Hybrid User Interfaces:
Design Guidelines and Implementation Examples

by

Sehyun Ahn

M.S. Civil and Environmental Engineering
Stanford University, 2000

Submitted to the Department of Civil and Environmental Engineering
in Partial Fulfillment of the Requirements for the Degree of

CIVIL ENGINEER IN CIVIL AND ENVIRONMENTAL ENGINEERING

AT THE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

FEBRUARY 2006

Signature of Author: _____
Department of Civil and Environmental Engineering
January 13, 2006

Certified by: _____
Steven R. Lerman
Professor of Civil and Environmental Engineering
Thesis Supervisor

Accepted by: _____
Andrew J. Whittle
Professor of Civil and Environmental Engineering
Chairman, Departmental Committee on Graduate Students

Hybrid User Interfaces:
Design Guidelines and Implementation Examples

by

Sehyun Ahn

Submitted to the Department of Civil and Environmental Engineering
on January 13, 2006 in Partial Fulfillment of the
Requirements for the Degree of Civil Engineer in
Civil and Environmental Engineering

ABSTRACT

A hybrid user interface is a new type of computer user interface that achieves high usability by combining features of graphical user interfaces and command line interfaces. The main goal of a hybrid user interface is to increase the efficiency of a system that is used to perform repetitive tasks. By adopting the string-based input mechanism of command line interfaces, users of a hybrid user interface are able to populate graphical components using only the keyboard, eliminating the inefficiency of the computer mouse for repetitive tasks. Especially, for applications that require repetitive tasks such as entering multiple data and managing system administration, a hybrid user interface enhances the efficiency of the system significantly. A hybrid user interface can be developed as a new application or can supplement an existing graphical user interface when the efficiency of the system is of major concern.

Thesis Supervisor: Steven R. Lerman
Title: Professor of Civil and Environmental Engineering

Contents

List of Figures

List of Tables

List of Examples

Chapter 1

Introduction

1.1 Overview

In modern computer systems, user interface refers to the series of actions, such as keystrokes with the computer keyboard and movements of the computer mouse, that the user executes to control the system, as well as the information – usually graphical, textual, and auditory – that the system presents to the user. There are various types of user interfaces in computing – for instance, batch interfaces, command line interfaces, graphical user interfaces, Web-based user interfaces, and so on. The design of a user interface significantly affects the amount of work the user must perform in order to provide the input for the system, interpret the output from the system, and learn how to use the interface. Usability is a quality attribute for the design of user interface, and it assesses how easy a particular user interface is to learn and use, considering the human psychology and physiology of the user as well as the following quality characteristics, which are further explained in the next chapter:

- Learnability
- Efficiency

- Memorability
- Error frequency and recovery
- Subjective satisfaction

The goal of this thesis is to review some of the most widely used user interfaces in computing, especially with regard to their design approaches and usability, and to propose a new type of user interface with increased usability and the design guidelines for developing such an interface.

1.2 Thesis Motivation

Among many user interfaces currently existing in computer systems, the graphical user interface and command line interface are the most widely used ones. Graphical user interfaces generally accept the user input via graphical interactive components by using various input devices, such as the computer keyboard and the computer mouse, and provide the graphical output on the computer monitor. Compared to other user interfaces, a graphical user interface is easy to learn, easy to perform and remember operations, and highly intuitive. However, it has the following limitations:

- Complicated menu structure: In most graphical user interface applications, available operations and options are located under different menus, which usually are based on an application-specific, multi-level hierarchy.

- Inefficiency: Compared to command line interface, a graphical user interface is relatively slow to perform tasks, and many advanced users find that they work more efficiently with a command-driven interface. In particular, a graphical user interface is extremely inefficient performing repetitive tasks, such as entering multiple data and managing system administration [Shaffer, 2004].

In command line interfaces, the user provides the input by typing a command string with the computer keyboard, and the system provides the output by printing text on the computer monitor. Experienced users of command line interface can perform tasks very quickly, and most operations can be executed in a consistent form. However, a command line interface has its own drawbacks:

- Hard to learn: A command line interface has steep learning curve and takes users a long time to learn the even basic operations.
- Hard to remember: A command line interface has low memorability, which means that it is hard to remember the operation and option commands after not using the interface for a long period of time.
- High error frequency: The exact commands for operation and option are not easy to remember and the command line interface does not check the command strings for syntax when they are entered. Although most command line interfaces provide lists of available commands that users can look up, referring to the list while entering the input significantly lowers the efficiency of the interface. Especially, it takes the beginners of the interface a long time to memorize all the available

commands, which could cause frequent input errors at the early stage of the learning curve.

This thesis presents the Hybrid User Interface, which is a new type of user interface in computing that utilizes the advantages of graphical user interfaces and command line interfaces, in an attempt to increase the usability of the user interface and solve some of the limitations described above. The main goal of a hybrid user interface is to improve the efficiency of graphical user interface – especially elements that are used to perform repetitive tasks – while maintaining its well-known advantages, by adopting features of the command line interface. The overall design of a hybrid user interface is mainly based on the design of typical graphical user interface, but with the added command line that takes string-based inputs from the user. These string-based inputs populate the graphical components very efficiently without using the computer mouse, which is time-consuming and ineffective in performing repetitive tasks.

1.3 Thesis Organization

In accordance with the research objective, scope, and approach already discussed, the organization of this thesis is as follows:

- Chapter 2: This chapter reviews the usability of user interfaces, including the definition and quality characteristics. In addition, usability considerations for

interface design and the concept of user-centered design – which is the design process based on the concept of usability – are explained.

- Chapter 3: This chapter discusses graphical and command line user interfaces, which are the two most widely used user interfaces in computing. Each interface is examined for its design approach, advantages and limitations, and most importantly, usability.
- Chapter 4: This chapter presents the new type of user interface, Hybrid User Interface, and its design principles, such as:
    - Design goal
    - Requirements
    - Component design
    - Interaction design
- Chapter 5: This chapter presents implementation examples for hybrid user interface systems.
- Chapter 6: This chapter concludes this thesis by summarizing the concept of hybrid user interface, presenting recommendations for the application of the interface, and discussing future direction in this area of research.

Chapter 2

Usability and Interface Design

2.1 Usability Definition

Part 11 of ISO 9241 standard [British Standards Institution, 1998] defines usability as "the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use". Effectiveness is defined as the accuracy and completeness with which specified users can achieve specified goals, and efficiency is the resources used to achieve the effectiveness. Satisfaction means the comfort and acceptability of the system to its users and other people affected by its use.

In user interface engineering, usability is a quality attribute for the design of user interface. It takes into account the human psychology and physiology of the user, and measures the quality of user's experience when interacting with the interface. Usability is important characteristics of the user interface as it assesses the interface for effectiveness, efficiency, and user satisfaction, which are directly related to the productivity of the user.

2.2 Usability Characteristics

Usability is characterized by the following five concepts that affect the user's experience with a specific interface and provide the basic framework for designing productive user interfaces.

2.2.1 Learnability

A user interface with high learnability allows its users, who have never interacted with the interface before, to easily learn how to use the software and accomplish basic tasks. The concept of learnability can be broken down into the following five components [Louis, 2003]:

- Familiarity: Users expect the interface to be structured in certain way, and the design of the interface should mirror this expectation properly. For instance, users expect the word processing application to have certain components, such as an application menu, formatting tools, a main area for writing, status bar, and so on. To build an interface with high familiarity, it is important to understand that the interface must meet the needs and expectations of novice users, who typically do not have technical expertise or training.
- Consistency: Users anticipate the interface to behave in a consistent way. Especially in a graphical user interface, users get confused if graphical components of the interface look different from one area to another. Therefore, it is essential that the operations and interactions must be done in a consistent

manner, following well-structured design components and patterns. A user interface with consistency also brings familiarity to its users.

- Generalizability: A user interface needs to contain some of the general and frequently-used elements and attributes with which its intended users are already familiar. Tab implementation in graphical user interface is one of the examples for generalizability. Most users are familiar with this functionality from using other similar user interfaces, so developers should try to incorporate it with their interfaces. To create a user interface with high generalizability, developers must research interfaces with similar functionalities, discover common design components among them, and incorporate those components into the interface being developed.
- Predictability: A user interface should work in the way its users expect. Since different users have different expectations due to various levels of experience and expertise, creating a universally predictable user interface is not an easy task. For instance, users of Windows applications have different expectations from the Apple system users when looking for the "Options" menu. An interface with good predictability is designed for its intended users and is designed to meet their expectations. In general, following the generalizability principle above helps design a predictable interface.
- Simplicity: It is obvious that simpler interface is easier to learn. However, advanced users may prefer the interface with more features and functionalities, which makes the interface not so simple. A good way to implement simplicity that can satisfy users with different demands is to provide multiple user settings, such

as the regular user setting and the expert user setting that can be customized by the user.

2.2.2 Efficiency

Once the user has learned how to use the interface, the next factor that affects usability is how quickly the user can perform a specific task. This is the efficiency of the user interface. In order to build an interface with high efficiency, developers need to interact with everyday users of the interface and implement features and functionalities that can increase the speed at which users perform particular tasks. One of the good examples of such features is the quick keys, or keyboard shortcuts, such as CTRL-C and CTRL-V that the users execute to copy and paste text in Windows application. In addition, consistency in the design of interactive components as well as the clear communication between the user and the interface via messages and feedbacks help creating an interface with high efficiency.

2.2.3 Memorability

Memorability means how easily a user can reestablish a given level of efficiency after not using a specific user interface for a significant period of time. Memorability is an important attribute of usability because users may not use the interface everyday, and developers should make sure that the interface is intuitive and easy to remember. One way of improving memorability is to use design components that users can easily associate with their intended functionalities. For instance, users assume that the diskette

icon on a Windows application means saving file by intuition, and therefore, they are able to remember it easily. An interface with high generalizability is likely to have good memorability, since its users keep coming across the same interactive components when they use other similar interfaces.

2.2.4 Error Frequency and Recovery

Most users turn away from applications with the greatest functionalities if they encounter too many errors using them. Unfortunately, there is no such thing as perfect application and users are bound to run into some occasional errors. Therefore, it is important to build a user interface that can minimize user errors. In case of an error, a good user interface must guide its users in the right direction and present them with the proper error message by following these guidelines:

- Existence of error messages: A user interface should notify its users of any errors, whether it is occurred due to the application's flaw or the way the user is using the application. If no error message exists in the interface, users cannot tell whether their operations are successful or not.
- Understandable error message: An error message should contain information that regular users, who are neither computer scientists nor programmers, can understand. An error message such as "Error 2EF3987D – Invalid Memory Location" does not help the user understand the nature of error or the meaning of the error message.

- Precise error message: An error message should be clear, and it must present information directly related to the particular error. If the error message is too general or vague such as "File System Error", it is difficult to pinpoint the cause of the error making it hard for the user to fix the problem.
- Correct error message: Developers must check each error message carefully and make sure that it provides correct information about the corresponding error.
- Consistent error message: Error messages should be presented in a standard and consistent way. It is important to have a standard format for error messages that includes elements such as a message or icon specifying that there is an error, description of the error, and procedures that the user can follow to fix or notify the problem.
- Helpful error message: Error messages should explain the user how to fix the error and how to proceed with the task. If the error is due to the system or application failure, the error message should provide a way to notify the programmer of the error.

As much as it is important to build an application with minimum errors, its user interface must follow the above guidelines and implement the useful error message system that can help users recover quickly from any errors they encounter.

2.2.5 Subjective Satisfaction

The concept of subjective satisfaction lies in the fact that users want the interface to be intuitive and behave in the way they expect. Although it is impossible to satisfy every

single user with a single interface, a user interface with high usability pleases most users most of the time. If a user interface has the learnability, efficiency, memorability, and good error recovery system, its users are likely to be satisfied with their overall experience when using the software.

In addition to the above factors, graphical user interfaces should have the look and the feel that are comfortable and familiar to the users. Unfortunately, most programmers are not designers, and often, some of the best functional user interfaces cannot satisfy their users due to the lack of sophistication, completeness, and attractiveness of the interface. One way to solve this problem is the cooperation of developers, designers, and usability experts from the early stage of the development process.

2.3 Usability Considerations

The first step to create a user interface with high usability is to ask a set of right questions to the intended users, so the developers can define the scope, functionality, and design of the interface. The followings are some of the representative questions that developers must ask before starting the development process. The answers to these questions can be acquired by conducting user and task analysis:

- Who are the intended users of the interface?
- What do the intended users need to do or want to do with the interface?

- What is the typical background – technical, professional, or demographic – of the intended users?
- What is the context in which the intended users work?

In addition, developers should work closely with the intended users throughout the development process and try to get their feedbacks by asking questions related to the characteristics of usability. The questions below are some examples and they can be answered by conducting user-focused requirements analysis, building user profiles, or performing usability testing:

- Can the users easily complete their intended tasks at their intended speed?
- How much training do the users need?
- What documentation or other supporting materials are available for the users?
- What are the types of errors that the users make most frequently using similar interfaces?
- How many errors do the users make when using similar interface? Can the users recover from errors easily?

2.4 User-Centered Design

In order to create a highly usable user interface, it is important to involve usability experts or intended users from the beginning of the design and development process. User-centered design is the most widely used approach that places the user at the center

of development. It builds a user interface that satisfies the characteristics of usability by observing the intended users' interactions with the interface and their performance with the intended tasks. User-centered design attempts to answer the questions about the usability consideration, and it uses the answers to drive the design process and improve the usability of the interface.

2.4.1 Design Principle and Process

The goal of the user-centered design is to involve and understand the intended users throughout the development process. According to ISO 13407, "Human-Centered Design Processes for Interactive Systems" [International Organization for Standardization, 1997], user-centered design is built on the following design principles:

- The active involvement of users
- An appropriate allocation of function between user and system
- The iteration of design solutions
- Multidisciplinary design teams

In addition, ISO 13407 provides guidance on achieving these design principles by incorporating user-centered design activities throughout the life cycle of design and development. There are four essential activities that need to be performed from the earliest stages of design process:

- Understand and specify the context of use

- Specify the user and organizational requirement
- Produce design solutions
- Evaluate designs with users against requirements

Following these design principles and activities assures that the development process fully reflects the views and needs of the intended users, which results in a user interface with high usability.

2.4.2 User Involvement

Involving the intended users actively from the beginning of the development is the key concept of the user-centered design. Users should be part of every development step, from the establishment of the requirement, to the delivery of the system. During the initial design process, developers must take the users' inputs by conducting surveys, interviews, and focus groups, and use the results to set up the preliminary requirements for the interface. If a prototype is available, users should test it and give feedbacks to the developers. Throughout the user interface design steps, users should participate in the series of usability tests and make suggestions for the interface. Any problems found at each testing should be notified to the developers and fixed before the final delivery is made. It is important that, during any user-involved testing, developers must observe users interacting with the interface and try to validate the assumptions about the intended users and analyze the intended tasks, workflow, and goals.

2.4.3 Iterative Design and Testing

Throughout the development life cycle of the user interface, the design process must be highly iterative, so that the design of the interface can be tested continuously by the developers and the intended users [Greenberg, 1996]. Figure 2.1 illustrates the iterative user interface design and development process.

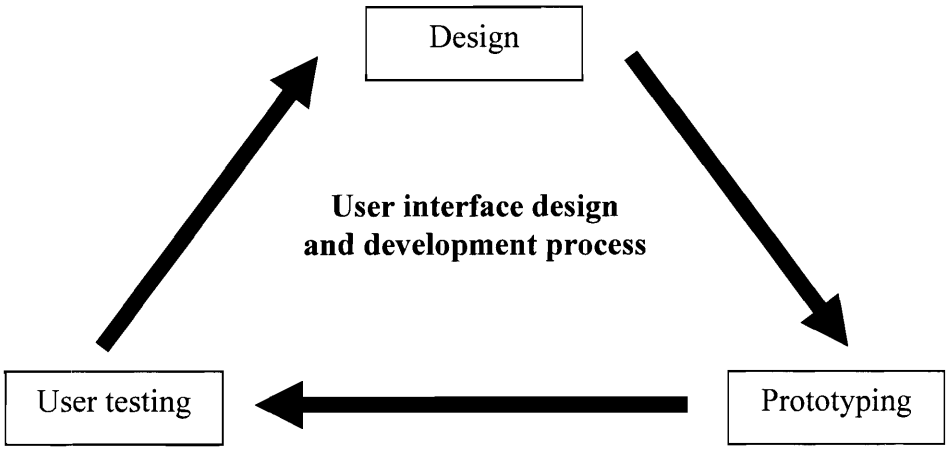

Figure 2.1 Iterative User Interface Design Process

Usability testing is a way to find out how well a user interface works for its intended purpose. It measures how well the users respond to a specific user interface in terms of learnability, efficiency, memorability, error frequency and recovery, and subjective satisfaction, by observing users interact with the interface in a realistic situation, interviewing users for their satisfactory levels, and discovering any errors and areas of improvement. A user interface needs to be evaluated throughout its life cycle, and especially during the early stage of design, developers should validate the requirements

and predict the initial usability. Prototyping is one of the early evaluation methods, which is cost-effective and capable of getting a lot of useful feedbacks from users. These feedbacks should be fed back to the design process before moving to the next stage of development. If a prototype is not available, paper mockup test can be conducted to get early feedbacks from users. Using the printouts of the interface, users can interact with the elements of the interface and comment on their impressions and expectations. Later in the life cycle, a full usability test for the interface should be conducted to assess how well the user interface meets the system requirements. Any findings at this stage could be used for next version of the system.

Chapter 3

User Interface in Computing

3.1 Overview

In modern computing, user interface refers to the series of actions that the user performs to control the system as well as the information the system presents to the user. Based on the input and output devices, interaction elements, and interaction style, there are many types of user interfaces available, and this chapter reviews the two most widely used interfaces in modern user interface engineering: graphical user interface and command line interface.

3.2 Graphical User Interface

Graphical user interface, or GUI, is a type of computer interface that takes advantage of the computer system's graphics capabilities to interact with the users and the underlying functionalities of the applications. In general, users provide the input for the system with the input devices such as the computer keyboard and the computer mouse, and the system returns graphical output on the computer monitor. Graphical user interface is composed

of various graphical components such as windows, frames, buttons, and menus. Figure 3.1 is an example of graphical user interface.

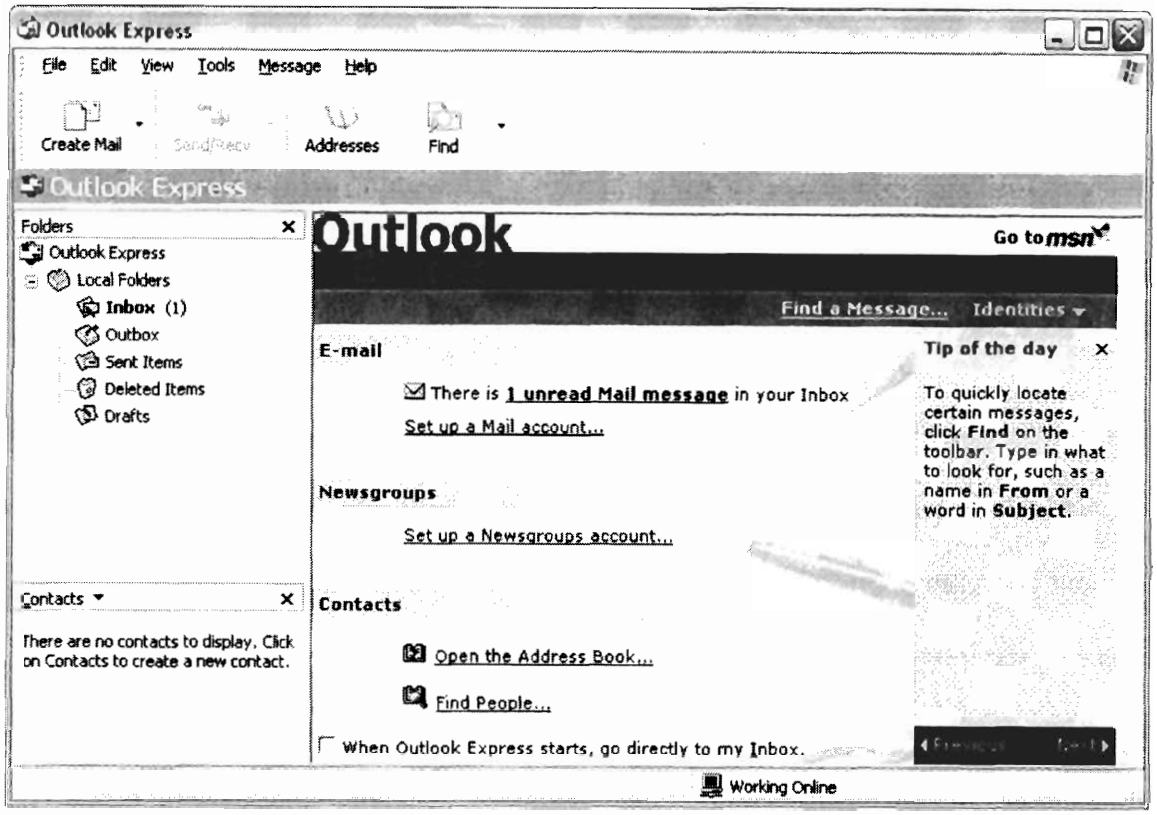

Figure 3.1 Example of Graphical User Interface

3.2.1 History of Graphical User Interface

The first concept of a graphical user interface began with the first real-time graphic display systems for computers: Ivan Sutherland's Sketchpad and the SAGE (Semi Automatic Ground Environment) project, which developed an automated control system for collecting, tracking, and intercepting enemy bomber aircraft. In the 1960s, Doug Engelbart, who at that time was inspired by the memex desk-based information machine

developed by Vannevar Bush, built the On-Line System that incorporated a mouse-driven cursor and multiple windows, as a part of the Augmentation of Human Intellect project at Stanford Research Institute. Later in the early 1970s, Engelbart's work led to the progress at Xerox PARC (Palo Alto Research Center), and the WIMP (Windows, Icons, Menus, and Pointer) paradigm was created and eventually appeared commercially in the Xerox 8010 system in 1981.

In 1984, the Lisa and Macintosh teams at Apple Computer, which included former members of the Xerox PARC group, released the Macintosh, the first commercially successful product to use a graphical user interface. Following the success of the Macintosh, several GUI systems such as DESQview, GEM (Graphical Environment Manager), and Amiga Intuition, had been developed before Microsoft released its first version of Windows system in 1985. With the enormous popularity of Windows 3.0 released in 1990, graphical user interfaces became the major interface choice for modern computers and variety of graphical user interface systems has been developed. Followings are some examples of modern graphical user interface systems:

- GEOS (Graphic Environment Operating System): Developed by Berkeley Softworks for the Commodore 64. Released in 1986.
- RISC OS (Reduced Instruction Set Computing Operating System): Developed by Acorn Computers for the 32-bit ARM (Acorn RISC Machine) based Acorn Archimedes. Released in 1987.

- NeXTSTEP: Developed by NeXT Computer for the NeXT line of computers. Released in 1989.
- OS/2: Developed by Microsoft and IBM to replace DOS. Released in 1987 and the later version (2.0) became the basis for Windows 95.
- BeOS: Developed by Be Incorporated for AT&T Hobbit-based computers. Released in 1991.
- NeWS (Network extensible Window System): Developed by Sun Microsystems in the late 1980s.
- X Window System: Developed by MIT as Project Athena for the Unix-based systems in the early 1980s.
- Mac OS X: Developed by Apple Computer and released in 2001. Primarily built on technology from NeXTSTEP.
- Windows Vista: Microsoft's next-generation operating system. Planned for release in late 2006.

3.2.2 Advantages and Limitations

A well-designed graphical user interface is intuitive and it presents menus and operations that are easy to learn. Most users can perform basic tasks without any experience with the interface or any type of training, and even computer beginners can learn how to use it relatively easily and quickly. Also, graphical user interfaces can free users from learning and memorizing complex commands that are used in a command line interface. Another advantage of graphical user interface is that it is easy to provide help to the users via

online support or built-in documentations, and its error messages are generally easy to understand and follow, enabling users troubleshoot the problem.

Compared to a command line interface, a graphical user interface is not very efficient. Finding and executing operations and options usually require certain number of mouse movements and mouse clicks, and if the interface is complex with many graphical components and multilayered menus, the execution time quickly adds up to the low efficiency and productivity. Especially, if the interface is used for certain tasks – such as system administration – that are highly repetitive, it can cause a serious efficiency problem.

3.2.3 Design Guidelines

In order to develop an effective graphical user interface that minimizes the above limitations, it is important to consider the following design guidelines [Stone, 2005]:

- Avoid lengthy menus and order menu items meaningfully
- Use menu names that reflect the menu items' functionalities
- Use consistent grammar, layout, and terminology
- Incorporate a logical grouping and sequencing of the interactive components

3.3 Command Line Interface

Command line interface, or CLI, is a way of interacting with computer systems by using a series of command strings that are entered as a line of text. Users provide the input for the system with the computer keyboard, and the system returns the output as text on the computer monitor. Command line interface presents a command prompt that indicates where a command should be entered. Figure 3.2 shows an example of command line interface.

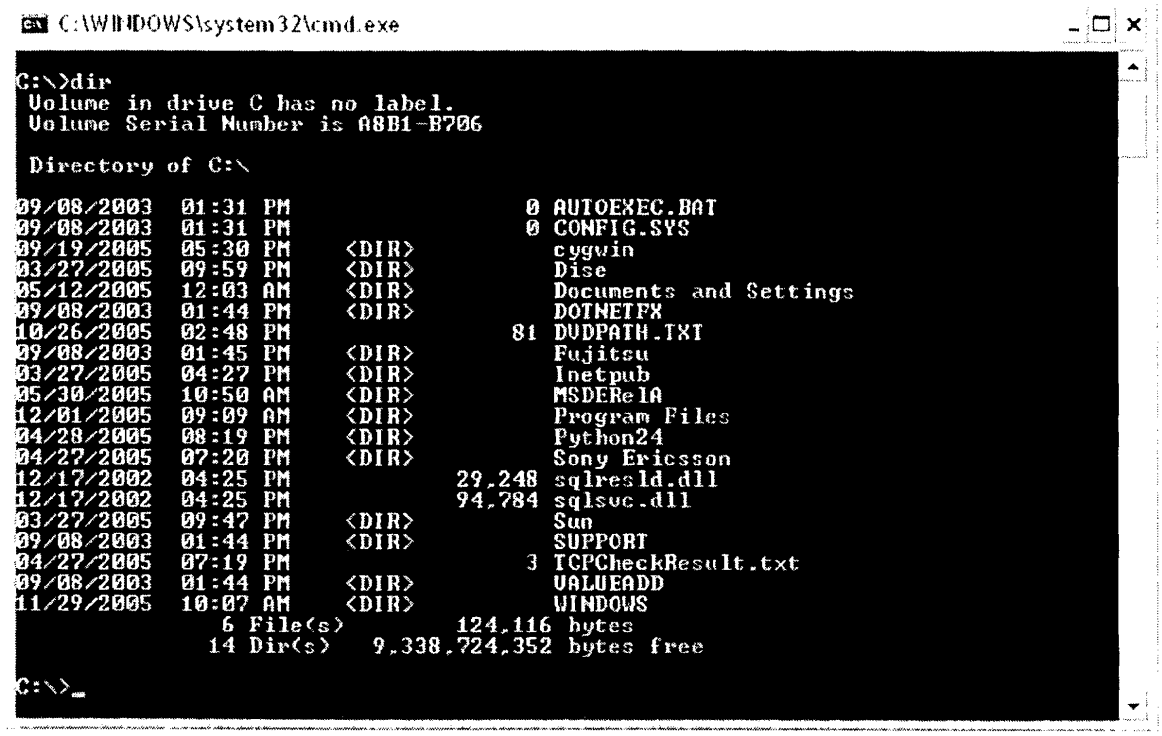

Figure 3.2 Example of Command Line Interface

3.3.1 Advantages and Limitations

Command line interface is powerful tool that offers direct access to the functionality of the system. The input command strings contain a number of operations, options, or parameters that can be applied to many objects at once in a very consistent way, which makes the command line interface very effective. In general, command line interface is more preferred by advanced users, who are able to express complex combinations of commands and perform tasks in a rapid matter, with a sense of being in control of the system. In addition, command line interface can also double as scripting programming language that can perform operations in a batch processing mode without user interaction. A specific operation can be analyzed and saved as a script, which can be executed later without further analysis and design effort. This characteristic makes a command user interface a great tool for performing repetitive tasks.

One limitation of command line interface is that it is difficult to learn. Command line interface has stiff learning curve and takes long to learn the even basic operations. Commands need to be remembered, as the interface doesn't provide which commands are available. Therefore, users have to remember complex command sequences that might be different from one system to another. This complicated command structure is the main reason for the low memorability of command line interface, which means that it is hard to remember the operation and option commands after not using the interface for a long period of time.

Another limitation is that command line interface does not check the command strings for syntax when they are entered, which causes frequent input error. Although most command line interfaces provide lists of available commands that users can look up, referring to the list while entering the input significantly lowers the efficiency of the interface. So, the typical approach for most users is to memorize the commands and enter them rapidly, in order to keep the high efficiency of the system. However, in this approach, inexperienced users encounter numerous input errors, and frequently, the error is the consequence of a single typing mistake in a long command. Also, due to the nature of input command strings being complex, it is hard to provide useful error messages and online assistance. Therefore, unlike graphical user interface, training of the user is usually necessary.

3.3.2 Design Guidelines

The following is a set of design guidelines for command line interface that can help developers setting up development plans and design strategies relevant to the interface being built [Stone, 2005]:

- Use specific and meaningful names for commands
- Use consistent syntax for all commands
- Make commands as short as possible to prevent typing errors
- If commands can be abbreviated, use common abbreviations following consistent rules

3.4 GUI vs. CLI

Table 3.1 compares graphical user interface and command line interface with regard to some of the characteristics of usability. The goal of this thesis is to develop a modified graphical user interface, or hybrid user interface, that increases the efficiency of the graphical user interface by incorporating the characteristics of the command line user interface.

Usability Characteristics	GUI	CLI
Learnability	High	Low
Efficiency	Low	High
Memorability	High	Low
Error Frequency and Recovery	High	Low

Table 3.1 Comparison of Graphical User Interface and Command Line Interface

Chapter 4

Hybrid User Interfaces

- Design Guidelines

4.1 Design Goal

Hybrid user interface is a new type of interface in computing that utilizes the design components of both graphical user interfaces and command line interfaces in an attempt to increase the usability of the underlying system. The main design goal of a hybrid user interface is to supplement a graphical user interface and improve its efficiency, especially when it is used to perform repetitive tasks. By bringing the elements of command line interfaces into typical graphical user interfaces, users are able to set the values for the graphical components – such as text boxes, check boxes, and combo boxes – by entering string-based commands in a text box or text field that is designated as the command prompt. To ensure syntactically correct user input, a hybrid user interface contains an area that displays available keywords and commands, graphical components to be completed if they are required, and examples of correct inputs. Also, the interface warns users of any invalid inputs. A well-designed hybrid user interface prevents users from the

extensive use of the computer mouse, which can be time-consuming and decrease the efficiency of the system when performing repetitive tasks.

4.2 Requirements

The first step of designing a hybrid user interface is to find out the requirements for the system being developed. Identifying the requirements is important because they are used in the later steps of the software development process as the guidelines for defining tasks and interactions between the graphical components and the string-based commands. In general, the following questions should be asked to identify the intended users, the objective of the system, and the required tasks and interactions:

- Who are the primary users?
- What do the users want to do with the system?
- What area of domain is the interface developed for?
- What are the tasks and interactions that the interface should carry out?

4.2.1 Who are the Users?

One of the essential aspects of designing a good user interface is to know the intended primary users and match the attributes of the interface design to the characteristics of the users. User characteristics that are particularly relevant to the design of hybrid user interfaces are computer and IT experience, educational background, motivation, attitude, and organizational culture. Since hybrid user interfaces combine the design elements of

both graphic and command line user interfaces, it is important that the users of the system have some level of experience using different types of user interfaces or take the necessary training. Also, users of a hybrid user interface should have strong motivations and attitudes toward accepting new technology for higher productivity and efficiency as well as willingness to learn new tools and applications. In order to find out the user characteristics, developers can ask the intended users direct questions if they know who the users are. If the intended users are unclear, developers need to interview knowledgeable people – such as domain experts, supervisors, project managers, and personnel managers – who know about the intended users and the domain for which the interface is developed.

4.2.2 What do the Users Want?

Once the intended users of the system are identified, the next step is to find out what they want to do or plan to do with the interface. In general, there are two types of user needs: felt needs and expressed needs. Felt needs are usually hidden or hard to identify because users may not know what the interface may be able to offer unless they have hands-on experience. So, most of the times, users do not realize they have specific needs or cannot clearly articulate what they want; it's often an "I will know it when I see it" situation. On the other hand, expressed needs are what users actually want or what they hope to accomplish using the interface. Experienced users might be able to identify features that they have needed for years, which could be an essential requirement for the new system being developed. Therefore, for the development of a hybrid user interface, knowing the expressed needs is very important for defining the tasks and interactions that the interface

must perform. It is also a starting point for establishing the functional requirements and specifications for the application.

4.2.3 Understanding the Domain

Domain is the area of expertise or specialization for which a user interface is developed. In user interface design, knowing the domain is as important as knowing the users and the users' needs because different domains generally have different requirements. For instance, a system administration application could be developed for both a financial firm and a university. However, since they are two completely different domains, the requirements and specifications for each application – even if they are same type of application – could be significantly dissimilar. For the development of a hybrid user interface, it is important to find out expert knowledge that is relevant to the system. Domain-specific information provides the specialized understanding that is essential to define the required tasks and interactions.

4.2.4 Defining the Tasks and Interactions

To design an efficient hybrid user interface, developers need to understand the functionality that the system must provide to meet the intended user's domain-specific needs. To find out specific tasks that the interface must handle, task analysis could be performed. Task analysis is the process of examining the way users perform their tasks. By understanding how users work and identifying the characteristics of their tasks,

effective interactions between the graphical components and the commands can be designed.

In addition, for the development of a hybrid user interface, the sequence of the user tasks could be also important as it decides the order of command strings that the users enter. For some hybrid user interfaces, the efficiency of the interactions among the design components might depend on the order of input commands. Another important point is the frequency of performing user tasks. For a hybrid user interface that is used frequently, efficiency of the system becomes the driving characteristic of the interface, and it should provide necessary tools such as highly customizable commands that can maximize the efficiency of the user interface. Such an interface may be more difficult to learn due to the complexity of commands, but frequent use of the interface can increase the learnability of the system.

4.2.5 Usability Requirements

As mentioned earlier in this chapter, the main goal of hybrid user interface is to increase the usability of the system. Therefore, it is important that a hybrid user interface meets the key usability requirements to create a usable user interface and satisfactory user experience. Table 4.1 shows the usability characteristics and the design solutions by hybrid user interfaces.

Usability Characteristics	Design Solutions
Learnability	Hybrid user interfaces incorporate just-in-time instruction systems that are continuously updated and displayed on a designated help area, making the interface easy to learn.
Efficiency	As a user types in command strings, associated graphical components get populated instantly, increasing the efficiency of the user interface.
Memorability	Available keywords and commands are displayed on a designated help area, which makes it unnecessary to memorize commands.
Error Frequency and Recovery	Command prompt is accompanied by a text field that works as a command feedback for alerting the user of any syntax error and data invalidity.

Table 4.1 Design Solutions of Hybrid User Interfaces for Usability Characteristics

4.3 Component Design

Once the process of gathering requirements is complete, the next step is to create a physical design of a hybrid user interface and its components. The overall design of the interface should follow the design approaches defined earlier in order to maximize the usability of the system.

4.3.1 Overall Design

In general, a hybrid user interface is composed of three subareas: GUI area, CLI area, and Help area. Different applications may present these areas in slightly different ways, but the typical layout of the interface looks like Figure 4.1.

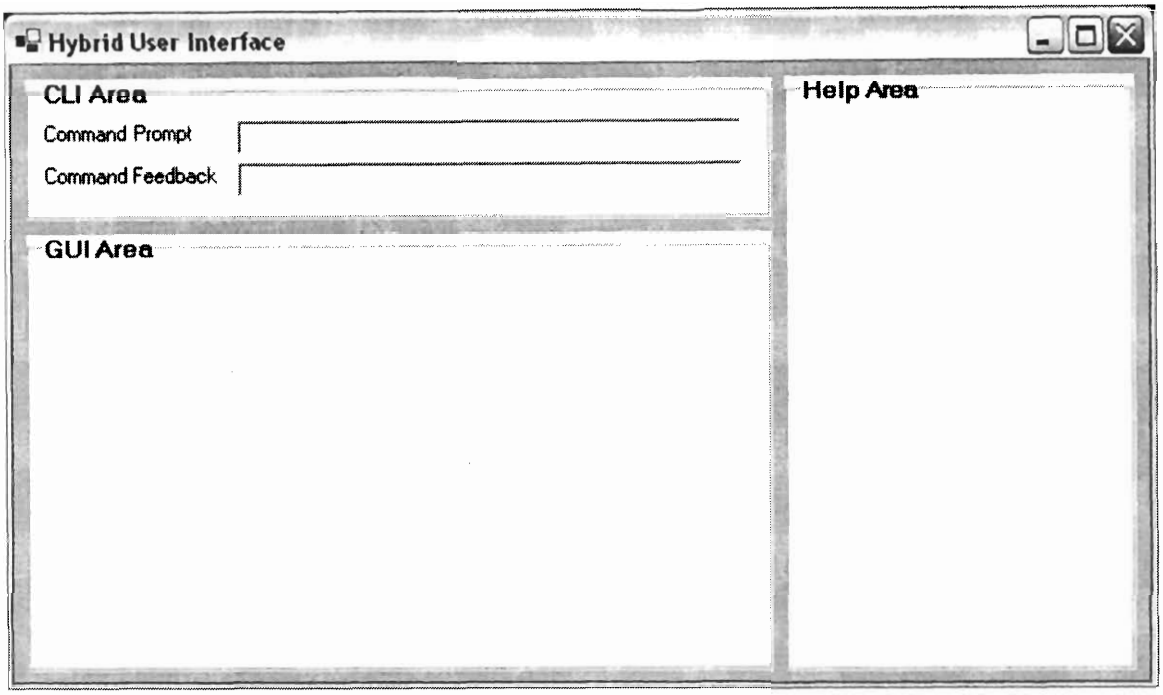

Figure 4.1 Typical Layout of a Hybrid User Interface

The overall hybrid user interface should be designed as simple as possible to keep the interface clear and natural to the users. Also, the design of each subarea must be consistent, making it useful to perform required tasks with high efficiency. Subareas and the interface as a whole should be clearly laid out allowing users to be able to distinguish them easily. The structure of the interface should reflect the expectations of the intended users and the domain.

4.3.2 Design of CLI Area

CLI (Command Line Interface) area contains a command prompt and a command feedback, implemented by a couple of text boxes. The role of a command prompt is to take the input command strings from the user and parse them into a series of tokens that

are used to populate the graphical elements in the GUI area. If there are any errors – for instance, invalid data type or incorrect keyword – in the user input, specific error message is displayed instantly in a command feedback area. Figure 4.2 shows a typical design of CLI area.

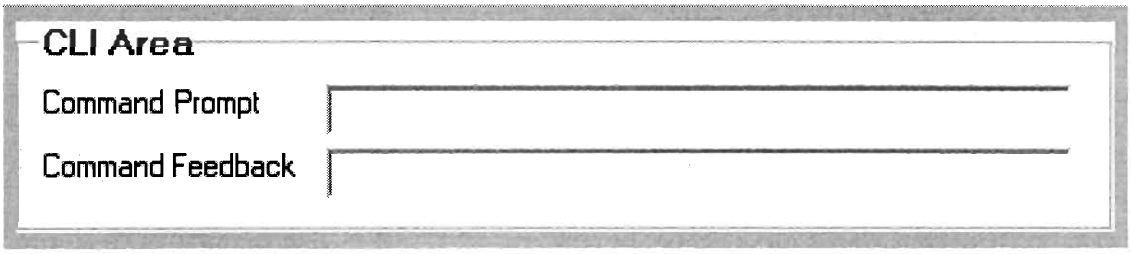

Figure 4.2 Design of CLI Area

The important aspect of designing an effective CLI area is that the process of parsing command strings and checking for errors must be done in real time. That is, as users type the command strings, a hybrid user interface should continuously check the input and keep the users informed of any mistakes.

4.3.3 Design of GUI Area

GUI (Graphical User Interface) area of a hybrid user interface is designed following the general development process of a graphical user interface and is composed of various graphical widgets such as button, text box, and combo box. Existing graphical user interfaces can be adopted as a GUI area if the hybrid user interface being developed is to supplement the present system. Unique design feature of a hybrid user interface is that all the graphical elements are linked to the equivalent command strings. That is, not only do

command strings populate the corresponding graphical elements, but any changes to the graphical elements also update the matching command strings, keeping the users up-to-date regarding their input.

4.3.4 Design of Help Area

Help area displays the just-in-time instruction for input commands, examples of valid input, and list of required fields that are not fulfilled. Just like the CLI area, help area updates its content in real time, allowing the users to check their progress on their input as they type the command strings.

4.4 Interaction Design

Based on the requirements and the physical layout of the graphical components, designing the interactions between different areas of the interface is the next step in the development of a hybrid user interface. These interactions must correctly and efficiently reflect the tasks that users want to perform.

4.4.1 Overall Interaction

In order to manage overall interactions among the subareas of a hybrid user interface, a controller that directs various interactions between components is required. Figure 4.3 is a high-level illustration of overall interactions of a hybrid user interface, with a controller in the middle of three subareas.

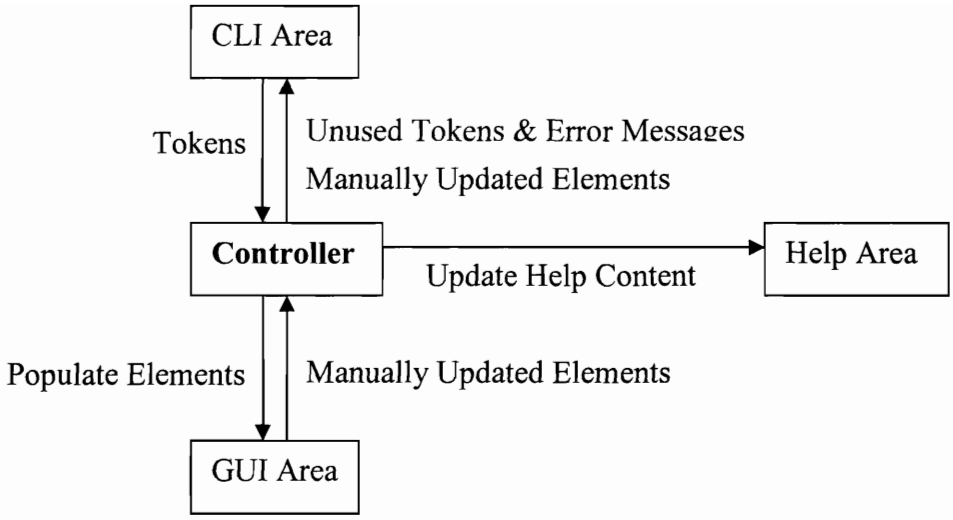

Figure 4.3 Overall Interactions of a Hybrid User Interface

The first job of a controller is to take the tokens from the CLI area and populate the associated graphical components in the GUI area. If any token is left unused, controller generates an error message and displays it in a command feedback of the CLI area. Also, when users update the value of a graphical element manually, controller modifies the command strings accordingly to keep the subareas in sync. As users enter the input commands, controller updates the content of the help area to assist users with the right instructions.

4.4.2 CLI Area Interaction

In a CLI area of a hybrid user interface, a command prompt is implemented to take the input commands and parse them into a series of tokens, which are used to set the values

of the corresponding graphical components in the GUI area. Once the parsing process is complete, the tokens are sent to the controller to populate the GUI area of the interface. If any token is left unused by the controller due to no matching graphical components, an exception is thrown and a text-based error message is displayed in a command feedback alerting the user.

A) Parsing Commands

A command prompt parses the command strings continuously into a series of tokens. Developers can set the frequency of parsing, depending on the nature of the application. Some of the implementation examples introduced in the next chapter parses the command whenever a user enters a white space or when the input is idle for one second. During the process of parsing commands, each token is assigned a data type that is predefined by the developer of the system. These tokens must be generic without any prior knowledge about the GUI area, and the only information that a token holds are the string value it represents and the data type of the string. Generally, developers can choose any type of data for the system, including the following examples:

- String
- Limited string defined by the developer
- Floating point number without sign
- Floating point number with sign
- Integer without sign
- Integer with sign

- Specially formatted numbers such as date or currency
- Or any combination of string and number defined by the developer

If a command string is not one of the data types – special symbol '$', for instance – predefined by the developer, it is considered an input error and an exception is thrown. Once the parsing is complete, all the valid tokens are stored in an instance of a data structure and sent to the controller, which consumes the tokens and populates the associated elements in the GUI area. Figure 4.4 is an example of a system that processes a grade for a student. It parses command strings into tokens using a white space as the delimiter for the input. In this example, user input only holds a name of a student, a numeric grade, and a corresponding letter grade. Therefore, developers need to define three data types for the system: string, limited string, and floating point number without sign. If a user accidentally enters a negative number as a numeric grade, the hybrid user interface catches the error and notifies the user.

B) Unconsumed Tokens

It is possible that some of the valid tokens can be left unconsumed by the controller of a hybrid user interface, even if they contain valid data. For instance, if a user enters "John Doe 91.5 92.3 A" in the above example, there is no error parsing the command strings into tokens because all the strings are of valid type. However, when the controller sets the values of the graphical elements, the second number, "92.3", does not get consumed, assuming the first number, "91.5", is used before the second number. If any token is unconsumed, the controller returns the unused token, which is wrapped up in an

exception object, back to the CLI area and an error message is displayed in the command feedback to inform the user.

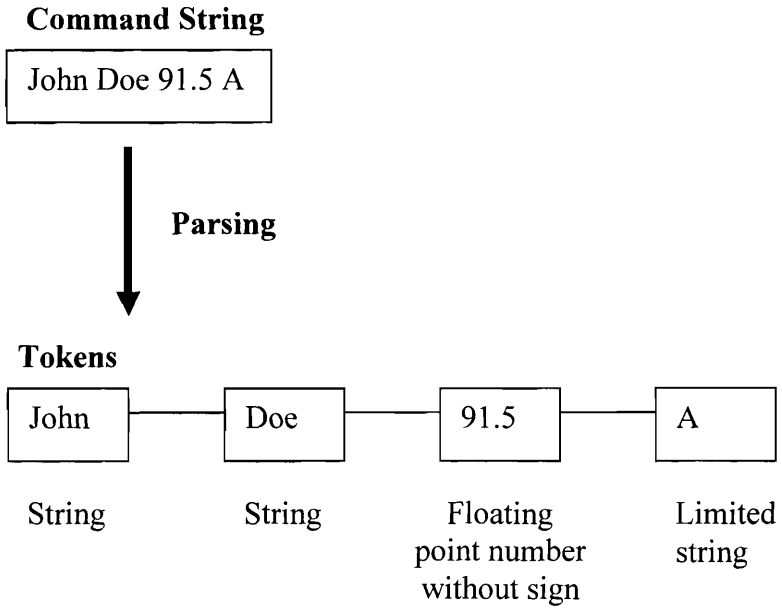

Figure 4.4 Example of Parsing Commands into Tokens

4.4.3 GUI Area Interaction

Graphical elements in the GUI area are populated by using the tokens passed to the controller. First, developers should decide the order of elements to be populated since many tokens could carry exact same data type. In the previous example of processing grade for a student, the token representing 'first name' carries the same data type, string, as the token for 'last name'. So, without the proper ordering of the elements, when the GUI area tries to consume the first token "John", it does not know whether it is the first name of the student or the last name. Therefore, the developer of the system should

enforce a rule that 'first name' must be consumed before 'last name'. If the system has to accept many tokens with the same data type, developers can use keywords to distinguish tokens. For instance, instead of "John Doe", system can be implemented to accept "fname:John lname:Doe" by using a colon as a delimiter and "fname" and "lname" as keywords. If keywords are used to differentiate tokens, additional features such as command completion can be implemented in the CLI area to increase the speed of user input. For example, a string "fn" can be automatically converted into a string "fname", assuming that no other keywords start with the characters 'f' and 'n'. Moreover, if "fname" is the only keyword beginning with the character 'f', typing 'f' can complete the string "fname". However, it is important to remember that using too many keywords and delimiters can significantly lower the efficiency of a hybrid user interface.

A) Set Values from Commands

Tokens are passed to the controller stored in an instance of a data structure. Any type of data structure can be used as long as the tokens can be extracted in a consistent order. In most cases, tokens are pulled out in an order they were stored. Figures 4.5 through 4.10 show the process of populating graphical elements using a series of tokens generated from the previous examples of the grade processing system. In Figure 4.8, the pointer skips to the next token, since the data type of the token and the graphical elements does not match. The next cycle looks for any remaining tokens and tries to consume it if there is an element that is not populated. If all the elements are populated and there is any unused token, an exception is thrown and an error message is returned to the command feedback in the CLI area.

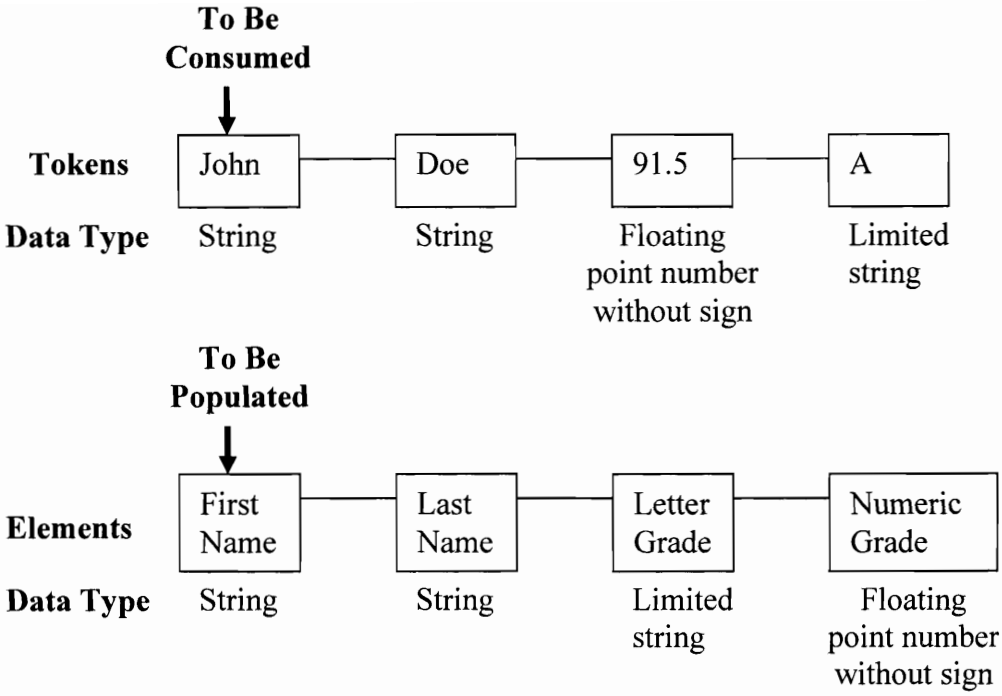

Figure 4.5 Populating Graphical Elements: First Cycle

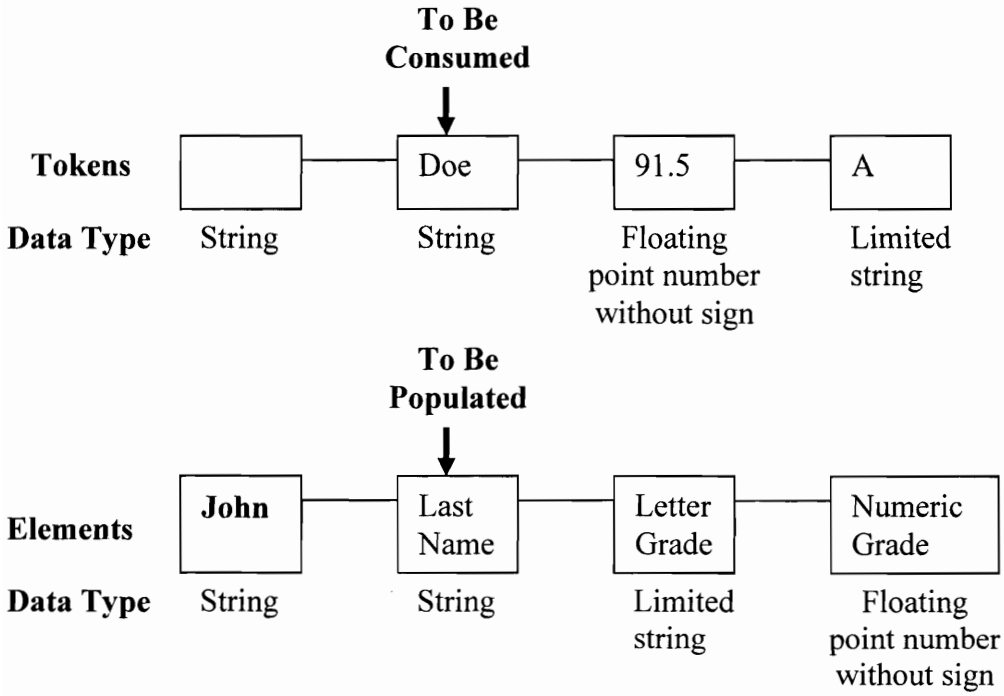

Figure 4.6 Populating Graphical Elements: Second Cycle

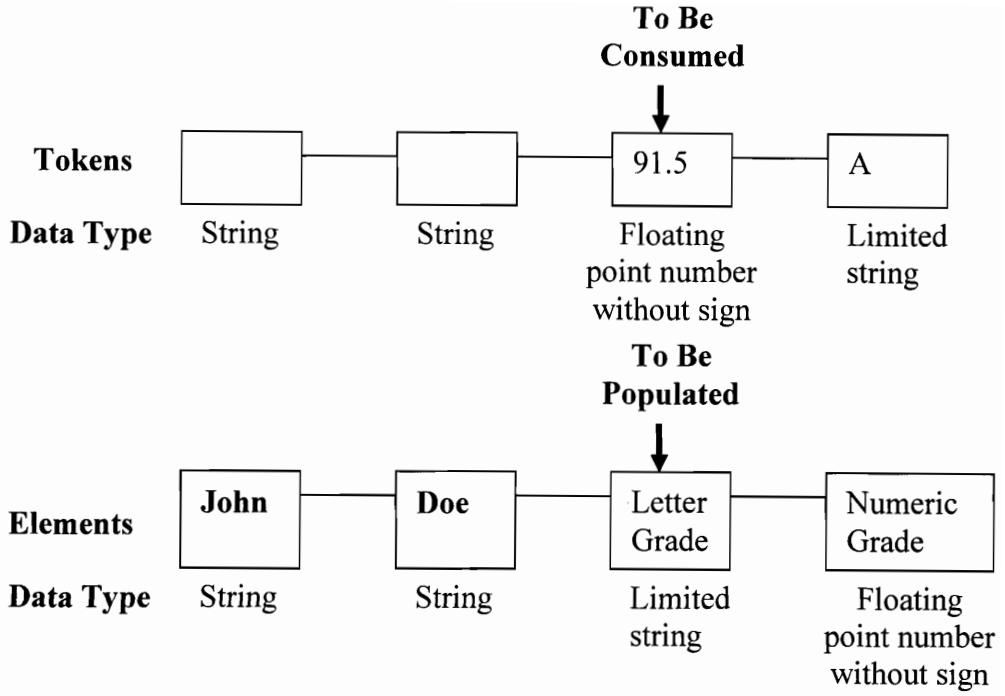

Figure 4.7 Populating Graphical Elements: Third Cycle

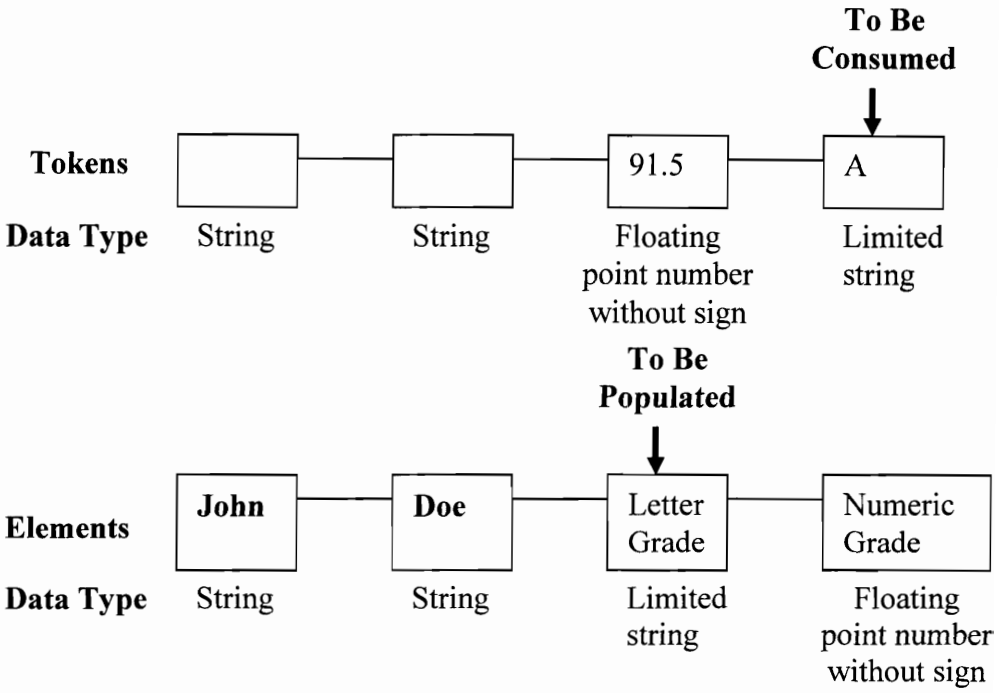

Figure 4.8 Populating Graphical Elements: Fourth Cycle

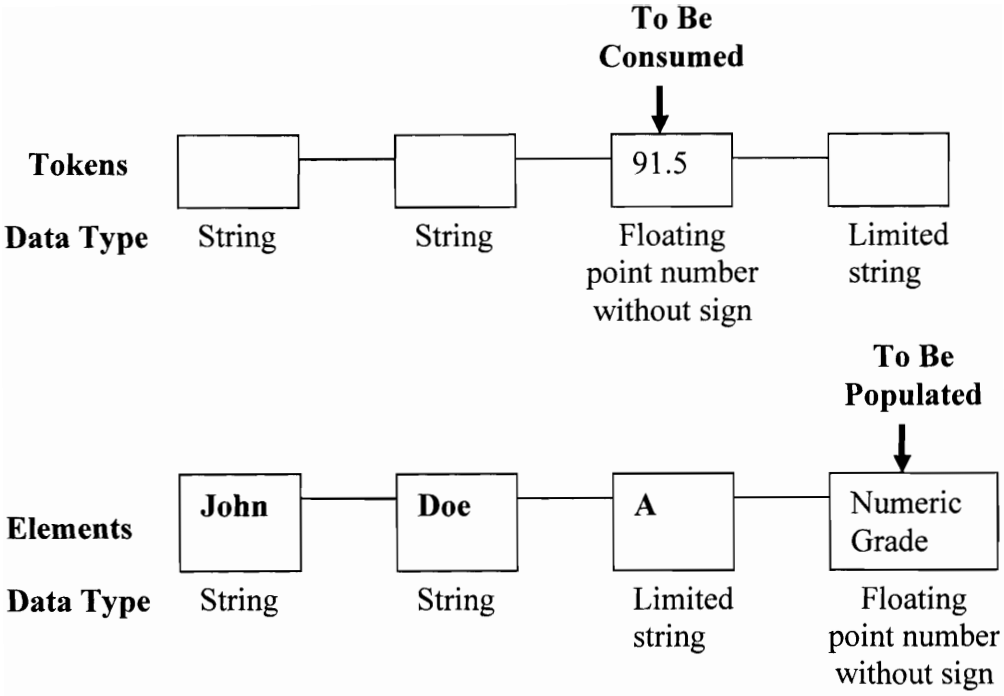

Figure 4.9 Populating Graphical Elements: Fifth Cycle

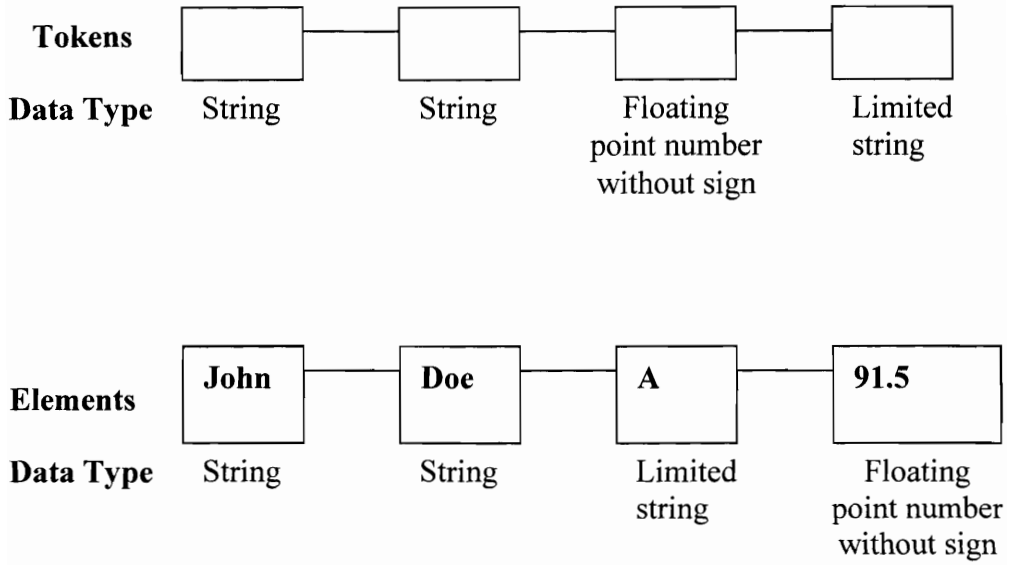

Figure 4.10 Populating Graphical Elements: Complete

B) Manually Update Values

In addition to the process of populating elements using tokens, users can manually update the values for the graphical elements, just like using a regular graphical user interface. When a value gets changed by the user, the GUI area fires an event, which is passed back to the CLI area to update the corresponding command string. For this purpose, a copy of the original tokens must be made before the tokens are sent to the controller. Generally, tokens can be stored in a hash table with unique ID for each token. The event from GUI area finds the right command string in a hash table using the associated ID and updates the value.

4.4.4 Help Area Interaction

Based on the graphical elements defined by the developer, a hybrid user interface can generate a just-in-time instruction for input commands. The system checks the elements in the order they are supposed to be populated and displays information for the next element that is on the queue. For instance, in Figure 4.6, 'Last Name' is the next field that should be populated and the system generates and shows the relevant information, such as the next field to be filled, correct data type, and example of input. As users complete the elements one by one, the content of the help area gets updates continuously, assisting the user make the correct input.

Chapter 5

Hybrid User Interfaces

- Implementation Examples

5.1 Overview

This chapter presents implementation examples for some of the design guidelines introduced in the previous chapter, focusing mainly on the processes of parsing commands and populating graphical components. All the codes are developed in C#.NET programming language, and for the simplification of the codes, the `using` statements are omitted. The `using` statement imports classes that are required to run the application and is equivalent to the keyword `import` in Java programming language.

5.2 Parsing Commands

In general, a command string is composed of substrings that are separated by delimiters; for instance, a user command, "John Doe 91.5 A", contains 4 different substrings, "John", "Doe", "91.5", and "A", separated by white spaces. In a hybrid user interface, each of

these strings is represented by an instance of `CommandPart` class, and the whole command string, which holds a reference to the collection of `CommandPart` objects, is an object of `Command` class. Each `CommandPart` object is a token that is passed to the controller of a hybrid user interface in order to fill the corresponding graphical element.

Example 5.1 `Command` Class

```
public class Command
{

      private ArrayList tokens;

      // Property for tokens
      public ArrayList Tokens
      {
            get{return tokens;}
      }

      // Constructor
      public Command(string commandText)
      {
            tokens = ParseCommandIntoTokens(commandText);
      }

      // Method for parsing command into tokens
      public ArrayList ParseCommandIntoTokens(string command)
      {
            ArrayList commandParts = new ArrayList();
            string tempCommand = command.TrimStart();
            while(!tempCommand.TrimEnd().Equals(""))
            {
                  CommandPart eachPart
                        = GetNextToken(ref tempCommand);
                  commandParts.Add(eachPart);
            }
            return commandParts;
      }

      // Rest of the code omitted
      // ......
      // ......

}
```

Example 5.2 is the `GetNextToken()` method of `Command` class that uses a text-manipulating technique called Regular Expression in order to examine a command string and extract a token from it. A token is taken out of a command string based on the command type of each string. If there is no predefined command type matching a given token, the system throws an exception for an input error, which is displayed in the CLI area.

Example 5.2 `GetNextToken()` Method of `Command` Class

```
public CommandPart GetNextToken(ref string command)
{

      // Get the next token separated by " "

      // Case 1. String
      Regex regex = new Regex(@"^[A-Za-z]+(\s)?");
      Match match = regex.Match(command);
      if (match.Length != 0)
      {
            CommandPart part
                  = new CommandPart(match.ToString().Trim(),
                        CommandPart.CommandType.String);
            Command
                  = command.Substring(
                        match.ToString().Length).TrimStart();
            return part;
      }

      // Case 2. Date in the format of MM/DD/YYYY
      regex = new Regex(@"^(\d){2}/(\d){2}/(\d){4}+(\s)?");
      match = regex.Match(command);
      if (match.Length != 0)
      {
            CommandPart part
                  = new CommandPart(match.ToString().Trim(),
                        CommandPart.CommandType.Date);
            Command
                  = command.Substring(
                        match.ToString().Length).TrimStart();
```

```
            return part;
      }

      // Case 3. Integer with sign
      regex = new Regex(@"^(\+|\-)(\d)+(\s)?");
      match = regex.Match(command);
      if (match.Length != 0)
      {
            CommandPart part
                  = new CommandPart(match.ToString().Trim(),
                        CommandPart.CommandType.IntegerWithSign);
            Command
                  = command.Substring(
                        match.ToString().Length).TrimStart();
            return part;
      }

      // Many more cases possible based on the application
      // Rest of the code omitted
      // ……
      // ……

      // In case of no matching
      throw new System.Exception(
            "Do Not Understand Command: " + command);

}
```

An instance of `CommandPart` class holds a string value and its command type that is predefined by the developer based on the characteristics of the application being developed. In Example 5.3, all the available command types are defined in the `CommandPart` class, as a set of named constants called 'enumerator list' in an enumeration, `CommandType`.

Example 5.3 `CommandPart` Class

```
public class CommandPart
{

      // Enumeration for command type
```

```
        public enum CommandType
        {
                String,
                LimitedString,
                IntegerWithSign,
                Date

                // More type available
                // ……
                // ……
        }

        private string value;
        private CommandType type;

        // Property for value
        public string Value
        {
                get{return value;}
        }

        // Property for type
        public CommandType Type
        {
                get{return type;}
        }

        // Constructor
        public CommandPart(string value, CommandType type)
        {
                this.value = value;
                this.type = type;
        }

        // Rest of the code omitted
        // ……
        // ……

}
```

Once the command string is successfully broken down into a series of `CommandPart` objects, they are stored in a data structure – an instance of `ArrayList`, for example – to which the `Command` object holds a reference. Then, the controller of a hybrid user interface passes the Command object to the `Populate()` method of the underlying

object. In a hybrid user interface, a GUI area contains at least one underlying object that represents the whole GUI area and has references to objects that are directly connected to graphical elements one by one. For the grade processing system introduced earlier, the underlying object can be an instance of `StudentGrade` class, or any class that holds references to the instances of `FirstName`, `LastName`, `LetterGrade`, and `NumericGrade`. In this example, there are four graphical elements directly linked to these four instances, and the underlying object controls them.

Example 5.4 `Parse()` Method of `Controller` Class

```
public void Parse(Command command)
{

      // Rest of the code omitted
      // ......
      // ......

      underlyingObject.Populate(command);

      // Rest of the code omitted
      // ......
      // ......

}
```

In the current example, input command strings get converted into tokens when a user enters a white space. In order to control the frequency of parsing commands, each key stroke must be carefully checked. The important issue regarding the frequency of parsing is that only completed strings should be parsed in order to maintain the efficiency of the system. For instance, when a user tries to input "John", the system must not parse an incomplete string such as "J", "Jo", or "Joh". It should wait until the user completes a

string and enter a white space before parsing the command. The following example is the `KeyPressed()` method of `CommandAndStatus` class that controls the frequency of parsing tokens.

Example 5.5 `KeyPressed()` Method of `CommandAndStatus` Class

```
public class CommandAndStatus : System.Windows.Forms.UserControl
{

      private char lastKey;

      // Rest of the code omitted
      // ......
      // ......

      private void KeyPressed(object sender,
            System.Windows.Forms.KeyPressEventArgs e)
      {
            lastKey = e.KeyChar;

            // 13 is the numerical value of a white space
            if (e.KeyChar == 13)
            {
                  ParseCommand();

                  // Rest of the code omitted
                  // ......
                  // ......
            }
      }

}
```

5.3 Populating Elements

The first step of populating graphical elements in a hybrid user interface is to create a list of available elements for the command-based interaction. Example 5.6 is based on the previous example of processing student grade that contains four elements needed to be

populated from command strings: `FirstName`, `LastName`, `LetterGrade`, and `NumericGrade`. An object type `ParsingElement` represents any graphical elements that can be populated directly from a command prompt. `ParsingElement` is a public interface that contains a public method `ParseTokens()`, and it is implemented by all the available graphical elements.

Example 5.6 `GenerateEmptyParsingElements()` Method

```
public ParsingElement[] GenerateEmptyParsingElements()
{

      ParsingElement[] emptyElements = new ParsingElement[4];
      emptyElements[0] = new FirstName();
      emptyElements[1] = new LastName();
      emptyElements[2] = new LetterGrade();
      emptyElements[3] = new NumericGrade();

      return emptyElements;

}
```

The order of elements in the above example decides the order of processing tokens. Since there can be many elements carrying exact same data type, it is important to define the order of graphical elements that consumes tokens. In general, the element with the lowest index gets processed first and the index increases by one to move to the next element. In this example, the first token that carries a string is assigned to the element for the first name of a student, as it comes before the element for the last name. The order of processing elements must be displayed in the help area to make sure that users understand the correct input sequence.

Example 5.7 `Populate()` Method

```
public void Populate(Command command)
{

      // Step 1. Generates an empty array of parsing elements
      ParsingElement[] elements = GenerateEmptyParsingElements();

      // Step 2. Fills the elements using the command
      FillElementsFromCommand(elements, command);

      // Rest of the Code Omitted
      // ……
      // ……

}
```

After an empty array of type `ParsingElement` is generated based on the order of available elements, `FillElementFromCommand()` method carries out the populating process described in Figure 4.5 through Figure 4.10, using the completed command that contains tokens.

Example 5.8 `FillElementFromCommand()` Method

```
public static void FillElementsFromCommand(
      ParsingElement[] elementList, Command command)
{

      if (command.IsValid)
      {
            foreach(ParsingElement eachElement in elementList)
            {
                  eachElement.ParseTokens(command);
            }
      }

}
```

Each `ParsingElement` object has a method called `ParseTokens()` that takes the `Command` object and uses the first available token with the matching data type to set the value for the graphical element. A consumed token is removed from the `Command` object. If any token is left unused after the complete cycle of populating graphical elements, an exception is thrown and an error message is displayed in a CLI area to inform the user of invalid input.

Example 5.9 `ParseTokens()` Method of `FirstName` Class

```
public void ParseTokens(Command command)
{

      ArrayList tokens = command.Tokens;

      foreach (object token in tokens)
      {
            CommandPart eachCommandPart = token as CommandPart;
            if (eachCommandPart.Type ==
                  CommandPart.CommandType.String)
            {
                  firstName = eachCommandPart.Value;
                  isValid = true;
                  tokens.Remove(token);
                  return;
            }
      }

}
```

5.4 Managing Interactions

Interactions between a CLI area and a GUI area, such as parsing command strings and populating graphical elements, are managed by an instance of class called `Controller`. An instance of `Controller` holds references to the objects that represent CLI area,

GUI area, help area, and underlying object. The most important method of `Controller` class is the `Parse()` method that populates the underlying object using a `Command` object and its tokens. The following example shows the high-level outline of the `Controller` class.

Example 5.10 `Controller` Class

```
public class Controller
{
      private CommandAndStatus CLIArea;
      private GUIControl GUIArea;
      private RichTextBox hintArea;

      private StudentGrade underlyingObject;

      public Controller(CommandAndStatus CLIArea,
            GUIControl GUIArea, RichTextBox hintArea)
      {
            this.CLIArea = CLIArea;
            this.GUIArea = GUIArea;
            this.hintArea = hintArea;

            // Set the "Controller" on CLI area and GUI area
            CLIArea.Controller = this;
            GUIArea.Controller = this;

            this.GUIArea.HintArea = hintArea;

            // Rest of the Code Omitted
            // ......
            // ......
      }

      public bool IsAllRequiredElementsFilled()
      {
            return GUIArea.IsAllRequiredElementsFilled();
      }

      public void Parse(Command command)
      {
            if (command != null)
            {
                  GUIArea.Command = command;
```

```
            }

            underlyingObject.Populate(command);
            GUIArea.Object = underlyingObject;

            if (command.Tokens.Count > 0)
            {
                  throw new Exception("Invalid Command: " +
                  command_.CommandPartsToBeConsumed[0].ToString());
            }
      }

      // Rest of the Code Omitted
      // ......
      // ......

}
```

Under the management of controller, subareas of a hybrid user interface interact with each other passing commands, tokens, events, and messages. Controller also makes sure that the CLI area, GUI area, and help area are all in sync assisting users performing their tasks efficiently.

Chapter 6

Conclusion

6.1 Summary

This thesis has presented a hybrid user interface that combines the features of a graphical user interface and a command line interface in order to increase the usability of the system. By adopting the string-based input mechanism of command line interfaces, users of a hybrid user interface are able to set the values for graphical components using only the keyboard, eliminating the inefficiency of the computer mouse for repetitive tasks. In particular, for applications that require repetitive tasks such as entering multiple data and managing system administration, a hybrid user interface enhances the efficiency of the system significantly. A hybrid user interface can be developed as a new application or can supplement an existing graphical user interface when the efficiency of the system is of major concern.

6.2 Future Work

In order to quantify the efficiency of hybrid user interfaces and the improvement over typical graphical user interfaces, end-user testing must be conducted. For a successful

end-user testing, participants with different backgrounds and expertise should perform four to five different tasks including:

- First impression task
- Highly repetitive task
- Highly frequent task
- Help issue task

The current design of hybrid user interface mainly focuses on populating graphical elements of the system from command strings that users input. The future goal of the research is to implement a command prompt that extends the functionality to other areas of a typical graphical user interface, such as menus, operations, and options. That is to develop a user interface in which a user can perform most of the graphical user interface interactions without using the computer mouse. To achieve this goal, the algorithms for handling different types of tokens must be able to control complex parsing and populating processes. The ultimate goal of hybrid user interface is to develop a graphic-based user interface that is faster and more efficient than a command line interface.

Appendix

- Code Examples

Command.cs

```
public class Command
{
	public event System.EventHandler OnCommandTextUpdate;
	public bool IsValid = false;

	public Command(string commandText_)
	{
		this.m_orginalCommandParts =
			ParseCommandIntoCommandParts(commandText_);

		if (this.m_orginalCommandParts != null &&
			this.m_orginalCommandParts.Count > 0)
		{
			this.IsValid = true;
		}

		this.m_commandPartsToBeConsumed =
			this.m_orginalCommandParts.Clone() as ArrayList;
		this.m_hashOfTradeEleTypeToCommandPart = new Hashtable();
	}

	private ArrayList m_orginalCommandParts;

	private ArrayList m_commandPartsToBeConsumed;
	public ArrayList CommandPartsToBeConsumed
	{
		get
		{
			return this.m_commandPartsToBeConsumed;
		}
	}

	private string m_commandText;
	public string CommandText
	{
		get
```

```
        {
            return this.m_commandText;
        }
        set
        {
            this.m_commandText = value;
            if (OnCommandTextUpdate != null)
            {
                OnCommandTextUpdate(this, null);
            }
        }
}

private Hashtable m_hashOfTradeEleTypeToCommandPart;
public Hashtable TradeEleTypeToCommandPartHash
{
        get
        {
            return this.m_hashOfTradeEleTypeToCommandPart;}
        }
}

public void UpdateCommand(ITradeElement tradeEle_)
{
        if (this.m_hashOfTradeEleTypeToCommandPart != null)
        {
            string key = tradeEle_.UniqueKey;
            if (this.TradeEleTypeToCommandPartHash.Contains(key))
            {
                CommandPart[] commandParts =
                    this.TradeEleTypeToCommandPartHash[key]
                        as CommandPart[];

                if (tradeEle_.ElementType ==
                    TradeElementType.MaturityDate)
                {
                    MaturityDate date = tradeEle_
                        as MaturityDate;
                    tradeEle_ = date.Date;
                }
                else if (tradeEle_.ElementType ==
                    TradeElementType.EffectiveDate)
                {
                    EffectiveDate date = tradeEle_
                        as EffectiveDate;
                    tradeEle_ = date.Date;
                }

                IParsingElement parsingEle = tradeEle_
                    as IParsingElement;
                if (parsingEle != null)
                {
                    for (int i = 0; i < commandParts.Length;
                        i++)
                    {
                        CommandPart eachPart =
                            commandParts[i];
```

```
                                    eachPart.UpdateByTradeElement(
                                          parsingEle, i);
                              }
                        }

                        this.CommandText = ReGenerateCommandText();
                  }
            }
      }

private string ReGenerateCommandText()
{
      string command = "";
      foreach (object token in this.m_orginalCommandParts)
      {
            CommandPart eachCommandPart = token as CommandPart;
            string commandPartString = eachCommandPart.Value;
            if (commandPartString != "")
            {
                  command += commandPartString + "  ";
            }
      }
      return command;
}

protected ArrayList ParseCommandIntoCommandParts(string command_)
{
      ArrayList commandParts = new ArrayList();

      string command = command_.TrimStart();
      while(! command.TrimEnd().Equals(""))
      {
            CommandPart eachPart = GetNextField(ref command);
            commandParts.Add(eachPart);
      }
      return commandParts;
}

protected CommandPart GetNextField(ref string command_)
{
      // Check for an IMM date
      Regex regex = new Regex(@"^[A-Za-z]{3}(\d){2}(\s)?");
      Match match = regex.Match(command_);
      if (match.Length != 0)
      {
            CommandPart part =
                  new CommandPart(match.ToString().Trim(),
                        CommandPart.CommandType.IMMDate);
            command_ =
                  command_.Substring(
                        match.ToString().Length).TrimStart();
            return part;
      }

      // Check if the first part is string
      regex = new Regex(@"^[A-Za-z]+(\s)?");
      match = regex.Match(command_);
```

```
if (match.Length != 0)
{
      CommandPart part =
            new CommandPart(match.ToString().Trim(),
                  CommandPart.CommandType.STRING);
      command_ =
            command_.Substring(
                  match.ToString().Length).TrimStart();
      return part;
}

// Check for date offset such as 10Y, 10y, 6M, or 6m
regex = new Regex(@"^(\d)+(Y|y|M|m)(\s)?");
match = regex.Match(command_);
if (match.Length != 0)
{
      CommandPart part =
            new CommandPart(match.ToString().Trim(),
                  CommandPart.CommandType.DateOffset);
      command_ =
            command_.Substring(
                  match.ToString().Length).TrimStart();
      return part;
}

// Check for date such as 3/15/05 or 3/15/2005
regex = new
      Regex(@"^(\d){1,2}/(\d){1,2}(/(\d){2,4})?(\s)?");
match = regex.Match(command_);
if (match.Length != 0)
{
      CommandPart part =
            new CommandPart(match.ToString().Trim(),
                  CommandPart.CommandType.DateString);
      command_ =
            command_.Substring(
                  match.ToString().Length).TrimStart();
      return part;
}

regex = new Regex(@"^(\d){4}/(\d){1,2}/(\d){1,2}(\s)?");
match = regex.Match(command_);
if (match.Length != 0)
{
      CommandPart part =
            new CommandPart(match.ToString().Trim(),
                  CommandPart.CommandType.DateString);
      command_ =
            command_.Substring(
                  match.ToString().Length).TrimStart();
      return part;
}

// Check for decimal with sign
regex = new Regex(@"^(\+|\-)(\d)+\.(\d)*(\s)?");
match = regex.Match(command_);
if (match.Length != 0)
```

```
{
      CommandPart part =
            new CommandPart(match.ToString().Trim(),
                  CommandPart.CommandType.NumberWithSign);
      command_ =
            command_.Substring(
                  match.ToString().Length).TrimStart();
      return part;
}

// Check for int with sign
regex = new Regex(@"^(\+|\-)(\d)+(\s)?");
match = regex.Match(command_);
if (match.Length != 0)
{
      CommandPart part =
            new CommandPart(match.ToString().Trim(),
                  CommandPart.CommandType.NumberWithSign);
      command_ =
            command_.Substring(
                  match.ToString().Length).TrimStart();
      return part;
}

// Check for float with decmial point without sign
regex = new Regex(@"^(\d)+\.(\d)*(\s)?");
match = regex.Match(command_);
if (match.Length != 0)
{
      CommandPart part =
            new CommandPart(match.ToString().Trim(),
                CommandPart.CommandType.NumberWithoutSign);
      command_ =
            command_.Substring(
                  match.ToString().Length).TrimStart();
      return part;
}

// Check for int without sign
regex = new Regex(@"^(\d)+(\s)?");
match = regex.Match(command_);
if (match.Length != 0)
{
      CommandPart part =
            new CommandPart(match.ToString().Trim(),
                CommandPart.CommandType.NumberWithoutSign);
      command_ =
            command_.Substring(
                  match.ToString().Length).TrimStart();
      return part;
}

// Check for a sign
regex = new Regex(@"^(\+|\-)?(\s)?");
match = regex.Match(command_);
if (match.Length != 0)
{
```

```
                CommandPart part =
                        new CommandPart(match.ToString().Trim(),
                                CommandPart.CommandType.SIGN);
                command_ =
                        command_.Substring(
                                match.ToString().Length).TrimStart();
                return part;
            }

            throw new System.Exception(
                "Do not understand the command starting from:"
                    + command_);
        }
}
```

CommandPart.cs

```
public class CommandPart
{
    public enum CommandType
    {
        STRING,
        NumberPlusString,
        NumberWithSign,
        NumberWithoutSign,
        SIGN,
        DateOffset,
        DateString,
        IMMDate
    }

    public string Value;
    public CommandType Type;

    public CommandPart(string value_, CommandType type_)
    {
        Value = value_;
        Type = type_;
    }

    public override string ToString()
    {
        return Value.ToString();
    }

    public void UpdateByTradeElement(
        IParsingElement parsingEle_, int order_)
    {
        switch (parsingEle_.ElementType)
        {
```

```
case TradeElementType.BuySell:
{
      BuySell ele = parsingEle_ as BuySell;
      if (this.Type == CommandType.STRING)
      {
            this.Value = ele.ToString();
      }
      else if (this.Type == CommandType.SIGN)
      {
            if (ele.BuyOrSell ==
                  BuySell.BuySellType.Sell)
            {
                  this.Value = "-";
            }
            else
            {
                  this.Value = "+";
            }
      }
      else if (this.Type ==
            CommandType.NumberWithoutSign)
      {
            if (ele.BuyOrSell ==
                  BuySell.BuySellType.Sell)
            {
                  this.Value = "-" + this.Value;
                  this.Type =
                        CommandType.NumberWithSign;
            }
      }
      else if (this.Type ==
            CommandType.NumberWithSign)
      {
            if (ele.BuyOrSell ==
                  BuySell.BuySellType.Sell)
            {
                  this.Value = "-"
                        + this.Value.Substring(1);
            }
            else
            {
                  this.Value = "+"
                        + this.Value.Substring(1);
            }
      }
      break;
}

case TradeElementType.PayRec:
{
      PayRec ele = parsingEle_ as PayRec;
      if (this.Type == CommandType.STRING)
      {
            this.Value = ele.ToString();
      }
      else if (this.Type == CommandType.SIGN)
      {
```

```
            if (ele.PayOrRec ==
                  PayRec.PayRecType.Rec)
            {
                  this.Value = "-";
            }
      }
      else if (this.Type ==
            CommandType.NumberWithoutSign)
      {
            if (ele.PayOrRec ==
                  PayRec.PayRecType.Rec)
            {
                  this.Value = "-" + this.Value;
                  this.Type =
                        CommandType.NumberWithSign;
            }
      }
      else if (this.Type ==
            CommandType.NumberWithSign)
      {
            if (ele.PayOrRec ==
                  PayRec.PayRecType.Rec)
            {
                  this.Value = "-"
                        + this.Value.Substring(1);
            }
            else
            {
                  this.Value =
                        this.Value.Substring(1);
            }
      }
      break;
}

case TradeElementType.Notional:
{
      Notional ele = parsingEle_ as Notional;
      if (this.Type ==
            CommandType.NumberWithoutSign)
      {
            float numberinMillion = ele.Amount
                  / 1000000;
            this.Value = numberinMillion.ToString();
      }
      else if (this.Type ==
            CommandType.NumberWithSign)
      {
            float numberinMillion = ele.Amount
                  / 1000000;
            string sign = this.Value.Substring(0, 1);
            this.Value = sign
                  + numberinMillion.ToString();
      }
      break;
}
```

```
case TradeElementType.Rate:
{
      Rate ele = parsingEle_ as Rate;
      if (this.Type ==
            CommandType.NumberWithoutSign)
      {
            this.Value = ele.ToString();
      }
      break;
}

case TradeElementType.Spread:
{
      Spread ele = parsingEle_ as Spread;
      if (this.Type ==
            CommandType.NumberWithoutSign)
      {
            this.Value = ele.ToString();
      }
      break;
}

case TradeElementType.PV01:
{
      PV01 ele = parsingEle_ as PV01;
      if (this.Type == CommandType.NumberWithoutSign
            || this.Type ==
                  CommandType.NumberWithSign)
      {
            this.Value = ele.ToString();
      }
      break;
}

case TradeElementType.InstrumentDate:
{
      InstrumentDate ele = parsingEle_
            as InstrumentDate;
      this.Value = ele.ToString();
      if (ele.Type == InstrumentDate.DateType.Date)
      {
            this.Type = CommandType.DateString;
      }
      else
      {
            this.Type = CommandType.DateOffset;
      }
      break;
}

case TradeElementType.DateCountBasisAndPayFreq:
{
      DateCountBasisAndPayFreq ele = parsingEle_
            as DateCountBasisAndPayFreq;
      if (this.Type == CommandType.STRING)
      {
            this.Value = ele.ShortCode;
```

```
        }
        break;
}

case TradeElementType.Ctpy:
{
        CounterParty ele = parsingEle_ as CounterParty;
        if (this.Type == CommandType.STRING)
        {
                this.Value = ele.Shortcode;
        }
        break;
}

case TradeElementType.Broker:
{
        Broker ele = parsingEle_ as Broker;
        if (this.Type == CommandType.STRING)
        {
                this.Value = ele.ShortCode;
        }
        else if (this.Type ==
                CommandType.NumberWithoutSign)
        {
                if (ele.Credit != null)
                {
                        this.Value = ele.Credit.ToString();
                }
        }
        break;
}

case TradeElementType.Sales:
{
        Sales ele = parsingEle_ as Sales;
        if (this.Type == CommandType.STRING)
        {
                this.Value = ele.ShortCode;
        }
        else if (this.Type ==
                CommandType.NumberWithoutSign
                        && order_ == 1)
        {
                this.Value =
                    ele.CreditStandardQuality.ToString();
        }
        else if (this.Type ==
                CommandType.NumberWithoutSign
                        && order_ == 2 )
        {
                this.Value =
                        ele.CreditHighQuality.ToString();
        }
        break;
}

default:
```

```
                        break;
                }
        }
}
```

CommandAndStatus.cs

```
public class CommandAndStatus : System.Windows.Forms.UserControl
{
        private System.Windows.Forms.TextBox textBoxStatus;
        private System.Windows.Forms.TextBox textBoxCommand;
        private System.Windows.Forms.Label labelCommand;

        private ITradeController m_tradeController;
        private Command m_command;

        private char m_lastKeyChar;
        private System.Windows.Forms.Timer m_timer;
        private System.Windows.Forms.Label label1;

        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.Container components = null;

        public CommandAndStatus()
        {
                InitializeComponent();

                this.m_timer = new Timer();
                this.m_timer.Interval = 1000;
                this.m_timer.Tick += new EventHandler(this.OnTimerInvoked);
        }

        public ITradeController TradeController
        {
                set
                {
                        this.m_tradeController = value;
                }
                get
                {
                        return this.m_tradeController;
                }
        }

        public Command Command
        {
                set
                {
```

```
            m_command = value;
        }
        get
        {
            return m_command;
        }
    }

    public void FoucsOnCommandTextBox()
    {
        this.textBoxCommand.Focus();
    }

    /// <summary>
    /// Clean up any resources being used.
    /// </summary>
    protected override void Dispose( bool disposing )
    {
        if(disposing)
        {
            if(components != null)
            {
                components.Dispose();
            }
        }
        base.Dispose(disposing);
        this.m_timer.Dispose();
    }

    /// <summary>
    /// Required method for Designer support - do not modify
    /// the contents of this method with the code editor.
    /// </summary>
    private void InitializeComponent()
    {
        this.textBoxStatus = new System.Windows.Forms.TextBox();
        this.labelCommand = new System.Windows.Forms.Label();
        this.textBoxCommand = new System.Windows.Forms.TextBox();
        this.label1 = new System.Windows.Forms.Label();
        this.SuspendLayout();
        //
        // textBoxStatus
        //
        this.textBoxStatus.Anchor =
            ((System.Windows.Forms.AnchorStyles.Top
                | System.Windows.Forms.AnchorStyles.Left)
                    | System.Windows.Forms.AnchorStyles.Right);
        this.textBoxStatus.BackColor =
            System.Drawing.SystemColors.Window;
        this.textBoxStatus.Enabled = false;
        this.textBoxStatus.Font =
            new System.Drawing.Font("Microsoft Sans Serif", 8.25F,
                System.Drawing.FontStyle.Bold,
                    System.Drawing.GraphicsUnit.Point,
                        ((System.Byte)(0)));
        this.textBoxStatus.ForeColor = System.Drawing.Color.Black;
        this.textBoxStatus.Location =
```

```
      new System.Drawing.Point(128, 26);
this.textBoxStatus.Name = "textBoxStatus";
this.textBoxStatus.ReadOnly = true;
this.textBoxStatus.Size = new System.Drawing.Size(596, 20);
this.textBoxStatus.TabIndex = 1;
this.textBoxStatus.TabStop = false;
this.textBoxStatus.Text = "";
this.textBoxStatus.TextAlign =
      System.Windows.Forms.HorizontalAlignment.Center;
//
// labelCommand
//
this.labelCommand.Font =
      new System.Drawing.Font("Microsoft Sans Serif", 8.25F,
            System.Drawing.FontStyle.Regular,
                  System.Drawing.GraphicsUnit.Point,
                        ((System.Byte)(0)));
this.labelCommand.ForeColor =
      System.Drawing.SystemColors.WindowText;
this.labelCommand.Location =
      new System.Drawing.Point(2, 4);
this.labelCommand.Name = "labelCommand";
this.labelCommand.Size = new System.Drawing.Size(124, 20);
this.labelCommand.TabIndex = 56;
this.labelCommand.Text = "Fast Input Command";
this.labelCommand.TextAlign =
      System.Drawing.ContentAlignment.MiddleCenter;
//
// textBoxCommand
//
this.textBoxCommand.Anchor =
      ((System.Windows.Forms.AnchorStyles.Top
            | System.Windows.Forms.AnchorStyles.Left)
               | System.Windows.Forms.AnchorStyles.Right);
this.textBoxCommand.BackColor = System.Drawing.Color.White;
this.textBoxCommand.Font =
      new System.Drawing.Font("Microsoft Sans Serif", 10F,
            System.Drawing.FontStyle.Bold,
                  System.Drawing.GraphicsUnit.Point,
                        ((System.Byte)(0)));
this.textBoxCommand.ForeColor =
      System.Drawing.Color.Orange;
this.textBoxCommand.Location =
      new System.Drawing.Point(128, 2);
this.textBoxCommand.Name = "textBoxCommand";
this.textBoxCommand.Size =
      new System.Drawing.Size(596, 23);
this.textBoxCommand.TabIndex = 0;
this.textBoxCommand.Text = "";
this.textBoxCommand.KeyPress +=
      new System.Windows.Forms.KeyPressEventHandler(
            this.textBoxCommand_KeyPress);
this.textBoxCommand.MouseHover +=
      new System.EventHandler(
            this.textBoxCommand_MouseHover);
this.textBoxCommand.TextChanged +=
      new System.EventHandler(
```

```
                this.textBoxCommand_TextChanged);
        //
        // label1
        //
        this.label1.Font = new System.Drawing.Font(
              "Microsoft Sans Serif", 8.25F,
                    System.Drawing.FontStyle.Regular,
                          System.Drawing.GraphicsUnit.Point,
                                ((System.Byte)(0)));
        this.label1.ForeColor =
              System.Drawing.SystemColors.WindowText;
        this.label1.Location = new System.Drawing.Point(2, 26);
        this.label1.Name = "label1";
        this.label1.Size = new System.Drawing.Size(122, 22);
        this.label1.TabIndex = 57;
        this.label1.Text = "Command Feedback";
        this.label1.TextAlign =
              System.Drawing.ContentAlignment.MiddleCenter;
        //
        // CommandAndStatus
        //
        this.Controls.AddRange(new System.Windows.Forms.Control[]
              {this.label1,
               this.textBoxStatus,
               this.labelCommand,
               this.textBoxCommand});
        this.Name = "CommandAndStatus";
        this.Size = new System.Drawing.Size(728, 50);
        this.ResumeLayout(false);
}

private void textBoxCommand_KeyPress(
        object sender, System.Windows.Forms.KeyPressEventArgs e)
{
        this.m_lastKeyChar = e.KeyChar;
        this.m_timer.Stop();
        this.m_tradeController.CloseHint();

        if (e.KeyChar == 13)
        {
              ParseCommand();
              if (this.textBoxStatus.Text.Length == 0)
              {
                 if (
                    this.TradeController.IsAllNeededFieldsFilled())
                 {
                    this.TradeController.SendTradeRequest();
                    this.ParentForm.Close();
                 }
                 else
                 {
                    this.textBoxStatus.Text =
                          "required fields unfilled";
                    this.m_tradeController.ShowHint();
                 }
              }
        }
```

```
}

public void ParseCommand()
{
      this.textBoxStatus.Clear();
      this.textBoxStatus.Enabled = false;

      try
      {
            m_command = new Command(this.textBoxCommand.Text);
            m_command.OnCommandTextUpdate +=
                  new System.EventHandler(
                        this.HandleManuallyCommandUpdate);
            this.TradeController.Parse(m_command);
      }
      catch(Exception e_)
      {
            this.textBoxStatus.Enabled = true;
            this.textBoxStatus.BackColor = Color.White;
            this.textBoxStatus.ForeColor = Color.Black;
            this.textBoxStatus.Text = e_.Message;
            this.textBoxCommand.ForeColor = Color.Orange;
            return;
      }

      if (this.TradeController.IsAllNeededFieldsFilled())
      {
            this.textBoxCommand.ForeColor = Color.Green;
      }
      else
      {
            this.textBoxCommand.ForeColor = Color.Orange;
      }

      this.m_tradeController.ShowHint();
}

public void UpdateCommand(object sender_)
{
      ITradeElement element = sender_ as ITradeElement;
      if (m_command != null && element != null)
      {
            m_command.UpdateCommand(element);
      }
}

private void HandleManuallyCommandUpdate(
      object sender_, System.EventArgs args_)
{
      Command commandObj = sender_ as Command;
      this.textBoxCommand.Text = commandObj.CommandText;
      this.textBoxCommand.SelectionStart =
            this.textBoxCommand.Text.Length - 1;
}

private void textBoxCommand_TextChanged(
      object sender, System.EventArgs e)
```

```
	{
		if (this.m_lastKeyChar == 32)
		{
			ParseCommand();
		}
		else
		{
			StartParsingTimer();
		}
	}

	public void StartParsingTimer()
	{
		this.m_timer.Start();
	}

	public void OnTimerInvoked(
		object sender, System.EventArgs eventArgs)
	{
		this.m_timer.Stop();
		this.ParseCommand();
	}
}
```

Bibliography

1. British Standards Institution. 1998. *BS EN ISO 9241-11:1998. Ergonomic Requirements for Office Work with Visual Display Terminals. Part 11: Guidance on Usability*. British Standards Institution.
2. Greenberg, Saul. 1996. Teaching Human-Computer Interaction to Programmers. *Interactions*, Vol. 3, No. 4, 62-76.
3. International Organization for Standardization. 1997. *13407 Human-Centered Design Processes for Interactive Systems*. Draft International Standard ISO/DIS 13407.
4. Liberty, Jesse. 2003. *Programming C#, Third Edition*. O'Reilly.
5. Louis, Tristan. 2003. *Usability 101* [on-line]. Available from http://www.tnl.net/blog/
6. Mayhew, Deborah J. 1999. *The Usability Engineering Lifecycle: A Practitioner's Handbook for User Interface Design*. Morgan Kaufman.
7. Mayhew, Deborah J. 1992. *Principles and Guidelines in Software User Interface Design*. Prentice Hall.
8. Nielsen, Jakob. 1994. *Usability Engineering*. Morgan Kaufmann Publishers.
9. Olsen, Dan R. 1998. *Developing User Interfaces*. Morgan Kaufmann Publishers.
10. Quesenbery, Whitney. 2003. The Five Dimensions of Usability. *In Content and Complexity: Information Design in Technical Communication*, eds. Michael Albers and Beth Mazur. Lawrence Erlbaum Associates.
11. Reimer, Jeremy. 2005. *A History of the GUI* [on-line]. Available from http://arstechnica.com/articles/paedia/gui.ars
12. Rubin, Tony. 1988. *User Interface Design for Computer Systems*. Ellis Horwood Limited.
13. Sells, Chris. 2004. *Windows Forms Programming in C#*. Addison-Wesley.
14. Shaffer, George. 2004. *Linux, OpenBSD, Windows Server Comparison: Ease of Learning vs. Use & Repetitive Tasks* [on-line]. Available from http://geodsoft.com/opinion/server_comp/usability/repetitive.htm
15. Shneiderman, Ben. 1980. *Software Psychology*. Winthrop Publishers.
16. Stone, Debbie, Caroline Jarrett, Mark Woodroffe, and Shailey Minocha. 2005. *User Interface Design and Evaluation*. Morgan Kaufmann Publishers.
17. U. S. Department of Health and Human Services. *Usability Basics* [on-line]. Available from http://www.usability.gov/basics/index.html