"Improving Software Design & Development Education through Technological Innovation"
A Summary Report

National Science Foundation Grant CDA-9214892

Rensselaer Polytechnic Institute
October 1996

Principal Investigators:
Edwin H. Rogers, Ephraim P. Glinert, Robert P. Ingalls, David R. Musser
--- Department of Computer Science
Cheryl A. Geisler
--- Department of Language, Literature & Communication

Grant Period:
1 September 1992 - 31 August 1995

This report summarizes a three year grant four years after its start date. It covers the original motivation and objectives and where in those years the opportunities created by the funding have led us.

Motivation & Goals:

In the proposal we wrote,

"This proposal seeks support for the development, the integration into courses and the evaluation of novel tools and techniques addressing pressing issues in education in software design and implementation.
. . .
"This proposal emphasizes design before programming and higher-level programming through the reuse of generic and special purpose software components. Its three themes are better design through better use of language during conceptual stages of design; better design through object modeling and dynamics; and better software products through use of libraries of reliable, reusable software."

A Convergence of Concerns.

This work has been predicated on three convictions:

1... Modern design is multi-disciplinary and collaborative.
2... The software crisis demands new design paradigms.
3...The best software should be largely composed of recyclable code.

This led us toward three related initiatives:

a... Computer-based support for collaborative design with an emphasis on verbal communication.
b... Object-oriented modeling to bridge between conceptual design and implementation.
c... Access to modern libraries of software components.

Initiatives & Accomplishments:

Under this grant we have enhanced educational opportunities through new programs and facilities in each area and continue to evolve them for local and national use.

I. Collaborative Design.

Getting concepts and requirements correct early on is crucial in any large project. This generally requires in depth consultation with clients and among the various stakeholders and specialists whose expertise must inform and guide the work. Mutual understanding and trust are key. The best communication tools for bridging the viewpoints of diverse people rely heavily on natural language, graphics and personal communication.

Driven by these considerations, we undertook our own multi-disciplinary design of facilities for collaborative design. We realized early that the need and the core issues are not peculiar to software design, and so we were able to engage a broad range of parties in the effort.

The most visible product of an extended, user-oriented design effort is the Rensselaer Design Conference Room or "DCR" (http://dcr.rpi.edu/). This is a 650 square foot conference room with a 90 square foot observation and development annex.

The DCR is a laboratory for exploration of modes of collaboration in design teams and of modes of support of design team conferencing. It has at its center a large conference table with spaces for six conferees at integrated design stations .

Whiteboards, tackboards, flipcharts, overhead projection, tables etc. offer traditional media for presentation and discussion of ideas. National networks and an electronic whiteboard enrich access to information and the expression of ideas.

The conference setting places no visual or acoustic barriers between participants. All electronic and media tools are discreetly situated to serve rather than dominate design team efforts.

Each design station consists of a computer workstation with keyboard and screen in a side desk to the left or right of each participant. Buried in the hexagonal conference table are three large shared monitors driven by a video/file server located in the annex. The conferees may join a collaboration session on a named project from their individual design stations. They may in turn take control of the server and the visual space on the shared monitors, moving and transforming information for all to view. These and auxiliary shared monitors placed elsewhere in the DCR show the same image.

The Collaboration Net software developed for the DCR enables participants to use whatever professional software tools they feel appropriate as they convey their concerns and ideas to other conferees. We think of the CN as meta-groupware.

Acoustic and video recording of sessions have enabled us do conduct detailed studies of team processes and performance. They also enable a team to keep a detailed record of proceedings and to backtrack to critical points in the discussion.

The DCR has been used by numerous teams from numerous disciplines and several cross-discipline teams. Documentary studies of team performance have been conducted with teams from courses in software engineering and helicopter design. Other users have been from courses in algorithm design, electronic arts, architectural CAD, and web site design. In addition, a Ph.D. thesis has addressed issues related to protocols for sharing the public screen in an empirical study conducted with teams of students.

The Collaboration Net concept is now being extended to distance collaboration with mixed platform capability. It will also be applied in a 36-42 seat collaborative classroom in which classes organized into six or more teams will meet. This facility will support six major kinds of interactive learning activities: instructor demonstration, peer learning, team meeting, instructor consultation, client consultation, and presentation & critique. (Architect sketches are appended.)

II. Object-Oriented Modeling.

Since 1989 object-oriented modeling has been a key part of the Rensselaer course Software Design & Documentation. It has been taught not as a programming tool but first as a conceptual tool useful in formalizing requirements domains and second as a means of architecting software systems.

In 1993 PIs Ingalls, Musser and Rogers developed a course for delivery via satellite video, fax and the Internet to industrial sites in five states based on these ideas and the Object Modeling Technique (OMT) of colleagues at General Electric and described in the text by Rumbaugh, etal. [13]. The same materials were used again the following year in a single site, distance course via PictureTel.

OMT has been used as the basis for OO education in the Software Design & Documentation course since 1991 as well. The OMT CASE tool has been used with great effect. Part of the assessment in the case of the single site distance course revealed a marked improvement in understanding between software engineers and upper management due to OO modeling.

As OO programming is being introduced into freshman and sophomore courses here and elsewhere, we expect to find that students will need to practice OO thinking outside the programming domain in order to become really effective OO programmers.

III. Standard Template Library: EnDoc & BALM.

PI David Musser has been a principal in the creation and application of the Standard Template Library (STL). In July 1994 the ANSI/ISO C++ Standards Committee voted to adopt STL as part of the standard C++ library.

The Standard Template Library is a new C++ library that provides a set of easily composable C++ container classes and generic algorithms (template functions). The container classes include vectors, lists, deques, sets, multisets, maps, multimaps, stacks, queues and priority queues. The generic algorithms include a broad range of fundamental algorithms for the most common kinds of data manipulations, such as searching, sorting, merging, copying, and transforming.

The most important difference between STL and all other C++ container class libraries is that most STL algorithms are generic: they work on a variety of containers and even on ordinary C++ arrays. This makes these template functions highly reusable so that the considerable effort given to making them correct, reliable and efficient is easily justified.

A key factor in the library design is the consistent use of iterators, which generalize C++ pointers, as intermediaries between algorithms and containers. A precise classification of iterators into five categories is the basis for determining which algorithms can be used with which containers, and is the main guide to extension of the library to include new algorithms that work with STL containers, or new containers to which many STL generic algorithms can be applied. Extensibility of the library as new and better algorithms are discovered is an important library attribute.

Among the applications of STL has been its use in undergraduate computer science courses starting with the freshman year. Students are given early exposure to the benefits of highly reusable software and to thinking in terms of generic algorithms. To support this activity, Dr. Musser has authored a text [12] and conceived the EnDoc project [1] to provide hypermedia documentation for software libraries. Working with graduate research assistants, he has applied his ideas to STL particularly and made the result available world-wide at http://www.cs.rpi.edu/~musser/stl.html. Appendix A gives a fuller description of the EnDoc project.

Experience in teaching C++ programming with STL has been accumulated for two years. Like others, we have encountered formidable obstacles to teaching C++ to beginning students, arising from the complexity of the language and the shortcomings of its support tools. Particularly perplexing are the cryptic error messages produced by C++ compilers. This has led us to initiate the BALM[1] project aimed at providing helpful explanations of compiler diagnostics with hints as to where to look for the real problem, and specifically to provide better diagnostics for users of STL. Appendix B gives fuller description of the BALM project.

Dissemination Activities:

Web Sites:

[1] BALM is rumored to stand for Banish All Lousy Messages.

Design Conference Room: http://dcr.rpi.edu/
Standard Template Library: http://www.cs.rpi.edu/~musser/stl.html

Publications:
[1, 2, 3, 4, 5, 6, 7, 9, 10, 12]

Evaluation Activities:

The collaborative design initiative has incorporated a series of evaluation activities. During the design of the DCR, a focus group workshop was held involving a cross-section of the university community and selected guests from industry to explore design alternatives [4].

A careful formative study of student teams in the course Software Design & Documentation were conducted in the academic year 1993-94 and are reported in [7]. The DCR and its Collaboration Net software were put to use by this course in the Spring 1995 term with two student teams, one working with traditional tools and one working with the DCR, carefully followed in an effort to identify dominant effects of the DCR (and to help debug DCR shortcomings). This provided valuable background for the lab study of electronic medium sharing protocols reported in [3].

Detailed monitoring of an aeronautical engineering helicopter re-design team was conducted in the last academic year with formative and summative evaluation of the data gathered now in progress.

The Standard Template Library s use in undergraduate programming courses has been put to the test in Rensselaer s Computer Science 2 course. Assessment has been primarily anecdotal but certain lessons are clear. The WWW documentation EnDoc has been especially valuable (as also evidenced by hit rates), and the need for better diagnostic tools is crystal clear (hence the BALM Project).

Benefits Seen & Expected:

The DCR has stimulated great interest among all who have seen and used it. For those groups who have made the effort to commit to its use, team focus and product have been notably advanced. Among educators the DCR has stimulated or reinforced a recognition that collaborative computing is an important emerging paradigm. Computer support of intellectual teamwork can take a variety of forms, and the DCR is one of these. A key benefit for education will be the resultant creation of new collaborative educational facilities, such as the Collaborative Classroom Rensselaer is creating on the DCR/Collaboration Net model. We intend to work with other colleges exploring or adopting such innovations.

The use of object oriented modeling in domain understanding and system definition has improved mutual understanding among clients, software developers and management. The ability to teach this effectively has made our students much more interesting to industrial recruiters. Moreover, from distance education programs broadcast to industrial sites, we have received particularly encouraging feedback from both in-house software groups and

their upper management on mutual understanding and productivity due to this methodology.

We expect that significant productivity gains will be realized in professional software engineering from the use of STL. These will be due to supporting facilities (like EnDoc and BALM) and the educational programs which teach the use of containers, iterators, generic algorithms, function objects and the like.

Participants:

The following students have been engaged in the work of this grant. Those with asterisks have received financial support either as research assistants or hourly employees. Degrees earned are shown; parentheses indicate degrees in progress.

Frank Adessa, (B.S., Computer Science)
Joy Chik, (B.S., Computer Science)
Robert Cook, (Ph.D., Computer Science)*
Jeff Corthell, (B.S., Computer Science)
Anne Ferraro, Ph.D., Computer Science*
Feleicia Gondamulia, B.Arch., Architecture*
Cynthia Haller, Ph.D., Rhetoric*
Darren Hayduk, B.S., Computer Science*
Lee Honeycutt, (Ph.D., Rhetoric)*
Thomas Larson, (B.S., Computer Science)
Jean Etienne LaVallee, (B.S., Computer Engineering)*
Chun-Da Lee, B.S., Computer Engineering*
Sean Lensborn, (B.S., Computer Science)*
Bernadette Longo, Ph.D., Technical Communication*
George Matey, B.S., Computer Science*
Susan Mings, (Ph.D., Technical Communication)*
Christopher Parker, (B.S., Computer Science)*
Ben Rockwell, M.S., Computer Science*
Kevin Saxton, B.Arch., Architecture*
Kristen Schmitt, B.Arch., Architecture
Jason Swartz, B.S., Computer Science*
Geoffrey Wenger, M.S., Computer Science*
Kenneth Zalewski, M.S., Computer Science*

Hundreds of other students have taken part in formal assessment of our work or have been in classes which have used the innovations stimulated by this grant.

The following faculty have taught using innovations introduced with the help of this grant.

Chris Boese, Instructor, LL&C
Cheryl Geisler, Associate Professor & Chair, LL&C (geislc@rpi.edu)
Ephraim Glinert, Associate Professor, CS (glinert@cs.rpi.edu)
Mark Goldberg, Professor, CS
Robert Ingalls, Executive Officer & Lecturer, CS

Maureen Kinsella, Assistant Professor, Arts
M.S.Krishnamoorthy, Associate Professor, CS
Andrew Lemnios, Professor, Aeronautical Engineering
Brian Lonsway, Adjunct Assistant Professor, Architecture
David Musser, Professor, CS (musser@cs.rpi.edu)
Edwin Rogers, Professor, CS (rogerseh@cs.rpi.edu)
Michael Skolnick, Associate Professor, CS
David Spooner, Professor, CS
Charles Stewart, Associate Professor, CS
John Tobin, Assistant Professor, Architecture (tobinj@rpi.edu)

Appendix A:

EnDoc---hypermedia documentation for software libraries

One of the goals of the project ``Improving Software Design and Development Education Through Technical Innovation'' has been to develop and document software libraries suitable for use by undergraduates in Computer Science courses, supporting the development of a curriculum in which software reuse is taught and consistently practiced, instead of the predominant ``program everything from scratch'' approach of today's curricula and textbooks. By permitting and encouraging students to make as much use as possible of good software libraries, we believe we will enable them to concentrate more effort on, and develop more insight into, other areas of software development besides programming. They will be able to spend more time on system specification, design, and testing, and they will be able to accomplish more interesting and satisfying software development projects than in traditional approaches in which they must spend the majority of their time implementing basic algorithms and data structures. Emphasizing frequent and intelligent use of software library routines is also naturally supportive of software reuse approaches that are gaining currency in many industry projects.

It is not enough just to tell students that certain software libraries are available for their use. They must be made aware of the functions available in the libraries, and they must learn how to read and understand the documentation well enough to find a function that meets a particular need. With an extensive library that provides alternative functions for the same task, they ideally should learn how to select one of several available functions that is best in the context in which it is to be placed (e.g., the best of five or six sorting algorithms for the type of data they expect to have to sort). Unless sufficient attention is given to all of these issues, many students will fail to see or to achieve the advantages of using a library routine over coding the needed function from scratch. All of which leads us to the main motivation for the EnDoc project [1]: the idea that much better and more accessible documentation for software libraries is needed than what is typically provided, either in printed library reference manuals or on-line.

In EnDoc, we use hypermedia as a key technology for on-line documentation of the software libraries. For assisting in selection and use of software components from a library, hypermedia provides for presentation of information at various levels of depth, suited to different stages of the student's learning, and with much greater use of visual and dynamic information than is possible with hardcopy documents. In the EnDoc project, hypermedia documentation of the C++ Standard Template Library (STL), a major part of ANSI/ISO Draft Standard for C++, was prepared according to design principles described in [1]. STL provides a set of easily composable C++ container classes and generic algorithms (template functions). The container classes include vectors, lists, deques, sets, multisets, maps, multimaps, stacks, queues and priority queues. The template algorithms include a broad range of fundamental algorithms for the most common kinds of data manipulations, such as searching, sorting, merging, copying, and transforming. The critical difference between STL and all other C++ libraries is that most STL algorithms are generic: they work on a variety of containers (the same source code is adapted by the compiler to the form of each container).

The EnDoc hypermedia documentation describes all of the STL generic algorithms (more than 100). Each datasheet for an algorithm or group of related algorithms contains an example showing simple ways to use the algorithm(s), and each such example can be executed by the user. The EnDoc STL documentation has been accessible since October, 1994, on the World Wide Web via using the URL

```
http://www.cs.rpi.edu/~musser/stl.html
```

Electronic mail regarding the site and access logs indicate extensive use. (Measurements taken in April 1995 showed an average of over 200 accesses per day.) A search of the web (using the AltaVista web page index) revealed about 400 other pages with links to the EnDoc STL documentation, as of October 25, 1996.

At Rensselaer, the EnDoc STL documentation has been used extensively in several sections of three undergraduate courses. Two of these courses (Computer Science II and Data Structures and Algorithms) have usually been conducted in a studio class room (with up to 60 students using 30 workstations), for which we developed more than two dozen worksheets that both introduce fundamental computer science material and reinforce them with on-line programming exercises and solutions. The emphasis on C++ and building programs out of software library
components makes this material highly relevant to the kinds of real software development and maintenance problems that students will encounter later on, especially in industry projects.

--- D.R.Musser

Appendix B:

BALM---relief from cryptic compiler error messages

The C++ programming language has emerged as the predominate language used in undergraduate computer science education. The language has many advantages over either of the languages formerly taught to most beginning undergraduates (Pascal or C), based on modern features that aid in constructing large, complex programs, including support for object-oriented programming and generic programming. Successful teaching of C++ to beginning undergraduates (or high-school students) requires careful initial selection of a subset of its features and gradual introduction of additional features only after mastery of the subset. Thus most textbooks do not attempt to cover the real issues of object-oriented programming in the beginning, leaving them to later chapters or to texts for more advanced courses. In theory, this should work well because even without the object-oriented features the language C++ provides better support than Pascal or C for most of the simple kinds of programming exercises suitable for beginning students. This is particularly so if use of standard library routines is emphasized, since the C++ Standard Library is much more extensive than libraries for Pascal or C. One can teach how to use linked lists in simple programs, for example, without having to teach the complexities involved in writing a linked list class definition.

In practice, however, there are formidable obstacles to teaching C++ to beginning students, arising from the complexity of the language and the shortcomings of its support tools. Particularly perplexing are the cryptic error messages produced by C++ compilers. When given a program with syntactic or compile-time type errors, compilers typically give a diagnostic message based only on local information at the site of the infraction. The novice programmer is left with no clue as to the real source of the problem, which may be a missing or mistyped declaration many lines away or in a different source file entirely. The diagnostic is usually stated in the tersest possible form, and no hint is offered as to where to start looking for the real problem.

The compiler diagnostic problem is only exacerbated by the use of standard libraries, particularly libraries like STL that make extensive use of the template features of the language. If used exactly according to their specifications, the STL template classes and functions can provide many of the capabilities needed in an application program. As already mentioned, student programming exercises or projects that use STL can be much more substantial and interesting than if every data structure or algorithm has to be programmed from scratch. But if a student invokes a C++ template class or function definition incorrectly, even slightly, the result often is a long string of completely baffling compiler diagnostics, which appear to pinpoint the problem *in the library source code*, not in the student's own code. Even worse, the library code is likely to use language features and programming techniques that are totally unfamiliar to the novice programmer.

To address these problems, a small project called BALM was initiated with the goals of (1) generally providing helpful explanations of compiler diagnostics, including hints as to where to look for the real problem, and (2) specifically attempting to provide better diagnosis of errors in the use of STL.

For the general facility for explaining compiler diagnostics, it was decided to focus efforts on the compiler used most extensively at Rensselaer (but one that is also used extensively at many educational institutions), the GNU C++ compiler. The diagnostic messages produced by this compiler are fairly typical of those of many compilers. (Generally they are somewhat terser, and there is no additional explanation available, as there is with some commercial compilers such as Borland's or Microsoft's.) Although there is no fully-realized GNU C++ software development environment, a modest set of tools is available including several debuggers and the GNU Emacs editor with C++ editing and compilation modes. Emacs includes the ability to maintain two windows, one with a cursor on a compilation diagnostic and the other with a cursor in the source code file on the line to which the diagnostic refers. The BALM explanation facility is an extension of this Emacs error facility, providing a third window containing, for the current diagnostic, an explanation and suggestions about where the real source of the problem might lie. So far only a small number of such explanations (relative to the large number of different diagnostics the compiler can emit) have been written; but the facility includes a mechanism for capturing unexplained diagnostics and source file (with the student's permission) and automatically mailing it to a BALM project member who can then attempt to reproduce the problem and write an explanation for it. The BALM facility is being put into initial use in the Computer Science II course currently being taught at Rensselaer.

A large part of the blame for the poor compiler diagnostics for uses of template library components can be placed on the C++ language itself. There is no language-supported means of specifying type-requirements on the type parameters that appear in templates. For example the requirement that a type parameter of a template class support a + operation cannot be stated as part of the class interface; the check is done only at such time as the template class is specialized by substituting an actual type for the type parameter. If a program specializes the template class with an actual type that is missing a + operation, the problem is detected only when the compiler is scanning the code in the class definition where + is used, and the compiler issues the diagnostic in terms of that code. This can be totally baffling to the beginning student (or even to the expert at times), in addition to undermining his or her confidence in the quality of the library components themselves.

If C++ did provide a way to specify type requirements in interfaces (as does Ada 95, for example), failures to meet those requirements could be much more clearly diagnosed by the compiler. The checking would be done against the interface specification, and the diagnostic would be given citing the header of the incorrectly invoked template, rather than some point that is possibly deep within the template's definition. Unfortunately, although it has been proposed that some means of expressing type requirements in interfaces be added to C++, no such facility has been added in the (now nearly completed) C+ Draft Standard.

The STL-specific part of the BALM project attempts to alleviate this problem for STL templates, using a limited form of type-requirement specification that is (crudely) supported by the language. The specification technique used is adapted from an idea discussed by Stroustrup [14, p. 345]. The basic idea is simple: to express constraints on a template type parameter at the site of the class or function header, one simply introduces right after the header dummy uses of all operations that are required anywhere in the class or function definition. There are, however, several problems in using this technique. First, although the dummy operation uses will force the compiler to give diagnostics about misuse at the beginning of the template definition, other diagnostics will still be given for all of the uses deep within the definition, still causing confusion. Second, in template functions it can be very difficult to write the dummy uses without interfering with the correctness of the actual body of the function. (For template classes, Stroustrup shows how the dummy uses can be placed in a member function so that they do not interfere with the actual class implementation, but C++ does not allow nested function definitions.) Finally, for templates that have many constraints, a misuse might lead to diagnostics for most or all of the constraint violations, when in many cases only a single diagnostic would be adequate to pinpoint the problem.

To address these shortcomings we use a modified form of Stroustrup's technique to specify type parameter requirements of STL components. First, we do not attempt to combine the dummy-use technique with the actual implementation; instead we employ it in a *separate* set of header files that are to be used when initially compiling. These header files contain the same components as the actual headers, but with dummy-use constraint checks as their bodies rather than the actual code bodies they have in the normal header files. Second, in most cases the dummy-use constraint checks do not include dummy uses to check all constraints; instead, they are limited to ones that can detect the most common kinds of illegal specializations. That way, a misuse of a template class or function results in a minimal number of diagnostic messages (in most cases only one). Third, each constraint check is

expressed with the aid of a preprocessor macro such that the dummy use code itself is seen by the compiler but not by a human reader, at the site of the check, where it might cause confusion. Instead, the macro name indicates mnemonically the nature of the constraint being checked.

Thus, if one compiles with these STL constraint-checking headers, any compiler diagnostic related to use of a library component points to the line with the mnemonic name of the check, and that line comes soon after the component header. No diagnostics are given for lines in the component implementation, because the implementation is not even present in these headers. Thus in no case is the programmer confronted with diagnostics that appear to be indicating a bug in the component implementation.

After one corrects STL component-misuse problems in one's own code and gets it to compile with the STL constraint-checking headers, one can then proceed to recompile it with the normal STL headers. In most cases this compilation should immediately succeed (unless there is some STL component-misuse problem that was not detected by the limited constraint checks in the constraint-checking headers).

At present, the STL constraint-checking header files are still under development. When they are completed we plan also to revise and expand the general BALM diagnostic explanation facility to work together with the constraint-checking facility to give fuller explanations of, and hints for fixing, type-constraint violations.

--- D.R.Musser

Publications & Presentations:

[1] R.Cook,Jr., D.R.Musser, & K.Zalewski, Enhanced Documentation of Software Libraries Using Hypermedia, RPI Computer Science Department Technical Report 94-21, Troy, NY, October 1994.

[2] A.Ferraro, E.Rogers, & C.Geisler, Team Learning Through Computer Supported Collaborative Design, *Proceedings, Computer Supported Collaborative Learning*, 1995.

[3] A.Ferraro, *An Examination of the Collaborative Design Process Using Multiple Media Resources and Sharing Protocols*, Ph.D. Thesis in Computer Science. RPI, 1996.

[4] C.Geisler, E.Rogers, & J.Tobin, Designing for Collaborative Design Education, , *EDUCOM*, San Diego, CA, 1994.

[5] C.Geisler, L.Honeycutt, & E.H.Rogers, Professionalization & Multidisciplinarity: The Reemergence of Public Discourse, *American Educational Research Association,* New York, NY, 1996.

[6] C.Geisler, E.H.Rogers, & C.Haller, Disciplining Discourse: Discourse Socialization in the Affiliated Professions of Software Engineering Design, 1996.

[7] C.Haller, *Topics in Rhetorical Invention and Technological Design: The Hermeneutics of Software Engineering*, Ph.D. Thesis in Rhetoric, RPI, 1995.

[8] L.Honeycutt, etal, The Rensselaer Design Conference Room, Web site -- `http://dcr.rpi.edu/`, 1994.

[9] L.Honeycutt, *The Design Conference Room User Guide*, RPI, July 1996.

[10] S.Mings, C.Geisler, & E.Rogers, Hypertext Technology in Information Design, *Proceedings, IEEE Professional Communications Conference*, 1993.

[11] D.R.Musser, The Standard Template Library, Web site - `http://www.cs.rpi.edu/~musser/stl.html`, 1994.

[12] D.R.Musser & A. Saini, *STL Tutorial and Reference Guide*, Addison-Wesley, Reading, MA, 1996.

Other References:

[13] J.Rumbaugh, M.Blaha, W.Premerlani, F.Eddy, & W.Lorensen, *Object-Oriented Modeling and Design,* Prentice-Hall, Englewood Cliffs, NJ, 1991.

[14] B.Stroustrup, *The Design and Evolution of C++,* Addison-Wesley, MA, 1994.