IMPACTS OF OBJECT-ORIENTED TECHNOLOGIES: SEVEN YEARS OF SEL STUDIES

Mike Stark

SOFTWARE ENGINEERING BRANCH
Code 552
Goddard Space Flight Center
Greenbelt, Maryland 20771
(301) 286-5048

ABSTRACT

This paper examines the premise that object-oriented technology (OOT) is the most significant technology ever examined by the Software Engineering Laboratory. The evolution of the use of OOT in the Software Engineering Laboratory (SEL) "Experience Factory" is described in terms of the SEL's original expectations, focusing on how successive generations of projects have used OOT. General conclusions are drawn on how the usage of the technology has evolved in this environment.

INTRODUCTION

The Software Engineering Laboratory (SEL) sponsored by the National Aeronautics and Space Administration/ Goddard Space Flight Center (NASA/ GSFC), has three primary organizational members: the Software Engineering Branch of NASA/GSFC, the Department of Computer Science of the University of Maryland, and the Software Engineering Operation of Computer Sciences Corporation. It was created in 1976 to investigate the effectiveness of software engineering technologies applied to the development of applications software. As it seeks to understand the software development process in the GSFC environment, the SEL measures the effects of various methodologies, tools, and models against a baseline derived from current development practices.

In the SEL production environment, the language usage is approximately 70 percent FORTRAN, 15 percent Ada, and 15 percent C. This is in contrast to the almost 100-percent FORTRAN environment in 1985. Projects typically last between two and four years, and they range in size from 100,000 to 300,000 source lines of code (SLOC). A typical project consists of between 20 percent and 30 percent code reused from previous projects.

The SEL has examined many technologies, some of which have major effects on how software is developed in the SEL production environment, where ground-support software is produced for the Flight Dynamics Division (FDD) at Goddard Spaceflight Center (GSFC). One technology, Object-Oriented Technology (OOT), has attracted special notice in recent years, causing Frank McGarry, head of Goddard's Software Engineering

Branch, to remark a year ago that "Object-Oriented Technology may be the most influential method studied by the SEL to date" (Reference 1).

THE EXPECTATIONS AND REALITY OF OOT

The development of highly reusable software is one of the promises of OOT. The initial expectation for OOT was that this increased reuse would yield benefits in the cost and the reliability of software products. In addition, it was expected that OOT would be more intuitive than the structured development traditionally used in this environment, making the development process more efficient. Therefore, the SEL expected that, in addition to the reuse benefits, the cost of developing new code would also decrease.

The specific measures applied to assess the effect of OOT include cost in hours per thousand source lines of code (KSLOC), reliability by measuring errors per KSLOC, and the duration of the project in months. To date, OOT has been applied on eleven projects in the SEL. These projects can be grouped into three families of completed projects and an ongoing effort to develop generalized flight dynamics application software.

The completed projects (Figure 1) include three early Ada simulators built between 1985 and 1988, as well as three FORTRAN ground-support systems developed from the Multimission Three-Axis Attitude Support System (MTASS) and four telemetry simulators developed from multimission simulator code, all of which multimission applications were developed between 1988 and 1991.

During the seven years the SEL has been experimenting with OOT, developers have gained more understanding of which object-oriented concepts are most applicable in the FDD environment. The most important part of the evolution is the application of object-oriented concepts to a greater portion of the development life cycle over time. The knowledge gained during the development of these three families of systems is being applied in the development of generalized flight dynamics applications.

Despite its later appearance chronologically, the MTASS family of systems (Figure 2) should be

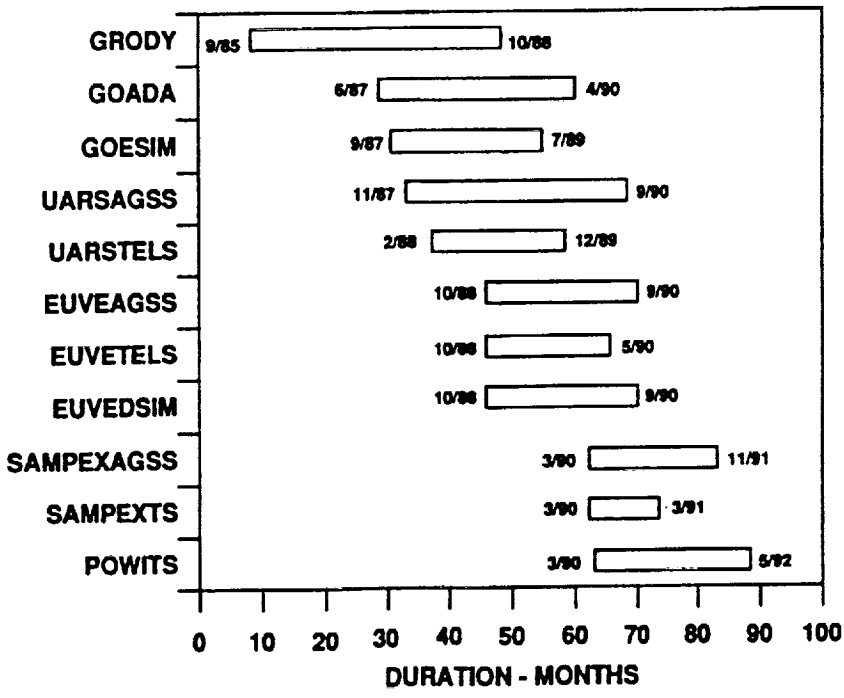

Figure 1. Projects Using Object-Oriented Technology

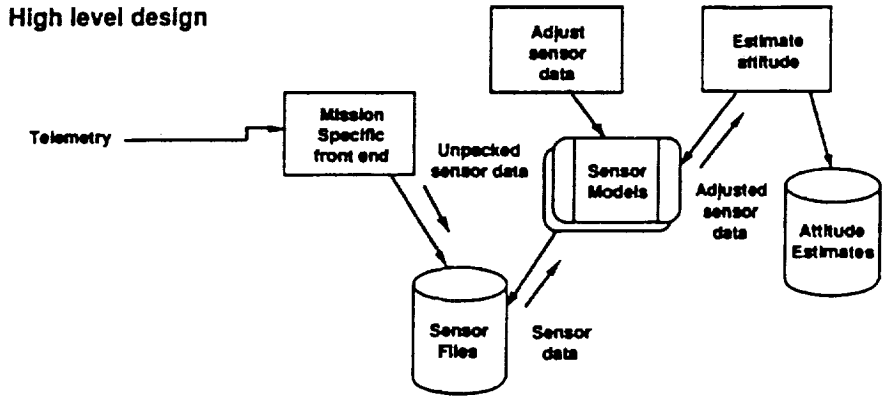

Figure 2. MTASS Design

examined first because it represents a modest infusion of OOT. MTASS started with a ground-support system that was developed as a common system for two different satellites, the Upper Atmosphere Research Satellite (UARS) and the Extreme UltraViolet Explorer (EUVE) satellite. It was then reused for the Solar, Anomalous, and Magnetosphere Particle Explorer (SAMPEX).

All ground-support systems read in telemetry and produce attitude (spacecraft orientation) estimates. The difference is that, where previous systems had stored all sensor data in one file specifically designed for the mission, MTASS developed separate interface routines and file formats for each kind of sensor. Only one mission-specific, front-end telemetry processor had to be developed for each new mission.

This basic grouping of data and of operations on the data is the most important object-oriented concept in the FDD environment. This change alone increased code reuse from the baseline 20 percent to 30 percent to around 75 percent or 80 percent.

It should be emphasized that the use of OOT on these projects was modest. The implementation language is FORTRAN, and the standard structured design notation was used to document the system. The object-orientation of the sensor model design was recognized during coding rather than consciously planned during design. Nonetheless, this one simple concept has had tremendous benefit in developing ground-support software faster and at a lower cost.

The earliest purposeful use of object-orientation in the SEL environment was associated with the introduction of Ada in 1985. The first Ada project, the Gamma Ray Observatory (GRO) Dynamics Simulator in Ada (GRODY), was developed as an experiment in parallel with an operational FORTRAN simulator. Previous Ada experiments (Reference 2) had produced designs and code that looked like Ada versions of FORTRAN systems. To avoid this, the GRODY team was trained in a variety of design methods, including Booch's Object-Oriented Design (OOD) method (Reference 3), stepwise refinement, and process abstraction. In addition, one of the team members had an academic background in OOD.

OOD emerged as a clear favorite, but in early 1985 Booch's method was not mature enough to support large production projects. Stark and Seidewitz developed the General Object-Oriented Design (GOOD) method during the GRODY project to meet these needs (Reference 4). Its first application was on the Geostationary Operational Environmental Satellite (GOES) Dynamics Simulator in Ada (GOADA), a project started in 1987. The GOES Telemetry Simulator (GOESIM) was also implemented in Ada. GOESIM was developed using structured design techniques, although GRODY packages designed with an object-oriented approach were reused on GOESIM.

The goal of the early Ada simulation projects was to learn the appropriate use of the Ada language, with a view towards increasing software reuse. Other goals were considered less important. The GRODY team, for example, was specifically instructed not

to worry about the real-time requirement being imposed on the FORTRAN simulator, and in fact GOADA was able to achieve higher than usual reuse from GRODY code. However, the lack of attention to performance led to systems with disappointing performance.

The SEL responded to this issue by studying the performance of the GOADA simulator in detail to determine if the performance problems were caused by the Ada language, the OOD concept, or by the GOADA design itself. The studies estimated the effect of various improvements on the execution speed of a simulation. These improvements included changes such as removing repeated inversion of the same matrix from an integrators derivative function or simplifying the internal data structure of an objects state. The inefficiencies were not caused by the use of object oriented technologies, and improving the performance with these corrections would not compromise the object-oriented design. Figure 3 shows that making all these changes to the full simulator would improve performance to the levels attained by similar FORTRAN simulators.

The next generation of projects is a multimission telemetry simulation architecture, built around Ada generic packages. Figure 4 shows how two sensor models use a generic sensor package for common functions such as writing reports and simulated data files. Here, each sensor has its own specific modeling procedure that is used to instantiate the generic. In addition, these model procedures are built around other generics that provide common functionality such as modeling sensor failures or digitizing simulated sensor data. The arrows indicate dependencies between software modules. For example, the Gyro object depends on procedure Gyro_Model to provide gyro specific functionality, and it instantiates the Generic Sensor package to provide more general sensor capabilities. One of the interesting consequences of the extensive use of generics is that the system size decreased; the previous generation of Ada telemetry simulator contained 92 KSLOC, but this multimission simulator contains only 69 KSLOC.

This architecture was the first simulator designed to facilitate reuse from mission to mission. Unlike the MTASS system, this simulator does not need a mission-specific subsystem to handle telemetry; the telemetry formats can be set by run-time parameters. When this strategy is used appropriately, the reuse levels approach 90 percent verbatim code reuse, with the remaining part undergoing minor modifications.

While this 90-percent reuse level has helped reduce software costs and shorten development schedules, it has only done so on a limited class of systems. When the telemetry simulator was reused for a new class of systems (spin-stabilized spacecraft), the system complexity increased, reuse decreased, and run-time performance suffered. MTASS had a similar problem when it was applied to a spacecraft that did not have a sensor on which the original MTASS design depended.

In addition to variations between spacecraft, simulators and ground systems contain many common models. However, the current practice is to create separate systems from separate specifications. The way to account for variations between satellites and to exploit commonality between software systems is to perform domain analysis, rather than attempting to generalize the specification of a single satellite's simulator and ground-support system.

In the FDD, this domain analysis is being done as part of a generalized system development initiative. The attempt to develop generalized software to support multiple flight dynamics applications was based on the experiences of the projects described above. The multimission simulators demonstrated the feasibility of generic architectures, and it had been demonstrated that applying the object-oriented concepts of abstraction and encapsulation was sufficient to increase reuse dramatically. Finally, the existing designs were highly reusable, but had severe limitations in the areas of adaptability and run-time efficiency.

The key concepts selected for generalized system development in the FDD are to perform object-oriented domain analysis, and to have a standard implementation approach for the generalized models. Figure 5 shows a typical diagram from the generalized specifications.

The boxes are generalized superclasses with their subclasses listed inside; Gyro, Sun Sensor, and Star

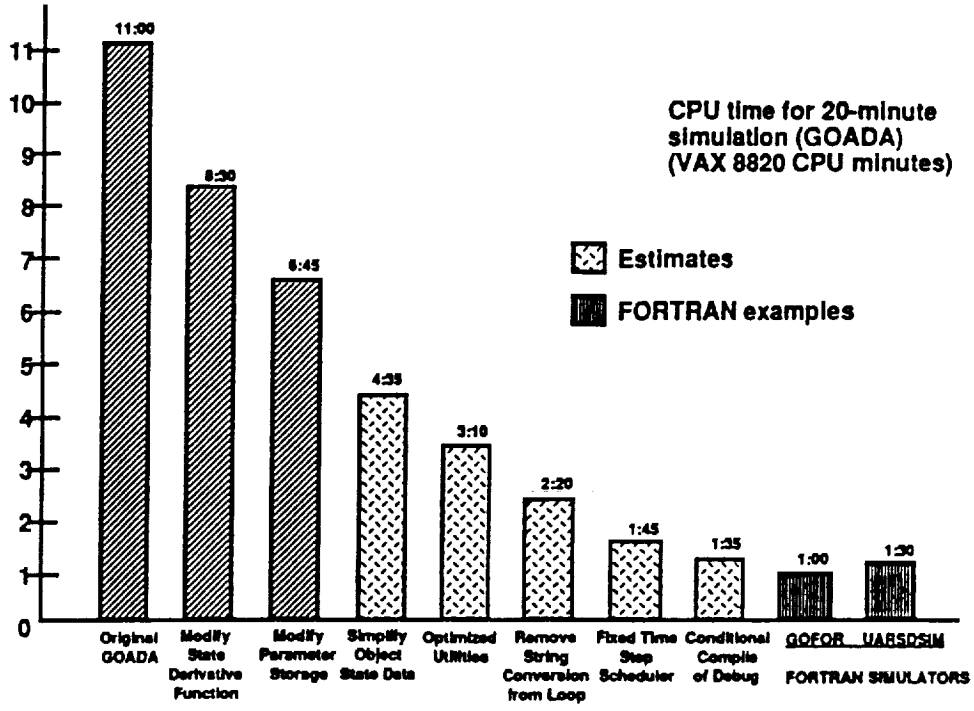

Figure 3. Impact of Performance Goals

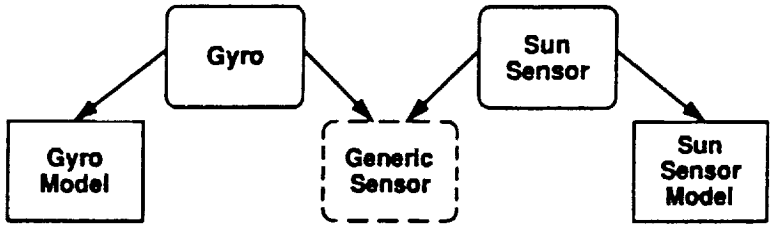

Figure 4. Multimission Telemetry Simulator Design

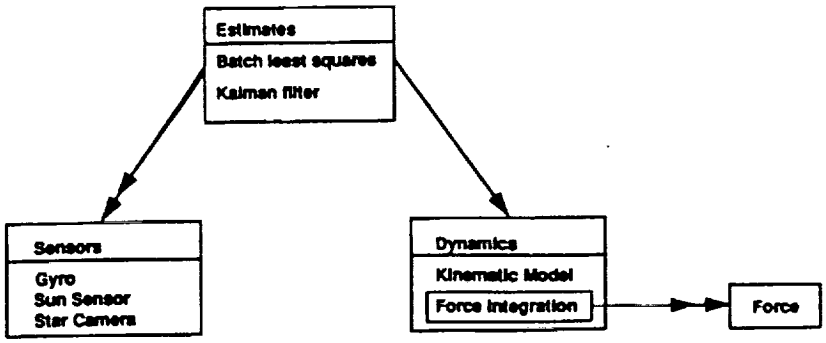

Figure 5. Generalized System Specifications

Camera, for example, are subclasses of Sensor. The arrows between categories represent dependencies between classes. For example, estimators depend on Sensor for measurements and Dynamics for state propagation. These dependencies are matched in the implementation with Ada generic formal parameters. The classes themselves are implemented as abstract data types in Ada packages. Each class shown on the diagram has a corresponding text specification that defines the member functions, user parameters, state data, and dependencies on other classes and categories. Categories also have text specifications for an abstract interface containing the functions common to all classes in the category. With this generalized development effort, object-oriented domain analysis and standard implementation, as well as other features of the object-oriented paradigm, are now being applied to the entire software life cycle.

With the successive generations of object-oriented development efforts defined, the next step is to examine how the SEL's approach has changed between 1985 and 1992. The approach has evolved in what concepts are used, when they are used in the life cycle, and how they are taught.

The concepts of data abstraction and encapsulation, used from the beginning, have themselves enabled the high reuse observed on the MTASS system; even the second Ada simulator attained higher reuse than is typical for similar FORTRAN simulators. The multimission telemetry simulator introduced the idea of inheritance by taking a general model for sensors and tailoring this model for each type of sensor. It also introduced the idea of parameterizing dependencies with Ada generic formal parameters. The generalized application work added the use of abstract data types, where previous systems had implemented objects as state machines. The generalized systems also have a superclass/subclass hierarchy limited to superclasses (called "Categories") and one level of subclasses for each superclass. Dynamic binding is coded using Ada case statements, not an object-oriented programming language feature.

Having support for object-oriented programming in Ada would remove the need to write this code, which would reduce development costs. However, the simple data abstractions provided by Ada packages have already increased reuse levels from approximately 40 to approximately 90 per cent of the delivered code, so the remaining potential cost reductions are dominated by those already attained. Dynamic binding would reduce the tedium of implementing case statements to handle run-time dispatching, but it is not the most important characteristic of object-oriented programming languages from a project cost point of view.

In addition to the increased reuse, the evolution to object-oriented development affected the reliability and changeability of the system. Table 1 shows the effort needed to determine what change is necessary to correct an error or to otherwise enhance a system.

Table 1. Changes Needed to Correct Errors or Enhance System

	Effort to Isolate Changes				
Project	< 1 hr	1 hr-1 day	1 day-3 day	> 3 day	Total
GOESIM	116	102	27	7	262
UARSTELS	205	77	10	5	297
SAMPEXTS	8	7	0	0	15

These data are shown for three telemetry simulators. GOESIM is an early Ada project whose design is similar to previous FORTRAN projects. UARSTELS is the first simulator in the multimission telemetry simulator family, and SAMPEXTS is a simulator that reuses from UARSTELS. The second-generation systems have a far greater proportion of changes that take less than one hour to isolate. These results support the claim that object-oriented designs produce systems that are more easily modified because of the information hiding provided by objects and classes.

The types of errors that occur also changed over time. Table 2 shows the classification of errors for the same three systems described above.

These data show that the development of UARSTELS, the initial second-generation system, was slightly more error prone than other projects. While overall errors were increasing, though, errors relating to interfaces and data structures were substantially reduced. Again, this is consistent with the perceived benefits of abstraction and information hiding. Even more striking is the complete elimination of interface errors for high-reuse projects such as SAMPEXTS.

The other notable change is in how OOT affected the development process. In the MTASS system, it had minimal impact, as the design approach was structured, with the object orientation being recognized during coding. Both generations of simulators used object-oriented design and object-based coding based on Ada packages; the generalized system project added an object-oriented approach to defining specifications. It is anticipated that having an object-oriented view throughout the life cycle will make the use of the technology easier by removing the need to recast functional specifications into an object-oriented design.

While object-oriented analysis has not been used for most systems, the high-reuse architectures have been influenced by how the specifications are written. Typical specifications have focused on a single satellite mission, and they specify the simulation and ground-support software separately. The building of the high-reuse MTASS and telemetry simulator systems was possible because the flight dynamics analysts wrote a single specification for the UARS and EUVE missions; the simulator and ground-support systems were still specified separately. The limitations of these specifications is one factor that led to a domain-analysis approach, so that a wider range of satellites can be supported and commonality between ground support and simulation can be exploited. The domain-analysis team switched from a structured to an object-oriented approach as they attempted to write a generalized specification.

Because the generalized system development is still in design, the impact of object-oriented analysis cannot yet be measured. But the use of object-oriented design has changed the development process by shifting work to the design phase. This is due to the high reuse allowing the production of an initial build by integrating existing components. SAMPEXTS thus demonstrated a system that met a large proportion of the requirements at the Critical Design Review. Table 3 shows the distribution of developer effort over the main phases of a development project.

Table 2. Classification of Errors

	Error Class						
	Data Startup	Computational	Logic	External Interface	Internal Interface	Initialization	Total
GOESIM	52	21	10	10	13	21	127
UARSTELS	25	40	43	9	3	39	153
SAMPEXTS	0	4	3	0	0	3	10

Table 3. Developer Effort Over Main Development Phases

	Effort Distribution by Phase		
Project	Design	Code	Test
GOESIM	29%	44%	27%
UARSTELS	25%	39%	36%
SAMPEXTS	48%	18%	34%

The SEL provided training in Ada and design techniques for the early Ada simulator experiments, but not for the later multimission simulators. The MTASS FORTRAN system involved no training in OOT, as the project did not set out to use a new language or design technology. The subjective experience of the SEL has been that the application of OOT was not so intuitive as expected, as functional decomposition has been successfully applied for more than 15 years. The SEL, recognizing that transition to a new technology must factor in the time required to learn the new way of thinking, is creating a new training program that captures the lessons learned on previous projects and describes the overall object-oriented software development process as well as specific language and design concepts.

The goal of bringing new technology into the SEL is to measurably improve the software development process. Figure 6 shows the project characteristics of the three multimission simulator projects.

The UARSTELS project was developed to be reused for future simulators, and the projects labeled EUVETELS and SAMPEXTS represent the first two projects to reuse this architecture. Costs were reduced by a factor of 3, change and error rates were reduced by a factor of 10, and project cycle time was cut roughly in half. However, we have already shown that when an attempt was made to reuse this architecture for a different class of projects there were difficulties adapting the code, and run-time performance was unsatisfactory.

The generalized system effort is attempting to gain the benefits shown for this single family of projects over a wider variety of flight dynamics applications. This will allow the FDD to support more missions simultaneously, and it will free resources to concentrate on improving existing capabilities or defining new ones.

SOME CONCLUSIONS

This paper addresses the question, "Is Object-Oriented Technology, then, truly the most influential method studied by the SEL to date?" The conclusion of the SEL is that OOT does promote reuse, sometimes even neglecting other important issues like run-time efficiency. When coupled with domain analysis, OOT enables high reuse across a range of applications in a given environment. While the reuse expectations were met, the use of OOT was not so intuitive as expected, partly because the technique was new to an organization with a mature structured development process. The other factor affecting the ease of transition is the inherent and growing complexity of flight-dynamics problems; OOT may be a better process but, in addition to software techniques, skilled designers are still needed to solve difficult problems.

Still, few (if any) of the other technologies studied here have effects so widespread or so profound as OOT. In fact, OOT is the first technology that covers the entire development life cycle in the FDD. It is an entirely new problem-solving paradigm, not simply a new way of performing familiar tasks in a traditional life cycle. It has been demonstrated to

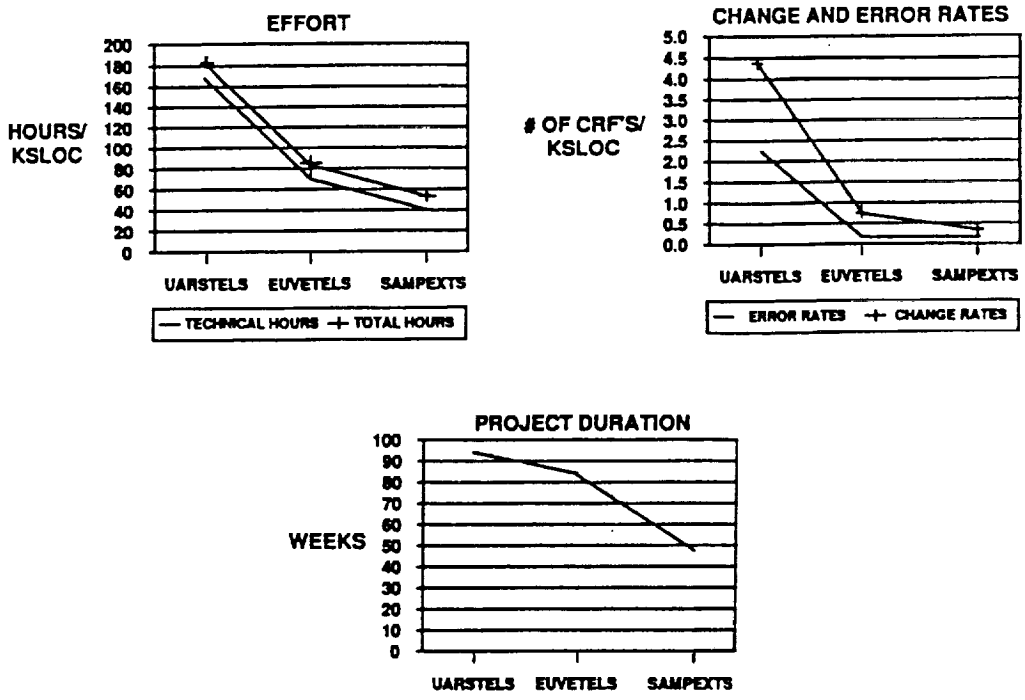

Figure 6. Project Characteristics, Multimission Simulators

expand the reusability and reconfigurability of software, with resultant improvements in productivity and development cycle time. In this sense, OOT is arguably the most influential technology studied by the SEL.

REFERENCES

1. McGarry, Frank E., and Waligora, Sharon, "Recent Experiments in the SEL," *Proceedings of the Sixteenth Annual Software Engineering Workshop*, Greenbelt, MD, December 1991, pp. 77-85.

2. Basili, Victor R., and Katz, Elizabeth E., "Software Development in Ada," *Proceedings of the Ninth Annual Software Engineering Workshop*, Greenbelt, MD, November 1984, pp. 65-85.

3. Booch, Grady, *Software Engineering With Ada* (First Edition), Benjamin/Cummings, Menlo Park, CA, 1983.

4. Seidewitz, E., and Stark, M., *General Object-Oriented Software Development*, SEL-86-002, August 1986.