Software Reuse Metrics for an Industrial Project

Richard N. Ferri
rnf@orbit.hr.att.com

Raghavan N. Pratiwadi
rnp@rosina.hr.att.com

Lynn M. Rivera
lma@rosina.hr.att.com

Mohammed Shakir
shakir@rosina.att.com

John J. Snyder
jjs@rosina.hr.att.com

D. W. Thomas
dwt@rosina.hr.att.com

AT&T Network and Computer Services
480 Red Hill Road
Middletown, NJ 07748-3098 USA

Yih-Farn Chen
chen@research.att.com

Glenn S. Fowler
gsf@research.att.com

Balachander Krishnamurthy
bala@research.att.com

Kiem-Phong Vo
kpv@research.att.com

AT&T Labs – Research
180 Park Avenue
Florham Park, NJ 07932-0971 USA

Abstract

In 1990 a project was established at AT&T to build applications that manage telephone systems. Since then the project has successfully completed over 20 applications comprising about 500,000 lines of source code. These systems are used daily by hundreds of managers and operators to monitor and provision the AT&T long distance telephone network. The project's success can be attributed directly to an early commitment in making software reuse a major component of its software development process. A critical factor was the establishment of a feedback loop between consumers and producers of reusable software to foster continual improvement and extension of reusable code repositories. Progresses in the feedback loop are measured by five different reuse measures. While no one measure is "best" as each provides a different perspective on reuse, two derived from the consumer/producer model have proven particularly useful: use of reusable library components and reuse growth factor. The latter, developed in this study and described below, helped uncover a new opportunity for reuse that was not obvious from other measures.

To Be Published in:
Proceedings of the
Fourth International Symposium on Software Metrics
IEEE Metrics '97
Albuquerque, New Mexico, USA
November 5 - 7, 1997

Introduction

In 1990 a group of software engineers at AT&T embarked on designing and building a set of interactive end-user systems to collect and display data generated by telecommunications switches for use by designers, engineers, planners, and managers. From the outset, the project committed to a framework for software reuse to increase development effectiveness by leveraging from the many common tasks in the application suite. The software reuse framework consisted of three major components:

- Employing a layered software architecture [15] with layers ranging from very high-level code with parameterized macros to fairly low-level C code interfacing directly with file and database systems,
- Forming a single organization containing both consumers and producers of reusable software components, and
- Fostering close communications between consumers and producers (in some cases, they are the same people) of reusable software components.

As a result of this reuse framework, the software development process contains an inherent feedback loop between consumers and producers of reusable software in which:

- Consumers advise producers of their needs on an ongoing basis, and
- Producers observe how applications use their software and improve the software therewith.

In this way, existing components are continually improved and extended to meet new requirements while new components may be added. The practice of copying and modifying code is minimized and confined only to where there are no existing components covering newly found requirements. Reuse measures were introduced to the project in 1995 to serve as a check on reuse levels and helped uncover new reuse opportunities.

At this time, the project consists of over 20 industrial applications comprising some 500,000 lines of code. These systems are used daily to monitor and provision our telephone network. Level of software reuse is high. The top 50 reusable components are called anywhere from 100 to over 10,000 times across applications and certain applications approach 60% of reusable code.

Five objective reuse measures [4] have been applied to the code to estimate current impact of reuse and stimulate new reuse. While each measure has strengths and weaknesses (see Table 7 in the Discussion section), two derived from the consumer/producer model [2] have been particularly useful: use of reusable library components (for producers) and reuse growth factor (for consumers). The latter, developed in this study and introduced in the Measurements section below, helped identify a new opportunity for reuse that was not obvious from other measures.

This study describes the environment created to support software reuse in developing a family of products and explores how software reuse metrics and measurements can be used to help encourage and support software reuse.

The Project

The daily management of a telephone network comprises many activities. For example, a major part is provisioning [13], which collects, monitors and equips switches, reviews trunk forecasts, and develops plans for the order, installation, or removal of equipment. Much of the raw data are generated automatically by the switches and processors in the telephone network. Portions of this data are periodically downloaded onto other systems and stored in relational databases for access and analysis. Over the years, many *ad hoc* software tools were built to support these activities. Around the late '80s at AT&T, it became clear that existing tools had become too cumbersome to use by average telephone operations staff.

In 1990, a project was started to build applications that tied together many of the existing tools and to build new tools more suitable for the management of a modern telephone network. For this task, the team chose EASEL [14, 6] a language and system to build end-user applications based on interactive constructs such as windows, forms, menus, and hypertexts. Data manipulation and analysis for applications and their users can be performed by EASEL code or can be handed out to existing computational tools written in other languages. This is facilitated by EASEL language constructs that map into two broad programming categories [15] shown in Figure 1.

Design Programming

Activities focused on the user interface and high-level tasks as seen from the user's point of view.

Computation Programming

Activities focused on how the high level tasks are to be implemented and with what data structures.

Figure 1. Interactive End-User Systems Architecture

Realizing that many similar application systems would be built, the project committed early to a framework of software reuse as outlined in the Introduction. The first major part in this framework was to divide a software application into different layers distinguished by different levels of programming.

Application Frames

Written in EASEL to accomplish tasks as seen by end users; typically rely heavily on parameterized macros.

Application-Specific C Functions

Functions for a specific application.

Parameterized Macro Library

High-level reusable software components; macro expansion generates EASEL code.

Common Frame Library

These fall into two areas: implementing tasks and subtasks on behalf of the user and providing computational capabilities such as string manipulation and common screen formats.

Common C Function Library

General-purpose functions, including calls to construct SQL queries for ORACLE databases.

Figure 2. The Project's Layered Architecture

Figure 2 summarizes the layers in a typical application. At the highest level, each application is modeled by a frame network [14] in which each frame captures a high level task. These tasks model groups of activities that a group of communications network engineers and managers would perform in a course of their daily work. A task may be

decomposed into components, such as selecting nodes in the network, components of a switch, traffic type, or reporting intervals. Many subtasks are common to more than one application and are implemented as macros parameterizable with application-specific values. The macros expand to calls to common frames and C functions which may dynamically create SQL queries[1] to access data stored in relational databases. Each of the bottom three layers in Figure 2 constitutes a library of reuable components.

In Figure 3, the application frame equip_menu shows how components from different architectural layers are used. This frame lets a user select a switch at a given location and review or update its hardware. The first task is to solicit and verify a location code from the user, (who may type it in or select it from a pop-up menu). This is done by calling the common macro Location, which, upon expansion, generates code for several subtasks: call the macro ReadFile with the name of the file to read for current locations, call the common frame MkList to format the data, write a block of code to conduct the user interaction, and call the common C function InList to verify the user's selection. The next task is to obtain the current hardware information from the database for the desired switch; this is done in two steps. First the application frame equip_sql is called with the desired location code to compose an SQL query; this frame hands off most of the work to the common C function FormSQL. Second, the query statement is passed to the common frame GoFetch, which calls the common C functions ExecQuery to execute the query and FetchTable to retrieve the query results; these two routines call matching ones in the project's database accessor to perform the actual database transactions. The query results are then displayed to the user by the MenuDisp macro, which manages the display as a scrollable and editable menu. A call to the frame equip_change collects any changes entered by the user; it calls equip_sql to create an SQL update statement, which is handed to the application-specific C function equip_update to validate the changes and update the database via the database accessor.

The Environment to Support Reuse

The project currently comprises over 20 applications that serve a wide variety of technicians, designers, engineers, planners, and managers watching over more than a dozen kinds of network elements, processors, and switches. Team members are assigned to build applications along with reusable components in different layers. Frequent negotiation and communication among team members help the initial interface design to quickly converge. During implementation, flaws and/or inadequacies in the components are continually discovered. However, the close association between users and providers of the reusable software minimizes any tendency to just copy and modify components for current needs. Instead, generic solutions are sought and codified in reusable code.

The creation of valuable reusable software involves much hard work. Our experience in this project parallels that recorded in [9]: the production of reusable software is much more effective when there is an active feedback loop that fosters close interaction between producers and consumers of such code. Except for time-proven components (*e.g.*, standard library functions), the lack of such a feedback loop often leads to the temptation to copy and modify (as occurred in [5]), where knowledge gained in a modification is quickly lost and convergence to final highly reusable software is not possible.

Our model of this active feedback is depicted in Figure 4. Project managers and architects help define the software architecture for the project. Producers of reusable components use specifications and input from component consumers to develop and test reusable components. Application developers, as the consumers of reusable components, work with user requirements and the reusable components to develop and test applications frames. Once the code for an application has stabilized, software reuse measurements can be made for the application. Producers and consumers, in cooperation with managers and architects, review and evaluate the reuse measures. This process, as shown near the end of the next section, can suggest new abstractions for implementation by producers in the reusable component libraries and for use by developers in successive versions of applications.

In this project, team members focused on an individual application use the reuse measurements presented in the next section to help understand, evaluate, and improve software reuse in their specific application. The reuse measurements can also be examined across a group of applications, looking for those with exceptionally good *and* poor measurements. By trying to understand the underlying causes of those cases, one endeavors to pass the benefits on to other applications – to improve those with poor values and to mimic those with excellent values,

When considering or practicing software reuse, two questions must be asked about the process:

- How effective is reuse at any given time?
- How can new reuse opportunities be identified and developed?

Various reuse measures were used to address these questions. Their use and our experience with them in the project are discussed next.

[1] The project's database accessor was built using Oracle's ProC. Accessor code was not included in this study's analyses; its functions are called from higher-level C libraries.

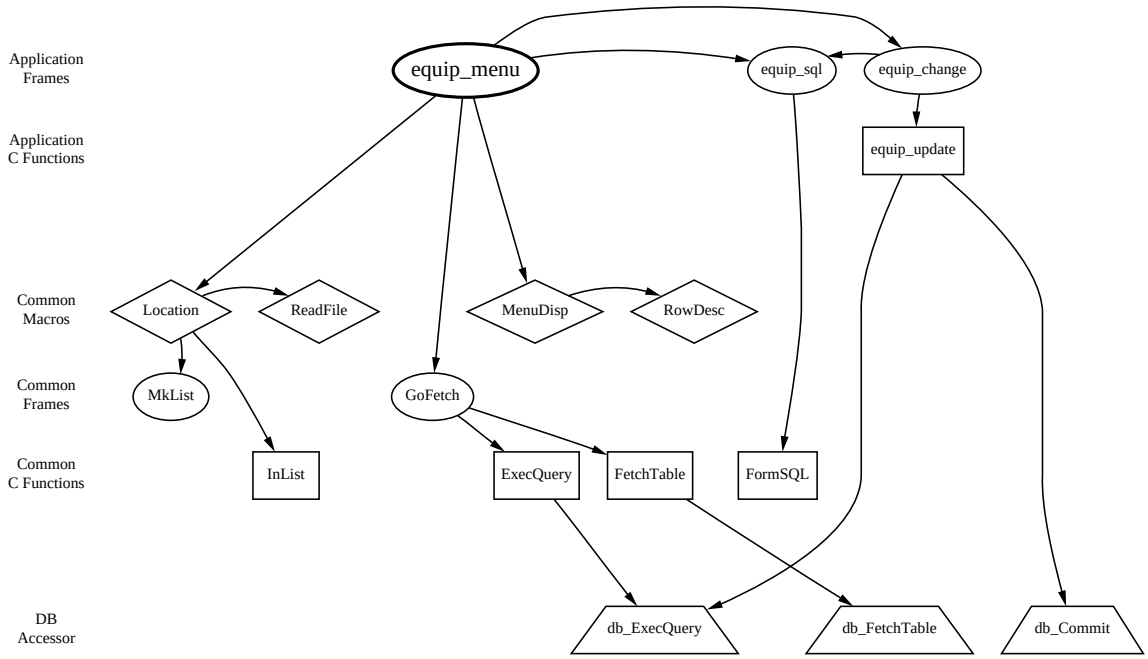

Figure 3. A Frame to View and Update Parts of a Switch

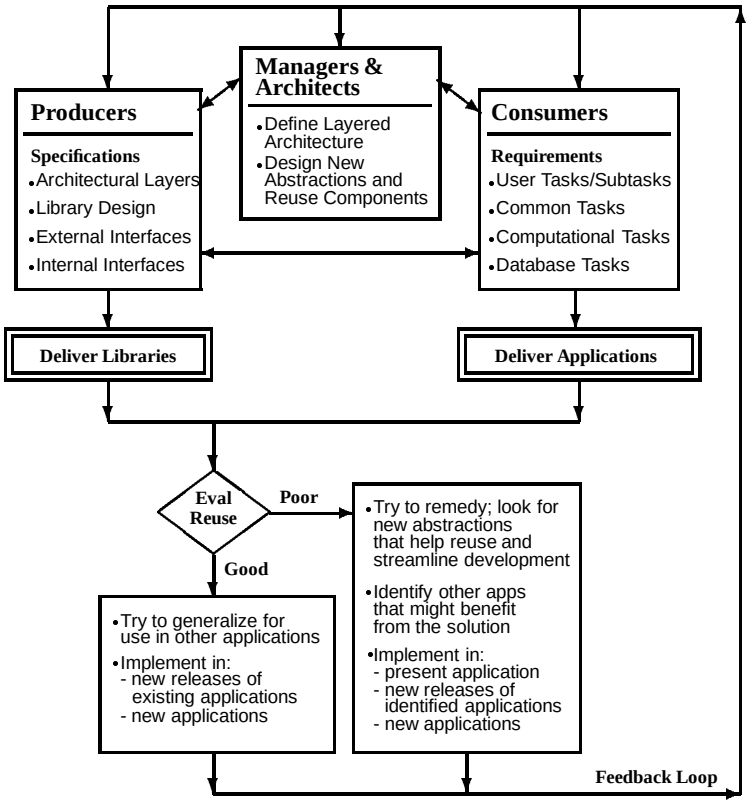

Figure 4. Feedback of Software Reuse Measurements into the Development Process

Objective Reuse Measures

A variety of software reuse models and measures have been proposed in the literature [7, 8, 10, 11]. Based on a consumer/producer model [2], Chen, Krisnamurthy and Vo [4] introduced the idea of code-based objective, complete, and precise reuse measures. Objective measures start with fixing some chosen code granularity such as shipped source [12], compilation units [1], files, functions, lines of code, etc. Then a complete and precise measure calculates exactly the amount of reusable code required for the functioning of the application being measured. Reference [4] described a model and methodology to compute such reuse measures. In particular, specific measures for the C language were proposed based on the C Information Abstraction System, CIA [3]. We follow their lead and the reuse measures used here build on their model. The perspective of their work was mainly on producers of reusable components. In this study, the additional and somewhat different needs of *consumers* of reusable components (the application developers) helped extend their model for two new measures (see below).

Since the project uses two different programming languages, EASEL and C, the common code granularities adopted include: (1) macros, frames, and functions, at one level and (2) the number of non-comment source lines (NCSL) at another. Software reuse measures were applied to the source code of 20 project applications comprising nearly 500,000 lines of code. To keep the tables in this report manageable and readable, measurements are presented below for 6 of the 20 analyzed applications, selected to help highlight various points in the discussion and to show the diversity within the family of applications.

App	Description
B	Manage a signaling network.
C	Monitor switch memory use; a batch report writer with its own data analysis library.
K	Monitor a digital network service; reports scrolled on the screen or printed.
M	Manage and provision physical memory in switches; offers many dynamic forms populated by database calls.
Q	Manage a new network service; offers many reports for a variety of users.
R	Size future switch memory and software; produces batch reports.

Table 1. Six of Twenty Applications Studied

The five reuse measures used in this study are:

- Code expansion via macros,
- Reuse frequencies of library components,
- Reuse percentages of library components,
- Code profiles, and
- Reuse growth factors.

Each measure reveals some aspect of software reuse and leads to different insights and observations about the code. Code profile and reuse growth factor are derived below in this study from the model in reference [4].

Code expansion via macros

The highest layer of reusable code consists of macros that implement parameterized, high-level reusable objects and generate EASEL code. These macros encapsulate details of complicated tasks that are tedious to implement directly in EASEL. For example, a common construct to display information is a dynamic form in which the total number of form fields is a run-time parameter; fields are typically populated by database calls and may be edited by users to update the database (after passing any required validation procedures). Some applications include many dynamic forms of varying sizes. Using the macros, a developer can specify the needed form in a dozen lines of code, which may be expanded to hundreds of lines of EASEL code.

App	Application Frames			
	No. Frames	Source NCSL	Generated NCSL	Expansion Factor
B	34	2,511	5,629	1 : 2.24
C	4	251	734	1 : 2.92
K	133	7,320	23,449	1 : 3.20
M	178	8,251	62,277	1 : 7.55
Q	248	26,497	81,216	1 : 3.07
R	112	7,372	15,368	1 : 2.08

Code Generated by Macro Calls

Table 2. Code Expansion Factors

Table 2 shows for each selected application the number of frames, the non-comment source lines (NCSL) of frame code written by the developers, and the NCSL of generated code. The ratio of the NCSL counts is the code expansion factor. Applications in the sample vary from 4 to over 200 application-specific frames containing anywhere from 250 to 26,000 NCSL. Macro expansion of the source code results in 700 to 80,000 lines of code with code expansion factor ranging up to 1 : 7.55 for Application M, which makes the most use of macros for dynamic menus and forms.

Code expansion factors affirm to developers that reuse is significant and encourage them to make more extensive use of the macros. The literature, however, suggests that code expansion factors overstate reuse (see [2]), as the code generated by a macro is counted each time the macro is used. Furthermore, this measure does not account for reuse of important components in other architectural layers, such as the reusable frames and reusable C functions in this project.

Reuse frequencies of library components

As a part of on-going maintenance of reusable code, it is useful to know usage frequencies of various components. Frequently used components are often well-tested and well-designed, so less maintenance effort is needed for them while infrequently used components need more examination.

Calls to Top 5 Library Components			
Component	Comm. Macros	Comm. Frames	Comm. C Functions
First	593	13,791	4,427
Second	593	9,504	2,797
Third	593	7,616	2,658
Fourth	422	2,091	1,401
Fifth	416	1,860	1,221

Number of Calls to Library Components				
App	Comm. Macros	Comm. Frames	Comm. C Functions	Total Calls
B	658	699	374	1,731
C	62	59	4	125
K	344	1,580	1,679	3,603
M	965	9,264	4,545	14,774
Q	4,760	8,636	1,881	15,277
R	860	1,179	1,075	3,114

How Often Library Components Are Called

Table 3. Reuse Frequencies

The top portion of Table 3 shows how often the top five components in each library are called by all applications. The three most popular common macros (used 593 times) are typically used as a set. Though it is not true here, such a numerical coincidence is sometimes useful in pointing out a higher level abstraction that can encompass the given macros. The common reusable frame with the highest number of calls, 13,791, is used for displaying current status information.

The lower portion of Table 3 shows how many calls each application makes to components in each of the libraries, with the last column showing the total number of component calls for each application.

While high numbers in either portion of the table reflect favorably on reuse, they are unnormalized frequency counts. An application with twice as much code might have twice as many calls to reusable components, yet the intensity of reuse may be similar.

Reuse percentages of library components

Producers of reusable components are often interested in finding out how much of a library is being used by applications (see [4]). Table 4 reports the number of components called in each of the project libraries. The INLIB row shows the total number of components in each library, including internal library functions, and the fact that all of them (100%) are being used.

App	Comm. Macros		Comm. Frames		Comm. C Functions	
	No.	%	No.	%	No.	%
B	48	43	27	45	97	44
C	17	15	10	16	6	2
K	44	39	21	35	90	41
M	46	41	25	42	118	54
Q	46	41	18	30	71	32
R	37	33	20	33	103	47
InLib	111	100	59	100	217	100

How Much Is Reused From Each Library

Table 4. Use of Reusable Library Components

Application B makes use of 45% of the common frames while Application M uses 54% of the common C function library. From the point of view of managing the reuse feedback cycle, such high percentages indicate that library producers can benefit from direct communications with application developers to learn of possibilities for improving components in reusable libraries.

Application C, on the other hand, makes only modest use of the three common reusable component libraries; the next metric reveals more about this application.

Code profiles

When an existing component is reused, the development effort for a needed functionality is saved. For a particular application, one may ask how much code would have

App	App Frames	App-Spec C Functions	Comm Macros	Comm Frames	Comm C Functions	Entire Application
	%	%	%	%	%	NCSL
B	43	0	18	7	32	5,926
C	18	44	24	11	3	1,370
K	71	2	7	2	18	10,344
M	67	2	7	2	22	12,259
Q	87	4	3	1	5	30,113
R	72	0	6	3	19	10,257

How Much Code Written In Each Architectural Level
Is Used By An Application

Table 5. Application Profiles

been written had the resuable components not been available from other applications. Although this is hard to answer in general, an approximate answer can be obtained by collecting all components used by each application and pretending that they were implemented for that application. The amounts of code collected at the different architectural layers form the code profile for an application.

Each row in Table 5 shows the percentage of an application's total NCSL that resides in each architectural layer (as described in Figure 2). The percentages across a row sum to 100%. The table points out useful information, such as Application B makes extensive reuse of the common library components which comprise 57% (18 + 7 + 32) of its code profile. It is also seen here that Application C, which makes only light use of the common libraries, as noted in Table 4, has 44% of its code in its own analysis library, due to its dependence on unique databases and specialized reports. But even this specialized library benefits from moderate use of common macros, which, in turn, access common frames and common C functions. Application Q also makes little reuse, with application-specific code comprising 91% (87 + 4) of its profile. This means that a new application resembling B may be potentially easier to build than one resembling Q. The next metric provides another perspective on application Q.

Reuse growth factors

Once one has a code profile, one can focus more closely on the application-specific layers and ask how much the code written at these layers "grows" as it links in needed reusable components from lower architectural layers. The total size of all needed code (as indicated by the total NCSL derived in the application profiles, Table 5) relative to the amount of code in the top two application-specific layers is termed the reuse growth factor. Table 6 shows that the reuse growth factors for the selected applications range from 1.11 to 2.38. Only one project application has a reuse growth factor approaching 3.0; it is a recent, small application that took good advantage of existing reusable components.

App	RGF
B	1 : 2.38
C	1 : 1.63
K	1 : 1.41
M	1 : 1.49
Q	1 : 1.11
R	1 : 1.39

How Much Code Is Reused From
Lower Architectural Layers

Table 6. Reuse Growth Factors (RGF)

The poor reuse growth factor and skewed code profile for Application Q deserves further examination. In fact, application Q looks fairly respectable with respect to the other reuse measures. Its code expansion factor is 3.07 (Table 2) and component reuse percentages are 41% for macros, 33% for frames and 32% for C functions (Table 4). Upon further examination, it is seen that Q has about 250 user interface frames (Table 2), which fall into only a handful of distinct styles. Interestingly, multiple instances of a particular style were created by copying and modifying similar EASEL frames. This pointed to an opportunity for another layer of reusable software, which led to a simpler way to specify these similar frames. Now screens of one common style are specified by customizing parameters in an interactive template; this lets developers concentrate on the unique specifications needed for a given screen without having to worry about blocks of repetitive code and macro calls.

Reuse Measure	Interest Group	Strengths	Weaknesses
Code Expansion Factor	Application Developers	Shows power of macros; simple to compute	May overstate reuse; misses components in other architectural levels
Reuse Frequency	Library Developers	Shows which components are heavily, rarely, or never used	Does not indicate reuse intensity; may indicate "popularity" more than value
Reusable Library Components	Library Developers	Shows how much of a library is used by an application; does not overcount reuse	More helpful to library producers than to consumers
Code Profiles	Software Architects	Shows how much code is used at each architectural level	Numbers for each architectural level may obscure interesting relationships
Reuse Growth Factor	Application Developers	Summarizes reuse across all architectural levels	Focuses on an application rather than a library of reuse components

Table 7. Strengths and Weaknesses of Five Reuse Measures

App	Expan. Factor	Reuse Freq.	Reuse Comp.	Code Profle	Growth Factor	Comment
B			+		+	Makes good use of Common Frame and Common C Libraries
C	+	−	−	−	+	Batch report writer; uses Application-Specific Library and Common Macros
K	+		+	−	−	Produces on-screen reports; uses no database update components
M	+		+		+	Newer large application with dynamic forms; uses over 40% of each common library
Q	+	+	+	−	−	Looks good *except* for code profile and growth factor; identifies opportunity for additional reuse
R	−		+			Batch report writer; uses a third of each common library

Table 8. Reuse Summary for the 6 Selected Applications

Discussion

In the previous section, the description of each reuse measure pointed out its particular perspective on software reuse. Table 7 highlights some of the pros and cons of each and indicates the software producer or consumer likely to be interested in the measure. Architects and managers supporting both producers and consumers may find benefits from each.

For the six selected applications, Table 8 summarizes how they fared relative to each reuse measure. Note that no application was uniformly good (+ for top quartile) or poor (− for bottom quartile) across all reuse measures. This indicates that different applications, even those within one project, are able to take advantage of software reuse at various architectural levels and to varying extents. We agree with the literature that macro code expansion factors tend to overstate reuse (as the code generated by a macro is counted each time the macro is used); further, we have seen that macro code expansion factors can ignore reuse of components in reusable libraries. For these reasons, we favor treating macros on the same footing as function calls, which can be done in the framework of reusable library components.

Measures for the use of reusable library components are particularly helpful to producers of common libraries and the measures can accommodate source-code definitions of both macros as well as functions and/or subroutines. Reuse growth factors are particularly helpful to consumers of common libraries who are developing applications; this measure indicates how much advantage they are able to take of common libraries at lower architectural levels. The relevance of these two metrics is not surprising, given their direct theo-

retical grounding on the consumer/producer model.

Even though application developers on the project were quick to point out that reuse growth factors tend to be only about half that of code expansion factors, they accept the new metric since it encompasses code from all architectural levels. Code expansion factors and reuse frequencies are now regarded as comfort measures – nice to know, but not all that informative. Code profiles, on the other hand, can be much more revealing, even though they can produce lots of tabular output; perhaps they are best viewed graphically. From the perspective of application developers as consumers of reusable library components, reuse growth factor distills a key essence of software reuse from the profile data.

In this report, we have related our experience with developing and managing software reuse in a successful large industrial project. To summarize, the main ingredients in our software development framework are:

- Layering software to reflect levels of programming,
- Developing reusable libraries for suitable layers,
- Actively managing feedback loops between consumers and producers of reusable software, and
- Using a variety of reuse measures to confirm that reuse is being done effectively, elicit reuse opportunities, and stimulate development of new reuse components.

The management of the reuse feedback loop (Figure 4) is greatly helped with the use of objective reuse measures to both find out the extent of reuse and identify strengths and weaknesses. The use of object reuse measures in a consumer/producer model implicitly assumes that reusable components are used without modification. In our environment, a reusable repository is expected to continually evolve along with new requirements. However, component reuse can be treated as reuse at any given point in time, since only component producers update libraries and not component users. A recent study by Devanbu *et al.* [5] reported a limited experiment with seven student projects using three libraries: MotifApp, a GNU C++ library, and a C++ database library. It was found that there was no unmodified software reuse. In each case, library code was copied and changed. Our experience suggests that either the students did not discover how the libraries were intended to be used or the libraries were not mature enough and could have benefited from further refinement based on a reuse feedback loop similar to the one in this project.

Finally, though no single reuse measure can give a complete picture, the variety of measures presented and used in this study help form a more balanced and complete view of software reuse. In this project, the average code expansion factor for application frames is 3.5; average reuse of components in reusable libraries ranges up to 54%; reuse growth factors range from 1.5 to 3.0 for some of the newer applications. Together these numbers imply that software reuse in the project may have reduced development effort by a factor of at least 2 and possibly 3 times. These estimates are bolstered by the high reuse frequencies for the top 50 library component calls, which range from 100 to over 10,000. Project management confirms that, without software reuse, at least 2 or 3 times as much staff would have been required to develop and maintain the twenty applications.

References

[1] W. Agresti and W. Evanco. Projecting software defects in analyzing Ada designs. *IEEE Transactions on Software Engineering*, 18(11):988–997, 1992.

[2] B. H. Barnes and T. B. Bollinger. Making Reuse Cost-Effective. *IEEE Software*, 8(1):13–24, January 1991.

[3] Y. F. Chen. Reverse Engineering. In B. Krishnamurthy, editor, *Practical Reusable UNIX Software*. John Wiley & Sons, 1995. Chapter 6.

[4] Y. F. Chen, B. Krishnamurthy, and K. P. Vo. An Objective Reuse Metric: Model and Methodology. In *Fifth European Software Engineering Conference*, September 1995.

[5] P. Devanbu, S. Karstu, W. Melo, and W. Thomas. Analytical and Empirical Evaluation of Software Reuse Metrics. In *The 18th Internation Conference on Software Engineering*, pages 189 – 199, 1996.

[6] G. S. Fowler, J. J. Snyder, and K. P. Vo. End User Systems, Reusability, and High Level Design. In *USENIX Symposium on Very High Level Languages*, pages 101–118. USENIX, October 26-28 1994.

[7] W. Frakes and C. Terry. Software Reuse: Metrics and Models. *ACM Computing Surveys*, 28(2):415–435, 1996.

[8] J. Gaffney and T. Durek. Software reuse – key to enhanced productivity: Some quantitative models. *Information and Software Technology*, 31(5):258–267, 1989.

[9] B. Krishnamurthy, editor. *Practical Reusable UNIX Software*. John Wiley & Sons, 1995.

[10] W. Lim. Effects of Reuse on Quality, Productivity, and Economics. *IEEE Software*, 11(5):23–30, Sept. 1994.

[11] H. Mili, F. Mili, and A. Mili. Reusing Software: Issues and Research Directions. *IEEE Transactions on Software Engineering*, 21(6):528–562, June 1995.

[12] J. Poulin, J. Caruso, and D. Hancock. The Business Case for Software Reuse. *IBM Systems Journal*, 32(4):567–594, 1993.

[13] R. F. Rey, editor. *Engineering and Operations in the Bell System*. AT&T Bell Laboratories, 1983.

[14] J. J. Snyder and K. P. Vo. EASEL – An Application-Building Language. In B. Krishnamurthy, editor, *Practical Reusable UNIX Software*. John Wiley & Sons, 1995. Chapter 4.2.

[15] K. P. Vo. IFS: A Tool to Build Application Systems. *IEEE Software*, 7(4):29–36, July 1990.