SimpleOOD:
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 and Tables.................................................................................................................. ii
Abstract.................................................................................................................................................. 1
Acknowledgements.................................................................................................................................. 2
Rationale and Significance..................................................................................................................... 3
Problem Statement.................................................................................................................................. 4
Literature Review..................................................................................................................................... 5
  Importance of Analysis and Design in Software Development......................................................... 5
  Benefits and Pitfalls of the Object-oriented Approach................................................................. 7
  Modeling Techniques for Object-oriented Analysis and Design...................................................... 8
  UML....................................................................................................................................................... 13
  CASE Tools in Software Development............................................................................................ 15
  Adoption of CASE Tools.................................................................................................................... 16
  Solutions to the Problem of CASE Tool Adoption........................................................................... 22
Materials and Methods........................................................................................................................ 25
  Software and Hardware Used........................................................................................................... 25
Design of the Investigation.................................................................................................................. 26
  Overview............................................................................................................................................. 26
  Operation............................................................................................................................................ 26
  The ObjectControl ActiveX Control............................................................................................... 30
  MainModule ....................................................................................................................................... 31
  MainForm............................................................................................................................................ 32
  MainCanvasForm............................................................................................................................... 32
  Data Entry Forms............................................................................................................................ 35
  RepositoryForm and the Repository................................................................................................. 44
  Other Forms ....................................................................................................................................... 45
  The Component Class Module......................................................................................................... 47
  The Project Class Module................................................................................................................ 47
  The ComponentFactory and JavaComponentFactory Class Modules......................................... 48
  The LanguageSpecs and JavaSpecs Class Modules........................................................................ 48
  The MenuSwitchboard Class Module.............................................................................................. 49
  The ResourceCenter and JavaResourceCenter Class Modules.................................................... 49
  Software and Hardware Specifications............................................................................................ 50
  Procedures......................................................................................................................................... 50
  Methods for Observation and Interpretation.................................................................................... 51
Results.................................................................................................................................................... 54
  Unit Testing....................................................................................................................................... 54
  Bugs................................................................................................................................................... 55
  Evaluations of Beta Testers............................................................................................................... 55
Discussion............................................................................................................................................. 57
  Future Work....................................................................................................................................... 58
Vita......................................................................................................................................................... 60
References............................................................................................................................................. 62
LIST OF FIGURES AND TABLES

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. General application architecture
Figure 9A. The ObjectControl ActiveX control (default size)
Figure 9B. The ObjectControl ActiveX control (iconized size)
Figure 10. MainCanvasForm
Figure 11. ClassDialog data entry form
Figure 12. InterfaceDialog data entry form
Figure 13. ConstantDialog data entry form
Figure 14. VariableDialog data entry form
Figure 15. ConstructorDialog data entry form
Figure 16. MethodDialog data entry form
Figure 17. Data entry form for GUI component files
Figure 18. Data entry form for external resource files
Figure 19. Data entry form for argument lists
Figure 20. RepositoryForm
Figure 21. Sample questionnaire for 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. Forms and Modules used in SimpleOOD

Table 4. Bugs discovered during testing
ABSTRACT

Object-oriented programming is used widely to develop complex software systems. The design phase of software construction is particularly important to the development of high-quality, maintainable software. Many Computer Assisted Software Engineering (CASE) tools are available to help developers create software designs. Unfortunately these tools tend to be rather complex, and mastery of the tools also requires mastery of one or more complex analysis and design methodologies. As a result, most object-orient design tools tend to have a rather steep learning curve that discourages their use in industry. This paper describes the design and creation of SimpleOOD, an object-oriented design tool with a simplified user interface that can be used without knowledge of any complex design methodology.
ACKNOWLEDGEMENTS

First I would like to thank my parents, Don and Vesta West, for all of their support through the years. I would never have gotten where I am today without them.

I would also like to thank my advisor, Dr. Keith Miller, who taught me how much more important software design and testing are than hacking out code. His software engineering class spurred my interest in the areas of CASE tools and reusable components.

I also thank the other members of my committee, Dr. Ted Mims and Dr. Gary Trammell, for all of their help with this project, particularly with the writing of the proposal and the final report.

I thank Katerie Gladdys and Dennis Scheibe for their time and effort with beta testing the final application.

Finally, I would like to thank all of my friends, in the Media Services department and elsewhere, for helping me to stay in touch with the rest of the world while I labored on this project.
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 proposed to develop a simplified CASE tool that would assist developers with the analysis and design phases of object-oriented software development. The CASE tool that was developed is usable by anyone familiar with object-oriented programming concepts and does not require knowledge of any particular object-oriented analysis or design method, or of any special sets of notation. The tool's primary goal is to serve as a "whiteboard" to allow developers to easily visualize the overall design of a project as it evolves. The tool also provides support for certain tedious activities associated with software development, such as the generation of skeleton code for object classes. It is further simplified in that it does not attempt to cover phases of the software lifecycle beyond analysis and design, such as implementation or testing.
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.](image)

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, 1999). 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.
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 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 can 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><strong>Tool</strong></th>
<th><strong>Creator</strong></th>
<th><strong>Cost</strong></th>
<th><strong>Analysis/Design Methods Supported</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>GDPro (version 4.1)</td>
<td>Advanced Software Technologies, Inc.</td>
<td>$2995 (Windows)</td>
<td>UML</td>
</tr>
<tr>
<td></td>
<td>&lt;www.advancedsw.com&gt;</td>
<td>$4795 (UNIX)</td>
<td></td>
</tr>
<tr>
<td>Paradigm Plus</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>(version 3.6)</td>
<td>&lt;www.platinum.com&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rational Rose</td>
<td>Rational Software Corporation</td>
<td>$4287 (enterprise edition)</td>
<td>UML</td>
</tr>
<tr>
<td>(version 2000e)</td>
<td>&lt;www.rational.com&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rhapsody in C++</td>
<td>I-Logix, Inc.</td>
<td>$895</td>
<td>UML</td>
</tr>
<tr>
<td>Solo Edition</td>
<td>&lt;www.ilogix.com&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(version 2.2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Together Enterprise</td>
<td>TogetherSoft Corporation</td>
<td>cost varies with reseller</td>
<td>UML, Coad-Yourdon</td>
</tr>
<tr>
<td>(version 4.0)</td>
<td>&lt;www.togethersoft.com&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>With Class 99</td>
<td>MicroGold Software, Inc.</td>
<td>$825 (enterprise edition)</td>
<td>UML, OMT, Booch, Coad-Yourdon, Shlaer-Mellor, Martin-Odell</td>
</tr>
<tr>
<td>(version 6.0)</td>
<td>&lt;www.microgold.com&gt;</td>
<td></td>
<td></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
<table>
<thead>
<tr>
<th>Processes</th>
<th>must support the object-oriented paradigm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>must allow for reuse of software components</td>
</tr>
<tr>
<td></td>
<td>must allow the combination of different methods at different software development phases</td>
</tr>
<tr>
<td></td>
<td>must adapt to the company’s organization process</td>
</tr>
<tr>
<td>Development Services</td>
<td>include tools to support analysis, design and implementation</td>
</tr>
<tr>
<td></td>
<td>must be able to express in notation the different models of the different object-oriented methods</td>
</tr>
<tr>
<td></td>
<td>provide version control services</td>
</tr>
<tr>
<td></td>
<td>closely link analysis, design, and programming activities</td>
</tr>
<tr>
<td>Management Services</td>
<td>support two types of users: developers and managers</td>
</tr>
<tr>
<td></td>
<td>provide backup to the definition and management of the project structure</td>
</tr>
<tr>
<td>Support Services</td>
<td>provide the ability to allow a team of developers to work concurrently and in different locations</td>
</tr>
<tr>
<td></td>
<td>support visualization of the information</td>
</tr>
<tr>
<td></td>
<td>support control of the definition, access, and use of the components of the system</td>
</tr>
<tr>
<td></td>
<td>support the planning and control of the project</td>
</tr>
<tr>
<td>General Services</td>
<td>browsing</td>
</tr>
<tr>
<td></td>
<td>integration of interfaces</td>
</tr>
<tr>
<td>Mechanisms (for a monolithic environment)</td>
<td>client/server architecture allowing communications among tools</td>
</tr>
<tr>
<td></td>
<td>Portable Common Tool Environment (PCTE) standard</td>
</tr>
<tr>
<td></td>
<td>object-oriented databases</td>
</tr>
<tr>
<td>Mechanisms (for a federative environment)</td>
<td>client/server architecture allowing communication among tools</td>
</tr>
<tr>
<td></td>
<td>standard import/export mechanisms</td>
</tr>
<tr>
<td></td>
<td>independent databases</td>
</tr>
<tr>
<td></td>
<td>support integration of the interfaces of the various tools</td>
</tr>
</tbody>
</table>

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.
MATERIALS AND METHODS

Software and Hardware Used

*SimpleOOD* was written in Microsoft Visual Basic 6.0, Professional Edition. Most of the icons in the application were created from scratch using Adobe Photoshop 5.0. Some icons were taken from the icon library provided by Visual Studio 6.0. Several icons were extracted from screen captures of various applications. The freeware program Screen Rip32 Version 1.0 beta (written by J. Elaraj from Progeny Software) was used for capturing screen shots. This program was also used along with Photoshop to obtain certain desired colors that were used in creating the application. The online help file was created using the Microsoft HTML Help Workshop, version 4.74.8702.0. The final installation package was created using the Package and Deployment Wizard that comes with Visual Basic 6.0. The language specification for the Java 2 Software Development Kit, Standard Edition, from Sun Microsystems, was used as the reference for the Java programming language.
DESIGN OF THE INVESTIGATION

Overview

The general architecture of SimpleOOD is illustrated in Figure 8. In the figure, forms are indicated by gray boxes and white boxes indicate modules. The only module MainModule is a standard Visual Basic module. All other modules are Visual Basic class modules. The black ObjectControl box represents a custom ActiveX control. The figure indicates the primary relationships between the various forms and modules. A complete listing of the application's forms and modules is shown in Table 3.

Operation

Users begin by either creating a new project or by opening an existing project. A project is defined as the collection of top-level components that are grouped together to form the design of a software system. Each project has an associated MainCanvasForm instance, and virtually everything the user does occurs on this form. Top-level components (components that are used to encapsulate entities) are represented by instances of the ObjectControl ActiveX control. Subcomponents are components that can only exist within a top-level component. In the case of Java, there are two top-level components: classes and interfaces. Java subcomponents include methods, constructors, variables, and constants. Once a top-level component and its associated ObjectControl have been created, users add the subcomponents via the ObjectControl.
Individual top-level components can be exported to, or imported from, the Repository. Users can also save projects, open existing projects, and generate skeleton code for all the top-level components in a project. Skeleton code consists only of declarations; the definitions, or implementation details, must be added using a standard text editor.
Figure 8. General application architecture showing the primary relationships between the forms and classes.
<table>
<thead>
<tr>
<th><strong>Data Entry Forms</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>ClassDialog</td>
<td>data entry form for classes</td>
</tr>
<tr>
<td>CommonDialogForm</td>
<td>data entry form for GUI components and external resources</td>
</tr>
<tr>
<td>ConstantDialog</td>
<td>data entry form for constants</td>
</tr>
<tr>
<td>ConstructorDialog</td>
<td>data entry form for constructors</td>
</tr>
<tr>
<td>CreateArgumentListForm</td>
<td>data entry form for argument lists</td>
</tr>
<tr>
<td>InterfaceDialog</td>
<td>data entry form for interfaces</td>
</tr>
<tr>
<td>MethodDialog</td>
<td>data entry form for methods</td>
</tr>
<tr>
<td>VariableDialog</td>
<td>data entry form for variables</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Other Forms</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>AboutForm</td>
<td>displays information about the application</td>
</tr>
<tr>
<td>MainCanvasForm</td>
<td>used for placement and manipulation of ObjectControls</td>
</tr>
<tr>
<td>MainForm</td>
<td>container form for MainCanvasForm instances</td>
</tr>
<tr>
<td>NewProjectDialog</td>
<td>form for choosing the programming language for a new project</td>
</tr>
<tr>
<td>RepositoryForm</td>
<td>provides access to the repository</td>
</tr>
<tr>
<td>UserMessageForm</td>
<td>general purpose form for displaying warning or error message to users</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Standard Modules</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>MainModule</td>
<td>startup module; contains commonly used utility methods</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Class Modules</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Component</td>
<td>encapsulates an object-oriented component</td>
</tr>
<tr>
<td>ComponentFactory</td>
<td>responsible for creating Components</td>
</tr>
<tr>
<td>JavaComponentFactory</td>
<td>Java-specific ComponentFactory; creates Java language components</td>
</tr>
<tr>
<td>JavaResourceCenter</td>
<td>Java-specific ResourceCenter; provides access to Java resources</td>
</tr>
<tr>
<td>JavaSpecs</td>
<td>encapsulates the rules for creating Java components</td>
</tr>
<tr>
<td>LanguageSpecs</td>
<td>encapsulates the &quot;rules&quot; for creating the components of an object-oriented language</td>
</tr>
<tr>
<td>MenuSwitchboard</td>
<td>responsible for properly directing menu options chosen from the dynamic menus of the MainCanvasForm</td>
</tr>
<tr>
<td>Project</td>
<td>encapsulates an application design project</td>
</tr>
<tr>
<td>ResourceCenter</td>
<td>responsible for providing access to resource files containing icons, value lists, etc.</td>
</tr>
<tr>
<td>ObjectControl</td>
<td>ActiveX control; representation of a top-level component on MainCanvasForm</td>
</tr>
</tbody>
</table>

Table 3: Forms and Modules used in *SimpleOOD*. Abstract class modules are indicated by *italics.*
The ObjectControl ActiveX Control

The heart of *SimpleOOD* lies in the ObjectControl custom ActiveX control, an example of which is shown in Figure 9A. During program use, a new ObjectControl instance is created and bound to every top-level component that is created. The ObjectControl is the user's gateway into the component that it represents. It displays a hierarchical tree structure that shows all subcomponent categories and all existing subcomponents. Nodes are added to the tree as subcomponents are created. Existing nodes can be modified or deleted as desired. The ObjectControl's size is scalable – it can be enlarged or reduced as needed in order to better see its contents. Collapsing the root node reduces an ObjectControl to an icon, minimizing its footprint on the screen. Figure 9B shows an example of an iconized ObjectControl.

![Figure 9. The ObjectControl ActiveX control. A) default size; B) iconized.](image)

A

B
The principle behind using the ObjectControl is to allow users to visualize an evolving design as a whole. Rather than having to remember a list of source files and their contents, users can easily view at any time the contents of a particular object simply by going to that object's associated ObjectControl. Users can optimize their design on the canvas before generating skeleton code, and thus reduce the chances of generating syntax errors by continually modifying source code.

The categories that appear in the tree of the ObjectControl are determined by the contents of a data file, which allows an arbitrary set of categories to be displayed. A companion data file is used to specify the icons used for the nodes. The ObjectControl was designed this way in order to permit future support of additional programming languages without having to change the code contained in the ObjectControl itself.

**MainModule**

MainModule is the startup module for the application. It handles the creation and initialization of new projects, and always maintains a handle on the currently active project. MainModule initiates the reading and writing of the project data files, and is responsible for generating skeleton code. It also provides various global utility methods, such as a method that determines whether or not a given string is a valid filename according to the restrictions of the operating system. Another utility method is used to display custom error messages to the user.
**MainForm**

MainForm is a Multiple Document Interface (MDI) form. It was created in order to allow the capability of having multiple projects open simultaneously. Each instance of MainCanvasForm that is created is a child form of MainForm. Aside from serving as a container for all open MainCanvasForms, MainForm has very little functionality.

**MainCanvasForm**

MainCanvasForm, shown in Figure 10, serves as the container, or "canvas" for the ObjectControls that are created by users. It also provides access to all the menu commands available for a project. Each project that is opened or created is bound to exactly one instance of MainCanvasForm. The menu bar of MainCanvasForm consists of two sections, the *static* section and the *dynamic* section. The static section contains the commands that are applicable to all projects, such as opening a project, saving a project, and generating skeleton code. The contents of the dynamic section are dependent on the programming language used for a project and are based on the contents of a data file that stores the menu structure.

In the current version of *SimpleOOD* the dynamic menu consists of the following menus: Classes, Interfaces, Constants, Variables, Constructors, Methods, GUI Components, and External Resources. Each menu contains commands that are appropriate for its component/subcomponent type. The Classes and Interfaces menus contain commands for creating, importing,
deleting, modifying, and exporting classes and interfaces, respectively. The menus for Constants, Variables, Constructors, and Methods all contain commands for adding, modifying and deleting the respective subcomponent type, as well as a command for deleting all existing subcomponents of that type. The GUI Components and External Resources menus are similar, the only difference being that they lack a modify command. Since the only information stored for these subcomponents is a filename, to modify one of these subcomponents the user must delete the old subcomponent and then add a new one.
Figure 10. *MainCanvasForm*, containing three top-level components.
**Data Entry Forms**

There are eight different data entry dialog forms, one for each type of component or subcomponent that can be created. The data entry forms package the information entered by the user into a Dictionary data structure, which can be retrieved from the forms via a public method. These forms also provide a method for initializing a form with existing information, allowing users to modify an existing component without having to re-enter all the previous data. Figures 11 – 19 show the various data entry forms used for the Java programming language.

Validation of the data occurs in the LanguageSpecs class rather than in the data entry forms, with a few exceptions. For some components certain modifiers are incompatible with each other, so ideally the data entry forms should prevent those modifier combinations from being chosen. As an example, for a Java method the modifier *abstract* cannot be used in combination with any other method modifier. The data entry form for methods enforces this rule by disabling the checkboxes for the other modifiers when the *abstract* modifier is chosen. For the sake of simplicity the implementation that accomplishes this was placed into the data entry form.

Due to the language differences between programming languages, the data entry forms are inherently language-specific. In the current version of *SimpleOOD* only data entry forms for Java components are included. If support is added for additional programming languages, each language must have its own set of data entry forms.
Figure 11. *ClassDialog*, the data entry form for Java classes.
Figure 12. *InterfaceDialog*, the data entry form for Java interfaces.
Figure 13.  *ConstantDialog*, the data entry form for Java constants.
Figure 14. *VariableDialog*, the data entry form for Java variables.
Figure 15. ConstructorDialog, the data entry form for Java constructors.
Figure 16. *MethodDialog*, the data entry form for Java methods.
Figure 17. Data entry form for links to Java GUI component files.

Figure 18. Data entry form for links to external resource files.
Figure 19. Data entry form for Java arguments.
RepositoryForm and the Repository

The repository feature of SimpleOOD allows users to save potentially reusable components in a centralized area from which they can be imported into other projects. Components in the repository are organized based on their identifiers. A new folder is created for each unique identifier. Within each such folder, an arbitrary number of versions of that particular component may be stored. In order to be able to distinguish one version from another, the version number of the component is added to the component's filename. For this reason a component must be assigned a version number before it can be exported to the repository.

RepositoryForm, shown in Figure 20, is the user's interface with the repository. The left frame of RepositoryForm contains a tree structure that displays the current contents of the repository. The right frame is used to display the skeleton code for a selected repository component. The skeleton code is displayed automatically when a component is selected. Components are imported into the current project by either double-clicking on a component's name or by selecting a component and choosing the Import command from the menu.

RepositoryForm can be used in two different modes, Normal mode and Import mode. Normal mode is used when users select the Repository command from the Tools menu of MainCanvasForm. In this mode all existing components compatible with the programming language of the current project are visible. Users have the options of importing a component into the current project,
deleting a component (or an entire folder of related components), and refreshing the repository view. Import mode is used when the user selects the "Import" command from one of the MainCanvasForm menus. In this mode only those components of a specified type are displayed. For example, if the user chooses "Import class" from MainCanvasForm, classes are displayed but interfaces are not.

**Other Forms**

The form *NewProjectDialog* appears when a new project is created and is used to select the programming language on which to base the new project. The form contains three language options, Java, C++ and Visual Basic, although at this point only Java projects can be created. The form *AboutForm* is similar to the form displayed in many applications when the user selects the "About Application" option from a menu. It displays general information about the program version and the author.

The form *UserMessageForm* was designed to be a replacement for the standard message dialog box provided by Visual Basic. It has been customized to display an icon indicating whether a particular message is a warning message or an error message. Messages are displayed in a scrollable text box so that lengthy messages can be displayed without the need for increasing the size of the dialog box.
Figure 20. RepositoryForm, showing skeleton code for the selected class.
**The Component Class Module**

The Component class module encapsulates the notion of an object-oriented component, whether it is a top-level component or a subcomponent. Methods are included for adding items to a component, deleting items, and changing items. The data for a component are stored in a Dictionary data structure, which can store key-value pairs and return a listing of all existing keys. A Dictionary also has the ability to store any combination of data types. The Component class can be used to store the data for any desired top-level component or subcomponent simply by using a different set of key-value pairs.

**The Project Class Module**

As its name implies, the Project class module encapsulates the notion of a design project. It contains the collection of the top-level components that form the design. It also contains methods for adding new components and deleting existing components, as well as methods for assigning and retrieving references to the supporting classes (LanguageSpecs, ComponentFactory, etc.) that it uses. While MainModule initiates the reading and writing of project files, it is the responsibility of the Project class to read and write the information for the individual components. Two separate flat text data files are maintained for a saved project. One file contains the information for the components. The other file contains all the information needed to properly construct and position the ObjectControls as they appeared on the screen when the project was saved.
The ComponentFactory and JavaComponentFactory Class Modules

The ComponentFactory class module is an abstract class with the primary function of creating or modifying top-level components or subcomponents. It includes methods for returning a newly created or modified component and for building an ObjectControl for a component. Since the commands for creating or modifying components are issued from the dynamic portion of the MainCanvasForm menu, the ComponentFactory class must be able to process a menu selection to identify the type of component involved. A method exists in ComponentFactory for this purpose. The JavaComponentFactory class module extends ComponentFactory and contains the implementation for creating Java language components. It has the additional responsibility of returning an instance of the proper data entry form that the user must use to enter the specific details for a component.

The LanguageSpecs and JavaSpecs Class Modules

The LanguageSpecs class module is an abstract class that encapsulates the syntactic rules of a programming language. This class is thus responsible for verifying that the data entered for a particular component obeys the syntax rules. It also contains the methods for parsing a component's data into the declaration for that component, as it appears in skeleton code. The JavaSpecs class is an extension of LanguageSpecs that encapsulates the syntax of the Java language. Additional subclasses of LanguageSpecs would be required in order to include support for other programming languages.
The MenuSwitchboard Class Module

Since the contents of the dynamic menu of MainCanvasForm can vary, the MenuSwitchboard class was created to process selections made from the dynamic menu and redirect execution to the form or class module designed to handle that event. Depending on the selection that is chosen, the state of the dynamic menu changes. For instance, when an ObjectControl representing a "class" component becomes the active ObjectControl, only the menu options that apply to a class component should be enabled. The MenuSwitchboard class is responsible for setting the state of the dynamic menu to reflect the current state of the program.

The ResourceCenter and JavaResourceCenter Class Modules

The function of the ResourceCenter is to provide access to the various resource data files that are required for a particular programming language. These resource files include the files that contain the menu options for the dynamic menu of MainCanvasForm, the categories of components and the icons used by the ObjectControl, and the icons used by the Repository. It is an abstract class that must be extended to provide support for a particular programming language. The JavaResourceCenter class module provides support for the Java language.
Software and Hardware Specifications

*SimpleOOD* is designed solely for use with the Windows operating system. It can be used with Windows98, Windows NT (service pack 6), and Windows 2000. It is recommended that the computer on which *SimpleOOD* is installed contain at least 128 Mb of RAM, and have a Pentium CPU. The *SimpleOOD* application itself is quite small, occupying only ~516 Kb of disk space. However, the application requires the installation of several dynamic link libraries (DLLs) and controls, written by Microsoft, which brings the total space requirement to approximately 9 Mb. Some of these files are typically present on most computer systems, while others are not. An arbitrary amount of additional free disk space is required for storage of user-generated repository components and projects. Therefore a conservative estimate of the recommended amount of total free disk space is approximately 20 Mb.

Procedures

The *SimpleOOD* application was implemented in an incremental fashion. Features were added one at a time and subjected to limited preliminary testing so that each feature could be immediately evaluated for feasibility and performance. This protocol allowed each feature to be re-designed, if necessary, at an early stage with minimal damage to the rest of the design. It also permitted closer adherence to the projected time schedule, since features could be eliminated if it became apparent that implementing them would cause undue delays.
Prior to release for beta testing, all forms and modules in the application were subjected to unit testing. Once all bugs discovered during unit testing had been removed, the application underwent limited system testing to ensure that it functioned properly as a whole. It was tested on three different operating systems: Windows98, Windows NT (service pack 6), and Windows 2000. It was also tested on several different computers to ensure that it could be installed and run properly under different system configurations.

The final application sent to beta testers was an installable package that automatically installs all required supporting files, such as dynamic link libraries and controls. Beta testers were asked to spend approximately one to two hours experimenting with the application and then to evaluate it using a provided questionnaire.

**Methods for Observation and Interpretation**

The three primary goals to be achieved in the completed application are usability, usefulness, and performance. Usability is crucial because the whole purpose of this project was to create an object-oriented design tool that can be used easily without experience in any formal design method. Developers who use the program should not face the steep learning curve that is characteristic of complex design tools. Usefulness is equally important since developers will not use the application unless it provides beneficial capabilities. In the case of performance, the application must be able to process user commands in a reasonable length of time. There should be no lengthy delays while waiting for a
dialog box to appear, and the application’s execution should not slow the overall system performance substantially. Additionally, the number of bugs in the final product should be minimal. The degree to which the application meets the above goals will be determined primarily by the evaluations of beta testers, since the problem being studied by this project is predominantly an issue of usability. Beta testers will be asked to answer the questions on the questionnaire shown in Figure 21. Criticisms of the beta testers will be used as guidelines for improving any future incarnations of the application.
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 21: Sample questionnaire for beta testers.
RESULTS

Unit Testing

The beta version of the application passed unit testing very well. All methods in all modules yielded the expected results for all test cases. During system testing it was discovered that the ObjectControl ActiveX control consumes a very large amount of memory. In one test on a system with 128 Mb RAM with 72% of RAM free initially, the creation of 58 classes (which produced 58 ObjectControl instances) brought the free RAM down to 0%. Although the 58th ObjectControl instance was created, it was unable to properly populate its icon image list. A failsafe that had been designed to handle situations where the data file that contains those icons could not be read successfully handled this problem by displaying an appropriate error message and using the default icon that had been hardcoded into the control. When an attempt was made to create a 59th ObjectControl instance, the operating system displayed an "Out of Memory" error message and shut down the program.

Bugs

Table 4 lists the bugs that were discovered in the application during beta testing. Bug #1 was discovered on a system with a Windows Desktop theme that was not using the default color scheme. The result was that inactive text boxes were blacked out, making the text contained in them unreadable. The bug also exposed mask colors that were used to make the edges of certain icons blend
into the background. Bug #2 resulted in the loss of one element of functionality, in that no files could be added under the GUI Components category. Bug #3 allows users to create a class or interface that inherits from itself, which is not allowed in Java.

<table>
<thead>
<tr>
<th>Bug</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>background colors of some controls were incorrect when the standard desktop color scheme was changed</td>
</tr>
<tr>
<td>2</td>
<td>neither *.java nor *.class files were accepted as valid java GUI component filenames</td>
</tr>
<tr>
<td>3</td>
<td>the Superclass listbox on the ClassDialog allowed a class to be a subclass of itself</td>
</tr>
</tbody>
</table>

Table 4. Bugs discovered during testing.

**Evaluations of the Beta Testers**

Only two beta testers provided evaluations of the application. Finding volunteers to beta test the application proved to be more difficult than expected. Partly this was due to the fact that most people who were asked to be beta testers had little or background in Java. In fact, one volunteer had minimal programming experience, but happened to have extensive visual arts experience. This individual found the ObjectControl an excellent organizational feature and provided some suggestions on how to use the icons in the ObjectControl to greater advantage.

The second volunteer had Java experience, although it was not extensive. This individual found the skeleton code generation feature to be useful, and provided some very good suggestions on how to improve the skeleton code that
is generated. For instance, each category of subcomponent has a heading in the skeleton code. It was suggested that if a class contains no constants, for example, then there should be no heading in the skeleton code for constants. Additionally, all subcomponent declarations are double-spaced in the skeleton code. It was suggested that for constant and variable declarations an option be provided to single-space those declarations. Finally, this individual pointed out that finding the generated skeleton code on the disk was not intuitive.

Both beta testers were able to learn the basics of the application in a relatively short period of time (1-2 hours). Creation of the various components and subcomponents was found to be quite simple. It was agreed by both testers that the repository feature was useful because of its ability to store reusable pieces of code. No problems were reported with saving projects or exporting components to the repository. No instances were reported where the application froze, locked up the system, or crashed unexpectedly.
DISCUSSION

Only three bugs were discovered in *SimpleOOD* during beta testing. Analysis of the causes of those bugs indicates that they can all be corrected easily. The bug that prevented *.class* and *.java* files from being recognized as valid Java files prevented the addition of GUI component files, and thus made that element of functionality unavailable to users. The other two bugs were relatively minor and did not significantly interfere with the use of the application. The fact that no crashes or freezes were reported suggests that the application is robust, at least within the limits of the testing that was performed.

The most serious problem encountered was the large resource consumption of the ObjectControl ActiveX control. This problem imposes a severe limit on the size of the project that can be created using the application, and can create additional difficulties if multiple applications are running simultaneously. Since the ObjectControl is very graphic-intensive, one approach to solving this problem would be to streamline the way the control displays its images. Currently each ObjectControl instance contains its own image list for the icons that it uses. These multiple image lists are redundant and could be centralized by storing a single image list in MainCanvasForm. There are also more efficient methods for displaying graphics. One example is the use of Windows metafiles (*.wmf) as opposed to the bitmaps that are currently used.

Overall it is concluded that the *SimpleOOD* application is a potentially useful design tool. Based on the opinions of the beta testers, the ObjectControl
ActiveX control is intuitive, and it promotes the organization of top-level object-oriented components and their subcomponents. The tool's ability to generate skeleton code reduces the chances for syntax errors, since users must type less code. The fact that the beta testers were able to quickly learn how to use SimpleOOD implies that it may have a reduced learning curve compared to similar tools. Additional testing, involving direct comparisons between SimpleOOD and several similar tools, is required before this conclusion can be drawn with certainty.

**Future Work**

In its current form the SimpleOOD application is somewhat limited in its capabilities. Several improvements can be made to increase its versatility and usefulness. The addition of support for other object-oriented languages such as C++ would allow the tool to be used by developers who are not familiar with Java. There are several ways in which the skeleton code feature can be enhanced. For instance, when a Java class inherits from an interface it must provide implementation for all the methods in that interface. It would be helpful for the skeleton code that is generated for such a class to automatically include the declarations for those interface methods. In order to make SimpleOOD more compliant with the Java language specification, support needs to be included for the creation of nested classes that are contained inside other classes.

One very useful design tool that SimpleOOD does not currently provide is a way to show interactions between the various components in a design. An
ideal way to implement such a feature would be to allow users to specify interactions on the same canvas that is used for placing the top-level components. This would allow users to visualize both the components and their interactions on the same screen, as opposed to having to refer to separate documents, one for the components and one for the interactions.
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


62


