Proceedings of the 1987 Winter Simulation Conference
A. Thesen, H. Grant, W. David Kelton (eds.)

SOFTWARE REUSE AND SIMULATION

Richard Reese
General Dynamics
Data Systems Division
P.O. Box 748 - MZ 2203
Ft. Worth, TX 76101

Dana L. Wyatt
North Texas State University
Dept. of Computer Science
P.O. Box 13886
Denton, TX 76203

ABSTRACT

Software reuse technology has the potential of impacting significantly the software development process. This paper addresses issues of software reuse and their application to simulation software. A simple classification scheme is presented to assist in the identification of software components. Historical approaches to reuse are then discussed along with desirable attributes of reuse systems and components. Following this, impacts of reuse technology on the software development process are discussed. The paper concludes with sections on the application of reuse to simulation and a methodology for developing reuse systems.

1. INTRODUCTION

The underlying concept of reuse is the identification and extraction of software components from other projects. Software reuse is an old concept which has recently received renewed attention as a result of efforts to address the high cost of software development. Contributing to this renewed interest has been advances in hardware performance, artificial intelligence, database systems, and in programming language design. For example, the DoD's considerable investment in the development of Ada and supporting tools is partially a result of the belief that features of Ada will support reusability.

Software reuse is defined as the isolation, selection, maintenance, and use of software components in the development and maintenance of a software project. A component is any element of a software system and can include elements from requirements specification, design, code, and testing. The term, software component, refers to any documentable aspect of the project. For example, a software component may be a user's guide, data flow diagram, high-level language code, or test plan. It is important to note that reuse is not limited to code, but can be extended to include any element of a software development project that requires effort.

To make reuse possible, it is necessary to develop a methodology for the reuse of these components. This method must encompass not only the use of the components, but also the initial selection, incorporation, maintenance, and overall management of the components. This reuse process, as shown in Figure 1, illustrates the interaction of the user, administrator, and the reuse system. The reuse administrator is responsible for the development and maintenance of a database of reuse components. Users access the system as needed to search for and select components.

The primary benefit often cited for the reuse of software is the expected improvement in productivity. If components are reused, then less time will be required to develop systems. In addition to productivity, other benefits include improved quality, reliability, maintainability, and timeliness of the software system.

2. TAXONOMY OF REUSE COMPONENTS

Many facets of a software development project have potential for reuse. Each phase and activity in the software development process is repeated for most projects. The key to successful reuse is to specify components in such a way that future projects might benefit.

A reuse system is based upon a set of underlying components which can be classified and quantified. This section will provide a simple classification scheme to clarify the term component. Section 4 will address methods of quantifying individual components.

The taxonomy used herein is based upon two factors: the level of reuse components and the functionality of reuse components. The term, level, is used to denote a grouping of components according to phases of the software development process. A functionality classification groups components by their application area. The complexity of component classification is more complex then presented here. A more in-depth discussion of this issue is presented in (Prieto 1897, Kartashev 1986).

2.1. Levels of Reuse Components

Components may be classified by the software development life cycle phases or activities. The result of such a classification scheme might contain components relating to the following:

Requirements Specification
Design

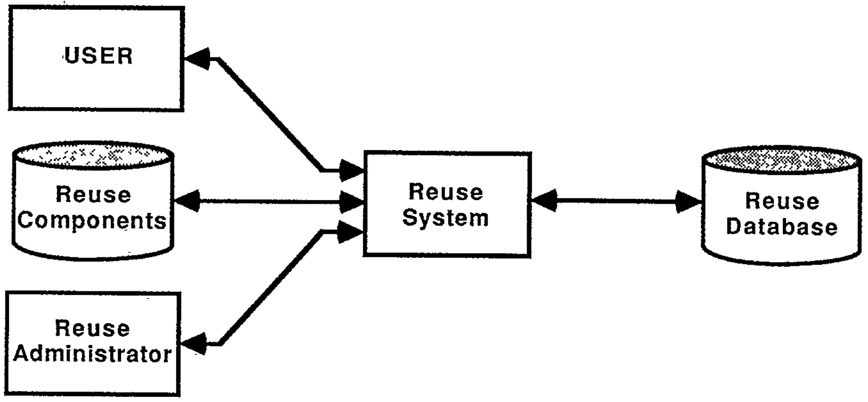

Figure 1: Reuse System Process

Code
Testing
Verification/Validation
Configuration Management
Manuals

The documents produced by each of these activities may be treated as components for reuse. The effort required to select and insert these components, as well as their eventual usability, will vary from component to component.

Examination of the software development activities reveals that coding is only one activity in the process and, as pointed out in Boehm's text (Boehm 1981), is not always a major activity in terms of effort. Figure 2 illustrates the average activity distribution for medium-size software projects. The left-hand column represents the activities performed in each phase listed on the top row. These numbers, representing percentages, are based on COCOMO estimates (Boehm 1981). For example, the coding phase consumes 54% of the total activity in a software development project, yet only 55% of that activity is devoted exclusively to coding.

This figure emphasizes that there are other areas where reuse may be applicable. Notice that the maintenance effort is not shown. Yet, maintenance is usually the major cost of a project, making up 70-90% of the total life-cycle cost. Reuse should also address this activity to lower the cost of this phase.

2.2. Functionality of Reuse Components

This classification scheme is based on a grouping of components by application. There are hundreds of possible functional application areas from which to choose. This paper addresses the simulation-specific functional components in section 6. Examples of other functional areas include: I/O, graphics, statistics, and communication protocols.

A functional classification can aid in the component selection process. It does not preclude the classification of a component under two or more groups.

3. REUSE APPROACHES

Historically, there have been attempts to develop reusable software routines. These are commonly represented as a collection of utility subroutines. However, these have tended to be very primitive in nature. More ambitious attempts in the effort to reduce or eliminate the software development process have recently surfaced in the form of program generators. Although somewhat successful, they tend to support a very limited domain.

	Plans and Requirements	Product Design	Progrm-ming	Integration and Test
	8	18	54	28
Requirements	46	10	3	2
Design	14	42	6	4
Programming	6	12	55	40
Test Planning	4	6	6	4
Verification and Validation	8	8	10	25
Project Management	12	11	7	8
CM/Quality	4	3	7	9
Manuals	6	8	6	8

Figure 2: Activity Distribution

3.1. Subroutine Libraries

Subroutine libraries are generally implemented as libraries of functions or procedures written in a single language and stored in object module format. The operating system's linker accesses the library to obtain the needed modules.

This is a simple, low level approach. It lacks any degree of user interface sophistication. In addition, the libraries generally consist of components which are too small to be of practical use in large-scale projects. The granularity of the components is not conducive to designing-in-the-large.

The language in which the subroutines are written limits their applicability to projects written in other languages. While it is possible to combine subroutines written in different languages, it is often a difficult process because of interface and data structure incompatibilities. For example, modern block structure languages such as C, Pascal, and Ada use a program stack to manage subroutine invocations and recursion. Older languages do not support this approach. Difficulties may even arise because of different calling conventions among compilers on the same machine.

Data structure incompatibilities also account for reduced reusability. Different memory allocation schemes for the storage of structures and varying internal representations for primitive data types are common differences in languages. Advanced data types such as pointers and enumeration types may also differ considerably from one language to another. Support, or lack thereof, of dynamic memory allocation further complicates the matter. These differences all contribute to the difficulty of combining subroutine components of different languages.

The development of Ada packages is offering some promise towards reducing the difficulties of reuse procedures. Generic packages containing useful operations on a generic data type allow a programmer to create instances of the package for various data types. For example, consider a routine which swaps two values:

```
TEMP = A;
A    = B;
B    = TEMP;
```

This swap procedure might be needed for integers, reals, and records. Many languages would require the development of separate routines for each conceivable type even though the only difference is the typing of the variables. Ada, however, allows creation of a generic routine in which a data value and its corresponding type may essentially be passed to the routine, thus avoiding the need for the programmer to develop separate routines.

In general, however, subroutine components typically leave little room for variation. They are usually fixed in their function, interface, and frequently in their data structure. Their applicability to a general reuse philosophy is limited since they do not have the flexibility to adapt readily to different applications.

3.2. Software Generators

The current state-of-the-art limits the applicability of software generators for three reasons. First, most software generators produce a large amount of code in the process of creating an application. This is often due to inherent inefficiencies in the generator and in the difficulty of accurately specifying an application's requirements. Without a large amount of specification, it is difficult to prune unwanted functions from the system. As more options are provided for specifying the nature of the generated software, the generator becomes more complex.

Secondly, existing software generators have been limited to well defined and understood application domains. Software generators for simulation have been developed (Mathewson 1981). However, the nature of many simulation projects is such that the application area is often new and not thoroughly understood. As a result, software generators are rarely available.

Finally, the software generated by a software generator is often difficult to enhance because of poorly documented source code. This is because the flexibility exhibited by the generated software is dependent upon many of the same factors which make code reusable, such as data structures and interface design.

4. ATTRIBUTES OF REUSE SYSTEMS AND COMPONENTS

This section discusses those factors which contribute to successful reuse systems and components. The reuse system is essentially a storage and retrieval system used to access individual components.

Central to both a reuse system and its components is the issue of quality. The component and system should be of high quality as there is no substitute for a high quality product.

4.1. Attributes of Reuse Systems

There are several attributes which can contribute to the successful reuse component: language independence, composition of components, standard interfaces, accessibility, and selection.

4.1.1. Language Independence. The reuse component, if at all possible, should be language independent. This will increase its range and application because it is not limited to a single language. This often means that the underlying algorithm of a code component is embedded in a psuedo-language, a language which allows translation from the psuedo-language to the target language. Several systems use this approach (Bassett 1987, Kaiser 1987, Lenz 1987).

4.1.2. Composition of Components. The reuse system should allow easy construction of complex systems from elementary components. The term, building block, is often used to describe this methodology. It is based upon concepts borrowed from more mature disciplines such as electrical engineering in which construction of a system is made from a more or less standard set of components. However, this analogy breaks down for a number of reasons.

Computer science is not as mature as other fields in which standards and theories are better defined and understood. Concrete identification of these building blocks is essential. The nature of computer science is such that standards may never be present except in a few well-defined and understood areas of application.

4.1.3. Standard Interfaces. Standard interfaces are an important cornerstone for successful integration. A difficult integration may become a very expense and time consuming part of a project. The interface design will stay with the component throughout its existence, therefore, a good interface design is paramount.

4.1.4. Accessibility and Selection. When a component is to be selected and subsequently used, it is important that it be readily accessible. If it is not then the customer will not be as eager to spend the time it may take to access the component. The problems of centralized and distributed libraries come to bear. A centralized library makes it easier to maintain and control the components but is often not easily accessed. A distributed library is easier to access but is more difficult to maintain and control.

A good selection with a wide range of options is important. If the user can't find it, he won't use it. If he consistently fails to find what he needs, he will stop using the reuse system.

A significant reuse problem is the storage and retrieval of the project's components. Questions to consider include:

What format to use for storage?
How to fragment the components?
Which attributes should be used
for retrieval?

These problems demand careful consideration as initial decisions will set the tone of the reuse system for most of its life. If an inappropriate format is used then it may be difficult to add new component attributes or to obtain new information.

4.2. Attributes of Reuse Components

There are several criteria which can be used to evaluate software components being considered for reuse. Figure 3 is a list of many of these criteria (Adapted from Tracz 1987).

The more important aspects of a successful component include a standard set of features and options, a usage and maintenance record, documentation, and the ability to adapt to new uses. The other attributes are important, but usually found in a limited domain.

4.2.1. Component Features. In the evaluation of software, it is important that the software provide the minimum set of standard features for the function being performed. If the software function is that of a standard queue, then it should support the basic queue operations of enqueue, dequeue, provide some mechanism to handle the queue full condition, and provide a test for queue empty.

Options will be important for some implementations but will be an expensive extra for others. An example option for a queue might be a priority mechanism which would dequeue elements on a priority basis. Rearrangement of the order of elements might also be useful in some applications. These options are not free and the impact of any option should be carefully documented. An important point is that a given feature may be an option for one application and a necessity for another. The options should be packaged to reflect these differing viewpoints.

4.2.2. Usage and Maintenance Records. A usage record of a component should be maintained. The record would typically include information about the number of times selected, its frequency of use, how long the component has existed, and possibly the types of applications where it has been used.

The maintenance record should contain information relating to the frequency of bug detection and correction. The severity,

Quality
Standard Features
Options
Usage Record
Maintenance Record
Reputation
Warranty
Appearance
Standards/Documentation
Accessibility
Price
Trial Access
Adaptability
Execution Characteristics
Standard Interfaces

Figure 3: Evaluation Criteria for Software Reuse

type, and corrective action should be recorded. 'This record, when used in conjunction with the usage record, can provide a potential user with insight into the durability and reliability of the component. If the component has been corrected frequently or if modifications to the component typically result in a large number of subsequent bugs, then the quality of the component may be questioned. A component with a few bugs found earlier in its usage history with few or none found later suggests a robust and reliable component.

4.2.3. Documentation. The documentation which supplements the component is crucial to understanding and maintaining the component. The quality of the documentation is as important as the code itself. The code appearance often reflects the care put into the development of the code. Difficult to follow and contorted control sequence may be indicative of poor design. However, readability is sometimes sacrificed for efficiency and should therefore be considered when evaluating the appearance of the component.

4.2.4. Adaptability. The adaptability of the component is often an important factor in the selection process. If the component is not exactly what was needed, then a hopefully large percentage of it may be usable if modified. The ease of this modification effort is important as well as the number of variations possible. A good structured design will go a long ways to improving the adaptability of a component.

5. IMPACTS OF REUSE

The concepts of reuse can be applied throughout the software life cycle and impacts all attributes which are commonly used to quantify software. It affects software and the development of software in terms of productivity, maintainability, portability, and quality.

5.1. Productivity

One of the primary reasons for reusing software is to achieve an increase in productivity. However, it is one thing to say that it will be increased and another to actually demonstrate conclusively that the improvement will or did actually take place. Central to this issue is the metrics used to measure productivity. The two most commonly used metrics have been lines of code for the general computer science community and Albrecht's Function Points (Albrecht 1983) for the data processing community. Both have been confined largely to the coding phase in spite of the fact that the majority of effort occurs elsewhere. Thus, to use either lines of code or function points is to neglect, or at least gloss over, the effort performed in these other areas. Until an unbiased and all encompassing set of metrics is developed, no reliable or accurate productivity measurement will be consistently available.

So where does that leave the claim of increased productivity due to reuse? Claiming that reuse will improve productivity is justifiable in most environments when a core set of software components can be identified. While the reliability of the estimate may not be extremely high, it will be sufficiently accurate to provide justification for the productivity improvement claim.

5.2. Maintainability

Maintenance is generally considered to be the most expensive aspect of the total life cycle cost of a system. This cost can be decreased if systems are built which are robust and relatively error free. If reuse components which have been thoroughly tested in previous applications are used, the chance of discovering errors will be reduced. This will contribute to less effort spent on the maintenance of software.

A high quality, reusable component will make the software maintenance phase considerably easier. If the component library is one which is continually enhanced and expanded, then future enhancements may be partially taken care of by the growth of the library. Assuming that the library's components possess consistent interfaces and data structures, the addition of new components to existing components of the system should be simpler.

5.3. Quality

The overall quality of a software system developed from reusable components will be better than a functionally equivalent system developed without reusing software. The primary reason for this improvement originates from the additional effort required up front to insure the interfaces are complete and consistent and from the prior effort which went into the development of the components.

A major problem with many large systems is the difficulty of integrating the system as a whole. Poorly understood and changing interfaces are often present during project development. This difficulty will be partially lessened as the reused components become stable. In addition, the quality of the individual components should be better as they have already been field tested.

5.4. Future Impacts

In the long run, it may prove that reuse will serve as a tool to improve the ability to measure productivity. As answers to the questions of what to reuse and how to fragment a project for reuse are answered, we will begin to better identify the building blocks which constitutes many systems. These building blocks will represent the standards by which future effort can be compared and estimated. It may be found that a particular network protocol implementation always contains a certain set of core components and a secondary set of components which enhance and expand the protocol's capabilities. These will become the basis for estimating the total effort require for modification and integration of the network protocol.

The methodologies of reuse will also affect the way software is developed. Instead of designing unique software components for each new system, available components will be examine to determine if they are applicable. Attempts will then be made to design the system with these components in mind. This implies that a successful component must be of high quality and must exhibit flexibility.

6. APPLICATIONS TO SIMULATION

The issues of reusability discussed herein apply to all types of software, including simulation software. The use of these techniques are particularly adaptable to the area of simulation software. Long known for its cost overruns and poor performance, simulation software often has a credibility problem in the eyes of management (Russel 1983). However, adoption of a reuse philosophy and the subsequent creation of a reuse library by management is expected to improve the simulation software development process and increase the credibility of simulation results.

It is expected that initial management reaction will be skeptical because simulations are often viewed as one-time, or disposable, software products. This contrasts to other types of software such as graphics, in which the potential for reuse is recognized easily. This section will discuss areas of simulation model and program development which lend themselves towards a reuse environment.

6.1. Areas of Reuse

If one were to ask a simulationist if reuse potential existed in the development of models and/or programs, the answers would vary. However, most would agree that, in discrete simulation, there are standard procedures needed to support a simulation application. In addition, many would argue that certain patterns exist in models of a given domain. The separation of functions into simulation support functions and model functions will form the basis for discussion of potential reuse areas.

6.1.1. Simulation Support Function Level. The support functions required in virtually all discrete simulation applications include time management, queue management, statistics collection, random number generation and data I/O. This is one of the more obvious areas of discrete simulation in which reuse technology might be beneficial. In fact, the GASP (Pritsker 1974) and SIMPAS (Bryant 1981) simulation languages used this observation to develop subroutine libraries for high-level languages which make simulation from these languages simpler. Special purpose simulation languages such as SIMSCRIPT (Russel 1983) and SLAM (Pritsker 1984) also recognize this need and incorporated these support functions into the language itself.

However, examination of a 1983 survey by Christy (Christy 1983) indicates that a large percentage of the simulation software developed across the country was being done in languages other than those designed for simulation. The top five languages, in order, were: FORTRAN, GPSS, special purpose languages (such as IPPS, SIMPLAN, and EXPRESS), BASIC, and PL/I. Even though this survey is several years old, it is expected that the only language which might significantly alter the survey results is Ada.

The benefits of a reusable library of simulation support routines include a reduction in the effort required to manage the development of simulation programs and increased time available for the development of models. These support libraries can be extended to include debug routines, validation and verification tools, and graphics and animation support.

6.1.2. Model Function Level. The reuse of model components is a less obvious, but equally viable area for reuse. This is especially true in domain-specific environments in such areas as manufacturing, CAD/CAM, and computer networking.

It is already known that companies which specialize in a domain-specific problem area have tools which aid in the development of models. In particular, these tools often take the form of model and/or program generators. For example, computer vendors often offer services to customers to aid in the configuration of systems for a particular purpose. To adequately determine the configuration, these vendors have departments which develop simulation models of a proposed configuration to determine the performance of the proposed configuration. However, the development of these models is sometimes tedious and can take several days.

To improve productivity, some vendors have developed libraries of models which can be extracted and altered using automated software generation tools. Using this method, models of proposed configurations can be developed much more quickly and efficiently. Errors are reduced and the confidence in the model results improves. This is an example of reusable components in a domain specific environment. However, it is not necessary that the automation of model development be taken this far given the existence of a taxonomy, storage, and documentation strategy that would ensure the utility of a reuse library.

7. ESTABLISHING A SIMULATION SOFTWARE REUSE SYSTEM

Once the decision has been made to develop a software reuse system, this decision needs to be carried through to completion. This section addresses the issues which will effectively accomplish that goal. A basic assumption of this section is that the system is to built within an organizational framework.

The process of developing a software reuse system is, in essence, an iterative one. Once the system is built, it must be utilized and maintained. The successful continued use of the system will require ongoing management support and the establishment and evaluation of system objectives. Figure 4 outlines a set of steps which can be used in the development of a software reuse system. Each step will be discussed in varying detail.

It is crucial to obtain upper management approval and commitment for this development effort. If management does not support this effort with the proper commitment of resources and encourages the system's subsequent use, then the system will not realize its full potential.

It is necessary to establish an individual or group with the responsibility for the system. If this is not accomplished, then there will be a lack of system accountability. In addition, there should be a focal point to reduce duplication of effort and provide standards. It will be that individual's responsibility to ensure that the system's objectives and plans are accomplished.

The successful reuse system must be accepted by the people who will use the system. This acceptance must be cultivated and can not be demanded. Support should be developed as early as possible, in order to assure an accurate assessment of the system's needs.

While sometimes overlooked, the specification of objectives, alternatives, and constraints must be developed. The objectives provide a goal. The alternatives provide options to be traded against often

1. Obtain upper management commitment
2. Assign responsibility for the system
3. Cultivate broad-based support
4. Identify objective, alternatives, and constraints
5. Evaluate alternatives and select the best
6. Prepare phased implementation plan
7. Obtain authority to proceed
8. Implement the plan
9. Follow up and reiterate the plan

Figure 4: Steps in the Development of a software reuse system

conflicting goals, subgoals, and constraints. The constraints will define the limitations of the system which may be imposed for any number of reasons. The objectives, alternatives, and constraints need to be clear, precise, and complete.

Once the alternatives have been examined, it is necessary to evaluate them in light of the constraints imposed upon the system. Whenever possible, economic benefit analysis should be performed based upon as unbiased set of evaluation rules as possible. The alternatives should not be considered in isolation only, but also in combination with each other. This helps to avoid possible sub-optimization decisions.

Once the system has been carefully evaluated, a phased implementation plan should be developed. Careful consideration should be given to the plan to insure its correctness and completeness. Authority is then needed to precede once the plan has been accepted. This is then followed by the implementation step.

These steps, while sequential, require re-examination of preceding steps of the process because one step will impact the actions taken in another. This impact normally has minimal impact upon earlier steps, but could be significant if inherent problems are not adequately resolved early.

There is a price to pay for any reusable scheme, that it the cost of education. There will be a definite learning curve which has to be overcome before the full power and utility of the system becomes available. Any organization which seriously attempts to reuse software will eventually have to pay this price.

8. CONCLUSION

General issues relating to software reuse and its application to simulation software were addressed in this paper. An important point developed was the concept that a software component is not limited only to code. Reuse can be applied to any documentation used in the development of a project. In addition, a simple classification scheme was presented to assist in the identification of software components. The last section of the paper presented a general method for developing a reuse system.

The reuse of software requires commitment and effort to properly accomplish the goals of reuse. It also involves a modification of the software development process. More effort must be made initially to insure that components will be reused. Designers must begin thinking of reuse as an initial step in the design process.

Reuse systems and components must be developed. Components will most likely be developed from scratch so as to incorporate the desirable features of a high quality component. Reuse must be engineered from the start. Attempts to retrofit existing software into components generally require more effort than to develop the components from scratch.

The basic concepts behind reuse are straightforward. What is needed is an effort to develop reusable components, incorporate them into a reuse system, and to modify the current development methodologies to take advantage of this technology. Reuse will not work without a commitment and an application of effort to this end.

REFERENCES

Albrecht, A., and Gaffney, A. (1983). "Software Function, Source Lines of Code, and Development Effort Prediction: A SOftware Science Validation", IEEE Transactions on Software Engineering, 9, 6, 639-648.

Bassett, P. (1987). "Frame-Based Software Engineering", IEEE Software, 4, 4, 9-16.

Boehm, B. (1981). Software Engineering Economics, Prentice-Hall, Englewood Cliffs, New Jersey.

Bryant, R. (1981). SIMPAS User's Manual, Department of Computer Science and Academic Computing Center, University of Wisconsin - Madison, Madison, Wisconsin.

Christy, D., and Watson, H. (1982). "The Application of Simulation: A Survey of Industry Practice", Interfaces, 13, 5, 47-52.

Kaiser, G. and Garlan, D. (1987). "Melding Software Systems from Reusable Building Blocks", IEEE Software, 4, 4, 17-24.

Kartashev, S. and Kartashev, S. (1986) "Guest Editor's Introduction", Computer, 19, 2, 9-13.

Lenz, M., Schmid, H. and Wolf, P. (1987). "Software Reuse through Building Blocks", IEEE Software, 4, 4, 34-42.

Mathewson, S. (1981). A DRAFT II/SIMON Manual, Department of Management Science, Imperial College, London.

Prieto-Diaz, R. and Freeman, P. (1987). "Classifying Software for Reusability", IEEE Software, 4, 1, 6-16.

Pritsker, A. (1974). GASP IV Simulation Language, John Wiley and Sons, New York, New York.

Pritsker, A. (1984). Introduction to Simulation and SLAMII, Halsted Press, New York, 2nd Edition.

Russel, E. (1983). Building Simulation Models with SIMSCRIPT II.5, CACI, Inc., Los Angeles, California.

Tracz, W. (1987). "Reusability Comes of Age", IEEE Software, 4, 4, 6-8.

AUTHOR'S BIOGRAPHIES

RICHARD M. REESE is a software design specialist with the Data Systems Division of General Dynamics where he is a member of the Product Software Group. Currently, his responsibilities include the development of a division-wide software cost estimating methodology and the development of a network protocol. He received a B.A.S.S and M.S. from Stephen F. Austin State University in 1978 and 1979. In 1983, he received a Ph.D. in computer science from Texas A&M University. His research interest include the areas of software cost estimating, software engineering, simulation, and software reuse. He is a member of the ACM and IEEE.

Richard M. Reese
General Dynamics
Data Systems Division
P.O. Box 748 - MZ 2203
Ft. Worth, TX 76101
(817) 777-3992

DANA L. WYATT is an assistant professor in the Computer Science Department at North Texas State University. She received a B.A.S.S and M.S. from Stephen F. Austin State University in 1978 and 1979. In 1986, she received a Ph.D. in computer science from Texas A&M University. Her current research interests include distributed simulation, simulation support tools, and databases. She is a member of the ACM, IEEE, and SCS.

Dana L. Wyatt
North Texas State University
Department of Computer Science
P.O. Box 13886
Denton, TX 76203
(817) 565-2767