



**Designing Optimal Hierarchies for
Information Retrieval with
Multifunction Displays**

By

Gregory Francis

Purdue University

Aircrew Health and Performance Division

July 1998

DTIC QUALITY INSPECTED 1

Approved for public release, distribution unlimited

**U.S. Army Aeromedical Research Laboratory
Fort Rucker, Alabama 36362-0577**

19980908 010

Notice

Qualified requesters

Qualified requesters may obtain copies from the Defense Technical Information Center (DTIC), Cameron Station, Alexandria, Virginia 22314. Orders will be expedited if placed through the librarian or other person designated to request documents from DTIC.

Change of address

Organizations receiving reports from the U.S. Army Aeromedical Research Laboratory on automatic mailing lists should confirm correct address when corresponding about laboratory reports.

Disposition

Destroy this document when it is no longer needed. Do not return it to the originator.

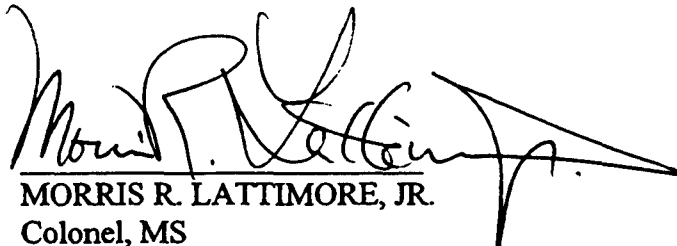
Disclaimer

The views, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other official documentation. Citation of trade names in this report does not constitute an official Department of the Army endorsement or approval of the use of such commercial items.

Human use

Human subjects participated in these studies after giving their free and informed voluntary consent. Investigators adhered to AR 70-25 and USAMRMC Reg 70-25 on Use of Volunteers in Research.

Reviewed:



MORRIS R. LATTIMORE, JR.
Colonel, MS
Director, Aircrew Health &
Performance Division

Released for publication:



JOHN A. CALDWELL, Ph.D.
Chairman, Scientific Review
Committee



for CHERRY G. GAFFNEY
Colonel, MC, SFS
Commanding

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 0704-0188	
1a. REPORT SECURITY CLASSIFICATION Unclassified			1b. RESTRICTIVE MARKINGS		
2a. SECURITY CLASSIFICATION AUTHORITY			3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release, distribution unlimited		
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE					
4. PERFORMING ORGANIZATION REPORT NUMBER(S) USAARL Report No. 98-33			5. MONITORING ORGANIZATION REPORT NUMBER(S)		
6a. NAME OF PERFORMING ORGANIZATION U.S. Army Aeromedical Research Laboratory		6b. OFFICE SYMBOL (If applicable) MCMR-UAD	7a. NAME OF MONITORING ORGANIZATION U.S. Army Medical Research and Materiel Command		
6c. ADDRESS (City, State, and ZIP Code) P.O. Box 620577 Fort Rucker, AL 36362-0577			7b. ADDRESS (City, State, and ZIP Code) Fort Detrick Frederick, MD 21702-5012		
8a. NAME OF FUNDING / SPONSORING ORGANIZATION		8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER		
8c. ADDRESS (City, State, and ZIP Code)			10. SOURCE OF FUNDING NUMBERS		
			PROGRAM ELEMENT NO. 62787A	PROJECT NO. 30162787A879	TASK NO. PB
11. TITLE (Include Security Classification) Designing Optimal Hierarchies for Information Retrieval With Multifunction Displays (U)					
12. PERSONAL AUTHOR(S) Gregory Francis					
13a. TYPE OF REPORT Final		13b. TIME COVERED FROM TO		14. DATE OF REPORT (Year, Month, Day) 1998 July	
15. PAGE COUNT 64					
16. SUPPLEMENTAL NOTATION					
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) Cockpit design, hierarchy, multifunction displays, workload		
FIELD	GROUP	SUB-GROUP			
19. ABSTRACT (Continue on reverse if necessary and identify by block number) Modern aircraft use computer screens with a push button interface to replace a variety of single-purpose instruments. Such multifunction displays (MFDs) are gradually being introduced into military helicopters, with future aircraft likely to be highly dependent on computers. Studies have shown that poor design of MFD hierarchies has a significant impact on user satisfaction and performance. The purpose of this study was to extend a theoretical analysis of hierarchy search into a methodology for gathering data and building a hierarchy layout that minimized the time needed to find items in a hierarchy. Pilot studies demonstrate the effectiveness of the methodology and show that optimizing hierarchy layout may lead to a 25% reduction in search times.					
20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS			21. ABSTRACT SECURITY CLASSIFICATION Unclassified		
22a. NAME OF RESPONSIBLE INDIVIDUAL Chief, Science Support Center			22b. TELEPHONE (Include Area Code) (334) 255-6907		22c. OFFICE SYMBOL MCMR-UAX-SS

Table of contents

	<u>Page</u>
Introduction.....	1
General approach	3
A hierarchical interface	3
Measuring motor time	5
Measuring categorization time	5
Optimizing hierarchical layout	7
Testing.....	7
Example 1	7
Example 2.....	12
Conclusions.....	12
References	15
Appendix A. Hierarchy search computer software.....	17
Appendix B. Summary of Java classes	18
Appendix C. Java source code.....	28

List of figures

	<u>Page</u>
1. Schematized drawing of two pages from an MFD display and its push-button interface.	2
2. Three displays from the program designed to investigate user interactions in hierarchical search.....	4
3. The program for measuring motor time.....	6
4. Mean search times	9
5. Hierarchy displays along the path to Monterrey, Mexico, for the optimized hierarchy.	11
6. Mean search times for users in the second pilot study.	13
A-1. The Java classes written to explore hierarchy searches.	17

Introduction

Military and civilian aircraft in the 1960's and 1970's used many separate gauges, dials, lights, switches, buttons, circuit breakers, control wheels, and levels in tightly packed aircraft cockpits. The introduction of new instruments and data sources forced a competition for limited cockpit space. This competition was partly alleviated by the introduction of microcomputers and video displays into the cockpit environment. Multifunction displays (MFDs), capable of presenting a variety of information from different sources, replaced many electromechanical devices, thereby freeing room in the aircraft cockpit. Current MFDs are often similar in appearance and usage to automated teller machines in that crew members push buttons to move through a hierarchy of display pages containing instructions, information, or lists of user-activated functions. They increase the total amount of available information, with the limitation that only some of it is visible at any given time. An additional benefit of MFDs is to provide a simpler layout of cockpit instrumentation, so that crew members spend less time scanning for information and more time piloting the aircraft. The reduction in pilot workload due to the introduction of MFDs in the cockpit was a primary factor in eliminating the need for flight engineers in many current generation transport aircraft.

Figure 1 schematizes MFDs as they are used in a variety of modern aircraft. Information is supplied on a large computer monitor. Push-buttons surround the monitor to allow the crew to interface with the MFD computer. Figure 1A shows real-time status information from the aircraft engines and other aircraft systems (SYS). Figure 1B shows targeting information. The push-buttons along the sides of the MFD are associated with software-generated display labels, indicating jumps to additional display pages containing related information. Pressing a soft-key causes the MFD to display a new page containing the information or functions indicated by the key's label.

MFDs typically contain a wide range of single and multistep functions. The type of objects and information displayed on the MFD, the data acquisition channels that are represented by the displayed objects, the set of active database links, as well as the functions that soft-keys can activate are commonly grouped together logically on one or more interconnected display pages. Pilots dynamically select a display based on the information and functionality desired to accomplish changing flight management or combat tasks such as situational awareness, navigation, communications, systems monitoring, battlefield and threat monitoring, and targeting.

Despite the significant impact of MFDs on the layout of instrumentation in aircraft cockpits and the responsibilities of crew members, little is known about how users search for information in such systems. Several studies have investigated the physical characteristics of the displays and the push-button interface (e.g., Rash and Becher, 1982; Hannen and Cloud, 1995; Klymenko et al., 1997). These studies help insure that crew members can see the monitor and reach the buttons for a variety of conditions (e.g., direct sunlight using protective gloves). Other studies explore the opportunity to create new types of information displays (e.g., Braithwaite et al., 1997). In contrast, there has been little research to insure that crew persons can quickly search through the hierarchy of information in the MFD database to retrieve needed information.

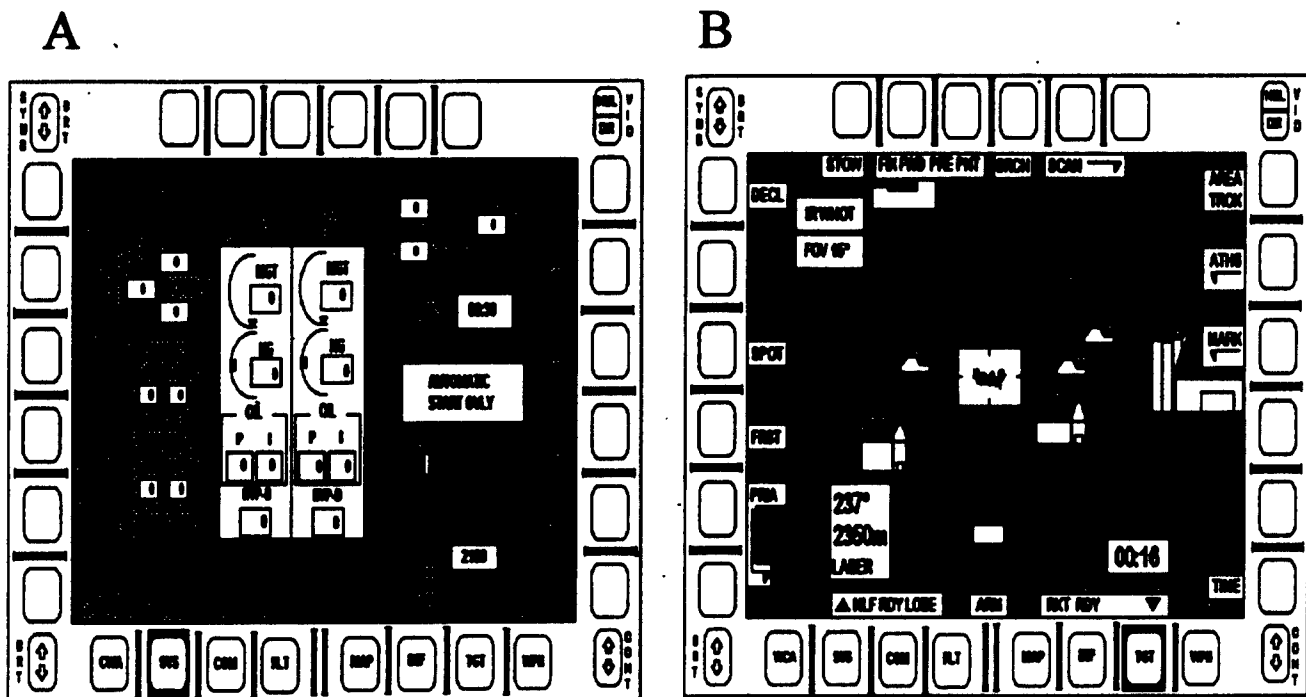


Figure 1. Schematized drawings of two pages from an MFD display and its push-button interface. In A, the systems page shows information on engines and includes legends along the right to indicate that pressing the associated button will cause the display to present the requested information. In B, the same display screen shows a page with targeting information.

The military guidelines (MIL-STD-1472D) for development of the hierarchical structure of information in the MFD provide few instructions and little justification. The small number of studies investigating hierarchy design issues may reflect the difficulty of the problem. As described in Francis and Reardon (1997), small changes in part of a hierarchy can have profound effects on search times elsewhere in the hierarchy. Such sensitivity makes general guidelines difficult to apply. As a result, hierarchy creation currently remains an artistic endeavor, depending primarily on the experience and intuition of the designer.

The few studies exploring the impact of hierarchy design suggest that it is important. In non-military domains, hierarchy design has been identified as a key factor in overall performance and satisfaction with an MFD type device (Seppala and Salvendy, 1985; Cook and Woods, 1996). Studies using simulated military aircraft suggest that MFD hierarchy design may affect crew workload and situation awareness (e.g., Reising and Curry, 1987; Sirevaag et al., 1993).

To promote a more rigorous analysis of hierarchy design, Francis and Reardon (1997) developed a mathematical framework that considers a variety of factors in hierarchy design. The current document shows how to apply that framework to particular cases of hierarchy design.

General approach

MFDs trade a search of physical space for a search of virtual space through the hierarchy of information. Other things equal, it is desirable to arrange the hierarchy of information in a way that minimizes the search time. Francis and Reardon (1997) provided a theoretical framework to consider this issue. They identified an optimization method that chooses the best layout of information in the hierarchy. For the optimization method to succeed, it required a model of the time needed to search through the virtual space of the MFD hierarchy. In this section, we briefly summarize the model proposed by Francis and Reardon and show how to apply it to a particular design task.

Each page in an MFD hierarchy defines a unique path of button pushes that terminates when the page is shown. Measuring the time to reach a hierarchy page requires knowing the time needed to move to and push the various buttons along the path to the page. These movement and push times all contribute to a *motor* term. These times are possibly distinct from the time required to decide which button to push. Such decisions require reading various choices until the option leading to the target page is identified. The times to read, interpret, and decide to select contribute to a *categorization* term. Together with any *computer response* time, this analysis suggests that the time to reach a page in the hierarchy will be:

$$T = (\text{motor}) + (\text{categorization}) + (\text{computer response}).$$

Each of these terms is likely to vary with the pattern of button pushes that define the path, the options to be categorized along the path, and the information displayed by the computer. By considering the variations in these variables for different buttons and items, the optimization method selects a layout that maximizes performance according to any designer-imposed constraint. This approach uses information in the details, or microstructure, of the human-computer interface to maximize performance.

The analysis in Francis and Reardon (1997) was theoretical; it described equations and techniques for identifying an optimal layout of information when specific data were available. We now explore methods for gathering the needed data and demonstrate the utility of the method. For that purpose, we developed a suite of computer programs to investigate hierarchical searches. The programs will be described briefly in the ensuing text and more fully in the appendices. The next section describes the basic interface used to explore hierarchical search. Data were gathered from this interface using methods described in subsequent sections. The data were then used to build an optimal layout of pages that considers the microstructure properties of the human-computer interaction. The validity of the optimization method was then verified with further experimentation.

A hierarchical interface

To investigate the microstructure of hierarchical information retrieval, we wrote a program in the Java programming language that allowed users to select virtual buttons with mouse controls. Selecting a button moved the user through a virtual hierarchy of information and generated a new set of options associated with each button. Figure 2 shows snapshots of the interaction window at various positions along a path to a target item. In this case, the display portrays

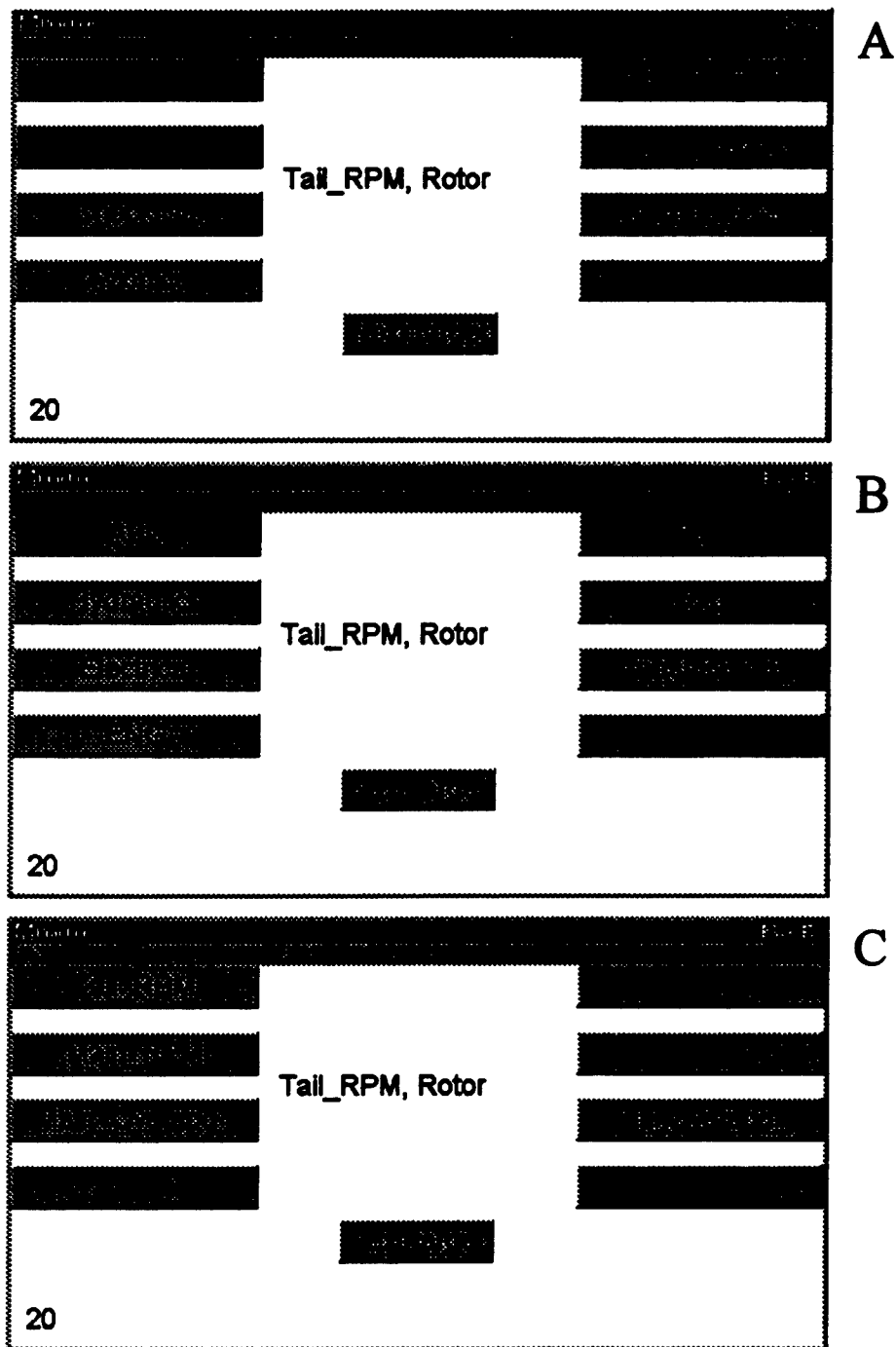


Figure 2. Three displays from the program designed to investigate user interactions in hierarchical search. The user is given a target item in the center of the window and moves through the hierarchy to find the item. The user moves through the hierarchy with successive mouse clicks on the appropriate buttons to reach the target. A shows the top level, where buttons code major systems in a helicopter. B shows a screen from the second level, where buttons code various aircraft systems. C shows a screen from the bottom level, where buttons code various types of rotor information, including Tail_RPM, the target item.

aircraft information suitable for use in a military helicopter. After pressing the *Next target* button, a target item (sometimes paired with its parent to help the user find it) was displayed in the center frame. The user was to move through the hierarchy toward the target item as quickly as possible.

Figure 2a shows the top level of the hierarchy, where the options are various aircraft information and the target is *Tail-RPM, Rotor*. Selecting the *Aircraft-SYS* button changes the labels on the buttons to those shown in Figure 2b, where the options are various choices in aircraft systems. Selecting the *Rotor* button changes the labels to those in Figure 2c, which show options for the rotor system. Selecting the *Tail-RPM* button means that the user has found the desired item. This basic scheme was used in a number of different ways to explore hierarchical search.

Measuring motor time

Defining the motor term of the model requires identifying the time needed to make button presses and to move the mouse control between buttons. This cannot be done during a normal hierarchical search because the time to find an item includes both the time to physically push the correct buttons and the time to categorize items. We hypothesized that the motor time could be isolated in a situation where the user knew in advance where to move the mouse control. In such a situation there would be no categorization time.

Thus, we created a program that required the user to select pairs of buttons. Figure 3 shows a snapshot of the program window. Each button is numbered between 0 and 7. The center region specified the pair of buttons that were to be pressed in succession. The program measured the time between the first and second button presses. This measure was repeated for every combination of successive button presses, including repeated selection of the same button. (The only exception was that the user was never asked to make a movement that ended in the *Next target* button, as such movements never occurred during searches of the hierarchy.) The time for each movement was measured several times, and the average stored in a file for later use. Because the user could plan the movement before the first button press, we hypothesized that the movement time was a pure measure of how long it took to physically move and initiate the mouse control.

Measuring categorization time

The optimization technique described in Francis and Reardon (1997) required, in addition to the motor times, the time needed to categorize items. Unlike the technique for measuring motor times, there seems to be no direct method of measuring categorization times. Any measure of response time necessarily will include both categorization and motor times. We need to disentangle these terms so that the model can predict response times when the hierarchical layout is restructured and the items are paired with new motor times.

To disentangle response times, we modified the basic hierarchy search program so that it measured the time between successive button presses as the user went through a path toward a target item. The time between button presses was coded by item name and stored in a file for later use. We refer to these measures as *between item search times*.

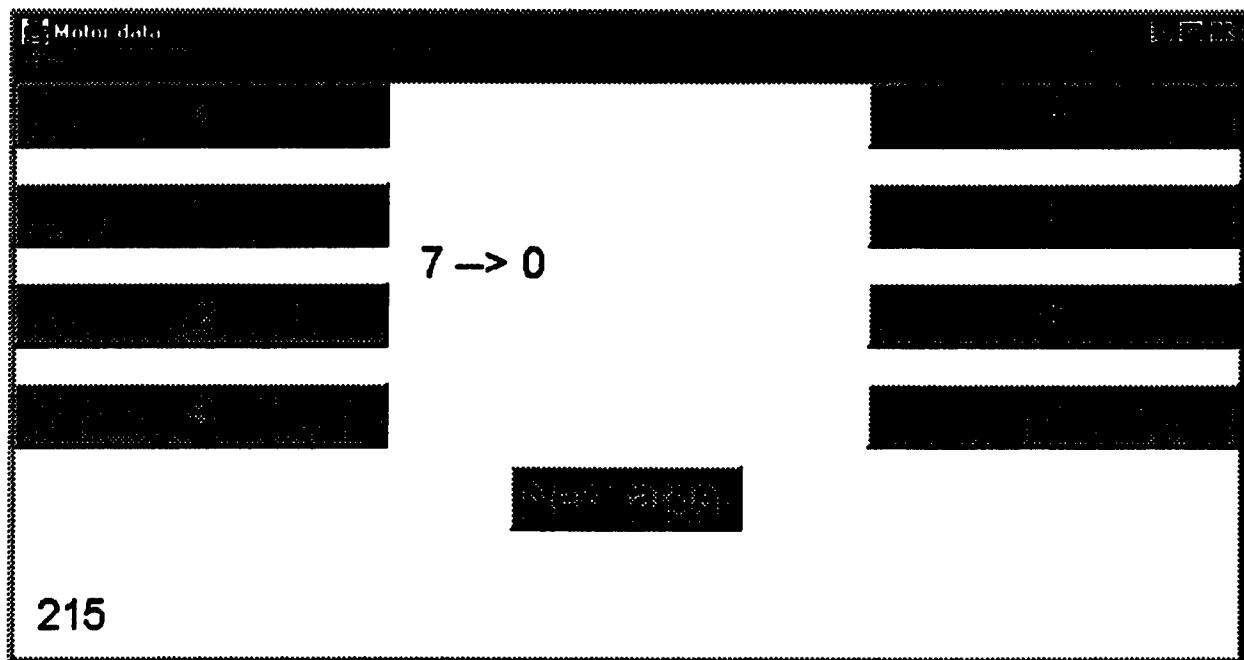


Figure 3. The program window for measuring motor time. The center panel displays a sequence of button presses for the user to perform. The program measures the time between the first and second button press. Since the user can find the buttons to be pressed before starting the movement, the time should include only motor time. Motor time was measured in this way for every pair of buttons.

Next, we used the previously measured motor times between button presses and the between item search times to derive categorization times. We used the following logic. The between item search time includes both motor and categorization time. Thus, subtracting the motor time from the between item search time should leave an estimate of the categorization time. This calculation was performed for every item in the hierarchy.

This approach has the benefit of avoiding another difficult problem in measuring categorization times. Previous attempts to model hierarchical search have noted that performance critically depends on the search strategy utilized by the user (Lee and MacGregor, 1985; Paap and Roske-Hofstrand, 1986; Vandierendonck, Van Hoe and De Soete 1988). In an extreme case, if the user is very familiar with searching through the hierarchy and knows the button presses needed to reach each item, categorization time will be negligible. On the other hand, if the user has no experience searching through the hierarchy, categorization time will be substantial and highly dependent on the details of the user's strategy. More commonly, a user will know the button presses needed to reach some items in the hierarchy but will need to search labels to find the path for other items. Such effects are likely to be highly dependent on the items in the hierarchy and their significance to a particular user, so there is probably no way to model the effects of learning.

Our measure of categorization time avoids modeling learning effects by measuring the resulting behavior that depends on those effects. From the point of view of predicting search times, it does not matter *why* some items are categorized more quickly than others, what does

matter is how long it takes to categorize each item. Our measure calculates the needed information directly without worrying about the underlying details.

With the motor and categorization data, it is possible to predict the time required by the user to find an item for *any* layout of information in the hierarchy. To make this prediction for a single item, the computer simply notes the path needed to reach the target item, the buttons that must be pushed along that path, and the items that must be categorized along the path. The time for all button pushes and movements and the time for all categorizations sum to equal the predicted search time.

Optimizing hierarchical layout

With the ability to predict search times for any layout of information in the hierarchy, it is possible to search through different layouts for the one that minimizes expected search time. Unfortunately, there are so many different possible distributions of items in the hierarchical structure that it is not feasible to consider them all. Instead, we used a computational technique called simulated annealing, as described in Francis and Reardon (1997). This computational algorithm sifts through the possible hierarchical layouts to consider only those that have the best chance of generating small search times. While the algorithm does not guarantee to find the optimal layout of items in the hierarchy, in practice it usually produces a layout with a search time close to the optimal.

Testing

A final program takes the hierarchical layout generated by the optimization program and generates the button interface for user interaction. As the user searches for specified target items, the computer keeps track of the search times. After the user is finished searching for items, the program writes to a file the predicted mean search time (as generated by the optimization program) and the mean actual search time (as measured during the user's interaction). These are then compared to each other to consider the accuracy of the model's predictions. They are also compared to the original time required to search for items in a non-optimal hierarchical layout of items. The next two sections describe pilot studies that used this general approach to hierarchy design.

Example 1

The first example shows application of the method using data gathered from a single user (the first author). The hierarchy portrayed geographical information (continent, country, city) instead of the aircraft information portrayed in Figure 2. This change was incorporated to insure that subjects not familiar with aircraft systems could participate. The methodology of building an optimal hierarchical layout remains the same regardless of the information in the hierarchy. Subjects were asked to move through the hierarchy to click on the button for a city, country, or continent.

The programs described above were run on a laptop computer. Of note, the mouse control was utilized through a touchpad device, which is common on a variety of laptop computers. A touchpad is a small touch-sensitive pad. A light touch on the pad gives the user control of the cursor placement. Dragging a finger along the surface of the touchpad moves the mouse cursor in the same direction. A mouse click is initiated by quickly tapping twice on the touchpad in the desired location. The touchpad device is useful for laptop computers because it offers the functionality of a mouse with small space requirements. However, precise control of cursor movement is somewhat difficult with the touchpad, and correction adjustments are frequently necessary. It is also sometimes difficult to start and stop movement of the mouse cursor, especially for small movements. As a result, sometimes a larger movement can be accomplished more quickly than a short movement. None of these characteristics affected the basic approach to hierarchy design, and motor movement times were gathered as described above. For each movement, the average of 10 replications was used as the measure of motor time.

If all items in the hierarchy are accessed equally often, and the user is very familiar with the path for each item, there is no difference between hierarchical layouts. Such situations are probably very rare. For most MFD applications, some items are searched for more often than others. The goal of the design process is to place frequently searched items at the end of hierarchical paths that are quickly accessed. To emulate the inhomogeneity of search frequency, we created artificial *mission scenarios*. Each scenario required the user to search for a fixed set of 20 randomly selected items from the full (268 item) set. In each scenario 10 of the items were searched for 5 times and the other 10 were searched for once. For each scenario, we gathered categorization data and built and tested an optimal hierarchy.

To gather categorization data, the hierarchical layout was partially randomized so that every item was located underneath its appropriate header category, but was in a random (fixed) position under that header (e.g. each city remained under its appropriate country, but was randomly assigned to a button). The randomization was used to insure that there were no order cues (e.g. alphabetical order) that would guide the user's search process. A scenario was run twice. On the second run of the scenario, between-item time was measured for each item encountered in the hierarchy. These items included both the target items in the mission scenario and the items located along the paths to reach the target items. We did not use data from the first run of the scenario, as it would likely show strong learning effects for those items frequently searched. The average between-item time for every item in the hierarchy was stored in a file to be used by the optimization program.

A program that created an optimal hierarchy converted the motor time data and the between-item data into independent motor time data and categorization time data. For the mission scenario, the program then considered different hierarchical layouts to identify the one that minimized predicted search time. This was a time-consuming process, requiring approximately 45 minutes for a scenario. When the optimization procedure finished, it wrote to a file the hierarchical position of each item.

Finally, a testing program read in the hierarchical data generated by the optimization program, and the user participated in two runs of the scenario. The testing program gathered data on the second run of the scenario to measure the mean time to reach an item in the optimal hierarchy. Figure 4a shows the expected time required to find a single item in the hierarchy for three different mission scenarios. For each scenario, three values are plotted: the expected search

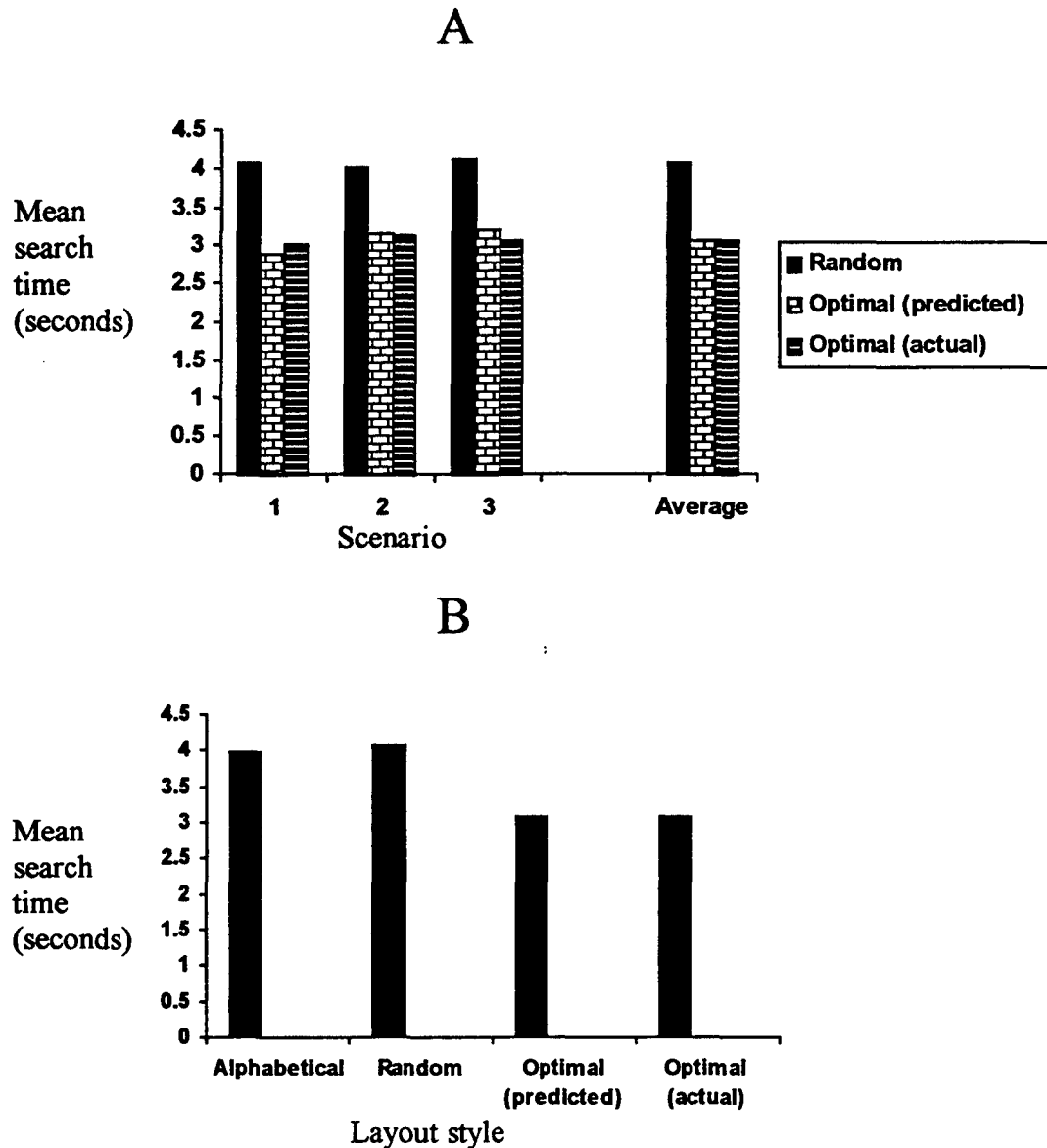


Figure 4. Mean search times. A shows expected search times for three scenarios and the average. Three measures are plotted for each scenario. Random indicates that the layout of information was randomly ordered on its appropriate page. Optimal (predicted) is the model's prediction of expected search time, using the layout of information that minimizes predicted search time. Optimal (actual) is the search time for the optimal layout as measured through user interaction. B shows averages across three trials for alphabetical, random, and optimal layouts.

time for the random hierarchy (used to gather between-item data), the model-predicted expected search time for the optimal hierarchy, and the actual (from user testing) expected search time for the optimal hierarchy. Averages across the three scenarios are also plotted.

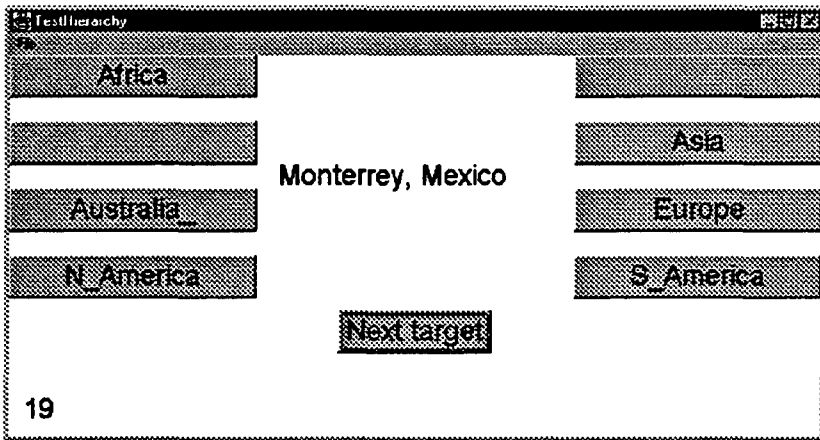
For the random hierarchies it took approximately 4 seconds to find an item in the hierarchy. When the items were rearranged according to the optimization procedure, it took approximately 3 seconds to find an item. This corresponds to a 25% reduction in search time, a substantial savings when one considers the large number of searches in an MFD.

Also noteworthy is the close correspondence between the predicted and actual performance on the optimal hierarchy. (By its design, the model must agree perfectly with user performance on the random hierarchies.) The strong agreement between the predicted and actual performance suggests that the model of search times accurately captures many of the important characteristics of hierarchical search.

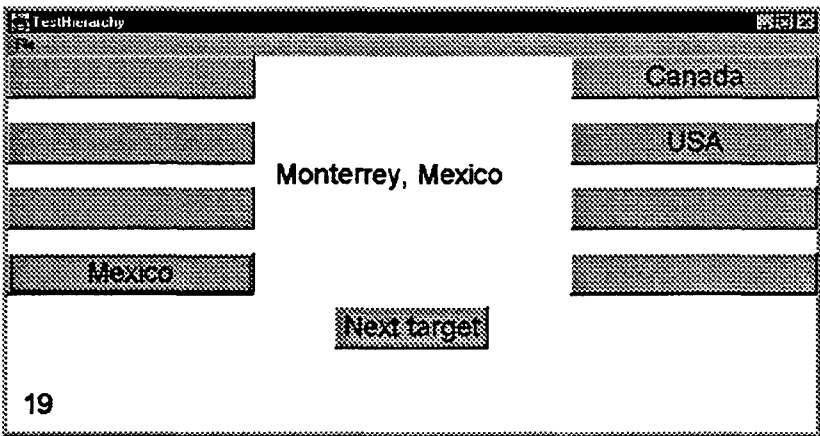
Finally, we wanted to compare the performance on the random and optimal hierarchies to what we suspect would be the default ordering in many situations. We measured mean search times for situations where the hierarchy items were ordered alphabetically on the buttons. Figure 4b shows the averages of three scenarios for the alphabetical, random, and optimal hierarchies. The alphabetical search times are similar to the random search times, and both are substantially larger than the optimal search time.

It is instructive to note some characteristics of the optimal layout. With the touchpad mouse controller, it takes substantial time to initiate a movement. As a result, the optimal layout created paths for the most frequently used items that involved repeated pressing of the same button. Figure 5 demonstrates the path for a commonly accessed item. Those items that could not be placed along a repeating path had paths that minimized movement time. In general, this organization is consistent with the guidelines suggested by military standards (MIL-STD-1472D). However, the computational method considers more. Certain buttons were more easily accessed than other buttons and certain paths in the hierarchy were more often traversed than other paths. It is no trivial task to decide which set of paths should be associated with which buttons because changing the location of one item requires additional changes among the children of that item. At the same time, one cannot identify the best location of items at the top levels without considering the best locations of items of their children. This type of circular dependence makes the layout choices very complicated. The computational approach is able to weigh all these dependencies simultaneously to generate the best overall hierarchical structure.

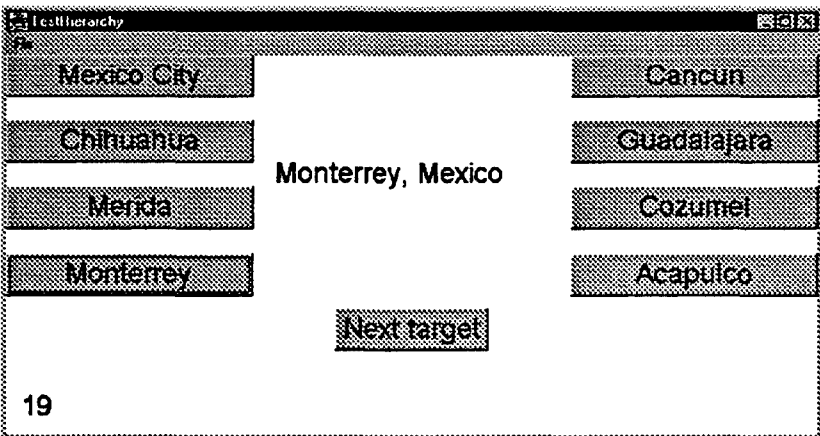
The overall feel of searching for information was that the target item would likely be found where the user expected it to be and would be easily accessible. We suspect that in addition to reducing search times, such optimal hierarchies will produce fewer errors and increase overall user satisfaction. We have not yet investigated these issues.



A



B



C

Figure 5. Hierarchy displays along the path to Monterrey, Mexico, for the optimized hierarchy. The user needs to make only one movement (from Next target to N_America) and then simply pushes the same button repeatedly to move through the hierarchy. Items that cannot be placed in such paths (because of interference from other item paths) are placed on paths that minimize movement time.

Example 2

A second study was run to insure that the methodology was general to a number of different conditions. For this purpose, three additional users participated. None had extensive practice working with the geographical hierarchy interface. The programs were run on a PC computer with a standard hand-held mouse.

For subject 1, motor time estimates were based on the average of five replications for each movement. The mission scenario was created in the same way as in Example 1. For subjects 2 and 3, motor time estimates were based on the average of three replications for each movement. The mission scenario consisted of seven different items, with individual items assigned a unique number of replications ranging from 1-7. For these users there were 28 target searches.

Figure 6 plots the mean search times for the users. The mean search times are generally smaller than for Example 1, probably indicating the more efficient control of the hand-held mouse versus the touch-pad device. The effects of optimization are not as strong as in Example 1. This could be because the users were less practiced searching through the hierarchy and made more mistakes (two participants indicated that they sometimes forgot which continent was associated with a country). Consistent with this interpretation, user 3 seems to show strong practice effects. If such a result were verified, it would emphasize the importance of gathering data from experienced users (which are more likely to mimic crew persons in military aircraft). There may also be floor effects where the advantage of repeated button pushes is not as great for the hand-held mouse as for the touch-pad device. Despite these possible confounds, the optimal hierarchies did result in overall shorter search times than the random hierarchies. For subject 1, additional explorations of search times with alphabetized hierarchies found mean search times slightly above 4 seconds.

Conclusions

We have developed a methodology to apply the theoretical framework of Francis and Reardon (1997) to the design of hierarchy layouts. The key insight in this methodology is to factor between-item times into motor and categorization times. By measuring motor times separately and subtracting them from between-item times, the method avoids many complicated issues that would otherwise prevent accurate prediction of search times.

We developed computer software to explore hierarchy search and gather data for designing optimal hierarchies. Two pilot studies demonstrate the utility of the methodology. The hierarchies that minimized predicted search time were found to be substantially better than random or alphabetically organized hierarchies. The results verified the benefit of optimizing hierarchy layout and also verified the adequacy of the model at predicting search times.

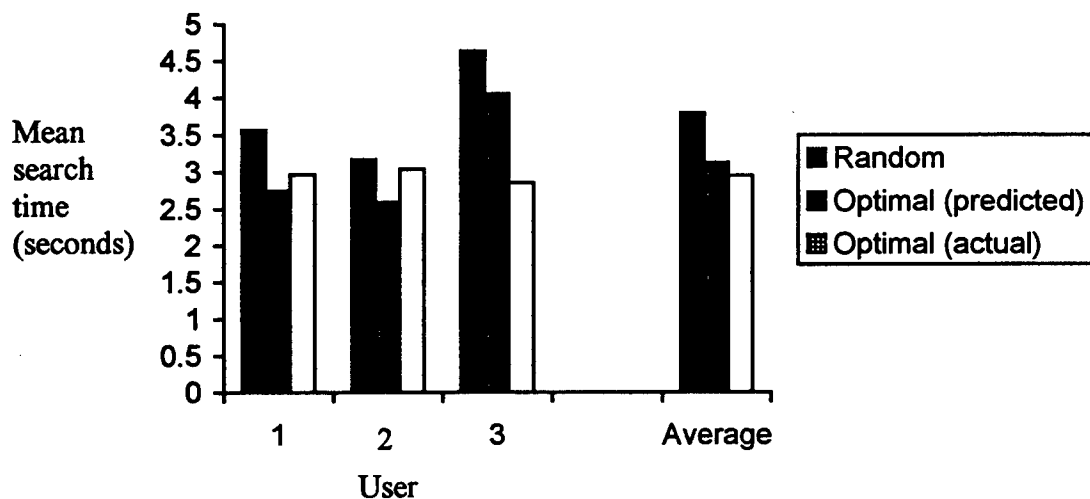


Figure 6. Mean search times for users in the second pilot study. The optimized hierarchies resulted in shorter search times than the random hierarchies.

To apply the methodology to the design of real MFDs requires gathering motor time in a real (or accurately simulated) cockpit, accurate measures of between-item search times, and good measures of the frequencies with which crew members use the various MFD functions. With this data, it should be possible to optimize the layout of items on buttons and reduce search times. However, one should consider a number of other issues before applying the optimization approach to MFDs in cockpits. First, there may be some functions that must be reached within specific time constraints or they are of no use to the crew (e.g., taking evasive action under fire). The optimization method should take such constraints into account. Second, the motor time data may vary depending on the function being searched for. For example, a pilot going through checklist procedures before take-off may need to spend very little time on flight controls and can quickly move through button pushes of the MFD. In contrast, a pilot taking evasive actions under fire may need to keep his hands on flight controls as much as possible, thereby increasing the average time needed to push buttons on the MFD. The model of search times needs to consider that some items may be associated with conditions that necessarily slow search time. With such considerations, the optimization method can design the hierarchy layout to accommodate those restrictions. Third, real MFDs often place restrictions on which buttons can be used. For example, in Figure 1A, the entire left side of the display is covered by engine information and is unavailable for labels linking to other pages. Such restrictions have not been considered in the hierarchy search programs considered here. Fortunately, there is nothing in the basic methodology to prevent consideration of these issues. Their resolution may require additional programming and data collection, but the theoretical framework remains unchanged.

A related issue concerns user variability. Even in the pilot studies, there are notable differences between participants' search times (there were differences in the mission scenarios as well). An MFD in an aircraft must accommodate a variety of users. As such, proper design of the hierarchy must gather data from a variety of users and create a *distribution* of motor and categorization times. With such data, it should be possible to design the hierarchy so that it

optimizes performance over the distribution of users. Nothing in the methodology prevents such design, although it will require that substantial amounts of data be gathered from a variety of users.

The current work provides the first, to our knowledge, scientific method to optimize hierarchical layout that considers the details of the human-computer interactions. Our analysis and experimental results suggest that the method may have a significant impact on usability of MFDs. Given the growing use of MFDs in both military and civilian aircraft, it is important to insure that they are designed to allow efficient retrieval of information. Our methodology provides a means to that end.

References

- Braithwaite, M., Durnford, S., DeRoche, S., Alvarez, E., Jones, H., Higdon, A., and Estrada, A. 1997. Flight simulator evaluation of a novel display to minimize the risks of spatial disorientation. Fort Rucker, AL: U.S. Army Aeromedical Research Laboratory. USAARL Report No. 97-11.
- Cook, R. and Woods, D. 1996. Adapting to new technology in the operating room. Human Factors. 38: 593-613.
- Department of Defense. 1981. Military standard: Human engineering design criteria for military systems, equipment, and facilities. MIL-STD-1472D.
- Flanagan, D. 1996. Java in a nutshell. O'Reilly & Associates, Sebastopol, CA.
- Francis, G. and Reardon, M. 1997. Aircraft multifunction display and control systems: A new quantitative human factors design method for organizing functions and display contents Fort Rucker, AL: U.S. Army Aeromedical Research Laboratory. USAARL Report No. 97-18.
- Hannen, M., and Cloud, T. 1995. A case study in the design and testing of hands-on controls: The Longbow Apache grip development process. In: Proceedings of the American Helicopter Society 51st Annual Forum. 1417-1435.
- Klymenko, V., Harding, T., Martin, J., Beasley, H., Rash, C. and Rabin, J. 1997. Image quality figures of merit for contrast in CRT and flat panel displays. Fort Rucker, AL: U.S. Army Aeromedical Research Laboratory. USAARL Report No. 97-17.
- Lee, E., and MacGregor, J. 1985. Minimizing user search time in menu retrieval systems. Human Factors. 27: 157-162.
- Morrison, M. (ed.) 1997. Java Unleashed: Second Edition. Sams.net Publishing, Indianapolis, IN.
- Paap, K., and Roske-Hofstrand, R. 1986. The optimal number of menu options per panel. Human Factors. 28: 377-385.
- Rash, C., and Becher, J. 1982. Analysis of image smear in CRT displays due to scan rate and phosphor persistence. Fort Rucker, AL: U.S. Army Aeromedical Research Laboratory. USAARL Report No. 83-5.
- Reising, J., and Curry, D. 1987. A comparison of voice and multifunction controls: Logic design is the key. Ergonomics. 30: 1063-1077.
- Sirevaag, E., Kramer, A., Wickens, C., Reisweber, M., Strayer, D., and Grenell, J. 1993. Assessment of pilot performance and mental workload in rotary wing aircraft. Ergonomics. 36: 1121-1140.

Seppala, P. and Salvendy, G. .1985. Impact of depth of menu hierarchy on performance effectiveness in a supervisory task: Computerized flexible manufacturing system. Human Factors. 27: 713-722.

Vandierendonck, A., Van Hoe, R., and De Soete, G. 1988. Menu search as a function of menu organization, categorization, and experience. Acta Psychologica. 69: 231-248.

Appendix A. Hierarchy search computer software

The appendices describe the computer software used to investigate hierarchical search. All software was written in the Java programming language (for a discussion of Java, see Flanagan, 1996; Morrison, 1997). This language was chosen because it has built-in commands for creating windows, buttons, and handling user interfaces. Java programs also have the advantage of being machine-independent, meaning that the programs will run on any machine platform (PC, Macintosh, Unix), provided that platform supports a Java virtual machine.

Java is an object oriented programming language, meaning the programmer defines *classes* that contain attributes and methods for manipulating the attributes. One benefit of this programming approach is that a class can inherit characteristics of another class, thereby reducing the need to rewrite code. Figure A-1 shows the relationships between the classes used to investigate hierarchical search.

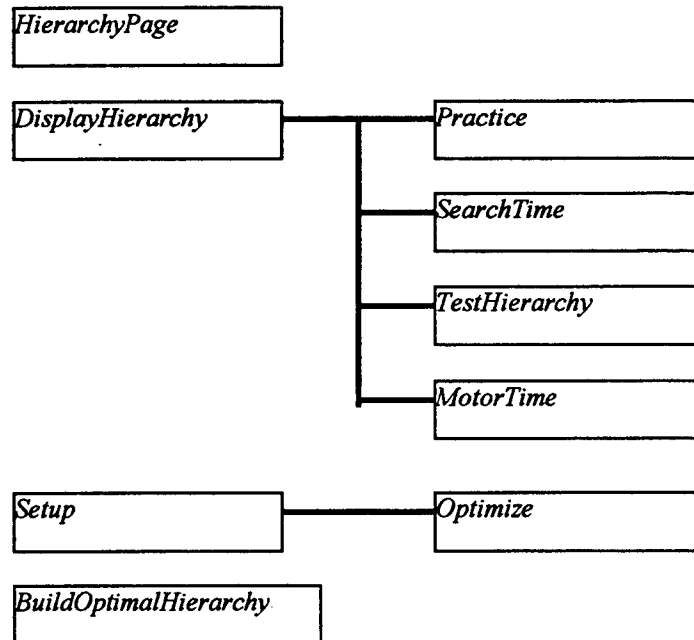


Figure A-1. The Java classes written to explore hierarchy searches. *HierarchyPage* provides data structures and methods useful for working with an item in the hierarchy. It is used by many of the other programs. *DisplayHierarchy* is a class for basic windowing and interfaces with a given hierarchy and mission scenario. The classes *Practice*, *SearchTime*, *TestHierarchy*, and *MotorTime* all derive from this class and add or change methods to compute different statistics and read/write to different files. *Setup* provides a general scheme for creating a hierarchy structure. *Optimize* modifies the general scheme to create an optimal hierarchy. *BuildOptimalHierarchy* provides an interface to go through each step in the process of building a hierarchy optimized for a single mission scenario.

Appendix B. Summary of Java classes

This section provides object specifications for each Java class used. For each class, the object specification provides a brief description of the class' purpose, the attributes associated with the class, and the methods used by the class to carry out calculations.

HierarchyPage

Description

The HierarchyPage class represents an item in the hierarchy. It keeps track of the item name, the path of button pushes needed to reach the item, the single-number position that corresponds to the path, the frequency with which the page is searched for, and the level at which the button path terminates.

Attributes

Attribute Name	Data Type	Description
name	String	The name of the page.
categorize_time	int	A statistic of how long it takes to reach the page. Its precise definition depends on the class that invokes the HierarchyPage.
path	int []	An array that describes the sequence of button pushes needed to reach the page.
num_replics	int	The number of times the page is to be searched for in the mission scenario.
replics	int	The number of times the page has been searched for.
num_buttons	int	The number of buttons in the hierarchy.
num_levels	int	The number of levels in the hierarchy.

Methods/Events

Method Name	Return Type	Parameters	Description
getPosition	int		Returns the position of the page.
getLevel	int		Returns the level of the page.
setPathfromPosition	void	int position	Derives the path of the page from a given position.
setPath	void	int[] path	Sets the path of the page given an array.

DisplayHierarchy

Description

The DisplayHierarchy class reads in hierarchical information from a data file and creates a window with buttons to display the hierarchy. It handles all button presses and relabels the buttons to emulate movement through the hierarchy. It also provides routines for gathering statistics on user performance.

Attributes

Attribute Name	Data Type	Description
randGen	Random	A random number generator.
num_buttons	int	The number of buttons in the hierarchy.
num_levels	int	The number of levels in the hierarchy.
b	Button []	An array of buttons. Hierarchy labels are placed on these buttons.
target	String	The name of the item the user is to find.
search_for_counter	Label	A label that displays information in the middle panel of the window.
next_target	Button	The <i>Next target</i> button.
level	int	Identifies the current level of the hierarchy during user search.
found_target	boolean	Notes whether the user has found the target item.
start_time	long	Used for gathering reaction time data.
num_trials	int	The total number of trials in a mission scenario.
trial	int	The current number of trials that have been performed.
bp_count	int	A count of how many button presses have been made during the current search.
path	int []	Keeps track of the current sequence of button pushes generated by the user.
page	HierarchyPage[]	The pages of the hierarchy.
block	int	The number of blocks that have been run.
num_blocks	int	The number of blocks that are to be run during a testing session.
target_index	int	The index of the target item.

Methods/Events

Method Name	Return Type	Parameters	Description
action	boolean	Event e, Object arg	Handles all user-generated events.
update_statistics_no t_target	void	int [] path	Not implemented.
update_statistics_tar get	void	int [] path	Not implemented.
get_data_from_file	void		Reads in hierarchy and mission scenario data from the file "data/items.txt".
write_to_file	void		Not implemented.
findPagewithPath	int	int [] path	Identifies the index of the hierarchy page that has the given path of button pushes.
GetSearchItem	void		Selects the next item for the user to search for.
buildDisplayPage	void		Relabels all buttons to emulate movement through the hierarchy.

Practice

Description

The Practice class instantiates the DisplayHierarchy class. It has no new attributes or methods.

Description

The SearchTime class extends the DisplayHierarchy class by providing calculations of the between-item times.

Attributes

Attribute Name	Data Type	Description
replics	int []	An array that provides, for each item, a count of how often that item has been encountered while the user searches the hierarchy.
first_replics	int	A count of how many times the user has encountered any item for the first time.
first_time	int	A measure of how the between-item times for the first time an item is encountered.

Methods/Events

Method Name	Return Type	Parameters	Description
update_statistics_no t_target	void	int [] path	Calculates between-item time for the current user selection.
update_statistics_tar get	void	int[] path	Calculates between-item time for the current user selection.
get_data_from_file	void		Differs from method in super only in the declaration of some variables.
write_to_file	void		Writes average between-item times, hierarchical layout, and mission scenario to file "data/search_time.txt".

MotorTime

Description

The MotorTime class extends the DisplayHierarchy class. It uses the methods and data structures in DisplayHierarchy to create a window and hierarchy interface, however, it redefines other methods to explore movement times between pairs of buttons.

Attributes

Attribute Name	Data Type	Description
replics	int []	An array that provides, for each pair of buttons, a count of how often the user has performed the movement between the pair.
num_replics	int	The number of times the user must make a movement between each pair of buttons.
target1	String	The name of the first button in a movement pair.
target2	String	The name of the second button in a movement pair.
button1	int	The index of the first button in a movement pair.
button2	int	The index of the second button in a movement pair.
found_target1	boolean	Set to <i>true</i> when the user selects the first button in the movement pair.
time	int []	An array that provides, for each pair of buttons, the sum of time required to move between the pair.
stat	int []	An array that provides, for each pair of buttons, the sum of time squared required to move between the pair. Used to calculate standard deviations of motor times.

Methods/Events

Method Name	Return Type	Parameters	Description
action	boolean	Event e, Object arg	Handles all user-generated events.
GetSearchPattern	void		Randomly selects the next pair of buttons for the user to make a movement.
get_data_from_file	void		Sets up hierarchy, does not actually read from a file.
write_to_file	void		Writes, for every pair of buttons, the average motor time and the standard deviation to file "data/motor.txt".

Description

The TestHierarchy class extends the DisplayHierarchy class by providing calculations of the time needed to find the target item.

Attributes

Attribute Name	Data Type	Description
original_est	int	The total time required to search the mission scenario with the original hierarchical layout. Read in from a data file.
est	int	The predicted time required to search the mission scenario with the optimal layout. Read in from a data file.

Methods/Events

Method Name	Return Type	Parameters	Description
update_statistics_target	void	int[] path	Calculates the time taken by the user to find the target item.
get_data_from_file	void		Reads in the optimal hierarchical layout and mission scenario information from the file "data/optimal.txt".
write_to_file	void		Writes mean search time data (original, predicted, actual) to file "data/mean_time.txt".

Setup

Description

The Setup class reads in user supplied data files, which define a hierarchy, and creates a mission scenario. It modifies the hierarchy layout so it is partly randomized, and writes the hierarchy layout and mission scenario information to a file.

Attributes

Attribute Name	Data Type	Description
num_buttons	int	The number of buttons in the hierarchy.
num_levels	int	The number of levels in the hierarchy.
num_labels	int	The number of pages in the hierarchy.
page	HierarchyPage []	The pages of the hierarchy.

Methods/Events

Method Name	Return Type	Parameters	Description
get_data_from_file	void		Reads in hierarchical information, starting with the file "Item_names/level0.txt". Then reads in other files with names matching the item names.
create_new_hierarchy	void		Randomizes the hierarchy layout and defines a mission scenario.
write_to_file	void		Writes the hierarchical layout and mission scenario information to the file "data/items.txt".
findPagewithPath	int	int [] path	Identifies the index of the hierarchy page that has the given path of button pushes.
swap	void	int level, int item, int new_item	Swaps the paths of pages item and new_item. Also changes the paths of the children of these items to keep the hierarchical order intact.

Description

The Optimize class extends the Setup class. It reads from files between-item time data and motor time data. It calculates categorization time data and defines a model of search times. It then uses an optimization technique to find a hierarchical layout that minimizes predicted search time. The optimal hierarchy (along with its mission scenario and predicted search time) is written to a file.

Attributes

Attribute Name	Data Type	Description
motor_time	int []	The time needed to move between each pair of buttons.
ast	int	The time needed to go through the mission scenario with the original layout.
original_est	int	The time needed to go through the mission scenario with the original layout. Using class computations. Useful for comparing to <i>ast</i> to detect bugs.
current_est	int	The predicted search time for the current hierarchical layout.
local_minima	int	The predicted search time for the best hierarchical layout yet found by the optimization routine.

Methods/Events

Method Name	Return Type	Parameters	Description
get_data_from_file	void		Reads in hierarchical, mission scenario, and between-item time data from file "data/search_time.txt". Creates hierarchy pages. Also reads in motor time data from file "data/motor.txt".
create_new_hierarchy	void		Computes categorization time and stores it in hierarchy pages. Uses an optimization procedure to find the hierarchical layout that minimizes predicted search time.
local_minima_check	boolean		Returns <i>true</i> if the current hierarchy layout is a local minima of predicted search time.
computed_expected_search_time	int		Returns the predicted time (in milliseconds) for the user to search through the mission scenario with the current hierarchy layout.
write_to_file	void		Writes the hierarchical layout and mission scenario information to the file "data/optimal.txt".
swap	void	int level, int item, int new_item	Swaps the paths of pages item and new_item. Also changes the paths of the children of these items to keep the hierarchical order intact.

BuildOptimalHierarchy

Description

The BuildOptimalHierarchy class provides an interface to guide a user through the creation of gathering all needed data and testing an optimal hierarchy. It calls each class as needed.

Attributes

Attribute Name	Data Type	Description
b	Button []	An array of buttons.

Methods/Events

Method Name	Return Type	Parameters	Description
action	boolean	Event e, Object arg	Handles all user-generated events. Calls appropriate classes according to button.

Appendix C. Java source code

This section provides the source code of each Java class.

```
/* class HierarchyPage
```

This class provides an object to contain all the data relevant to a specific page in a hierarchy. This class is used by several different programs.

Written by Greg Francis, Purdue University
August 1997

The views opinions, and/or findings contained in this report are those of the author and should not be construed as an official Department of the Army position, or decision, unless so designated by other documentation.

```
*/
```

```
class HierarchyPage
```

```
{
    int num_levels, num_buttons;
    String name;
    int categorize_time;
    int replicas, num_replicas;
    int path[] = new int [num_levels];

    HierarchyPage(int num_levels, int num_buttons, String name, int categorize_time,
        int replicas, int num_replicas)
    {
        this.num_levels = num_levels;
        this.num_buttons = num_buttons;
        this.name = name;
        this.categorize_time = categorize_time;
        this.replicas = replicas;
        this.num_replicas = num_replicas;
    }

    // returns the position that corresponds to the page's path
    int getPosition()
    {
        int position=0;
        int temp_position=0, sum=0;
        for(int i=0;i<num_levels;i++)
        {
            if(path[i] != -1)
            {
                temp_position = temp_position*num_buttons + path[i];
                sum += (int)Math.pow((double)num_buttons,(double)(i));
            }
        }
        position = temp_position*sum;
        return position;
    }

    // returns the level of the page
    int getLevel()
    {
        int level=-1;
        for(int i=0;i<num_levels;i++)
        {
            if(path[i] == -1)
```

```

        {
            level = i-1;
            i=num_levels;
        }
    }
    return level;
}

// computes the page's path for the given index
// this method must be explicitly called
void setPathfromPosition(int position)
{
    // set default
    for(int i=0;i<num_levels;i++)
        path[i] = -1;
    int size[] = new int [num_levels];
    int level=0,sum=0;

    // compute range of indices for each level
    for(int i = 0;i<num_levels;i++)
    {
        size[i] = (int)Math.pow((double)num_buttons,(double)i);
        sum += size[i];
        if(position < sum)
        {
            level = i;
            i=num_levels;
        }
    }
    // work backwards through levels to find path
    for(int i=level;i>0;i--)
    {
        // find position in level i
        int temp_position = position - (sum-size[i]);
        path[i-1] = (int)(temp_position%num_buttons);
        // reset index as parent index
        sum=0;
        for(int j=0;j<i-1;j++)
            sum+=size[j];
        position = sum + (int)(temp_position/num_buttons);
    }
}

// this method sets the path by copying an array
void setPath(int[] temp)
{
    for(int i=0;i<num_levels;i++)
        path[i] = temp[i];
}
}

```

```
import java.awt.*;
import java.util.Random;
import java.util.Date;
import java.io.*;
```

```
/* class DisplayHierarchy
```

This class sets up a display screen with buttons and then specifies targets for the user to find in the hierarchy. Data on the hierarchy is read in from the file "data/items.txt". No output file is created.

Written by Greg Francis, Purdue University
August 1997

The views opinions, and/or findings contained in this report are those of the author and should not be construed as an official Department of the Army position, or decision, unless so designated by other documentation.

```
*/
```

```
public class DisplayHierarchy extends Frame
{
```

```
    Random randGen = new Random();
    int num_buttons, num_levels;
    int num_labels;
    Button b[];
    String target;
    Label search_for, counter;
    Button next_target;
    int level=0;
    boolean found_target=true;
    long start_time;
    int num_trials=0, trial =0;
    int bp_count=0;
    int [] path;
    HierarchyPage page[];
    int block =0;
    int num_blocks=1;
    int target_index;
```

```
    public DisplayHierarchy(String title)
    {
        super(title);
```

```
        //set size of SearchTime window
        this.resize(600,400);
```

```
        // Set font
        Font font = new Font("Helvetica",Font.PLAIN,24);
        setFont(font);
        //Create menubar
        MenuBar menubar= new MenuBar();
        this.setMenuBar(menubar);
        //Create file menu. Add Close.
        Menu file=new Menu("File");
        file.add(new MenuItem("Close"));
        menubar.add(file);
```

```
        Random randGen = new Random();
```

```
        get_data_from_file();
```

```
        // define range of path variable for later use
        path = new int [num_levels];
```

```

        b=new Button(num_buttons];
        for(int i=0;i<num_buttons;i++)
            b[i] = new Button("      ");

        //Establish panels for buttons and info
        Panel left = new Panel();
        Panel right = new Panel();

        left.setLayout(new GridLayout(num_buttons/2,1,10,20));
        for(int i=0; i<num_buttons/2;i++)
            left.add(b[i]);

        right.setLayout(new GridLayout(num_buttons/2,1,10,20));
        for(int i=num_buttons/2; i<num_buttons;i++)
            right.add(b[i]);

        this.setLayout(new BorderLayout(5,5));
        this.add("West",left);
        this.add("East",right);

        // Set up everything else on bottom panel
        Panel bottom= new Panel();
        bottom.setLayout(new GridLayout(2,1,10,20));
        search_for = new Label ("Press button to start  ");
        Panel next = new Panel();
        next.setLayout(new FlowLayout(FlowLayout.CENTER));
        next_target = new Button("Next target");
        next.add(next_target);
        counter = new Label (" "+(num_trials));
        bottom.add(next);
        bottom.add(counter);
        this.add("Center",search_for);
        this.add("South",bottom);
        this.pack();
        this.show();
    }

    public static void main(String args[])
    {

        DisplayHierarchy f = new DisplayHierarchy("Hierarchy");

    }

    /* This method handles all user interactions with the hierarchy.
    It changes button labels according to movement through the hierarchy.
    It notes when the target has been found.
    It calls methods for measuring various response times. */
    public boolean action ( Event e, Object arg)
    {
        if(e.target instanceof MenuItem)
        { //Watch for quit command
            String label = (String) arg;
            if(label.equals("Close"))
                dispose();
        }

        if(e.target instanceof Button)
        {
            String s= (String)arg;

            if (e.target == next_target && found_target)
            {
                if ( trial < num_trials)
                {
                    GetSearchItem();           // Get new target
                    found_target=false;
                }
                else // see if another block is needed
                {
                    block++;
                }
            }
        }
    }

```

```

        if(block==num_blocks)
        {
            search_for.setText("All done...Thanks!");
            write_to_file();
        }
        else if(block < num_blocks) // reset everything for the next block
        {
            search_for.setText("Next block...");
            for(int i=0;i<num_labels;i++)
            {
                page[i].replics = 0;
                page[i].categorize_time=0;
            }
            trial =0;
            found_target=true;
        }
    }
}

if( !s.equals(target))
{
    //If not target, act on pushed button to move through hierarchy
    for(int i =0;i<num_buttons;i++)
    if(e.target == b[i])
    {
        path[bp_count] = i;

        update_statistics_not_target(path);

        // Relabel buttons to emulate moving through the hierarchy
        bp_count++;
        // If go through bottom of hierarchy, reset to top page
        if(bp_count>=num_levels-1)
        {
            bp_count=0;
            for(int j=0;j<num_levels;j++)
                path[j] = -1;
        }

        buildDisplayPage(path, bp_count, b);

        return true;
    }
}
// If target, record between-item time and prepare for next item
else if(found_target) // ignore repeated pressings of target button
{
    for(int i =0;i<num_buttons;i++)
    if(e.target == b[i])
    {
        found_target=true;
        path[bp_count] = i;
        update_statistics_target(path);
        bp_count++;
        if(bp_count>=num_levels)
        {
            bp_count=0;
            for(int j=0;j<num_levels;j++)
                path[j] = -1;
        }
    }
    search_for.setText("Press 'Next target'");
}

return true;
}
}

```

// This method updates the statistics needed when there is a button press that

```

// does not result in the target being found
public void update_statistics_not_target(int[] path)
{

}

// This method updates statistics for a button press that
// results in the target being found.
public void update_statistics_target(int[] path)
{

}

// This method reads data from the input file
public void get_data_from_file()
{
    try
    {
        String filename = "data/items.txt";
        File f = new File(filename);
        FileInputStream labels = new FileInputStream(f);
        DataInputStream label_file = new DataInputStream(labels);

        // Get num_levels and num_buttons
        String s = label_file.readLine();
        //parse out needed info
        // get num_levels
        int end = s.indexOf(",");
        String temp = s.substring(0,end);
        num_levels = (int) Float.valueOf(temp).floatValue();

        // get num_buttons
        temp = s.substring(end+1);
        num_buttons = (int) Float.valueOf(temp).floatValue();

        // Compute number of labels in Hierarchy for later use
        for(int i=0;i<num_levels;i++)
            num_labels += (int) Math.pow((double) num_buttons, (double) i);

        // Create pages
        page = new HierarchyPage[num_labels];

        // get page info from data file
        for(int i=0;i<num_labels;i++)
        {
            s = label_file.readLine();

            //parse needed information
            // get name
            int name_end = s.indexOf(",");
            String name = s.substring(0,name_end);
            // get position
            int position_end = s.indexOf(",name_end+1);
            String s2 = s.substring(name_end+1,position_end);
            int position = (int) Float.valueOf(s2).floatValue();
            // get time (always equals 0)
            int time_end = s.indexOf(",position_end+2);
            String s3 = s.substring(position_end+2,time_end);
            int time = (int) Float.valueOf(s3).floatValue();

            //get replications
            int reps_end = s.lastIndexOf(",");
            s2 = s.substring(reps_end+1);
            int reps = (int) Float.valueOf(s2).floatValue();

            num_trials += reps; // keep track of how many trials there will be

            // set up page info
            page[i] = new HierarchyPage( num_levels, num_buttons, name, time,
                0, reps);
            page[i].setPathfromPosition(position);
        }
    }
}

```

```

    }
}
catch(Exception e)
{
    System.out.println("Error: "+e.toString());
}
}

// This method writes data to the output file
public void write_to_file()
{
}

// This method identifies the index of the page that has the specified path
// of button presses.
public int findPagewithPath(int[] path)
{
    int item=-1;

    for(int i=0;i<num_labels;i++)
    {
        boolean found_it= true;
        int check[] = new int [num_levels];
        check = page[i].path;
        for(int j=0;j<num_levels;j++)
        {
            if(check[j] != path[j])
            {
                found_it=false;
            }
        }
        if(found_it)
        {
            item = i;
            i=num_labels;
        }
    }
    return item;
}

// This method gets the next target item for the user to search for. It considers
// how often each item is to be searched.

public void GetSearchItem()
{
    // pick an item at random, but not item zero
    int item = (int)(Math.abs(randGen.nextInt())%(num_labels-1))+1;

    // make certain the item is to be searched for
    while(page[item].replics >= page[item].num_replics)
        item = (int)(Math.abs(randGen.nextInt())%(num_labels-1))+1;

    page[item].replics++;

    target = page[item].name;
    target_index = item;

    // get button-presses and level for selected page
    int temp[] = new int [num_levels];
    temp = page[item].path;
    level = page[item].getLevel();

    // If city, identify country to ease search
    String s="";
    if(level==2){
        // find page that corresponds to parent
        int parent_path[] = new int [num_levels];
    }
}

```

```

        for(int i=0;i<num_levels;i++)
            parent_path[i] = temp[i];
        parent_path[level] = -1;

        int parent_index = findPagewithPath(parent_path);
        s = ", "+page[parent_index].name;
    }

    trial++;
    search_for.setText(target+s);
    counter.setText(""+(num_trials-trial));

    for(int i=0;i<num_levels;i++)
        path[i] = -1;

    bp_count=0;
    buildDisplayPage(path,bp_count,b);
    Date now = new Date();
    start_time = now.getTime();    // start clock for first pair of button presses
}

// This method builds the display for the appropriate level and path taken
// by the user as he moves through the hierarchy

void buildDisplayPage(int [] path, int level, Button[] b)
{
    int [] temp = new int [num_levels];

    for(int i=0;i<num_levels;i++) // copy path to dummy array
        temp[i] = path[i];

    // find label for each button
    for(int k=0;k<num_buttons;k++)
    {
        temp[level] = k;
        b[k].setLabel(page[findPagewithPath(temp)].name);
    }
}
}

```

import DisplayHierarchy;

/* class Practice

This class sets up a display screen with buttons and then specifies targets for the user to find in the hierarchy. The hierarchical information is read in from the file "data/items.txt". It does not produce any output files.

**Written by Greg Francis, Purdue University
August 1997**

The views opinions, and/or findings contained in this report are those of the author and should not be construed as an official Department of the Army position, or decision, unless so designated by other documentation.

***/**

public class Practice extends DisplayHierarchy
{
 public Practice(String title)
 {
 super(title);
 }

 public static void main(String args[])
 {
 Setup stp = new Setup(); // Create mission scenario
 Practice f = new Practice("Practice");
 }
}

SearchTime

```
import java.awt.*;
import java.util.Random;
import java.util.Date;
import java.io.*;
import DisplayHierarchy;
```

```
/* class SearchTime
```

This class sets up a display screen with buttons and then specifies targets for the user to find in the hierarchy. The time between each pair of button presses (between-item time) is noted and coded by the label of the second button. Average between-item times are stored in the file "data/search_time.txt".

Written by Greg Francis, Purdue University
August 1997

The views opinions, and/or findings contained in this report are those of the author and should not be construed as an official Department of the Army position, or decision, unless so designated by other documentation.

```
*/
```

```
public class SearchTime extends DisplayHierarchy
```

```
{
```

```
    int [] replicas;
    int first_replics;
    int first_time;
```

```
    public SearchTime(String title)
```

```
    {
        super(title);
    }
```

```
    public static void main(String args[])
```

```
    {
        Setup stp = new Setup();    // create mission scenario
        SearchTime f = new SearchTime("Get between item time");
    }
```

```
    // This method updates the statistics needed when there is a button press that
    // does not result in the target being found
```

```
    // Overrides the method in super
```

```
    public void update_statistics_not_target(int[] path)
```

```
    {
```

```
        // verify that we are on the right path. Do not gather data for mistakes
```

```
        boolean error = false;
```

```
        for(int i=0;i<=bp_count;i++)
```

```
        {
```

```
            if(path[i] != page[target_index].path[i])
```

```
                error=true;
```

```
        }
```

```
        if(!error)
```

```
        {
```

```
            int index = findPagewithPath(path);
```

```
            if(index != -1)    // catch bug
```

```
            {
```

```
                Date now = new Date();
```

```
                // calculate between-item time
```

```
                page[index].categorize_time += (int)((long)now.getTime() - (long) start_time);
```

```
                start_time = (long)now.getTime();
```

```
                if(block == num_blocks-1) // keep track of encounters on last block
```

```
                    replics[index]++;
```

```
                System.out.println(page[index].name+" "+page[index].categorize_time+" "+replics[index]);
```

```
                // update estimates for first time encounters with an item
```

```

        if(block==0 && replicas[index]==1)
        {
            first_time += page[index].categorize_time;
            first_replica++;
        }
    }
}

// This method updates statistics for a button press that
// results in the target being found.
// Overrides method in super
public void update_statistics_target(int[] path)
{
    Date now = new Date();
    int index = findPagewithPath(path);
    if(index != -1)
    {
        page[index].categorize_time += (int)((long)now.getTime() - (long) start_time);
        start_time = (long)now.getTime();
        if(block == num_blocks-1) // keep track of encounters on last block
            replicas[index]++;

        System.out.println(page[index].name+" "+page[index].categorize_time+" "+replicas[index]);
        // update estimates for first encounters with an item
        if(block==0 && replicas[index]==1)
        {
            first_time += page[index].categorize_time;
            first_replica++;
        }
    }
}

// This method reads data from the input file
// Overrides method in super
public void get_data_from_file()
{
    num_blocks = 2; // overrides default in super
    try
    {
        String filename = "data/items.txt";
        File f = new File(filename);
        FileInputStream labels = new FileInputStream(f);
        DataInputStream label_file = new DataInputStream(labels);

        // Get num_levels and num_buttons
        String s = label_file.readLine();
        //parse out needed info
        // get num_levels
        int end = s.indexOf(",");
        String temp = s.substring(0,end);
        num_levels = (int) Float.valueOf(temp).floatValue();

        // get num_buttons
        temp = s.substring(end+1);
        num_buttons = (int) Float.valueOf(temp).floatValue();

        // Compute number of labels in Hierarchy for later use
        for(int i=0;i<num_levels;i++)
            num_labels += (int) Math.pow((double)num_buttons,(double)i);

        // Create pages
        page = new HierarchyPage[num_labels];
        replicas = new int[num_labels]; // a counter for how often each page is encountered

        // get page info from data file
        for(int i=0;i<num_labels;i++)
        {
            s = label_file.readLine();

```

```

        //parse needed information
        // get name
        int name_end = s.indexOf(",");
        String name = s.substring(0,name_end);
        // get position
        int position_end = s.indexOf(", ",name_end+1);
        String s2 = s.substring(name_end+1,position_end);
        int position = (int) Float.valueOf(s2).floatValue();
        // get time (always equals 0)
        int time_end = s.indexOf(", ",position_end+2);
        String s3 = s.substring(position_end+2,time_end);
        int time = (int) Float.valueOf(s3).floatValue();

        //get replications
        int reps_end = s.lastIndexOf(",");
        s2 = s.substring(reps_end+1);
        int reps = (int) Float.valueOf(s2).floatValue();

        num_trials += reps; // keep track of how many trials there will be

        // set up page info
        page[i] = new HierarchyPage( num_levels, num_buttons, name, time,
        0, reps);
        page[i].setPathfromPosition(position);
    }
}
catch(Exception e)
{
    System.out.println("Error: "+e.toString());
}
}

// This method writes data to the output file
// Overrides method in super
public void write_to_file()
{
    // Open output file
    try
    {
        FileOutputStream out_file;
        out_file = new FileOutputStream("data/search_time.txt");
        PrintStream out2 = new PrintStream(out_file);

        // Write info to data file
        StringBuffer sb1 = new StringBuffer(num_levels+" "+num_buttons);
        out2.println(sb1.toString());
        int current_est=0;
        System.out.println(first_time+" "+first_reps);
        for(int i=0;i<num_labels;i++)
        {
            current_est += page[i].categorize_time;
            if(reps[i]==0)
            {
                reps[i] = first_reps;
                page[i].categorize_time = first_time;
            }
            StringBuffer sb = new StringBuffer(page[i].name+" "+page[i].getPosition()+
            ", "+(int)((double)page[i].categorize_time/(double)reps[i])+" ", "+
            page[i].num_reps);
            out2.println(sb.toString());
        }
        StringBuffer sb3 = new StringBuffer("Total search time = "+current_est);
        out2.println(sb3.toString());
    }
    catch(Exception e)
    {
        System.out.println("Error: "+e.toString());
    }
}
}

```

```
import java.awt.*;
import java.util.Random;
import java.util.Date;
import java.io.*;
import DisplayHierarchy;
```

```
/* class MotorTime
```

This class sets up a display screen with buttons and then specifies movements for the user. The time to complete each movement is noted. The resulting data is stored in the file "data/motor.txt".

Written by Greg Francis, Purdue University
August 1997

The views opinions, and/or findings contained in this report are those of the author and should not be construed as an official Department of the Army position, or decision, unless so designated by other documentation.

```
*/
```

```
public class MotorTime extends DisplayHierarchy
{
    String target1, target2="Next target";
    int button1,button2;
    boolean found_target1=true;
    int reps[][];
    int time[][];
    int stat[][];
    int num_reps;

    public MotorTime(String title)
    {
        super(title);
    }

    public static void main(String args[]) {
        Frame f = new MotorTime("Motor time data");
    }
}
```

```
// This method handles all user interactions with the button presses.
// It notes the time to make the first button press, the second button press, and
// it stores the difference in an array for statistics.
public boolean action ( Event e, Object arg)
{
```

```
    String s="";
    if(e.target instanceof Button)
        s = (String) arg;

    if( s.equals(target1) && !found_target1)
    {
        // If first target is selected
        Date now = new Date();
        start_time = (long)now.getTime(); // note time
        found_target1=true;
        return true;
    }
}
```

```
if( s.equals(target2) && found_target1)
{ // If found second target after finding first
    Date now = new Date();
```

```

        // Update statistics
        if(trial>0)
        { //but not for first press of "Next Target" button
            // Save search time in array for later statistical computation
            time[button1][button2] += (int)((long)now.getTime() - (long) start_time);
            stat[button1][button2] += (int)Math.pow((double)((long)now.getTime() -
                (long) start_time),2); // to calculate variance
        }

        //When all done write statistics to file
        if(trial == num_trials)
            write_to_file();

        if ( trial < num_trials)
            GetSearchPattern(); // Get new movement pattern
        else
            search_for.setText("All done...Thanks!");
        found_target1=false;
    }

    if(e.target instanceof JMenuItem){ //Watch for close command
        String label = (String) arg;
        if(label.equals("Close"))dispose();
    }
    return false;
}

// This methods sets up everything. It takes its name from the super, where information
// is read in from a file. There is no file-input here.
public void get_data_from_file()
{
    num_buttons =8;
    num_levels =1;

    // Compute number of labels in heirarchy for later use
    for(int i=0;i<=num_levels;i++)
        num_labels += (int)Math.pow((double)num_buttons,(double)i);

    // create variables for statistical calculations
    reps = new int [num_buttons+1][num_buttons+1];
    time = new int [num_buttons+1][num_buttons+1];
    stat = new int [num_buttons+1][num_buttons+1];
    num_reps=3;
    trial= 0;
    num_trials=num_reps*(num_buttons+1)*(num_buttons);

    // Do not need movements ending in "Next Target", so fill repic counter for
    // those movements
    for(int k=0;k<num_buttons+1;k++)
        reps[k][num_buttons] = num_reps;
}

// This method writes statistics on motor time to a file
public void write_to_file()
{
    // open output file
    try
    {
        FileOutputStream out_file;
        out_file = new FileOutputStream("data/motor.txt");
        PrintStream out2 = new PrintStream(out_file);

        for(int i=0;i<num_buttons+1;i++) // go through every pair of button pushes
            for(int j=0;j<num_buttons+1;j++)
            {
                double average = 0.0;
                double st_dev = 0.0;
            }
    }
}

```

```

        // Calculate average
        average = (double)time[i][j]/(double) num_replics;

        // Calculate standard deviation
        if(num_replics>1)
            st_dev = Math.sqrt(((double)stat[i][j] -
                (double)num_replics*Math.pow(average,2))/(double)(num_replics-1));

        // Write to file the page index, average, standard deviation
        StringBuffer sb5 = new StringBuffer(i+" "+j+" "+average+" "+
            st_dev);
        // Open output file

        out2.println(sb5.toString());
    }
}
catch(Exception ex)
{
    System.out.println("Error: "+ex.toString());
}
}

// This method identifies a pair of button pushes for the user to make
// It takes into account how many times each pair must be performed
public void GetSearchPattern ()
{
    // Get new pair
    button1 = (int)(Math.abs(randGen.nextInt())%(num_buttons+1));
    button2 = (int)(Math.abs(randGen.nextInt())%(num_buttons+1));

    while(replics[button1][button2]>=num_replics)
    {
        button1 = (int)(Math.abs(randGen.nextInt())%(num_buttons+1));
        button2 = (int)(Math.abs(randGen.nextInt())%(num_buttons+1));
    }

    replics[button1][button2]++; // Update replication count for selected pair
    trial++; // Update trial counter

    // specify targets
    if(button1 == num_buttons) // button 8 is "Next target"
        target1 = "Next target";
    else
        target1 = ""+button1;

    if(button2 == num_buttons)
        target2 = "Next target";
    else
        target2 = ""+button2;

    // put labels on buttons
    for(int j=0;j<num_buttons;j++)
        b[j].setLabel(""+j);

    search_for.setText(target1+" -> "+target2);
    counter.setText(""+(num_trials-trial));
}
}

```

TestHierarchy

```
import java.util.Date;
import java.io.*;
import DisplayHierarchy;
```

```
/* class TestHierarchy
```

This class reads in hierarchy data from "data/optimal.txt" and creates a display with buttons to allow the user to search for items in the mission scenario. The total search time is computed and the mean search time is written to the file "data/mean_time.txt".

Written by Greg Francis, Purdue University
August 1997

The views opinions, and/or findings contained in this report are those of the author and should not be construed as an official Department of the Army position, or decision, unless so designated by other documentation.

```
*/
```

```
public class TestHierarchy extends DisplayHierarchy
{
    int original_est;
    int est;

    public TestHierarchy(String title)
    {
        super(title);
    }

    public static void main(String args[])
    {
        TestHierarchy f = new TestHierarchy("Get between item time");
    }

    // This method updates the statistics needed when there is a button press that
    // does not result in the target being found
    // Overrides the method in super
    public void update_statistics_not_target(int[] path)
    {
    }

    // This method updates statistics for a button press that
    // results in the target being found.
    // Overrides method in super
    public void update_statistics_target(int[] path)
    {
        Date now = new Date();
        int index = findPagewithPath(path);
        if(index != -1)
            page[index].categorize_time += (int)((long)now.getTime() - (long) start_time);
    }

    // This method reads data from the input file
    // Overrides method in super
    public void get_data_from_file()
    {
        num_blocks = 2;    // overrides default in super
        try
        {
            String filename = "data/optimal.txt";
            File f = new File(filename);
            FileInputStream labels = new FileInputStream(f);
        }
    }
}
```

```

DataStream label_file = new DataInputStream(labels);

// Get num_levels and num_buttons
String s = label_file.readLine();

//parse out info
// get num_levels
int end = s.indexOf(",");
String temp = s.substring(0,end);
num_levels = (int) Float.valueOf(temp).floatValue();

// get num_buttons
temp = s.substring(end+1);
num_buttons = (int) Float.valueOf(temp).floatValue();

// Compute number of labels in Hierarchy for later use
for(int i=0;i<num_levels;i++)
    num_labels += (int) Math.pow(((double)num_buttons, (double)i));

// Create pages
page = new HierarchyPage(num_labels);
path = new int [num_levels];
// get page info from data file
for(int i=0;i<num_labels;i++)
{
    s = label_file.readLine();
    //parse needed information
    // get name
    int name_end = s.indexOf(",");
    String name = s.substring(0,name_end);
    // get position
    int position_end = s.indexOf(", ",name_end+1);
    String s2 = s.substring(name_end+1,position_end);
    int position = (int) Float.valueOf(s2).floatValue();
    // get time
    int time_end = s.indexOf(" ",position_end+2);
    String s3 = s.substring(position_end+2,time_end);
    int time = (int) Float.valueOf(s3).floatValue();
    //get replications
    int repe_end = s.lastIndexOf(",");
    s2 = s.substring(repe_end+1);
    int repe = (int) Float.valueOf(s2).floatValue();

    num_trials += repe;    // keep track of how many trials

    // set up page info
    page[i] = new HierarchyPage( num_levels, num_buttons, name, time,
        0, repe);
    page[i].setPathfromPosition(position);
}

// Get original search time    for random hierarchy
s = label_file.readLine();
end = s.lastIndexOf(" ");
temp = s.substring(end+1);
original_est = (int) Float.valueOf(temp).floatValue();

// Get predicted search time for optimal hierarchy
s = label_file.readLine();
end = s.lastIndexOf(" ");
temp = s.substring(end+1);
est = (int) Float.valueOf(temp).floatValue();
}
catch(Exception e)
{
    System.out.println("Error: "+e.toString());
}
}

// This method writes data to the output file

```

```

// Overrides method in super
public void write_to_file()
{
    // Open output file
    try
    {
        FileOutputStream out_file;
        out_file = new FileOutputStream("data/mean_time.txt");
        PrintStream out2 = new PrintStream(out_file);

        // Write info to data file
        // add up actual search times, gathered with this program
        int actual_est=0;
        for(int i=0;i<num_labels;i++)
            actual_est += page[i].categorize_time;

        StringBuffer sb3 = new StringBuffer("In seconds/item -- Original search time = "+
            ((double)(original_est/num_trials)/1000)+
            "\nExpected search time = "+((double)(est/num_trials)/1000)+
            "\nActual search time = "+((double)(actual_est/num_trials)/1000));
        out2.println(sb3.toString());
    }
    catch(Exception e)
    {
        System.out.println("Error: "+e.toString());
    }
}
}

```

```
import java.util.Random;
import java.util.Date;
import java.io.*;
```

```
/* class Setup
```

This class reads in user-supplied data files that define a hierarchical structure. It re-orders the hierarchical structure, defines a mission scenario, and writes all the information to the file "data/items.txt".

Written by Greg Francis, Purdue University
August 1997

The views opinions, and/or findings contained in this report are those of the author and should not be construed as an official Department of the Army position, or decision, unless so designated by other documentation.

```
*/
```

```
public class Setup
```

```
{
    int num_buttons, num_levels;
    int num_labels;
```

```
    HierarchyPage page[];
```

```
    public Setup()
```

```
    {
        get_data_from_file();

        create_new_hierarchy();
    }
```

```
    public static void main(String args[])
```

```
    {
        Setup f = new Setup();
    }
```

/* This method reads in data from specified files. Names of items in the first level are in a file called "item_names/level0.txt". The names of items in the subsequent levels are in files with filenames "item_names/<item_name>.txt", where <item_name> is given in the level0.txt file. */

```
    public void get_data_from_file()
```

```
    {
        // Get info from category data files
        try
        {
            String filename = "item_names/level0.txt";
            File f = new File(filename);
            FileInputStream labels = new FileInputStream(f);
            DataInputStream label_file = new DataInputStream(labels);

            // Get num_levels and num_buttons
            String s = label_file.readLine();

            //parse out info

            // get num_levels
            int end = s.indexOf(",");
            String temp = s.substring(0,end);
            num_levels = (int) Float.valueOf(temp).floatValue();
            // get num_buttons
            temp = s.substring(end+1);
            num_buttons = (int) Float.valueOf(temp).floatValue();
        }
    }
```

```

// Compute number of labels in Hierarchy for later use
for(int i=0;i<num_levels;i++)
    num_labels += (int)Math.pow((double)num_buttons,(double)i);

// Create pages
page = new HierarchyPage[num_labels];

for(int i=0;i<num_labels;i++)
{
    page[i] = new HierarchyPage( num_levels, num_buttons, "", 0, 0, 0);
    page[i].setPathfromPosition(0);
}

// Load in item names and position in Hierarchy, first level only
System.out.println("Loading names from file.");
for(int j=1;j<num_buttons+1;j++)
{
    s = label_file.readLine();
    // set paths
    int [] path = new int[num_levels];
    path = page[j].path;
    path[0] = j-1;

    page[j].name = s.trim();
}
}
catch(Exception e)
{
    System.out.println("Error: "+e.toString());
}

// Now load in other items using name from parent as filename
for(int item=1;item<num_labels;item++) //cycle through all pages, except top, which is nothing
{

    int level= page[item].getLevel();

    if(level < num_levels-2 && level != -1)    // set children of parent
    {
        // get path for parent page
        int[] temp = new int[num_levels];
        temp = page[item].path;

        String s = page[item].name;
        if(s.length()>1) // do not look for file if filename is blank
        {
            try
            {
                FileInputStream labels = new FileInputStream("Item_names/"+s+".txt");
                DataInputStream label_file = new DataInputStream(labels);
                for(int j=0;j<num_buttons;j++)
                {

                    String s1=label_file.readLine()+" ";

                    if(s1.length()>0 && !s1.equals("null"))
                        s1 = s1.trim();
                    else
                        s1 = " ";

                    //build path for child page
                    int[] temp1 = new int[num_levels];
                    for(int k=0;k<num_levels;k++)
                        temp1[k] = temp[k];
                    temp1[level+1] = j;

                    // find next unused page and put item there
                    for(int m=1;m<num_labels;m++)
                    {
                        int [] int2 = new int[num_levels];

```

```
/* This method scrambles the order of the hierarchy so it is partially randomized.
It also defines the mission scenario.*/
public void create_new_hierarchy()
```

```

{
    Random randGen = new Random();

    // Now scramble paths to make a random hierarchy
    System.out.println("Randomizing hierarchy");
    for(int item=1; item<num_labels; item++)
    {
        // get button-presses and level for selected page
        int temp[] = new int [num_levels];

        temp = page[item].path;
        int level = page[item].getLevel();

        //now pick a new button from the same branch in the hierarchy

        int new_btn = temp[level];
        while(new_btn == temp[level])
            new_btn = (int)(Math.abs(randGen.nextInt())%num_buttons);

        // identify path for new item
        int new_temp[] = new int [num_levels];
        for(int i=0; i<num_levels; i++)
            new_temp[i] = temp[i];
        new_temp[level] = new_btn;

        // now find the item with this path
        int new_item = findPagewithPath(new_temp);
        swap(level, item, new_item);
    }

    // Fix naming problem (bug-fix)
    for(int i=0; i<num_labels; i++)
    {
        if(page[i].name.equals(" "))
            page[i].name="";
    }

    // Now set up replications
    System.out.println("Creating mission scenario");
    //System.out.println("Setting up replication data.");
    // 7 items with corresponding replics
    for(int i=1; i<=7; i++)
    {
        int item = (int)(Math.abs(randGen.nextInt())%(num_labels-1))+1;
        while(page[item].num_replics !=0 || page[item].name.length()<1)
            item = (int)(Math.abs(randGen.nextInt())%(num_labels-1))+1;
        page[item].num_replics = i;
    }

    // Write data to file
    write_to_file();
}

/* This method writes the created hierarchy to the file "data/items.txt". */
public void write_to_file()
{
    // Open output file
    try
    {
        FileOutputStream out_file;
        out_file = new FileOutputStream("data/items.txt");
        PrintStream out2 = new PrintStream(out_file);

        // write new Hierarchy to file
        StringBuffer sb1 = new StringBuffer(num_levels+"", "+num_buttons);
        out2.println(sb1.toString());
        for(int i=0; i<num_labels; i++)
        {

```

```

        StringBuffer sb = new StringBuffer(page[i].name+" "+page[i].getPosition()+
            ", "+page[i].categorize_time+" "+page[i].num_replicas);
        out2.println(sb.toString());
    }

    StringBuffer sb3 = new StringBuffer("Original search time = "+0+"\n"+
        "Estimated new search time = "+0);
    out2.println(sb3.toString());
}
catch(Exception e)
{
    System.out.println("Error: "+e.toString());
}

}

// This method takes a vector describing a path of button pushes and returns the index
// of the item with that path.
public int findPagewithPath(int[] path)
{
    int item=-1;

    for(int i=0;i<num_labels;i++) // go through all items in the hierarchy
    {
        boolean found_it= true;
        int check[] = new int [num_levels];
        check = page[i].path;
        for(int j=0;j<num_levels;j++) // go through path of each item to see if it matches
        {
            if(check[j] != path[j])
            {
                found_it=false;
            }
        }
        if(found_it)
        {
            item = i;
            i=num_labels;
        }
    }
    return item;
}

// This method swaps the hierarchical positions of two items. It is
// complicated because it also has to swap all the children of those items.
public void swap(int level,int item, int new_item)
{
    int temp[] = new int[num_levels];
    int new_temp[] = new int[num_levels];
    int check[][] = new int [num_labels][];

    for(int i=0;i<num_labels;i++)
        check[i] = new int[num_levels];

    temp = page[item].path;
    new_temp = page[new_item].path;

    // first copy all paths into a large array with changes made
    for(int i=0;i<num_labels;i++)
    {
        boolean change1=true, change2=true;

        int hold[] = new int[num_levels];

        hold = page[i].path;

        for(int j=0;j<num_levels;j++)
            check[i][j] = hold[j];
    }
}

```

```

for(int j=0;j<=level;j++)
{
    // check to see if path of current item includes the to-be-swapped paths
    if(check[i][j] != temp[j])
        change1=false;
    if(check[i][j] != new_temp[j])
        change2=false;
}
// making changes in path of current item (if necessary)
if(change1)
    check[i][level] = new_temp[level];
if(change2)
    check[i][level] = temp[level];
}

// now copy everything back with changes included
for(int i=0;i<num_labels;i++)
    page[i].setPath(check[i]);
}
// end of Setup class
}

```

```

import java.util.Random;
import java.util.Date;
import java.io.*;
import Setup;

```

```

/* class Optimize

```

This class reads in data from "data/motor.txt" and "data/search_time.txt" and converts that data into separate motor and categorization times. It then modifies the hierarchy from "data/search_time.txt" into one that minimizes predicted search time. The final hierarchy is stored in the file "data/optimal.txt".

Written by Greg Francis, Purdue University
August 1997

The views opinions, and/or findings contained in this report are those of the author and should not be construed as an official Department of the Army position, or decision, unless so designated by other documentation.

```

*/

```

```

public class Optimize extends Setup
{

```

```

    int motor_time[][];
    int est=0, original_est, current_est, local_minima;

```

```

    public Optimize()
    {
        super();
    }

```

```

    public static void main(String args[])
    {
        Optimize f = new Optimize();
    }

```

/* This method reads in data from files. It reads in between-item search times and hierarchical layout from the file "data/search_time.txt" and it reads motor times from the file "data/motor.txt".

overrides method in super */

```

    public void get_data_from_file()
    {
        // Get info from category data files
        try
        {
            String filename = "data/search_time.txt";
            File f = new File(filename);
            FileInputStream labels = new FileInputStream(f);
            DataInputStream label_file = new DataInputStream(labels);

            // Get num_levels and num_buttons
            String s = label_file.readLine();

            //parse out info
            // get num_levels
            int end = s.indexOf(",");
            String temp = s.substring(0,end);
            num_levels = (int) Float.valueOf(temp).floatValue();

            // get num_buttons
            temp = s.substring(end+1);
            num_buttons = (int) Float.valueOf(temp).floatValue();

```

```

// Compute number of labels in Hierarchy for later use
for(int i=0;i<num_levels;i++)
    num_labels += (int) Math.pow((double) num_buttons, (double) i);
// Create pages
page = new HierarchyPage[num_labels];
// Create array for motor_time
motor_time = new int[num_buttons+1][num_buttons+1];
// get page info from data file
for(int i=0;i<num_labels;i++)
{
    s = label_file.readLine();

    //parse needed information
    // get name
    int name_end = s.indexOf(",");
    String name = s.substring(0,name_end);
    // get position
    int position_end = s.indexOf(", ",name_end+1);
    String s2 = s.substring(name_end+1,position_end);
    int position = (int) Float.valueOf(s2).floatValue();
    // get between-item time
    int time_end = s.indexOf(", ",position_end+2);
    String s3 = s.substring(position_end+2,time_end);
    int time = (int) Float.valueOf(s3).floatValue();
    //get replications
    int reps_end = s.lastIndexOf(",");
    s2 = s.substring(reps_end+1);
    int reps = (int) Float.valueOf(s2).floatValue();

    // set up page info
    page[i] = new HierarchyPage( num_levels, num_buttons, name, time,
                                0, reps);
    page[i].setPathfromPosition(position);
}
System.out.println("Loaded page information.");

// Get actual search time from file
s = label_file.readLine();
end = s.lastIndexOf(" ");
temp = s.substring(end+1);
ast = (int) Float.valueOf(temp).floatValue();
}
catch(Exception e)
{
    System.out.println("Error: "+e.toString());
}
System.out.println("Loaded between-item times..");

System.out.println("Estimated st = "+ compute_expected_search_time());

System.out.println("Loading motor data.");

// Get motor parameters from data file
motor_time = new int [num_buttons+1][num_buttons+1];
try
{
    String filename = "data/motor.txt";
    File f = new File(filename);
    FileInputStream motor_pf = new FileInputStream(f);
    DataInputStream motor_pd = new DataInputStream(motor_pf);

    for(int i=0;i<num_buttons+1;i++)
        for(int j=0;j<num_buttons+1;j++)
        {
            float time;
            double stdev;
            String s = motor_pd.readLine();

            // Parse information out of string
            // First 5 spaces are button codes and white space

```

```

        // need to find next white space to identify end of time integer
        int end = s.indexOf(" ", 5);
        String s2 = s.substring(4, end);
        time = Float.valueOf(s2).floatValue();
        motor_time[i][j] = (int) time;
    }
    System.out.println("Loaded motor information.");
}
catch(Exception e)
{
    System.out.println("Error: " + e.toString());
}
}

/* This method writes a hierarchy to the file "data/optimal.txt".
overrides method in super */
public void write_to_file()
{
    // Open output file
    try
    {
        PrintStream out2;
        FileOutputStream out_file;
        out_file = new FileOutputStream("data/optimal.txt");
        out2 = new PrintStream(out_file);

        StringBuffer sb1 = new StringBuffer(num_levels + ", " + num_buttons);
        out2.println(sb1.toString());
        for(int i=0; i<num_labels; i++)
        {
            StringBuffer sb = new StringBuffer(page[i].name + ", " + page[i].getPosition() +
                ", " + page[i].categorize_time + ", " + page[i].num_replica);
            out2.println(sb.toString());
        }
        StringBuffer sb3 = new StringBuffer("Original search time = " + original_est + "\n" +
            "Estimated new search time = " + current_est);
        out2.println(sb3.toString());
    }
    catch(Exception e)
    {
        System.out.println("Error: " + e.toString());
    }
}

/* This method creates a new hierarchy that minimizes predicted search time.
overrides method in super */
public void create_new_hierarchy()
{
    Random randGen = new Random();
    // Convert search_time data into categorization_time data by compensating for
    // motor times

    System.out.println("Converting between-item time data into categorization_time data.");

    for(int i=1; i<num_labels; i++)
    {
        int [] path = new int[num_levels];
        path = page[i].path;
        int level = page[i].getLevel();
        int ctime=0;
        if(level>0)
            ctime = page[i].categorize_time - motor_time[path[level-1]][path[level]];
        else //movement from "Next item" button
            ctime = page[i].categorize_time - motor_time[0][path[level]];
        page[i].categorize_time = ctime;
    }

    original_est = compute_expected_search_time();
    current_est = original_est;
    local_minima = original_est;
}

```

```

System.out.println("Actual search time = "+ast+"\nEstimated search time= "+original_est);

// actual search time (ast) and estimated search time (original_est) may differ
// due to rounding errors

// initialize simulated annealing parameters
int temp_count=0;
double temperature_init = (double)ast;
double temperature;

boolean found_local_minima = false;
// randomize Hierarchy
System.out.println("Looking for a new local minima.\nRandomizing hierarchy...");
for(int item=1;item<num_labels;item++)
{
    // for each item, swap its position with another from the same branch
    int level = page[item].getLevel();

    //now pick a new button from the same branch in the hierarchy
    int new_btn = page[item].path[level];
    while(new_btn == page[item].path[level])
        new_btn = (int)(Math.abs(randGen.nextInt())%num_buttons);

    // identify path for new item
    int new_temp[] = new int [num_levels];
    for(int i=0;i<num_levels;i++)
        new_temp[i] = page[item].path[i];
    new_temp[level] = new_btn;

    // now find the item with this path
    int new_item = findPagewithPath(new_temp);
    swap(level,item,new_item);
}

// paths scrambled, compute expected search time for scrambled hierarch
current_est = compute_expected_search_time();

// now keep going through search process until it finds a local minimum
while(!found_local_minima)
{
    System.out.println("Looking for local minima...");
    // Now make changes to the Hierarchy
    int count =0;
    while(count < num_labels)
    {
        // update simulated annealing parameters
        temp_count++;
        temperature = temperature_init/(2+0.0005*((double)temp_count));

        // pick an item at random, but not item zero (it has no label)
        int item = (int)(Math.abs(randGen.nextInt())%(num_labels-1))+1;

        // get button-presses for selected page
        int level = page[item].getLevel();

        //now pick a new button from the same tree in the hierarchy
        int new_btn = page[item].path[level];
        while(new_btn == page[item].path[level])
            new_btn = (int)(Math.abs(randGen.nextInt())%num_buttons);

        // identify path for new item
        int new_temp[] = new int [num_levels];
        for(int i=0;i<num_levels;i++)
            new_temp[i] = page[item].path[i];
        new_temp[level] = new_btn;

        // now find the item with this path
        int new_item=findPagewithPath(new_temp);

        // Now swap items
        swap(level,item,new_item);
    }
}

```

```

// If search time does not improve, swap back with probability
int swap_est = compute_expected_search_time();

// compute simulated annealing probability
double sa_prob = Math.exp(-(double)swap_est/temperature);
sa_prob = sa_prob/(1+sa_prob);

// pick a random number between 0 and 1
double prob = randGen.nextDouble();
// keep swap if search time decreases, or random number less than sa_prob
if( swap_est < current_est || sa_prob >= prob)
{
    System.out.println(current_est+" "+swap_est+" count="+count
        +" temp "+temperature+" "+temp_count+" "+sa_prob+" "+prob);
    count=0; // reset counter of non-kept swaps
    current_est = swap_est; // update current search time
}
else // swap back
{
    count++; // update counter of non-kept swaps
    //swap back if search time is worse; if the same, don't bother
    if(swap_est > current_est)
        swap(level,item,new_item);
}

// write new Hierarchy to file if it is the best found so far
if(current_est < local_minima && temp_count>num_labels)
{
    local_minima = current_est;
    System.out.println("Writing current best Hierarchy to file 'best.txt'.\n"+
        " Estimated search time = "+current_est);
    write_to_file();
}
}
// after enough ineffective swaps (count > num_labels)
// verify that we have a local minima, otherwise, keep making swaps
found_local_minima = local_minima_check(current_est);
}
}

```

/* This method verifies that the current hierarchical layout is a local minima, meaning that swapping a single item to any other possible position in its branch would not decrease the predicted search time. */

```

public boolean local_minima_check(int current_est)
{
    System.out.println("Verifying that we have a local minimum...");
    // go through all items
    for(int item=0; item<num_labels; item++)
        if(page[item].num_replicas > 0) // make certain the item is looked at
        {
            // get button-presses and level for selected page
            int temp[] = new int [num_levels];
            temp = page[item].path;
            int level = page[item].getLevel();

            //now go through all buttons from the same branch in the hierarchy
            for(int new_btn = 0; new_btn<num_buttons; new_btn++)
            {
                // build path for swap-to button
                int new_temp[] = new int [num_levels];
                for(int i=0; i<num_levels; i++)
                    new_temp[i] = temp[i];
                new_temp[level] = new_btn;

                // now find the item with this path
                int new_item=findPagewithPath(new_temp);

                swap(level,item,new_item);
            }
        }
}

```

```

        // if search time does not improve, swap back
        int swap_est = compute_expected_search_time();
        if( swap_est < current_est)    // keep swap
        {
            current_est = swap_est;
            System.out.println("This was not a local minimum.");
            return false; // exit and tell the optimization procedure to keep looking
        }
        else // swap back
        {
            swap(level,item,new_item);
        }
    }
}

// if no swap improved predicted search time, the current layout must be a
// local minimum
System.out.println("This was a local minimum.");
return true;
}

/* This method takes the current hierarchical layout, runs through all the
items in the mission scenario, and computes the predicted search time
for finding all the items.
It does this by noting the path and items the user must follow and categorize
to find the target items. It then adds up motor and categorization times
as appropriate. */
public int compute_expected_search_time()
{
    int est=0;

    // cycle through all items in mission scenario
    for(int i=0;i<num_labels;i++)
    {
        if(page[i].num_replicas >0)
        {
            int search_time = 0;
            int temp[] = new int[num_levels];
            temp = page[i].path;

            // go through path of button presses
            // always starts with "Next Target" (button 8)
            search_time += motor_time[8][temp[0]];
            for(int k=0;k<num_levels-1;k++)
                if(temp[k]!=-1)
                {
                    if(temp[k+1]!=-1)
                        search_time += motor_time[temp[k]][temp[k+1]];
                    // build vector for current movement through path
                    int new_temp[] = new int [num_levels];
                    for(int k2 =0;k2<num_levels;k2++)
                    {
                        if(k2<=k)
                            new_temp[k2] = temp[k2];
                        else
                            new_temp[k2] = -1;
                    }
                    // now find the item with this current path and add its categorization
                    // time to search time for target item
                    int new_item=findPagewithPath(new_temp);
                    search_time += page[new_item].categorize_time;
                }
            // multiply search time for item by number of times it is searched for
            est += search_time*page[i].num_replicas;
        }
    }

    return est;
}
}

```

BuildOptimalHierarchy

```
import java.awt.*;
import java.io.*;
```

```
/* class BuildOptimalHierarchy
```

This class provides an interface to go through the steps needed to build an optimal hierarchy. It calls, in correct sequence other programs that gather needed data and compute the optimization algorithm.

Written by Greg Francis, Purdue University
August 1997

The views opinions, and/or findings contained in this report are those of the author and should not be construed as an official Department of the Army position, or decision, unless so designated by other documentation.

```
*/
```

```
public class BuildOptimalHierarchy extends Frame
{
    Button b[];

    public BuildOptimalHierarchy(String title)
    {
        // set up frame with option buttons

        super(title);

        //set size of display
        this.resize(600,400);

        // Set font
        Font font = new Font("Helvetica",Font.PLAIN,24);
        setFont(font);

        //Create menubar
        MenuBar menubar = new MenuBar();
        Menu file;

        this.setMenuBar(menubar);
        //Create file menu. Add Quit.
        file=new Menu("File");
        file.add(new MenuItem("Quit"));
        menubar.add(file);

        // create buttons
        b = new Button [5];
        b[0] = new Button("Practice");
        b[1] = new Button("Motor data");
        b[2] = new Button("Hierarchy search");
        b[3] = new Button("Optimization");
        b[4] = new Button("Testing");

        setLayout(new GridLayout(5,1,10,20));
        for(int i=0;i<5;i++)
        {
            add(b[i]);
            b[i].disable();
        }
        b[0].enable();

        this.pack();
        this.show();
    }
}
```

```

    }

    public static void main(String args[])
    {

        BuildOptimalHierarchy cmb = new BuildOptimalHierarchy("Build Optimal Hierarchy");

    }

    public boolean action ( Event e, Object arg)
    {
        if(e.target instanceof MenuItem)
        { //Watch for quit command
            String label = (String) arg;
            if(label.equals("Quit"))System.exit(0);
        }

        if(e.target instanceof Button)
        {
            if (e.target == b[0]) // User goes through practice trials
            {
                b[0].setLabel("Just a second...");
                for(int i=0;i<5;i++)
                    b[i].disable();

                Setup stp = new Setup();
                b[0].setLabel("Practice");
                Practice st = new Practice("Practice");

                // enable button for next step
                b[1].enable();
                return true;
            }
            else if (e.target == b[1]) // User gathers motor time data
            {
                // check to see if motor time data is already gathered
                // if not, gather it
                File f = new File ("data","motor.txt");
                if(!f.exists())
                    MotorTime motor = new MotorTime("Motor data");

                for(int i=0;i<5;i++)
                    b[i].disable();
                b[2].enable();
                return true;
            }
            else if (e.target == b[2]) // User gathers between-item time data
            {
                b[2].setLabel("Just a second...");
                for(int i=0;i<5;i++)
                    b[i].disable();

                Setup stp = new Setup();
                b[2].setLabel("Hierarchy search");
                SearchTime st = new SearchTime("Hierarchy search");

                b[3].enable();
                return true;
            }
            else if (e.target == b[3]) // Computer builds model and creates optimal hierarchy
            {
                for(int i=0;i<5;i++)
                    b[i].disable();
                b[3].setLabel("Come back in 1/2 hour");

                Optimize opt = new Optimize();
                b[3].setLabel("Optimize");
                b[4].enable();
                return true;
            }
        }
    }

```

```

else if (e.target == b[4]) // User gathers data with optimal hierarchy
{
    TestHierarchy th = new TestHierarchy("Testing");

    for(int i=0;i<5;i++)
        b[i].disable();
    return true;
}
}
return false;
}
}

```