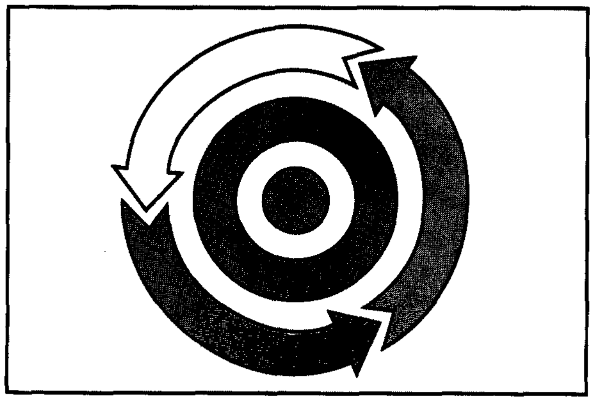
6. R. Glazer, "Measuring the Value of Information: The Information-Intensive Organization," *IBM Systems Journal* **32**, No. 1, 99–110 (1993).
7. J. S. Poulin, "Issues in the Development and Application of Reuse Metrics in a Corporate Environment," *Fifth International Conference on Software Engineering and Knowledge Engineering*, IEEE, San Francisco, CA (June 16–18, 1993), 258–262.
8. J. S. Poulin, D. Hancock, and J. M. Caruso, "The Business Case for Software Reuse," *IBM Systems Journal* **32**, No. 4, 567–594 (1993, this issue).
9. G. Booch, *Software Engineering with Ada*, Benjamin Cummings, Menlo Park, CA (1987).
10. STARS, *Repository Guidelines and Standards for the Software Technology for Adaptable, Reliable Systems (STARS) Program*, IBM CDRL No. 0460, STARS Technology Center, Affiliates Desk, Suite 400, 801 N. Randolph Street, Arlington, VA 22203 (March 15, 1989).
11. J. Nielsen, *Hypertext and Hypermedia*, Academic Press, Inc., New York (1990).
12. W. B. Frakes, "Software Reuse, Quality, and Productivity," *Proceedings of the International Software Quality Exchange '92*, Juran Institute, Inc., San Francisco, CA (1992), pp. 9-9 to 9-18.
13. R. Prieto-Diaz and P. Freeman, "Classifying Software for Reusability," *IEEE Software* **4**, No. 1, 6–16 (January 1987).
14. E. Karlsson, S. Sivert, and E. Tryggeseth, "Classification of Object-Oriented Components for Reuse," *Proceedings of TOOLS'7*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1992), pp. 1–13.
15. RIG Technical Committee on Asset Exchange Interfaces, "A Basic Interoperability Data Model for Reuse Libraries (BIDM)," Reuse Interoperability Group (RIG) Proposed Standard RPS-0001, April 1, 1993. Note: The Reuse library Interoperability Group is a group of government, industry, and academic participants interested in the development of interoperability solutions. Their material is available from AdaNET (telephone 800-444-1458) and ASSET (telephone 304-594-3954), or RIG Secretariat, c/o Applied Expertise, 1925 North Lynn Street, Arlington, VA 22209.
16. R. Prieto-Diaz, "Implementing Faceted Classification for Software Reuse," *Communications of the ACM* **34**, No. 5, 88–97 (May 1991).
17. K. Laitinen, "Document Classification for Software Quality Systems," *ACM Software Engineering Notes* **17**, No. 4, 32–39 (October 1992).
18. Y. S. Maarek, D. M. Berry, and G. E. Kaiser, "An Information Retrieval Approach for Automatically Constructing Software Libraries," *IEEE Transactions on Software Engineering* **17**, No. 8, 800–813 (August 1991).
19. K. P. Yglesias, "Limitations of Certification Standards in Achieving Successful Parts Retrieval," *Proceedings of the 5th International Workshop on Software Reuse*, Palo Alto, CA (October 26–29, 1992), pp. YGL 1-5.

K. P. Yglesias
IBM Large Scale Computing Division
Poughkeepsie
New York



A reusable parts center

In 1991 the Reuse Technology Support Center was established to coordinate and manage the reuse activities within IBM. One component of the established reuse organization was a Reusable Parts Technology Center in Böblingen, Germany, with the mission to develop reusable software parts and to advance the state-of-the-art in software reuse.

The history of the Böblingen parts center dates back to 1981. It started as an advanced technology project looking at methods for reusable design. A recent activity was to develop a comprehensive class library for C++**. This library is offered together with an IBM product (IBM C Set ++ compiler).

In the beginning the goal of the project was to have an integrated software development system that supported the reuse of parts. A first approach tried to find appropriate parts by analyzing existing code, but lead to the belief that parts of code can easily be reused if they are realizations of abstract data types. Projects that followed verified this, and now the parts center offers implementations of abstract data types for different languages and operating systems. This entry in the forum describes how the parts center evolved and what experiences were gained by this effort.

The need for a parts center. Before the existence of a reusable parts center, reuse of code across project borders seldom took place. No organizational structures supported cross-project communication. In addition, the lack of a common design language made communication difficult. Many different description methods for code were in existence. An attempt was made to introduce a common design language within IBM, but it met with little success.

The code that was written day after day was not designed to be reused. References to global variables and shared control blocks were spread all over the code. Also, hardware or operating system dependencies were normally not isolated. Modularization was not used widely, so internal interfaces were not defined or, if defined, not standardized. The programming languages used did not support reuse, nor did they support good software techniques such as information hiding and encapsulation. Therefore, the code usually could not be reused without changes. As a result, only coding effort could be saved, but not the surrounding work of design, test, and maintenance.

Reusable design. Early investigations into reusable design showed that common structures exist in the area of data structures, operations, and modules. To avoid such parallel efforts in the future, a formal software development process could be established where inspection, verification, testing, and documentation of the code would be performed once and then made available to other development groups. This should lead to a gradual build-up of a central parts and tools database containing high-quality parts.

It was expected then, that developers could change their way of writing software so that an additional outcome of each project would be high-quality parts that could be supplied to the common database. To support the developers, a development system offering the following reusable design facilities was called for:

- A design language that allows formal specification of module interfaces and function semantics
- A process to ensure the correctness of a piece of design by prototyping, correctness proofs, or testing
- A database system that allows the storage and retrieval of a piece of design with a generalized search facility

The approach to find appropriate parts was to scan existing products and to identify replicated functions, either on the design or on the code level. Soon it became clear that an abundance of dependencies on global data existed. This led to the hypothesis that data abstraction, i.e., bundling the data with their operations, is the key to reusability.

Reusability through abstract data types. In 1983 a project was started to explore data abstraction as a software development and reuse technology in a real software product: a network communication program.

For this project, seven abstract data types called *building blocks* were written that represented a third of the total lines of code. Two implementation techniques for abstract data types, i.e., macro expansion and reentrant procedures, were investigated. Besides other advantages, the procedure approach supported the hiding of information and protection of the data against disallowed access, but needed temporary storage that had to be allocated and freed dynamically. A project report that discusses the advantages of the different approaches states that the decision has to be taken for each case individually and that a language supporting data abstraction in design and implementation is needed.

In the course of the pilot project the following experiences with abstract data types were observed:

- Abstract data types were a stabilizing factor during the design phase. Even drastic design changes had only a minor effect.
- Isolating data that logically belong together proved to be correct. Where logically unrelated data were packed together, the design became too complex and had to be changed later.
- The availability of exactly specified abstract data types reduced the complexity of the design.
- Since the abstract data types were tested thoroughly, the testing of the remaining code was speeded up significantly.

The building-block approach. Due to the good experiences with abstract data types, the features that are most important for reuse were summarized in a reuse model. Postulating five conditions of reuse, the future building-block approach was determined. The conditions are:

1. *Information hiding.* One does not need to know the inner workings of reused code; a description suffices in all respects for reuse.

2. *Modularity.* Reusable code reflects entities of functional coherence, consistency, and comprehensiveness.
3. *Standardization.* Reusable code is standardized code.
4. *Parameterization.* Reusable code is to be adapted to the specifics of reuse instances through controlled substitution.
5. *Testing and validation.* Reusable code must be error-free.

The project showed that the programming language (PL/S) primarily used within IBM at the time lacked features that would support the reuse conditions. Therefore, a language extension was envisioned, developed, and validated on a second pilot project with great success.[1]

The building-block language extension (BB/LX) follows the model of generic packages of Ada, with private parts for the implementation of abstract data types under the principle of information hiding. We found that something like the Ada library system was essential because it can bind code entities to programs without seeing the internals of the bound entities and because it can support the binding with as much syntax and semantics checking as possible.

In the PL/S environment there are two library systems: link libraries that contain output from compilation or assembly in a relocatable form, and macro libraries that contain macros and pieces of source code to be generated into the user code. Both approaches were investigated.

In the link library approach, code entities to be reused were PL/S procedures compiled individually with a self-contained scope of variables. An abstract data type was implemented according to Denert's approach.[2] Each procedure had multiple entries (one for each abstract data type operation) and a common part in which a data representation was declared for the abstract type.

This approach proved unsatisfactory because:

1. The required procedure call overhead, even for simple operations, was too high for people to accept reuse.
2. Parameters that were by their very nature generic had to be repeated on each and every operation invocation, making interfaces very complex and cumbersome.
3. The complete absence of compiler support to check the correctness of calls to the procedures (or procedure entries) caused much frustration for users.

In the macro-based approach, the performance problem is solved through in-line generation of reusable code. Support of generic parameterization is the domain of macro processing anyway, so we could program into BB/LX the syntax checking and the static semantics of building block use with fairly standard techniques.

Thus the link-library-based approach was abandoned in favor of the macro-library-based approach, which evolved into BB/LX.

BB/LX now provides the following functions:

- It checks the syntax of building-block operations as invoked by the user.
- It stores and maintains generic parameters for building blocks.
- It provides a mechanism like the private mechanism of Ada (to implement abstract data types).
- It performs static semantics analysis.

The goal of error-free code. As reusable code is expected to be used numerous times, any contained error would be multiplied; therefore, it is crucial that reusable code is error-free.

Fortunately, this goal can be reached easier for reusable code than for ordinary code—and the second pilot project proved that. The reason lies in the modularity and the independence of the building blocks.

A building block can be tested very completely before its actual use. This is possible because no modification of the code is required—or allowed. In addition, a test program can handle the testing easily because a building block is a well-structured and separate entity, has an accurate well-defined user interface, and has no other interdependencies with the user program. It is sufficient to call building-block operations from the test program. Usually, no scaffolding and setup of a test environment is required.

Also, the building blocks do not depend on any application or environment. Different hardware platforms are supported, making it possible to test building blocks on other systems than the one for which they were developed. Thus, independent testing is said to increase heavily the confidence of the users. Since building blocks are platform-independent, they can be tested by an independent group, even if they have a different software or hardware environment.

When programmers use a building block, they cannot introduce defects into the building-block code. This means that once a building block is free of defects it will remain free of them. Therefore, the quality inherent in building blocks will directly and proportionally improve the quality of the product in which they are reused.

Our experience has proven the quality gains to be significant. In one project, the quality (number of errors per lines of code) of the building-block code was about 9 times better during the function test and 4.5 times better during the component and system test than the rest of the code.

In another completed project, the results were even better. No errors in the building blocks were found during the entire test cycle of the product.

Development of the reuse environment. As more and more building blocks were produced, tools became necessary to support the user in finding and integrating the building blocks into programs and the environment. The reuse environment was intended to be the integrated development environment for users of Böblingen's software building blocks. As such, it supported ordering, shipping, maintenance, and (re)use of building blocks. It also offered comprehensive information about available building blocks and support communication among building-block users and developers (information exchange, problem reporting). Special tools for creating, compiling, and testing building-block applications were also provided.

Users could select and order building blocks from a central repository through the interactive functions provided by the reuse environment. A multilevel on-line information system (including a catalog of the available building blocks and an information base of all building-block specifications) aided in selecting the required building blocks.

The reuse environment was a prototype whose purpose was to gain experience with software reuse support in IBM programming centers. The experiences were expected to be used in the definition and design of a production reuse environment for systems programming.

Standardization. The expectation that BB/LX would enable the developers to write their own building blocks was not realized. Instead, there was increased demand for building blocks produced by a parts center. Since building blocks were developed in response to specific project requirements, they were often too specialized for general use and available too late. It was thus necessary to construct a comprehensive library of readily available reusable parts that had more general applicability.

As the building blocks were created for the common library, it became obvious that it is not enough to design separate building blocks carefully; the collection of building blocks needed to be designed as a whole.

We needed to standardize and generalize interface and behavior of the entire set of building blocks by defining common concepts for the different abstract data types.

As preparation for this building-block revision, a project was set up in 1984 that resulted in a comprehensive catalog of reusable abstract data types. This eventually led to the development of the Böblingen building-block catalog, which now defines and establishes the following for all BB/LX building blocks:

- Comprehensive collection of abstractions—all abstract data types described in standard textbooks, such as stack, queue, list, set, map, etc., are contained.
- Consistent and complete user interfaces—all building blocks of one abstraction offer the identical abstract interface, which is checked for completeness.
- Uniform operations semantics—operations with the same name behave conceptually alike.
- Uniform attributes—abstractions with the same attribute offer similar operations.
- Common terminology—the abstractions are described using a common vocabulary and common concepts.
- Hierarchical implementation and abstraction

structure—overview information and categorizing information is supplied.

The Böblingen building-block catalog describes about 10 different implementation features. The user can choose linked or unlinked implementation, bounded or unbounded implementation, and much more. This results in more than a thousand different combinations per abstraction. Although G. Booch does not offer as many implementation features in his book,[3] he still has over 400 different abstract data types with over 100 different queues.

To save coding effort while building the comprehensive library, a technique was developed that combined the use of generic characteristics with the ability to derive abstract data types from similar ones. This technique also allowed the reuse of specifications, test cases, and documentation. Thus the expectation that reuse saves effort in all development phases did materialize.

Production of C++ parts. In parallel with the implementation of the reuse methodology in PL/S, investigations of the use of the C++ language for building blocks started.[4] Since the required concepts of encapsulation, information hiding, and modularity are supported by C++, the development of a language extension was not necessary.[5] Also, no special technique to derive building blocks was necessary since the concept of inheritance is supported.

So the implementation of the building-block library for C++ did not require as much start-up effort as the development for PL/S. In August 1991, the first release of the class library (IBM's Collection Class Library) for C++ became available within IBM. By the end of 1991, 57 projects with 340 programmers were already using the C++ building blocks. In June 1993, the Collection Class Library became available as part of the IBM C Set++ compiler, for C++.

Support. After production for building blocks was established, it became obvious that for the parts that were not produced on a special order, support for the users is necessary.

Today, in system programming, assembler code is still being written or modified. The use of PL/S still is not widely accepted, and the use of still higher elaborated techniques is far beyond the normal daily working experiences.

In contrast, the experiences with the user support and consultancy for the C++ building blocks show that in the C++ domain, programmers are much more motivated to make themselves familiar with a new technology.

What is true for both domains is that few developers use abstract data types as if they were a usual part of the programming language. This makes support and consultancy in selecting and applying building blocks necessary.

Summary. During the creation and the establishment of the parts center, our group discovered what language features and what infrastructure were needed to enable the building and the distribution of reusable components. Many of the problems we had to solve were caused by the inadequate support of reuse through the language we initially used.

Significant progress was achieved when object-oriented languages like C++, which support reuse inherently, became available. Our experience showed that it is not enough to use an object-oriented language to make reuse happen, but high-quality class libraries are essential to get the desired productivity improvements.

**Trademark or registered trademark of AT&T.

Cited references

1. M. Lenz, H. A. Schmid, and P. F. Wolf, "Software Reuse Through Building Blocks," *IEEE Software* **4**, No. 4, 34–42 (July 1987).
2. E. Denert, "Software-Modularisierung," *Informatik-Spektrum* **2**, 204–218 (1979).
3. G. Booch, *Software Components with Ada: Structures, Tools, and Subsystem*, The Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA (1987).
4. J. Uhl and H. A. Schmid, "A Systematic Catalogue of Reusable Abstract Data Types," *Lecture Notes in Computer Science*, G. Goos and J. Hartmanis, Editors, Springer-Verlag, Berlin (1990).
5. M. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, AT&T Bell Laboratories, Addison-Wesley Publishing Co., Reading, MA (1990).

D. Bauer
IBM Large Scale Computing Division
Böblingen
Germany