A SIMPLIFIED CASE TOOL FOR
OBJECT-ORIENTED
ANALYSIS AND DESIGN

Presented in Partial Fulfillment of the Requirements
for the Degree Master of Science of Computer Science

By

Roger L. West

University of Illinois at Springfield

2001
# TABLE OF CONTENTS

List of Figures ........................................................................................................... ii

Rationale and Significance ..................................................................................... 1

Problem Statement .................................................................................................. 2

Literature Review ..................................................................................................... 3

  Importance of analysis and design in software development ....................... 3

  Benefits and Pitfalls of the Object-oriented Approach .............................. 5

  Modeling techniques for object-oriented analysis and design ................. 6

UML ....................................................................................................................... 11

CASE Tools in Software Development ............................................................. 13

Adoption of CASE Tools .................................................................................... 14

Solutions to the Problem of CASE Tool Adoption ....................................... 19

Plan of Work ......................................................................................................... 22

System Requirements .......................................................................................... 27

Time Schedule ....................................................................................................... 28

Vita ......................................................................................................................... 29

References ............................................................................................................ 31
LIST OF FIGURES

Figure 1. Relative cost of change during the software life cycle
Figure 2. Examples of some of the notation used in the Booch method
Figure 3. Examples of some of the notation used in the OMT method
Figure 4. Examples of some of the notation used in the OOSE method
Figure 5. List of standard elements used by the Unified Modeling Language
Figure 6. Effect of the learning curve on performance
Figure 7. Screen shot of a typical integrated object-oriented CASE tool
Figure 8. Examples of possible questions to be asked of beta testers
Table 1. Some available CASE tools for object-oriented analysis and design
Table 2. Proposed functionality requirements for an integrated object-oriented CASE environment
Table 3: Proposed functionality for a simplified object-oriented CASE tool.
RATIONALE AND SIGNIFICANCE

In object-oriented analysis and design it is often difficult to predict at the outset, for a given project, what the optimum set of classes will be and how those classes should interact. But success at this stage of object-oriented development is vital to the creation of robust software. Many Computer Assisted Software Engineering (CASE) tools are available to assist developers with object-oriented analysis and design. However, effective use of most of these tools requires that the developer possess a mastery of one or more of the many object-oriented analysis and design methods. The currently available methods can be quite complex, making extensive use of various types of diagrams and notations. Due to the complexity of these methods, the CASE tools that support them also tend to be complex, resulting in a steep learning curve for those attempting to use them. Productivity usually suffers during this learning period, discouraging many organizations and individuals from adopting the CASE tools as part of their development environment. Additionally, many developers may eschew the existing CASE tools simply because they are more comfortable, and more effective, using their own analysis and design methods. Such developers would benefit from a simplified object-oriented analysis and design tool that provides assistance with the creation of object-oriented components, but without the steep learning curve.
PROBLEM STATEMENT

The author proposes to develop a simplified CASE tool that will assist developers with the analysis and design phases of object-oriented software development. The proposed tool will be usable by anyone familiar with object-oriented programming concepts and will not require knowledge of any particular object-oriented analysis or design method, or of any special sets of notation. The tool's primary goal will be to serve as a "whiteboard" to allow developers to easily visualize the overall design of a project as it evolves. The tool will also provide support for certain tedious activities associated with software development, such as the generation of skeleton code for object classes. It will be further simplified in that it will not attempt to cover phases of the software lifecycle beyond analysis and design, such as implementation or testing. A summary of the tentative features of the proposed tool is shown in Table 3.
Importance of analysis and design in software development

One of the most challenging aspects of software engineering is how to maximize quality while minimizing development time. Many software process models have been proposed over the years to help software engineers more rapidly produce high-quality software. The driving force behind these process models was to impose a logical ordering on software development activities by separating the software lifecycle into discrete phases. Generically, the phases common to all the process models are, in order: analysis, design, coding, testing, and maintenance.

During analysis a problem is defined and requirements are gathered. Design deals with determining how to develop the software solution to the problem. The bulk of the code is written during the coding phase, and subsequently tested during the testing phase. Following testing the software is released as a product and is revised or corrected as necessary during the maintenance phase. There are several variations on this waterfall model, although technically speaking the ordering of the abovementioned phases is maintained. One exception is in object-oriented software development, where the design phase often overlaps the coding phase (Henderson-Sellers & Edwards, 1990).

Efficiency is lost whenever change is introduced into software development. Changes can be required in order to fix errors that are discovered
in the design or code. Changes are also often necessary if the user requirements change. Figure 1 illustrates how the cost of changing the software system increases dramatically as the software proceeds through the software lifecycle (Ambler, 1995; Pressman, 1997).

Figure 1: Relative cost of change during the software life cycle.

Exact values for the cost of change can vary due to a variety of factors, but generally speaking the later a change is introduced to the system, the more expensive it is to implement the change. Furthermore, the cost increase tends to follow an exponential rather than linear trend. Problems discovered during the maintenance phase are potentially catastrophic. Since analysis and design are concerned with defining the fundamental software architecture, these phases are of premiere importance to developing high-quality software in a cost-effective
manner. While testing is also absolutely crucial, a component that passes all tests is useless if it is not the "right" component for the job.

Benefits and Pitfalls of the Object-oriented Approach

In recent years the object-oriented approach to software development has become the paradigm for the development of many complex software systems. Martin & Odell (1992) summarize many of the numerous benefits to the object-oriented approach. With the object-oriented approach, developers can think more closely in terms of real-world entities. Often this simplifies the translation of a real-world view into a software system. Encapsulation of implementation details makes complex classes easier to use. Inheritance simplifies the creation of similar classes. Reusable classes allow for more rapid development and ultimately lead to greater stability and reliability of software systems. Modularity provides for easier programming and maintenance. Together these factors simplify development by making it easier for humans to conceptualize what they are trying to build.

Using the object-oriented approach is not always easy, despite its many advantages. In fact, lack of skill is one of the biggest obstacles to the growth of object technology (Tatsuta, 1996). Developers who are accustomed to thinking in a procedural manner must train themselves to think in an object-oriented manner, which can be difficult. One of the most problematic tasks is determining precisely what object classes must be created, a task that is performed during the transition from object-oriented analysis to object-oriented design (Kaindl,
During analysis, real-world objects are identified that are relevant to the problem to be solved. The design phase involves creating, and sometimes implementing, abstractions of the objects identified during analysis. Depending on the problem, it may be discovered during design that the set of classes actually created is not identical to the set of classes identified during analysis. For instance, depending on the approach taken to solve the problem, it may be more efficient to model a real-world entity as a variable rather than an object. In addition, it may be necessary to create additional support classes that do not correspond to any of the classes identified during analysis. An example of such a supporting class would be an adapter class for interacting with a specialized data structure that has been modified to optimize execution speed.

**Modeling techniques for object-oriented analysis and design**

Numerous methods have been created to assist developers with object-oriented analysis and design. These methods include: Booch (Booch, 1994), OMT (Object Modeling Technique) (Rumbaugh, et al., 1991), OOSE (Object-Oriented Software Engineering) (Jacobson, et al., 1992), UML (Unified Modeling Language) (Object Management Group, 1999), BON (Business Object Notation) (Waldén and Nerson, 1995), Martin-Odell (Martin and Odell, 1992), Coad-Yourdon (Coad and Yourdon, 1990), MOSES (Henderson-Sellers and Edwards, 1994), OSA (Object-oriented Systems Analysis) (Embley, et al., 1992), Syntropy (Cook and Daniels, 1994), SOMA (Semantic Object Modeling Approach) (Graham, 1995), and Shlaer-Mellor (Shlaer and Mellor, 1992). Brinkkepper, et al.
(1995) have published a comparative analysis of several of the most popular methods. Most object-oriented CASE tools typically support one or more of these methods.

In order to use any of the methods effectively it is necessary to intimately understand the concepts on which they are based, as well as become familiar with the particular notations involved. The notation can become quite extensive, depending on the method. Figures 2 through 4 show examples of only a portion of the notations used by the Booch (Booch, 1994), OMT (Rumbaugh, et al., 1991), and OOSE (Brinkkepper, et al., 1995) methods, respectively.
Figure 2: Examples of some of the notation used in the Booch method.
Figure 3: Examples of some of the notation used in the OMT method.
Figure 4: Examples of some of the notation used in the OOSE method.
The Unified Modeling Language (UML) is currently the most advanced of all the methods. It was originally conceived by Grady Booch, Ivar Jacobson, and James Rumbaugh, developers of the Booch, OOSE, and OMT methods, respectively (Object Management Group, 1999). It is currently the standard adopted by the Object Management Group, which is a consortium of companies and individuals involved in the continuing evolution of UML. Their goal is to create a single standardized language to support object-oriented analysis and design. UML is a unification of the Booch, OMT, and OOSE methods, but is designed to be simpler and more uniform than any of those methods alone. It differs from other methods in that it is a true language rather than simply a collection of symbols and notations for constructing diagrams. Models are created by "coding" the system design using the elements provided by the language. The language is extensible, meaning that new user-generated language elements can be created from existing elements.

While UML has been heartily accepted by many developers, there are some signs that it may begin to lose some of its favor (Kobryn, 1999). The most recent specification (version 1.3) is just over 800 pages long, which is only one indication of its complexity. UML makes use of a large number of different diagrams, each of which is intended to provide a different view of the system. There are significant problems with the extensibility mechanisms of UML, which are designed to allow users to create profiles that customize UML according to their own needs. There are 63 predefined standard elements in the language, as
shown in Figure 5. The semantics of many of these elements are somewhat vague, indicating that either these elements should be further defined or removed entirely. Numerous other improvements have also been proposed (Kobryn, 1999). The learning curve for mastering UML is steep. While the basic concepts can be easily grasped, much more time is required to fully assimilate the more advanced concepts. As a result, developers can quickly become overwhelmed by UML.

<table>
<thead>
<tr>
<th>access</th>
<th>«global»</th>
<th>«precondition»</th>
</tr>
</thead>
<tbody>
<tr>
<td>association</td>
<td>«implementation»</td>
<td>«process»</td>
</tr>
<tr>
<td>«association»</td>
<td>«implementationClass»</td>
<td>«realize»</td>
</tr>
<tr>
<td>«become»</td>
<td>implicit</td>
<td>«refine»</td>
</tr>
<tr>
<td>«call»</td>
<td>«import»</td>
<td>«requirement»</td>
</tr>
<tr>
<td>complete</td>
<td>incomplete</td>
<td>«responsibility»</td>
</tr>
<tr>
<td>«copy»</td>
<td>«instantiate»</td>
<td>self</td>
</tr>
<tr>
<td>«create»</td>
<td>«invariant»</td>
<td>«self»</td>
</tr>
<tr>
<td>«derive»</td>
<td>«library»</td>
<td>semantics</td>
</tr>
<tr>
<td>derived</td>
<td>local</td>
<td>«send»</td>
</tr>
<tr>
<td>«destroy»</td>
<td>«local»</td>
<td>«signalflow»</td>
</tr>
<tr>
<td>destroyed</td>
<td>«metaclass»</td>
<td>«stub»</td>
</tr>
<tr>
<td>disjoint</td>
<td>«metamodel»</td>
<td>«systemModel»</td>
</tr>
<tr>
<td>«document»</td>
<td>new</td>
<td>«table»</td>
</tr>
<tr>
<td>documentation</td>
<td>overlapping</td>
<td>«thread»</td>
</tr>
<tr>
<td>«executable»</td>
<td>parameter</td>
<td>«topLevel»</td>
</tr>
<tr>
<td>«I»</td>
<td>«parameter»</td>
<td>«trace»</td>
</tr>
<tr>
<td>«file»</td>
<td>persistence</td>
<td>transient</td>
</tr>
<tr>
<td>«framework»</td>
<td>persistent</td>
<td>«type»</td>
</tr>
<tr>
<td>«friend»</td>
<td>«postcondition»</td>
<td>«utility»</td>
</tr>
<tr>
<td>global</td>
<td>«powertype»</td>
<td>xor</td>
</tr>
</tbody>
</table>

Figure 5: List of standard elements used by the Unified Modeling Language.
CASE Tools in Software Development

The number of CASE tools currently available is staggering. CASE tools have been developed to aid in virtually every aspect of software development. CASE tools fall into two general categories: integrated or non-integrated. Non-integrated tools function separately, they tend to support only one task or phase in the software development life cycle, and they have no standard mechanism to allow them to interact with other tools. Integrated tools are designed to interact and communicate with other programs, and they tend to support multiple tasks or phases of the software development life cycle. The current trend is toward the development of integrated tools that attempt to cover every phase of software development (Norman & Chen, 1992). Often these integrated tools are packaged into single, highly complex, monolithic applications.

Stobart (2000) summarizes the potential benefits of CASE tools as follows: 1) they promote an interactive development environment; 2) they reduce the cost of developing software, especially during the maintenance phase; 3) they improve software quality; 4) they reduce the time required to develop software; 5) they promote consistency, which improves accuracy; 6) they make it practical to use structured techniques; and 7) they increase productivity. As with any tool, however, it must be stressed that these benefits will be fully realized only if the tool is used correctly.
Adoption of CASE Tools

Despite the potential benefits of CASE tools, their use is not universal. Even when an organization does introduce CASE tools, often the tools are not used at all, or only a fraction of their capabilities are used (Kemerer, 1992). The greatest problem appears to be the high cost associated with investing in a CASE tool. An organization must first adopt standards for their design methodology before they can choose which CASE tool to use. All developers in the organization must adhere to that methodology in order for the full benefits of the CASE tool to be realized. It is a daunting task to achieve complete acceptance of such standards since the practitioners in an organization typically differ with respect to their capabilities and their preferred styles of work (Albizuri-Romero, 2000).

Probably the single most influential aspect that deters the use of CASE tools is the learning curve phenomenon, particularly with respect to integrated CASE tools. The learning curve exists because time is required to become familiar with the tool and its underlying methodology, as well as to adjust to the altered working environment. Figure 6 illustrates the effect that adjusting to new technology has on performance (Chew, et al., 1991). Those who adopt new technology frequently believe their performance will initially remain the same, increase to a certain level, and then plateau. Typically, however, improvement occurs much later than expected, and it is preceded by a significant period of time during which performance actually declines. Early projects developed with the tool tend to be more costly than if they were developed using the old methods.
(Kemerer, 1992). The initial productivity can drop by 50% or more for at least the first six months. Furthermore, the total improvement in performance may not be as high as expected. It is therefore not surprising that many organizations have been reluctant to adopt CASE tools. Many managers feel that whatever benefits could be obtained by using CASE tools do not justify the investment (Summer, 1992).

Figure 6: Effect of the learning curve on performance.

The author’s review of related literature revealed no reliable published data describing the degree to which CASE tools are used by individual developers, as opposed to organizations. The high cost of many CASE tool packages may prohibit their use by developers outside the workplace. Table 1 lists some of the most popular integrated CASE tools, their current costs, and the analysis and design methodologies they support. Even if cost is not an issue,
developers must still spend time learning how to use whichever tool they choose. If the tool appears to be too complex it will likely be abandoned in favor of a simpler method.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Creator</th>
<th>Cost</th>
<th>Analysis/Design Methods Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>GDPro (version 4.1)</td>
<td>Advanced Software Technologies, Inc.</td>
<td>$2995 (Windows) $4795 (UNIX)</td>
<td>UML</td>
</tr>
<tr>
<td>Paradigm Plus (version 3.6)</td>
<td>Platinum Technology, Inc</td>
<td>N/A (must call for price)</td>
<td>UML, OMT, Booch, Martin-Odell, Shlaer-Mellor, Coad-Yourdon, and others</td>
</tr>
<tr>
<td>Rational Rose (version 2000e)</td>
<td>Rational Software Corporation</td>
<td>$4287 (enterprise edition)</td>
<td>UML</td>
</tr>
<tr>
<td>Rhapsody in C++ Solo Edition (version 2.2)</td>
<td>I-Logix, Inc.</td>
<td>$895</td>
<td>UML</td>
</tr>
<tr>
<td>Together Enterprise (version 4.0)</td>
<td>TogetherSoft Corporation</td>
<td>cost varies with reseller</td>
<td>UML, Coad-Yourdon</td>
</tr>
<tr>
<td>With Class 99 (version 6.0)</td>
<td>MicroGold Software, Inc.</td>
<td>$825 (enterprise edition)</td>
<td>UML, OMT, Booch, Coad-Yourdon, Shlaer-Mellor, Martin-Odell</td>
</tr>
</tbody>
</table>

Table 1: Some available CASE tools for object-oriented analysis and design.

Prices listed were obtained from the web sites of the manufacturers and were current as of June 6, 2000.

Figure 7 shows a typical opening screen that appears when an integrated CASE tool is launched. The developer is typically inundated with a myriad of
menu options and toolbar buttons, which can be very confusing until the tool is mastered. But before the tool can be mastered, the analysis and design methods supported by the tool must be mastered, which also requires a significant amount of time. Often the developer may decide to avoid using the tool, believing as many organizations do that the time and effort invested are not worth the potential benefits promised by the tool. Therefore, the decision of whether or not to use CASE tools can be impacted greatly by the learning curve phenomenon.
Figure 7: Screen shot of a typical integrated object-oriented CASE tool.
Solutions to the Problem of CASE Tool Adoption

There are several ways to deal with the problem of the learning curve with respect to the use of CASE tools. One obvious solution is to simply accept that CASE tools must be inherently complex and difficult to use, and spend whatever time is needed to master the tools themselves and the methodologies they support. Any performance or productivity losses that are initially incurred must likewise be accepted as inevitable side effects that will hopefully be compensated for in the long run. At the other extreme, CASE tools could be viewed as good ideas that are simply not practical at this time because of their potential negative impact on productivity. People have a tendency to become comfortable with techniques that have proven successful for them. They are also resistant to change that has no guarantee of improving success, especially when profits are at stake.

A third possible solution is to try to find a way to allow some of the benefits of CASE tools to be realized while at the same time minimizing the disruption caused by using them. Ideally this would entail the creation of a simplified tool that would provide many of the benefits of CASE tools without requiring mastery of any particular methodology. This solution immediately raises the question of what functionality to include in such a tool. Losavio, Matteo, & Pérez (1999) have proposed a set of functionalities that should be included in any integrated object-oriented CASE environment. These functionalities are listed in Table 2. As can be seen from the table, some of the proposed functionalities are not involved directly with the software development life cycle, but instead deal with
## Processes
- must support the object-oriented paradigm
- must allow for reuse of software components
- must allow the combination of different methods at different software development phases
- must adapt to the company's organization process

## Development Services
- include tools to support analysis, design and implementation
- must be able to express in notation the different models of the different object-oriented methods
- provide version control services
- closely link analysis, design, and programming activities

## Management Services
- support two types of users: developers and managers
- provide backup to the definition and management of the project structure

## Support Services
- provide the ability to allow a team of developers to work concurrently and in different locations
- support visualization of the information
- support control of the definition, access, and use of the components of the system
- support the planning and control of the project

## General Services
- browsing
- integration of interfaces

### Mechanisms (for a monolithic environment)
- client/server architecture allowing communications among tools
- Portable Common Tool Environment (PCTE) standard
- object-oriented databases

### Mechanisms (for a federative environment)
- client/server architecture allowing communication among tools
- standard import/export mechanisms
- independent databases
- support integration of the interfaces of the various tools

Table 2: Proposed functionality requirements for an integrated object-oriented CASE environment.

management activities. Although these functionalities may be important for organizations that must manage teams of developers, they are not vital to the individual developer. The authors also propose that the tool should be able to support the notation of the various object-oriented analysis and design methods.
Attempting to support several different methods greatly adds to the complexity of the tool, which makes the learning curve for the tool higher.

A design tool that is easier to use would benefit both experienced and inexperienced developers. Developers who are new to object-oriented development would still be productive while they are gaining experience. They will be able to quickly experiment with various design ideas without the immense overhead of an integrated tool. Even those who have fully embraced an existing methodology along with a complex integrated CASE tool may still benefit from a simplified tool in certain cases. More importantly, those who prefer to use their own methodology would not be hampered by the need to learn a new, complex methodology as well as the intricacies of a complex CASE tool.
PLANNED WORK

Requirements Analysis

Table 3 lists the proposed functionalities to be included in the CASE tool, as well as certain features that will not be supported. This list is tentative, since it may be discovered during development that additional features are required or some existing features may prove to be unworkable, or impractical due to time limitations. The tool will be object-oriented, and will be written in Visual Basic to take advantage of the GUI development features of the Visual Basic IDE. A list of the expected hardware and system requirements for the completed tool are included in the proposal.

Analysis/Design

The overall tool will be developed in an incremental fashion, giving highest priority to the most important features. Since the success of this tool will depend heavily on the ease of use of the user interface, the GUI will be developed first, which should help in identifying all of the various actions that can be taken (e.g., adding a new class, deleting a class, etc.). The internal workings of the tool will be developed to support the GUI.

The main approach for the internal implementation will be to first identify the commonalities of three common object-oriented programming languages (C++, Java, Visual Basic). This analysis will be used to derive a generalized design for the core of the tool. Features that are language-specific will be added
<table>
<thead>
<tr>
<th>Feature</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>creation of designs for all common object-oriented components</td>
<td>yes</td>
</tr>
<tr>
<td>creation of designs for object interactions</td>
<td>yes</td>
</tr>
<tr>
<td>creation of projects consisting of collections of existing components and links to files generated using other applications</td>
<td>yes</td>
</tr>
<tr>
<td>components may be stored in a repository for reuse</td>
<td>yes</td>
</tr>
<tr>
<td>easy means of navigation through components contained in the repository and in projects</td>
<td>yes</td>
</tr>
<tr>
<td>generation of skeleton code for classes</td>
<td>yes</td>
</tr>
<tr>
<td>programming language support</td>
<td>Java (initially)</td>
</tr>
<tr>
<td>database format used for projects</td>
<td>flat text file</td>
</tr>
<tr>
<td>Operating system support</td>
<td>Windows 9x</td>
</tr>
<tr>
<td>online help manual</td>
<td>Yes (time permitting)</td>
</tr>
<tr>
<td>integration with other applications</td>
<td>no</td>
</tr>
<tr>
<td>support for coding and compilation of code</td>
<td>no</td>
</tr>
<tr>
<td>support for automated testing</td>
<td>no</td>
</tr>
<tr>
<td>support for reverse engineering</td>
<td>no</td>
</tr>
<tr>
<td>support for management services</td>
<td>no</td>
</tr>
<tr>
<td>support for specific OOAD methodologies and notations</td>
<td>no</td>
</tr>
</tbody>
</table>

Table 3: Proposed functionality for a simplified object-oriented CASE tool.

afterwards. The tool will initially support code generation for Java only. If time permits, support will be included for other languages. If time does not permit creation of an online user’s guide, the user’s guide will be supplied as a separate document. A high degree of modularity will be used in order to accomplish the incremental build of the tool. Diagrams, decision tables, data dictionaries, and
other design aids will be used as needed. All designs will be documented and incorporated into a programmer’s manual.

**Implementation**

Since the tool will be developed in an incremental fashion, features will be coded as soon as the designs for that feature have been completed. If design changes are required due to requirements changes or flaws that are discovered, work on this phase will be suspended and the analysis and design phases will be reinitiated to make the necessary design changes. Once those changes have been made the implementation phase will continue. Whenever a design change is required, a new design version will be created.

**Testing**

Testing will be automated wherever it is feasible to do so. Test plans and harnesses will be designed to unit test all methods. For each method complete branch testing will be performed. Where applicable, boundary testing will also be performed. Random testing may be performed if it is feasible to do so, and where time permits. Test plans, containing inputs and expected outputs, will be created for each unit test. All test plans will be saved in files to facilitate regression testing. After all methods have been unit tested, integration testing will be performed. The user interface will be tested to ensure that each interface component performs properly. In order to facilitate testing of the interface a
checklist will be made that contains all the functional interface components and their expected actions.

Finally the application will be system tested by using it to create sample software components and projects. If possible, individuals who are familiar with the basics of object-oriented programming will be enlisted as beta testers to assist with this stage of testing. They will be given no specific instructions on how they should test the application, but they will be encouraged to look for ways to "break" the application so that unanticipated bugs can be discovered and corrected. Beta testers will be asked to record a brief log containing the following information: 1) what they did during their testing; 2) what bugs they discovered; and 3) suggested improvements. Beta testers will also be asked to complete a questionnaire containing a set of questions designed to measure their personal opinions about how well the application performed. Examples of possible questions are listed in Figure 8. Another primary goal of the beta testing will be to execute the application under different hardware and software configurations. Some factors that will be considered are: 1) operating system; 2) processor type and speed; 3) what software is running concurrently on the system.
1) Was the user interface intuitive?

2) Were you able to easily create components (classes, methods, tasks, etc.)?

3) Did the lack of support for formal object-oriented analysis and design methods make the application easier to use, or more difficult to use?

4) Did the absence of support for other phases of the software lifecycle (e.g., implementation, testing) make the application more difficult to use?

5) Did the layout of the user interface allow easy navigation through the components you created?

6) Was the online help manual useful in answering any questions you had?

7) Was the speed with which the application performed acceptable, too slow, or too fast?

8) Did the application accurately save the information for the components you created?

9) If you encountered technical problems, did you receive an error message, and if so, was the error message helpful?

10) What improvements would you suggest for the application?

Figure 8: Examples of possible questions to be asked of beta testers
System Requirements

The application will be written using Visual Basic 6.0. It will be developed on a 500 MHz Pentium III system running Windows 98. The final product will be assembled into an installable package that will contain all files required for use of the application on a similar system. More precise system requirements cannot be accurately determined until the project is near completion. The tentative estimates for the recommended minimum system requirements are:

**System:**
- PC with a Pentium, Pentium II, Pentium III, or other comparable processor, 200 MHz or greater

**RAM:**
- 32 Mb or greater

**Operating System:**
- Windows 9x (may also run under Windows NT, Windows 2000, and/or Windows Me without further modification)

**Free Hard Drive Space:**
- 10 Mb for installation
- 20 Mb+ for use by the application (this additional space will be required for storage of user-generated components)
The following schedule is an estimate of the time expected to complete each phase of development, assuming an average of approximately 50 – 60 hours per week are spent on the project. An attempt was made to contain all activities within approximately one semester of time.

<table>
<thead>
<tr>
<th>Number of Weeks Spent</th>
<th>% of Time</th>
<th>Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>March 1 – March 30 (2001)</td>
<td>26</td>
<td>Implementation</td>
</tr>
<tr>
<td>April 1 – April 30 (2001)</td>
<td>24</td>
<td>Testing, beta evaluation, bug fixes</td>
</tr>
</tbody>
</table>
VITA

Roger L. West

Education

B.S. Microbiology, University of Illinois at Urbana-Champaign

Ph.D. Microbiology, Southern Illinois University

Experience

Media Assistant III, University of Illinois at Springfield
Postdoctoral researcher, Louisiana State University Medical Center

Postdoctoral researcher, Southern Illinois University School of Medicine

Graduate Research Assistant, Southern Illinois University School of Medicine

Graduate Research Assistant, Southern Illinois University at Carbondale

Course Work

From the University of New Orleans

SP 97 CSCI 1583 – Software Design and Development I

SP 97 CSCI 1581 – Software Design and Development Lab

SU 97 MATH 2721 – Introduction to Discrete Structures

AU 97 CSCI 2120 – Software Design and Development II

AU 97 CSCI 2450 – Machine Structure and Assembly Language Programming

SP 98 CSCI 2125 – Data Structures

SP 98 MATH 2511 – Introduction to Linear Algebra

SU 98 CSCI 4125 – Introduction to Databases
AU 98 CSCI 3401 – Systems Programming Concepts
SP 99 CSCI 4401 – Principles of Operating Systems I

From the University of Illinois, Springfield

AU 99 CSC 478 – Software Engineering Capstone
AU 99 MAT 321 – Applied Statistics
SP 00 CSC 483 – Intro to Computer Networks
SP 00 CSC 577 – Software Testing and Reliability Seminar
SP 00 CSC 582 – Design and Evaluation of User Interfaces
SU 00 CSC 470B – Developing Web Applications Using PERL
FA 00 CSC 473 – Programming Languages
FA 00 CSC 572 – Advanced Database Concepts
SP 01 CSC 550 – Master’s Project/Thesis
SP 01 CSC 499 – Empirical Software Testing (independent study)
REFERENCES


