in this way emulating a drag.

Usage:

drag <u>direction x</u> by *<u>distance x</u>*

“drag north by <u>Long</u>”

The mouse is “dragged” in the specified direction.

drag <u>pattern x</u> by *<u>distance x</u>*

“drag <u>spiral</u> by <u>short</u>”

Again a “dragging” occurs, but this time it follows a predefined dragging pattern and performs each segment of the pattern at short distance, resulting in a small spiral.

The *drag* command is only compatible with the type ***directon***, ***pattern,speed, distance*** and the operator ***by***.

dragSquare

The dragSquare command is used to drag a specific square to a specified coordinate or square or by a specified moving pattern or in the specified direction by the defined distance and by the selected speed.

Usage:

dragSquare *square x* to *coordinate x*

"dragSquare *tree* *to* *7,3"*

The cursor is focused on the specified square and then performs a drag to the given destination.

dragSquare *square x* *direction x* *by* *distance y*

"dragSquare *tree* *northeast* *by* *verylong*"

The cursor is focused on the specified square and then performs a drag in the given direction moving at the specified distance.

dragSquare *square x* *by* *pattern x* *by* *distance x*

"dragSquare *tree* by *zigzag* *by* *medium*"

The cursor is focused on the specified square and then performs a drag in the given pattern moving at the specified distance.

The *dragSquare* command is compatible with the type ***square, coordinate, direction, distance, pattern, speed*** and the operators ***by*** and ***to***.

dragCoordinate

The *dragCoordinate* command is the same as the *dragSquare* with the exception that is used to drag the type ***coordinate***. When invoked, the specified coordinate will be dragged to the specified coordinate or square, by the defined pattern, or in the specified direction by the defined distance and by the selected speed depending on the usage.

The *dragCoordinate* command is compatible with the type ***square, coordinate, direction, distance, pattern, speed*** and the operators ***by*** and ***to***.

dragActor

The *dragActor* command is the same as the dragSquare with the exception that is used to drag the type ***actor***. When invoked, the specified actor will be dragged to the specified coordinate or square, by the defined pattern, or in the specified direction by the defined distance and by the selected speed depending on the usage.

The *dragActor* command is compatible with the type ***actor, coordinate, direction, distance, pattern, speed*** and the operators ***by*** and ***to***.

click

The *click* command triggers a click at the current cursor position and is used in conjunction with the operator *by* and the identifier *clicktype* only.

Usage:

click

When executed, a click will be triggered on the current cursor location.

click by *clicktype x*

"click by rightclick"

When executed, a click will be triggered on the current cursor location in the specified mode.

The *click* command is compatible with the type ***clicktype*** and the operator ***by***

***click*Square**

The *clickSquare* command triggers the cursor to first move to the square location specified by square x, and then triggers a click in the fashion specified by the *clicktype* identifier, if any.

Usage:

clickSquare square x by *clicktype x*

"clickSquare house by rightclick"

When executed, the cursor will move to the corresponding square's location and trigger a click of the type specified (if not specified it will click as the default click (*leftclick)*)

The *clickSquare* command is compatible with types ***square***, ***clicktype*** and the operator ***by.***

***click*Actor**

The *clickActor* Command functions in the exact way as the *clickSquare command* with the exception that Actors are being clicked.

The *clickActor* command is compatible with type ***actor***, ***clicktype*** and the operator ***by***.

clickCoordinate

The *clickCoordinate* Command functions in the exact way as the *clickSquare* command with the exception that the types being clicked are Coordinates.

The *clickCoordinate* command is compatible with type ***coordinate***, ***clicktype*** and the operator ***by***.

move

The *move* command is used for moving the mouse cursor, and it's applied to variables of type square, coordinate and direction and operators *to* and *by*.

Usage:

move to <u>*typesquare x*</u>

"move to <u>house</u>"

When invoked the cursor moves to the specified square location

move to <u>*typecoordinate (x,y)*</u>

"move to <u>10,3</u>

When invoked the cursor moves to the specified coordinate location

move <u>direction x</u> by <u>distance y</u>

"move <u>southeast</u> by <u>verylong</u>"

When invoked the cursor moves in the specified direction by the specified amount

The *move* command is compatible with type ***square, coordinate, direction*** and operators ***to***, ***by***.

pasteText

The *pasteText* command is used for pasting text into a focused text field.

Usage:

pasteText <u>*text x*</u>

"*pasteText* <u>*Hello*</u>"

The text variable is copied into the clipboard and pasted into the current focused text field.

The *pasteText* command is only compatible with the type ***text***.

clearText

The *clearText* command is used for deleting text of a text field.

Usage:

clearText then *loop* <u>3</u> times

Three characters from the focused text field's text are deleted.

The *clearText* command is a stand-alone command.

sendKey

The *sendKey* command is used for emulating a keyboard stroke on the specified key.

Usage:

sendKey <u>*text x*</u>

"*sendKey* <u>*Y*</u>"

The system emulates a keyboard stroke of the letter Y.

The *sendKey* command is only compatible with the type *text*.

openDeveloper

The *openDeveloper* command is used for entering the developing mode of the system through speech whenever is not possible to interact with the system's GUI directly through the mouse.

Usage:

openDeveloper

The Developing mode is opened by the system

The *openDeveloper* command is a stand-alone command.

send

The *send* command is used for sending the stream of text already spoken to the lexical translator, in order to begin the command processing.

Usage:

send

The already spoken stream of text is sent to the lexical translator.

The *send* command is a stand-alone command.

undoAll

The *undoAll* command is used for deleting all phrases spoken in order to reset the stream of text to emptiness if a mistake was made.

Usage:

undoAll

The already spoken stream of text is cleared.

The *undoAll* command is a stand-alone command.

undoPhrase

The *undoPhrase* command is used for deleting the last phrase spoken in order to reset the stream of text to its previous content.

Usage:

undoPhrase

The last phrase spoken is deleted from the stream of already spoken text.

The *undoPhrase* command is a stand-alone command

captureIt

The *captureIt* command is used for capturing free windows (pop-ups) into the system.

Usage:

captureIt

Here the most front "free" window will be captured.

The *captureIt* command is a stand-alone command.

Selection Commands

The selection commands are used for switching visual interfacing content.

***selectStage*:**

The *selectStage* command selects the current stage, when executed all the registered squares for that stage are loaded by the system.

selectStage <u>stage x</u>

"selectStage <u>stage 1"</u>

Selects the stage identified as "stage 1" and loads all its corresponding squares, painting them onscreen.

The "*selectStage*" Command is only compatible with the type ***stage***.

***selectGrid*:**

The *selectGrid* command selects a grid for display, when executed all the registered squares for that grid get loaded and painted on screen by the system.

selectGrid grid x

"selectGrid paintgrid"

Selects the grid identified as "paintgrid" and loads all its corresponding squares.

The "*selectGrid*" Command is only compatible with the type ***grid***.

***selectActorProfile*:**

The *selectActorProfile* command is used to set the current actor profile, when invoked a list of actors corresponding to the selected profile is loaded and registered actors under this profile are placed graphically on screen.

selectActorProfile actorprofile x

"selectActorProfile profile 1"

Selects the actor profiled identified as "profile 1" and loads its corresponding actors onto screen.

The "*selectActorProfile*" Command is only compatible with the type ***actorprofile***.

***selectApplication*:**

The *selectApplication* command is used to focus a target application. If invoked, all the stages, grids and actor profiles corresponding to the specified application are automatically loaded by the system.

selectApplication app x

"selectApplication paint program"

Selects the application identified as "paint program" and loads all its corresponding stages, grids, and actor profiles.

The "*selectApplication*" Command is only compatible with the type ***app***.

4.4 Lexical and Syntax Analysis

4.4.1 Lexical Analysis

The lexical analysis is in charge of extracting lexemes from the input string in order to produce the corresponding tokens that are returned to the caller (syntax analyzer) one lexeme at a time. During this process, the input string is separated whenever a separator or terminator is found and discarded, in this way forming sub-segments of the stream. These sub-segments, then in sequence of appearance are further processed by removing white spaces and any other characters that are not relevant to the meaning of the program, in this way decomposing the stream into tokens. The resulting tokens are grouped in token sets and fed to the syntax analyzer in a token-set per token-set basis.

The Lexical Analyzer distributes its chores to four sub-programs, one in charge of getting the next stream input through an event handling function, another one in charge of building lexemes as described above, another tokenizing sub-program that takes care of removing non-relevant characters and finally a subprogram that takes care of the recognition of reserved words, constants and identifier names. The later with the purpose of validating the content of the data types of the command in question by looking them up in their corresponding tables to make sure they exist in the system and that no reserved word are being used.

Once a stream is processed at the Lexical Analysis, it's passed down to the Syntax Analysis phase in the form of token sets, each set representing an individual command.

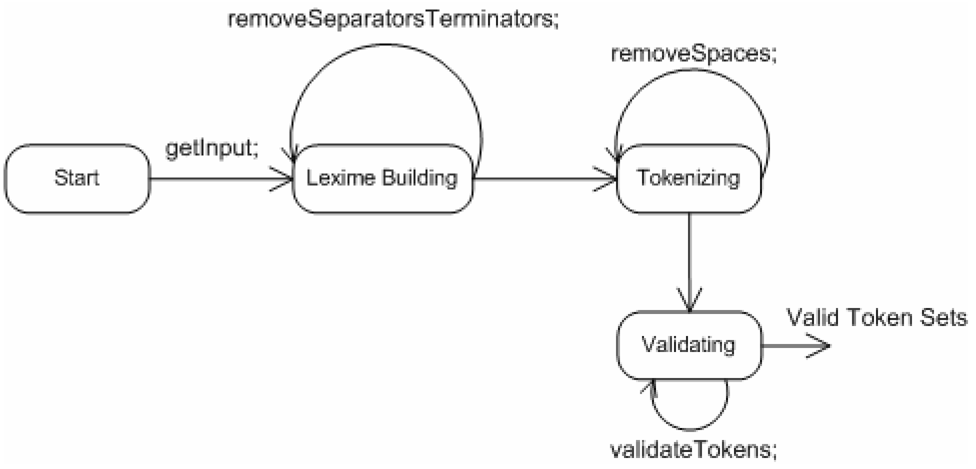


Figure 30. **Token Patterns Transition Diagram**

Figure 29 depicts the state transition diagram that provides the representation of the token patterns implemented for building the Lexical Analyzer of the system.

4.4.2 Syntactic Analysis

The syntactic analysis tasks include the checking of the syntax of the input program, providing error messages when syntactic errors are encountered and recovering from them to continue analysis on the input program [13]. In our approach a complete parse tree structure is traced in a top-down manner, rather than generated. As this structure is parsed, the placement of tokens is checked against the rules established in section **4.3.1**, making sure that no identifier or command is used as an operand in a way that it breaks an operator's rules. Successful parsing leads to the interpretation of commands into the target program that is executed to perform system related interactions to achieve the purpose of the command in question.

In our syntactic analysis we trace a leftmost derivation, tracing the parse tree in preorder, beginning with the root and following branches in left-to-right order. Expanding non-terminal symbols to get the next sentential form in the leftmost derivation, basing the expansion route on the type of the non-terminal symbol. Do to the simplicity and recursive nature of the language's grammatical rules, our approach implements a recursive descent parser rather than utilizing parsing tables to accomplish the syntactic analysis, in this way assuring that the next token represents the left most token of input that has not been used in the parsing, this token is compared against the first portion of all existing right hand sides of the non-terminal symbol, selecting the right hand sides where a match is found. Parsing expansion is directly affected by the number of operators used in the command in question, since operators are the non-terminal symbols that are used as checkpoints when selecting a parsing path.

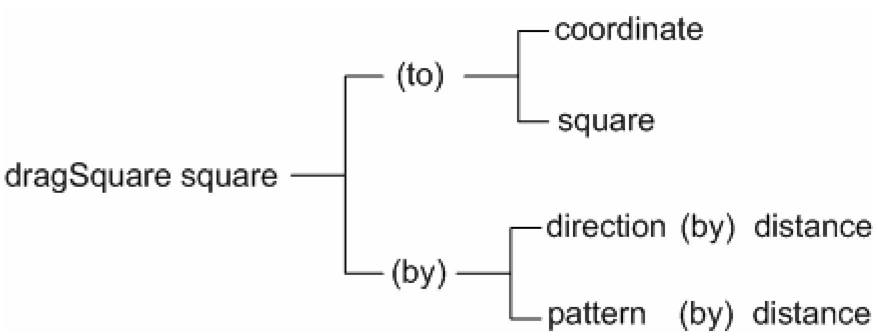


Figure 31. **Leftmost derivation parsing tree of the dragSquare command**

Figure 30 shows the corresponding parsing structure of the command "dragSquare", depicting all the possible right hand sides that can be matched to be the next leftmost derivation for any of the operators that compose the command.

4.5 Conclusion

This chapter presented the language definition that is implemented in our system to facilitate complex task interactions for providing flexibility and efficiency to overcome challenges of current interfacing mechanisms between speech recognizers and application systems. The chapter summarizes the structure of this language, the rules the language revolves around when parsing and checking syntax of commands, its compatible data types and a detailed view of each command and their compatibility constrains. This chapter provides readers with the basic information needed to understand how an interfacing mechanism can be constructed to simplify the voice commands used in the speech recognition process to interface to the application system while expanding their functionality.

CHAPTER FIVE

Application Examples and Evaluation

5.1 Introduction

In this chapter, we illustrate the procedures that are adapted to interface a target application using the proposed framework. This chapter will elaborate on scenarios that provide high and low level detail on the overall functional view of the system, and create a contrast with the challenges found on the other interfacing approaches addressed in Chapter 1. We will then evaluate our proposed system against the challenges and limitations imposed in the earlier chapter.

5.2 Procedures for Using the proposed Interface Framework to Interface an Application to a Speech Recognizer

The interfacing procedure is separated into multiple steps, each with its own set of scenarios. The following steps give the details:

- Step 1) Registration and Interfacing of the Target Application
  - The first step involved in interfacing an application to a speech recognizer through the proposed framework is to register a desired application into the system. Once registered, we create the visual interfacing environment by drawing reference zones on the transparent interface that lays on top of the application's GUI, in this way referencing application content such as buttons, containers and menus through the graphic registration of grids and squares, separating this content into stages that represent the different GUIs of the application.
    - This step is required every time a new application is interfaced with a speech recognizer
    - As an example to illustrate this step, we register the application BestWise 編輯手 version 2004 and reference its GUI's content

- Step 2) Recognizer Integration
  - The second step involves the integration of a chosen recognizer into the system by programming the recognizer's API calls that are used to start, setup and handle the

recognizer and as well as the calls involved in retrieving recognition content in the system's specialized recognizer interfacing component, in this way interfacing the recognizer into the system.

  - This step is only required when no recognizer has previously been integrated into the system.
  - As an example to illustrate this step, we interface our system with Microsoft's Speech-Recognizer V.6.1

- We then proceed to define the grammar definition file that the speech recognizer uses to recognize and interpret spoken content by incorporating the rules that were established in our language definition, allowing the checking of the syntax to occur twice.
  - This step is done each time a recognizer is integrated with the system.

- The vocabulary used to reference the target application in the first step is incorporated into the recognizer's grammar definition each time a new application is interfaced by the system in order to allow the recognition of the identifier names used for the graphical zones that compose the visual interfacing environment.
- Each time a recognizer is interfaced with the system, an XML file is composed that contains associations of commands in the format used in the recognizer's grammar with those commands of the language definition of the system. This resource is used as the source for translating spoken content into streams compatible with the system's syntactic analysis process.

■ Step 3) Macro Composition

- Once an application is properly interfaced with a speech recognizer, we compose a set of macro commands to simplify user interaction with the interfaced environment by wrapping complex and repetitive tasks into short, reusable context free commands.

■ Step 4) Interaction Evaluation

- In our last step, interactions with the speech-driven environment built in previous steps are performed in order to evaluate the system's overall functionality against the criteria built from the current approaches' challenges and limitations.

5.3 Step 1) Registration and Interfacing of the Target Application - “BestWise 編輯手 version 2004” is used as an example

Object registration uses the constrained genericity stated in [21, 22] and the “See-Trough Interface” paradigm for users to perform application software registration. In the following subsections, we depict the use of BestWise visual authoring software as an example to perform the application software registration.

5.3.1 Registering an Application

Our proposed framework allows for the coexistence of multiple applications, each application that is registered into the system must have its unique identifier that can be given by the user or set by the system with the name of the application’s executable file. Figure 31 depicts a snapshot of loading the application software into our proposed interface framework system for registration. The operation steps are labeled as 1, 2, 3, and 4 in the figure.

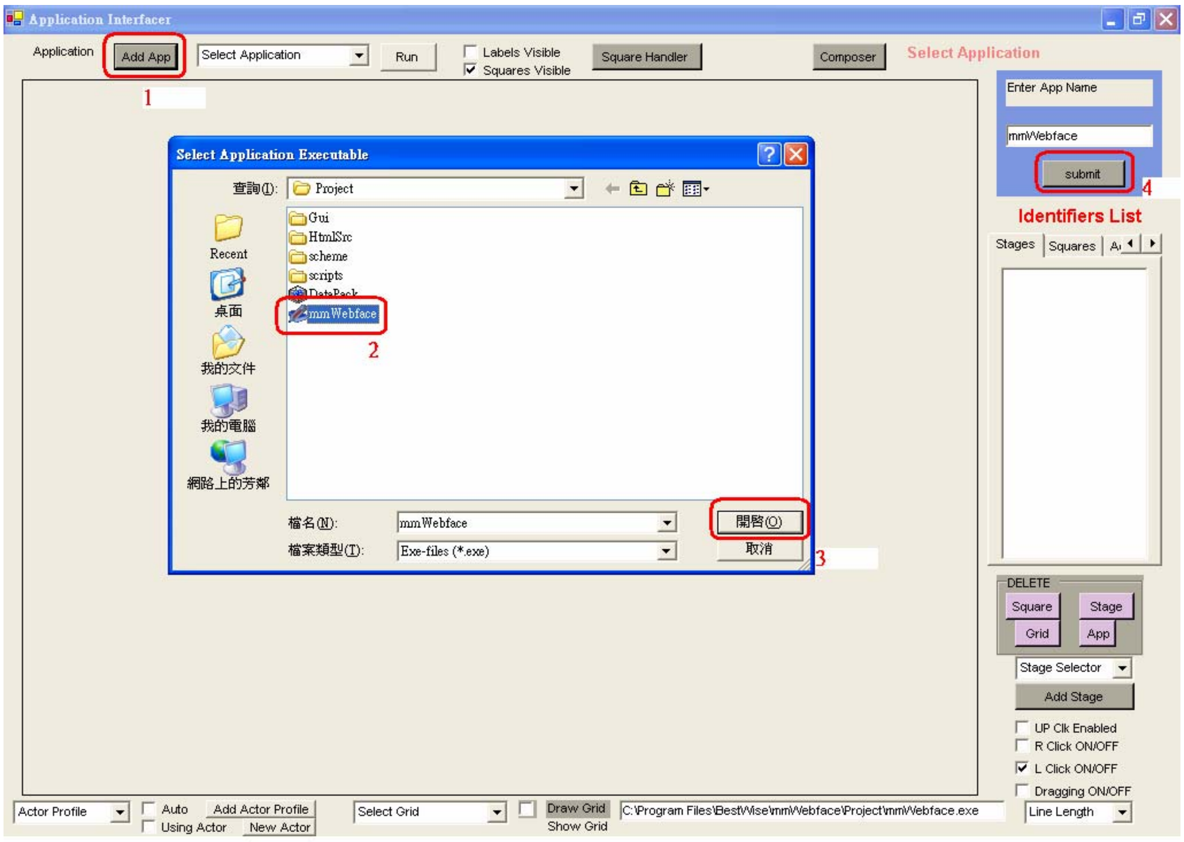


Figure 32. **Registering an Application**

Table 1 presents the description of the steps involved in the process of registering an application into the system.

Table 1. **Registering an Application**

Steps	**High-Level View**	**Low-Level View**
Select add application option	Administrator user presses the Add App button*(1)*, triggering an executable selection dialog to appear	System displays the executable selection dialog
Select Desire Executable	The administrator user then selects the desired application *(2)* and presses ok*(3)*. The application registration form appears containing the name of the selected application, the user may modify the name and continue the registration process by clicking on the submit button*(4)*	The *StoreHandler* component calls its internal method to register non-graphical objects, a directory is created with the name of the application under the *Bin* directory

5.3.2 Registering a Stage

Stages are used for organizing and separating the square zones that reference the different GUIs of the target application. Each stage has a name given by the user. To register a stage the user simply selects the register stage option and gives it an identifier name. Figure 32 depicts a snapshot of stage registration for the BestWise visual authoring software. The operation steps are labeled as 1, 2, and 3 in the figure.

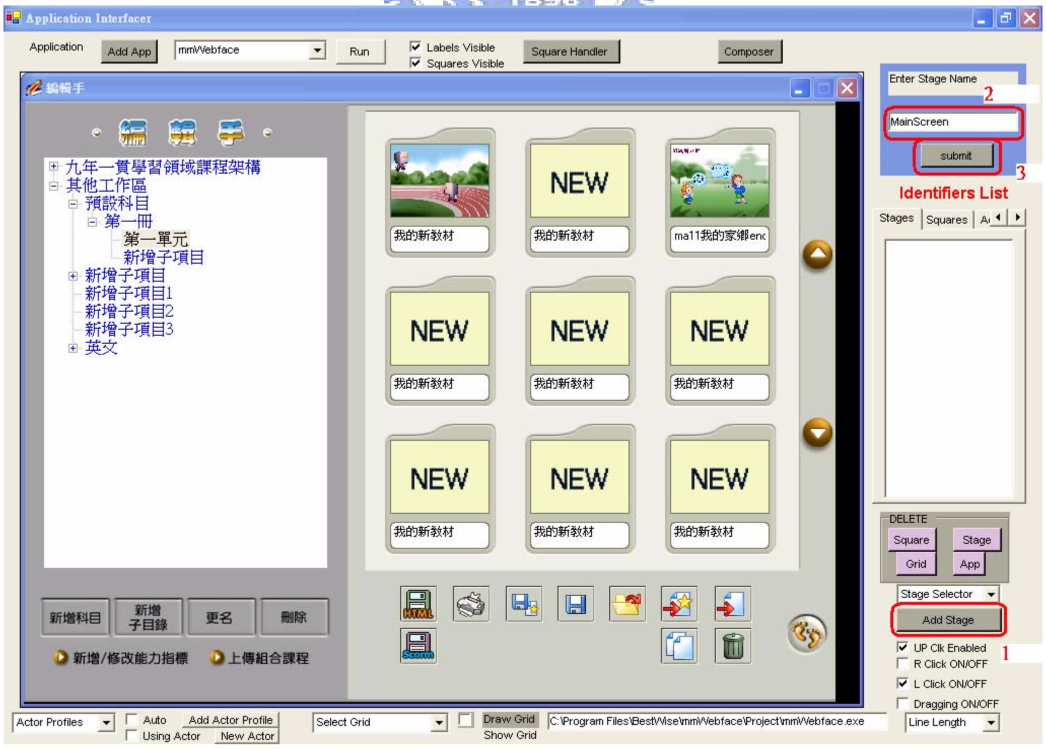


Figure 33. **Registering a Stage**

Table 2 presents the description of the steps involved in registering a stage into the system.

Table 2. **Registering a Stage**

Steps	**High-Level View**	**Low-Level View**
Select add stage option	The Administrator user presses the Add Stage button, Triggering the stage registration dialog to appear	The system makes visible the stage registration dialog
Enter Reference Name	Administrator user enters a name to reference this stage in the textbox found at the stage registration dialog. To finalize the process the user presses on the submit button of the stage registration dialog	The *StoreHandler* component calls its internal method to register non-graphical objects, inside this method a directory is created with the name of the stage, placing it inside the *Stages* directory under the current application's path

5.3.3 Registering a Grid

Grids are composed of auto-generated square objects and are used to reference panes and containers of the target application, allowing for a localized referencing through coordinates. Each grid has a name given by the user, and they are registered through drawing on the desired interaction zone. Figure 33 depicts a snapshot of grid registration for the BestWise visual authoring software. The operation steps are labeled as 1, 2,3,4 and 5 in the figure.

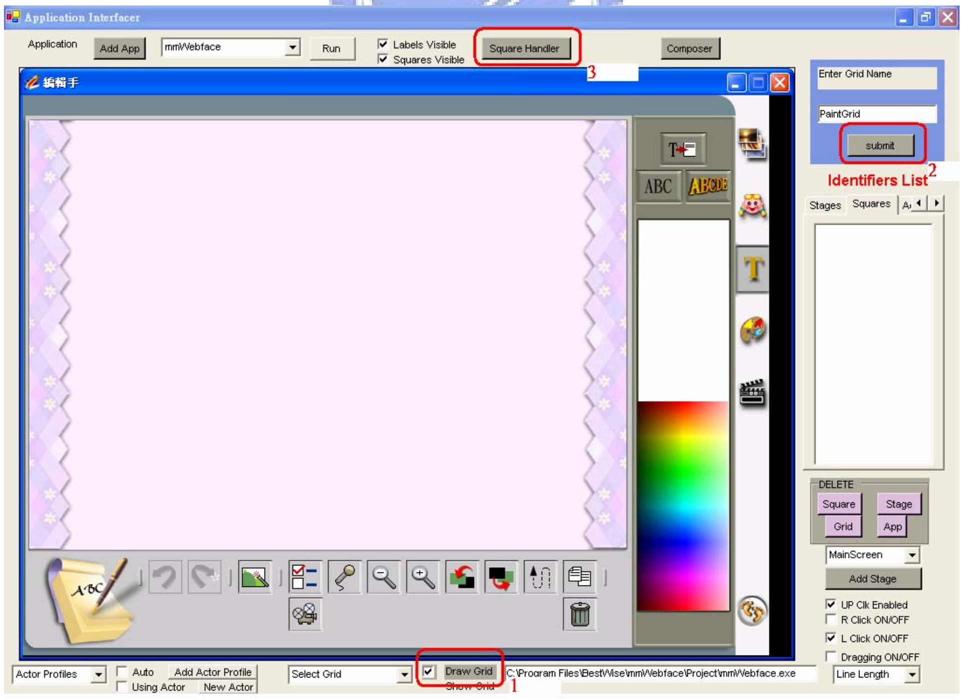


Figure 34(a). **Registering a Grid**

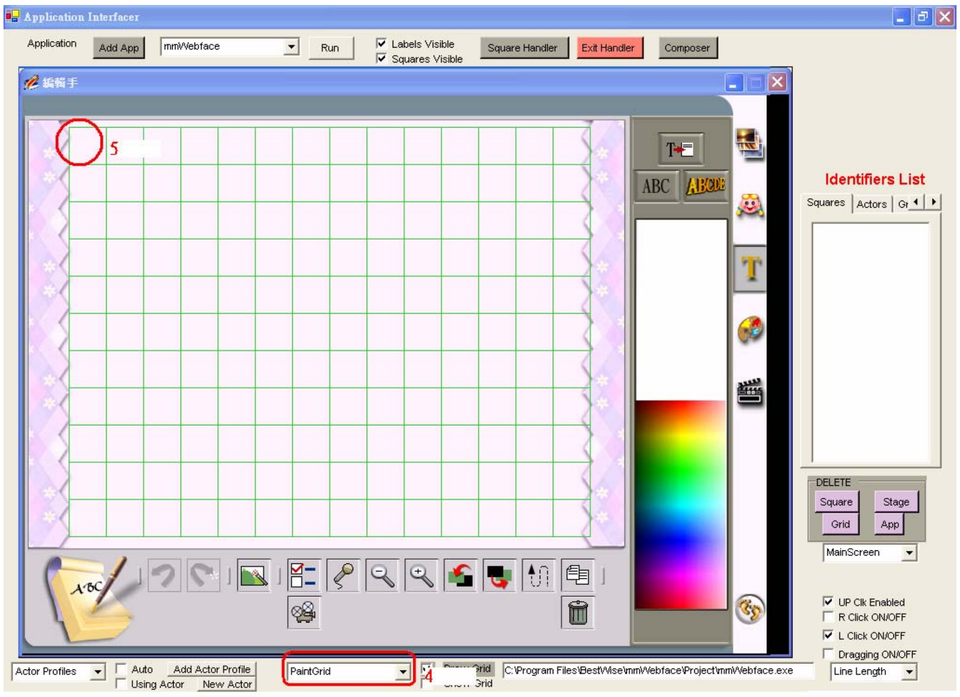


Figure 35(b). **Registering a Grid**

Table 3 presents the description of the steps involved in the composition of a grid into the system.

Table 3. **Registering a Grid**

Steps	**High-Level View**	**Low-Level View**
Register Grid	Administrator user clicks on the *draw grid* checkbox, triggering the display of grid registration dialog where the user enters a name for the grid and presses on the submit button*(3)*	System displays the grid registration dialog and the *StoreHandler* component invokes its internal method to register non-graphical objects, inside this method a directory is created in the *Grids* directory under the current application's path, with the name of the grid specified by the user
Enter Visual Environment Composition Mode	Administrator user presses on the *Square Handler* button	The system modifies the current environment by allowing all features used in composition of environments
Draw Grid	Administrator user selects the previously registered grid from the grid listbox and drags the mouse to draw the grid	The *Grid Composer* component is invoked. The component's design is based on the Factory Design Pattern [19] acting as a dynamic factory to create objects of type square, each one being a coordinate of the grid. In the *Grid Composer* a method is invoked to create each square object. Each square is then stored to file invoking the *Square Mapping Mechanism's* method in charge of square information handling to store the squares attributes to file

5.3.4 Registering a Square

Squares are referencing objects used to interface buttons or zones of applications, each square has a name given by the user and they are registered by drawing them on top of the interaction zone to interface. To register a square one must first select the desired stage to associate the square with. Figure 34 depicts a snapshot of square registration for the BestWise visual authoring software. The operation steps are labeled as 1, 2,3 and 4 in the figure.

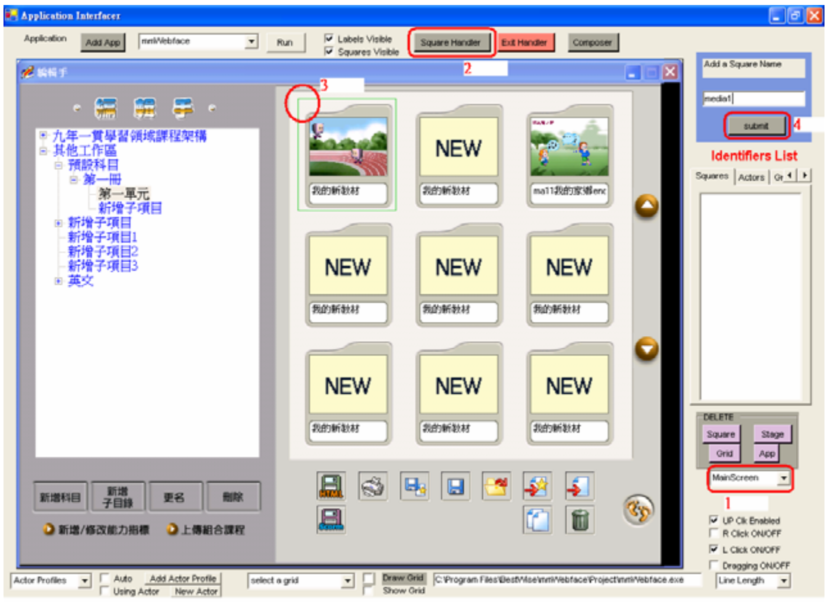


Figure 36. **Registering a Square**

Table 4 presents the description of the steps involved in registering a square object into the system.

Table 4. **Registering a Square**

Steps	**High-Level View**	**Low-Level View**
Select Stage	Administrator user selects the desired stage	System loads all reference squares associated with that stage if any and draws them on screen
Enter Visual Environment Composition Mode	Administrator user presses on the *Square Handler* button	The system modifies the current environment by allowing all features used in composition of environments
Drawing a square	Administrator user draws a square on top of the desired zone of the application(3). Square registration dialog appears where a name to identify this square is entered by the user(4)	The system makes the square registration dialog visible to the user
Submit Data	Administrator user presses on the *submit* button	A square object gets stored to file by invoking the *Square Mapping Mechanism component's* method in charge of square information

		handling, storing the square's given name, its coordinates and dimensions into a file under the corresponding stage directory

5.3.5 Registering an Actor Profile to Create Actors

Actor Profiles are used to separate and organize actors that reference dynamic content of the different GUIs of the target application. Actor Profiles have a name that is auto-assigned by the system. Table 5 presents the description of the steps involved in registering an actor profile into the system.

Table 5. **Registering an Actor Profile**

Steps	High-Level View	Low-Level View
Add Actor Profile	End user speaks the "addactorprofile" command	The *StoreHandler* component calls its internal method to register an actor profile. Inside this method a directory gets created with a name that is auto-assigned by the system, this folder is placed inside the *Actors* directory under the current applications path

As one can see the process is very similar to the one used to register stages or grid objects because the same method is used to register all these type of objects, however differentiating the way the different objects are treated by applying constrained genericity [22].

5.4 Step 2) Interfacing the Recognizer- Microsoft's Speech Recognizer V.6.1 is used as an example

The Microsoft's Speech Recognizer V.6.1 is selected as an example for the target recognizer and its Development Kit [16] is used to set-up handle the basic functionalities of the speech engine and to receive recognition content from it. First of all, we need to implement the grammar specification program according to the speech recognizer's grammar. The specification program consists of interaction commands and the vocabulary used (Figure 35) to interface the target application [23]. Next, we need to define the recognizer's grammar; a set of rules is defined through extensible markup language (Figure 36), in order for the speech-recognition engine to recognize spoken commands. This set of rules is used by the speech-recognizer to validate recognized words, restricting the possible words or sentences chosen during the sound recognition process. In this way a syntactic analysis is performed twice, once on the speech-recognition engine side and secondly in the syntactic analyzing processes of our proposed system.

Each rule defines separate sentence context, composed sentences are formed through the reference of existing rules to form a compound rule. Decisive selection is applied to provide more flexibility in speech and avoiding re-definition of context. These rules are structured in the exact same way the commands are defined in the system's language definition, obeying the definition token placement rules for operators and commands.

```
<!--BELLOW IS ALL APPLICATION DEPENDANT, SO ITS THE ONLY
PART OF THIS DEFFENITION THAT SHOULD BE MODIFIED-->
<!--variables for use with outsorting tool-->
<RULE NAME="sqrs">
<l> <P>save</P> <P>saveas</P> <P>player</P> <P>new</P>
<P>normal</P> <P>duplicate</P>    Continues…..
```

Figure 37. **Recognition Vocabulary**

```
<RULE NAME="dragsquare" TOPLEVEL="ACTIVE"> <P>dragsquare</P> <o> <RULEREF NAME="sqrs"/> <o> <p>to</p>
                                            <l> <RULEREF NAME="sqrs"/> <RULEREF NAME="coord"/> </l> </o>
                                            <o> <o> by </o> <RULEREF NAME="mse"/> <o> <p>by</p>
                                            <RULEREF NAME="mse"/> </o> </o> </o> <o><RULEREF NAME="then"/>
                                            </o> </RULE>
```

Figure 38. **Composed rule definition that uses references to other lower-level rules**

The definition of a translation XML resource file (Figure 37) that associates commands in the recognizer's grammar format into context in the system's internal language is made. This translation resource allows for generic integration of other recognizers and languages into the system.
In this case, the translation was not necessary since the system's internal language matches the one specified at the speech recognizer; however it was done to illustrate this step.

```
<grammar>
<word NAME="Actor">Actor</word>
<word NAME="Profile">Profile</word>
<word NAME="boolean">boolean</word>
<word NAME="true">true</word>
Continues…..
```

Figure 39. **Translation Repository**

5.5 Step 3) Macro Command Registration

To simplify interaction with the recognizer, we define macro commands in the specification program for target application interaction. Without the support for macros, users will have to speak long sentences with complicated syntax to achieve a set of interaction tasks. For this reason we compose context-free macros to abstract multiple tasks into one reusable command that approximates to the user's natural way of speaking and enhances speech recognition's efficiency. Figure 38 depicts a snapshot of a macro command registration for the BestWise visual authoring software. The operation steps are labeled as 1,2,3,4 and 5 in the figure.

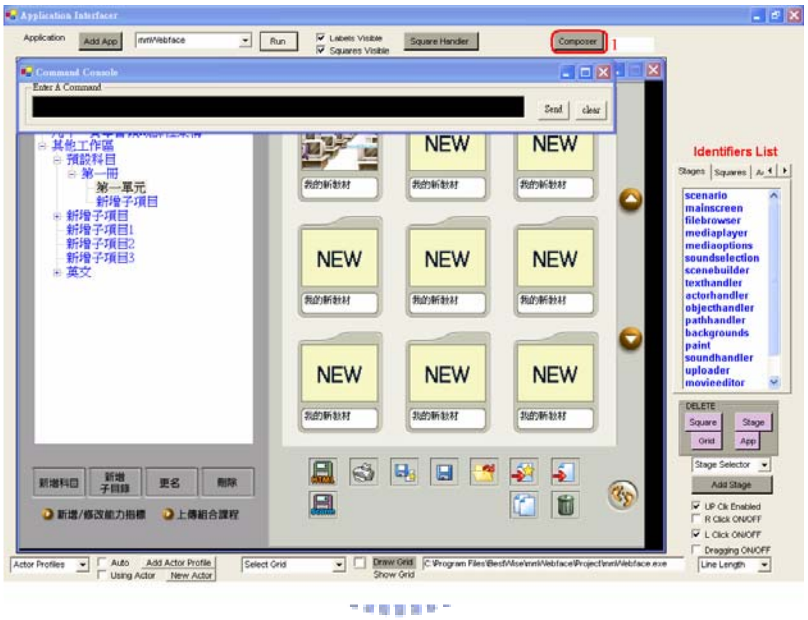
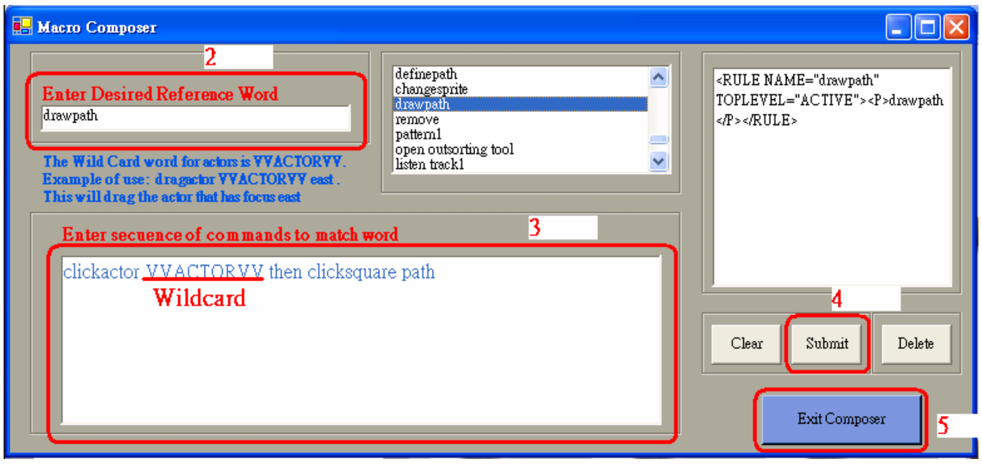


Figure 40(a). **Registering a Macro**



Enter Desired Reference Word

drawpath

The Wild Card word for actors is VVACTORVV.
Example of use: dragactor VVACTORVV east.
This will drag the actor that has focus east

definepath
changesprite
drawpath
remove
pattern1
open outsorting tool
listen track1

```
<RULE NAME="drawpath"
TOPLEVEL="ACTIVE"><P>drawpath
</P></RULE>
```

Enter secuence of commands to match word

clickactor VVACTORVV then clicksquare path

Figure 41(b). **Registering a Macro**

Table 6 presents the description of the steps involved in registering a macro command.

Table 6. **Registering Macro**

Steps	**High-Level View**	**Low-Level View**
Enter Macro Composer	End User enters the macro composer mode by clicking on the "composer" button*(1)*	The system loads the macro composing GUI
Define a Macro Command	End User first defines a keyword to identify the macro command. The user then composes the macro command, defining it in the system's internal language. In this case a macro utilizing wildcards is composed so that it can be applied to more than one actor*(3)*	The *Macro Composer* generates the XML structure that will be inserted into the speech engine's grammar definition so that the macro keyword can be recognized by the speech-recognizer
Submit Macro Command	End user presses the submit button*(4)* and exits the composer*(5)*	The *Macro Composer* stores the macro command in the macro command repository and regenerates the speech engine's grammar definition file, inserting the XML structure created for this macro command in the previous step

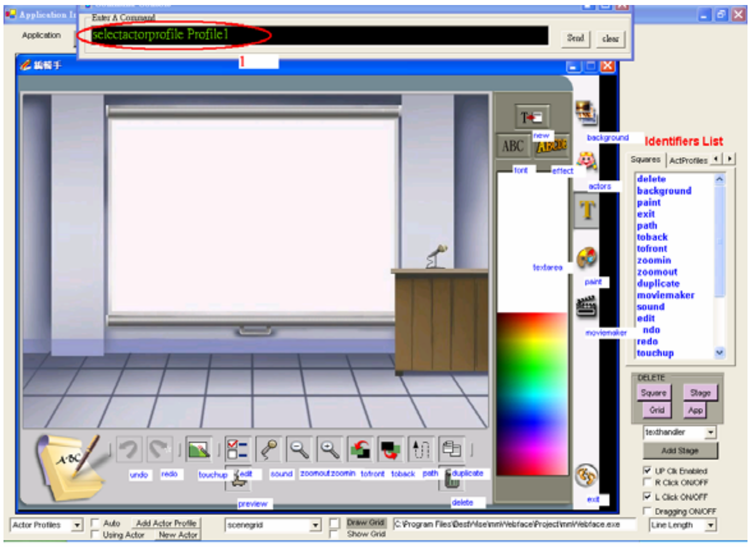
5.6 Step 4) Interacting with the Interfaced Environment

The following scenarios focus on overall user interaction with the predefined interfacing environment.

5.6.1 Registering an actor

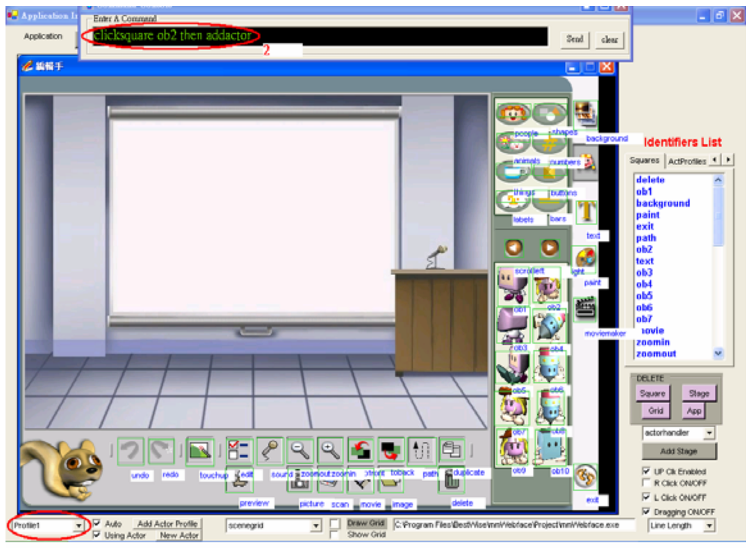
Actors are used to reference dynamic content of applications so that they can be interacted with through voice commands. In order to avoid the possible confusion, in our system, we organize actors into actor profiles to avoid displaying all actors of an application at once.

To create a new actor, one must select an actor profile to associate it with, and then speak the "add actor" command. Figures 39(a) 39(b) 39(c) depict the snapshots of runtime actor registration for the BestWise visual authoring software. The operation steps are labeled as 1 and 2 in the figures.



selectactorprofile Profile1

Figure 42(a). **Registering an Actor**



clicksquare ob2 then addactor

Figure 43(b). **Registering an Actor**

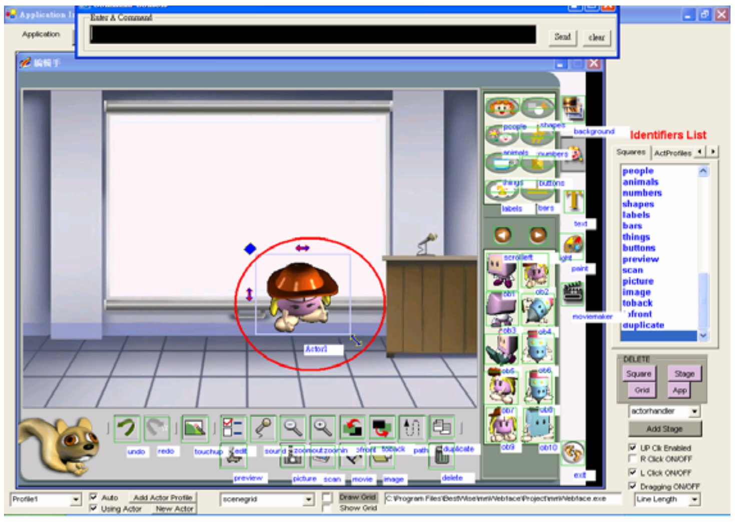


Figure 44(c). **Registering an Actor**

Table 7 presents the description of the steps involved in registering an actor.

Table 7. **Creating Actor**

Steps	**High-Level View**	**Low-Level View**
Select Actor Profile	End user speaks the *selectactorprofile Profile1* command	The *Event Driven GUI* component invokes the *Actor Handler* component where the selection of the actor profile takes place
Add actor	The user then speaks the “addactor” command. In this particular case the macro command “addactor ob2” is used that translates to a command written in the system’s internal language “clicksquare ob2 then addactor”.	The “clicksquare” command triggers an invocation to the *mouseAI* component through the *Event Driven GUI* component to set the clicking stile to “button interaction”. The square to click is selected, retrieving its coordinates through the *Square Mapping Mechanism* component that reads them from file. The *mouseAI*’s sendclick method is then invoked, performing a click on the specified coordinates. The *ActorHandler* component creates a visual label that contains the name of the actor and places it on the corresponding coordinates. The coordinates information and auto-assigned name of the actor is stored to file through the *StoreHandler* component

5.6.2 Dragging an Actor

To be able to drag an actor, a grid must be present on screen in order to assign an actor a new location. In the following scenario we first load a grid and then perform a “dragactor” command to drag an actor to a specific coordinate of the grid. Figures 40(a) 40(b) 40(c) depict the snapshots of dragging an actor. The operation steps are labeled as 1 and 2 in the figures.