CASE Tool Simplification

Via Task-Sensitive Metaphor

Rozita Naghshin

A thesis

in

Department of Computer Science

Presented in Partial fulfillment of the Requirements for the degree of Master of computer science at Concordia University Montreal, Quebec, Canada

August 2004

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-612-94750-5
Our file Notre référence
ISBN: 0-612-94750-5

The author has granted a non-exclusive license allowing the Library and Archives Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou aturement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Acknowledgements

I would like to express my gratitude to all those who made this work Possible.

First I wish to thank my supervisor Dr. Ahmed Seffah for his invaluable guidance and encouragement.

I would also like to thank all HCSE members for making the hours in the lab an enjoyable experience.

I am grateful to Bardia, a very special person in my life, for his understanding, support and encouragement.

Above All, I would like to express my deepest gratitude to my family for their constant love and support.

Abstract:

Computer-Aided Software Engineering (CASE) and in particular IDEs (Integrated Development Environments) tools play a major role in the software development and maintenance process. They promise to assist software engineers in achieving several complex software development tasks while increasing their productivity and performance. However, the complexity of their interfaces has made these tools not always useful and very difficult to learn and master. One of the causes for this complexity is the fact that these tools are functionality-oriented rather than human-centric. The interface of these tools has not been developed according to the real users' characteristics, and there is a gap between the user's conceptual model and CASE tool features. In this thesis, we proposed a framework to simplify the complexity of the interface. Our approach starts from the development of an effective set of developers' personae that model developers' behaviors and characteristics. Using these personae, we developed a method to reduce the complexity of CASE tools. This method combines a task-sensitive user interface metaphor with visual communication design principles. Several empirical studies were conducted while building and validating our framework.

Table of Contents

Chapter 1: Introduction 8

1.1. Motivation 8

1.2. Objectives and Methodology 10

1.3. Thesis Organization 12

Chapter 2: CASE Tools and CASE Tools Usability 13

2.1. Overview: CASE Tools 13

2.1.1. Definition 13

2.1.2. Classifications of CASE Tools: 13

2.2. Usability Issues Related to CASE Tools 16

2.2.1. Usability and Quality in Use 16

2.2.2. Empirical Studies on CASE Tools 18

2.2.3. Usability Issues Related to Complexity of the Case Tools 19

2.3. Case Study: Empirical Studies on Microsoft Visual C++ 22

2.3.1. Cognitive Walkthrough 23

2.3.2. Conducting CW to Evaluate Microsoft C++ Program 24

Chapter 3: Developer Persona 26

3.1. Using Persona as a UCD Method 26

3.1.1. About UCD 26

3.1.2. Persona and UCD 29

3.1.3. Persona in Scenario Based Design 33

3.2. How to Create Effective Set of Software Developer personae 35

3.3. Case study: Building a Set of Developer Personae for Microsoft C++ 38

3.3.1. Method 38

3.3.2. Results 41

3.3.3. Refining Persona Descriptions Using Behavioral Patterns 44

Chapter 4: Visual Simplification of CASE Tools 47

4.1. The Impact of UI Simplification on Usability Issues Related to CASE Tools 47

4.2. Visual Simplification Guidelines (Color, Icons, Layout) 50

4.2.1. Simplifying the Layout 50

4.2.2. Eliminating Redundancy 56

4.2.3. Simplifying Objects on Screen: 59

4.2.4. Choosing the Right Color and Typography 61

4.2.5. The Smart Use of Dynamic Displays, Animation and Sound 65

4.3. UI Simplification and User's Persona 66

Chapter 5: Task-sensitive CASE Tools 71

5.1. Overview 71

5.2. Task-Oriented User Interface 72

5.2.1. Wizard Dialog Box 72

5.2.2. Task Sensitive Toolbars 74

5.2.3. Task-Sensitive Interfaces 75

5.3. Case Study: Task-Sensitive Interface for Microsoft Visual C++ 77

5.3.1. User Task Analysis 77

5.3.2. Dividing Functions According to 5 Major Tasks 78

5.3.3. Sorting Factions for Related to Each Task 80

5.4. Task sensitive metaphor: a solution to complex CASE tools 86

5.4.1. Overview on UI Metaphors 86

5.4.2. Complexity of User Interface Related to UI Metaphors 91

5.4.3. Metaphor and User's Persona 95

5.4.4. Task Sensitive Metaphors 97

5.5. Case Study: Developing a Task-Sensitive Metaphor for Microsoft C++........ 99

5.5.1. Task Analysis 100

5.5.2. Finding an Appropriate Metaphor that Covers all Tasks in Problem Domain 100

5.5.3. Checking With the User's Characteristics in the Set of Persona 103

5.5.4. Mapping the Metaphor Attributes to the Domain Tasks 104

5.5.5. Testing the Metaphor 108

Chapter 6: Conclusion and Future Investigations 109

List of Figures

Figure 1.1. the steps involved in developing a task-sensitive interface for CASE tools. *11*

Figure 2.1. Level of CASE tools' integration *15*

Figure 3.1. UCD Activities *29*

Figure 3.2. The method to obtain persona *32*

Figure 3.3. The experimental infrastructure and equipment *40*

Figure 3.4. A screenshot of the video of a test session recorded using Camtesia *41*

Figure 3.5. Comparison of different types of interaction with the system for occasional users *43*

Figure 3.6. Comparison of Different types of interaction with the system for expert users *43*

Figure 4.1. Usability and UI complexity *48*

Figure 4.2. Simplifying the user interface by grouping the related elements in the interface. *51*

Figure 4.3. Inconsistency in layout: Lack of unity; no repetition *51*

Figure 4.4. Lack of unity and poor alignment has created unnecessary forms *52*

Figure 4.5. Visual unity through alignment *52*

Figure 4.6. Lack of contrast makes the interface difficult to understand *53*

Figure 4.7. Contrast and grouping draws user's eyes to the buttons on the left corner *53*

Figure 4.8. Standard view of Macromedia flash software *54*

Figure 4.9. Macromedia flash after user's customization *55*

Figure 4.10. Minimizing menu bar using intelligent agent *56*

Figure 4.11. Presenting a function in both tool bar and menu bar *57*

Figure 4.12. Redundancy in J Builder *58*

Figure 4.13. UI simplification though eliminating redundancy *59*

Figure 4.14. Extensive details and embellishment of icons *60*

Figure 4.15. A good simulation of tangible user interface for a Media Player *60*

Figure 4.16. Gratuitous dimensionality 61

Figure 4.17. Careless use of color in Icons 62

Figure 4.18. Wise use of color: Using color for grouping 62

Figure 4.19. Using 2 different colors for warning message dialog 62

Figure 4.20. Error in color: Color vision challenged users cannot recognize red vs.green. 63

Figure 4.21. Using type in tool palettes to add both function and style in Dreamweaver 64

Figure 4.22. BBC world news web site (demonstrates several uses of typography in volume) 64

Figure 4.23. Animated character used by Microsoft 66

Figure 4.24. Relationship between complexity and Usability 68

Figure 4.25. Design of Apple iMac version PC for 2 different groups of users 69

Figure 5.1. Wizard dialog box in Microsoft Access 73

Figure 5.2. Property toolbar when the user selects Text 75

Figure 5.3. Property toolbar when the user selects Button 75

Figure 5.4. Task tree for finding a word in a document (Farrell and Breimer 2000) 76

Figure 5.5. Major steps for creating task sensitive user interface 77

Figure 5.6. Task tree for major tasks in Microsoft Visual C++ 78

Figure 5.7. Dividing functions in to 5 major tasks 79

Figure 5.8. Microsoft Visual C++ task flow 80

Figure 5.9. Mapping functions into the interface for the task B (debug) 82

Figure 5.10. User's view of functions sorting for Task B (Debug) 82

Figure 5.11. Sorting for Debug-Edit-Compile ((A∩B∩C)-D) using USort 83

Figure 5.12. Result of sorting of for (Debug-Edit-Compile) 84

Figure 5.13. Overly literal translation of a real object 93

Figure 5.14. Poor real object UI metaphor 94

Figure 5.15. Unnatural way of interaction 94

Figure 5.16. Natural way of interaction 94

Figure 5.17. Choosing Einstein verses a puppy as a metaphor for an intelligent agent *95*

Figure 5.18. Using the same set of labels and metaphor in windows' applications *97*

Figure 5.19. The methodology for developing Task-Sensitive metaphor *99*

Figure 5.20. Icon representing Project management task *104*

Figure 5.21. Microsoft C++ interface (Edit and File Management) *105*

Figure 5.22. Edit Icon *105*

Figure 5.23. Compile Icon *106*

Figure 5.24. Microsoft C++ interface (Compile and file management) *106*

Figure 5.25. Debug Icon *107*

Figure 5.26. Microsoft C++ interface (debug and file management) *107*

List of Tables

Table 3-1. A set of persona for Microsoft Visual C++ *36*

Table 3-2. A sample of our data gathering for task "creating a project" *42*

Table 3-3. Revised Set of Personae for Microsoft Visual C++ *46*

Table 5-1. Sorting functions for 4 major tasks *81*

Table 5-2. Final result of function sorting *85*

Table 5-3. Microsoft C++ main tasks and Desktop Metaphor *100*

Chapter 1: Introduction

1.1. Motivation

In this thesis, I investigated visual simplification of complex CASE tools and in particular Integrated Development Environments (IDE). In the most recent version of the IEEE's Software Engineering Body of Knowledge, Carrington (2004, p. 10) offered the following goals of a software development environment.

> Software development environments are computer-based tools that are intended to assist the software development process. Tools allow repetitive, well-defined actions to be automated, reducing the cognitive load on the software engineer. The engineer is then free to concentrate on the creative aspects of the process. Tools are often designed to support particular methods, reducing any administrative load associated with applying the method manually. Like methods, they are intended to make development more systematic. They also vary in scope from supporting individual tasks to encompassing the complete life cycle.

Unfortunately, there is evidence that these goals are often not met in practice, and part of the problem seems to involve usability. For example, many developers have described IDEs to be difficult to learn and use. Also, printed documentation for IDEs, online help, and related training materials are often presented in language that is precise but esoteric and difficult to understand (Seffah & Rilling, 2001).

One of the major factors that contribute to rejection of CASE tools by developers is their complex user interface. The user interfaces of IDEs tend to be quite complex and difficult to understand, and developers may use only a small proportion of the total

available functionalities. The remaining functionalities may be perceived as not useful for specific tasks, or the developer may not even be aware that they exist (e.g., Desmarais & Liu, 1993). There are also concerns that the representation of software artifacts, such as classes in an object-oriented programming environment, often does not facilitate program comprehension. For example, developers' productivity may not be significantly improved than using visual instead of textual representations (Blackwell, and Green, 1999)

Some researches have attributed problems just described to a conceptual gap between functionality-oriented organization of IDE user interfaces and problem-solving needs of developers (Budgen and Thomson, 2003, Jarzabek & Huang, 1998, Shneiderman, 2002). It could also be described as a gap between the software engineers who develop CASE tools and the software engineers who use them but do not develop them. That is, the two groups of developers just mentioned may think in different ways about the task of programming.

However, such method-oriented tools may not directly support problem-solving activities, such as hypothesis testing of the comparison of alternative solutions. That is, developers probably do not think solely in terms of a particular programming method, so in this sense there may be a mismatch between tool user interface design and types of problem-solving skills preferred by developers.

1.2. Objectives and Methodology

The main objectives of this thesis are:

1. Conducting an empirical study that aims to clarify the main limitations of the current GUI of CASE tools,
2. Conducting empirical study that aims to model Developers while building and refining a set of empirical-based developer personae,
3. Investigating on the idea of adopting task-sensitive approach and visual communication principles to simplify the complex user interface of CASE tools.

Throughout my thesis, I will investigate how to simplify the complex interface of IDEs. I will also introduce the idea of simplifying CASE tools by developing a Task-sensitive User interface. Obviously these goals cannot be achieved without investigating the usability problems related to CASE tools and understanding the characteristics of end-users – software developers.

In order to validate and illustrate the applicability of my studies, I will show, a step-by-step approach on how to develop a task-sensitive interface for Microsoft visual C++ program. As outlined in figure 1.1, my research investigations includes of 2 parts. In the first part via some empirical studies, including Cognitive Walkthrough test, I will first analyze the usability problems related to CASE tools. Then by observing developers while interacting with the program, I will develop and refine a set of developer personae for Microsoft Visual C++.

Using the outcome of the first part, and considering the developers' characteristics, I will then try to simplify the Microsoft C++ interface and make the program more understandable by developing a task-sensitive interface.

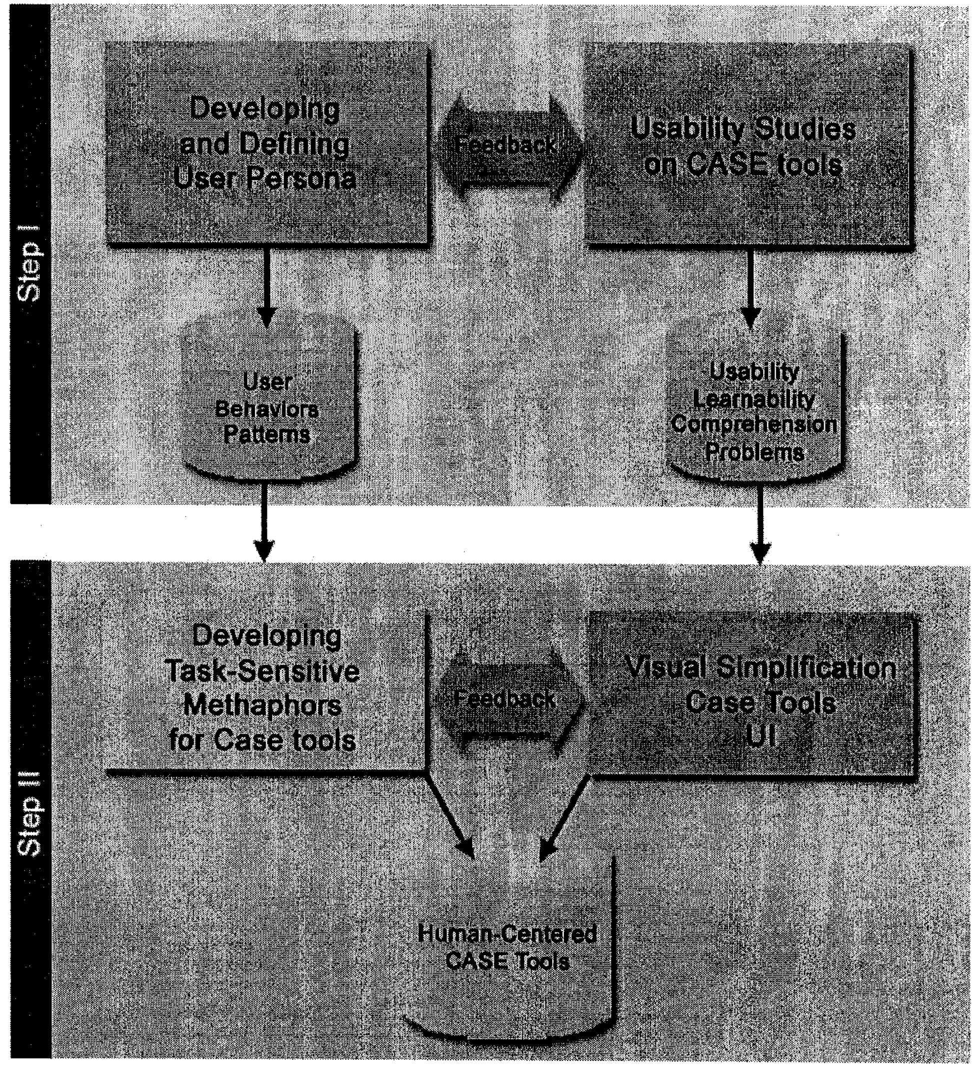


Figure 1.1. The steps involved in developing a task-sensitive interface for CASE tools.

1.3. Thesis Organization

The organization of this thesis is as follows.

After this introductory chapter, **chapter 2 "CASE Tools and Usability Problems Related to CASE Tools**" presents a brief review of CASE tools Then, it reviews the major usability problems related these tools. Finally it provides the result of my first empirical study on usability problems related to Microsoft visual C++ program.

Chapter 3 "Developer Persona" first introduces the idea of using persona as a tool in Human-Centered Design. It also explores the way of effectively constructing a set of personae. Using cognitive walkthrough, it provides a method to refine a set of personae for Microsoft C++ program.

In **Chapter 4 "Visual Simplification of Complex User Interfaces of CASE Tools"** I will review the methods for visual simplification of user interfaces. Then I will explore how user's characteristic/ Persona can influence the degree of simplification.

In **Chapter 5 "Task-Sensitive CASE Tools"** first I will introduce the concept of task-oriented user interfaces. Then, I will show how this approach has lead to a task sensitive interface for Microsoft Visual C++. Additionally, I will explore the idea of reducing visual complexity of software by proposing an original task-sensitive metaphor. I will then illustrate this idea through a step-by-step approach to develop a task-sensitive metaphor for Microsoft Visual C++.

Finally, I will summarize my work in **Chapter 6**, and I will present some ideas for future investigations.

Chapter 2: CASE Tools and CASE Tools Usability

This chapter provides a review on the CASE tools, followed by the main usability problems related to these tools. In a case study, it also demonstrates usability problems related to Microsoft Visual C++.

2.1. Overview: CASE Tools

2.1.1. Definition

One of the most challenging aspects of software engineering is how to maximize quality while minimizing software development span. Computer-Aided Software Engineering (CASE) technologies are designed to provide assistance for one or more software engineering activities in the process of software development. These activities include any of the managerial, administrative, or technical aspects of a software development method. The main goal of using CASE tools is the reduction of the time and cost of software development and software maintenance, and the enhancement of the product quality (Juric and Kuljis 1999, Zarrella, 1990).

2.1.2. Classifications of CASE Tools

There are several ways of classifying CASE tools, for example, interactive CASE tools such as Program Editor, or not-interactive such as Compiler. Most classifications of CASE tools start by considering whether the tool is upper CASE, lower CASE, or integrated CASE. An upper CASE tool (front end CASE) provides support for the early

stages in the systems development life cycle such as requirements, system analysis and logical design. These tools may also offer graphical editors that produce diagrams of system models, such as the Unified Modeling Language (UML) as well as documentation. In contrast, a lower CASE tool (back end CASE) focuses on later stages in software development life cycle. These tools support construction of software systems, such as code generation, compilation and testing. Integrated CASE tools support both the early and later stages. (Jarzebek and Huang, 1998, Zarrella, 1990)

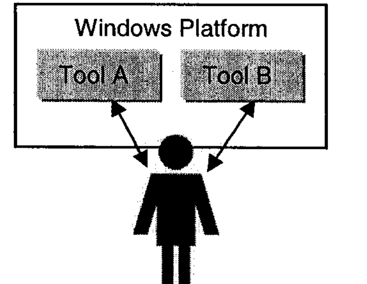
CASE tools are referred to as integrated simply because they either share a similar user interface within which to control CASE tools that are actually separate software products, or because a CASE tool package offers several tools to support more than one phase of the software development life cycle. The main goal of integrating tools is data sharing in a software development lifecycle. Data-exchange tools allow different CASE tools to share data, typically by importing or exporting data in a common format. Tool bridges, on the other hand, allow more seamless sharing of data. (Baik and Boehm, 2000, Pressman 2004)

Another way to classify CASE tools is by their level of integration. Wasserman (1990) classifies CASE tools into five different groups based on the level of their integration in software development environments.

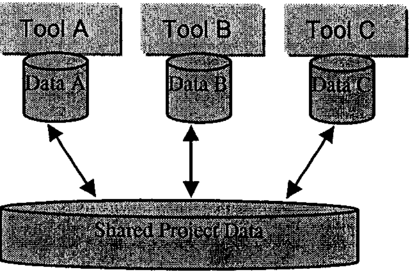
1. *Platform Integration*: when several tools run on the same operating system platform. (Figure 2.1-a)
2. *Data Integration*: when tools share data. (Figure 2.1-b)
3. *Presentation Integration*: when several tools have a similar user interface. (Figure 2.1-c)

4. *Control Integration*: when one tool controls the operation of other tools. (Figure 2.1-d)

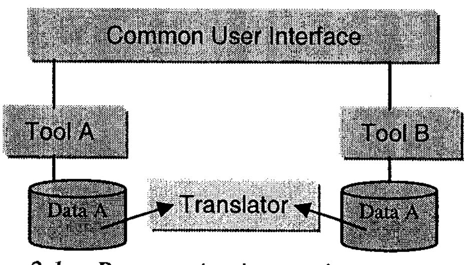
5. *Process/full Integration:* when tools support an explicit process model. (Figure 2.1-e)



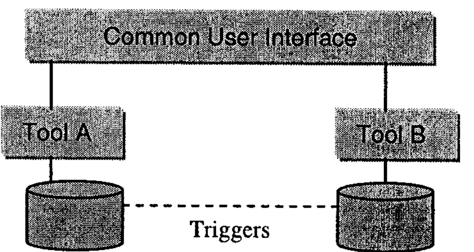
2.1.a. Platform integration



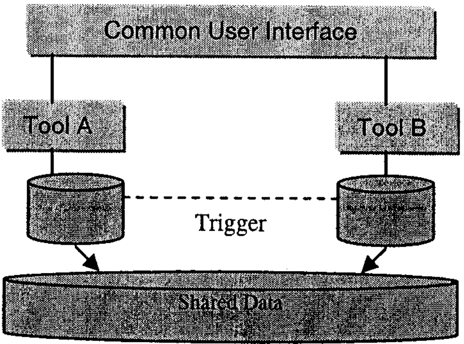
2.1.b. Data Integration



2.1.c. Presentation integration



2.1.d. Control integration (trigger mechanism)



2.1.e. Process/full Integration

Figure 2.1. Level of CASE tools' integration

Fuggetta (1993) classifies CASE tools according to the wideness of their support into three categories:

- *Tools*: support only specific task in the software-production process.
- *Workbenches*: support one or a few software process activities by integrating several tools in a single application.
- *Environments*: support all or at least part of the software production process with a collection of *Tools* and W*orkbenches.*

Finally CASE tools can also be classified according to the software development methodologies and programming method that they support. For instance some CASE tools support object-oriented programming methods such as C++ or Java, while other CASE tools specifically deal with Relational databases.

2.2. Usability Issues Related to CASE Tools

2.2.1. Usability and Quality in Use

The concept of usability has been defined differently in the various standard computer science literatures. ISO 9241-11(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.

According to ISO/IEC 9126-1 and 9216-4 (2000) usability is the capability of the software product to be understood, learned, used and attractive to the user when used in a specific context of use. In ISO/IEC 9126-1 five different useability factors are defined:

1. *Understandability*: The capability of a software product to help the user understand whether the product is suitable and how it can be used to achieve specific goals.
2. *Learnability*: The degree in which the software product helps the user learns how to use it.
3. *Operability:* The capability of the product to help user operate and control the software.
4. *Attractiveness:* The extent in which the product is attractive to the user
5. *Usability compliance:* The degree of adherence to standards, style guides or conventions concerning usability.

IOS/ IEC 9216-4 (2000) introduces and defines quality in use as a higher-order software quality attribute. According to IOS/ IEC 9216-4, there are four different factors of quality in use. These factors are:

1. *Effectiveness:* the capability of the software product to help the user user achieve specific goals with accuracy and completeness;
2. *Productivity:* the degree in which the software product enables the user to expend an appropriate amount of effort and/or resources in relation to the outcomes achieved.
3. *Satisfaction:* whether a software product satisfies the user.
4. *Safety:* The capability of the software product to minimize potential risks for harm to people, businesses, property, or software.

2.2.2. Empirical Studies on CASE Tools

As stated previously, one of the most challenging aspects of software engineering is how to maximize software quality while reducing development time. CASE tools are developed to help the software development process by increasing productivity, improving product quality, contributing to the software maintenance, and making software development a more enjoyable task. Even though the CASE market has rapidly grown during the past decade (A survey of the CASE tool market showed the annual worldwide market for CASE tools was $4.8 billion in 1990 and grew to approximately $12 billion in 1995 (Aaen, 1992, Russell, 1993). Current experiences suggest that CASE technology hardly delivers those promised benefits. After more than a decade, CASE tools have not been widely used. Therefore this phenomenon of sparsely usage of CASE tools is worthy of careful investigation.

Due to the importance impact of CASE tools on software development design, there have been several empirical studies, and all these studies report a relatively limited use of the tools.

In a survey done by Juric and Kuljis (1999), 233 system developers were asked several questions about CASE tools such as: if the CASE tools being used; The number of features within the tool that are being used; if CASE tools change the job of the systems developer in an unattractive way; and if the people who are expected to use CASE tools motivated to use them. The result of this survey showed that CASE tools were not being used in many companies. Moreover, among the companies that had adopted CASE tools, few people were actually using the tools.

Another survey of 53 companies, they found that 39 (73.5%) had never used CASE. Of the 14 companies who had tried CASE, five had subsequently abandoned use of the tools. People within these fourteen companies believed that use of CASE tools improved documentation quality, improved analysis, and resulted in systems that were easier to test and maintain (Juric,R. and Tanik, F.B. 1996). However, they also found the use of CASE tools difficult and time consuming. In a follow-up survey of thirteen managers who had been using CASE tools two years earlier reported discontinuation of use of CASE tools. The reasons for abandonment included cost, extensive training time, and lack of measurable returns. (Lending, and Chervany1998)

Livari (1996) surveyed several software development companies and found that about 70% of CASE tools were not being used one year after their purchase, and only about 5% were widely used not to capacity. Mover over, the developers tended to use only a relatively small number of CASE tool functionalities out of the total set available. Livari also found that the single most important factor in prediction of CASE tool usage was perceived voluntaries; if developers believed that applying of CASE tool was up to them, they tended not to use the tool.

2.2.3. Usability Issues Related to Complexity of the Case Tools

CASE tools are amongst the most complex software, and mastery of the tools requires extensive efforts from potential users. The complexity of current CASE tools results in a steep learning curve for those attempting to use them. A survey that was done by Seffah and Rilling on 2001 shows some usability problems related to complexity of the CASE tools, which are:

1. Most of the functionalities in integrated CASE tool are not always visible to the user;
2. Complex user interface overloads the user memory;
3. Tools do not provide the users with clear error messages;
4. CASE tools do not speak in users' language.

This complexity is due to the a) complexity of software development methods these tools support, b) the number of functions, and c) complexity of their User Interface.

a. Software Complexity and Software Development Methods.

Mastery of the Case tools requires mastery of one or more complex software design methodologies. The currently available methods that are supported by current CASE tools are quite complex, requiring extensive use of various types of diagrams and notations. A survey that was conducted by Jarzabek and Huang (1998) showed that the systems developers who used CASE tools were using formal methodologies more often than systems developers who do not use CASE tools. In other word most of the case tools only support formal software development methodologies. Due to the complexity of these methods, the CASE tools that support them also tend to be complex, resulting in a steep learning curve for those attempting to use them. Moreover, attempting to support several different methods greatly adds to this complexity.

The support from CASE tools usually is provided through an automated control of rules and an informative feedback on the violations of rules. Any violation of *rules* may result in the CASE tool preventing a user/developer from proceeding in a non-methodological way or simply issuing a warning/error message. The *process* support from CASE tools should include a guideline, encouragement or enforcement of using

prescribed steps defined by a methodology. (Baik and Boehm, 2000) More importantly, those who prefer to use their own methodology would not be hampered by the need to learn a new, complex methodology as well as the intricacies of a complex CASE tool.

b. Software Complexity and Number of Functions

As we discussed before, CASE tool users are not aware of most of the functionalities of the tools and relatively use a small amount of the features. Although these functionalities may be important for organizations that must manage teams of developers, they are not vital to the individual developer. Moreover, attempting to support several different methods greatly adds to the number of features and consequently increases the complexity of the tool, which makes the learning curve for the tool higher. Ultimately, even the simplification of tools by limiting the number of functions would benefit developers.

c. Software Complexity and User Interface

One of the basic concerns especially throughout CASE tools integration is the issue of developing a consistent, useable user interface. There is a general feeling that the user interface contributes significantly to the acceptance and learning time for a product. Moreover, more emphasis is being applied to integration of CASE tools through integration of user interface. The concept of integration with the end-user ranges from something as simple as maintaining a consistent user interface to something as complex as providing support for an expert system interface to aid in detailed design. In particular, a standardized user interface would help enhance user acceptance of CASE tools. Therefore, in addition to the underlying system, user interface consistency is becoming more of an issue to CASE tool producers. The fact that the user interface is the easiest of

the integration areas on which to standardize, CASE tool designers shouldn't promote a quick inefficient solution. If the CASE tool designers do not carefully analyze the requirements of the user interface, without taking the context of the tool usage into account, they may end up with an interface standard that promotes consistency at the expense of usability (Zarrella, 1990).

2.3. Case Study: Empirical Studies on Microsoft Visual C++

This study is a follow up of heuristic evaluation on CASE tools by Kline and Seffah in 2002. In the first study a total of seven professional C++ programmers participated in the heuristic evaluation of Microsoft Visual C++ version 6. The developers had relatively similar backgrounds and demographic characteristics: Most were in their late twenties or early thirties, and had an average of about 10 years experience with computer programming. All participants were familiar with C++ CASE tools, and were asked to give specific examples of usability problems. The same group also completed the Software Usability Measurement Inventory (SUMI).

The results of this experience showed few usability problems related to user control, flexibility, and minimalist design. However, there were major problems with other areas such as error handling, program help, and program comprehension.
During the second study, we used cognitive Walkthrough methods to collect more data of usability problems as well as user's interaction habits.

2.3.1. Cognitive Walkthrough

Direct observation of how a user interacts with a computer program provides a significant amount of data about both the software and the user. In the area of usability engineering, Cognitive Walkthrough (CW) is a review technique. It involves asking users to describe how they would perform certain tasks by "stepping through" the user interface. CW is widely used to evaluate the usability and learn-ability issues (e.g., how well the UI supports learning by doing) and effectiveness (e.g., how well the UI supports particular tasks). Such usage of CW is grounded in Lewis and Polson's CE+ theory of exploratory learning (Sim, Singer, and Storey 2001, Lewis, and Rieman, J. 1993, Weinstein, 1998). The CE+ theory is an information-processing model of human cognition that describes human-computer interaction in four steps:

a) The user sets a goal to be accomplished with the system (i.e.,, "edit a piece of software with the goal to understand it");
b) The user searches the interface for currently available services (menu items, buttons, command-line inputs, etc.);
c) The user selects the feature that seems likely to make progress toward the goal (i.e., highlight the name of classes and main variable in the program).
d) The user performs the selected action and evaluates the system's feedback for evidence that progress is being made toward the current goal.

For most realistic tasks that a user would attempt with a system, these four steps are repeated many times to achieve a series of sub-goals that define the complete task. The CW examines each of the correct actions needed to accomplish a task and evaluates whether the four cognitive steps will accurately lead to those actions.

The origin of the CW approach to evaluation is the code walkthrough familiar in software engineering (Tahir, 1997). CW requires a detailed review of a sequence of actions done by users. In code walkthrough, the sequence represents a segment of the program code that is stepped through by the developers to check certain characteristics (e.g., that coding style is adhered to) (Mikkelson, N., and Lee, W. O. 2000).

2.3.2. Conducting CW to Evaluate Microsoft C++ Program

Within the scope of this research, we used cognitive and code walkthroughs as a novel way for collecting useful observation data on usability, learnability, and program comprehension and construction of effective user archetype or persona, We will discuss this case study in details in the next chapter (Naghshin, Seffah, and Kline, 2003).

Data collected during our test included user interaction patterns and usability learnability and comprehension problems. We calculated the percentage of user's errors for each task, and we collected common errors for each group of users. Some of the usability, learnability and comprehension problems that emerge from this analysis are:

a) The "less experienced" participants started their tasks by simply opening the main program file without specifying the project. Even though the program let the user compile the program without specifying the project, it caused fatal errors related to the linking of files. This was an obvious kind of "beginner's mistake", but it was not prevented by the software.

b) The software represented the same functions in different ways, (e.g., often menu bar and the icon tool bar), which made the interface overly complex, especially for less experienced participants. In our test, the users sometimes

could not understand the fact that these representations concerned the same function until they had tried them both.

c) Mistakes related to link errors were not obvious to less experienced participants during debugging. They could not understand what the problem was, so instead of solving the problem, they used different paths to do the same task (e.g., running directly, compiling then running, or building all files then running), which would not solve the problem.

d) Both less experienced and expert participants had trouble understanding error messages during code compilation. All participants found the language of these messages to be overly technical and confusing.

e) The software did not provide the user with enough feedback, especially when they tried to compile the program. The messages received during compilation were confusing, even for the more experienced users.

f) Both less experienced and more experienced participants complained about the lack of a way to view the structure of the program in hierarchical way. The absence of this type of view made the C++ program more difficult to understand.

g) The software did not provide simple archival capability such as renaming a file, changing a folder or saving a project under a different name. Most of the users tried to do these tasks out of the Visual C++ program; however, this generated encountered link errors during program compilation.

Chapter 3: Developer Persona

In this chapter I will review the method of using persona effectively during the software development process. I will also demonstrate a step-by-step approach to build developers' personae for Microsoft visual C++.

3.1. Using Persona as a UCD Method

3.1.1. About UCD

Early studies on traditional software development tools and practices show that traditional methods have disappointingly small effects on improving software development (Curtis, Krasner, & Iscoe, 1987). In a study of around 8,000 software development projects conducted by over 300 American companies, the Standish Group found that only 16 per cent of projects were successful (completed on time and on budget, with all features and functions as initially specified). The three main reasons for project failure were: (1) Lack of user input, (2) incomplete requirements and specifications, (3) changing requirements and specifications (2-Standish Group, 1995). On the other hand, studies of companies that have adapted UCD approach in the software development process demonstrate a significant business improvement by improved usability.

User Centered Design (UCD) is the current term to describe a design philosophy that has been around for decades under different names such as human factors engineering, ergonomics engineering, usability engineering, or user engineering. UCD is a design philosophy that places the user at the center of the development process. It

refers to a multidisciplinary design approach based on the active involvement of users for a clear model of user archetype and task requirements, and the iteration of design and evaluation. It is considered as the key to product usefulness and usability. (Nielsen, 1994; Vredenburg & Butler, 1996).

UCD approach comprises a set of several steps, methods and tools designed to assist engineers and developers in addressing the issues related to usability in design of interactive systems. The UCD approach assists in the process of collating design information obtained using a variety of user oriented data gathering techniques. The concentrate of the UCD approach is that it provides a structure to assist the developer in assuring that relevant design issues have been considered in a user oriented manner. According to Gould (1991), the main four principles in UCD is as follow:

1. Early and continual focus on users, through direct contact to understand cognitive, behavioral, attitudinal, and characteristics of users and their goals
2. Integrated design, where all aspects of usability evolve in parallel and under one focus
3. Continual testing with users to qualitatively and quantitatively measure performance of intended users doing real work with simulations and prototypes
4. Iterative design, where an interactive system is progress is modified based upon the results of user testing cycles.

There is wide range of UCD techniques, and a subset of these is currently in common use. Some evaluation techniques, such as formal user testing, can only be applied after the interface design or prototype has been implemented. Others, such as

heuristic evaluation, can be applied in the early stages of design. Each technique has its own requirements, and generally different techniques uncover different usability problems. The following methods are the most popular and the most practiced techniques in UCD approach:

- **Heuristic evaluation,** also known as expert evaluation, is a technique used to identify potential problems that operators can be expected to meet when using a computer or a telematics application;
- **Usability walkthrough**. Users, developers and usability specialists review a set of designs individually and then meet to discuss each element of the design in a walkthrough meeting.
- **Prototyping**. Designers create simulations of interface elements (menus, dialogues, icons, etc.) using paper, video, or computer techniques, which are then presented and discussed with the users.
- **User-based observation** is used to obtain design feedback. This is a relatively quick and inexpensive way to conduct a user-based evaluation of a working system or prototype.
- **Co-operative evaluation**. Users employ a prototype as they work through task scenarios. They explain what they are doing by talking or 'thinking-aloud', which is recorded on tape and/or captured by an observer.
- **Questionnaires**. SUMI (the Software Usability Measurement Inventory) is a prime example of the use of questionnaires to collect subjective feedback.
- **Cognitive workload**. Measuring cognitive workload involves assessing how much mental effort a user expends while using a prototype or deployed system.

- **Focus groups** bring together various stakeholders in the context of a facilitated but informal discussion group.
- **Individual interviews** are a quick and inexpensive method to obtain subjective feedback from users based on their practical experience of a product.

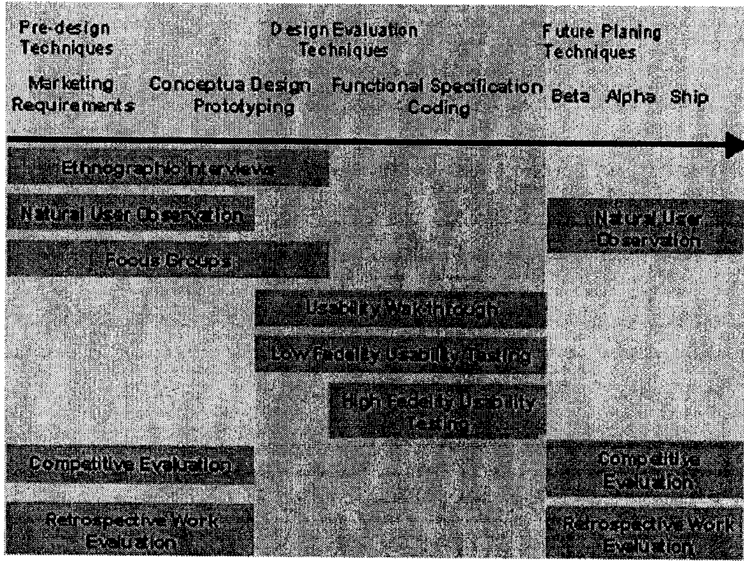


Figure 3.1. UCD Activities

3.1.2. Persona and UCD

In user-centered design method, it is obviously essential to take into account the users' characteristic: disabilities, age, gender cultural background as well as computer knowledge. Moreover, user motivations and attitudes can have an important impact on his or her performance and automation. The methods for studying User's work habits and behaviors have been adapted by empirical studies in software engineering from intermediary disciplines, such as human-computer interaction and business management. These methods can be divided into two groups: psychological and social. Psychological approaches are centered on studying individuals in controlled experiments. These studies

require performing tasks or filling out questionnaires, and they may occur in a laboratory or an office. Sociological methods are centered on making observations of *people working in systems*. Sociological studies tend to have a broader focus, looking at interactions between people and particularly in groups. (Naghshin, Seffah, and Kline, 2003)

The main principle of all UCD methods is the evaluators, as well as the software designers, work with real users. However, in practice it is impossible to interact with the real users throughout all phases of software development, and having the targeted user sitting by a software developer is unrealistic and impossible. Moreover, in the case of most commercial software products, such as Microsoft, targeted users are too broad and hardly accessible. Hence, most of the user characteristics and behaviors might be overlooked by software developers during design and development phase.

According to Allan Cooper (1999), a well-known HCI practitioner and the "Father of Visual Basic", in order to please a broad audience of users, the logic of designing broad functionalities in a product is erroneous. Cooper introduces "user persona" as a model of a user (also called a user archetype) and sometimes user profiles, which can be used during a user's absence.

The persona is a fictional and detailed "real-life" character that captures and represents the behaviors, goals and motivations of a group of actual or potential users of a software product. The persona focuses on modeling users' personal and practical goals instead of depicting a psychological profile or class. (Weinstein, 1998)

The idea of creating personas to improve the process of creating a product is not a new concept; it has been part of marketing studies for some time. The goal of the marketing team is to produce the maximum appeal to their audience. They generally

create a set of personae which best represents their marketing audience based on statistical and demographics data. By establishing personae, they are able to create a marketing campaign that will be successful in exactly the areas that they choose to focus on.

In software design, the goal is to develop a product that best meets the needs and goals of the users. Cooper proposed personae for modifying the design of this product (Cooper 1999). As mentioned, a persona is a profile of a fictional character that describes attributes of a certain type of potential program user. A persona typically includes a fictional name and may also include information about family, educational background, and work goals. If constructed effectively, a persona should be sufficiently informative and engaging so that it redirects the focus of the development process toward the end user and their needs.

A typical product commonly has several types of users who regularly use or maintain the product. A distinctive persona for each type of user must be created which should include name, age, gender, technology environment, education, and job scenario. Personal information such as, "Charles plays golf and sometimes tennis …" should be added to provide more information and make the persona engaging for software developers/designers.

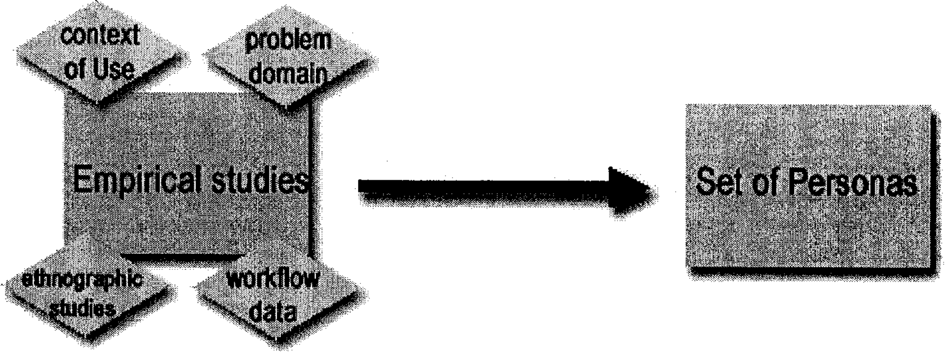


Figure 3.2. The method to obtain persona

The persona is obtained by analysis of domain, usage and workflow data gathered through rapid ethnography studies and contextual inquiry research methods. (Figure 3.2) As an example, productmarketing.com, has produced a persona Robin, the product manager based on its customers' experiences.

> "Robin is a product manager for an enterprise B2B vendor with a direct sales force. She manages all aspects of product management for three products. She is 35 years old with a college degree and some MBA classes. She earns $85,000 a year and is eligible for an $8,000 bonus based on company profit, product revenue, and personal quarterly goals. Robin is a power user of Microsoft® Office XP. She runs Internet Explorer 6 on Windows® XP. She has a reasonably new laptop running 1024 x 768 resolution. Using Microsoft® Outlook®, Robin ends and receives about 100 emails each day, and she attends a dozen internal meetings each week."

3.1.3. Persona in Scenario Based Design

A scenario is a written story that describes the future use of a system from a specific and often fictitious user's point-of-view. Scenarios are used for a variety of purposes: to evaluate system functionality, to design attributes and features, and to test theory. The scenario can be used at many levels in the design process. It is common to use a scenario in the beginning of the design process to illustrate user needs, goals and actions. Some designers use scenarios during the whole design process and refer to them repeatedly. Others use them only as an offset for the creative process, never referring to them again. (Nielsen, 2002)

However, scenarios alone are not enough. The scenarios rarely presented the users as vivid characters. At best, more often than not, they are the stereotype average user. It has raised some questions such as: how can you predict the goals and actions of a user when you don't know anything about the user as a person?; why use descriptions of users that the reader can't engage in?; and what does it take to write a good description of a user? Without an appropriate persona, it would be impossible to be involved with the user through a scenario, especially when the user's experiences are far from your own and the lack of involvement would make it difficult for the software designers to predict and imagine the user's actions.

Cooper defines scenarios as "a concise description of a persona using a software-based product to achieve a goal." Cooper puts an emphasis on the user's goals, whether it is company goals or personal goals. The personas are hypothetical archetypes of actual users, defined and differentiated by their goals. They are described from a goal hierarchy, where personal goals have priority, to practical goals, and practical goals that are affected

by the company goals. For Cooper, it is important to create believable personas, which can be accomplished by creating specific details and being precise in the description.

Ex. **Angela persona**: Angela is a 31-year-old PR consultant who is based in Los Angeles, but who has customers throughout the entire West Coast. Angela often has to travel during the week.

Angela's Goals:

- Always be on time for client meetings
- Travel without hassle
- Don't feel stupid

Angela's Scenario: Angela is on her way to Seattle and has a 30-minute layover in an unfamiliar airport. She really wants to grab a cup of coffee before she heads to her connecting flight. After Angela disembarks, the airport map and service details are downloaded to her PDA via a wireless local network, using Bluetooth. Angela quickly finds her favorite coffee shop in the list, and sees it is only a few minutes walk away. The Way-finder shows Angela exactly how to find the coffee shop, with handy landmarks indicated on her map. Angela follows the directions the Way-finder gives her, fully finds the coffee shop. Soon she's enjoying a double-tall, fat- free Mocha Latte Grande, with sprinkles. Now Angela needs to find her way to the gate. She uses the Way-finder to look up the gate for her connecting flight, and then follows the directions it gives her. Angela arrives at her gate with plenty of time to spare. This description is an example of the Goal-Directed method. It is built on descriptions of users and scenarios. Angela is described from her personal and practical goals.

While Cooper focuses on the description of the user (personas, in his term), Carroll does not rely on the depiction of users as such. The user-descriptions are embedded in the scenarios. However, in both cases using an engaging user persona has made the scenario more interesting and has given a better description of the user's specific goals. (Carroll, 2000)

3.2. How to Create Effective Set of Software Developer personae

Many programmers as well as executives, sales representatives, and marketing personnel mistake the CASE tool developer as the ideal CASE tool user targets. The CASE tool designers are rarely good representatives of targeted customers. First of all the CASE tool developer may use a software development method that is totally different from the method the user will employ. Secondly, tool developers and tool users may use differed programming languages. More importantly, quite often CASE tool developers have the expertise in this area when compared to a regular user.

Creating a set of developer personae will help CASE tool designers to have a better understanding of users. Using personae during CASE tool development process, one company realized that their developers of Java™ tools were not representative of their users of Java tools. While both sets of developers were using Java, the vendor programmers were vastly more advanced than the customer programmers. (productmarketing.com) Table 3.1 shows a set of personae that for Microsoft visual C++ program.

A Set of Personae for Microsoft Visual C++		
Bob Johnson	**Jims Smith**	**Charles Butler**
• 21 year old • Student, First year in computer science. • Plays hockey in the university's team. • Knows a little bit of programming and uses Microsoft C++ for his assignments. • He lives far from the school and is often late. He wants to finish his assignment as soon as possible and go home.	• 24 year old • Masters student in computer engineering • Internet and gaming addict • Works in Microsoft as a C++ programmer. • He loves his job and he loves playing with the software and discovering new feature that not everyone knows. • Very fast learner and hard worker.	• 46 year old • Project manager of a small size company • He plays golf and sometimes tennis • He knows programming in different languages. • He hates asking people how to do things and loves to give the impression of a smart person. • The worst thing anyone can tell him is he is not fast enough.

Table 3-1. A set of persona for Microsoft Visual C++

It's easy to collect a set of user characteristics, give a fictional name, and call it a user persona; however, it's not so easy to create and effective user persona that can be used as a design tool. The major risk is the challenge to get the right persona or set of personas without stereotyping. Persona implies choices and biases that could over generalize or exclude user. A persona needs to be solid and concrete, based into reality.

A persona can be (a) primary, which needs are so unique that it calls for a distinct interface form and behavior, (b) secondary, which needs are going to be fulfilled by the by the primary interface with minor modification / addition (Cooper 1999). According to Cooper, to please a broad audience of users, the logic of designing a broad array of functionalities in a product is wrong. This strategy inevitably end up creating awkward

and complex-to-use products that tries to please everyone and in fact pleases no one in a significant manner.

Personae should be based primarily on user and empirical data. For example, the set of our personae in Table 3-1 may represent distinct types of users of the software. However, looking at this set, we notice that Bob only needs to finish a limited task as fast as possible, which is one of the Thomas' goals as well. Therefore, if we satisfy Thomas, Bob will satisfy also, and we can then consider Bob a non-effective or secondary persona. This is because each project targets different users in different contexts of use.

Each persona needs to have a name, an occupation, and personal characteristics, such as his likes and dislikes, his needs, and desires. Each persona should have specific goals related to the project. These goals can be personal (e.g., having fun), work-related (e.g., hiring staff), or practical (e.g., avoiding meetings). Moreover, Grudin and Pruitt alleged repeated use of the same personas will starch them so much that they loose precise representatively. Cooper believes that for each project a different set of personae should be constructed. This is because each project targets different users in different contexts of use.

Finally we should keep in mind that replacing the real user with user persona in a software development process, especially during usability evaluation, is a mistake. Personae should be only used as an additional support medium when the real user is not accessible, or not able to identify the real user.

In the next section, I will demonstrate how one can elicit and refine an effective persona through walkthroughs. Similar to our approach, Tahir (1997) suggested creating user profiles through contextual inquiry. Mikkelson (2000) explains also some of the

difficulties in describing and using personae within a scenario-based approach. These practitioners, along with Cooper, are clear in positioning persona descriptions as the starting point, which usable products can be constructed.

3.3. Case study: Building a Set of Developer Personae for Microsoft C++ program

The main goal of this case study was to enhance our initial personae of Microsoft Visual C++ developers we developed based on domain analysis, data gathered through rapid ethnography studies and contextual inquiry research methods. (Table 3.1) through observations of the interaction habits as well as usability, comprehension, and learn-ability problems encountered by users of this particular CASE tool. We actually used a modified form of CW in that we applied it to the UI of an extant CASE tool instead of a planned software program. We provided our test users a sample program of medium difficulty and asked them accomplish five major tasks common to most CASE tools.

3.3.1. Method

Research participant: We conducted the CW with a total of 6 users. They were all computer science graduate students with similar backgrounds in C++ programming knowledge. They were all still programming in C++ during their graduate studies or their work in industry. A total of three users could be considered more expert in the use of Microsoft Visual C++ due to extensive prior experience with this tool. In this sense their backgrounds may be more like that of the "Jim" persona in Table 3-1. They were very motivated and wanted to "show off" their expertise in using Microsoft Visual C++. The other 3 participants were more occasional users of Microsoft Visual C++ in that they had

been mostly working with another case tool for C++. The size of the program used for the tasks described below was around 1000 lines of code stored in 11 different files. This program includes six major classes and was written by one of the investigators (R.N).

Tasks: As mentioned, it is essential in CW to identify relevant tasks. The five kinds of tasks common to basically any integrated CASE tool involve the creation of a project, editing of source code, compilation of the whole application or part of it, debugging, and archiving the generated application, related code source, and documentation. In our evaluation, we tried to ensure that these kinds of tasks did not require specific knowledge of the C++ language per se.

***Used Infrastructure and Equipment*:** We used two computers, one for the user and one for the observer (R.N.), and a camera for recording the user's interaction with the keyboard (Figure 3.3). We also used the following software: Adobe Premier used to import the keyboard interaction from camera via a USB port, Camtasia and Timbuktu Pro Enterprise Edition Multi-Platform Remote Control. Camtasia Studio is a video screen-recording studio made up of five applications for recording, editing and sharing videos of human interactions with software applications. Camtasia was used to record both the use of the keyboard and mouse and the user's orientation toward the screen. The Timbuktu suite is a multi-platform solution for user support, systems management, telecommuting, and collaboration across a LAN, WAN, the Internet or dial-up connections. It was used to allow the observer to operate a remote computer as if he or she was actually sitting in front of that computer. This allows the observer and the user to transfer files and folders or communicate by instant message, text chat, or voice intercom. Timbuktu was used in this study to remotely record the user interactions (Figure 3.4).

The six users were given a sheet of paper that described the nine tasks of the CW. Each user then worked alone at a computer in one room while the researchers monitored him/her from another.

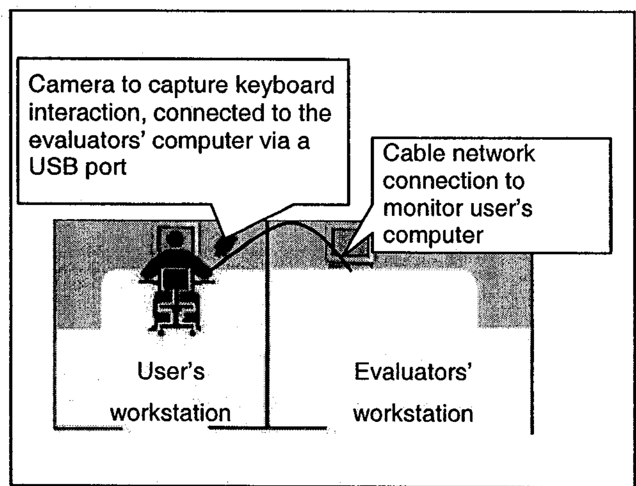


Figure 3.3. The experimental infrastructure and equipment

The six users were given a sheet of paper that described the nine tasks of the CW. Each user then worked alone at a computer in one room while the researcher monitored him/her from another computer in a separate room. If the user had a problem and asked a question, the researcher would answer to it (fully or partially, depending on the question).

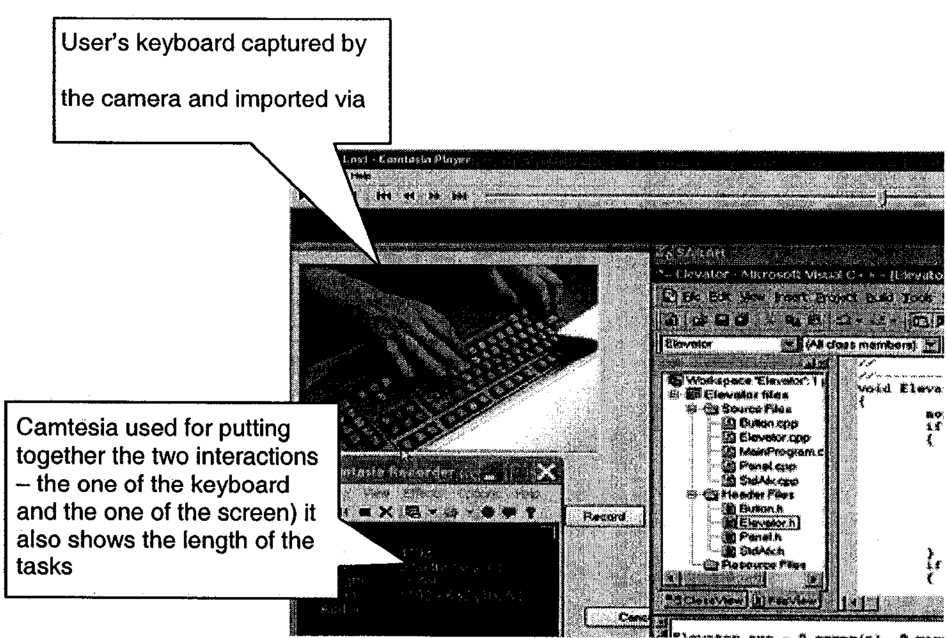


Figure 3.4. A screenshot of the video of a test session recorded using Camtesia

3.3.2. Results

Data collected during our test included user interaction patterns and usability problems. Both kinds of data are described below.

Users' behaviors patterns: Data about program interaction were saved in Microsoft Excel files for each task and user as is shown in Table 3.2.

Create a project					
	Open C++	Open the main program			
Interactions					
Progression Time	Start	SStar t			Finish
	[illegible]	[illegible]		[illegible]	
Menu bar				1	
Icons		1			
Short cuts					
Out side C++	4				
Right click menu					
Direct interaction with the objects					
Dialogue box			1		3
Interaction with the window					

Table 3-2. A sample of our data gathering for task "creating a project"

Then we calculated the total number of interactions for each user (Table 3.2). At the end we calculate the average of using each type of interaction for each group of users and compare them as a pie charts (Figures 3.5 and 3.6).

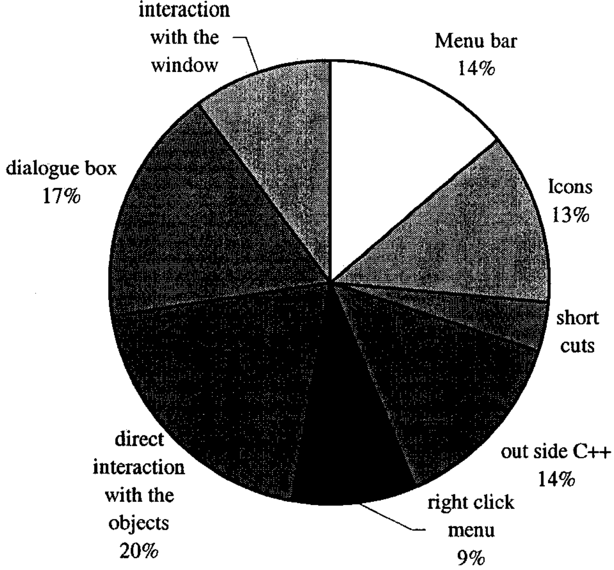


Figure 3.5. Comparison of different types of interaction with the system for occasional users

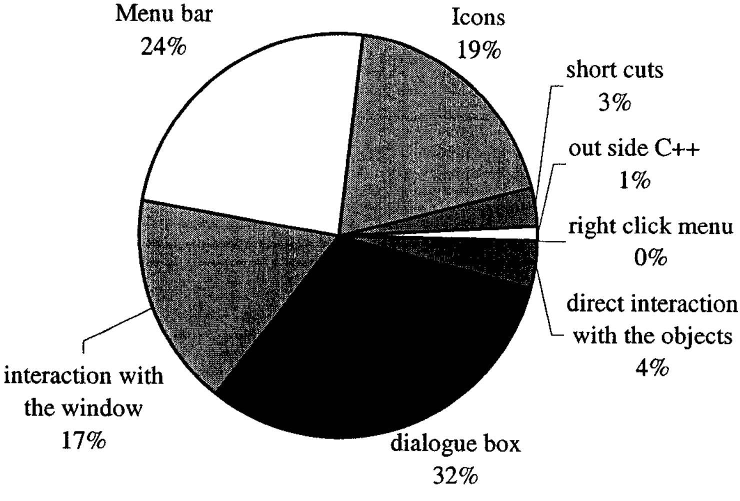


Figure 3.6. Comparison of Different types of interaction with the system for expert users

Some user behaviors patterns that we gathered by playing back our test and comparing the pie charts are summarized below:

1. In Visual C++ and many other CASE tools, the main functions are represented in the tool bars for quick access, as well as the in the menu bar. Even though users should use the toolbar more than the menu bar as it is more obvious, our results indicated that they used both the menu bar and the toolbar. Moreover, when faced with an error message, occasional users tried the same function both in the menu bar and tool bars.
2. The right click menu to display properties or options was used mainly by expert users, which made the interaction faster and easier. However, the occasional users rarely used this tactic.
3. Keyboard shortcut usage was minimal except for functions such as Ctrl+C or Ctrl+V, which are generic short cuts in most Windows-based software. However, in our study, the use of the software was limited to not more than 30 minutes. They may start using short cuts after a certain period of time (more than 2 hours) for some repetitive tasks.
4. Direct interaction with the objects on screen (such as clicking on an icon indicating a header file) was used for navigation in the program, and all users attempted this type of interaction but unfortunately occasionally the system did not allow the use of direct interaction.

3.3.3. Refining Persona Descriptions Using Behavioral Patterns

After conducting all tests, we added details to our original personae based on the description of potential usage environment, typical workday (or other relevant time

period), current solutions and frustrations, relevant relationships with others, and goals. In this process, we attempted to avoid any temptation to include irrelevant personal detail. One or two tidbits of personality can bring an otherwise dull persona to life, but too much biography will be distracting and will make the persona less credible as an analytical tool. If every aspect of the description of persona cannot be related back to data, it's not a persona.

The new of personae descriptions for this study after incorporating the results of the CW are presented in Table 3.3. Briefly, these updated personae include some biographical information, but now most descriptions concern preferred means of program interaction given the biographical data. For example, the description of Thomas Johnson, the less experienced hypothetical user, is based mainly on results of the CW with the less experienced actual users. The kind and specificity of information included in this updated persona should now be both engaging and informative for making tools more human-centric.

Jims Smith	Charles Butler
 • 24 year old • Master student in computer engineering. • Internet and gaming addict. • Works in Microsoft as a C++ programmer. He loves his job and he loves playing with the software and discovering new feature that not everyone knows. • Very fast learner and hard worker. • Jim believes C++ version 6 has a lot of functions, which makes it a powerful tool, and he has checked computer magazine to see if there is any new function added in C++ version 7. • He loves fast way of interaction. He tries right-clicking a lot and trying to remember useful shortcuts. • He is very impatient. When compiling takes long he loose his patience and restart the computer.	 • 46 year old • Project manager of a small size company • He plays golf and sometimes tennis • He knows programming in different languages • The worst thing anyone can tell him is he is not fast enough. • Thomas wants to see only basic functions. He believes 60% of existing functions in Microsoft C++ unnecessary and only for marketing reasons. • He doesn't understand why he has the run function in toolbars and menu bars. Aren't they doing the same thing? • He never uses right clicks or short cuts; he uses what he sees • He doesn't know that compiling a file without creating a project can generate errors, and when he gets errors he blames the software. • He hates long error messages. He prefers to see clear short messages. He never goes through all long messages anyway • He wants a hierarchical view of the program to understand the project without asking his employees, or spending his weekend going through the files rather than playing golf! • He can't believe he is prevented from renaming files, and he's afraid to ask others for help since he believes a project manager should know these tasks.

Table 3-3. Revised Set of Personae for Microsoft Visual C++

Chapter 4: Visual Simplification of CASE Tools

This chapter provides an overview on how by reorganizing and simplifying the visual representation of software's functionality to the user we can make the software more usable, keeping the tool's structural complexity. It also demonstrates the effect of users' characteristics on this simplification.

4.1. The Impact of UI Simplification on Usability Issues Related to CASE Tools

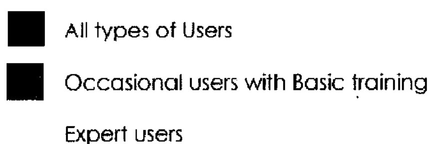
A graphic interface directs the users focuses, and makes the organizational structure of the computer system or multimedia document visible and accessible to the user. DOS, UNIX and other command-line operating systems have long been criticized for the complexity of their user interface. This interface evolved in the late 60's and early 70's with the emergence of Graphical User Interface (GUI). Researches showed that people could learn to use applications with a graphical interface more quickly than with commands. The graphical interfaces were also easier to remember and helped users get more done quickly. They also were more fun to use.

Computers have revolutionized the way people work, but the effectiveness of human computer interfaces has limited the effectiveness of computers. User's demands on software have changed; they expect to be able to sit down and use software with little or no frustration. For software users, ease-of-use has become a prime factor in decision making about which software to buy. Time is valuable; people do not want to read hundreds of pages of manuals and spend their time figuring out how the software works.

Even though GUIs enabled a revolution within a revolution, increasing visual complexity of these interfaces has put their effectiveness at risk. (Myers B. A. 1993)

The UI complexity will increase when the structure of software becomes more complex, and in turn will make the software less user friendly and more difficult to learn. As Tullis (1983) believes, reducing complexity increases usability; therefore the software is usable for larger range of end-users (Figure 4.1).

As we said before CASE tools are among the most complex software applications and one of the major issues that have an important impact on CASE tools complexity is the complexity of their user interface.



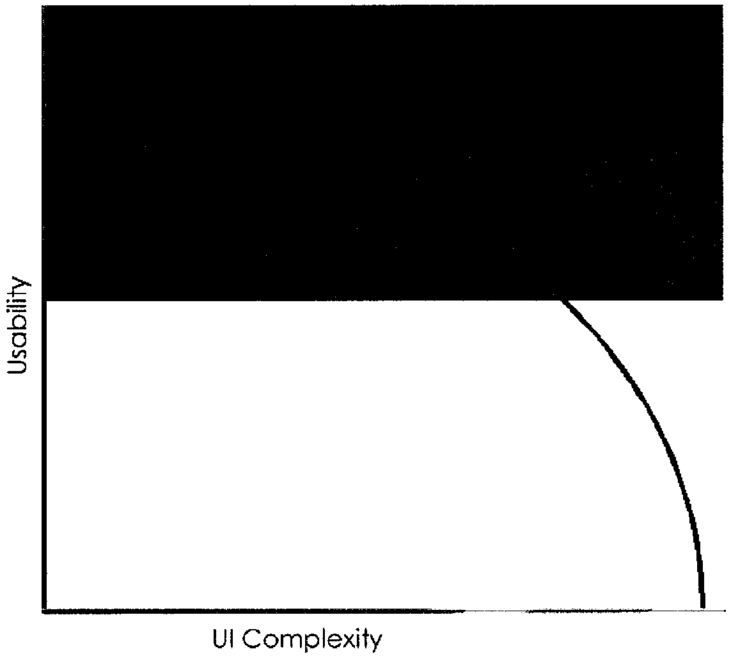


Figure 4.1. Usability and UI complexity

All empirical studies done on case tools shows significant usability problems related to user interface. Even though by reducing the functionality of the CASE tools and creating less complex software, this results in diminishing the software capability, which is against the ultimate goal of most of the CASE tool producers.

Amongst Norman's Principles of Good Design are visibility, good conceptual model, and good mapping between actions and results, between the controls and their effects, and between the system state and what is visible, (Norman, 1988, p.54) which can be achieved through a simple, elegant design. The term elegance derives from the Latin *eleger*, meaning to "choose out or "select carefully". Elegance in design is seen in the novel approach that solves a problem in a highly economical, simple way. Less is usually more - if a simple design will work, why complicate matters. Elegant solution produces a maximum of satisfaction from an absolute minimum of components. Techniques of simplicity, clarity, and consistency of user interfaces can improve the communication, effectiveness and performance (productivity) for the complex software applications. (Mullet and Sano, 1995, p 19) In particular, the use of appropriate typography, layout, color, animation, and symbols can assist software designer to achieve more efficient, effective communication with more diverse users' communities. (Norman 1988)

Mullet and Sano (1995) believe that simplicity in design depends upon three closely related principles: the elements in the design must be unified to produce a coherent whole; the parts must be refined to focus the viewer's attention on their essential aspects; and the fitness of the solution to the communication problem must be ensured at every level. The benefits of simplicity are functional as well as aesthetic:

- **Approachability.** Simple designs can be rapidly understood well enough to support immediate use or invite further exploration
- **Learnability.** A simple design makes the system easier to learn
- **Usability.** Simple designs that eliminate unnecessary variation of detail make the variation that remains more outstanding and informative, which can enhance usability. (Mullet and Sano, 1995, p 26-28)

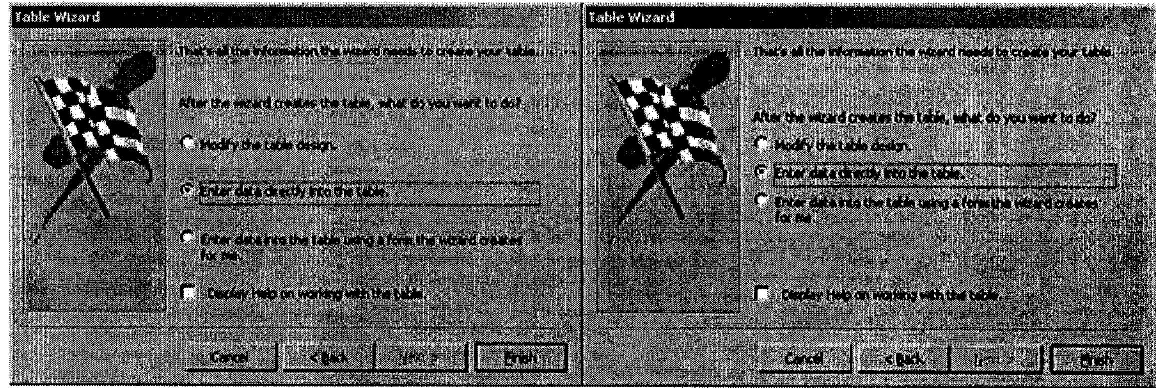
4.2. Visual Simplification Guidelines (Color, Icons, Layout)

We can simplify a visual design by: a) simplifying the lay out; b) eliminating redundancy; c) simplifying the icons, symbols, and signs; d) choosing the right set of colors and typography; and e) the smart use of dynamic displays, animation and sound.

4.2.1. Simplifying the Layout

There are four visual design principles that one can apply to any layout, whether for a book, brochure, web page, or computer software interface to make the interface simpler and more understandable.

i) Proximity Grouping Related Items: The principle of proximity states that related items should be placed physically close together and be separated from unrelated items by space. Grouping related items gives visual clues to the logical organization of the information. (Figure 4.2)



a) Violates principle of proximity *b) Visual unity through proximity*

Figure 4.2. Simplifying the user interface by grouping the related elements in the interface

ii) Visual Consistency in Layout: The principle of consistency indicates that the designer should keep the consistency of the layout by keeping the position and size of a visual element throughout the design. Users should not wonder whether different words, situations, or actions mean the same thing. (Figure 4.3)

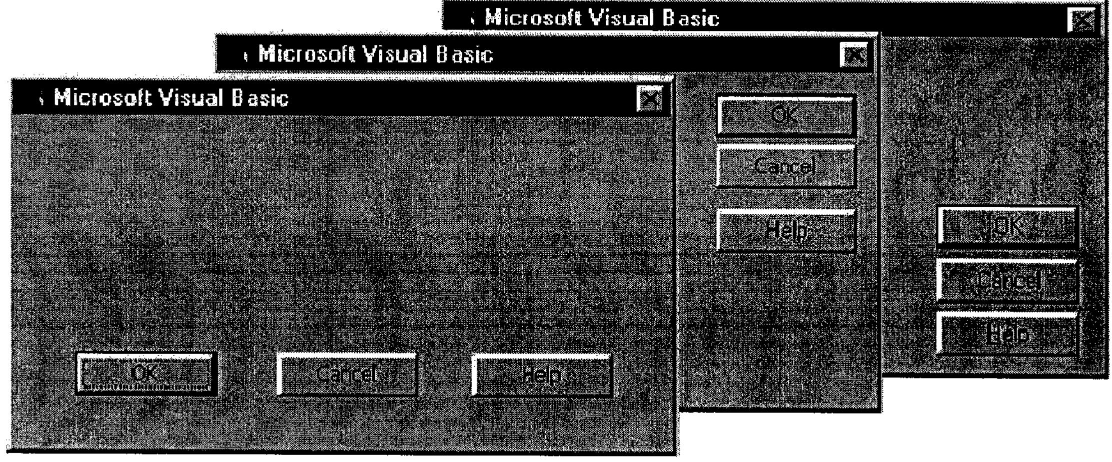


Figure 4.3. Inconsistency in layout: Lack of unity; no repetition

iii) Alignment: Alignment in layout creates visual unity and simplifies the design by eliminating unnecessary forms on the screen

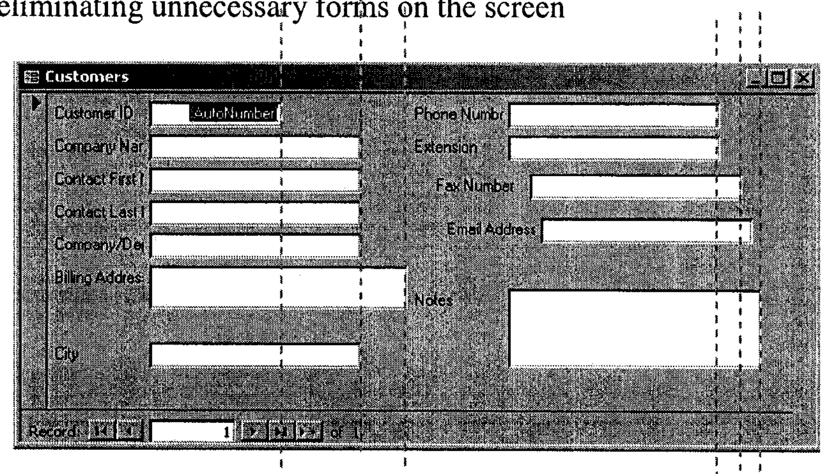


Figure 4.4. Lack of unity and poor alignment has created unnecessary forms, which makes the interface complex

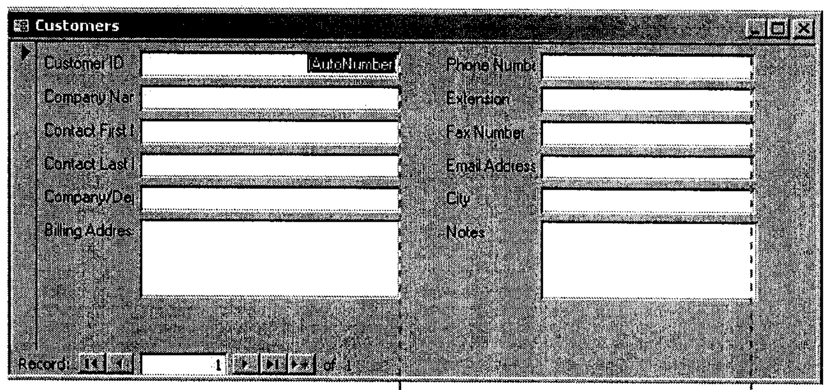


Figure 4.5. Visual unity through alignment

iv) Visual Contrast in Layout: Any successful design must have a clear focus on one or a small number of modular units. This focus make the interface easy to understand by directing the user's attention to the most important / essential task, or the starting point. The interface designer can create this focus by making elements very different (adding contrast). The visual contrast must be established by manipulating the perceptual qualities of size, value, orientation, texture, shape and position.

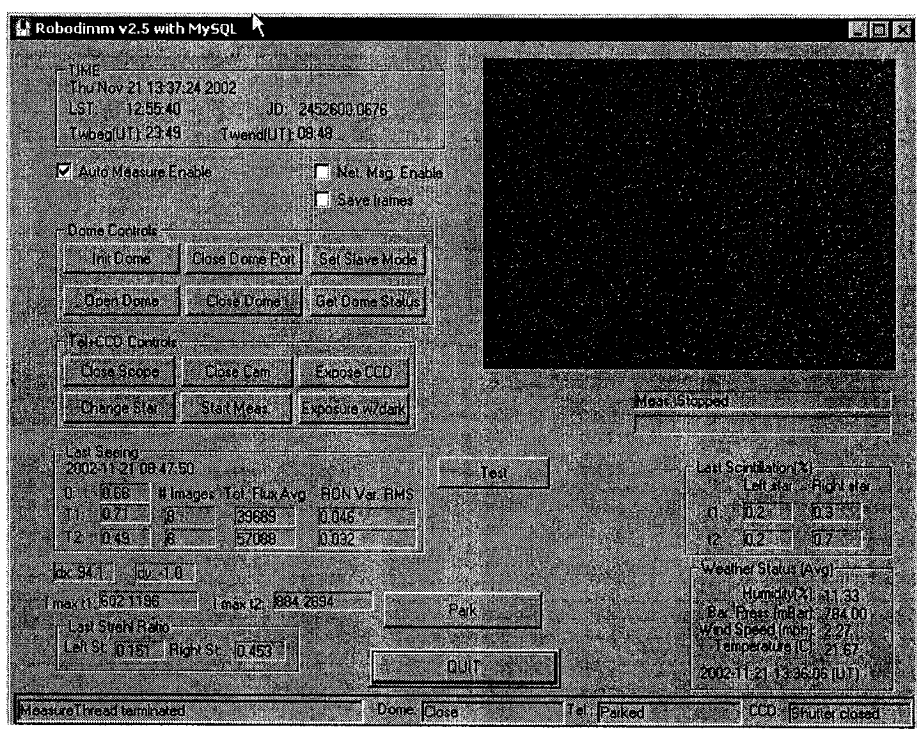


Figure 4.6. Lack of contrast *makes the interface difficult to understand*

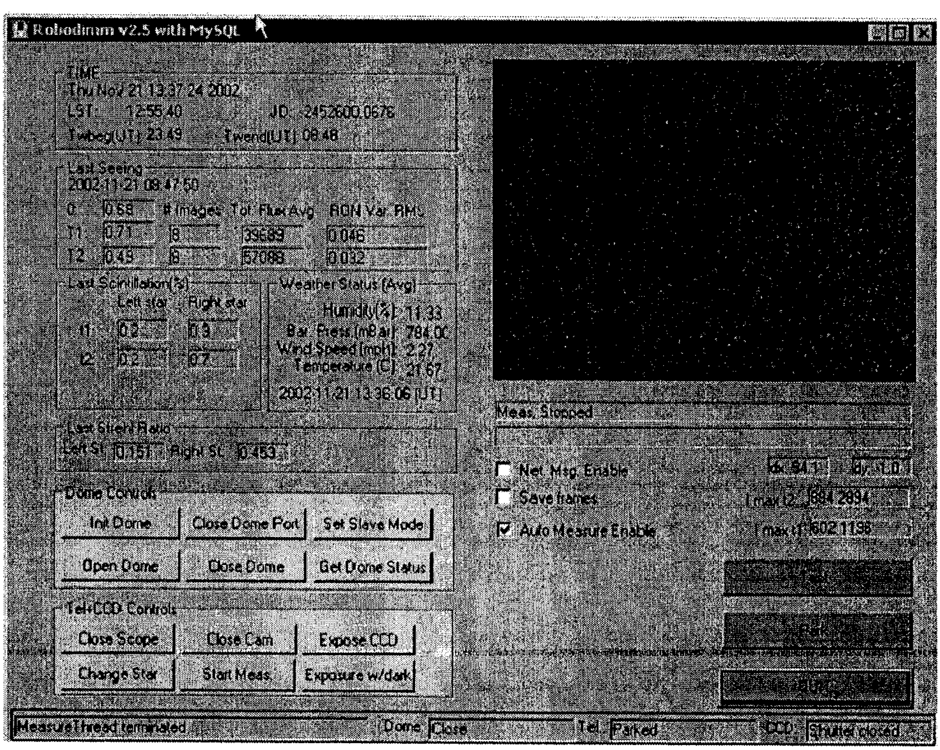


Figure 4.7. Contrast and grouping draws user's eyes to the buttons on the left corner

-Layout Customization: Customization of user interface is one way that makes the software more flexible. In this case, UI can be changed and adapted to the user's needs either by the user himself or by an experienced agent. In the case of less complex tools such as Internet Explorer or Media Player, UI customization increases the user satisfaction since the user experiences more control of the software. However, customization of complex user interface either by the user or though an experienced agent can make the software very difficult to use.

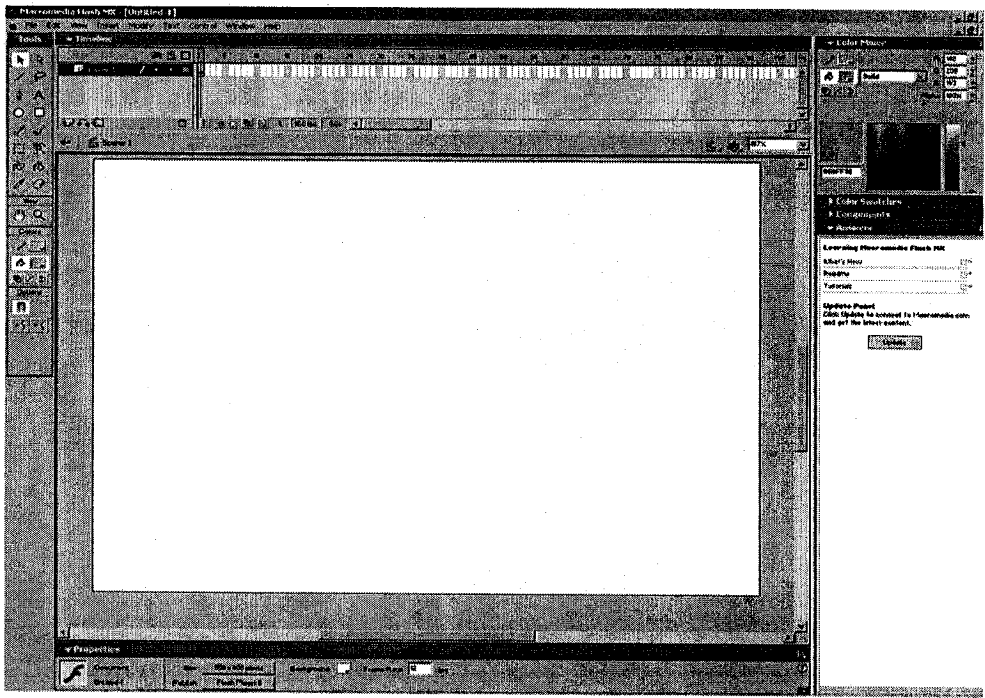


Figure 4.8. Standard view of Macromedia flash software

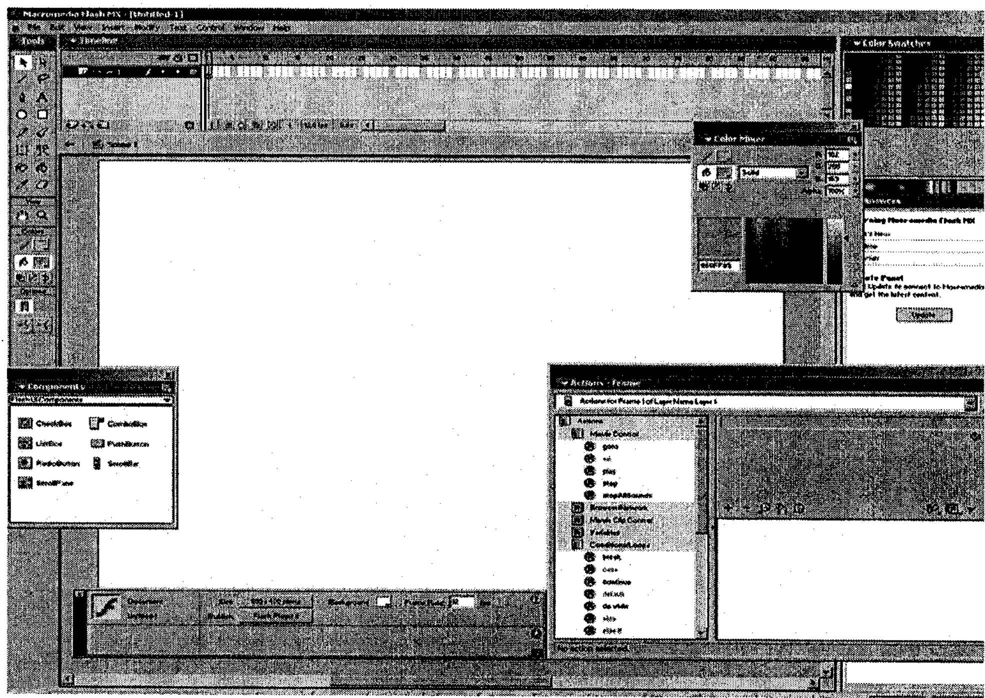
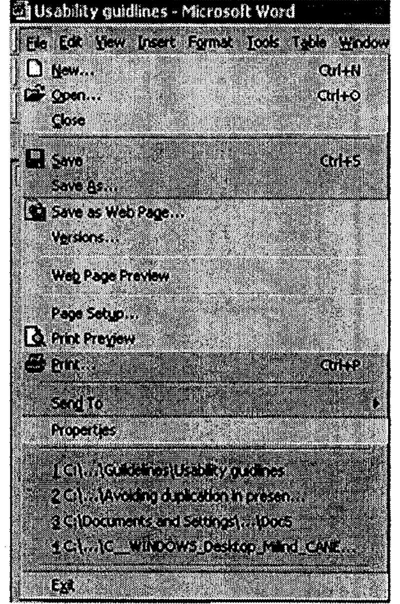


Figure 4.9. Macromedia flash after user's customization

Not all users are necessarily good visual designers. By rearranging, adding or deleting elements on the screen, they make the layout more complex and increase the user's memory load since the screen is always changing. (*Figure 4.8 and 4.9*) This customization sometime has been done by an experienced UI. The agent commonly gathers the data related to user interactions and changes the interface accordingly. This only serves to make the users more confused since the user does not have any direct control on these changes. For example, the menu bar in all Microsoft tools changes according to the frequency of using the items in it (*Figure 4.10*). This increases the user's memory load since people memorize the items in a list according to their place related to items besides them, which is being changed by the agent.



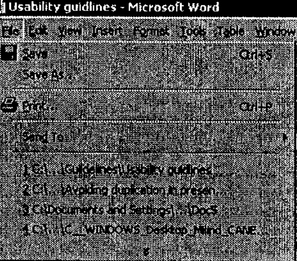


Figure 4.10. Minimizing menu bar using intelligent agent

4.2.2. Eliminating Redundancy

Redundancy can be defined as providing multiple modes of interacting with a software product. Most of the user interface designers and software developers tent to enhance flexibility of user interface by adding redundancy to accommodate a variety of users, irrespective of physical description, level of attention, level of experience or cultural background. It gives the users the opportunity to use whatever senses, skills and knowledge they possess to have a satisfying and successful experience (Nielsen and Molich, 1990). This flexibility increases UI complexity. Therefore in the case of very complex tool, it makes the software very difficult to use.

For example, the CASE tools representing alternative interaction methods for a specific feature has made the interface very complex, which confuses the users. Most of the functions in all CASE tools are accessible in different way, such as tool bars, menu bar, and right click, as well as keyboard shortcuts. Keyboard shortcuts and right click,

which are commonly used by expert users, make the interaction faster without adding any complexity to the interface. However, presenting the same feature by adding an object in the interface increases UI complexity and causes confusion. (Figure 4.11)

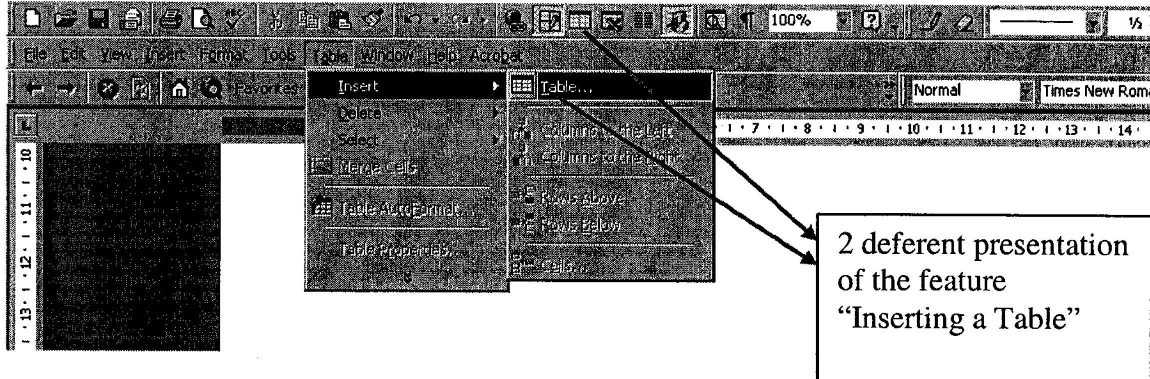


Figure 4.11. Presenting a function in both tool bar and menu bar

In the usability study we conducted in Microsoft Visual C++, we realized that duplication usually confuses the users. Our participants were not sure if a feature in the menu bar and the tool bar were the same; therefore, they would spend additional time trying both. We observed that users were frequently trying different way to activate the same feature even though they had already attempted this using a different control. Avoiding duplication in presenting a feature on an interface and selecting the most appropriate natural interaction technique is one of the best ways to decrease interface complexity.

Mulett (1995, p. 56-59) believes that a UI designer should eliminate any unneeded redundancy remains through: a) Reviewing the functional role played by each element in the design; b) Looking for situation where multiple elements have the same functionality;

c) Examining if a single element, possibly after minor modification, can fulfill the role of both elements; and f) Eliminating duplications.

As you can see in Figure 4.12, the modified file icons can play both roles of opening the files as well as demonstrating the already opened files; hence the right hand side file icons are not needed. (Figure 4.13)

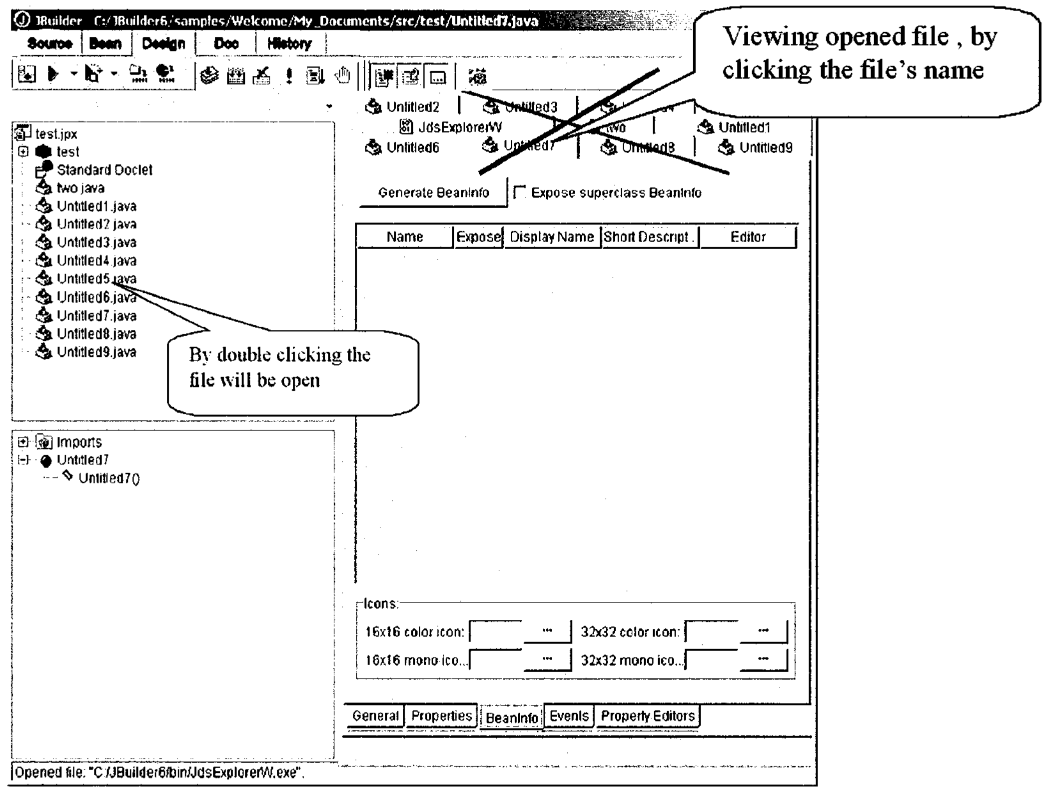


Figure 4.12. Redundancy in J Builder

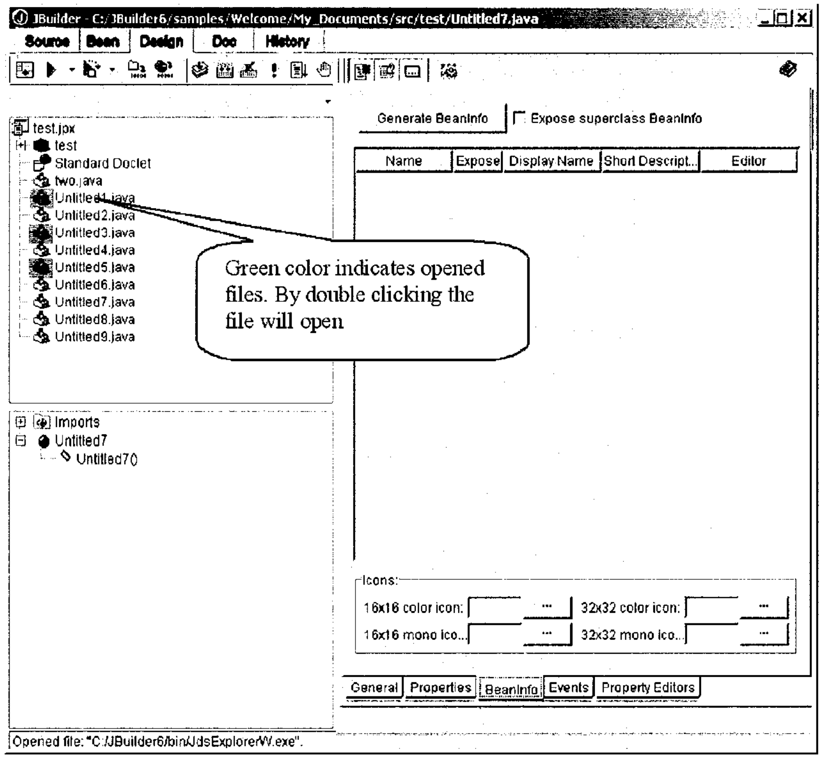


Figure 4.13. UI simplification though eliminating redundancy

4.2.3. Simplifying Objects on Screen:

Extensive detail and exaggeration of photographic realism is difficult to resist. In the GUI more effort has been expended on faithfully replicating familiar object than on uncovering the unique characteristics and qualities of the new electric media itself. (Mullet and Sano 1995) Graphical embellishments that serve only to emphasize on the "realism" of the design such as brushed aluminum button or the secular reflections on spherical plastic buttons rarely add to the long-term visual appeal of the product. They subvert rather than enhance the communications by making the screen more crowded (Figure 4.14).

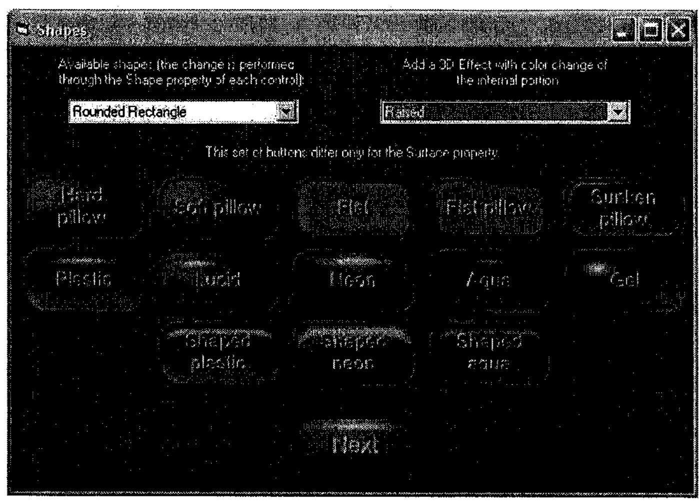


Figure 4.14. Extensive details and embellishment of icons

Another mistake that some UI designer make is unjustified dimensionality. Most people love the sense of tangibility simulated by the popular pseudo-3D rendering technique in which highlighted and shadowed borders create a physically raised surface. It can even be argued that this visual treatment plays a valuable role in identifying "pushable" controls (Figure 4.15). However, the increasing use of 3D in situations that do not take advantage of the added dimensionality cannot be justified. As you can see in the Figure 4.16, the 3D structure of tool bars is a complex animated rotation whenever the user switches to a different set of tools. This design makes the interface less serious and more difficult to use.

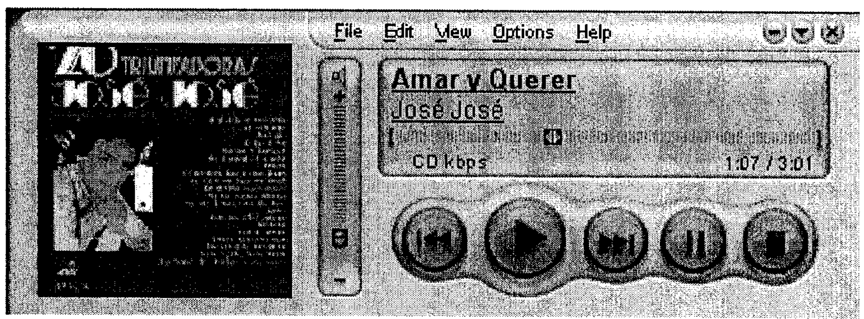


Figure 4.15. A good simulation of tangible user interface for a Media Player

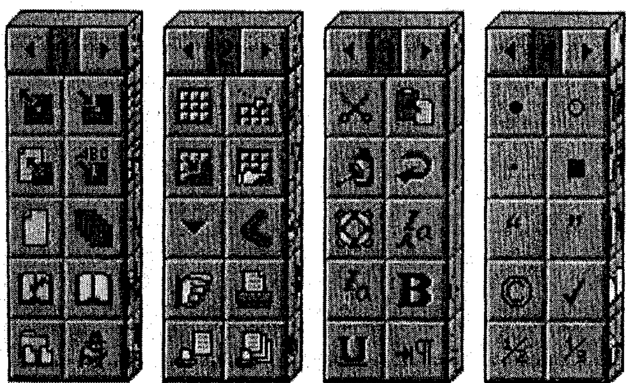


Figure 4.16. Gratuitous dimensionality

Josef Muller suggests three basic techniques to simplify the objects on screen:

a) Regularizing the elements of the screen

b) Reducing the detail of design elements to it's essence

c) Combining elements for maximum leverage

4.2.4. Choosing the Right Color and Typography

Typography and color are two a major part of the graphical user interface (GUI). They are among the powerful design tools that can contribute to the quality of use and enhance software understandability.

Color is certainly useful when a warning or informational message needs to stand out such as: a) warning an urgent event; b) evoking a mood or environment, c) giving the user cues, or d) grouping sets of information (Mullarky, 1998). However, hasty and extensive use of color can result in increasing complexity of the user interface. Using too many colors causes a "visual chaos", similar to that caused by too many typefaces, and will be less appealing. (Constantine, 2002)

Figure 4.17. Careless use of color in Icons

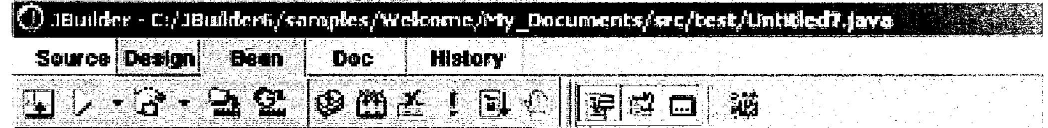


Figure 4.18. Wise use of color: Using color for grouping

i) Colors should be used consistently in the interface. Same color should refer to the same message throughout the interface to avoid confusion (*Figure 4.19*).

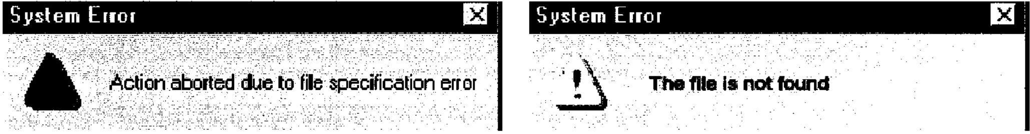


Figure 4.19. Using 2 different colors for warning message dialog

ii) Color should match the user's characteristics such as age and cultural background. Different colors have different meaning to different people that varies greatly from one culture to another. Fore instance Red, means danger in the U.S., death in Egypt, and life in India.

iii) Relying only on color can cause usability problems. It is estimated that approximately 7% of all adults have some form of color vision deficiency. If a software feature relies only on color distinction to separate items on screen, 7% of all users will not see this difference (Figure 4.20). A good principle for using color in the user interface

is to design the screen in black and white, make sure they are workable, and then add color to enhance the design.

00:16
3 min timer
I. INFORMATION FORM
Error: Please correct the information in red
Leave blank: | Date of Birth: 10/11/2004
Last Name: John | First: Smith | M.I.: *
Dealership: *
Address: 204 Gaspe #569
City: Montreal | State/Province or Country: Quebec
Zip/Postal Code: H3c 1H
Phone: 514-674-8697 | Fax: 514-674-8697
Email: *
Leave blank: | Leave blank:
BACK NEXT QUIT

Figure 4.20. Error in using color: Color vision challenged users cannot recognize red versus green.

Good user interface design also depends on the understanding of how type works as a visual element. There are two ways of using type: in isolation and in volume. Type in isolation can be used as an element incorporated into a larger design. This type is most commonly used in user interface design as a word or phrases to convey a message, or in relation to other visual elements such as icons, button, menus, or other user interface conventions. (Figure 4.21) Type in isolation favors a font design with strong distinction, something that carries a stylistic message in a concentrated space. Type in velum is used to present continuous linear text. The presentation of type is controlled by left-to-right, top-to-bottom visual syntax and margins. (KAHN and LENK, 1998)

Figure 4.21. Using type in tool palettes to add both function and style in Dreamweaver version 01

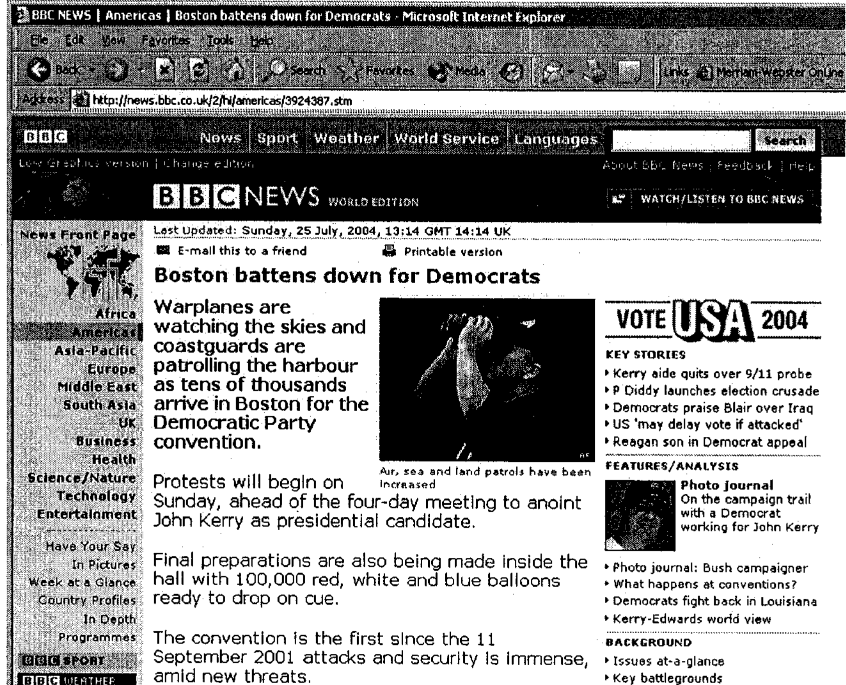


Figure 4.22. BBC world news website (demonstrates several subtle and effective uses of typography in volume)

Wise use of typography can enhance the interface by making it more readable, more comprehensible, more memorable, and more appealing. It also reduces the complexity of the software. Some principles of simplifying the interface by choosing the right typography are as follow:

a) Reduce the number of typestyles on screen. Extensive use of number of typeset increases the amount of form and shape in screen, which makes the screen visually complex.

b) Choose appropriate typestyles that are suitable for digital documents. Studies indicate that there is no difference between paper and high quality screen displays. Nevertheless, reading from screens remains anecdotally problematic. In part, this situation may be due to differences between the conditions of the experiments. For example, Times Roman is a serif font commonly used to display on-line text; however, this font was not designed for screen display and the wide range of gray makes it very difficult to read on screen. On the other hand, Georgia and Verdana were both designed for screen display. A study conducted by Dan Boyarski on the subject of reading for comprehension and subjective preference of on-line fonts, revealed that Georgia font performed well and facilitated ease of reading on-line text compared to Times Roman. (Marcus, A. 1989)

c) Within each typeface, choose or design a set of enhanced letterforms and symbols with which to represent the text effectively.

4.2.5. The Smart Use of Dynamic Displays, Animation and Sound

Animation and sounds makes interfaces more enjoyable, and understandable. However, empirical results indicate that there is not enough evidence to guide the design of animated interfaces to improve human performance (Robertson, Card and Mackinlay 1993). Very little is known about the design and effective use of animation in user interfaces. A majority of designers use animation and sound only to make the interface more pleasant and engaging for the user. Even though these sounds and animations may provide amusement for a short period of time, they eventually become an annoyance, especially when the user does not have the control to stop this function. When using

Microsoft Word program, all users are familiar with the little paper clip or the puppy dog that pops up to help. It sits quietly and gives the user some tips from time to time. As amusing it looks, most users find it annoying. They desperately try to find the way to get rid of it, which is not very easy.

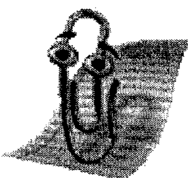


Figure 4.23. Animated character used by Microsoft

With the use of sound or an animated character, the user should always have the option of controlled the function. More importantly, the designer should always make sure that they are essential to the goal and they do not distract the user from their main task. As Susan Motte (1998) state, " if the software causes users to focus on the tool instead of their natural task-flow, the tool is not supporting their work".

4.3. UI Simplification and User's Persona

There is no doubt that reducing the visual complexity enhances the usability. However, this simplification shouldn't make the interface oversimplified and unattractive. Usable designs are not necessarily enjoyable to use. The surprise is that aesthetically pleasing objects enable the users to work better.(Norman, 2004) Many study studies show that users do not like "simple" screens (Waloszek, 2000). A system with minimal complexity is boring to look at. Sometimes users prefer more visually enjoyable than more capable devices. Color screen cell phones are one of the best examples. Color in phone screen does not make the phone more usable or efficient.

However, most of the people replace their mono-color phone, which is functioning very well, with a fancy color screen cell phone with different ring tons and screen images. Apple computer found that when it introduced the colorful iMac computer, sales boomed, even though those fancy cabinets contained the very same hardware and software as Apple's other models, ones that were not selling particularly well.

On the other hand, as we said, adding too many visual elements to the interface and creating a screen with maximum complexity is also not desirable as it can be visually confusing and less productive. Users want convenience; they want simplicity. So now the question is how simple an interface should be?

T. Comber and J. R. Maltby suggest that there is a trade off between usability (U) and complexity (C) with a relationship of the form $U = f(C)$ where U is a maximum for some intermediate value of C (Figure 4.24). Given a curve of the above shape, the optimum complexity will occur at the point $dU/dC = 0$ ie midway between the two extremes of complexity. It would be expected that for this optimum complexity an application would be easier to learn, quicker to use, have less user errors, and have the highest level of user satisfaction. (Comber and Maltby, 1995) Function of f(C) depends not only on the software's structural complexity, the platform and the environment user is working in, it also depends on the user's characteristics or persona.

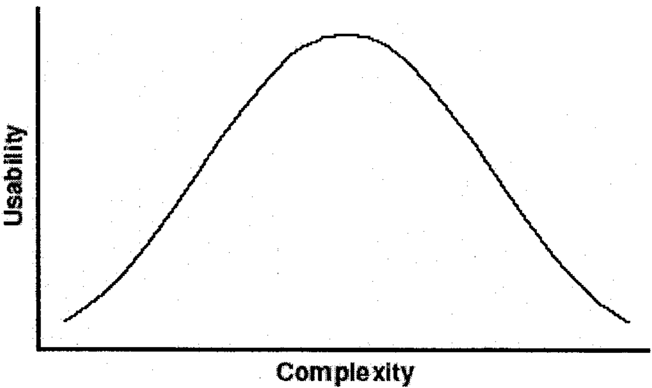


Figure 4.24. Relationship between complexity and Usability

The habit of a user's interaction, cultural background, age, and disability can effect the user's taste and mental model of a software product. Donald Norman states in his book "Emotional Design" that any design has three different aspects: visceral, behavioral, and reflective. Even though the impact of these three aspects is undeniable, it varies from one product to another and from one group of users to another.

Behavioral design deals with the effectiveness of use. Appearance is not very important. Of significance is the performance and functionality. In most behavioral design, functionality comes first, which is more important where security and safety are of concern. It is also of important for early adaptors and expert users, as they want a more powerful device with vast capabilities. They are not really concerned about the look and feel of the software (Norman, 2004)

Visceral design concerns itself with appearances. It is all about immediate emotional impact. It has to feel good and look good, which varies from user to user. The same thing that excites one group of users may disappoint others. For instance, Apple Computer has invested quite a bit on the appearance of the computer, which is appealing

to graphic designers and visual artists. However, the serious look of the PC is acceptable for people in engineering since they are after functionality rather than design. (Figure 4.25)

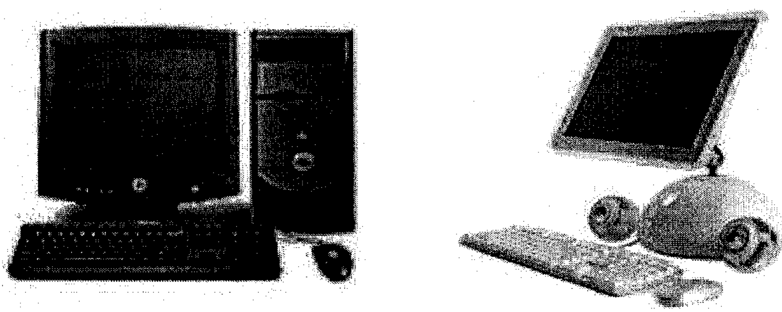


Figure 4.25. Design of Apple iMac version PC for 2 different groups of users

Finally, reflective design considers the rationalization and intellectualization of a product. Reflective design covers a broad territory. It is all about message, culture, the meaning of a product, or its use. Attractiveness is a visceral-level phenomenon; the response is related entirely to the surface look of an object. However, an unattractive object can become attractive, and software with fewer capabilities can be more acceptable when the product gives the message that the user is looking for. Some users use Netscape instead of Explorer not because Netscape is more useable or attractive, but mostly because it is a free product.

Quite often users utilize a product not just because of the design or even the usability factors but use it to associate themselves to a certain social group. This emotional feeling toward an object can change as the fashion changes. Yahoo Messenger versus MNS messenger is a good example as they are almost identical products offering the same service. For example, I once asked an 18-year-old programmer to transfer a file

via Yahoo Messenger. He said that he doesn't use Yahoo anymore because it is not "cool". It seems the older generation use Yahoo Messenger and MNS is more popular amongst teenagers as they think that using MSN connect them to the group of "cool people". (Waloszek, 2000)

With this in mind, the designer should consider the following questions during the interface simplification: what does the user need?; how simple does the user prefer? what message the product is giving to the end-users?; and, what is the best and simplest interaction design? Effective design cannot be achieved without understanding the user and considering the user's characteristics/ persona during the software development process.

Chapter 5: Task-sensitive CASE Tools

This chapter introduces task-sensitive approach to simplify and minimize user interface of CASE tools. It also provides an investigation on the effective use of UI metaphors to simplify the user interface. Finally it demonstrate a step-by-step approach to construct a task-sensitive metaphor for Microsoft Visual C++ program

5.1. Overview

As previously discussed, most of today's software, especially CASE tools, are function oriented. The interface usually is a collection of features that are offered immediately to the user regardless of the user's mental model and goal. In today's software market, there is competition between software development companies to produce software that has the highest number of features. Moreover, attempting to support several different methods and considering all types of users during the software design greatly adds to the number of features which in turn increases the complexity of the software. CASE tool users are not aware of most of the tool's functionalities, and use a relatively small amount of the features. (Spool Snyder1998). In the usability test which we conducted using Microsoft Visual C++, we realized that even expert users were unaware of approximately 40% of the features, and most of the users were utilizing only 30% of the functionalities offered by the software. (Seffah, 2002)

One way to reduce the software complexity is to reduce the structural complexity by reducing the number of features the software offers. This, however, can make the software less capable. A better solution may be to reduce the complexity of the software

by developing a task-sensitive interface. Stan Jarzabek (1998) believes that current CASE tools are too complex and not attractive enough to the users. He suggests that CASE tools should provide a natural process/task-oriented rather than method-oriented development.

5.2. Task-Oriented User Interface

Task-sensitive or task-orientation user interface is based on an analysis of the user's use of the program, and determine the functions and information that are required to accomplish that task using the program. The features that are not needed to do a specific task are hidden. By using task-oriented interface, we can reduce the number of features on the screen, which reduces the complexity of the user interface.

5.2.1. Wizard Dialog Box

One example of a task-sensitive interface is "Assistant" which is used in the Mac OS. or "Wizard" used in Windows.(Usability First) Wizard is a special type of dialog box that is designed to simplify what might otherwise be a more complex task. It accomplishes this by taking the user through a step-by-step procedure.

Wizards have been a part of workstation products since the early 1990s. A Wizard or Assistant is a task-oriented dialog that automates as much of a user task as possible. It typically has a "Next" and/or "Back" buttons for moving to the next or previous steps in the task, as well as a "Cancel" button to exit. (Figure 5.1)

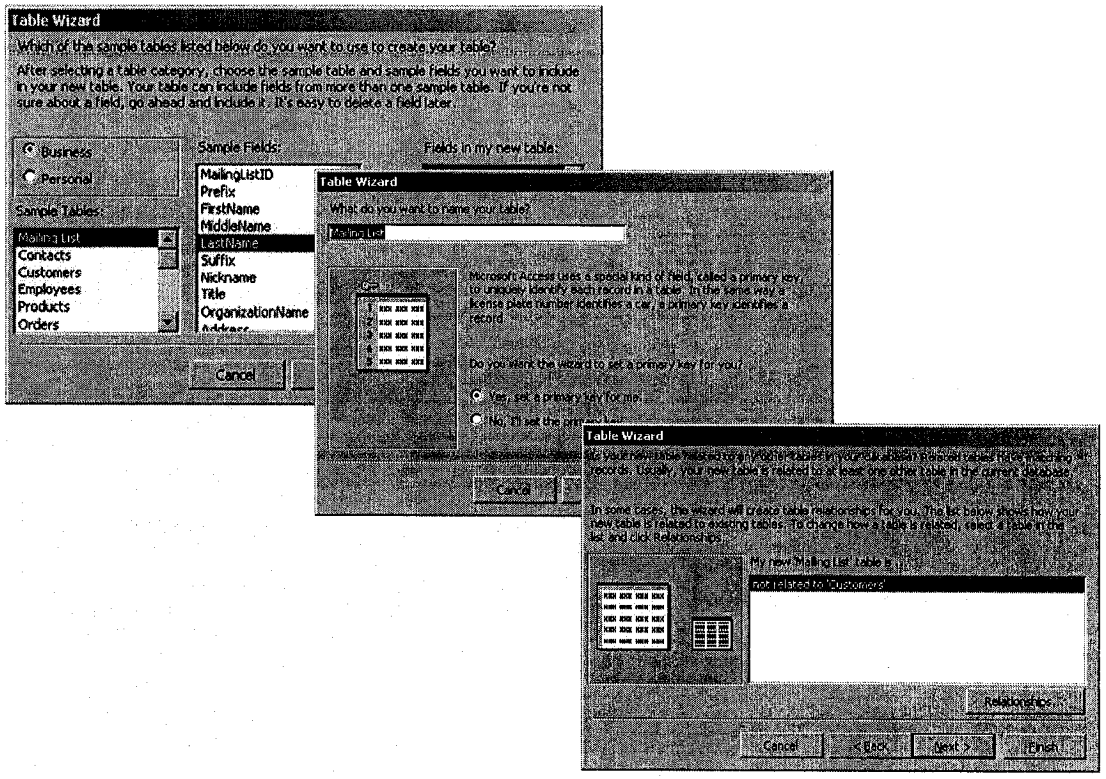


Figure 5.1. Wizard dialog box in Microsoft Access

Most wizards ask the user a minimum number of questions, then use the information gathered from the user to do a given task. Most wizards are used for on of the following purposes:

1. Installing software,
2. Entering large amounts of related data,
3. Creating complex objects, and
4. Performing complex procedures

Wizards are an effective way to help users, especially novice users, get their jobs accomplished. Wizards offer: a) *Simplification* when the steps for completing a task are

very complex; b) *Support* when expertise is required; and c) *Mental relief* when the user has to pay attention to a large number of associated factors.

Many developers are considering wizard-like enhancements to their product, however, their use is limited to a number of sub tasks and the entire user interface cannot be based on a wizard. (Tidwell and Fuccella 1997) First of all, wizard limits the user to do a task in only one way. Therefore in many cases, this restriction to the user's decision-making freedom may come at the expense of efficiency since the user has less control over the task. Moreover, as the user becomes more familiar with the system, the user doesn't want or need to use the wizard, since it is more time consuming and less flexible. (Spool and Snyder 1998) Therefore, the wizards can only be used as a *supplementary tool*, and should NOT replace the actual program itself.

5.2.2. Task Sensitive Toolbars

Another approach to create a task-sensitive user interface is changing the interface according to the task the user is trying to accomplish. An example, of this this approach is used in the property toolbox of Macromedia Flash program. As you can see in Figure 5.2 and 5.3, the property of items changes when the user chooses different objects on screen (in this case text and button). This approach makes the interface more understandable for the user by giving a number of choices related to a specific object the user selects. However, we cannot apply this approach to the whole interface. If you assume that the interface changes entirely whenever the user selects an object or a procedure; the interface will be very confusing for the user and will increase the user's memory load. In addition, determining what the user desires to accomplish from the user's interaction, the object the user selects, or the feature the user calls, is not easy as

quite often the user selects an object without having a specific goal in mind or can accomplish different tasks choosing an object.

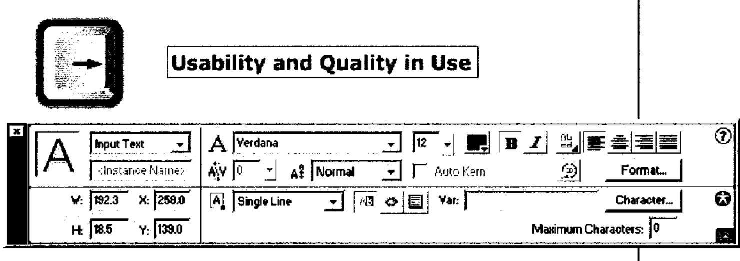


Figure 5.2. Property toolbar when the user selects Text

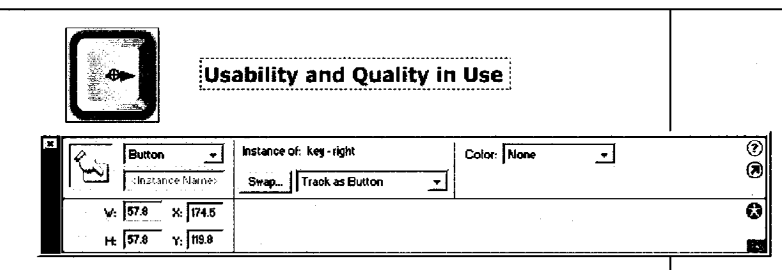


Figure 5.3. Property toolbar when the user selects Button

5.2.3. Task-Sensitive Interfaces

In our approach to create a task-sensitive user interface, we simply tried to divide the whole interface into the major tasks the user can perform, and show only the features related to that specific task. The first step to develop a task-sensitive user interface is to understand the users and the user's goals. The result are shown as a task model of the software using a task tree. In this model tasks are represented as hierarchical and/or trees of goals, as outlined in Figure 5.4 (Farrell and Breimer 2000).

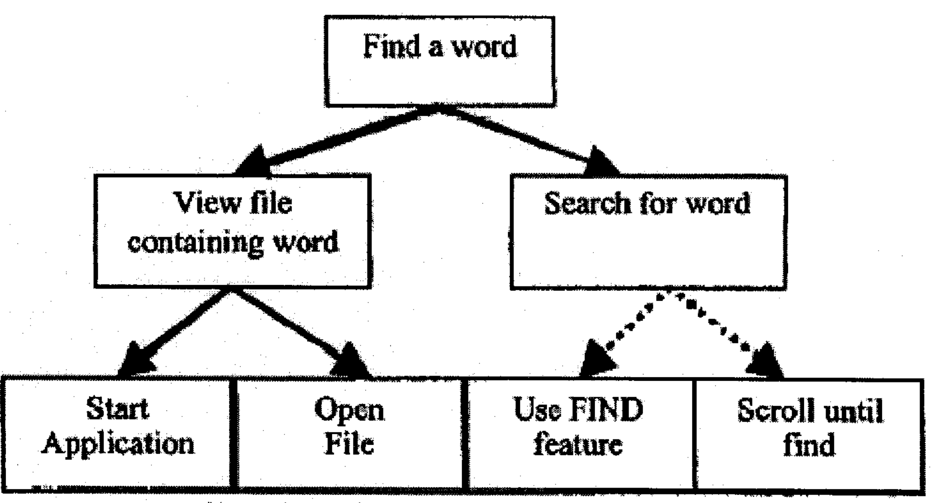


Figure 5.4. Task tree for finding a word in a document (Farrell and Breimer 2000)

The next step is to determine if there are any variations in accomplishing a specific task, and what are the specific features or steps required to accomplish the task? One task can be performed by using an alternative set of software features. In this case the designer should determine the group of features needed for all alternative ways. *(Paterno', 1996)* Finally the screen should be organized for each major task by grouping the related functions together.

In the next section, as a case study, I will demonstrate a step-by-step approach on how to reorganize the interface to create a task-sensitive user interface for Microsoft Visual C++.

5.3. Case Study: Task-Sensitive Interface for Microsoft Visual C++

The object of this case study is to minimize the user interface by showing only the features related to that specific task. The steps involved in this case study are: user task analyses and identifying the major task; identifying the functions required during each tasks; and sorting the functions related to each tasks. (Figure 5.5)

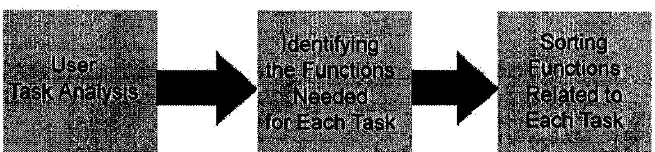


Figure 5.5. Major steps for creating task sensitive user interface

5.3.1. User Task Analysis

During our task analysis, we found out that the major tasks user can perform using Microsoft Visual C++ are:

a) Creating or opening a program,

b) Editing a program,

c) Compiling and running a program,

d) Debugging, and

e) Archiving (such as changing the name of the file, changing the directory or creating documentation for the program)

As you can see in the figure 5.6, the user always start by opening or creating the program, and then edits, compiles or debugs the program. Occasionally the user will also archive tasks such as changing a program setting or printing a file. All tasks except for

opening/creating are optional. However during our CW test we realized the users usually follow the sequence of editing and compiling, and if there is an error then debugging, and again editing and compiling until they are satisfied with the program.

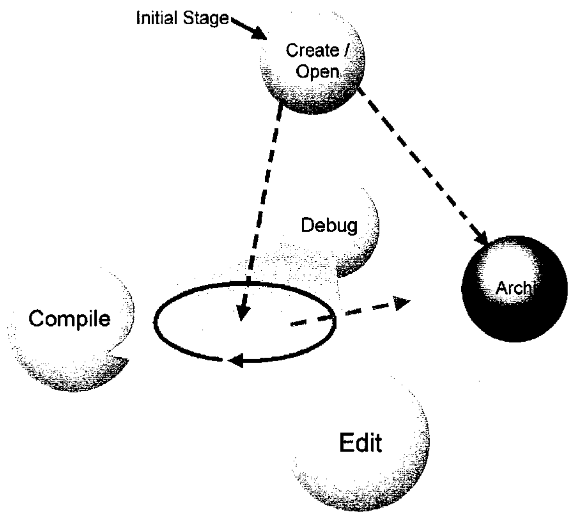


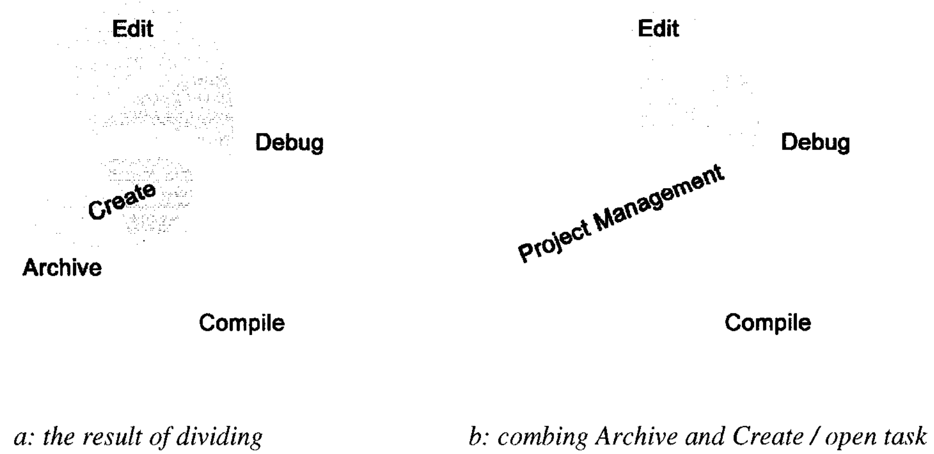
Figure 5.6. Task tree for major tasks in Microsoft Visual C++

5.3.2. Dividing Functions According to 5 Major Tasks

Apart from the main Microsoft C++ software, there are other Microsoft tools that developers can use during the software development process. However, in this research framework we considered only the main C++ program. During this step we went through the interface and wrote down all 120 functions on five separate 3" X 4" cards. We then prepared five boxes which were labeled Create, Edit, Compile, Debug, and Archive. We prepared one extra box for unknown functions.

We asked seven experts and three "occasional" users to put each function name to the relative task box that the function would be used. Users could put a function name into all five boxes if the function was needed in all major tasks, or put the card into the "Unknown" box if they were not familiar with the function. During the first attempt we

were able to divide 75 of the 120 functions. 90% of the users were not familiar with the remaining functions. Therefore, we divided the rest of the functions ourselves using Microsoft tutorial and the "help" option. Figure 5.7 (a) shows the result of this division.



a: the result of dividing *b: combing Archive and Create / open task*

Figure 5.7. Dividing functions in to 5 major tasks

As you can see in figure 5.7-a Create is a sub-task of Archive. Therefore, we combined Archive and Create tasks and identified it as "Project Management". The new task flow in the software development process starts with a file management task (to open or create a project). The user then goes to a loop of editing, compiling and debugging until they are satisfied with the project (Figure 5.8). During our CW usability test we realized that the user quite often came out of the loop to the Project Management task and went back to the loop again.

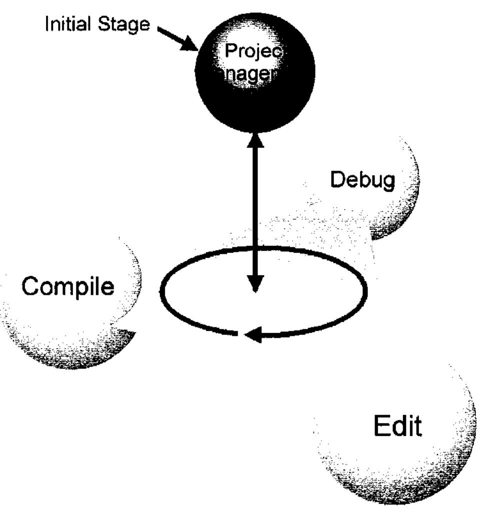


Figure 5.8. Microsoft Visual C++ task flow

5.3.3. Sorting Factions for Related to Each Task

Initially we divide the interface into four tasks: A) edit; B) debug; C) compile; and D) project management. As discussed previously, to reduce the users' memory load, the position of a function that can be used for two or more major tasks should remain the same while the user switches from one task to another. Hence, for each main task we divided the functions into several sets depending on its intersection with other main tasks. For example, for Task "A" (edit) as you can see in figure 5.9 we have the fallowing sets

{

$A - (B \cup C \cup D)$

$A \cap B - [(A \cap B \cap C) \cup (A \cap B \cap D)]$

$A \cap D - [(A \cap B \cap D) \cup (A \cap C \cap D)] = \varnothing$

$A \cap B \cap C - (A \cap B \cap C \cap D)$

$A \cap B \cap D - (A \cap B \cap C \cap D) = \varnothing$

$A \cap C \cap D - (A \cap B \cap C \cap D) = \varnothing$

$A \cap B \cap C \cap D$

}

Therefore we can divide the interface of task A into 4 set of:

{

$A - (B \cup C \cup D)$

$(A \cap B) - (C \cup D)$

$A \cap B \cap C - (A \cap B \cap C \cap D)$

$A \cap B \cap C \cap D$

}

You can see the sorting for other tasks in Table 5.1.

A: Edit	$A - (B \cup C \cup D)$ $(A \cap B) - (C \cup D)$ $A \cap B \cap C - (A \cap B \cap C \cap D)$ $A \cap B \cap C \cap D$
B: Debug	$B - (A \cup C \cup D)$ $(A \cap B) - (C \cup D)$ $(C \cap B) - (A \cup D)$ $A \cap B \cap C - (A \cap B \cap C \cap D)$ $A \cap B \cap C \cap D$
C: Compile	$C - (B \cup A \cup D)$ $(C \cap B) - (A \cup D)$ $A \cap B \cap C - (A \cap B \cap C \cap D)$ $A \cap B \cap C \cap D$
D: Project Management	$D - (A \cup B \cup C)$ $A \cap B \cap C \cap D$

Table 5-1. Sorting functions for 4 major tasks

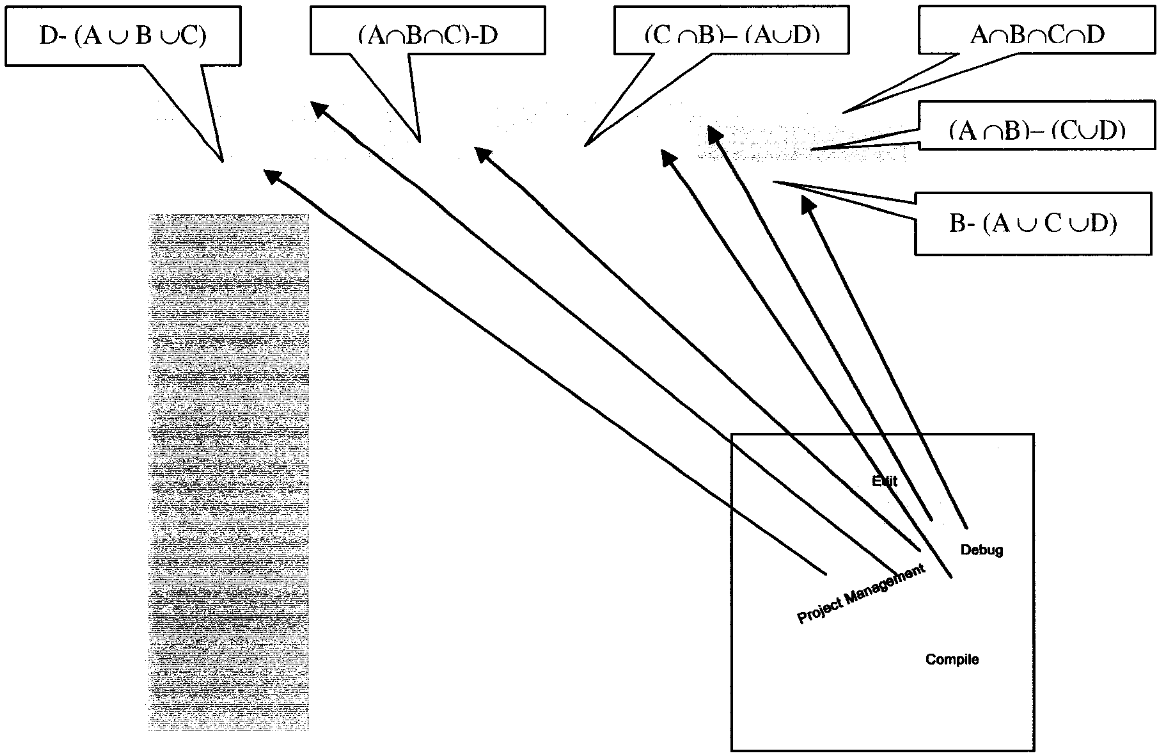


Figure 5.9. Mapping functions into the interface for the task B (debug)

Please note that even though we separate the position of sets of (A ∩B)– (C∪D), (C ∩B)– (A∪D), B- (A ∪ C ∪D) and (A∩B∩C)-D on screen, the user doesn't need to recognize these different sets. User only need to distinguish the functions related to task B from those for task D and those that can be used for both task D and A, since we are offering Task B and D at the same time (Figure 5.10)

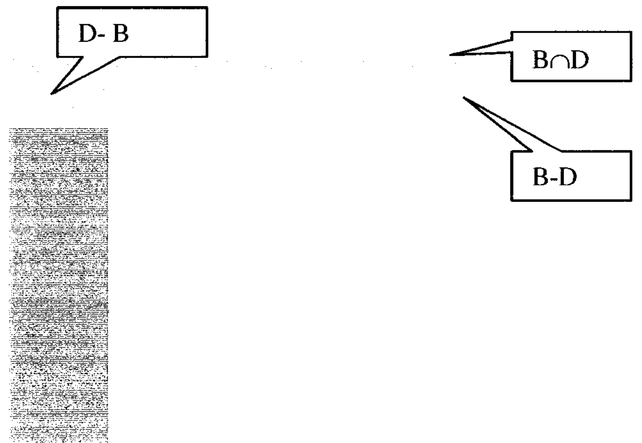


Figure 5.10. User's view of functions sorting for Task B (Debug)

Finally, in order to reduce the number of functions in the interface, we sorted the functions to several sub sets using EZSort (beta release). We asked seven users to sort each set of functions to different groups and name the sorted sets. Figures 5.11 and 5.12 show sorting for intersection between Debug, Edit and Compile, and table 5.2 shows the final result of our sorting. In the next session we will discus how to demonstrate this functions to the user by developing Task-sensitive Metaphor.

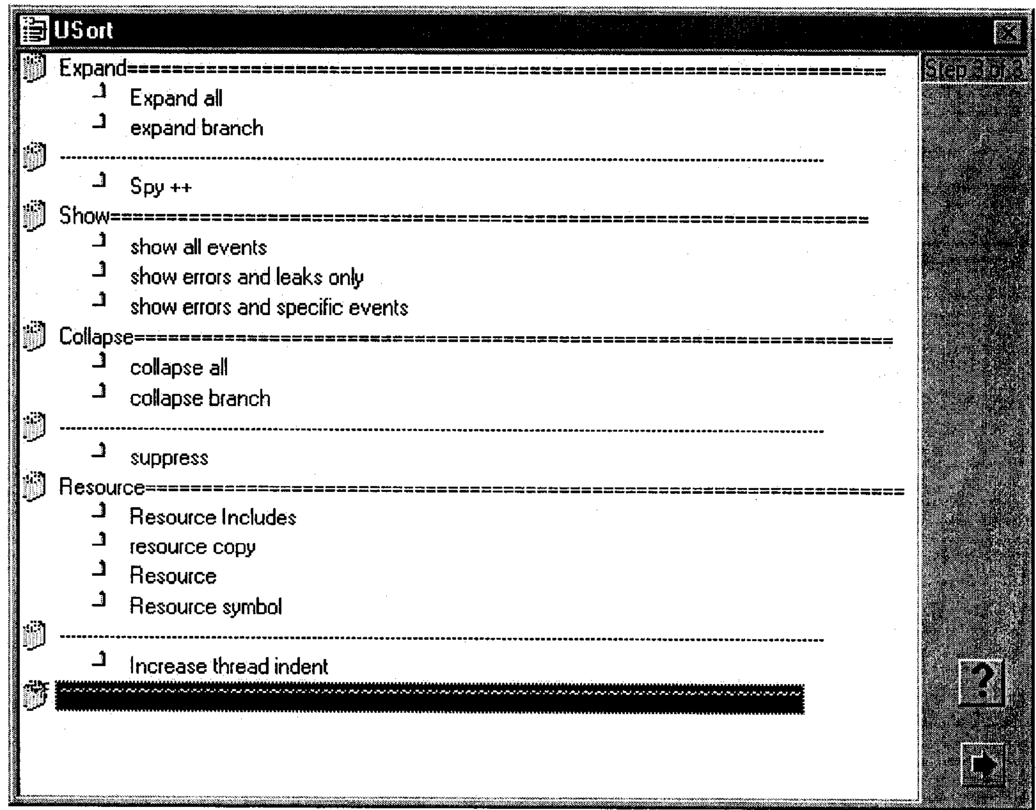


Figure 5.11. Sorting for Debug-Edit-Compile (A∩B∩C)-D) using USort

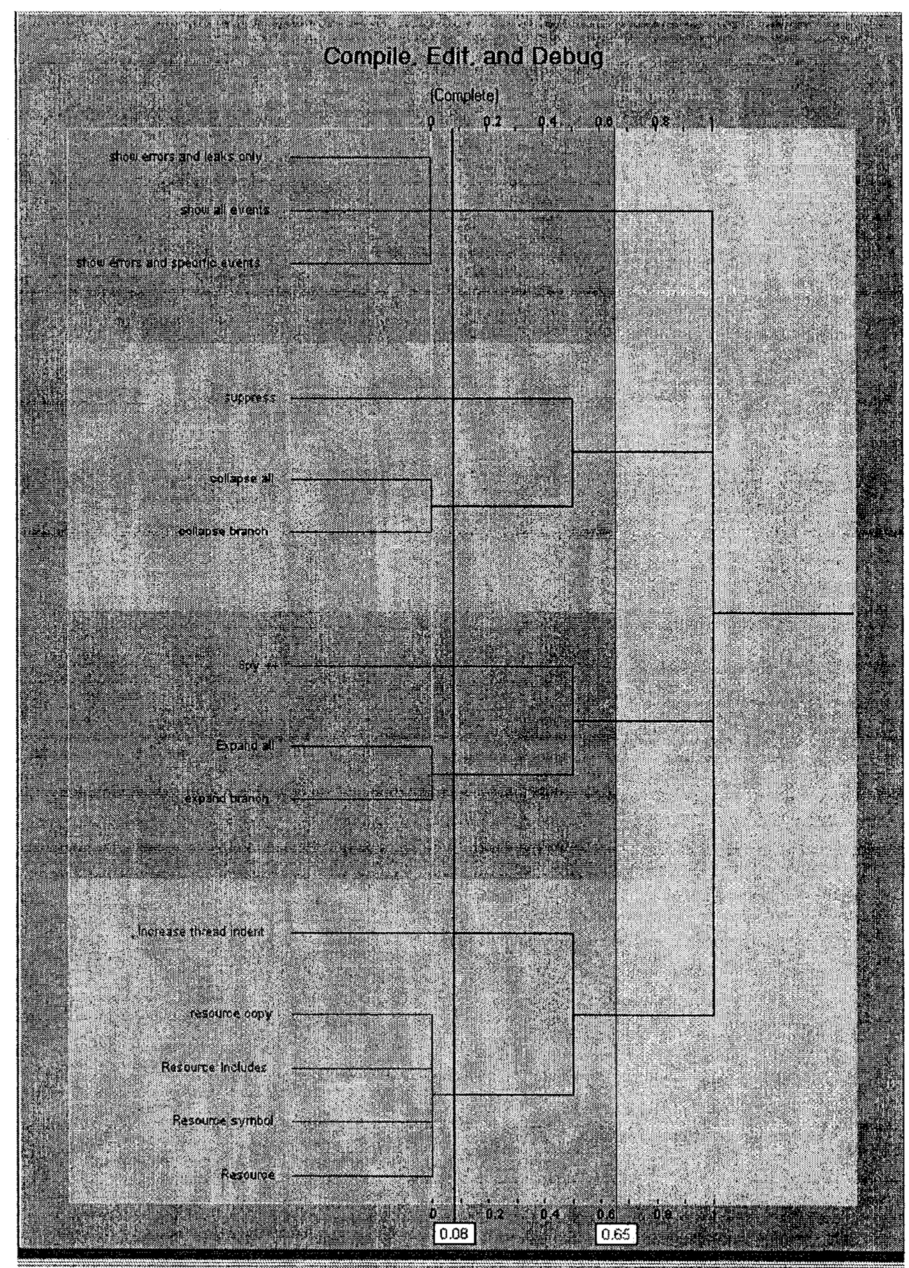


Figure 5.12. Result of sorting of for (Debug-Edit-Compile)

Debug	1. Active control X error container, 2. Debugger remote connection, 3. Break points, 4. Error (Find next error, find previous error, Find All Errors, error look up,) 5. Debug(start debugging, Integrated debugging,) 6. Report (Report errors and events, Report errors immediate)
Debug-compile	1. Allocated memory, 2. Batch build with bound checker, 3. Setting, 4. Parameter info 5. Show (show errors and leaks only, show all events, show errors and specific events)
Debug-edit	1. Arguments 2. Register control
Edit	1. New alt abject 2. Complete word, 3. Micro (options micro, play quick macro, smart quick macro) 4. List members,
Compile	1. Build (Build, Build all, Build with bound checker) 2. Compile, 3. Compliance report, 4. Output, 5. Run, 6. Link, 7. Sequence numbers,
Debug-Edit-Compile	1. Program file (save, Open to edit, new Cpp program, new class, Include to the project, Close,) 2. Properties 3. Resource (resource copy, Resource Includes, Resource symbol, Resource,) 4. Spy ++, 5. Collapse (collapse all, collapse branch,) 6. Expand (Expand all, expand branch) 7. Suppress, 8. Thread indent, (Decrease thread indent, Increase thread indent)
Project Management	1. Add to project (new, new folder, files new class, new form), 2. Work space (close a work space, open workspace, save a work space, recent workspace,) 3. Events (Event summary, Find next event instance), 4. Project (Open a project, Close a project, setting, type info, version information, Create a new Project, Set active project, save, insert project into workspace,)
All	1. Bookmark 2. Close, Minimize, Maximize 3. Find, 4. Edit (Cut, Delete, Copy, Paste) 5. Print 6. Help, 7. Go (go, to source, OLE/Com object viewer mfc tracer,) 8. Tools

Table 5-2. Final result of function sorting

5.4. *Task sensitive metaphor: a solution to complex CASE tools*

5.4.1. Overview on UI Metaphors

User-interface metaphor is the process of representing the computer system with objects and events from a non-computer domain". (Merriam-Webster) Metaphors play an essential role in creating a world, an atmosphere, and/or a context in which the action occurs. The *conceptual model* is the overall view of the system or part of the system, and a metaphor is a mapping relation between aspects of the conceptual model and the world at large. (Rauterberg and Hof, 1995)

The idea of using metaphors has been around for as long as computers have existed. Before the appearance of the Graphical User Interface (GUI) and the invention of the mouse, the basic metaphor for human-computer interaction was "a figure of speech" or commended language in which a word, denoting one subject or idea, is used in the context of another. (Drakos and Moore 1999, Wilson, Rauch, and Paige 1992) Computer interfaces started using graphical metaphors by inventing GUI. The "desktop" metaphor was popularized by Xerox for GUI in Apple Computers and contains office references (desk top, documents, folders) mixed with building references (windows, trash cans). Some examples of metaphors used in GUI are:

- The mouse is a hand (and therefore, the cursor is a finger).
- Icons are Objects (and similarly, words, ranges of text, windows and other graphical entities are objects).
- An object on the screen is an object in storage (that is, it corresponds to a file or a stored data).
- A folder is a container and folders are objects.

- Amount of substance is amount of time. We can see this metaphor in "progress bars" (Brockerhoff, 2000)

New metaphorical references and enrichments of the existing references are occurring all the time. Using metaphors in user interfaces enable users to comprehend, use, and remember information more quickly, with greater ease, and with deeper satisfaction. (Drakos and Moore 1999, Brockerhoff, 2000) However, recently, there has been concerns over the fact whether metaphors really achieve the benefits commonly claimed, and how we can use metaphors effectively (Drakos and Moore 1999, Marcus, A., 1997)

In this section we offer an overview on UI metaphors and the related usability problems. At the conclusion of this section we will introduce task-sensitive metaphors as an effective way to reduce UI complexity.

- Metaphors' Categories

We can classify UI metaphors in to the following categories:

i. **Verbal and Visual Metaphors:** Verbal metaphors usually represent an action related to the system. Typical examples of verbal metaphors include:

   - Move (purposeful traversal): navigate, drive, fly
   - Browse (low goal-oriented review of options): window shopping, thumbing through books,
   - Scan (very rapid browsing): fast review of scrollable items, fast review of buildings, objects, people, billboards on highway at high speed
   - Locate: to point, touch, encircle item(s)
   - Select: to touch, poke, grab, lasso,,place finger, and/or slide

- Create: add (new), copy
- Delete: throw away, destroy, lose, recycle, shred. Delete (temporary or permanent) sometimes consists of dragging a file icon to a trash can.
- Visual Metaphors are physical objects used to communicate with the computer, applications, documents, and data. Some examples of this kind of metaphors are:

- Desk: Drawers, files, folders, papers, paper clips, stick-on note sheets
- Document: Books, chapters, bookmarks, figures; newspapers, sections, magazines, articles, newsletters, forms
- Photography: Albums, photos, photo brackets/holders
- Containers: Shelves, boxes, compartments
- Tree: Roots, trunk, branches, leaves
- Network, diagram, map: nodes, links, landmarks, regions, labels, base (background), legend
- Cities: Regions, landmarks, pathways, buildings, rooms, windows, desks (Marcus, A., 1997, Brockerhoff, 2000)

ii. **Literal and Explicit Metaphors**. Metaphors may be either literal or explicit. A metaphor is literal if the actual shaper is presented on screen, such as file icon in windows. Explicit metaphor is used as conceptual shapers of design without any direct representation on the interface. For example, no actual desktop is literally presented on the screen, although the concept of the desktop is used. (Constantine, 1998)

iii. **Novel and Conventional Metaphors:** Conventional metaphors are those that are already used by the target audience, and the group of users understands the structure of the metaphor and the way it works. A novel metaphor is any metaphor that is not considered conventional for a group. Obviously this classification is dependent on how the group of relevant people is defined. Some metaphors that are conventional to one group may be novel to another. For example, ejecting a CD by tossing to the recycle bin is conventional for Apple users but novel to most of the PC users. A good novel metaphor will eventually become a conventional metaphor as it is absorbed into a group's way of thinking. (Drakos and Moore 1999)

iv. **Process and Element Metaphors:** Process metaphors explain some matter of system functionality by comparing it to a real-world process. This enables the users to take their knowledge of how to perform a real-world task or process and apply it to the interface without specific training. An element metaphor is used to inform the user which process metaphors are active in the form of a sound, graphic or text, such as error bib. (Constantine, 1998)

v. **Composite interface metaphors:** composite metaphors are a combination of different types of metaphors representing multiple mental models. The potential problem with composite metaphors occurs when the conceptual models mismatched between two metaphors.

Other Categories of metaphors are:

i. **Orientational Metaphors:** Lakoff and Johnson (1999) characterize orientational metaphors as metaphors that ``give a concept of direction in terms of space". For example, metaphor GOOD and IMPORTANT are up. Orientational metaphors are strongly based in our physical and cultural experiences, and they are not generally universal. For example, in the western language a navigation metaphor for "Progress" is usually to the right. Therefore when we move through a task wizard, the `next' button often points to the right. This sort of metaphor probably comes from our experience of reading, and therefore is not the same for people in Arabic country since they read from right to the left.

ii. **Ontological Metaphors:** Ontological metaphors are basic concepts of our existence, such as cause and effect. For example, we tell the user that ``an error prevented the system from opening that file", thus the metaphor ERRORS are entities capable of causing and preventing things. (Drakos and Moore 1999)

iii. **Structural Metaphors**: Structural metaphors characterize the structure of one concept, ``signifier", by comparing it to the structure of another concept, ``signified". The benefit is that by having experience with the metaphoric concept, the system structure becomes more immediately apparent. For example, in the desktop metaphor, the desktop is the signifier and the file-system is the signified. By using metaphors based on already understood concepts and objects related to a workspace, the filing system becomes clearer to the user.

iv. **Metonymies:** A Metonym is when we use a part of or an identity of a concept to refer to another that is related to it. For example, we show ``the crown" when we

mean the queen. In the interface we usually use metonymies in icons. An example of this would be the "magnifying glass" icon for searching though the system files. The magnifying glass icon refers to looking closely and exploring the data in a file system. Our choice of icons, therefore, has an effect on how the user perceives the thing referred to. (Lakoff, G., and M. Johnson 1999)

5.4.2. Complexity of User Interface Related to UI Metaphors

Even though using metaphors in user interface enhances the user's understanding of the system, most metaphors fail to improve usability and many make matters even worse. (Constantine, 1998) In this section we will study how metaphors can cause misunderstandings about the capabilities of the software as the user uses a metaphor to interconnect their mental model with a system.

a. Mismatching Between Target Domain and Source Domain

Some problems related to a metaphor can occur when the metaphor "source" domain is employed in ways that doesn't exactly matches system "target" domain. In general, the three basic problems with target domain and source domain are:

i. The target domain has features not in the source domain. For example, telling the user that your computer is like a desktop wouldn't lead the user to look for a scroll bar.

ii. The source domain has features that are not in the target domain. Users can put any kind of document in an office space and read it, but a user trying to do the same thing on current computer systems will normally fail.

iii. Some features exist in both domains but work very differently. Therefore, users may have trouble seeing beyond the metaphor to use the system in the

ways it was intended. For example, instead of using copy machine in an office to produce a copy of a document, the user should save the file as another name. Or instead of advancing the paper when using a typewriter to proceed to the next line, we simple need to press enter on the computer. (Wilson, Rauch, and Paige 1992, Norman 2000)

b. Overly Literal Translation of Real Object

Some of the worst problems that can make the interface even more complex for the user usually arise when real-world objects and their behaviors are simplistically simulated on screen, whether as structural or functional metaphors. Objects that are simple and straightforward to understand and use in the real world can become sources of confusion when translated onto GUI.

Even if the target domain and the source domain are exact matches, presenting a software artifact as a direct analog to a physical object can impose unnecessary conceptual restrictions on the design. For example most calculators simply replicate the difficult to manipulate design of existing real calculators. True, the user may immediately recognize the object pictured on-screen, but it does not necessarily mean that they will know how to use it. The calculator in figure 5.13 goes so far as to use shift to switch between functions. The purpose of this feature in the analog design is only economizing space, which is not necessary in GUI. A less similar to the real object and more adapted for GUI would be more useable and simpler. (Mullet, K. and Sano, D. 1995)

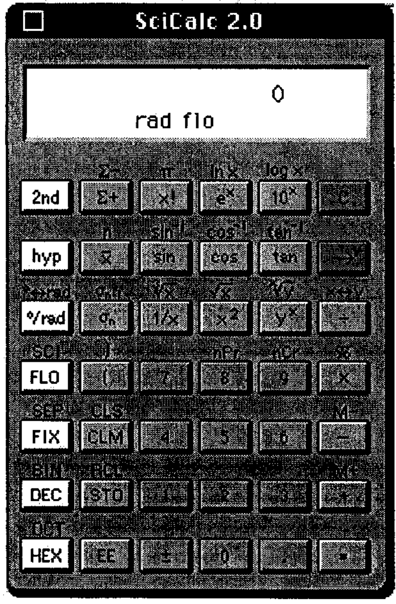


Figure 5.13. Overly literal translation of a real object

Metaphors chosen for interface must also be easy to represent. It is important that distinctive images, which are representative of the desired functionality, can be created in a limited presentation area. Over-detailed metaphors can be too restrictive and less effective. (Rauterberg and Hof, 1995) An example of this kind of misuse is simulating a ticket counter, as depicted in Figure 5.14 In this design, even checking schedules or frequent flyer points is made more difficult by sidewise labeling which is half obscured by Lucite. The user has to scan-and-pause with the mouse pointer to know what is available. (Constantine, 1998,Norman 2000)

Figure 5.14. Poor real object UI metaphor

Some interaction methods used in UI metaphors are not as natural when they apply for GUI, which makes the interface difficult to use. For example in Figure 5.15, turning the volume button is not easy especially for laptop users.

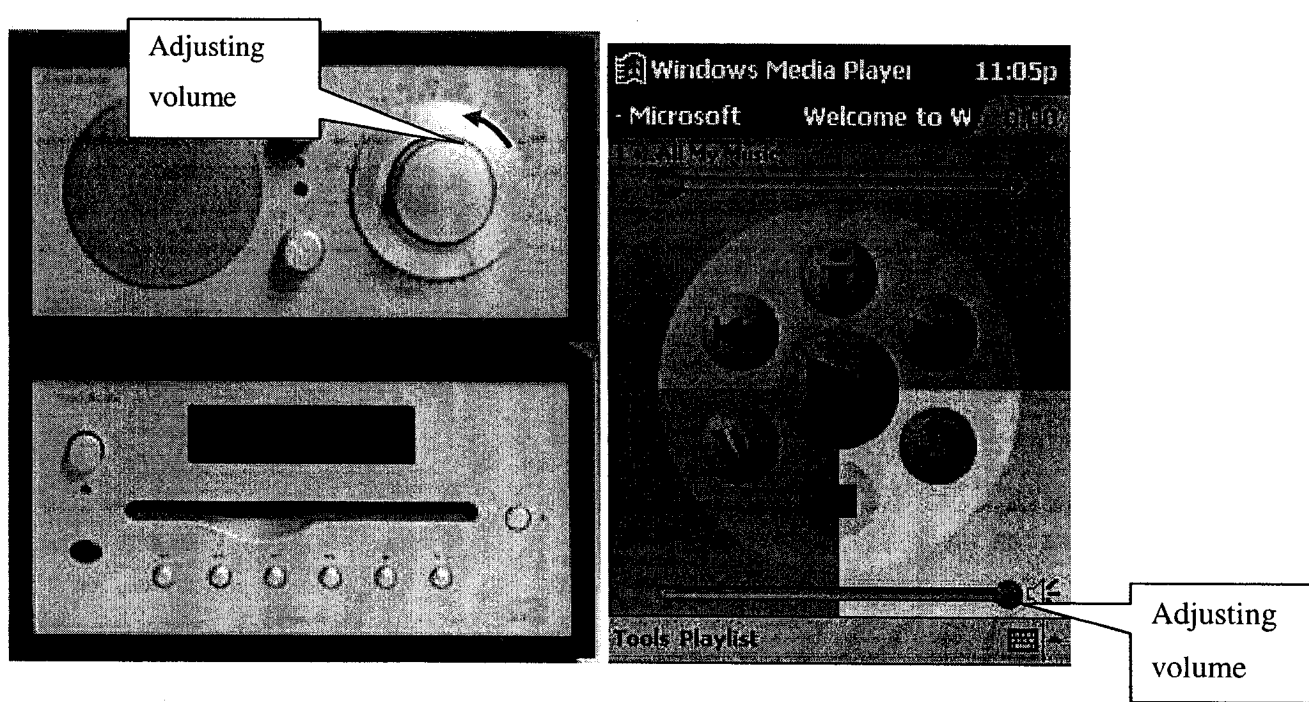


Figure 5.15. Unnatural way of interaction *Figure 5.16. Natural way of interaction*

UI Metaphors do not have to be complete but interfaces need to provide adequate clues on how to use a system. Real object metaphors should be adapted for the interface, having the same or similar relationships in the target domain as it did in the source.

5.4.3. Metaphor and User's Persona

The Macintosh allows users to drag the floppy disk to the wastebasket ("Trash") in order to eject the floppy. In real life, trashing an object means the object is not usable anymore; however when the user ejects a floppy, they may use it later.

In other cases, even though the domain knowledge used in metaphor is almost the same software domain, the metaphor does not reflect the software capabilities, which causes the user not to trust the software. Masahiro Mori, a Japanese robotics company, has argued that we are less accepting creatures that look very human, but do not have the capability of a human. (Norman, 2004) Therefore having an agent who looks like Albert Einstein give the user recommendations can be problematic. For most people Albert Einstein is one of the most intelligent characters on the earth and is considered someone who never makes a mistake; where as the agent's recommendations may not always be right. Therefore after a while the user looses their trust in the software. In this case using a puppy is more suitable. A puppy can make a mistake, but we usually forgive it since it is just a puppy. (Figure 5.17)

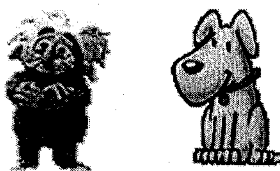


Figure 5.17. Choosing Einstein verses a puppy as a metaphor for an intelligent agent

This problem also arises when trying to develop interfaces, which is internationally acceptable (Nielson, 1990). It can often be extremely difficult to find a metaphor that is understandable and acceptable across a number of different cultures and groups of users.

An example of this problem is using the "stairs illusion" in order to show improvement in a process. When individuals from some cultures view a line drawing of steps they actually see it as a three dimensional representation of a step since they have the experience. However, individuals from cultures where stairs are not used will see the drawing as a simple two-dimensional design; they do not interpret it as "stairs".

Finally a metaphor should be enjoyable and engaging for the user. Users usually are looking for fun and pleasure. If the user does not find a metaphor engaging enough they may stop using the software. Professor Ishii in Spain suggests having colorful pinwheels spinning overhead to give some information such as the stock market. The higher rate of speed indicates the urgency of the information. (Norman, 2004) Of course we can show the information by text or numbers however it is not attractive enough for the users and will not have the same effect.

The engaging aspect of a metaphor varies from one group of users to another, and amongst different cultures. For example, in Europe people like cute puppies and keep them as pets. However, people in Eastern countries never keep a puppy as a pet. Even though they know what a puppy is, they don't find it engaging to represent an agent.

The best approach for developing a metaphor is to collaborate with prospective users. A metaphor, after all, is a tool to link your product to the user's mental model. We

cannot develop an effective metaphor without considering the user's characteristics or user's persona (Brockerhoff, 2000 Wilson, Rauch, and Paige 1992)

5.4.4. Task Sensitive Metaphors

Metaphors are tools to facilitate interface design and end-user interaction with an application. Despite their usefulness, with the structural complexity of many modern applications that support several tasks, a single metaphor may not support all the facilities that are needed. As the user uses a metaphor to mesh their mental model with an application, the metaphor may lead to misunderstandings when it does not cover the entire task in the software domain. For example, the "desktop" metaphor was developed around the user's tasks of editing and creating information. This metaphor was very simple for the users trying to prepare or publish a book, and was accepted by most of the users. Considering the level of success related to desktop metaphor, software designers apply it to all software with graphical interface. No matter what kind of task the software is supporting all applications in Windows have the same look and feel using the same set of "windows" metaphors. (Figure 5.18)

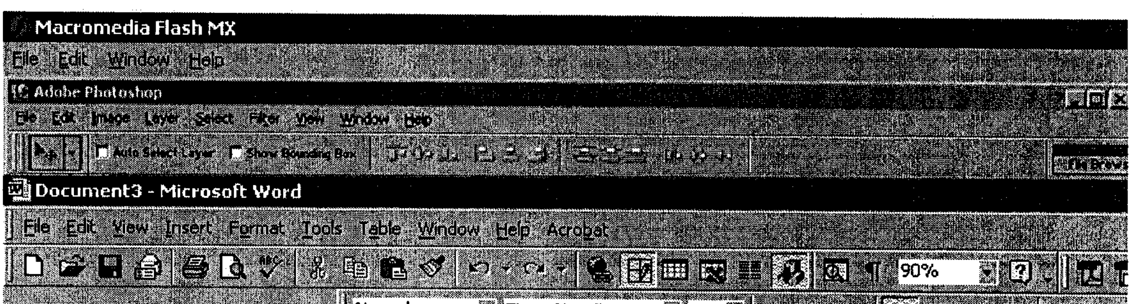


Figure 5.18. Using the same set of labels and metaphor in windows' applications

This metaphor, however, cannot be suitable for all application covering all tasks. We cannot really define programming with a desktop metaphor as there are aspects of the

web and browsers that inherently change the rules from the desktop world. In such circumstances, multiple or composite metaphors can be used (Richards, 1993). The potential problem with composite metaphors occurs when the conceptual models mismatch between two metaphors, which makes the interface more complex for the user. Overuse of different unrelated metaphors can lead to complicated interfaces as a result of introducing too much conceptual and cognitive complexity.

A proposed solution to this problem is developing a task-sensitive metaphor that can cover an application's task domain. We can achieve this through task analyzes and mapping each tasks/ sub-task to an appropriate metaphor.

In the next section we performed a case to attempt to develop a task-sensitive metaphor for Microsoft Visual C++ interface.

5.5. Case Study: Developing a Task-Sensitive Metaphor for Microsoft Visual C++

As shown in Figure 5.19, this case study consists of five main steps:

a) Task analysis of the problem domain;

b) Finding an appropriate metaphor that covers all tasks in the problem domain;

c) Checking the candidate metaphor with the user's characteristics in the set of persona;

d) Mapping the metaphor attributes to the domain tasks;

e) Testing with the users.

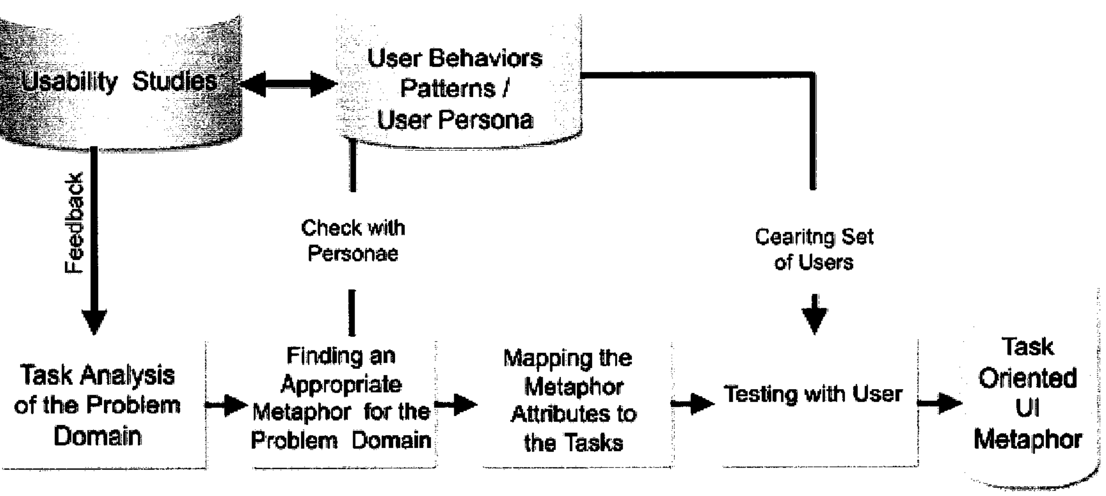


Figure 5.19. The methodology for developing Task-Sensitive metaphor

5.5.1. Task Analysis

From our previous study we recognized four major tasks using the Microsoft C++ program. These include: a) project management including archive and create/open; b) edit; c) compile and run; and d) debug.

5.5.2. Finding an Appropriate Metaphor that Covers all Tasks in Problem Domain

First we examined to see if the desktop metaphor could cover the entire tasks in our domain. If the desktop can cover all tasks in our problem domain, we do not need to develop a different metaphor since the users are already familiar with desktop. However, as you can see in Table 5-3, only editing and partially archived tasks can be mapped to the desktop metaphor.

Microsoft C++ main tasks	Related Desktop Metaphor
Project Manager	Partly covered: There is no definition of program object, or executable file in desktop metaphor
Editing	Partly covered: the concept of editing for a program is different from editing a document. Texts in a program are not just some documentation, they are commands or program objects that can be compiled and executed
Compile	Not covered: we cannot compile or execute a document
Debug	Not covered: finding an error is not defined in the desktop metaphor. Even if we can define syntax errors, logical errors cannot not be justified

Table 5-3. Microsoft C++ main tasks and Desktop Metaphor

Therefore, together with two-interface designer, we wrote down the elements of a familiar place or activity that could be used as a candidate metaphor. We also wrote down the related object and process of each candidate:

- **Spreadsheet:** row, column, calculation, cell, scripts run on multiple cells, graphical reports, templates, accounting, cost estimation, etc.
- **Road:** map roads, legend, exit, street names, landmarks, trip planner, one-way streets, detours, tolls, warehouse, shelves, fork lift, access method, **etc.**
- **Cafeteria:** trays, stations, take-out, place where you get utensils, place where you pay, automat, coffee refills, condiments, etc.
- **Building:** windows, hallways, heating system, doorway, elevators, room numbers, addresses, organization by floors and corridors, vending machines, electricity, power, distinctive interior decoration, **etc.**
- **Play God:** Create, world, creatures, recreating, evolution, angels, evil, etc.
- **Cooking:** oven, cooking utensils, ingredients, recipe, tasting, seasonings, etc.
- **Industrial City:** factories, roads, construction, productions, installation, map, etc.

We then chose three candidates that could cover the major tasks in our problem domain:

i. **Play God:** The computer screen is a world. This is the first and most obvious metaphor for GUIs. The user is put quite naturally in the position of a benevolent God looking down on a toy world which he can manipulate at will. The metaphors for our four tasks are:

- Project management: create a world by first defining a model of the program
- Editing: (Evolution)
- Compiling: designing creatures
- Debugging: detecting evil creatures and destroying them

ii. **Cooking:** User is a chef. The user can choose or create a suitable recipe (like a suitable programming project) prepare the ingredients, and cook it. The user can also taste the food to see if there is anything wrong and fix it or change the recipe. Therefore the metaphors for our 4 tasks are:

- Project management: choosing or writing recipes is like opening or designing a programming project
- Editing: preparing ingredients
- Compiling: putting the food into oven and cook it
- Debugging: tasting the food and fix it if it is possible or change the recipe and cook it again

iii. **Industrial City:** The user is the manager of an industrial city. They can design the city, create different factories, use the product from one factory in another, and test the final product. The metaphors for our four tasks are:

- Project management: Creating an industrial city and managing the city;
- Editing: Creating different factories, defining and writing what the process used in each factory should do;
- Compiling: assembling the final product;
- Debugging: testing the product to see if there is some mistake, and fixing the errors accordingly.

5.5.3. Checking with the User's Characteristics in the Set of Persona

During this phase we examined our three candidates' metaphors to see if they are compatible with the characteristic of the users in the set of personae for Microsoft C++ software (Table 3-3). The results of this study revealed that the Industrial city metaphor is more suitable with our set of personae.

God is not a suitable metaphor for the user, especially for Charles Butler our occasional user. The God metaphor is an expert that knows everything and does not need any help; where as there are novice and occasional users, who need help occasionally, and they make mistakes. Moreover the world is too general and has no limitation, but software has. Therefore creating a world is not a good metaphor for Microsoft Visual C++.

Cooking could be a good metaphor if our users enjoyed cooking. However, neither John nor Charles ever cooks. John always grabs a bite on his way to school, and Charles's wife usually prepares the food for him or he dines out.

Therefore, we considered industrial city as a metaphor for the new user interface. After all programming is a part of software engineering, which is not so far from other engineering fields.

5.5.4. Mapping the Metaphor Attributes to the Domain Tasks

During this phase we mapped the metaphor attributes to the tasks and sub-tasks, and then attempted to visualize the metaphor for each tasks.

i. **Project management:**

Figure 5.20. Icon representing Project management task

In our metaphor, the user is the head of an industrial city. His job is first specifying the kind of project or the type of industrial city he wants to work on. Each industrial city has several factories. The factories represent a C++ program file. Some factories produce a product that could be used in other factories. The products are C++ objects that can be used in other C++ files. In C++ programming using a C++ object or function in another program, we needed to link programs together. In an industrial city for using one product in another factory, we needed to build a road between them. Therefore, roads in the city represent links between factories. When visiting any city, people need to know where they are and what is happening in the city. Working on a programming project, the user needs to have an overview of the project. The user also needs to see the events related to the project. This is the main reason why we need to have a project management task and the program map available during all tasks.

Edit Print Find Go Bookmark Tools Help

Program file Properties Resource Spy ++ Collapse Expand Thread Indent Arguments Register Control

New alt abject Complete word Micro List members

Project Work Space Events

Add to Project CPP h

Main.cpp

Tool bar for the task edit

During the design, user can drag any kind of program file and drop it in to the project.

```
//-----------------------------------------------------------------
// Project:Elevator
//
//
// Purpose:To simulate the action of an elevator
//
// File:Main program
// Compiler:Visual C++
//
// Programmer:BM
//
// Date:November 5, 1999
//
//-----------------------------------------------------------------

#include "stdafx.h"
#include "Elevator.h"


void main(void)
{
   const int number = 15;
Elevator Otis(number);
Otis.prompt();
}
```

CPP Button h Panel CPP Panel h stdio h Elevator CPP Elevator h stdafx CPP main

Project map. The yellow construction signs indicate the user is working in those sections. The red arrow shows the position of the user

Debug Outputs

Figure 5.21. Microsoft C++ interface (Edit and File Management)

ii. **Edit:**

Figure 5.22. Edit Icon

Editing a program is the same as working in a factory. Each factory has some process (factions) and the end product. In a city the user can go to different factories. When the user is working on a program, there will be an "under the construction" sign beside the factory on the project map, and an arrow always indicates where the user is.

iii. Compile:

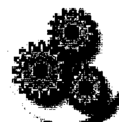


Figure 5.23. Compile Icon

We chose a real world metaphor, a compiler machine, for this task. The user can drag and drop a file, or a series of files to the machine. With animation, the machine shows the user the process of compiling. The executable files and the errors will be the end process of the compilation. The user can then run the executable file, or see the errors.

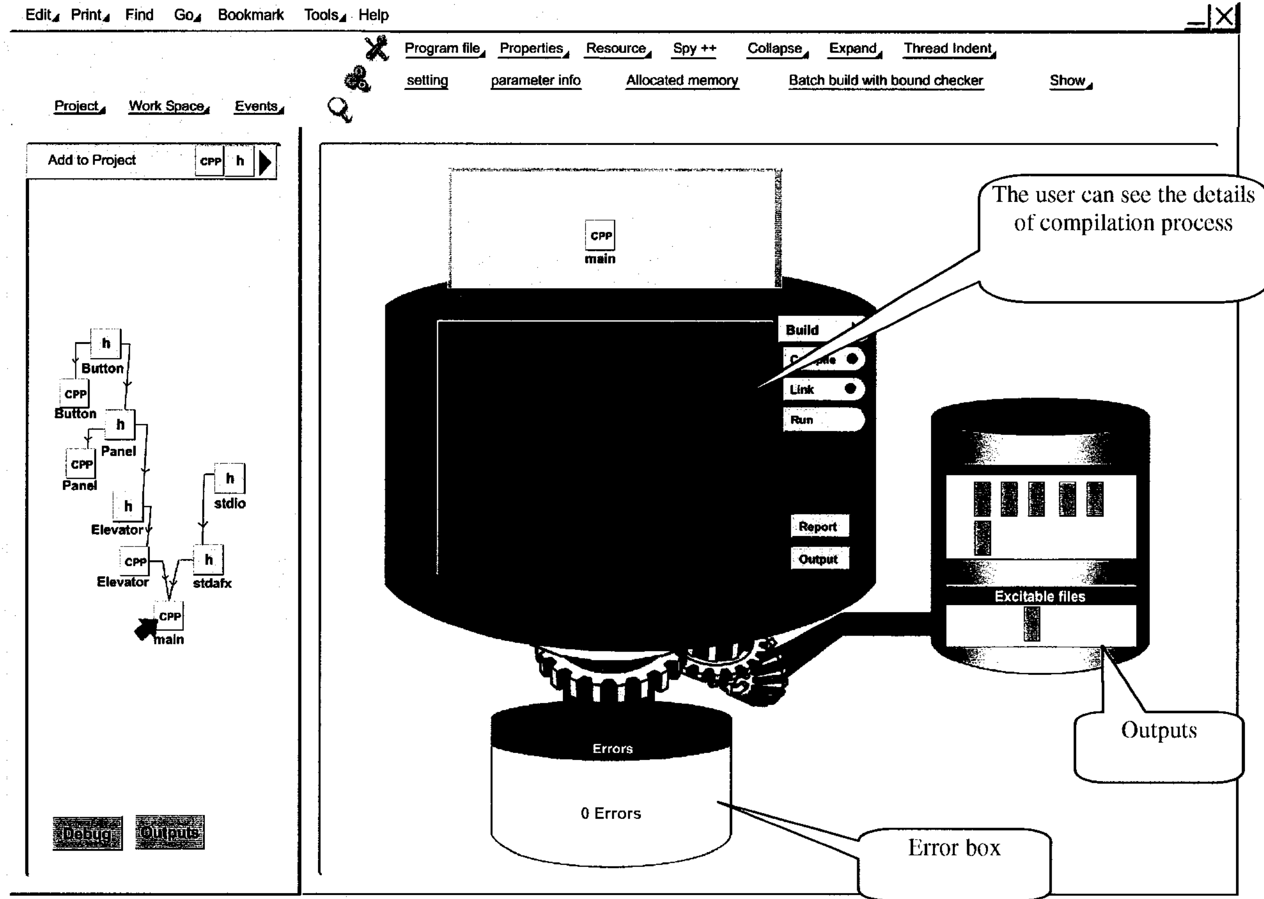


Figure 5.24. Microsoft C++ interface (Compile and file management)

iv. Debug:

Figure 5.25. Debug Icon

The process of debugging is similar to checking all the factories and the related product to determine if there are any problems. A problem can be related to a specific factory (C++ program object), or related to a road between two factories, which is closed (Link error). In either case, showing the errors on the map can help the user to detect the cause of the problem. For example, seeing the errors on the map in class

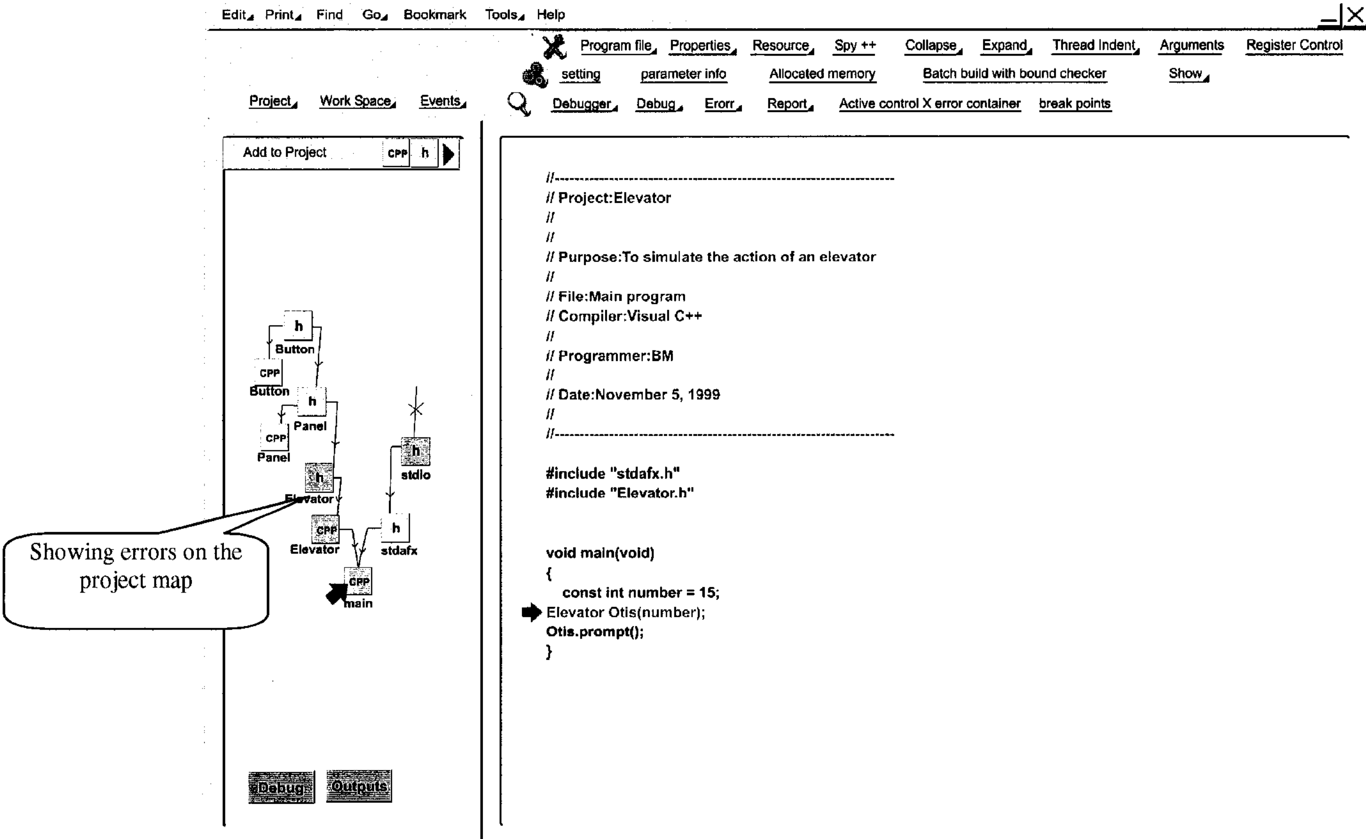


Figure 5.26. Microsoft C++ interface (debug and file management)

Please note that we limited the use of icons only for the main tasks because using too many icons makes the interface crowded and complex. Moreover, those programmers are more comfortable with text than visual image. In our cognitive method on Microsoft C++, we realized that users would go over the icons and wait for the hover text to give an explanation of the icon before clicking on it.

5.5.5. Testing the Metaphor

We made a low fidelity prototype and tested it with five programmers who were using Microsoft Visual C++. During the test we walked through the same nine tasks that we used previously in our CW method in Microsoft C++. We then asked the users what they thought about the interface. All agreed that the interface is simpler and more pleasant to work on. However, we should develop the interface and conduct a CW with a different group of users to see if the metaphor is really effective.

Chapter 6: Conclusion and Future Investigations

In this thesis, I demonstrated how to simplify the user interface of complex CASE tools by developing a human-centered, task-sensitive interface. First, I explored the usability problems related to CASE tools and especially IDEs that are among the most complex software's. The complexity of their user interface makes them difficult to learn, understand and master. Through a series of empirical studies, including CW method, I highlighted the usability problems related to the complex interface of these tools (Naghshin et la, 2003). Then, I demonstrated how the concept of persona could be expanded to model the behaviors of programmers - The users of CASE tools. Personas help us in the process of UI simplification. In the next step, I defined the general process of visual simplification and how simplification can vary according to the developer persona. Additionally, I introduced the idea of simplifying CASE tools by developing a task-sensitive user interface. I also demonstrated a step-by-step approach to create a task-sensitive metaphor for Microsoft C++ program.

The first contribution of this research is in constructing an effective set of developers' personae through empirical studies. The main idea of constructing personae is to try to understand who the developers are and how they work. Persona descriptions do not need to contain every detail. However, if we invent model of developers' and call them personae, we'll just have the same old problems with the software engineering tools packaged up in a new way. If we truly want to build a more human-centric software engineering tools, we should create developers' personae based on empirical evidence. I proposed to use the cognitive and code walkthroughs to help in the "upstream activities" of identifying and refining personae (Naghshin et la, 2004). Beside usability, learnability

and comprehension problems, CW helps also in highlighting some developer's behavioral patterns.

Another contribution of this research is the proposed concept of task-sensitive interface for CASE tools. Most of the CASE tools are functionality-oriented, which means that all functions are offered to the users at once. This makes the interface very difficult to use. Moreover, CASE tools are design to support a certain software development method, which is not necessarily followed by the end-users. I proposed to divide the interface according to developers' tasks and workflow. In a case study on Microsoft Visual C++ program, I demonstrated how to reorganize the interface according to five major tasks (Create / Open, Edit, Compile, Debug, and Archive). I asked 10 developers to decide during each of these five tasks which functionality they need. Using IBM Esort software, then I sorted the functionalities according to each task.

Another important contribution of this research is the idea of task-sensitive metaphor for reengineering a CASE tool user interface. The current desktop metaphor that has been used in all GUI interface does not cover all the major tasks that developers are asked to accomplish. This creates a gap between users' mental model and the software features and makes the interface very difficult to understand. Designers has tried to solve this problem by using multiple or composite metaphors. The problem with composite metaphors occurs when the conceptual models mismatch between 2 metaphors, which is the problem of today's CASE tools. Within the framework of my study, I developed a new metaphor for Microsoft C++ program that could cover all main tasks and be engaging enough for the users while encouraging them to explore a large set of features.

Task-sensitive CASE tools can be used to simplify complex user interfaces. However, it also lead to some fundamental problems that need to be further investigated. First of all the usefulness of task- sensitive interfaces cannot be fully examine without testing the actual interface with users. We need to fully develop a task sensitive tool and ask the users to use this tool for at least 2-3 weeks, since some interfaces are easy to learn, but they may not help the user's productivity in the long term. After testing the actual interface with users, we may need to slightly change our interface, making it more users-centric.

Another issue for future investigations consists to create a framework to adapt a UI metaphor to the different group of users. This adaptation could be at the level of changing the look and feel of interface, or at the level of adapting it to the user's task flow. For instance, in my case study I considered only software programmers. This task flow may be slightly different for project managers, or software designers. This adaptation is even more important during UI maintenance. For example in the case of adding any new function to a task-sensitive interface, we need to identify for which tasks the function can be used, and how we can add a function while maintaining the consistency of the UI.

Bibliography

Aaen, I. (1992), CASE *Tool bootstrapping (how little strokes fell great oaks)*.Next Generation CASE Tools IOS, Netherlands , (pp 8–17).

Baik, J.,Boehm, B.,(2000) *Empirical Analysis of CASE Tool Effects on Software Development Effort*, Technical Report USC publications

Blackwell, A.F., and Green, T.R.G. (1999), *Does metaphor increases visual language usability?* Proceedings of IEEE Symposium on Visual Languages, volume 99, page 246, IEEE Computer Society

Brockerhoff, R. (2000), *User Interface Metaphors*, MacHack Conference proceeding, June 19-21, 2000

Budgen, D., and Thomson, M. (2003). *CASE tools evaluation: Experiences from an empirical study*. Journal of Systems and Software,No. 67, (p. 55-75)

Carrington, D. (2004), *Software engineering tools and methods, Guide to the software engineering body o f knowledge*. IEEE Society

Carroll, J. M. (1999). *Five Reasons for Scenario-Based Design*, proceedings of the 32rid Hawaii International Conference on System Science

Carroll, J. M. (2000). *Making Use: scenario-based design of human-computer interactions*, Cambridge, Mass, MIT Press

Comber, T. and Maltby, J. R. (1995), *Evaluating Usability Of Screen Designs With Layout Complexity*, ACM Library

Constantine, L. (2002), *Application Note Bare Essentials: A not on simplifying User Interfaces by Simplifying Use Cases*, Constantine & Lockwood Ltd

Constantine, L. (1998) *Use and Misuse of Metaphor*, Constantine & Lockwood, Ltd

Cooper, A. (1999) *The Inmates Are Running the Asylum: Why High Tech Products Drive Us Crazy and How To Restore The Sanity*. Macmillan Publication 1999

Drakos, N. and Moore R. (1999) *A Taxonomy of User-Interface Metaphors, Computer Based Learning Unit*, University of Leeds. Copyright © 1997, 1998, 1999, Macquarie University, Sydney.

Desmarais, M.C., and Liu, J. (1993). *Exploring the applications of user-expertise assessment for intelligent interfaces*, Proceedings of the IGCHI, (p308) ACM Press

Farrell, R, Breimer, E. (2000), *A Task-based Architecture for Application-aware*, IU1 2000 New Orleans LA USA, ACM 2000

Fuggetta, A. (1993), *Politecnico di Milano and CEFRIEL, "A Classification of CASE Technology"*, IEEE publication 1993

Gould, J.D., Boies, S.J., Lewis, C. (1991), Making *usable, useful, productivity-enhancing computer applications*, Communications of the ACM, Vol.34, No. 1, (pp.75-85)

Guindon, R., Krasner, H. and Curtis, B.(1987), *Breakdown and processes during the early activities of software design*, Empirical Studies of programmers, second Workshop, Ablex Publication

Jarzabek, S., and Huang, R. (1998), *The case for user-centered CASE tools Communication engineering tools more developer-centric*, Manuscript submitted for publication.

Juric, R. and Kuljis, J., (1999*) Building an Evaluation Instrument for OO CASE Tool* Assessment for Unified Modelling Language Support, IEEE

Juric,R. Tanik, F.B.,Bastiani, D.(1996), *the unified method rules* , Proceedings of the Second World Conference on Integrated Design and Process Technology, (Austin, Texas, US) IDPT Volume 2, (p. 272-279)

KAHN P., and LENK, K. (1998) *Principles of Typography for User*, Interface Design, interactions Magazine November and December 1998

Kline, R., and Seffah, A. (2002), *Empirical study of software developers*, experiences, IEEE Workshop on Empirical Studies of Software Maintenance (WESS)

Lakoff, G., and M. Johnson 1999, *Philosophy in the flesh*, New York: Basic Books.

Lakoff, G., and M. Johnson 1980, *Metaphors we live by*, Chicago and London: University of Chicago Press.

Lending, D. and Chervany, N. L. (1998), *CASE Tools: Understanding the Reasons for Non-Use, Computer Personnel* - April 1998

Lewis, C., and Rieman, J. (1993), *Task-Centered User Interface Design Q A Practical Introduction.* Distributed via anonymous ftp (Internet address: ftp.cs.colorado.edu), International Journal of Man-Machine Studies 36, 741-773.

Livari, J., (1996), *Why Are CASE Tools Not Used*? Communications of The ACM, Vol. 39, N° 10

Marcus,A. (1989),*Color In User Interface Design: Functionality And Aesthetics*, HCI, Proceedings, May 1998

Marcus, A. (1997), *Metaphor Design in User Interfaces: How to Manage Expectation, Surprise, Comprehension, and Delight Effectively*, CHI 97 Electronic Publications: Tutorials

Mikkelson, N., and Lee, W. O. (2000), *Incorporating user archetypes into scenario-based design*, Proc. UPA

Motte, S. (1998)*: Device Design Methodology for Trauma Applications*. CHI 1998: 590-594

Mullarky, R.(1998), *Design for Interaction*, Adobe Magazine, October 1998

Mullet, K. and Sano, D. (1995), *Designing Visual Interfaces*, Communication Oriented Techniques, SunSoft Press

Myers B. A. (1993*), Why are Human-Computer Interfaces Difficult to Design and Implement*? CMU-CS-- Carnegie Mellon University, October 1993

Naghshin, R., Seffah, A., and Kline, R. (2003), *Cognitive walkthrough + persona= an empirical infrastructure for modeling software developers*, Proceedings of the 2003 IEEE Symposium on Human Centric Computering Languages and Environments (HCC' 03) (pp. 239) Auckland, New Zeeland

Nielsen, J., and Molich, R. (1990). *Heuristic evaluation of user interfaces*, Proc. ACM CHI'90 Conf. (Seattle, WA,1-5 April), 249-256

Nielsen, J. (1994). *Heuristic evaluation, Usability inspection methods* (pp25-62): Wiley

Nielsen, L.(2002), *From user to character- an investigation into user-descriptions in scenarios*, ACM Publications, October 2002

Norman, D.A. (1988) "*The Design of Everyday Things*", User Centered design, New York, Basic Books, (pp. 53-178)

Norman, D. A. (2000), *Why Metaphor is a Double-Edged Sword*, Interactions May/Jun 2000

Norman, D.A. (2004), *Emotional Design*, published by Basic Books, New York (pp45-106)

Paterno', F (1996), *Moving Tasks at the Center of the Development*, Execution and Evaluation Process, SIG HCI Bulletin, Volume 28, Number 3

Pressman, R. S. (2004), *Software engineering: A practitioner's approach*, McGraw-Hill

Productmarketing.com Buyer and user persona, November/December 2003, retrieved on June 2004

Rauterberg, M. and Hof, M., (1995) *Metaphor Engineering: a Participatory Approach* in: W. Schuler, J. Hannemann and N. Streitz (Eds.), Designing User Interfaces for Hypermedia. Springer 1995, pp. 58-67

Richards, S.M., (1990). *Guidelines for Electronic Book Production*, Interactive Systems Research Group, School of Computing and Mathematics, University of Teesside, Cleveland, UK

Robertson G. G., Card S. K., Mackinlay J. D. (1993), *Information Visualization Using 3D Interactive Animation*. CACM, Vol. 36, No. 4, 1993, pp. 57-71.

Russell, F. (1993), *The case for CASE Software Engineering: A European Perspective.* IEEE Computer Society Press,Los Alamitos, Calif.,(pp.531–547)

Seffah A, and Rilling J. (2001). *Investigating the relationship between usability and conceptual gaps for human-centric CASE tools*, IEEE Symposia on Human-Centric Computer Languages and Environments (p. 226)

Shneiderman, B. (2002). *Creativity support tools*, Communications of the ACM, (p. 116)

Sim, S. E., Singer, J., and Storey M. D. (2001): Beg, *Borrow, or Steal: Using Multidisciplinary Approaches in Empirical Software Engineering Research. Empirical Software Engineering* 6(1): 85-93

Spool J. M. and Snyder C. (1998), *Designing for Complex Products*, CHI'95 MOSAIC OF CREATIVITY May 7-11 1995 Tutorials

Tahir, M. F. (1997). *Who's on the other side of your software: Creating user profiles through contextual inquiry*, Proceedings of Usability Professionals Association Conference

Tidwell D. and Fuccella J.(1997), *Task Guides: Instant Wizards on the Web*, IBM Corporation, SIGDOC 97 Snowbird Utah USA

Tullis, T. S. (1983). *The formatting of alphanumeric displays: a review and analysis*, Human Factors, 25(6), 557-582

Vredenburg, K., & Butler, M. B. (1996), *Current Practice and Future Directions in User-Centered Design*, Usability Professionals' Association Fifth Annual Conference, Copper Mountain, Colorado

Usability First, *Resources Usability Glossary: assistant / wizard*, retrieved on July 2004, http://www.usabilityfirst.com/glossary/main.cgi?function=display_term&term_id=286

Waloszek, G. (2000), *What Does "Simple" Mean?*, Product Design Center, SAP's resource & forum for people-centric design (SAP.com) retrieved May 2004

Wasserman, A. I., (1990), *Tool Integration in Software Engineering Environments*, In Proc. Int. Workshop on Environments, Berlin, (pp137-149)

Weinstein, A., (1998), *Defining your market: winning strategies for high-tech, industrial, and service firms*. New York: Haworth Press.

Wilson D., Rauch T., and Paige J. (1992), *Generating Metaphors for Graphical User Interface,* ACM CHI '92 conference, third in a series.

Zarrella, P.F., (1990), *CASE Tool Integration and Standardization*, Technical Report, Software Engineering Institute, Carnegie Mellon University publications