HEWLETT-PACKARD
JOURNAL
February 1997

HEWLETT-PACKARD

table of contents

February 1997,
Volume 48, Issue 1

Articles

Building Evolvable Systems: The ORBlite Project
by Keith E. Moore and Evan R. Kirshenbaum

Developing Fusion Objects for Instruments
by Antonio A. Dicolen and Jerry J. Liu

An Approach to Architecting Enterprise Solutions
by Robert A. Seliger

Object-Oriented Customer Education
by Wulf Rehder

SoftBench 5.0: The Evolution of an Integrated Software Development Environment

The vision and objectives of the original SoftBench product have enabled it to continue to be a leader in the integrated software development market. For example, since SoftBench 1.0, over 80 third-party software tools have been integrated with SoftBench.

by Deborah A. Lienhart

HP SoftBench is an integrated software development environment designed to facilitate rapid, interactive program construction, test, and maintenance in a distributed computing environment. The SoftBench product contains an integration framework and a set of software development tools, as well as the ability to integrate tools from other sources.

SoftBench was released in 1989 and presented in the June 1990 HP Journal.[1] At that time, no one would have guessed the market changes that would occur during SoftBench's life. Fortunately, the vision and objectives of the original product designers have allowed SoftBench to continue to be a leader in the integrated software development market.

This article presents the actions that have made SoftBench a standard in the integrated software development market, the original SoftBench objectives that have stood the test of time, and the new technologies that have been incorporated into SoftBench. Other articles in this issue will present more information about the new technologies in SoftBench.

The different versions of SoftBench released since its introduction in 1989 are shown in Fig. 1.

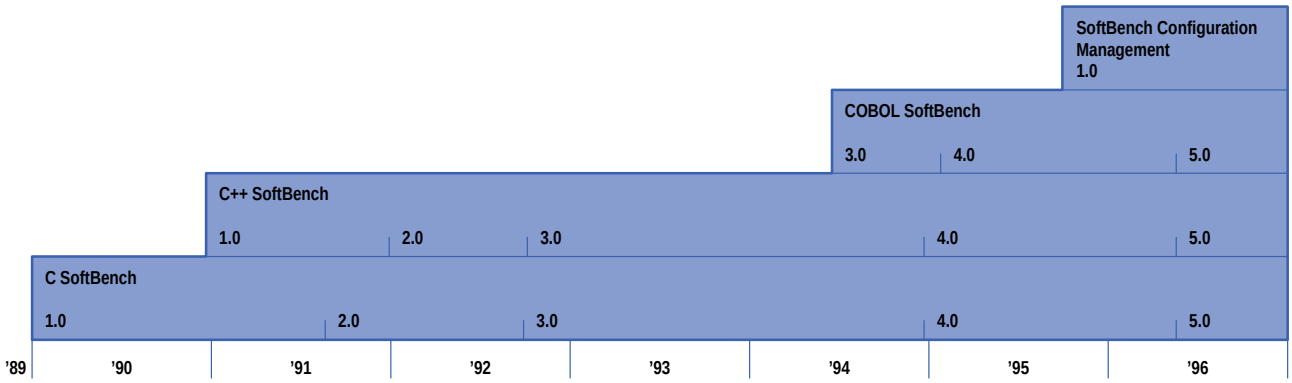


Fig. 1. *The different versions of SoftBench released since 1989.*

Making SoftBench the Standard

SoftBench defined the open, integrated CASE (computer-aided software engineering) market. The first big challenge was to make SoftBench pervasive in the market. We used several approaches, including leading standards development, working with software tool providers, and licensing the framework source code. HP started and supported CASE Communiqué, a standards body that focused on defining the messages used for intertool communication. This work was adopted as the basis of intertool communication standards for software development tools by the ANSI X3H6 Committee.

HP worked with software tool providers, both through CASE Communiqué and with independent software vendor (ISV) programs, to provide SoftBench integration for their tools. There have been over 80 third-party software tools integrated with SoftBench, and we continue to see interest from software tool vendors who want to integrate their tools with SoftBench.

The source licensing program was interesting to many companies for a number of reasons. Some companies ported SoftBench to their hardware, added some tools, and sold it to their customers. Several other companies have ported SoftBench to their own hardware for use by their internal development organizations. One company, SAIC (Science Applications International Corporation), conracts with customers to provide cross-development support for other, usually

non-UNIX,® platforms. This is used mainly for legacy system support or to develop software for platforms that can't support a native application development environment. This is the only part of the source licensing program that is still active. ***Article 5*** describes the activities of the Science Applications International Corporation.

The SoftBench broadcast message server (BMS) framework was adopted by other HP products and some customers' products, in addition to its use in SoftBench. The biggest user of the BMS framework is HP VUE (Visual User Environment). BMS provides the same open integration of desktop tools in HP VUE that it provides for software tools in SoftBench. For software developers, BMS also provides an integration of the desktop tools with software development tools.

Original Objectives of SoftBench

The SoftBench framework continues to provide the foundation for SoftBench and has stood the test of time. The following are the original objectives of the SoftBench architecture and the changes that have taken place.

Support Integrated Toolsets. This goal dictated that the SoftBench tools should cooperate to provide a task-oriented environment that lets users concentrate on what they want to do, not how to do it. SoftBench continues to provide a task-oriented environment by allowing tools to be started from each other. For example, in most SoftBench tools you can show the references for any symbol via the static analyzer.

Some users prefer a tool-focused environment, so SoftBench 5.0 has a new ToolBar to make it easier to see what tools are available in SoftBench (see Fig. 2).

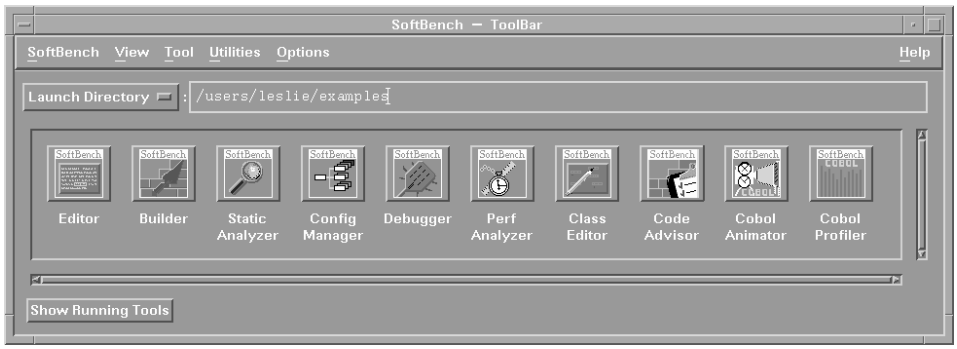


Fig. 2. *A ToolBar screen.*

Support Interchangeable Tools. The concept of plug and play, which allows users to exchange a SoftBench-supplied tool with one of their preference, has guided SoftBench's architecture and the development of standards in the CASE industry. Text editors and configuration management systems are the most common tools that users customize.

Support a Distributed Computing Environment. This goal required that all tool execution, data, and display should be designed for a network environment. This objective was based on the scenario of a software development team using a group of workstations with varying capabilities and shared project files on a central server. Providing distributed computing support in SoftBench has not only allowed it to work well in this scenario, but also has provided additional benefit in the ability to target computers and embedded systems that cannot support an application development environment. The biggest SoftBench customers make use of this capability.

Leverage Existing Tools. The reason for this objective was to protect customers' investments in software development tools by allowing these tools to fit into the SoftBench environment without modifying source code. This has worked well for lightweight integrations, but most customers have decided that the increased value of a deeper integration is worth adding a simple module to their source code.

Support Software Development Teams. Originally SoftBench included integration with RCS (Revision Control System) and SCCS (Source Code Control System) configuration management tools and support for accessing shared project files. In SoftBench 5.0, the SoftBench configuration management product SoftBench CM was added. SoftBench CM is based on the history management server, which has been used internally in HP for many years. SoftBench CM provides global source code management for software development teams whose members can be located anywhere around the world.

Support Multiple Work Styles. Software engineers do a number of different tasks during the course of a project, including design, prototyping, construction, defect fixing, and maintenance. Each of these tasks requires a different emphasis of the software development tools. For example, construction makes extensive use of the editor and builder, defect fixing is centered in the debugger, and maintenance starts with the static analyzer. Each of the tools is accessible from the others, which allows a task to have quick access to multiple tools or to transition between tasks.

Support Other Life Cycle Tools. SoftBench supports the integration of other tools that support the software life cycle, including documentation, test, defect tracking, and design tools. Most of the third-party tools integrated with SoftBench are in these categories.

Build on Standards. SoftBench has always been built on standards, such as the UNIX operating system, NFS and ARPA networking, the X Window System, and the OSF/Motif appearance and behavior. In SoftBench 5.0 we added integration with the Common Desktop Environment (CDE),[2] including CDE drag and drop.

New Technology in SoftBench

In the years since the first release of SoftBench, the breadth of tool support and functionality of the tools has increased significantly. This section briefly describes some of these additions.

Static Database. In SoftBench 3.0 a new object-oriented static database was placed under SoftBench's static analyzer. Earlier versions of the static analyzer could only analyze 30,000 to 40,000 lines of source code before reaching capacity limitations. The new static database does not have capacity limitations and performance is acceptable for up to about one million lines of source code.

In addition to the capacity and performance improvements, the object model of the new static database makes it more flexible for adding language types and queries. The SoftBench static analysis database is described in ***Article 3***.

Rule Engine. In SoftBench 5.0 a rule engine was implemented as part of the SoftBench CodeAdvisor product. A rule is implemented as a C++ class, which can access information in the static database and any other information available to it. The rules are run by the rule engine, which is integrated into the SoftBench program builder.[3]

A set of C++ coding rules is included in SoftBench 5.0. These rules check for dangerous coding practices, which are the ones that would create memory leaks or have unanticipated side effects. Information and examples needed to create rules are included in the SoftBench software developer's kit.

The SoftBench CodeAdvisor is described in ***Article 4***.

New Languages. The first release of SoftBench supported the C language. C++ SoftBench was added in 1991. C++ enhancements were made to the SoftBench tools and a C++-specific tool, the C++ Developer, was added. The C++ Developer was designed to be a training tool. It had a graphic display of the class inheritance hierarchy, and the user could add or delete classes and inheritance relationships from the graph. It could also automatically fix common coding problems before they were caught by the compiler. In SoftBench 5.0, the C++ Developer was replaced by the graphic editing functionality in the SoftBench static analyzer's class graph.

COBOL SoftBench[3] was added to the product family in 1994. It provides encapsulations of most of the MicroFocus COBOL tools.* The SoftBench development environment makes it easier for users to transition to the UNIX operating system from mainframe development environments. COBOL SoftBench provides a common development environment for C, C++ , and COBOL. This is especially helpful when debugging an application that is a combination of COBOL and C or C++. MicroFocus' Animator and SoftBench program debugger pass control of the application between themselves as the application moves between modules implemented in different languages.

SoftBench CM. The SoftBench configuration management product was introduced in 1995. It is based on the history management server, an internal tool that has been used for most of HP's software development.

SoftBench CM is a scalable configuration management tool that offers efficient code management capabilities for team members and work groups, including those who are geographically dispersed in distant locations. Based on a client/server architecture that is designed to allow access to multiple local or remote servers, SoftBench CM is easily accessed from any of the SoftBench tools (see Fig. 3).

SoftBench CM can manage different versions of any type of file. Many of our customers use SoftBench CM to version nonsoftware files, including project documents and bitmaps. A PC user interface has been developed that allows users in mixed UNIX and PC environments to create versions of their PC-based files along with their UNIX-based files.

Graph Views. In the original version of SoftBench there were only textual interfaces. In SoftBench 3.0, graphical interfaces were added to many of the SoftBench tools, including the dependency graph browser in the program builder, the static graph browser in the static analyzer, and the data graph browser in the program debugger.

In SoftBench 4.0 the underlying graph library was replaced by an implementation based on a third-party graphics library called ILOG Views. This implementation is much faster and will handle a lot more nodes than the old implementation. The static graph browser was replaced with three specialized graphs for files, functions, and classes.

In SoftBench 5.0, graphical editing capability was added to the SoftBench static analyzer's class graph and its name was changed to the class editor. ***Article 2*** describes the C++ SoftBench class editor.

* The COBOL SoftBench family is based on HP MF COBOL, HP's implementation of MicroFocus COBOL, which is based on technology from MicroFocus, Ltd.

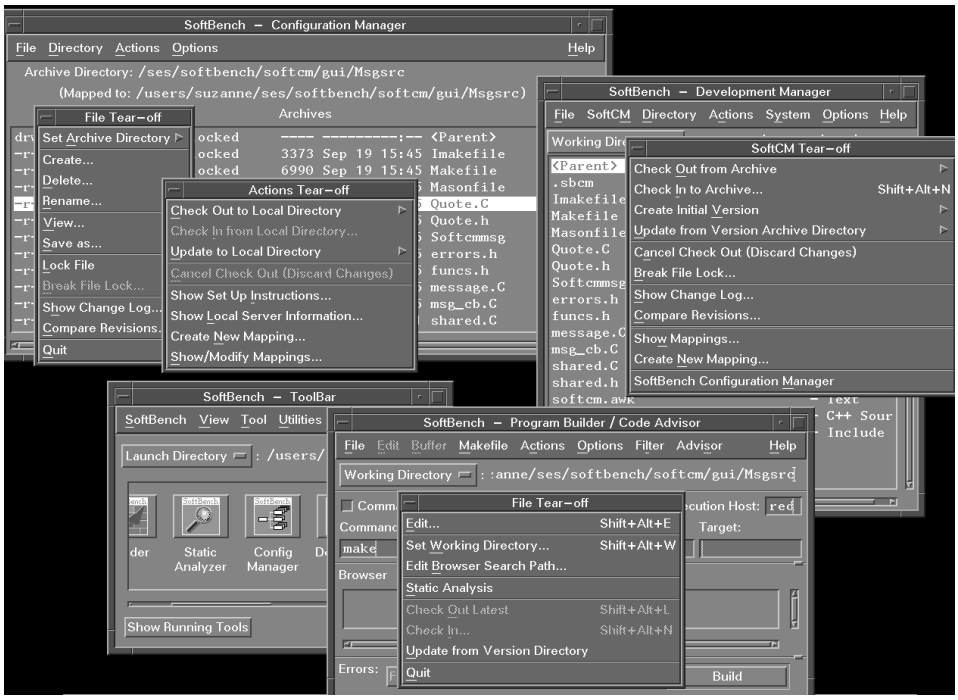


Fig. 3. *A SoftBench CM screen.*

More Platforms. SoftBench originally supported HP 9000 Series 300 workstations and HP 9000 Series 800 file servers. Support was added for the HP 9000 Series 400 and Series 700 workstations and HP 9000 Series 800 servers with X terminals. In 1991 SoftBench was released for SunOS and in 1993 support was added for Sun's Solaris operating system.

DDE Debugger. In SoftBench 4.0, the underlying debugger for SoftBench's program debugger was changed from xdb to the HP Distributed Debugging Environment (DDE) from HP's Massachusetts Language Laboratory. HP DDE's architecture isolates most of the debugger from specific information about the target operating system, computer language, user interface, and debug format. The SoftBench team implemented the SoftBench program debugger user interface on top of HP DDE and ported the whole thing to the Solaris operating system. This is the only development environment that supports both the HP-UX* and Solaris operating systems with a common debugger, using the compilers supplied by the system vendors.

ToolBar. SoftBench is a powerful development environment and as the user base has expanded we've placed more emphasis on making it easier to learn and use. Many times users requested tools that were already in SoftBench so we added an iconic ToolBar to make the available tools visible. The ToolBar supports drag and drop integration with HP VUE and CDE.

Conclusion

When SoftBench was first envisioned, UNIX software development tools consisted of compilers and debuggers, and real software engineers didn't use windows. SoftBench was the first integrated application development environment running on the UNIX operating system.

There wasn't much to work with then, just RFA (remote file access) and TCP/IP networking and the beginnings of the X Window System. Motif came along during the development of the first release of SoftBench and NFS came along later. When HP's Software Engineering Systems Division (SESD) developed the BMS (broadcast message server) for interprocess communication and it was included in HP VUE, it changed the capability of desktops for everyone.

Over the years SESD has developed new technology for the challenges brought on by the C++ language and larger applications. We also added a lot of graphics as the technology became available and workstation performance increased.

In the future, SoftBench will face new challenges associated with developing distributed applications that run in heterogeneous environments. We can look to the original objectives and architecture for a path that has stood the test of time.

References

1. *Hewlett-Packard Journal*, Vol. 41, no. 3, June 1990, pp. 6-68.
2. *Hewlett-Packard Journal*, Vol. 47, no. 2, April 1996, pp. 6-65.

3. C. Carmichael, "COBOL SoftBench: An Open Integrated CASE Environment," *Hewlett-Packard Journal*, Vol. 46, no. 3, June 1995, pp. 82-85.
4. A. Iyengar, T. Grzesik, V. Ho-Gibson, T. Hoover, and J. Vasta, "An Event-Based, Retargetable Debugger," *Hewlett-Packard Journal*, Vol. 45, no. 3, June 1995, pp. 33-43.

Applying a Process Improvement Model to SoftBench 5.0

Software organizations are under market pressure to reduce their cycle time and improve their development processes. The conventional approach is to work on one, usually at the expense of the other. For SoftBench 5.0 we decided to jump right in and attack both using a 12-month release cycle and CMM (Capability Maturity Model) level-2 processes. Using CMM-prescribed project management processes, we reduced SoftBench 5.0's cycle time by 35%, improved product usability, and improved our ability to predict release dates. We also greatly improved the organization's ability to select, plan, estimate, and track software projects.

Reference 1 describes the software improvement project at our division that put in place the CMM process. Here we briefly summarize CMM and our approach to using it for SoftBench 5.0.

Business Environment

SoftBench is an integrated application development environment for C, C++, and COBOL running on UNIX systems. It was first released in 1988. Since then the cycle time (that is, the time between one major release and the next) has varied from 18 to 24 months. In previous releases of SoftBench, the first part of the project was very unstructured. It typically involved market research, customer visits, prototyping, and design, but these activities were not well-integrated. At some point we would decide what functionality should make the release and what functionality would be rescheduled for the next release. A cross-functional team would be put into place to manage and focus the release. This model provided little control over requirements or schedule.

By the time we started SoftBench 5.0, we had taken important steps to improve our product development process. First, we had a life cycle in place based on user-centered design. We had piloted elements of the user-centered design process with SoftBench 4.0, but the life cycle had not been tested on a large-scale project. Second, we had organized into cross-functional business teams, which helped speed alignment between marketing and R&D by putting a single manager in charge of both functions. And finally, we had just completed the SoftBench 4.0 test phase on schedule, proving that we had the ability to plan and schedule the latter phases of a project.

To make matters more interesting, our new division manager, who had experience reducing cycle time, improving quality, and improving predictability using the Software Engineering Institute's Capability Maturity Model (SEI CMM), challenged us to get to CMM level 3 in 36 months, a process that normally takes two to three years just to go from level-1 to level-2 CMM compliance.

Capability Maturity Model

In 1987 the Software Engineering Institute (SEI), based at Carnegie-Mellon University, published the first version of the Capability Maturity Model (CMM). The initial intent of the CMM was to provide a process maturity framework that would help developers improve their software processes.

CMM describes five levels of software process maturity (Fig.1). At the *initial process level* (level 1) an organization operates without consistent application of formal procedures or project plans. When things get tight, the level-1 organization always reverts to coding and testing. At level 2, the *repeatable level*, controls are established over the way an organization establishes its plans and commitments. Requirements, plans, and procedures are documented, at least at the project level, which means the process could be repeated in the future as long as the type of software being developed doesn't change too much. At the *defined level* (level 3), the organization has documented both its management and engineering processes. This allows the organization to begin to improve the processes over time. Level 4, the *managed level*, is where an organization can quantitatively measure its development and management processes. Finally, at level 5, the *optimizing level*, the development process operates smoothly, and continuous improvement occurs on the defined processes established in the previous levels.

For each level of process maturity, CMM describes the related key practices that characterize that level of process maturity.

Each key process area is defined by a set of one or more goals, as well as the specific practices which, if followed, help achieve the goals. The key process areas and practices are intended to describe what needs to be done to efficiently and predictably develop and maintain software. The CMM does not attempt to specify how software should be developed and managed, leaving that interpretation to each organization, based on its culture and experience.

Project Infrastructure

We chose to move to level 3 by adopting CMM level-2 processes immediately on all new projects. SoftBench 5.0 was the first and largest project to use the new processes and our project infrastructure was designed to support this approach. The key components of our project infrastructure were: a life cycle based on user-centered design, a Web server connected to our configuration management system, and a process consultant and a project lead.

The life cycle had been under development for about a year and we had already used it successfully on some parts of the previous SoftBench release. The life cycle uses a simple waterfall model, augmented with CMM level-2 practices and user-centered design.

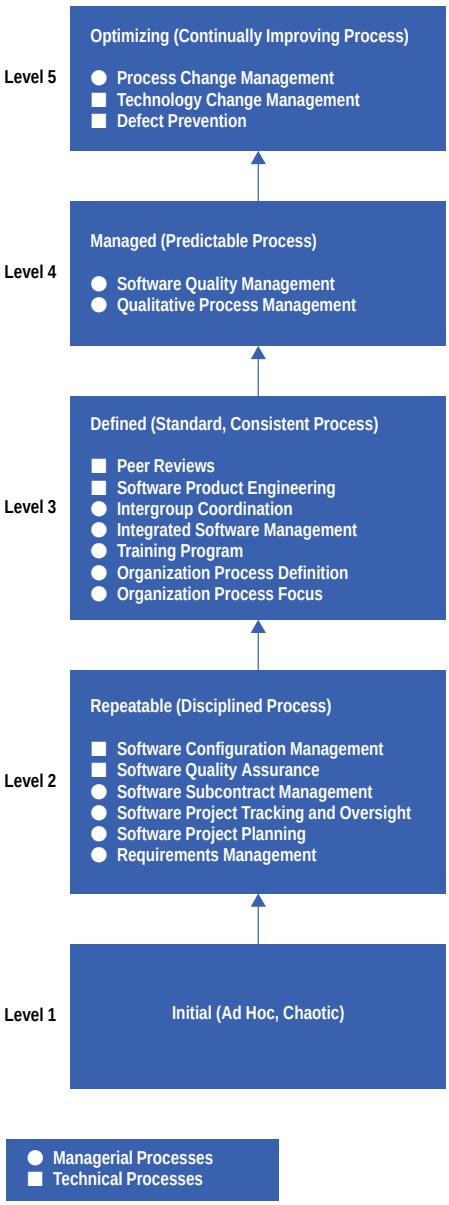


Fig. 1. *The five layers of the software Capability Maturity Model. As an organization adopts the practices specified in the model, its software processes should see greater productivity and quality.*

CMM level-2 practices ensure that requirements, plans, and schedules are documented, reviewed, and approved by management. Moreover, level-2 practices ensure that as requirements or designs change, the associated plans and schedules are revisited to make sure they are still valid.

User-centered design is based on the premise that a product's success depends on how well the product addresses the needs of the people who use it. User-centered design does this by involving potential users in key development activities, such as profiling user characteristics, characterizing goals and tasks, and validating potential product features and design alternatives.

All of our project documents were checked into SoftBench CM, SoftBench's configuration management system. A Web home page was created for the SoftBench project, allowing us to retrieve documents from SoftBench CM and display them with a Web browser, such as Mosaic or Netscape. The home page included a section for each of the SoftBench teams (to point to customer survey data, requirements, and designs), and sections for product documents, project planning documents, project schedules, and life cycle guidance. We've always checked project documents into our configuration management system, but the addition of the Web browser really improved the visibility and access to these documents. Fig. 2 shows our Web intranet structure.

The third key component of our project infrastructure was the process consultant and project lead. We had a full-time project lead and a full-time process consultant focused on the CMM practices, both as part of the formal management team. We also had a half-time user-centered design consultant from our human factors organization to help us apply the user-centered design techniques. Having these two individuals share accountability for both process and project management proved to be a major success factor.

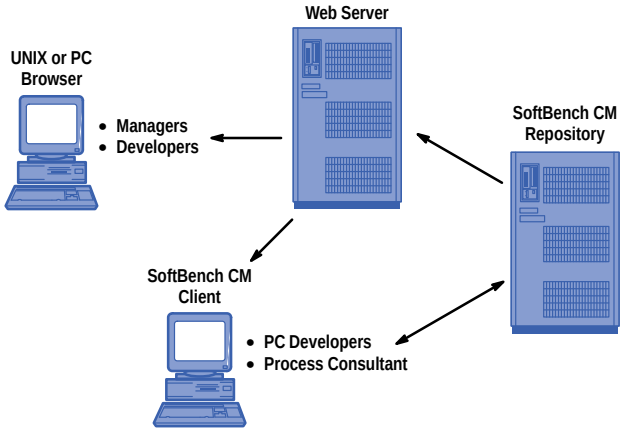


***Fig. 2**. The network configuration that supported the project infrastructure for the development of SoftBench 5.0.*

Reference

1. D. Lowe and G. Cox, "Implementing the Capability Maturity Model for Software Development," *Hewlett-Packard Journal*, Vol. 47, no. 4, August 1996.

Bibliography

1. W. S. Humphrey, *Managing the Software Process*, Addison-Wesley Publishing Company, 1989.

Acknowledgments

Many individuals contributed to the success of SoftBench 5.0 and our SEI initiative. We'd like to specifically acknowledge the following for their hard work and perseverance: Jack Cooley, Guy Cox, Doug Lowe, Alan Meyer, and Jan Ryles.

Deborah A. Lienhart
Project Manager
Software Engineering Systems Division

Scott Jordan
Process Consultant
Software Engineering Systems Division

The C++ SoftBench Class Editor

The C++ SoftBench class editor adds automatic code generation capabilities to the class graph of the SoftBench static analyzer. Novice C++ programmers can concentrate on their software designs and have the computer handle C++'s esoteric syntax. Experienced C++ programmers benefit from smart batch editing functionality and by having the computer quickly generate the program skeleton.

by Julie B. Wilson

The C++ SoftBench class editor allows the programmer to edit the class constructs in a C++ program using the SoftBench static analyzer's graphical interface. Using the class editor, the programmer can create and modify class hierarchies and edit class components.

Since the class editor is part of the static analyzer, let's look first at the functionality provided by the static analyzer. The static analyzer helps the programmer better understand the code. Through static queries, the programmer can understand a program's structure, assess the impact of changes, and change the architecture of the code when necessary. The static analyzer presents a wide variety of information about the code, including information about variables, classes, functions, and files. Through queries, the programmer can answer questions such as, "What functions and classes call this function?" or "What code accesses any element of this class?" The results of the queries can be displayed either textually or graphically. From either display, a simple double click takes the programmer directly to the source code that supports the displayed information.

To use the static analyzer, the programmer must first generate static information about the application. The default compile mode in the SoftBench program builder generates the static database (the Static.sadb file). When the programmer builds the application, the compiler places the static database in the directory in which the programmer compiled the code. All static queries rely on the information stored in this database.

Benefits of the Class Editor

SoftBench 5.0 adds editing capabilities to the class graph provided by the static analyzer. With the class editor, a novice C++ programmer can concentrate on software design, class hierarchy, data members, and member functions, not on C++ syntax. After each edit request, the class editor automatically generates the specified C++ code with correct syntax. The class editor also checks the work and doesn't let the programmer make typical beginner's mistakes like using the same class name twice.

Expert C++ programmers also benefit from the class editor. In addition to the program visualization capabilities of the graph, experts can quickly generate a program skeleton or make changes to an existing program's structure. Even more useful are the powerful, static-assisted edits that the class editor supports. Using the class editor, the programmer can change the name of a class or class member and all the appropriate changes are made in the source code. These changes can span many files. Because of the underlying static database, if the programmer changes the name of a member function x, the class editor knows exactly which instances of x are relevant and which instances are not.

Controlling Complexity

Fig. 1 shows an example of a C++ program with the classes and inheritance relationships displayed. The class editor provides the ability to display many relationships in addition to inheritance, such as friends, containment, and accesses by members of other classes.

Large C++ applications tend to have many classes and many relationships among the classes. The class editor provides several features to help control the complexity of what is displayed:

- Filters make it possible to display only the type of data in which the programmer is interested. For example, if the programmer only wants to see inheritance relationships, all other types of relationships can be filtered so they are not displayed on the graph.
- The programmer can reduce the complexity of the graph by hiding nodes that are not currently of interest.
- The programmer can add nodes to the graph directly by name or indirectly by querying about relationships with nodes already displayed on the graph.
- The programmer can expand and contract class nodes to show the data members and member functions in the node.

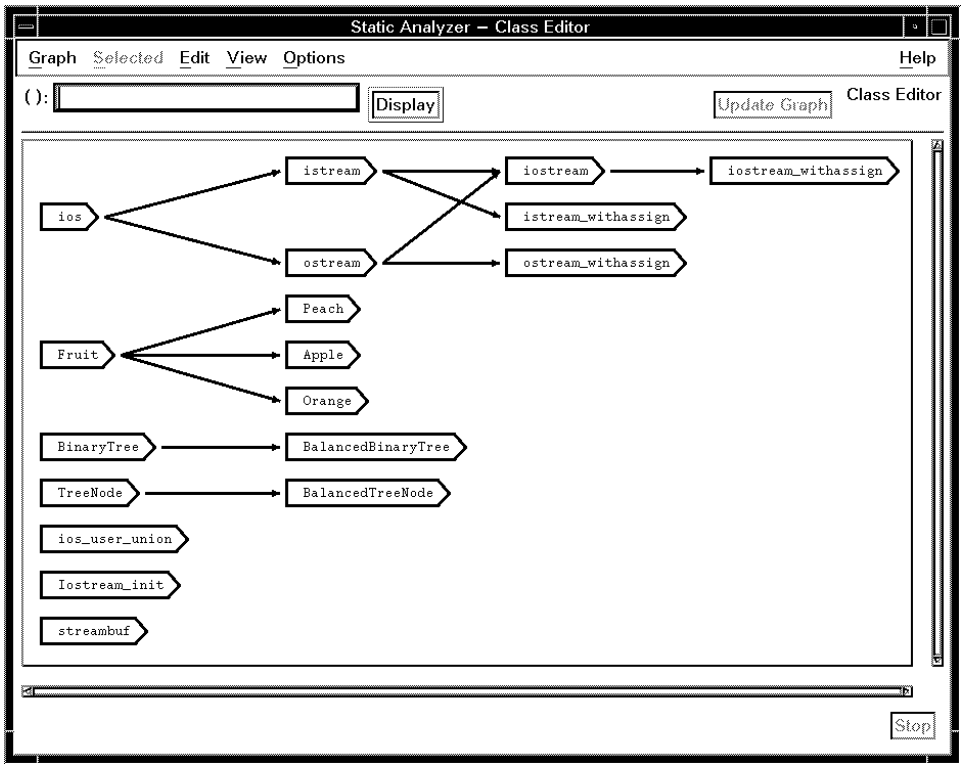


Fig. 1. *Class graph with all classes and inheritance relationships.*

Fig. 2 shows the same program that was represented in Fig. 1, but this time the visual display has been changed by filtering out all the classes from library header files. Additionally, two of the nodes have been expanded to show the member functions.

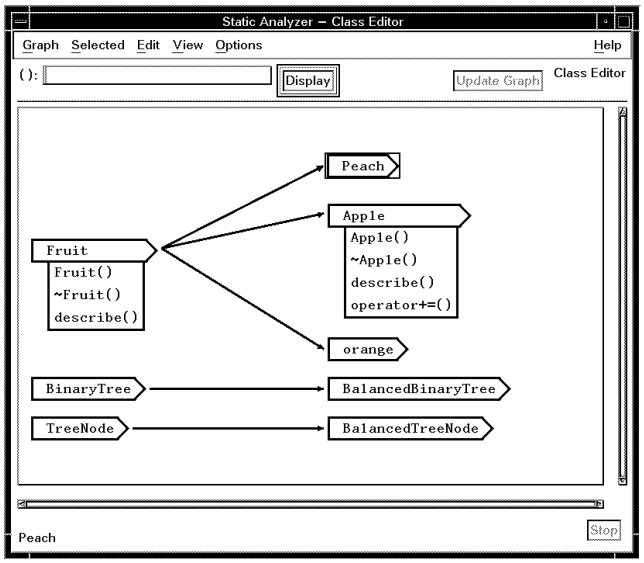


Fig. 2. *Simplified graph with only classes immediately under the programmer's control displayed and two nodes expanded to show the member functions.*

Changing the Class Hierarchy

Like any editor, the class editor allows the programmer to add, modify, and delete edited objects. For example, the programmer can add classes, inheritance relationships, member functions, and data members. Once these C++ structures exist, they can be modified or deleted. For example, the programmer can change an inheritance relationship from public to private or delete the relationship entirely.

If the programmer finds it necessary to restructure relationships by removing a class in the middle of an inheritance structure, the class editor makes the necessary edits to maintain the remaining relationships, as shown in Fig. 3. In this example, A is the base class of B, and B is the base class of C and D. Because the program architecture has been changed, the programmer no longer wants the B class. When B is deleted, the class editor automatically maintains the inheritance relationships so that A becomes the base class of C and D.

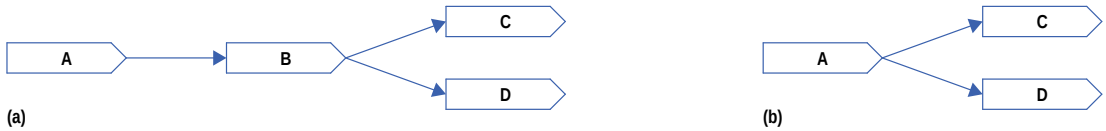


Fig. 3. *If the programmer removes a class in the middle of an inheritance structure, the class editor makes the necessary edits to maintain the remaining relationships. (a) Before the B class is deleted. (b) After the B class is deleted.*

Recovering from Editing Mistakes

The class editor remembers edit requests so that the programmer can undo them in reverse order. For example, if the programmer adds a base class relationship and then reconsiders, the Undo menu command on the Edit menu reads Undo Adding Inheritance.

Keeping the Static Database Up-to-Date

In SoftBench, compilations that produce static information are implemented with two parallel, independent build processes (see Fig. 4). The standard compiler, a cfront-based compiler, produces the error log and object (.o) files. The –y compiler option triggers the sbparse command, which is a subset of HP's ANSI C++ compiler. The sbparse command produces the static database, Static.sadb.

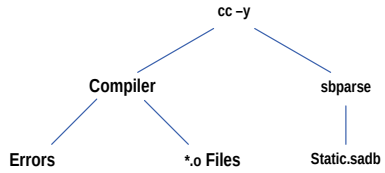


Fig. 4. *In SoftBench, compilations that produce static information are implemented with two parallel, independent build processes. The standard compiler produces the error log and object (.o) files. The sbparse command produces the static database, Static.sadb.*

The –nocode compiler option tells SoftBench not to run the cfront-based compiler. Since everything that the static analyzer knows depends on the underlying static database, each class editor edit request needs to update the static database. When the programmer requests an edit in the class editor, the class editor executes a compile with the –nocode –y compiler options, updating the static database without checking syntax and without producing .o files.

Using the Class Editor with a SoftBench Text Editor

The class editor saves after every logical edit. For example, if the programmer creates a new class, the underlying source code file changes when the programmer makes the request, and the class editor sends a FILE-MODIFIED message to let other tools know that the file changed.

If the programmer has a SoftBench text editor open while working in the class editor, the FILE-MODIFIED message causes the text editor to refresh the display of the file and the programmer can see the immediate propagation of the new source code.

Fig. 5 shows the sequence of events that occurs when the programmer makes an edit using the class editor:

1. The class editor performs pre-edit checks to make sure that the edit makes sense. Assuming that the request passes the pre-edit checks, the edit is displayed on the graph.

2. The class editor updates the underlying files that are impacted by the request.

3. The class editor sends a FILE-MODIFIED message to notify other tools that the edit took place.

4. The class editor executes a compile with –nocode –y
options, which updates the Static.sadb file.

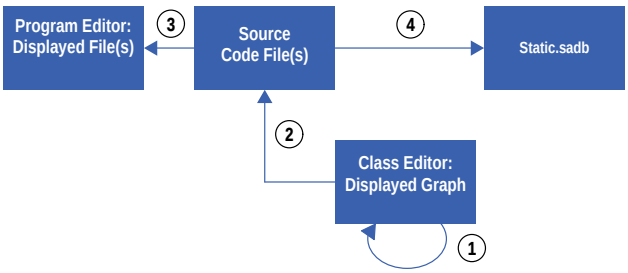


Fig. 5. *Sequence of completing a class editor edit. ① Edit displayed on graph. ② Files updated. ③ FILE-MODIFIED message results in redisplay of file in editor. ④ A compile with –nocode –y options updates the database.*

If the programmer chooses to make edits in the text editor, the sequence of events is slightly different (see Fig. 6):

1. When the programmer saves the file, the text editor updates the underlying file and sends a FILE-MODIFIED message.
2. The class editor receives the FILE-MODIFIED message and posts an information dialog box stating Undo disabled due to external edit. The class editor then erases the undo stack, since the external edits may have made the undo actions invalid.
3. The code changes in the text editor are not immediately propagated back into the class editor. The programmer must initiate the action that updates the static database and the graphical display. To update the static database, the programmer chooses the File: Analyze File Set menu command on the main static analyzer window. This menu command executes a –nocode –y compile.
4. After updating the static database, the programmer needs to select the Update Graph button in the class editor to display the code changes made in the text editor.

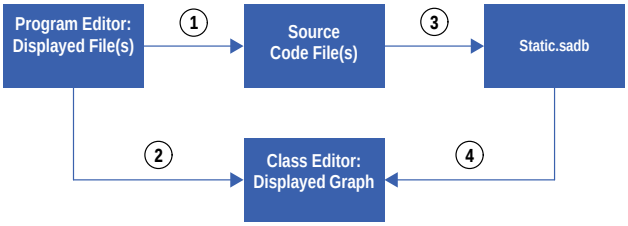


Fig. 6. *Sequence of updating the class editor after an external edit. ① Files updated. ② FILE-MODIFIED message disables the undo stack. ③ An Analyze File Set menu command triggers a compile that updates the database. ④ An Update Graph command displays the external edits on the graph.*

Working with Configuration Management

Edits in the class editor have the potential to change many files. For example, if the programmer changes the name of a class, several files may need to change. With the powerful, static-assisted editing, the programmer may not be aware of which files are changing. Consequently, the programmer can attempt to initiate edits on files that do not have write permission.

When the class editor runs into a problem with file permissions, it posts a dialog box giving the programmer three choices:

- Let the class editor check out the necessary files. This option is only valid if the files are under configuration management and available for checkout. The class editor completes the checkout process by sending a VERSION-CHECK-OUT message.
- Resolve the problem manually, then select Retry on the dialog box.
- Cancel the edit.

Fixing Compile Errors

The class editor does not introduce compile errors when it creates code. However, it is possible for the programmer to introduce compile errors. For example, the programmer might reference a function before creating it, make a typing error on a variable name or type when adding a data member, or make a syntax error in the body of a member function. Neither the class editor nor sbparse catches syntax errors of this type.

At first this model may appear surprising, but it actually works to the user's advantage. When the programmer uses a traditional text editor, code is not always compilable as it is being developed. The programmer may frequently create code objects out of order, mentally keeping track of what still needs to be done. The class editor functions in much the same way. If it detected every compilation problem, work would soon grind to a halt. Instead, the programmer can complete the code development tasks and let the compiler catch the syntax errors later.

Preserving White Space and Comments when Editing

The algorithm for completing an edit allows the class editor to preserve spaces, tabs, and comments in the code being edited. When the programmer specifies an edit, the static database provides the class editor with the specific positions in the source that need to be edited. The source code is then searched for "landmarks" to ensure that the right part of the code is being changed. Only minimal additions, substitutions, and deletions are done to the source file. For example, when a class is renamed, each reference is replaced by the new name, leaving any user-added comments or white space intact.

When more complicated things are changed, like the return type of a function, several consecutive tokens may be replaced with new text. In this case, any comments that are between the old tokens for that type are lost.

Troubleshooting

The error Unable to update the database is fairly common. It tends to occur with existing code that has compile errors, and it usually indicates a missing include file. To avoid this error, the programmer should make sure that existing code compiles without errors before starting to use the class editor.

Much more rarely, timing problems are encountered. When the programmer requests an edit, the first step is to make the edit visible on the graph, and the last step is to update the database (see previous discussion under "Using the Class Editor with a SoftBench Text Editor"). Because the class editor allows the programmer to begin the next edit as soon as the previous edit is visible on the graph, it is possible to experience a race condition. If the database is not yet up-to-date when the class editor attempts to complete its pre-edit checks for the next edit, the programmer will get an error message. For example, if the programmer creates a class, then attempts to add a member to the class before the create class edit is complete, the error Class <class name> not found will be issued. To resolve this error, the programmer should wait a moment and try again.

Conclusion

The static analyzer and the class editor together offer the C++ programmer a powerful program visualization and editing tool. The editing capabilities of the class editor facilitate program construction and editing. The code generation capabilities of the class editor facilitate program correctness and consistency. Code generated by the class editor is syntactically correct and consistently formatted. When the programmer makes a mistake using the the class editor, one or more edits can easily be backed out using the Edit: Undo menu command.

The filtering capabilities of the static analyzer allow the programmer to control the complexity of what is displayed and to conceal irrelevant details easily. The visualization capabilities of the static analyzer aid program comprehension. The programmer can choose to investigate many types of relationships in the code, and can easily access the underlying source code when more detail is needed.

Acknowledgments

The author wishes to acknowledge Wade Satterfield, the R&D engineer who developed the class editor, for his technical input and review of this article. The author also wishes to thank Carolyn Beiser, Jack Walicki, and Jerry Boortz for reviewing this paper and providing helpful suggestions.

Reference

1. F. Wales, "Theme 4 Discussion Report," *User-Centered Requirements for Software Engineering Environments*, Springer-Verlag, Nato Scientific Affairs Division, 1994, pp. 335-341. This article presents tasks to be facilitated. The tasks mentioned in the conclusion above are based on this task list.

The SoftBench Static Analysis Database

The static analysis database supports the SoftBench static analyzer and the C++, C, FORTRAN, Pascal, and Ada programming languages. The underlying data is isolated by a compiler interface and a tool interface.

by Robert C. Bethke

The SoftBench static analysis database, Static.sadb, is a repository for generic program semantic information. Within SoftBench the database supports the static analyzer along with graphical editing and rule-based program checking. The data model is relatively general and currently supports C++, C, FORTRAN, Pascal, and Ada.

The database also serves as a product and can be customized by the user. Its compiler interface and tool interface are documented and allow the integration of other languages and compilers and the use of custom analysis tools.

The Data Model

The underlying data is a set of persistent C++ objects. These objects serve to model the semantics of the program. The underlying persistent objects are isolated by the compiler interface and the tool interface. The isolation has important implications for allowing a variety of compiler integrations and provides flexibility in changing the underlying data management without affecting either the compilers or the tools.

Many of the persistent objects are language-generic (language-insensitive) and are intended to model all similar constructs. For example, a Struct object is used to model C structures and Pascal records. A Function object is used to model functions and procedures in all languages. In some cases, it is necessary to have language-specific objects because the semantics are too specific to apply to other languages. Examples of language-specific objects are C++ Class objects and Ada Module objects.

Each persistent object is assigned a unique object identifier known as a *handle*. Given an object's handle, it is possible to query the object by means of *methods* for relevant information such as its name, list of references, and so on. All associations among the persistent objects are maintained by these handles. For example, the association from a Variable object to its typedef object is maintained by the Variable's having the handle of its typedef as an attribute. One-to-many associations are maintained as a set of handles. For example, a File object will have a set of handles to associate all other source files included by it.

To illustrate associations, consider the following C code:

```
typedef struct S {int x; int y;} SType;
Stype var;
```

The associations among the semantic objects in this code fragment are shown in Fig. 1.

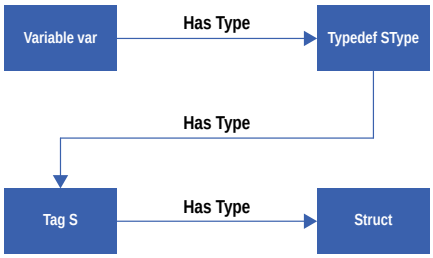


Fig. 1. *Associations among semantic objects for the C code example given in the text.*

Container objects are used to model scoping and binding and to organize the semantic objects for efficient updating and navigation. Each container has a set of handles for all objects contained in it and each object contained has the handle of its container. Examples of container objects are Files, Functions, and Classes. A File contains the program constructs defined in that source file, a Function contains its parameters and blocks, and a Class contains its members. For example, Fig. 2 shows the object containment for the following C++ class definition:

```
Class cls {
   public:
      cls (int x) {mem=x;}
   private:
      int mem;
};
```

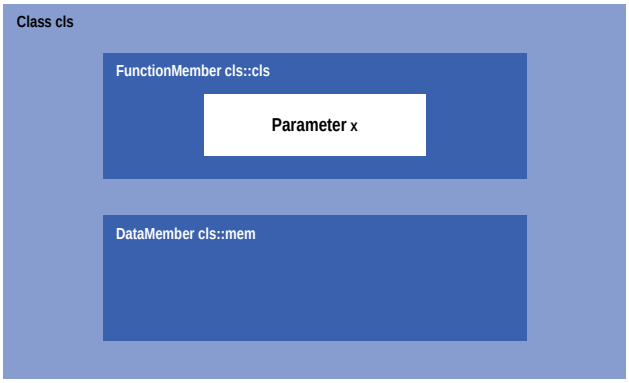


Fig. 2. *Object containment for an example C++ class definition.*

The Semantic Objects

The following is a partial list of the semantic objects stored in the database.

SymbolTable. The global SymbolTable is a container that serves as the root of navigation in the database. Its entries are all globally scoped semantic objects and Files in the database. There is only one global SymbolTable per database.

File. A File is a container that contains all semantic objects that are defined in a specific source file. Attributes of a File are its name, a language kind, and a set of *include* and *included by* associations with other Files.

Module. A Module is a container that contains all semantic objects that are defined within an Ada module. A Module must be contained within a File or within another Module. Attributes of a Module are its name and a set of imported associations with other Modules.

RefList. A RefList is an array of references that are associated with named objects in the database. Attributes of a RefList are the corresponding referent (the File in which the references originate) and the number of references in the list.

Macro. A Macro is a language-generic object for representing a preprocessor or language macro. Attributes of a Macro are its name and a set of RefLists.

Identifier. An Identifier is a language-generic object for representing a named symbol. This object is mostly used by weaker (scan-based) parsers that do not intend to distinguish certain categories of objects. Attributes of an Identifier are its name and a set of RefLists.

Label. A Label is a language-generic object for representing statement labels. Attributes of a Label are its name, an enclosing Block or Module, and a set of RefLists.

Variable. A Variable is a language-generic object for representing variables. Attributes of a Variable are its name and type, an enclosing Block or File, and a set of RefLists.

Function. A Function is a language-generic object for representing functions and procedures. Attributes of a Function are its name, a return type, a set of Parameters, an outer Block, a container (the enclosing File, Module, or Block), and a set of RefLists.

Parameter. A Parameter is a language-generic object for representing function parameters. Attributes of a Parameter are its name and type, an enclosing Function, and a set of RefLists.

Block. A Block is a container for representing a function block. Attributes of a Block are its begin and end line numbers, the File in which it is contained, and an enclosing Block or Function.

Typedef. A Typedef is a language-generic object for representing named program types. Attributes of a Typedef are its name, the type it denotes, an enclosing File or Block, and a set of RefLists.

Tag. A Tag is a language-generic object for representing aggregate (Enum, Struct, Class, and ClassTemplate) type names. Attributes of a Tag are its name, the aggregate it denotes, an enclosing File or Block, and a set of RefLists.

Enum. An Enum is a language-generic object for representing enumerated types. Attributes of an Enum are its corresponding Tag and a set of EnumMembers. RefLists to the enumeration are on the corresponding Tag.

EnumMember. An EnumMember is a language-generic object for representing enumeration constants. Attributes of an EnumMember are its name, an enclosing Enum, an ordinal value, and a set of RefLists.

Struct. A Struct is a language-generic object for representing program structures, records, and unions. Attributes of a Struct are its corresponding Tag and a set of DataMembers. RefLists to the Struct are on the corresponding Tag.

DataMember. A DataMember is a language-generic object for representing fields of a structure, union, class, or record. Attributes of a DataMember are its name and type, an enclosing Struct or Class, and a set of RefLists.

Class. A Class is a C++-specific object for representing C++ classes. Attributes of a Class are its corresponding Tag, a set of DataMembers, a set of FunctionMembers, a set of base and derived Classes, a set of friend Classes and friend Functions, a set of nested Classes within, and the ClassTemplate of which it is an instance. RefLists to the Class are on the corresponding Tag.

FunctionMember. A FunctionMember is a C++-specific object for representing C++ class member functions. Attributes of a FunctionMember are its name, a return type, a set of Parameters, an enclosing Class, the File in which it is defined, an outer Block, and a set of RefLists.

ClassTemplate. A ClassTemplate is a C++-specific object for representing class templates. Attributes of a ClassTemplate are its corresponding Tag, a set of DataMembers, a set of FunctionMembers, a set of FunctionTemplate members, a set of TemplateArguments, a set of base and derived Classes and ClassTemplates, a set of friends, and a set of Class instances. RefLists to the ClassTemplate are on the corresponding Tag.

FunctionTemplate. A FunctionTemplate is a C++-specific object for representing function templates. Attributes of a FunctionTemplate are its name, a set of TemplateArguments, and a set of Function or FunctionMember instances.

The Compiler Interface

From the compiler perspective the database can be thought of as a persistent symbol table for a set of source files such as a library or an application. The compiler sees the contents of only one compilation unit and emits information accordingly, but the database creates only objects that are not yet in the database. The database creates and merges all the program objects as the source files are compiled.

Compilation may result in objects being removed. Persistent objects are removed when they are old or are contained in objects that are old. For example, when a file has been modified and is being recompiled, the File is old and its contents are removed from the database. The compilation will proceed and instantiate the appropriate new objects contained in the File.

The database is incremental to the file level. If one source file in an application or library is changed, the compilation will result in the removal and repopulation of objects in that File. After the compilation the database is again consistent and available for queries from a reader.

The compiler interface is procedural in style and is intended to be easily added to most compilers. The interface is structured around the creation of objects and the establishment of associations and containment relationships among the objects.

The Tool Interface

From the tool perspective the database supports concurrency control to the extent of allowing multiple readers and one writer. A reader can have up to 256 databases open for reading. The reader must structure queries within a transaction and is allowed to leave the database open while it is being modified by a writer. The reader is notified of a change to the database via a callback when starting a transaction. Nested transactions are not supported.

The tool interface is a class library that reflects the underlying object model. Each persistent object is presented as a handle. Internally, each handle is mapped into a pointer to the real persistent object. All information pertaining to the object is made available via methods. Navigation among objects is supported by methods that return a handle or an iterator over a set of handles. For example, the following is a partial definition of the Symbol class.

```
class Symbol {
  public:
    Symbol(PerHandle symbolhandle);
    Symbol();
    ~Symbol();

    // Name, kind and attributes of the symbol.
    char *Name() const;
    PerKind Kind() const;
    Attributes Attrib() const;
```

```
    // Enclosing scopes of the symbol.
    DBboolean EnclosingFile(File &file) const;
    DBboolean EnclosingBlock(Block &block)
      const;

    // Iterator to all reference lists for this
    // Symbol.
    ITERATOR(RefList) RefLists() const;

  protected:
    PerHandle SymbolHandle;
};
```

The global SymbolTable is the root for all navigation. This object provides navigation and hashed searching to all globally scoped symbols. The following code segment illustrates how to access all globally scoped functions from the global SymbolTable.

```
SymbolTable symtab;
// Construct an iterator over all global
// functions.
ITERATOR(Function) functionitr =
   symtab.GlobalFunctions();
// For each function print its name and if the
// function is defined, the file in which it is
// defined.
ITERATE_BEGIN(functionitr) {
    File sourcefile;
    printf("%s", functionitr.Name());
    if (functionitr.EnclosingFile(sourcefile))
        printf(" contained in %s",
              sourcefile.Name());
    printf("\n");
} ITERATE_END(functionitr)
```

All of the relationships among the semantic objects are first-level. Hence, many of the interesting queries and rules will require a transitive closure* of the relationships. For example, consider the following function, which prints all the derived classes of a given class.

```
void derivedclasses(Class theclass) {
    // Iterate over immediate derived classes of
    // theclass.
    ATTRIBUTE_ITERATOR(Tag) tagaitr =
              cls.DerivedClasses();
    ITERATE_BEGIN(tagaitr) {
       // Print the class name.
       printf("%s\n", tagaitr.Name());
       Class dercls;
       // Navigate to the actual derived class
       // and recursively call derivedclasses to
       // print its derived classes.
       if (tagaitr.ClassType(dercls))
           derivedclasses(dercls);
    } ITERATE_END(tagaitr)
}
```

API Products

The database APIs (application programming interfaces) are available in the SoftBench 4.0 product and are used internally by the SoftBench parsers and tools. They are also used by some customers for compiler integrations. The tool interface is the fundamental component of the software developer's toolkit for user-defined rules.

* The transitive closure for a particular object under a particular transitive binary relationship is the set of objects descended from the particular object by way of the particular relationship. For example, if B is derived from A and C is derived from B, the transitive closure for the object A under the relationship "derived from" is the set of objects whose elements are B and C.

CodeAdvisor: Rule-Based C++ Defect Detection Using a Static Database

C++ SoftBench CodeAdvisor is an automated error detection tool for the C++ language. It uses detailed semantic information available in the SoftBench static database to detect high-level problems not typically found by compilers. This paper describes CodeAdvisor and identifies the advantages of static over run-time error checking.

by Timothy J. Duesing and John R. Diamant

C++ is a powerful successor to the C language that has all of C's features plus a lot more, including constructors, destructors, function overloading, references, inlines, and others. With this added power come more options to manage, more ways to do things right, and inevitably, more ways to go wrong. C++ compilers can find syntactical errors, but they do not find errors involving constructs that are legal yet unlikely to be what the programmer intended. Often, problems of this nature are left to be found during testing or by the end user. Attempts to find these defects at an earlier and less expensive stage of development sometimes take the form of code inspections or walkthroughs. While careful walkthroughs can find some of these errors, formal inspections are time-consuming and so expensive that they are usually only applied to small pieces of the code.

Since C++'s introduction in the early 1980s, a large body of experience with the language has accumulated and many works have appeared that describe common pitfalls in the language and how to avoid them.[1-5] While some of these problems can be quite subtle, some of them are also straightforward enough that a program can be created to detect them automatically,[6] as long as that program can be supplied with sufficiently detailed information about the code's structure. The SoftBench static database (see ***Article 3***), with its semantic information, provides an opportunity to create a tool that can do just that. This article is about such a tool: C++ SoftBench CodeAdvisor.

CodeAdvisor: An Automated Rule Checker

CodeAdvisor distills its knowledge of what are likely to be coding errors as a set of rules that alert the user to problems such as calling virtual functions from constructors, mixing iostream routines with stdio routines, local variables hiding data members, and so on. Each rule is a set of instructions that queries the static database for the information of interest and then performs the logic to test whether that potential error condition is present. When it detects a rule violation, CodeAdvisor displays the violation's location (file, line number) in an error browser that lets the user navigate quickly and easily to the problem site and use an editor to correct it. Online help is available to present more explanation of the violation, possible ways to correct the problem, references for further information, and when appropriate, exceptions to the rule.

CodeAdvisor detects rule violations by performing static analysis of the code using the SoftBench static database. Static analysis differs from the dynamic or run-time analysis done by debuggers, branch analyzers, and some performance tools in that all of the available code is examined. Dynamic analysis examines only code that is actually executed and cannot find defects in branches that are never taken. Also, dynamic analysis requires that the code be far enough along so that it can be actually executed. Static analysis, on the other hand, can be performed as soon as the code compiles, even if the code cannot yet successfully run.

Because it is automated, CodeAdvisor will tirelessly check all the rules it knows against all of the code. This is practical only for relatively small pieces of code during inspections done by hand. Unlike a human code reviewer, CodeAdvisor never gets so tired or bored that it misses a rule violation it's been programmed to find. While CodeAdvisor cannot replace inspections completely (there will always be problems that cannot be detected automatically), it can be a good complement to traditional code inspections, freeing developers to focus on higher-level problems by weeding out the detectable problems first.

Example Rule: Members Hidden by Local Variables or Parameters

Let's look at an example of one of the rules CodeAdvisor implements and examine how it uses the static database to find a rule violation. Consider the small program in Fig. 1. The class Vehicle with its two-line member function SetSpeed looks simple enough. The constructor for Vehicle sets the initial speed to zero, so we would expect to get a current speed of zero at the start of the program and we do. We might also expect that, after calling SetSpeed with a delta of 50, we would then get a current speed of 50. However, if we actually compile and run the program we find that we still get zero! Why? The problem is that a data member is hidden by a function parameter with the same name. In SetSpeed we've made an unlucky choice when

we named the parameter speed, since there is a data member of the same name in the class Vehicle. When speed is modified in SetSpeed, the compiler modifies the parameter rather than the data member. The compiler will not complain since we have given it unambiguous instructions, which it will follow perfectly. If we had chosen any other name for our local variable, the example would work as expected.

```
#include <iostream.h>

class Vehicle {
private:
     int speed;
public:
     int CurrentSpeed() const { return speed; }
     void SetSpeed(int newspeed, int delta = 0);
     Vehicle() { speed = 0; }
};

// SetSpeed takes an absolute speed plus a
// delta. If absolute speed is zero, use
// current speed. Other parameters should be 0
// (2nd one defaults to 0)
void Vehicle::SetSpeed(int speed, int delta)
{
     if (!speed) speed = CurrentSpeed();
     speed = speed + delta;
};

main()
{
     Vehicle car;
     cout << "Car's initial speed = "
          << car.CurrentSpeed()
          << endl;
     car.SetSpeed(0,50);
     cout << "Car's new speed = "
          << car.CurrentSpeed()
          << endl;
}
```

Fig. 1. *An example of a CodeAdvisor rule violation: members hidden by local variables or parameters.*

Even in this simple setting, an error like this can be difficult to spot at a glance. In a more complex and perhaps more realistic situation, this problem might never be found in a code inspection. If we bury a few subtle defects like this in a few megabytes of code we might find that they won't be found until actual execution exposes them as bugs.

Detecting an Error Using the Static Database

The problem, then, is how to find these kinds of defects before the user does. The context in which speed is used is what's important here. Using speed as a parameter in most cases is perfectly valid. The only case we need to worry about is when a parameter or local variable is used within the scope of a member function and it has the same name as a data member of that class. This is where the static database is needed to make this kind of rule checking possible. The static database contains, among many other things, information about what objects are global and local within a scope, and it understands what objects are member functions and what the associated parameter list is.

One way to create a rule to detect this particular error is to first query the database to find all the classes in a program. Once we have all the classes, we can query the database for all the member functions of those classes. Then we can examine each function's parameters and local variables looking for any members local to the class or inherited public or protected with the same name. If we find a match, we report a rule violation and output the file and line numbers of the offending symbols.

Of course, to make the rule robust, there are still a few little details that need to be considered in implementing the above algorithm. For instance, to be general, when we query the database for classes, we'll want to find class templates as well, and if we find any, we'll want to consider only the templates themselves and not their instances. Also, when we search for member functions of these classes we'll want to skip any compiler-generated functions that the C++ compiler may have created by default. We may also want to handle the cases where a symbol hides a member function as well as a data member. All the information needed to handle these details is available in the static database.

Exceptions to the Rule

The types of problems for which CodeAdvisor is targeted are not the obvious or even the obscure abuses of the C++ language. Compilers are fully capable of finding these types of errors. Rather, CodeAdvisor attempts to identify a more subtle kind of problem that might be characterized as constructs that experience tells us are almost certainly not what the programmer intended, even though they are fully legal within the language. We must include the word "almost," however, because occasionally some of the most unlikely constructs are in fact what the programmer intended. Deciding with certainty whether or not a suspicious construct will turn out to be a real problem may sometimes require knowledge that cannot be determined by a practical amount (or sometimes any amount!) of analysis, static or run-time.

To illustrate this, consider, for example, the CodeAdvisor rule that detects classes that are passed as a value parameter to a function. This may become a problem when the class passed is a derived class and virtual functions of that class are called within that function. This is because calls to that class's virtual functions will call the base class's versions, not the derived class's versions. The above conditions are easy enough to check for with the static database, but they alone do not guarantee an error condition. If the function is never passed a derived class instance, no problem will occur. In some special cases, static analysis might be able to detect this additional condition but in other cases involving complex conditional branching, detection would be impractical or impossible. Run-time analysis also might be able to detect this condition in special cases, but in cases of less than 100% branch coverage or conditional branching determined by many combinations of possible external data, detection again would be impractical. In this particular example, CodeAdvisor will report the rule violation even with imperfect information because even when the problem only potentially exists, it can cause a serious problem for later code maintainers. Each rule, however, must be evaluated on its own merits to consider the possible nuisance of false positives.

In this sense, the rules can be regarded as heuristic—that is, good but not perfect guesses that a given piece of code is a genuine error. Fig. 2 illustrates the nature of the problem when a rule has imperfect knowledge of the code. The area where a heuristic rule is satisfied still contains cases where no real error exists. To report these cases when there is a reasonable amount of uncertainty as to their validity would be to bombard the user with unwanted "noise" that would distract from other real problems.

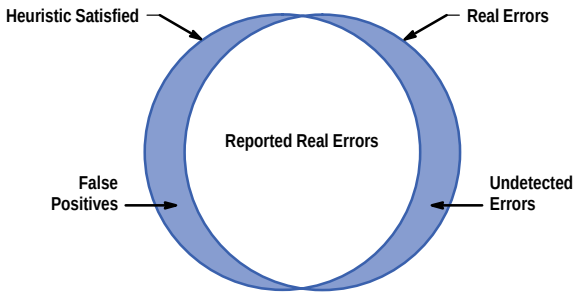


Fig. 2. *The problem of finding errors with imperfect information.*

We have reduced the noise factor in CodeAdvisor by adopting a philosophy of "no false positives" when implementing a rule. That is, when imperfect information prevents knowing with certainty if a construct causes a problem in the current setting, the code is given the benefit of the doubt unless there is also a serious potential for a future maintenance problem. In addition, for those occasional cases where a suspicious construct is reported but still deemed acceptable by the user, CodeAdvisor provides a filtering mechanism to allow the user to suppress the display of particular violations.

Summary

CodeAdvisor uses the information available in the SoftBench static database to implement a deeper level of error detection than is available with current compilers. CodeAdvisor's static analysis has advantages over run-time analysis because all of the available code is analyzed instead of only the branches that are actually executed. An automated rule checking tool like CodeAdvisor can contribute to the code development process at an early stage, where the cost of defect repair is less expensive. CodeAdvisor complements traditional code inspection and testing, allowing developers to focus on the higher-level problems by weeding out the detectable problems first.

References

1. S. Myers, *Effective C++*, Addison-Wesley, 1992.
2. T. Cargill, *C++ Programming Style*, Addison-Wesley, 1992.
3. M.A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.
4. Taligent Corp., *Well-Mannered Object-Oriented Design in C++*, Addison-Wesley, 1994.

5. *Programming in C++, Rules and Recommendations,* Translated from Swedish by Joseph Supanich, Ellemtel Telecommunication Systems Laboratories, 1990-1992.
6. S. Meyers and M. Lejter, "Automatic Detection of C++ Programming Errors: Initial Thoughts on a lint++," *Proceedings of the Usenix Association C++ Conference,* 1991.

Using SoftBench to Integrate Heterogeneous Software Development Environments

Migrating from mainframe-based computing to client/server-based computing can result in a heterogeneous collection of machines that do not interoperate, forcing software developers to deal with unfamiliar system commands and systems that cannot share data. A SoftBench control daemon is described that enables developers to integrate heterogeneous computing systems into efficient, tightly coupled software development environments with consistent, easy-to-use graphical user interfaces across all machines.

by Stephen A. Williams

Many companies today are migrating from mainframe-based computing environments to client/server-based technologies using various workstations and PCs. They are attracted to the client/server architecture because of industry claims of benefits like increased efficiency, lower operating costs, and less reliance on a particular vendor.

Often, however, the result is a heterogeneous collection of machines that do not interoperate well. Because the operating systems on the disparate machines all come with their own sets of tools, software developers must learn a new set of commands for each system that they use. In addition, developers must deal with the inconsistencies that arise when applications available on one system are not available on another and when data cannot be shared between machines because the different toolsets cannot communicate.

To solve these problems, the advanced system development and integration division of Science Applications International Corporation (SAIC) uses Hewlett-Packard's SoftBench product to integrate its customers' diverse systems into efficient, tightly coupled software development environments with consistent, easy-to-use graphical user interfaces. This article discusses why and how SAIC uses SoftBench to solve its customers' multiplatform software development problems. The article details some of the common pitfalls encountered when developing software in an open systems environment, explains how SAIC deploys SoftBench to integrate such systems, and concludes by discussing the benefits of such an integration.

Open Systems

Companies are adopting client/server-based open systems for a wide variety of reasons. Some companies hope to increase computing efficiency by distributing the data and processing load, thereby providing faster response times and quicker access to system resources. Other companies want to lower their development costs by using lower-cost, yet faster workstations and PCs. Yet others must move to open systems to remain compatible with their customers and keep a competitive edge in the marketplace.

While migrations to open systems can provide great dividends, they can also become more unwieldy than the systems they replace. Many client/server topologies contain a wide variety of machines, such as high-end servers running the UNIX® operating system, PCs running Microsoft® Windows, and legacy systems running proprietary operating systems. In addition, even similar machines will often run different operating systems (e.g., variations of the UNIX operating system) or even different versions of the same operating system. The resulting heterogeneous collection of machines makes it difficult to create an efficient and cooperative software development environment. Fig. 1 depicts an example of such an environment. Note that most applications cannot communicate with each other.

Although many open system standards exist to help such diverse collections of machines communicate, most of them are low-level network standards that simply provide a way for bits to be transferred between machines. The open systems community still lacks accepted high-level application standards that would allow disparate programs to interact with each other. Thus, applications from different vendors often cannot interoperate, which greatly restricts the benefits of implementing many client/server solutions. This lack of communication also means that data must be replicated across machines, wasting resources and increasing the risk of data inconsistency.

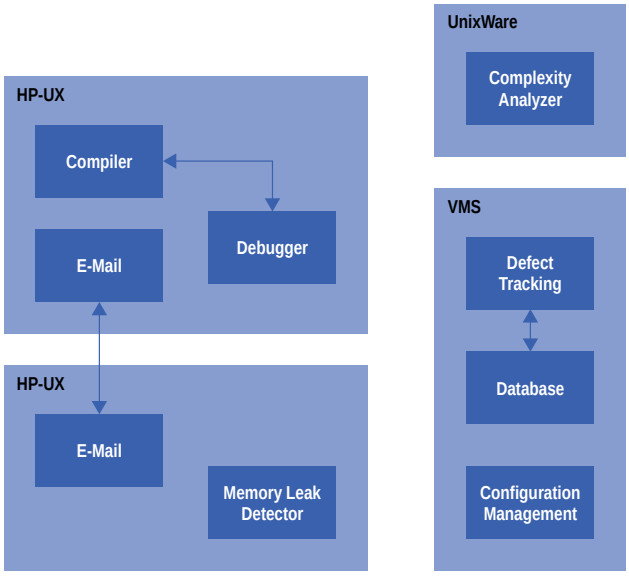


Fig. 1. *An example of a heterogeneous collection of machines in which the applications on different systems cannot cooperate or communicate with each other.*

Another problem with developing in a multiplatform, multivendor environment is the lack of consistency in the user interface to the systems. Because developers must deal with multiple operating systems, they have to learn how to operate each system's interface individually. This can be an especially formidable task considering the arcane commands used by some operating systems and the differences between file systems across platforms (i.e., hierarchical versus fixed depth, / versus \, etc.). Developers must also remember which system contains each application that they need, where it is located, and how to start it.

Furthermore, developers in a heterogeneous environment must learn how to operate the different user interfaces for each of the applications that they use. While some applications now have elegant graphical user interfaces, the look and feel of each system are often different. Also, many applications do not have graphical user interfaces at all, which requires that developers memorize command-line options to these programs. These inconsistencies not only lengthen a developer's learning curve, but also make developers less efficient when switching between applications.

Integration

Previous issues of the HP Journal have described how to use SoftBench to integrate disparate applications running under the HP-UX* and Solaris operating systems.[1,2,3] In this article we will concentrate on how to use SoftBench to integrate applications running on other platforms. The key to accomplishing this integration is to port SoftBench's subprocess control daemon (SPCD) to each operating system that is to be integrated. The SPCD provides a standard, robust, and secure method of executing subprocesses on remote systems. This is accomplished by providing an API through which encapsulations can interact with the SPCD over a network socket connection. Through the API, encapsulations can instruct the SPCD to start or stop a subprocess on the remote machine, send input to a subprocess, and receive output from a subprocess.

Thus, once the SPCD is ported to a given system, encapsulations† can be written for applications running on that system as easily as if the applications were running on an HP-UX or Solaris system running SoftBench.

Why SoftBench? There are several reasons why SAIC chose to use SoftBench to integrate heterogeneous software development environments. First, the standard SoftBench environment comes with a rich set of state-of-the-art software development tools, all of which use a consistent, easy-to-use graphical user interface. In addition, SoftBench provides a graphical user interface to the operating system (via the SoftBench development manager) which hides many of the intricacies of the operating system and its file system.

Another advantage to using SoftBench is the framework for interapplication communication it provides through SoftBench's broadcast message server. This framework allows applications with no direct knowledge of each other to communicate and therefore interoperate. This functionality allows one application to be substituted for another with no adverse effects on other applications. It also allows new applications to be integrated into the environment without making any changes to existing applications.

Probably the most important reason to use SoftBench is its extensibility. Through the use of the encapsulator library, which provides functions to communicate with the SPCDs, the SoftBench environment can be extended to include non-SoftBench applications. In addition, the encapsulated applications can run on any operating system to which the SPCD has been ported.

† A SoftBench encapsulation means integrating a tool into the HP tool integration architecture.

Using SoftBench for Integration. Given the above reasons for using SoftBench to integrate a heterogeneous software development environment, how does one go about implementing such an integration? The first step is to install SoftBench on at least one HP or Sun workstation. Note that it is not necessary to place such a workstation on each developer's desk because SoftBench can be run remotely using the X Window System and developers can use any machine running an X server. This includes DOS, Windows, MacOS, and most versions of the UNIX operating system. Thus, a company implementing a SoftBench environment can probably leverage much of its existing hardware inventory to keep costs down.

Next, the SPCD needs to be ported to each operating system in the environment that contains applications that need to be integrated. Of course, there's no need to port to HP-UX or Solaris since SoftBench (and thus the SPCD) already runs on those systems. As discussed earlier, the SPCD provides a standard method that SoftBench applications can use to execute subprocesses on remote systems. Although other methods of remote subprocess control could be used in such an integration, the SPCD is probably the best choice because it is specifically designed to work with SoftBench. Also, note that there is no need to port all of SoftBench since only the SPCD is needed for remote subprocess control.

Because the source code for the SPCD is not freely available, the SPCD can only be ported by Hewlett-Packard or its authorized agents. SAIC has been granted such authority in the past to complete SoftBench integrations for a number of its customers. The operating systems to which SAIC has already completed the SPCD ports include:

- UNIXWare
- MP-RAS
- VMS
- Pyramid DC/OSx
- Stratus FTX
- Windows NT
- Tandem Guardian
- Tandem OSS.

A port to MVS was started but not completed.

As can be seen from the diversity of the operating systems to which the SPCD has already been ported, the SPCD code is quite portable. However, there are a number of requirements that the SPCD makes of a target operating system. The list below details the basic requirements that SAIC uses to determine the level of effort in an SPCD port:

- An ANSI C compiler
- A C++ compiler
- A Berkeley-type TCP/IP sockets capability
- The capability for a process to start up and communicate with several subprocesses like the UNIX fork() and pipe() system calls
- The capability for a process to detect input from several sources at the same time like the UNIX select() system call
- An interface that allows system calls to be made from C
- A way to set the environment of a controlled process like the UNIX getenv() system call
- Functionality similar to the UNIX inetd server
- Network File System (NFS) capability.

Note that the SPCD has been ported to environments that do not have fork(), select(), the inetd server, or NFS. While these items do make the port much simpler, it is still possible to port to environments that do not include all of the items listed above.

Once the SPCD has been ported to the appropriate operating systems, custom encapsulations must be written for each of the applications to be integrated into the SoftBench environment. Each encapsulation's job is to act as an intermediary between a non-SoftBench application and the SoftBench environment, making it look like the application is a fully integrated SoftBench tool (see Fig. 2). Performing this job entails a number of responsibilities, such as starting the application to be integrated, establishing a connection to the SoftBench environment, and sending the appropriate notification messages to SoftBench whenever the encapsulated application performs an action about which another tool might want to know. Furthermore, the encapsulation must listen for messages requesting a service of the encapsulated application and then instruct the application to perform the requested task.

To simplify the process of writing an encapsulation, SoftBench comes with an encapsulator library that provides an easy-to-use API to the SoftBench environment. The encapsulator library provides functions to:

- Send and receive SoftBench messages through the BMS
- Control remote subprocesses using the SPCD
- Create graphical user interfaces that are consistent with other SoftBench tools.

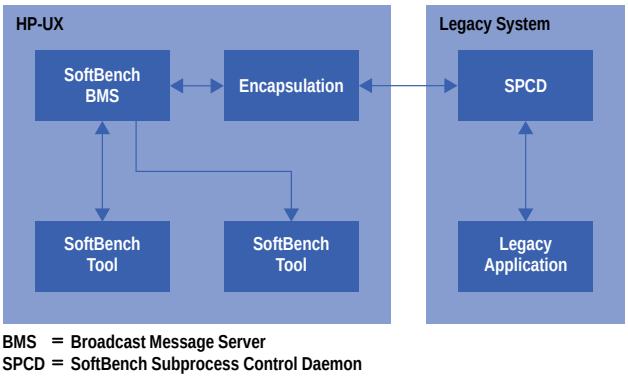


Fig. 2. *The organization of software components after SoftBench is set up. The SPCD has been ported to a legacy system, and an encapsulation has been written for each legacy application to be integrated in the SoftBench environment.*

Because the encapsulator library has only been ported to the HP-UX and Solaris operating systems, encapsulations that link with encapsulator routines must run on a machine using HP-UX or Solaris. While the encapsulator library could be ported to other operating systems, this is usually unnecessary since an encapsulation can use the SPCD to execute a subprocess on a remote host as easily as on a local host. This is one of the major advantages gained by porting the SPCD to all operating systems in the environment.

A few other limiting factors must be taken into account when writing encapsulations. First, it is difficult, if not impossible, to integrate applications that have no command-line interface. For example, if the only way to interact with an application is through a graphical user interface, then an encapsulation of that application must emulate mouse movements and button clicks to communicate with it. This is generally not a feasible option.

Another factor to consider when writing encapsulations is the granularity of the information provided by the application to be encapsulated. If the application does not give some sort of notification for each action that it takes, then the encapsulation will be limited in its interpretation of what the application is doing. For more information about the limitations of the encapsulator library see **reference 1**.

Once the necessary encapsulations have been written, the next step in integrating an application into a heterogeneous computing environment is to extend the SoftBench environment so that all of the desired applications are seamlessly integrated into it. This is accomplished by modifying the SoftBench configuration file `softinit` to include references to each of the new encapsulations (see Fig. 3). This action informs SoftBench about the new functionality that is now available through the encapsulations and how to access those encapsulations.

Modifications to `softinit` can also be used to inform SoftBench to replace existing tools with new encapsulations. For instance, the standard e-mail tool that comes with SoftBench could be replaced with an encapsulation of a local e-mail application. SAIC has used this capability to replace the debugger that comes with SoftBench with an encapsulation of the GNU debugger, `gdb`. This provides SAIC's customers with a fully integrated debugger that runs on any machine that `gdb` supports, which includes most modern operating systems.

To further integrate a development environment, the SoftBench message connector can be used to automate repetitive tasks and enforce software development processes. The message connector works by monitoring the BMS for a desired message and then executing a user-supplied routine whenever that message is seen. For example, suppose company policy requires that a complexity analysis program be run on all source code when it is checked into the configuration manager. To meet that requirement with no human intervention, the message connector could be configured to monitor the BMS for a message from the configuration management tool indicating that a file has just been checked in. Then, it would run the analysis program on that file, perhaps e-mailing the results back to the developer who checked in the file.

For software development processes that require more intricate interactions than the message connector can provide, SAIC's SynerVision product can be used. It provides a nextgeneration process management environment that helps teams manage the software engineering process, including such tasks as writing new software, debugging programs, maintaining existing systems, and porting to new platforms. Also, because SynerVision fully supports the SoftBench environment, no new encapsulations need to be written for it.

Note that the steps described above for integrating a heterogeneous software development environment with SoftBench do not need to be implemented all at once. Instead, the built-in extensibility of SoftBench allows one to take a progressive approach wherein applications are encapsulated one at a time and added to the environment as they are completed. Such an approach can smooth the migration path from a legacy system to an open system by eliminating the need for a complete switchover to the new technology.

```
#
# $HOME/.softinit –– user customizations to
# SoftBench initialization

#
# Editor

#
# To use ”vi” as the editor, uncomment the
# following line
EDIT   TOOL  NET   *  %Local% softvisrv –scope
net -types %Types%

# To use ”softedit” as the editor, uncomment the
#following line
#EDIT  TOOL  NET   * %Local% softeditsrv –scope
net –types %Types%

# To use ”emacs” as the editor, uncomment the
#following line
#EDIT TOOL NET     * %Local% emacs

#
# Configuration Management
#

DM   TOOL  DIR     * teflon softdm –host %Host%
–dir %Directory% –file %File%

# To use ”RCS” as the CM tool, uncomment the
# following line
CM   TOOL NET * teflon softrcs –scope net

# To use ”SCCS” as the CM tool, uncomment the
# following line
#CM  TOOL NET * spike softsccs

#
# Debugger
#
# To use GDB as the debugger, uncomment the
# following line
#DEBUG TOOL FILE * teflon /usr3/stevew/
#DebugBackends/softgdb/softgdb –d 255 –1 /tmp/
softgdb.log –host %Host% –dir %Directory% –file
%File%

#
# Tandem stuff
#

# To startup RSHELLSRV in debugging mode,
#  uncomment the following line
#RSHELLSRV TOOL HOST * teflon /usr/rshellsrv/
#rshellsrv –d 255
1 /tmp/log.rshellsrv.%Host%.$USER –host %Host%

# To startup RSHELLSRV in standard mode,
#uncomment the following line
RSHELLSRV TOOL HOST * teflon /usr/rshellsrv/
rshellsrv –host %Host%
```

Fig. 3. *A SoftBench configuration file softinit. This file contains references to each new encapsulation.*

Benefits of Integration with SoftBench

By extending SoftBench as described above, a heterogeneous collection of computing systems with disparate, incompatible tools can be transformed into an efficient, tightly coupled software development environment with consistent, easy-to-use graphical user interfaces across all machines. Fig. 4 shows the result of an example integration.

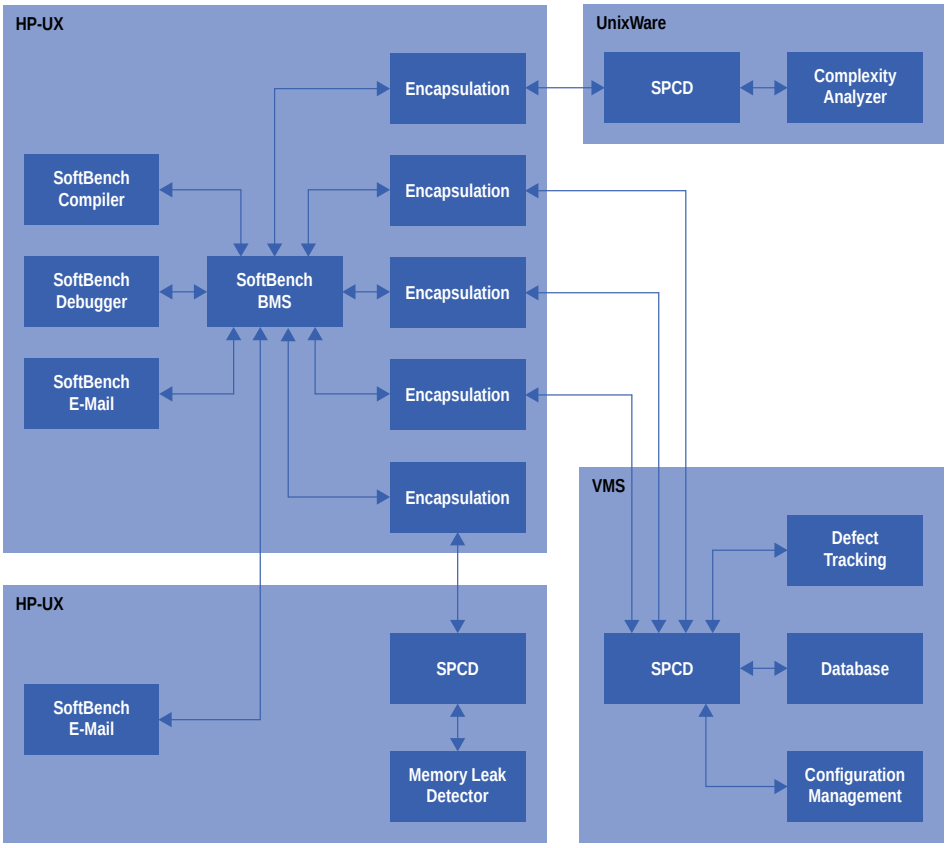


Fig. 4. *An example of an integration over several platforms. Note that there is an encapsulation for each application.*

Certainly, one of the biggest advantages of integrating with SoftBench is the realization of a standard, consistent user interface to all tools on all machines. This consistent interface minimizes the learning curve for developers by reducing the number of commands that they need to learn to use the environment. It also improves the efficiency of developers by simplifying their interactions with both applications and operating systems and by providing a means for data sharing between applications (e.g., cut and paste, drag and drop, etc.). In environments with legacy systems where developers have been using text-based terminals, the benefit of this graphical user interface can be enormous.

As discussed earlier, all SoftBench applications use the X Window System to display their graphical user interfaces. This provides the advantage that the complete software development environment is always available from any machine that has an X server. Furthermore, the environment looks and works exactly the same no matter what machine a developer uses, from a Macintosh PowerBook laptop running an X server to an HP 9000 workstation running HP VUE.

An environment integrated with SoftBench also provides the advantage that remote data access is transparent to the user. By using NFS and the automounter, SoftBench automatically retrieves data from remote machines without any user intervention. Developers only need to specify which machine contains the desired data, and SoftBench handles the rest. This benefits developers because they do not have to copy files back and forth between machines or know the intricacies of networked file systems.

Similarly, SoftBench provides the advantage that remote program execution is transparent to the user. By using SPCD, SoftBench can execute applications on remote machines without developer intervention. Developers no longer need to log into various machines to run the tools they need because SoftBench provides a centralized control center that places all tools at their fingertips. This lets the developer concentrate on the task at hand instead of worrying about logins, passwords, pathnames, and so on.

By providing transparent access to both data and applications, SoftBench allows resources to be spread across a distributed client/server topology without introducing complexity into its use. Developers get a unified view of their environment whether it contains one machine or one hundred, whether all their data is centralized on one server or distributed across

many systems. In addition, machines can be added to (or removed from) the environment without impacting developers simply by modifying SoftBench to use (or stop using) the given machines.

Another advantage of integrating with SoftBench is the rich set of state-of-the-art software development tools that come with SoftBench. These tools benefit developers by simplifying and expediting the edit-compile-debug cycle. The tools automate processes such as checking source files into and out of configuration management, building executables, and displaying errors found by the compiler. In addition, the tools can provide a graphical view of source code, allowing a developer to quickly learn unfamiliar code or find errors in program flow.

Furthermore, by encapsulating local and third-party applications in the environment, developers will have access to those applications as easily as if they were standard SoftBench tools. This benefits developers because they do not have to know on which host the applications exist or how to start them. Instead, the developer can start an application simply by selecting it from the list of applications in the SoftBench tool manager. In fact, by customizing the environment with the message connector, many applications can be started automatically.

As discussed earlier, the message connector and SynerVision can save developers time and effort by automating repetitive tasks and by enforcing software development policies such as ensuring that required tasks always occur and that those tasks are executed in the proper order. By enforcing well-defined policies, SoftBench can help increase the efficiency of the software development process and improve the quality of the finished product.

Conclusion

Software development in a heterogeneous computing environment can be a difficult proposition. Varying hardware and software platforms, incompatible tools, and inconsistent user interfaces are just a few of the trouble spots. However, Hewlett-Packard's SoftBench product can be used to solve these problems by providing a standard upon which to integrate the disparate components of such an environment. By porting SoftBench's SPCD to each operating system involved, all machines become equally and consistently accessible from SoftBench. Then, by encapsulating the applications on those systems, the applications become fully integrated SoftBench tools capable of interacting with other SoftBench tools.

Acknowledgements

SAIC's success in deploying SoftBench to solve its customers' problems is the result of a collaboration of exceptional talent at both SAIC and Hewlett-Packard. I would especially like to thank Vern Badham, Winn Rindfleisch, and Dave Romaine for their invaluable contributions to these projects and for their help in writing this article. I would also like to thank Curt Smith for inviting me to join his team at SAIC and John Dobyns for allowing me to continue my encapsulation work after I left SAIC. Lastly, I extend special thanks to Dick Demaine and the Software Engineering Systems Division of Hewlett-Packard for making SoftBench possible and for their continuing support of SAIC's work.

References

1. B.D. Fromme, "HP Encapsulator: Bridging the Generation Gap," *Hewlett-Packard Journal*, Vol. 41, no. 3, June 1990, pp. 59-68.
2. C. Gerety, "A New Generation of Software Development Tools," *Hewlett-Packard Journal*, Vol. 41, no. 3, June 1990, pp. 48-58.
3. J.J. Courant, "SoftBench Message Connector: Customizing Software Development Tool Interactions," *Hewlett-Packard Journal*, Vol. 45, no. 3, June 1994, pp. 34-39.

The Supply Chain Approach to Planning and Procurement Management

The supply chain approach models stochastic events influencing a manufacturing organization's shipment and inventory performance in the same way that a mechanical engineer models tolerance buildup in a new product design. The objectives are to minimize on-hand inventory and optimize supplier response times.

by Gregory A. Kruger

This paper describes the processes and equations behind a reengineering effort begun in 1995 in the planning and procurement organizations of the Hewlett-Packard Colorado Springs Division. The project was known as the *supply chain* project. Its objectives were to provide the planning and procurement organizations with a methodology for setting the best possible plans, procuring the appropriate amount of material to support those plans, and making up-front business decisions on the costs of inventory versus supplier response time (SRT),* service level to SRT objectives, future demand uncertainty, part lead times, and part delivery uncertainty. The statistical modeling assumptions, equations, and equation derivations are documented here.

Basic Situation

Consider a factory building some arbitrary product to meet anticipated customer demand. Since future demand is always an uncertainty, planning and procurement must wrestle with the task of setting plans at the right level and procuring the appropriate material. The organization strives to run the factory between two equally unattractive scenarios: not enough inventory and long SRTs, or excessive inventory but meeting SRT goals. In fact, more than one organization has found itself with the worst of both worlds—huge inventories and poor SRTs.

The supply chain project focused on characterizing the various stochastic events influencing a manufacturing organization's shipment and inventory performance, modeling them analogously to the way a mechanical engineer would model a tolerance buildup in a new product design.

Problem Formulation

For a particular product, a factory will incur some actual demand each week, that is, it will incur demand D_i in week i, for i = 1, 2, 3, ... From a planning and procurement perspective, the problem is that looking into the future the D_i are unknown.

Let P_i be the plan (or forecast) for week i in the future. Now for each week, the actual demand can be expressed as the planned demand plus some error: $D_i = P_i + e_i$.

The MRP (material requirements planning) system, running at intervals of R weeks, evaluates whether to order more material to cover anticipated demand, and if the decision is to order, how much to order. Given a lead time of L weeks to take delivery of an order placed to a supplier now for some part, the material in the supply pipeline must cover real demand for the next L + R weeks. By supply pipeline we mean the quantity of the part already on hand at the factory plus the quantity in orders placed to the supplier and due for delivery over the next L weeks.

For simplicity, assume for the remainder of this discussion that we are dealing with a part unique to one product and used only once in building the product. We will remove these constraints later but for now it will help to focus on the key concepts.

Define X to be the unknown but actual demand the factory will experience for this part over the next L + R weeks:

$$X = \sum_{i=1}^{L+R} D_i = \sum_{i=1}^{L+R} \left(P_i + e_i\right).$$

In statistical terminology, X is a random variable, that is, we cannot say with certainty the value it will take next, but with some assumptions about the nature of the planning errors (e_i), the distribution of X can be characterized. Specifically, we will make the assumption that the e_i are distributed according to the Gaussian (normal) distribution with mean zero and

* In standard terminology, SRT stands for "supplier response time." In this case, a better term would be "shipment response time," because the supplier being referred to is HP and not one of HP's suppliers. In this paper, we use the standard terminology for SRT, but the word "supplier" in all other contexts means one of HP's suppliers.

variance σ^2 (see Fig. 1). The assumption that the mean of the e_i is zero says that our plans are unbiased, that is, the factory is not consistently overestimating or underestimating future demand. Thus, the average of the differences between the plan and the actual demand over a reasonable period of time would be about zero. The normal distribution is symmetric, so we are saying there is equal probability in any week of actual demand being above or below plan. The variance measures how large the planning errors can get in either direction from zero.

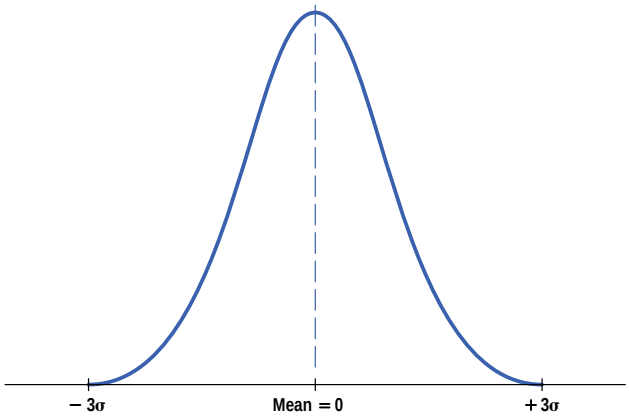


Fig. 1. *Assumed normal distribution of planning errors.*

We would like to know both the expected value of X and its variance. Knowing these two values will form the basis for the ultimate decision rules for replenishment order sizes placed to the supplier for our part.

We will use the following notation: E(x) represents the expected value of the random variable x, and V(x) represents the variance of the random variable x.

Before launching into the derivation of the expected value of the real demand over the next L + R weeks, note that L itself is a random variable. When an order is placed with the supplier, delivery does not always come exactly on the acknowledgment date. There is some uncertainty associated with when the replenishment order will arrive. Like the planning errors, we will assume that the delivery errors are normally distributed about zero. Thus:

$$E(X) = E\left(\sum_{i=1}^{L+R}\left(P_i + e_i\right)\right) \cong \sum_{i=1}^{E(L)+R} E\left(P_i + e_i\right)$$

$$= \sum_{i=1}^{\mu_L+R}\left(E\left(P_i\right) + E\left(e_i\right)\right) = \sum_{i=1}^{\mu_L+R} P_i.$$

The result will be precisely correct when the P_i are stationary (that is, the plan is a constant run rate) and will serve as an approximation when the P_i are nonstationary.

Determining the variance of X is more involved because the limit of the summation, L + R, is a random variable. The derivation can be be found in Appendix I. The result is:

$$V(X) \cong \left(\mu_L + R\right)\sigma_e^2 + \overline{P}_{L+R}^2\sigma_L^2.$$

where σ_e is the standard deviation of the errors e_i, σ_L is the standard deviation of L, and $\overline{P}_{L+R}$ is the average of the plan over L + R weeks.

The standard deviation of demand is the square root of this result. In practice, we estimate the standard deviation of demand by:

$$\hat{\sigma}_X \cong \sqrt{\left(\overline{L} + R\right)s_{DE}^2 + \overline{P}_{L+R}^2 s_{LE}^2},$$

where $\overline{L}$ is the average lead time from the supplier of this part, R is the review period, s_{DE}^2 is the variance of the difference between the weekly plan and the actual weekly demand, and s_{LE}^2 is the variance of the difference between the date requested and the date received. Lee, Billington, and Carter[1] give the same result when modeling the demand at a distribution center within a supply chain.

Knowing the variance of the demand uncertainty over L + R weeks, we can develop a decision rule for determining the amount of inventory to carry to meet the actual demand the desired percent of the time.

We define the *order-up-to level* as:

$$\text{Order-up-to Level} = \sum_{i=1}^{L+R} P_i + Z_{1-\alpha}\hat{\sigma}_X,$$

where $Z_{1-\alpha}$is the standard normal value corresponding to a probability α of stocking out. $Z_{1-\alpha}\hat{\sigma}_X$ is called the safety stock.

We define the *inventory position* as follows:

Inventory Position = On-Hand Quantity
+ On-Order Quantity
− Back-Ordered Quantity.

The purchase order size decision rule each R weeks for replenishment of this part becomes:

New Order Quantity = Order-up-to Level
− Inventory Position.

We are simply trying to keep the order-up-to level of material in the supply pipeline over the next L + R weeks, knowing we have a probability α of stocking out.

As you can see, the basic idea behind the statistical calculation of safety stock is straightforward. In practice, a number of complicating factors must be accounted for before we can make this technology operational. The list of issues includes:

- The chosen frame of reference for defining and measuring future demand uncertainty
- The impact of SRT objectives on inventory requirements
- The translation from part service level to finished product service level
- Appropriate estimates for demand and supply uncertainty upon which to base the safety stock calculations
- Purchasing constraints when buying from suppliers
- The hidden effect of review period on service level performance
- The definition of service level.

There are significant business outcomes from managing inventory with the statistical calculation of safety stock. These include the ability to:

- Predict average on-hand inventory and the range over which physical inventory can be expected to vary
- Trade off service level and inventory
- Trade off SRT and inventory
- Plot order aging curves so that you can see how long customers may have to wait when stock-outs do occur
- Measure the impact of reducing lead times, forecasting error, and delivery uncertainty
- Measure the impact of changing review periods and minimum order quantities to the supplier
- Stabilize the orders placed to suppliers so that they are not being subjected to undue uncertainties
- Reduce procurement overhead required for manipulating orders.

Turning off the Production Plan Overdrive

Many manufacturing planning organizations have traditionally handled the uncertainties of future demand by intentionally putting a near-term overdrive into the production plan (see Fig. 2). By driving the material requirements plan (MRP) higher than expected orders, a buffer of additional material is brought into the factory to guard against the inevitable differences between forecast and actual demand. In effect, this overdrive, or front loading, functions as safety stock, although it is never called that by the materials system.

While this practice has helped many factories meet shipment demands, it has also caused frustrations with nonoptimal inventory levels. Biasing the build plan high across all products does not consider that it is unlikely that all of the products will be simultaneously above their respective forecasts. Therefore, inventories on parts common to several products tend to be excessive. Also, this approach treats all parts the same regardless of part lead times, rather than allocating safety stock inventory based upon each part's procurement lead time. The factory can easily end up with inventories too high on short lead time parts and too low on longer lead time parts. Finally, the practice of building a front-end overdrive into the plan can lead to conflict between the procurement and production planning departments. Wanting to ensure sufficient material to meet customer demand, the planning department's natural desire is to add a comfortable pad to the production plan. Procurement, aware of the built-in overdrive in the plan and under pressure to reduce inventories, may elect to second-guess the MRP system and order fewer parts than suggested. Should planning become aware that the intended safety pad is not really there, it can lead to an escalating battle between the two organizations.

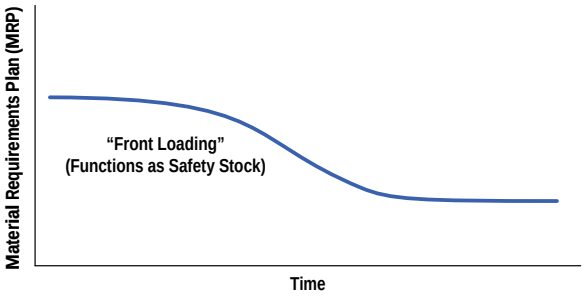


Fig. 2. *Many manufacturing planning organizations handle the uncertainties of future demand by intentionally driving the material requirements plan (MRP) higher than expected orders.*

Frame of Reference

Fundamental to the use of the statistical safety stock methods outlined in this paper is how one chooses to measure demand uncertainty, or in other words, what is the point of reference. The two alternative views are (see Fig. 3):

- Demand uncertainty is the difference between part consumption in the factory and planned consumption.
- Demand uncertainty is the difference between real-time customer demand and the forecast.

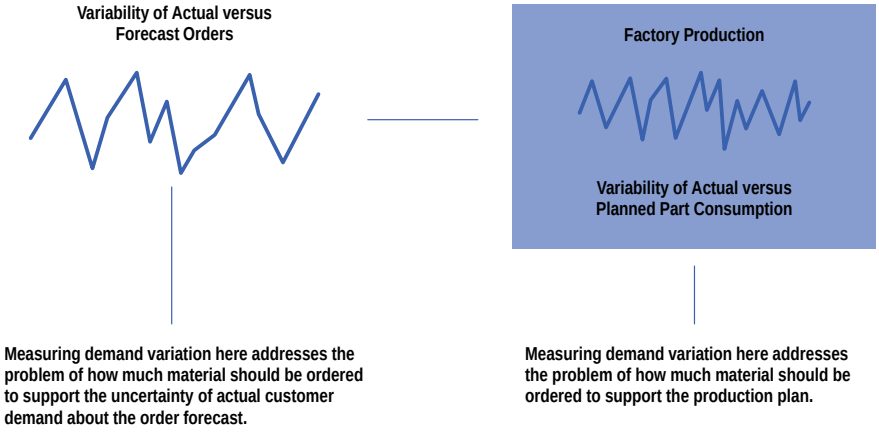


Fig. 3. *Frames of reference for measuring demand uncertainty. These two measures can be very different in a factory dedicated to steady build rates according to a build plan. In a factory fluctuating its production is response to actual orders, these two measures are more alike.*

Consider using part consumption within the factory versus build plan as the frame of reference. The function of statistical safety stocks here is to provide confidence that material is available to support the production plan. A factory with a steady-rate build plan would carry relatively little safety stock because there are only small fluctuations in actual part consumption. Of course, actual order fulfillment performance would depend upon finished goods inventory and the appropriateness of the plan. In this environment, the organization's SRT objective has no direct bearing on the safety stock calculations. The factors influencing the estimate of demand uncertainty and hence safety stock are fluctuations in actual builds from the planned build, part yield loss, and part use for reasons other than production.

If the point of reference calls for measuring demand uncertainty as the deviation between the forecast and real-time incoming customer orders, safety stock becomes a tool to provide sufficient material to meet customer demand. This factory is not running steady-state production but rather building what is required. Now the SRT objective should be included in the safety stock calculations since production does not have to build exactly to real-time demand if the SRT objective is not zero. From this perspective, statistical safety stocks, projected on-hand inventory, SRT, and service levels are all tied together, giving a picture of the investments necessary to handle marketplace uncertainty and still achieve order fulfillment goals.

In choosing between these two frames of reference for the definition of demand uncertainty it comes down to an analysis of factory complexity and timing. If factory cycle times are relatively short so that production is not far removed from customer orders, then demand uncertainty can be measured as real-time orders versus forecast. However, if factory cycle times are long so that production timing is well-removed from incoming orders, then demand uncertainty would best be measured as part consumption versus build plan.

SRT in Safety Stock Calculations

Appendix IV documents the mathematics for incorporating SRT objectives into the safety stock calculations. As has been discussed, using the SRT mathematics would be appropriate when measuring demand uncertainty as deviations of real-time customer orders from forecast. It is critical, however, that we understand how production cycle times affect the factory's actual SRT performance.

As stated in Appendix IV, if factory cycle time is considered to be zero, the SRT mathematics ensures that material sufficient to match customer orders will arrive no later than the desired number of weeks after the customer's order. Clearly, time must be allocated to allow the factory to build and test the completed product. In this paper, this production time is not the cycle time for building one unit but for building a week's worth of demand.

Care must be taken when using the SRT mathematics. Consider that the practice of booking customer orders inside the SRT window will place demands on material earlier than expected from the mathematical model given in Appendix IV. In practice, one should be conservative and use perhaps no more than half of the stated SRT as input to the safety stock model.

Part versus Product Service Level

The statistical mathematics behind the safety stock calculations are actually ensuring a service level for parts availability and not for completed product availability. This is true regardless of whether the chosen frame of reference for measuring demand uncertainty is part-level consumption or product-level orders. Since production needs a complete set of parts to build the product the question arises as to what the appropriate part service level should be to support the organization's product service level goals. Unfortunately, there is not a simple algebraic solution to this problem.

The exact answer is subject to the interdependencies among the probabilities of stocking out of any of the individual parts in the bill of materials. If we assume that the probabilities of stocking out of different parts are statistically independent, then the situation looks bleak indeed. For example, if we have a 99% chance of having each of 100 parts needed to build a finished product, independence would suggest only a $0.99^{100} = 36.6\%$ chance of having all the parts. Clearly the chance of stocking out of one part is not totally independent of stocking out of another. For example, if customer demand is below plan there is less chance of stocking out of any of the parts required. Just as clearly, there is not total dependence among parts. One supplier may be late on delivery, causing a stock-out on one part number while there are adequate supplies of other parts on the bill of materials. In the example mentioned, the truth about product service level lies between the two extremes, that is, somewhere between 0.99^{100} and 0.99.

As an operational rule of thumb, individual part service levels should be kept at 99% or greater. Of course, the procurement organization may choose to run inexpensive parts at a 99.9% or even higher service level so as never to run out. Then the service level on expensive parts can be lowered such that the factory gets the highest return on its inventory dollar. For example, a factory may run a critical, expensive part at a 95% service level while maintaining a 99.9% service level on cheaper components to achieve a product level goal of a 95% service level to the SRT objective.

Parts Common to Multiple Products

In the problem formulation section it was assumed that we were dealing with a part unique to a single product and used only once to build that product. First, recognize that the situation in which a part is unique to a single product, but happens to be used more than once to build the product, is trivial. If the product uses a part k times then the forecasted part demand is simply k times the forecast for the product. Similarly, the standard deviation of the forecast error for the part is simply k times the standard deviation of the forecast error for the product.

The more interesting situation arises when a part is common to multiple products. We will look at two alternative approaches to handling common parts, the second method being superior to the first. In the first approach, we will assume that the forecasting errors for the products using the common part are independent of one another. Since the total forecasting error for the part can be written as the sum of the forecasting errors for each of the products using the part, the standard deviation of the part forecasting uncertainty can be easily determined.

Consider a part used in j products and used k_i times in product i, where $i = 1, 2, ..., j$. Let DE represent the forecasting or demand error. Then:

$$DE_{part} = k_1 DE_{product1} + k_2 DE_{product2} + k_3 DE_{product3} + ... + k_j DE_{productj}$$

$$\sigma^2_{DEpart} = k_1^2 \sigma^2_{DEproduct1} + k_2^2 \sigma^2_{DEproduct2} + k_3^2 \sigma^2_{DEproduct3} + ... + k_j^2 \sigma^2_{DEproductj}.$$

The big problem with this approach is the assumption of independence of forecasting errors among all the products using the part. If, for example, when one product is over its forecast there is a tendency for one or more of the others to be over their forecasts, the variance calculated as given here will underestimate the true variability in part demand uncertainty.

The second approach to estimating forecasting uncertainty for common parts is to explode product-level forecasts into part-level forecasts and product-level customer demand into part-level demand and measure the demand uncertainty directly at the part level. For a part common to j products we simply measure the forecast error once as the difference between the part forecast and actual part demand instead of measuring the forecast errors for the individual products and algebraically combining them as before. Any covariances between product forecasting uncertainties will be picked up in the direct measurement of the part-level forecasting errors. Clearly, this is the preferred approach to estimating part demand uncertainty, since it avoids making the assumption of forecast error independence among products using the part.

Estimation of Demand and Part Delivery Uncertainty

The whole approach to safety stocks and inventory management outlined here is dependent upon the basic premise behind any statistical sampling theory—namely, that future events can be modeled by a sample of past events. Future demand uncertainty is assumed to behave like past demand uncertainty. Future delivery uncertainty is assumed to behave like the supplier's historical track record. This raises two issues when estimating the critical inputs to the safety stock equations: robust estimation and business judgment. Both of these issues are extremely dependent upon the chosen frame of reference, that is, whether we are measuring real-time customer demand or part-level consumption on the factory floor.

From a sample size perspective we would like to have as much data as possible to estimate both demand and delivery uncertainty. However, in a rapidly changing business climate we may distrust data older than, say, six months or so. If I am measuring demand uncertainty as the deviations between real-time customer orders and the forecast, do I want to filter certain events so they do not influence the standard deviation of demand uncertainty and hence safety stocks? It may be good business practice not to allow big deals to inflate the standard deviation of demand uncertainty if those customers are willing to negotiate SRT. In statistical jargon, we want our estimates going into the safety stock equation to be robust to outliers. Naturally, if the demand uncertainty is measured as part consumption on the factory floor versus planned consumption, data filtering is not an issue. It is possible that an unusual event affecting parts delivery from a supplier may be best filtered from the data so that the factory is not holding inventory to guard against supply variability that is artificially inflated.

A common situation is the introduction of a new product. Suppose the chosen point of reference is measuring demand uncertainty as real-time customer orders versus forecast. How do we manage a new product introduction? A viable option is to use collective business judgment to set the demand uncertainty even though there is technically a sample size of zero before introduction. Prior product introductions or a stated business objective of being able to handle demand falling within $\pm\Delta$ of the plan during the early sales months can be used to establish safety stocks. In fact, the organization can compare the inventory costs associated with different assumptions about the nature of the demand volatility. Estimates of average inventory investment versus assumed demand uncertainty obtained from the statistical models can help the business team select an introduction strategy.

Effect of Minimum Buy Quantities and Desired Delivery Intervals

In most cases, there are constraints on the order sizes we place to our suppliers, such that replenishment orders are not exactly the difference between the theoretical order-up-to level and the inventory position. These constraints may be driven by the supplier in the form of minimum buy quantities or ourselves in the form of economic order quantities or desired delivery frequencies. The net effect of all such constraints on order sizes is to reduce the periods of exposure to stock-outs.

For example, suppose the factory's plans predict needing 100 units of some part per week. Further suppose that the ordering constraint is that we order 1000 units at a time determined by either the supplier's minimum or our economic order quantity. This order quantity represents ten weeks of anticipated demand. Once the shipment of parts arrives from the supplier, there is virtually no chance of stocking out for several weeks until just before the arrival of the next shipment. Given this observation we see that safety stock requirements actually decrease as purchase quantity constraints increase (see **Appendix V**).

Although safety stocks decrease, average on-hand inventory and the standard deviation of on-hand inventory both increase. See **Appendix III** for formula derivations of the average and the standard deviation of on-hand inventory.

Effect of Review Period

Analysis of the equation for the standard deviation of demand uncertainty given above shows that as the review period R increases, σ_X increases, thereby driving up safety stock. This makes sense because the safety stock is there to provide the desired confidence of making it through R weeks without a stock-out. However, note that the service level metric itself is changing. For $R=1$, the service level gives the probability of making it through each week without a stock-out. For $R=2$, the service level gives the probability of making it through two weeks, for $R=3$, three weeks, and so on. Increasing review period therefore has an effect similar to that of minimum buy quantities. When operating at longer review periods, purchase quantities to the supplier are larger, since we are procuring to cover R weeks of future demand and not just one week of future demand. To keep the average weekly service level at the desired goal, safety stock would actually have to be throttled back as the review period increases because of less frequent periods of exposure.

Service Level Metric

Throughout this paper, service level has been defined as the probability of not stocking out over a period of time, usually on a weekly basis. There is another commonly used service level metric called the *line item fill rate* (LIFR). With the LIFR the issue is not whether stock-outs occur but rather whether there is at least the desired percentage of the required items available. For example, suppose in a week of factory production, demand for a part is 100 units but there are only 95 available. Measured in terms of LIFR, the service level is 95%.

Proponents of LIFR argue that the metric gives appropriate credit for having at least some of what is required, whereas the probability of stock-out metric counts a week in which there was 95% of the required quantity of a particular part as a stock-out.

When calculating safety stocks to a LIFR metric rather than multiplying the standard deviation of demand over the lead time plus the review period by a standard normal value, solve for k in the following approximation formula:[2]

$$\text{LIFR}_{\text{goal}} = 1 - \frac{\sigma_X}{\mu_D} e^{\left(-0.92 - 1.19k - 0.37k^2\right)}.$$

where μ_D is the average weekly demand. Then the safety stock is $k\sigma_X$.

Inventory versus Service Level Exchange Curves

A useful graphical output from the statistical inventory mathematics is the inventory versus service level exchange curve as shown in Fig. 4.

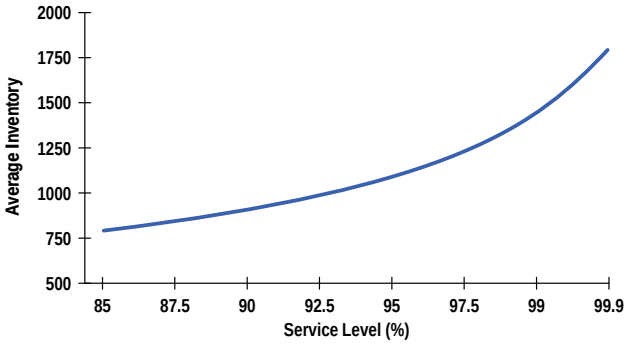


Fig. 4. *Average inventory as a function of service level.*

Such graphs demonstrate the nonlinear relationship between increasing inventory and service level given the constraints on the factory. The curve represents the operating objective. (Johnson and Davis[3] refer to this curve as the "efficient frontier.") By comparing historical inventory and service levels to the performance levels possible as indicated in Fig. 4, a factory can gauge how much room it has for improvement. In addition, procurement can determine where on the curve they should be operating based upon their cost for expediting orders. As can be seen in Fig. 4, a factory operating in the 90% service level range would get a lot of leverage from inventory money invested to move them to 95% service. However, moving from 95% to 99% service level requires more money and moving from 99% to 99.9% requires more yet. By comparing the cost (and success rate) of expediting parts to avoid stock-outs with the cost of holding inventory, the organization can determine the most cost-effective operating point.

Order Aging Curves

Another useful graphical output is the order aging curve. This curve in a sense tells the rest of the story about material availability to meet the SRT and service level objectives. More specifically, the curve demonstrates what type of service can be expected for SRTs shorter than the objective and how long customers can be expected to wait when you are unable to meet your SRT objective. Fig. 5 shows a family of order aging curves, each corresponding to a certain safety stock value determined by the stated SRT goal. We see, for example, that a factory holding safety stocks to support a 99% service level on a two-week SRT goal could, in fact, support a one-week SRT with a service level better than 90%. That same factory will almost surely have all orders filled no later than four weeks from receipt of customer order.

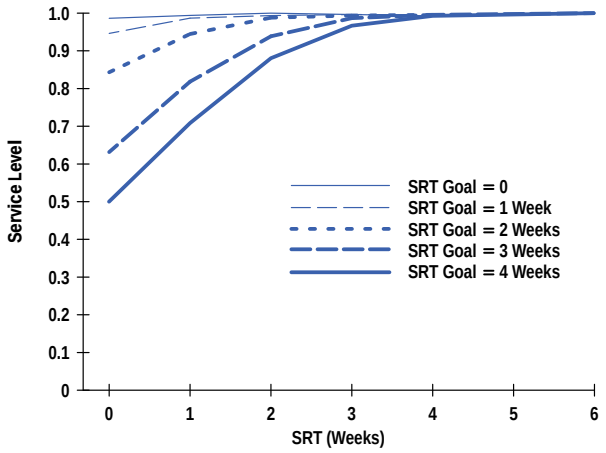


Fig. 5. *Order aging curves for differing SRT (supplier response time) goals.*

Theory versus Practice

Ultimately, the actual performance the factory experiences in the key metrics of service level to the SRT objectives and average on-hand inventory will depend upon whether the supply chain performs according to the inputs provided to the statistical model. All of the estimates are predicated upon the future supply chain parameters fluctuating within the estimated boundaries. As depicted in Fig. 6, we have built up a set of assumptions about the nature of the various uncertainties within our supply chain. If one or more of these building blocks proves to be inaccurate, the factory will realize neither the service level nor the inventory projected.

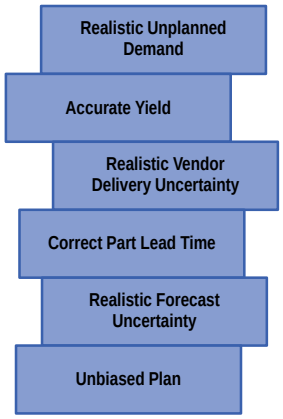


Fig. 6. *Supply chain inputs. The accuracy of the estimates of service level and on-hand inventory are dependent on the validity of the inputs.*

Acknowledgments

Special thanks to Rob Hall of the HP strategic planning and modeling group and Greg Larsen of the Loveland Manufcturing Center for assistance in the development of this theory and helpful suggestions for this paper. Not only thanks but also congratulations to the process engineering, planning, and procurement organizations of the Colorado Springs Division for reengineering division processes to put supply chain theory into practice.

References

1. H.L. Lee, C. Billington, and B. Carter, “Hewlett-Packard Gains Control of Inventory and Service through Design for Localization,” *Interfaces*, Vol. 23, no. 4, July-August 1993, p. 10.
2. S. Nahmias, *Production and Operations Analysis*, Richard Irwin, 1989, p. 653.
3. M.E. Johnson and T. Davis, *Improving Supply Chain Performance Using Order Fulfillment Metrics*, Hewlett-Packard Strategic Planning and Modeling Group Technical Document (Internal Use Only), 1995, p. 14.

Appendix I: Derivation of the Standard Deviation of Demand Given an R-Week Review Period

$$X = \sum_{i=1}^{L+R} D_i = \sum_{i=1}^{L+R} (P_i + e_i)$$

$$V(X) = E(V(X|L)) + V(E(X|L))$$

$$= E\left(V\left(\sum_{i=1}^{L+R}(P_i + e_i)\middle|L\right)\right) + V\left(E\left(\sum_{i=1}^{L+R}(P_i + e_i)\middle|L\right)\right)$$

$$= E\left(\sum_{i=1}^{L+R} V(P_i + e_i)\right) + V\left(\sum_{i=1}^{L+R} E(P_i + e_i)\right)$$

$$= E\left(\sum_{i=1}^{L+R} \sigma_e^2\right) + V\left(\sum_{i=1}^{L+R} P_i\right)$$

$$\cong \sum_{i=1}^{E(L)+R} E(\sigma_e^2) + V(\bar{P}_{L+R}(L + R))$$

$$\cong (\mu_L + R)\sigma_e^2 + \bar{P}_{L+R}^2\sigma_L^2$$

Hence,

$$\sigma_X \cong \sqrt{(\mu_L + R)\sigma_e^2 + \bar{P}_{L+R}^2\sigma_L^2}.$$

We estimate σ_X by:

$$\hat{\sigma}_X \cong \sqrt{(\bar{L} + R)s_{DE}^2 + \bar{P}_{L+R}^2 s_{LE}^2}$$

where: $\bar{L}$
= average lead time from supplier of this part

R = review period

s_{DF}^2
= variance of the difference between the weekly plan and the actual demand

$\bar{P}_{i,n}^2$
= average of the plan over $L + R$ weeks

s_{LE}^2
= variance of the difference between the date requested and the date received.

Appendix II: The Expected Value and Variance of On-Hand Inventory when there Are no Restrictions on Minimum Buy Quantities

Let: I = On-hand physical inventory
S = Order-up-to level
Y = Amount of part consumed in first L weeks of the (L + R)-week cycle
C_S = Cycle stock = stock consumption to date during the R-week portion of the (L + R)-week cycle
SS = Safety stock

$$I = S - Y - C_S$$

$$I = \left(\sum_{i=1}^{L+R} P_i + SS\right) - \left(\sum_{i=1}^{L} D_i\right) - C_S$$

$$E(I) = E\left(\sum_{i=1}^{L+R} P_i + SS\right) - E\left(\sum_{i=1}^{L} D_i\right) - E(C_S)$$

$$E(I) \cong \sum_{i=1}^{E(L)+R} P_i + SS - \sum_{i=1}^{E(L)} P_i - E(C_S).$$

We will consider C_S to be uniformly distributed between 0 and $\sum_{i=L+1}^{L+R} D_i$. Thus,

$$E(I) \cong \sum_{i=1}^{E(L)+R} P_i + SS - \sum_{i=1}^{E(L)} P_i - \frac{1}{2}\sum_{i=E(L)+1}^{E(L)+R} P_i$$

$$E(I) \cong SS + \frac{1}{2}\sum_{i=E(L)+1}^{E(L)+R} P_i = SS + \frac{R\bar{P}_R}{2}.$$

The variance of I is derived as follows.

$$V(I) = V(S) + V(Y) + V(C_S)$$

Even though the P_i are not all fixed, and hence S changes every R weeks, S is still a constant with respect to the inventory result during the last R weeks of every (L + R)-week cycle. Hence, $V(S) = 0$.

$$V(I) = 0 + V\left(\sum_{i=1}^{L} D_i\right) + V(C_S)$$

$$V(I) \cong \left(\sigma_{\hat{e}}^2 \mu_L + \sigma_L^2 \bar{P}_L^2\right) + V(C_S)$$

$$V(C_S) = E\left(V\left(C_S \middle| D_{L+1}, D_{L+2}, \ldots, D_{L+R}\right)\right) + V\left(E\left(C_S \middle| D_{L+1}, D_{L+2}, \ldots, D_{L+R}\right)\right)$$

$$E\left(C_S \middle| D_{L+1}, D_{L+2}, \ldots, D_{L+R}\right) = \frac{D_{L+1} + D_{L+2} + \ldots + D_{L+R}}{2}$$

$$V\left(E\left(C_S \middle| D_{L+1}, D_{L+2}, \ldots, D_{L+R}\right)\right) = V\left(\frac{D_{L+1} + D_{L+2} + \ldots + D_{L+R}}{2}\right)$$

$$= \frac{1}{4}V\left(\left(P_{L+1} + e_{L+1}\right) + \left(P_{L+2} + e_{L+2}\right) + \ldots + \left(P_{L+R} + e_{L+R}\right)\right)$$

$$= \frac{R\sigma_e^2}{4}$$

$$V\left(C_S \middle| D_{L+1}, D_{L+2}, \ldots, D_{L+R}\right) = \frac{\left(D_{L+1} + D_{L+2} + \ldots + D_{L+R}\right)^2}{12}$$

$$E\left(V\left(C_S \middle| D_{L+1}, D_{L+2}, \ldots, D_{L+R}\right)\right)$$

$$= E\left[\frac{\left(D_{L+1} + D_{L+2} + \ldots + D_{L+R}\right)^2}{12}\right] = \frac{1}{12}E\left(G^2\right),$$

where $G = D_{L+1} + D_{L+2} + \ldots + D_{L+R}$.

$$E\left(G^2\right) = \left(\sigma_G^2 + \mu_G^2\right) = \left[R\sigma_e^2 + \left(\sum_{i=L+1}^{L+R} P_i\right)^2\right]$$

$$E\left(V\left(C_S \middle| D_{L+1}, D_{L+2}, \ldots, D_{L+R}\right)\right) = \frac{1}{12}\left[R\sigma_e^2 + \left(\sum_{i=L+1}^{L+R} P_i\right)^2\right]$$

$$V\left(C_S\right) = \frac{1}{12}\left[R\sigma_e^2 + \left(\sum_{i=L+1}^{L+R} P_i\right)^2\right] + \frac{R\sigma_e^2}{4}$$

Hence,

$$V(I) \cong \sigma_e^2\mu_L + \sigma_L^2\bar{P}_L^2 + \frac{1}{12}\left[R\sigma_e^2 + \left(\sum_{i=L+1}^{L+R} P_i\right)^2\right] + \frac{R\sigma_e^2}{4},$$

where $\bar{P}_L$ is the average of the plan over the L-week period immediately before the R-week period in question.

Appendix III: The Expected Value and Variance of On-Hand Inventory when there Are Restrictions on Minimum Buy Quantities

Here we assume that restrictions on the size of orders placed to the supplier prevent procurement from ordering exactly the difference between the order-up-to level and the inventory position. The restriction in order size might be the result of a minimum buy size constraint placed by the supplier, a constraint that the order must be an integer multiple of a specified quantity, or the purchaser's desire that deliveries come at some delivery interval greater than weekly.

Let: Min = minimum order size constraint
Mult = multiple order size constraint
DI = desired delivery interval constraint.

Then the order size decision rule is given by:

$$\text{New order size} = M = k \times \text{Mult},$$

where k is the smallest integer such that:
1. M ≥ Order-up-to Level − Inventory Position
2. M ≥ Min
3. M ≥ DI × Average Weekly Demand.

Finally, we assume that the order is placed for the entire order quantity to be delivered L weeks later, that is, the order is not partitioned into pieces with separate delivery dates.

Let: I = On-hand physical inventory
S = Order-up-to level
Y = Amount of part consumed in first L weeks of the (L + R)-week cycle
C_S = Cycle stock = stock consumption to date during the R-week portion of the (L + R)-week cycle
SS = Safety stock
M = Order quantity
Δ = Increment above the order-up-to level S that the inventory position reaches as a result of having to order a quantity M.

$$I = (S + \Delta) - Y - C_S$$

$$E(I) = E(S) + E(\Delta) - E(Y) - E(C_S)$$

$$E(I) \cong \left(\sum_{i=1}^{L+R} P_i + SS\right) + E(\Delta) - \sum_{i=1}^{L} P_i - \frac{1}{2}\sum_{i=L+1}^{L+R} P_i$$

$$E(I) \cong SS + E(\Delta) + \frac{1}{2}\sum_{i=L+1}^{L+R} P_i.$$

To determine E(Δ) note that rather than buying strictly an amount equal to (S − Inventory Position) we buy a quantity M. Therefore, the difference between what would be ordered without minimums and what is ordered with minimums varies between 0 and M − 1. We will assume that this difference is uniformly distributed within this range. Thus:

$$E(I) \cong SS + \frac{M-1}{2} + \frac{1}{2}\sum_{i=L+1}^{L+R} P_i.$$

The derivation of the variance of I is as follows.

$$V(I) = V(S) + V(\Delta) + V(Y) + V(C_S)$$

$$V(I) = 0 + V(\Delta) + V\left(\sum_{i=1}^{L} D_i\right) + V(C_S)$$

$$V(I) \cong V(\Delta) + \sigma_e^2 \mu_L + \sigma_L^2 \bar{P}_L^2 + \frac{1}{12}\left[R\sigma_e^2 + \left(\sum_{i=L+1}^{L+R} P_i\right)^2\right] + \frac{R\sigma_e^2}{4}$$

$$V(I) \cong \frac{(M-1)^2}{12} + \sigma_e^2 \mu_L + \sigma_L^2 \bar{P}_L^2 + \frac{1}{12}\left[R\sigma_e^2 + \left(\sum_{i=L+1}^{L+R} P_i\right)^2\right] + \frac{R\sigma_e^2}{4}$$

where $\bar{P}_L$ is the average of the plan over the L-week period immediately before the R-week period in question.

Appendix IV: Incorporating SRT (Supplier Response Time) into the Safety Stock Calculations

A weekly review period is assumed.

Let: X = actual amount of demand for an arbitrary part in $L+1$ weeks.

$$S = \text{Order-up-to level} = \sum_{i=1}^{L+1} P_i + Z_{1-\alpha}\sigma_X.$$

X is assumed to be normally distributed with mean $(L+1)\mu_D$ and variance $\sigma_D^2(L+1) + \sigma_L^2\mu_D^2$.

Then $\text{Prob}(X \leq S) = 1 - \alpha$, so $1 - \alpha$ is the service level.

One-Week SRT

The probability that some demand is actually filled the week following its arrival is the probability that the order-up-to level over $L+1$ weeks covers demand incurred over just L weeks.

Let X* be the amount of demand in L weeks. X* is normally distributed with mean $L\mu_D$ and variance $\sigma_D^2 L + \sigma_L^2\mu_D^2$.

If SS_1 denotes the appropriate safety stock for a one-week SRT, the corresponding order-up-to level for a one-week SRT goal is $S_1 = \sum_{i=1}^{L+1} P_i + SS_1$. However,

$$\text{Prob}(X^* \leq S_1) = \text{Prob}\left(Z \leq \frac{S_1 - L\mu_D}{\sqrt{\sigma_D^2 L + \sigma_L^2\mu_D^2}}\right) = 1 - \alpha.$$

This implies that

$$Z_{1-\alpha} = \frac{S_1 - L\mu_D}{\sqrt{\sigma_D^2 L + \sigma_L^2\mu_D^2}}$$

$$S_1 = Z_{1-\alpha}\sqrt{\sigma_D^2 L + \sigma_L^2\mu_D^2} + L\mu_D.$$

The order-up-to-level will still be calculated by our in-house procurement system, POPLAN, as $S_1 = \sum_{i=1}^{L+1} P_i + SS_1$, so we now have two expressions for S_1. Assuming that $\mu_D = \bar{P}_{L+1}$,

$$S_1 = (L+1)\mu_D + SS_1 = Z_{1-\alpha}\sqrt{\sigma_D^2 L + \sigma_L^2\mu_D^2} + L\mu_D$$

$$SS_1 = Z_{1-\alpha}\sqrt{\sigma_D^2 L + \sigma_L^2\mu_D^2} + L\mu_D - (L+1)\mu_D$$

$$SS_1 = Z_{1-\alpha}\sqrt{\sigma_D^2 L + \sigma_L^2\mu_D^2} - \mu_D.$$

By using an order-up-to level of $\sum_{i=1}^{L+1} P_i + SS_1$, over $L+1$ weeks we will bring in enough material to cover the demand incurred in L weeks a percentage of the time equal to $(1 - \alpha) \times 100\%$.

Two-Week SRT

The probability that some demand is actually filled two weeks after its arrival is the probability that the order-up-to level over $L+1$ weeks covers demand over just $L - 1$ weeks.

Let X** denote the amount of demand in $L - 1$ weeks and let S_2 denote the order-up-to level appropriate for a two-week SRT. X** is normally distributed with mean $(L - 1)\mu_D$ and variance $\sigma_D^2(L - 1) + \sigma_L^2\mu_D^2$.

$$\text{Prob}\left(X^{**} \leq S_2\right) = \text{Prob}\left(Z \leq \frac{S_2 - (L - 1)\mu_D}{\sqrt{\sigma_D^2(L - 1) + \sigma_L^2\mu_D^2}}\right) = 1 - \alpha$$

This implies that

$$Z_{1-\alpha} = \frac{S_2 - (L - 1)\mu_D}{\sqrt{\sigma_D^2(L - 1) + \sigma_L^2\mu_D^2}}$$

$$S_2 = Z_{1-\alpha}\sqrt{\sigma_D^2(L - 1) + \sigma_L^2\mu_D^2} + (L - 1)\mu_D.$$

Since the POPLAN system will calculate order-up-to level as $S2 = \sum_{i=1}^{L+1} P_i + SS_2$. we have two expressions for the order-up-to level, S_2.

$$S_2 = (L + 1)\mu_D + SS_2 = Z_{1-\alpha}\sqrt{\sigma_D^2(L - 1) + \sigma_L^2\mu_D^2} + (L - 1)\mu_D$$

$$SS_2 = Z_{1-\alpha}\sqrt{\sigma_D^2(L - 1) + \sigma_L^2\mu_D^2} + (L - 1)\mu_D - (L + 1)\mu_D$$

$$SS_2 = Z_{1-\alpha}\sqrt{\sigma_D^2(L - 1) + \sigma_L^2\mu_D^2} - 2\mu_D.$$

General Case

In general, the safety stock required for a given SRT goal is given by:

$$SS = Z_{1-\alpha}\sqrt{\sigma_D^2(L + 1 - SRT) + \sigma_L^2\mu_D^2} - (SRT)\mu_D.$$

However, this equation only ensures arrival of material from the supplier no later than the SRT. It does not guarantee that the factory will actually have the final product built and ready for shipment to the customer no later than the SRT. Production cycle time must be incorporated into the equation to make the result useful in setting safety stocks to support product SRT objectives.

Let T_B denote the production cycle time required to build a week's worth of expected demand. Then

$$SS = Z_{1-\alpha}\sqrt{\sigma_D^2\left(L + 1 - SRT + T_B\right) + \sigma_L^2\mu_D^2} - \left(SRT - T_B\right)\mu_D.$$

Consider the three cases exhibited in Fig. 1. If we let build time be two weeks in all three cases and let SRT be 4, 2, and 0 weeks, respectively, then we have the following results.

Case A. $SS = Z_{1-\alpha}\sqrt{\sigma_D^2(L + 1 - 4 + 2) + \sigma_L^2\mu_D^2} - (4 - 2)\mu_D.$

Case B. $SS = Z_{1-\alpha}\sqrt{\sigma_D^2(L + 1 - 2 + 2) + \sigma_L^2\mu_D^2} - (2 - 2)\mu_D.$

Case C. $SS = Z_{1-\alpha}\sqrt{\sigma_D^2(L + 1 - 0 + 2) + \sigma_L^2\mu_D^2} - (0 - 2)\mu_D.$

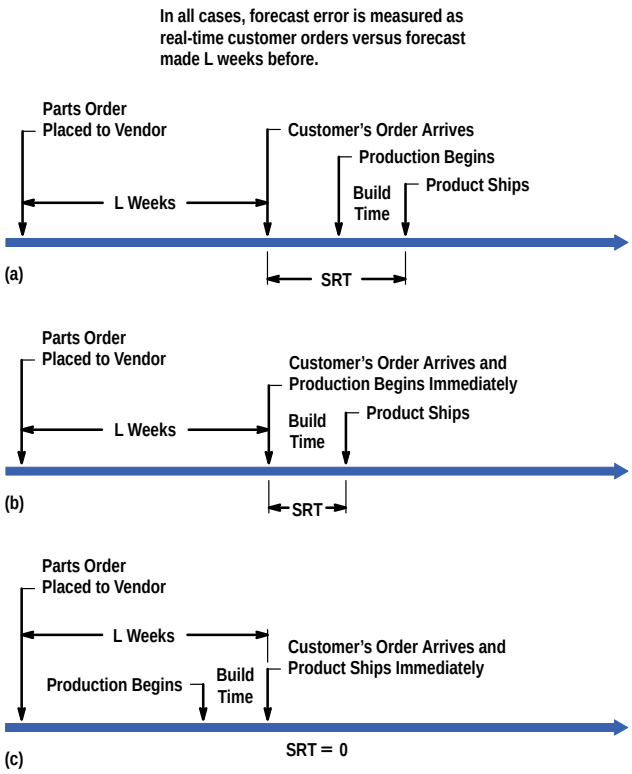


Fig. 1. *Production cycles for different SRT goals. (a) Case A: SRT = 4 weeks. (b) Case B: SRT = 2 weeks. (c) Case C: SRT = 0 weeks.*

Appendix V: Derating the Service Level to Account for Reduced Periods of Exposure to Stock-outs as a Result of Minimum Buy or Economic Order Quantities

When ordering parts from a supplier under either minimum or economic order size restrictions, with each arrival of a shipment from the supplier we would expect the service level to jump to 100% and then decay as indicated in Fig. 1.

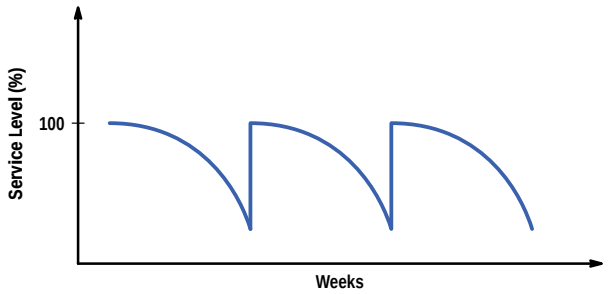


Fig. 1. *The service level jumps to 100% each time a shipment of parts arrives and then gradually decays.*

Since there is realistically only exposure to a stock-out as we approach the anticipated arrival of the next shipment from the supplier, we can afford to run a higher risk of stocking out during these times and still achieve an overall weekly service level objective. The larger the purchase quantity constraints, the less frequent the periods of exposure and, therefore, the lower the service level we can afford at the end of the decay cycle depicted in Fig. 1.

Given that purchase quantity constraints dictate minimum order quantities equivalent to W weeks of expected demand, the objective is to equalize the service level achieved on all parts regardless of the order frequencies. This will be accomplished by basing the service level on a *weekly equivalence*. Given a weekly review period, a weekly desired delivery interval, and no constraints on order sizes, the probability of making it through W weeks without a stock-out is given by:

$$(\text{Weekly Service Level})^{W}.$$

Therefore, if we are ordering in quantities equivalent to W weeks of expected demand, the service level used to determine safety stock should be derated to:

$$(\text{Weekly Service Level Objective})^{W}.$$

Example: We will order in quantities equivalent to four weeks of supply, and we desire a weekly equivalent service level of 99%.

$$\text{Derated Service Level} = (0.99)^{4} = 0.96.$$

Appendix VI: Estimating Weekly Demand Uncertainty from Monthly Data

The standard deviation of demand uncertainty used in the safety stock equation is a measure of the *weekly* uncertainty of real demand about the plan. Ideally, data should be taken on a weekly basis so that this statistic can be estimated directly as the sample standard deviation of the difference between the weekly plan and the actual demand. However, it is fairly common that such data is not readily available. Typically, the factory has data aggregated at the monthly level for comparing plans to actual demand. An estimate of weekly demand uncertainty can still be obtained if we make a simplifying assumption about the interdependence of the demand uncertainty from week to week.

Assumption: Demand uncertainties are independent from week to week within a month, that is, knowing the difference between the actual demand and the plan for this week does not give you any information for predicting the difference between the actual demand and the plan next week. If this is the case, then

$$52\sigma^2_{\text{weekly}} = 13\sigma^2_{\text{monthly}}$$

or

$$\sigma_{\text{weekly}} = \sqrt{\frac{3}{13}}\sigma_{\text{monthly}}.$$

Appendix VII: Adjusting Safety Stock to Account for Yield Loss

Procurement may wish to account for part *yield loss* in some situations. Here we use yield loss in a general sense to include additional part consumption either because of literal losses resulting from failures or damage or because of additional use of the part for unplanned reasons.

Let Y_i denote the weekly yield of an arbitrary part. We will assume that Y_i is distributed according to the binomial distribution.

The actual demand on a part per week is given by:

$$D'_i = \frac{D_i}{Y_i}$$

for week $i = 1, 2, 3, \ldots$. The expected value of the actual demand is:[1]

$$E(D'_i) = E\left(\frac{D_i}{Y_i}\right) \cong \frac{\mu_D}{\mu_Y} - \frac{1}{\mu_Y^2}\mathrm{cov}(D_i, Y_i) + \frac{\mu_D}{\mu_Y^3}V(Y_i).$$

We will assume that yield loss each week is not correlated with the demand each week. Then:

$$E(D'_i) \cong \frac{\mu_D}{\mu_Y} + \frac{\mu_D}{\mu_Y^3}V(Y_i).$$

$$V(Y_i) = \frac{\mu_Y(1-\mu_Y)}{n} \cong \frac{\mu_Y(1-\mu_Y)}{\mu_D/\mu_Y} = \frac{\mu_Y^2(1-\mu_Y)}{\mu_D}$$

where n is the average number of parts used per week and we have approximated n by the average weekly demand divided by the average yield. Thus,

$$E(D'_i) \cong \frac{\mu_D}{\mu_Y} + \frac{\mu_D}{\mu_Y^3}\left(\frac{\mu_Y^2(1-\mu_Y)}{\mu_D}\right) = \frac{\mu_D}{\mu_Y} + \frac{(1-\mu_Y)}{\mu_Y}.$$

When $\mu_Y \leq 50\%$, the term $\frac{(1-\mu_Y)}{\mu_Y}$ is less than or equal to one and has little effect on expected demand. Therefore:

$$E(D'_i) \cong \frac{\mu_D}{\mu_Y}.$$

The variance of the actual demand is:[1]

$$V(D'_i) = V\left(\frac{D_i}{Y_i}\right) \cong \left(\frac{\mu_D}{\mu_Y}\right)^2\left(\frac{\sigma_D^2}{\mu_D^2} + \frac{\sigma_Y^2}{\mu_Y^2} - \frac{2\mathrm{cov}(D_i, Y_i)}{\mu_D\mu_Y}\right).$$

As before, we will assume that yield loss is not correlated with the demand each week:

$$V(D'_i) \cong \left(\frac{\mu_D}{\mu_Y}\right)^2\left(\frac{\sigma_D^2}{\mu_D^2} + \frac{\sigma_Y^2}{\mu_Y^2}\right).$$

Again we will approximate σ_Y^2 by $\frac{\mu_Y^2(1-\mu_Y)}{\mu_D}$:

$$V(D'_i) \cong \left(\frac{\mu_D}{\mu_Y}\right)^2\left(\frac{\sigma_D^2}{\mu_D^2} + \frac{1-\mu_Y}{\mu_D}\right) = \frac{\sigma_D^2}{\mu_Y^2} + \frac{\mu_D(1-\mu_Y)}{\mu_Y^2}.$$

Therefore, by adjusting the expected weekly average demand by dividing by the average yield and adjusting the variance of the weekly demand uncertainty as indicated above, we can obtain approximate values for safety stock, average expected on-hand inventory, and the standard deviation of on-hand inventory using the results obtained earlier in this paper.

However, while we have adjusted the expected weekly demand by the yield loss, our in-house system, POPLAN, will not. Therefore, we must pass the impact of the yield adjustment to POPLAN via the safety stock parameter.

Let SS′ denote the safety stock obtained when using the yield-adjusted average demand and standard deviation of demand uncertainty as derived here. The objective is to pass a safety stock value to POPLAN that results in the appropriate order-up-to level.

The safety stock to pass to POPLAN is given by:

$$SS^* = \left[\frac{\mu_D}{\mu_Y}(L + R) + SS'\right] - \mu_D(L + R).$$

In words, calculate the safety stock and the order-up-to level using the yield-adjusted average weekly demand and the yield-adjusted standard deviation of weekly demand uncertainty, then subtract the product of the average weekly demand without yield adjustment and L + R.

Reference

1. A.M. Mood, F.A. Graybill, and D.C. Boes, *Introduction to the Theory of Statistics*, *Third Edition*, McGraw-Hill, 1974, p. 181, theorem 4.

A New Family of Sensors for Pulse Oximetry

This new family of reusable sensors for noninvasive arterial oxygen saturation measurements is designed to cover all application areas. It consists of four sensors: adult, pediatric, neonatal, and ear clip.

by Siegfried Kästle, Friedemann Noller, Siegfried Falk, Anton Bukta, Eberhard Mayer, and Dietmar Miller

Since the early 1980s, when pulse oximetry was introduced, this noninvasive method of monitoring the arterial oxygen saturation level in a patient's blood (SpO_2) has become a standard method in the clinical environment because of its simple application and the high value of the information it gives nurses and doctors. It is as common in patient monitoring to measure the oxygen level in the blood as it is to monitor heart activity with the ECG. In some application areas, like anesthesia in a surgical procedure, it is mandatory for doctors to measure this vital parameter. Its importance is obvious considering that a human being cannot survive more than five minutes without oxygen supply to the brain.

Before the advent of pulse oximetry, the common practice was to draw blood from patients and analyze the samples at regular intervals—several times a day, or even several times an hour—using large hospital laboratory equipment. These in-vitro analysis instruments were either blood gas analyzers or hemoximeters. Blood gas analyzers determine the partial pressure of oxygen in the blood (pO_2) by means of chemical sensors. Hemoximeters work on spectrometric principles and directly measure the ratio of the oxygenated hemoglobin to the total hemoglobin in a sample of blood (SaO_2).

HP pioneered the first in-vivo technology to measure a patient's oxygen saturation level without the need of drawing blood samples in 1976 with the HP 47201A eight-wavelength ear oximeter.[1] An earprobe was coupled through a fiber-optic cable to the oximeter mainframe, which contained the light source (a tungsten-iodine lamp and interference filters for wavelength selection) and receivers. This instrument served as a "gold standard" for oximetry for a long time and was even used to verify the accuracy of the first pulse oximeters in clinical studies.

The real breakthrough came in the 1980s with a new generation of instruments and sensors that were smaller in size, easier to use, and lower in cost. These new instruments used a slightly different principle from the older, purely empirical multiwavelength technology. Instead of using constant absorbance values at eight different spectral lines measured through the earlobe, the new pulse oximeters made use of the pulsatile component of arterial blood generated by the heartbeat at only two spectral lines. The necessary light was easily generated by two light-emitting diodes (LEDs) with controlled wavelengths. Small LEDs and photodiodes made it possible to mount the optical components directly on the sensor applied to the patient, avoiding the necessity of clumsy fiber-optic bundles.

Instruments and Sensors

The first pulse oximeters were standalone products. HP offered its first pulse oximetry devices as additional measurements for an existing monitoring product, the HP 78352/54 family, in 1988. A year later the Böblingen Medical Division introduced a new modular patient monitor, the Component Monitoring System,[2] for which a pulse oximeter module was also available, the HP M1020A (Fig. 1). The application was limited to adults and the only sensor available was the HP M1190A, an advanced design at that time. This sensor is the ancestor of the new sensor family presented in this paper.

Two years later, the HP 78834 neonatal monitor extended SpO_2 measurement to newborn applications. Third-party sensors were used.

Today, all typical monitoring application areas have discovered pulse oximetry: intensive care, operating rooms, emergency, patient transport, general wards, birth and delivery, and neonatal care. HP monitors serving these areas include the HP M1025A anesthetic gas monitor (1990), the HP Component Transport Monitor (1992), SpO_2 options for the HP M1722A and M1723A CodeMaster XL defibrillators (1994, Fig. 2), and recently, the HP M1205A OmniCare monitor and the HP 1350B maternal SpO_2 option for the HP XM Series fetal monitors (Fig. 3).

New SpO_2 Sensor Family

A new family of reusable HP pulse oximetry sensors is now available (Fig. 4). Lower in cost than previous reusable sensors and easier to use than adhesive disposable sensors, the new HP SpO_2 sensor family is hardware compatible with HP's installed base of pulse oximetry front ends. An upgrade to the software is necessary to update the calibration constants in the instrument algorithms to match the optical characteristics of the new sensors, such as spectra and intensity. The new

Fig. 1. *The HP M1020A SpO_2 front-end module for the HP Component Monitoring System.*

Fig. 2. *An HP CodeMaster defibrillator with SpO_2 channel.*

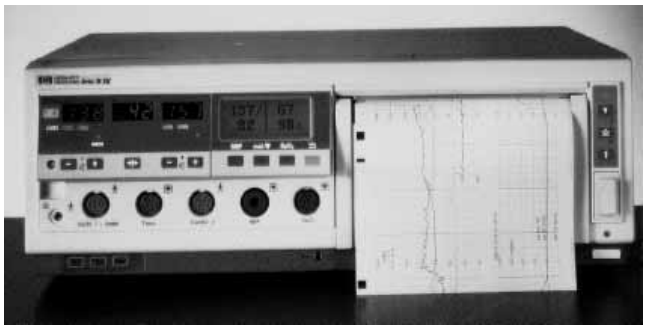


Fig. 3. *The SpO_2 channel in an HP XM Series fetal monitor monitors the mother during delivery.*

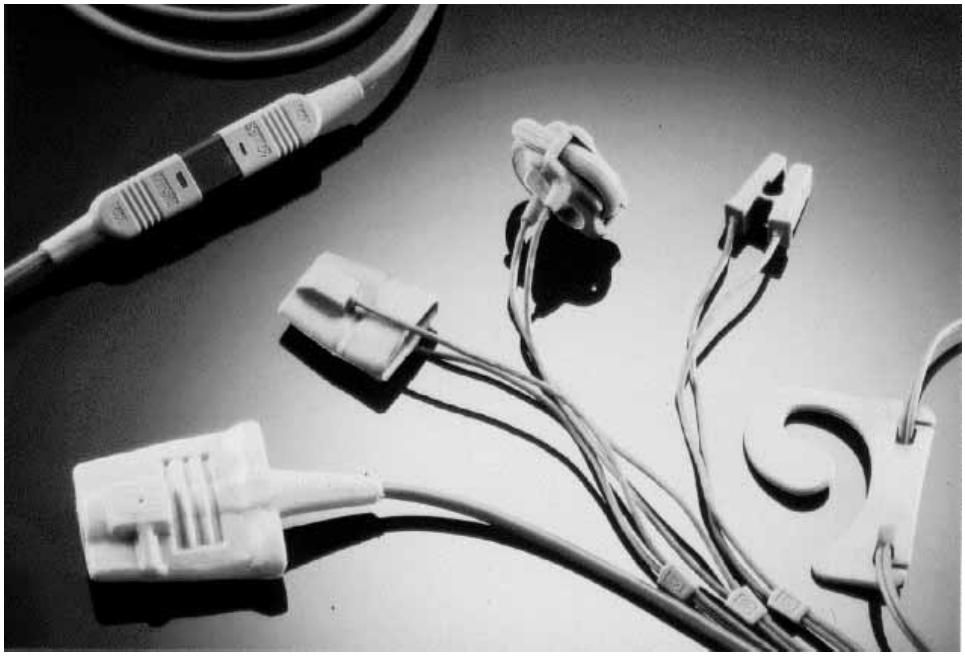


Fig. 4. *The new family of reusable HP pulse oximetry (SpO_2) sensors: (left to right) adult finger glove, pediatric finger glove, neonatal foot strap, ear clip.*

sensor family covers all application areas and consists of the HP M1191A (adult, new wavelength), M1192A (pediatric), M1193A (neonatal), and M1194A (clip).

SpO_2 Basic Measurement Principles

The breakthrough from oximetry to pulse oximetry came with the new LED technology in 1982 to 1985. LED light sources are very small and easy to drive, and have the great advantage that they can be mounted within the sensor together with a photodiode receiver (Fig. 5). For correct measurements at least two LEDs with different wavelengths are necessary. A suitable combination consists of a red LED (650 nm) and an infrared LED (940 nm). The red LED's wavelength has to be in a narrow range, which is not normally possible with standard commercially available LEDs. One way to overcome this is to provide in each sensor a calibration resistor matched to the actual LED wavelength. Another way is to select only LEDs with a fixed wavelength. This method becomes practical if the LED wafer production yields a narrow wavelength distribution. HP decided on this second method because the red LEDs could be obtained from the HP Optoelectronics Division, which had long experience in wafer production and was able to maintain a sufficiently narrow wavelength distribution.

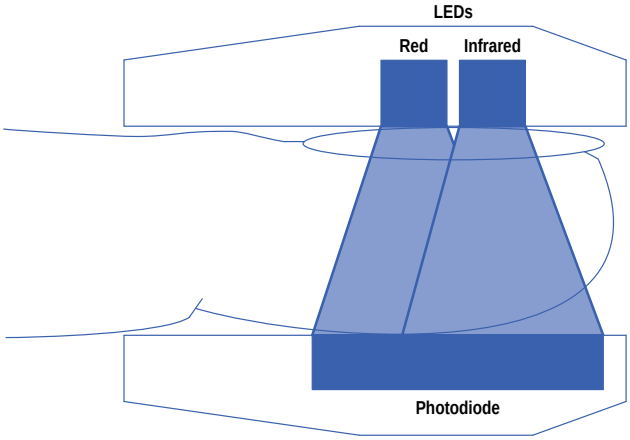


Fig. 5. *The basic components of an SpO_2 pulse oximeter sensor are two LEDs with different wavelengths as light sources and a photodiode as receiver.*

The front-end hardware applies a time multiplexed approach in which the two LEDs are switched on and off alternately. The time phases usually consist of a minimum of three: active red, active infrared, and a dark phase in which the ambient light is measured. There can be more than three phases to allow more LEDs to be powered in one multiplexing time frame or to allow additional dark phases. The phases are similar in duration. The modulation frequency (the complete frame repetition rate) typically ranges from 200 Hz to 2 kHz. The frequency spectrum of such a time multiplexed signal at the receiving photodiode consists of small bands (approximately ± 10 Hz) around the modulation frequency and its harmonics. Depending on the width of the individual LED pulses, the harmonic frequency content is of significant amplitude for several tens of harmonic orders.

For an idealized light absorbing model as shown in Fig. 6, the Lambert-Beer law applies. The intensity I of the light transmitted is related to the incident light I_0 by:

$$I = I_0 \exp(-\mathrm{Ext} \cdot c \cdot d), \qquad (1)$$

where Ext is the extinction coefficient and c is the concentration of a single light absorber with thickness d. Ext varies as a function of the absorbing substance and the wavelength of the light. Further assumptions for the validity of equation 1 are that the light source is monochromatic and has parallel propagation and that the absorber is optically homogeneous (no scattering effects).

Under these assumptions the model of Fig. 6 can be used to derive the basic pulse oximetric quantities. Fig. 7 shows a simplified model for the blood vessel system in tissue. With each heartbeat, the volume of the arteries increases before the blood is forced into the capillaries and from there into the veins. This change of arterial volume is the basis for pulse oximetry because it makes it possible to separate the arterial blood from all other absorbing substances.

Assume that there are N layers of absorbers and that the ith absorber layer has concentration c_i, thickness d_i, and extinction coefficient $\mathrm{Ext}(i,\lambda)$. From equation 1 it follows, at diastole, when there is a maximum of light intensity:

$$I_{max}(\lambda) = I_{LED}(\lambda) \exp\left(-\sum_{i=1}^{N} \mathrm{Ext}(i,\lambda) c_i d_i\right). \qquad (2)$$

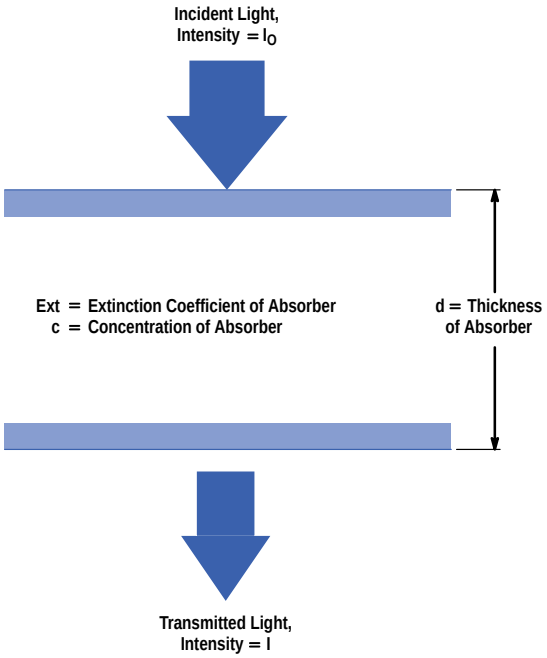


Fig. 6. *Idealized model for the validity of the Lambert-Beer law: a monochromatic light source, parallel light propagation (no point source), and no scattering.*

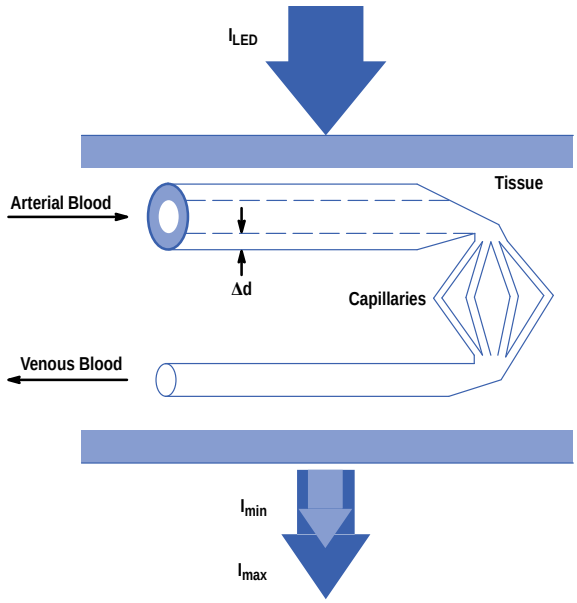


Fig. 7. *Simplified model for the blood vessel system. With each heartbeat, the arterial radius expands by an amount Δd, which yields a light intensity change from I_{max} to I_{min}.*

At systole, the maximum of the heartbeat, and under the assumption that only hemoglobin and oxyhemoglobin are active absorbers in the arterial blood, two additional absorbing parts are added in the exponent of equation 2, which yields the minimum of light intensity:

$$I_{min}(\lambda) = I_{max}(\lambda)\exp(-\Delta d(Ext(Hb,\lambda)[Hb] + Ext(HbO_2,\lambda)[HbO_2])), \quad (3)$$

where [Hb] is the concentration of hemoglobin and $[HbO_2]$ is the concentration of oxyhemoglobin. Dividing equation 2 by equation 3 and taking the logarithm yields the absorption of the arterial blood:

$$\ln\left(\frac{I_{max}(\lambda)}{I_{min}(\lambda)}\right) = \Delta d(Ext(Hb,\lambda)[Hb] + Ext(HbO_2,\lambda)[HbO_2]), \quad (4)$$

where Δd is the change in the arterial radius (see Fig. 7). The definition for the oxygen saturation in pulse oximetry is:

$$SpO_2 = \frac{[HbO_2]}{[Hb] + [HbO_2]}. \quad (5)$$

With two light sources (LEDs) of different wavelengths λ_1 and λ_2 the arterial expansion Δd can be eliminated by the following relation, which is called the *ratio*, $\mathbb{R}$:

$$\mathbb{R} = \frac{\ln\left(\frac{I_{max}(\lambda_1)}{I_{min}(\lambda_1)}\right)}{\ln\left(\frac{I_{max}(\lambda_2)}{I_{min}(\lambda_2)}\right)} = \frac{Ext(Hb,\lambda_1)(1-SpO_2) + Ext(HbO_2,\lambda_1)SpO_2}{Ext(Hb,\lambda_2)(1-SpO_2) + Ext(HbO_2,\lambda_2)SpO_2}. \quad (6)$$

Thus, the oxygen saturation SpO_2 is:

$$SpO_2 = \frac{\mathbb{R}Ext(Hb,\lambda_2) - Ext(Hb,\lambda_1)}{\mathbb{R}(Ext(Hb,\lambda_2) - Ext(HbO_2,\lambda_2)) + Ext(HbO_2,\lambda_1) - Ext(Hb,\lambda_1)}.$$

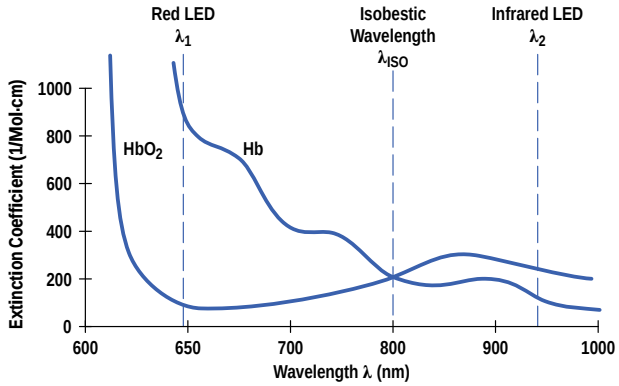


Fig. 8. *Extinction coefficients for hemoglobin Hb and oxyhemoglobin HbO_2 as a function of wavelength. A red LED with $\lambda = 650$ nm gives good resolution between HbO_2 (100% SpO_2) and Hb (0% SpO_2).*

For example, with LED wavelengths $\lambda_1 = 650$ nm and $\lambda_2 = 940$ nm,the extinction coefficients are (see Fig. 8):

$Ext(Hb,650) = 820\ (Mol \cdot cm)^{-1}$
$Ext(HbO_2,650) = 100\ (Mol \cdot cm)^{-1}$
$Ext(Hb,940) = 100\ (Mol \cdot cm)^{-1}$
$Ext(HbO_2,940) = 260\ (Mol \cdot cm)^{-1}$.

In Fig. 9 the SpO_2 is plotted as a function of the ratio $\mathbb{R}$. The Lambert-Beer relation is compared with a calibrated curve derived from real arterial blood samples from volunteers (see subarticle "**Volunteer Study for Sensor Calibration.**"). The deviations exist because conditions in the real case (complicated tissue structure, scattering effects, point light source, etc.) are different from the Lambert-Beer assumptions.

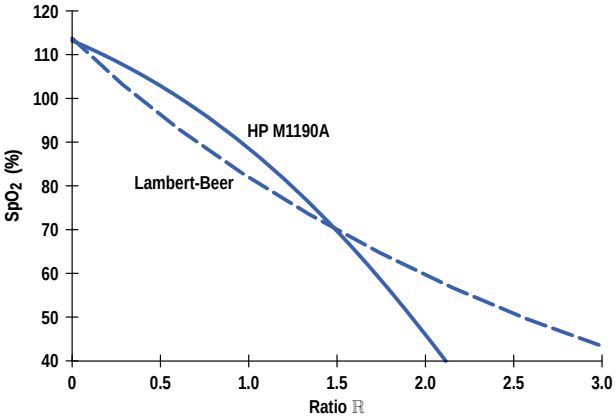


Fig. 9. *Theoretical (Lambert-Beer) and real calibration (arterial blood samples) curve for the HP M1190A adult sensor. The difference is mainly caused by scattering effects and nonideal light sources.*

Fig. 10 shows the sensor LED driver circuit and receiver circuit. The LEDs are driven in sequence at a repetition rate of 375 Hz in antiparallel fashion. At the photodiode the intensities arrive in the sequence red (R), infrared (IR) and dark. In the receiver circuit this signal is split into three paths: a red path, an infrared path, and a dark path. The dark intensity is subtracted from the red and infrared.

Fig. 11 shows the separated red and infrared patient signals with their I_{min} and I_{max} values caused by arterial pulsation, from which the ratio $\mathbb{R}$ can be calculated (equation 6).

Ambient Light and Electrical Noise

In a clinical environment, the sensor picks up ambient light and electromagnetic noise from various sources. The major source for ambient light is room illumination, typically fluorescent ceiling lamps, which have broad spectral bands with peaks at harmonics of the power-line frequency, 50 Hz or 60 Hz. Very often, electrical noise also comes from the power line and shows up as harmonics of the line frequency. Other well-known sources of large interfering electrical signals are the electrosurgery devices used in operating rooms, which can be very broadband.

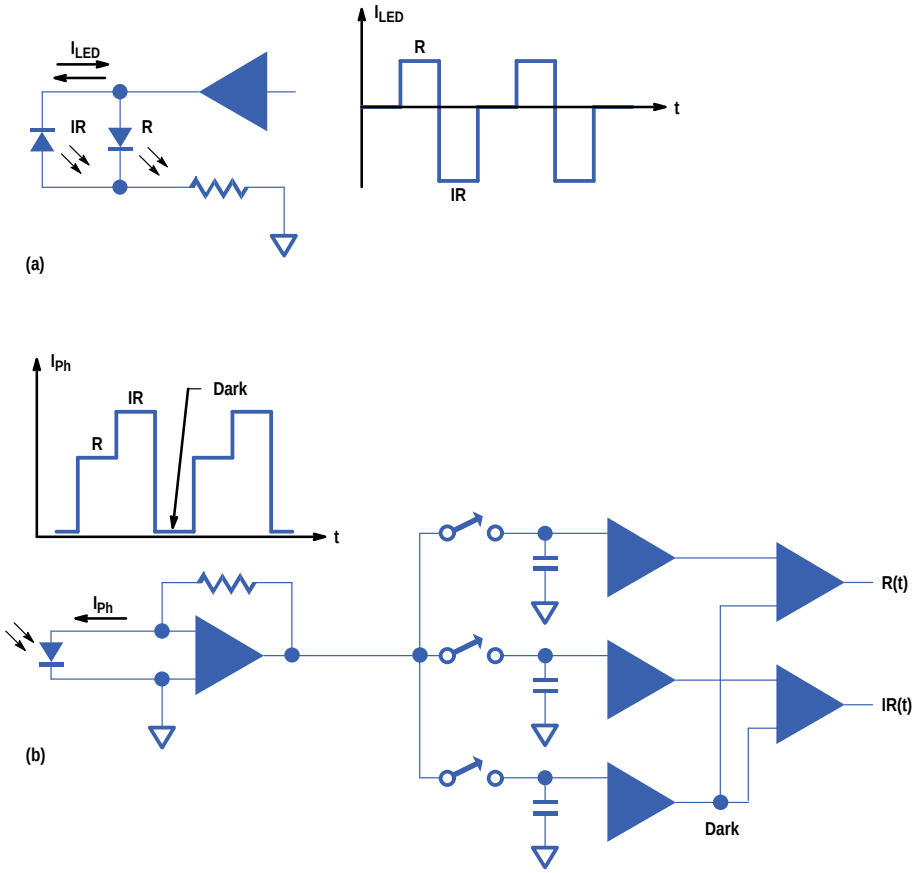


Fig. 10. *(a) Sensor LED driver circuit and (b) receiver circuit.*

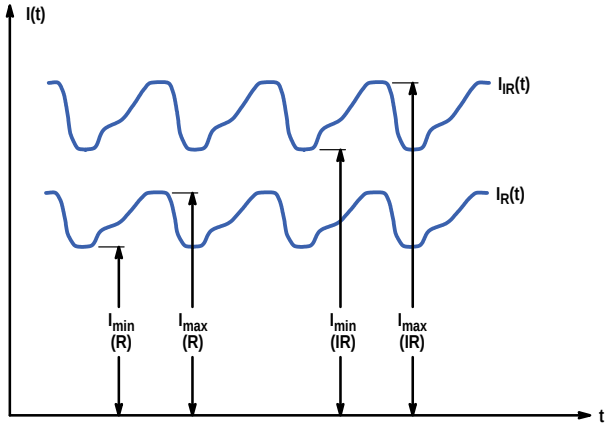


Fig. 11. *Separated red and infrared patient signals with their I_{min} and I_{max} values caused by arterial pulsation.*

Typical current levels at the sensor photodiode are around 1 μA dc with the blood current pulse modulated on the dc levels at a modulation depth of typically one percent. It is likely that the LED spectra including the desired signal and the optical or electrical noise spectra will overlap. Any noise lines in one of the LED modulation bands will be demodulated and folded down to the baseband, where they will contribute to poor signal-to-noise ratio (S/N). A very dangerous situation for the patient can occur in the monitoring of neonates, who are often treated with very bright UV lamps for bilirubin phototherapy. Neonates give poor SpO_2 signals because of poor vascular perfusion, so the bright UV ambient light can cause situations in which $S/N < 1$. A pulse oximeter is very likely to be misleading in these situations. It can derive values for pulse rate and oxygen saturation that are wrong because the input signals are dominated by noise.

Because interference can lead clinicians to apply incorrect care and therapy and cause harm or even death to patients, it must be avoided at all costs. A major goal for the sensor design was optimum optical and electrical shielding. Fig. 12 shows the pediatric sensor. Its closed housing is designed to shield the sensor from interfering ambient light.

Fig. 12. *The HP M1192A pediatric sensor has a closed housing to shield it from interfering ambient light.*

Movement Artifacts

Because the pulse oximetry method relies on the pulsatile part of the absorption, probably the most frequent cause of trouble is movement of the patient. Any movement usually causes movement of the sensor or the nonarterial tissue under the sensor and thereby leads to noise on the signals. A design goal for the new sensors was to be small and lightweight and to attach firmly to the patient. The cable was made as thin and flexible as possible consistent with the need for robustness, so that it adds little weight and stiffness, thereby helping to decouple the sensor from cable movements.

Cable Robustness

The clinical environment can be very harsh. Sensors fall off patients. People step on them and carts roll over them. Cables get squeezed between drawers and racks. The cables of medical sensors, in particular, have to be extremely robust. They are moved, bent, kinked, and treated with aggressive disinfectants.

A carefully selected lead composition and the use of nonbreakable material were goals for the cable construction. A new connector and interconnection concept are used. The interconnection is split into two parts: a short, thin, and more fragile cable is used with the sensors for low weight and minimum mechanical stiffness, while a longer, heavier, more robust cable was designed as an interface cable to the instrument.

The connector joining the cables (Fig. 13) is optimized for small size, low weight, and robustness. Special care was taken to provide very high insulation between the pins and to make the interconnect junction watertight to avoid leakage currents in humid environments like neonatal incubators. In older designs, saturated water vapor and salty residues from infusions or blood on connectors was a common source of problems, leading to erroneous measurement results.

Setting Design Goals

HP has offered a reusable SpO_2 sensor since 1988, but in one size only: the adult HP M1190A sensor. This sensor is very well-accepted. The objective for the new sensor project was to extend this sensor technology to a family of sensors covering all of the different application areas, so the customer is not forced to use a third-party sensor for application reasons.

(a) (b)

Fig. 13. *Plug and socket connector system.*

Based on experience with the HP M1190A sensor and on customer feedback we defined the following objectives:

- "Must" Objectives
  - Reusable sensors only
  - Cost competitive with disposable sensors
  - Clear, nonconfusing application
  - No burns on skin
  - State-of-the-art necrosis factor behavior (minimal local cell damage)
  - No penumbra effect
  - Influence of ESI (electrosurgery interference) as low as in HP M1190A
  - Backward compatibility with HP monitors (hardware)
- "Want" Objectives
  - Reliability equal to HP M1190A
  - Easy to use
  - Comfortable application over long period of time (several days)
  - Reliable fixing mechanism
  - Cleaning and sterilization by immersion in solutions
  - Mechanically robust design like HP M1190A
  - Cable size, length, flexibility, and quality similar to HP M1190A; alternatively, trunk cable and sensor cables
  - No influence of ambient light (operating room, bilirubin therapy, fluorescent lights)
  - Minimum motion artifacts
  - Backward compatibility with HP monitors (software)
  - Compatibility with competitive monitors.

Reusability was required because HP feels environmentally responsible for HP products. Most of the sensors on the market are disposable, which means that they are applied only once, after which they must be disposed of as medical waste. Reusable sensors are a small contribution to protecting the environment.

We used the Quality Function Deployment[3,4] (QFD) tool for developing these sensors. The starting point for QFD is the customer—what does the customer want? The customer requirements are weighted according to their relative importance, the corresponding engineering characteristics are listed, and step by step a matrix is built that provides the means for interfunctional planning and communication.

The three most important customer attributes we found are:

- Functionality. Minimize physiological effects like skin irritation and low perfusion. This means selecting the appropriate material and applying the appropriate clamping force.
- Performance. Ensure good signal quality. The most important issue was to select optical components to provide good light transmission.
- Regulations. The sensors had to meet U.S. FDA requirements and international safety and EMC standards.

We have had several clinical trials to verify that we understood the customer requirements correctly. At the release of the product for manufacturing we checked our solutions again to make certain that they are in accordance with the required customer attributes and engineering characteristics. We have been shipping the sensors for over half a year without any customer objections. This makes us fairly confident that the sensors meet customer expectations.

Design Concept

The next step after defining the project goals was to evolve the basic design concept. To reduce waste (even reusable parts have to be replaced eventually) we decided that each transducer would consist of two parts: an adapter cable to be used for all sensors and a sensor cable consisting of connector, transmitter, receiver, and a special sensor housing for the specific application site (finger of a child or small adult, foot or hand of a neonate, ear of an adult). We made this split since the lifetime of the adapter cable is longer (we estimated three times longer) than that of the sensor cable, which is much lighter in weight to reduce motion artifacts. A further advantage of the two-part design is the flexibility for future products to use the sensor without an adapter cable. The design required the development of a new 8-pin connector family.

To minimize the risk, because of the very good customer feedback for the existing adult sensor, we decided to change only the optical elements of the transducer.

The detailed design concept is shown in Fig. 14. The adapter cable is a shielded twisted-pair cable with four single conductors, a 12-contact male plug on the instrument side, and an 8-contact female connector on the sensor side. The sensor cable is a shielded twisted pair cable with two conductor pairs, an 8-contact male plug on the instrument side, a transducer consisting of transmitter and receiver molded in epoxy, and a special sensor housing.

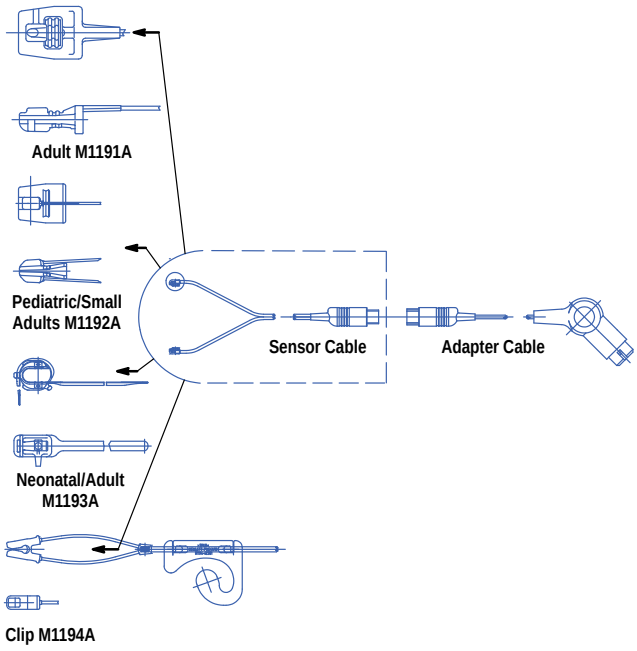


Fig. 14. *Design concept for the new sensor family.*

Housing

With the project goals in mind, the first proposals for the sensor housing were designed and prototype tooling was ordered to get parts ready for the first application tests. It was especially necessary to start with application tests as soon as possible for the neonate sensor, because this sensor would cover the biggest area and would be the most sensitive. The design of the pediatric sensor was more straightforward. It had to be similar to the existing adult sensor. For the other two sensors we approved a couple of proposals and ordered the prototype tooling for those.

With these samples we went into hospitals and spoke to nurses and medical technicians. When their response was positive, we began to improve the design step by step, making all changes in the prototype tooling as far as possible. If it was not possible to realize a necessary change, new prototype tooling was ordered. Only after this iterative process was complete did we order the final tooling.

The idea for the neonatal sensor, Fig. 15, was to place the transducer elements facing one another to make it easier to apply the sensor on foot or hand, and to have a long strap with a special fastener that allows application of the sensor on different foot or hand sizes. The transducer is positioned on the foot or the hand and the strap is threaded through the first latch and pulled slightly while holding the top of the transducer. The second latch is only used if the strap is too long.

The idea for the clip sensor was to integrate the spring for the necessary clamping force into the molded part (Fig. 16). The transducer is clipped onto the fleshy part of the earlobe. To minimize motion artifacts generated by patient movements a plastic fixing mechanism that hooks over the ear is provided.

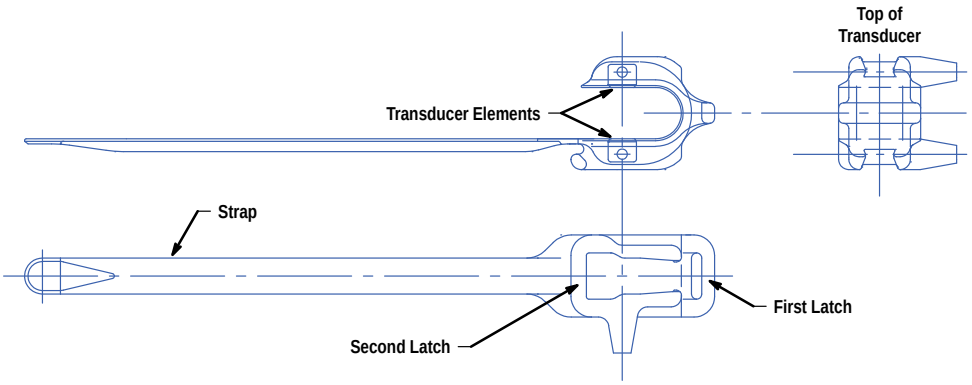


Fig. 15. *Neonatal sensor.*

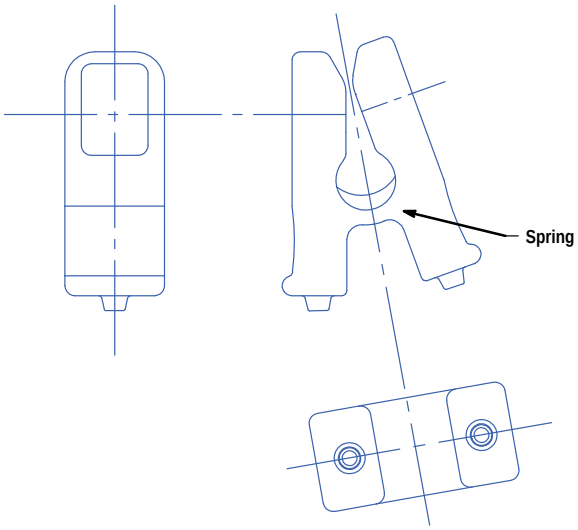


Fig. 16. *Clip sensor.*

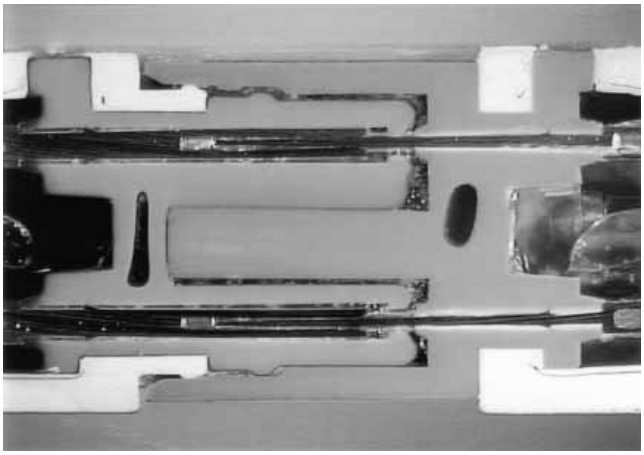


Fig. 17. *Cutaway view of two pins of the 8-pin connector between the adapter cable and the sensor cable. The connector is watertight when joined.*

Cable and Connector

Three different types of cables are used for the sensor family. For the adapter cable we use a very robust cable with an outer jacket made of polyurethane. The same adapter cable is used with all of the sensor types.

Two different sensor cables are used, one for the adult transducer and another for the rest of the family. They differ only in the outer jacket. For the adult sensor the outer jacket is made of silicone because of the manufacturing process. The sensor housing, which is made of silicone, is molded together with the cable and other elements in a molding machine. Because silicone can't be combined very well with different materials, the outer jacket must also be silicone.

For the rest of the sensor family we use a split, lightweight cable with an outer jacket made of polyurethane.

The construction of all three cables is similar. All are twisted-pair and have a Kevlar braid anchored in both the sensor and the connector to improve the strain relief.

The 8-pin connector between the sensor cable and the adapter cable also has a soft outer jacket made of polyurethane. The Kevlar braid is anchored inside the connector. Watertightness is achieved when the two halves of the connector are joined (see Fig. 17).

Optical Components

The optical elements are mounted on ceramic substrates shaped by cutting with a high-energy laser. The transmitter (Fig. 18) consists of two LED die (red and infrared) mounted on gold metallization. A photodiode on the receiver ceramic (Fig. 19) receives the sensor signal. A dome of epoxy material protects the elements and bond wires from mechanical stress. The wires of the transducer and the Kevlar braid are soldered and anchored on the backside of the ceramic.

To a first approximation, LEDs have a Gaussian intensity spectrum in which the peak wavelength is equal to the centroid wavelength. Because the red area (< 650 nm) of the extinction coefficients is very sensitive to wavelength variation (see Fig. 8) and the intensity distribution is not actually Gaussian and symmetrical, we use the centroid wavelength, which differs slightly from the peak wavelength, as an adequate characterization parameter for the LED (Fig. 20). Normally the

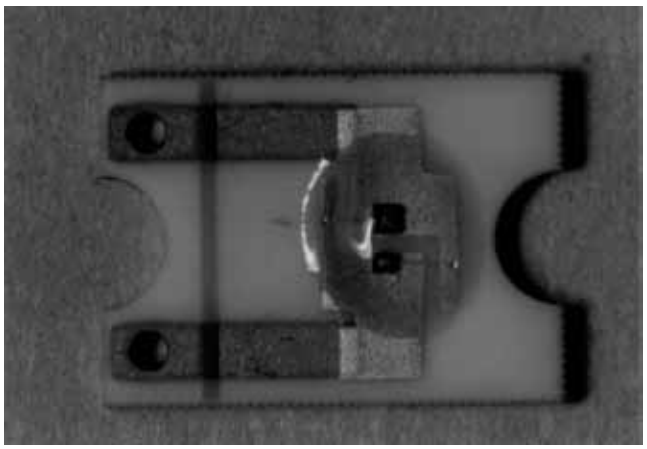


Fig. 18. *LED transmitter.*

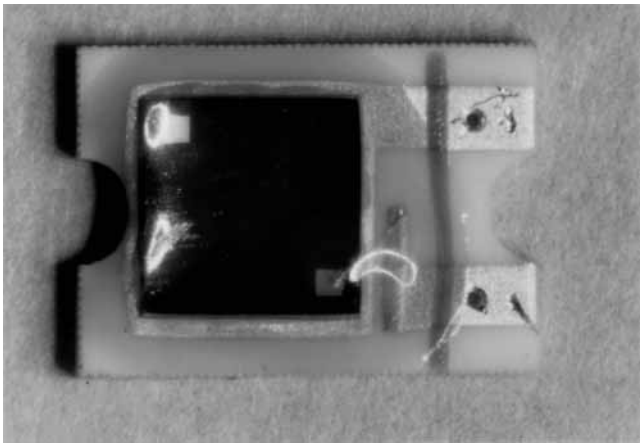


Fig. 19. *Photodiode receiver.*

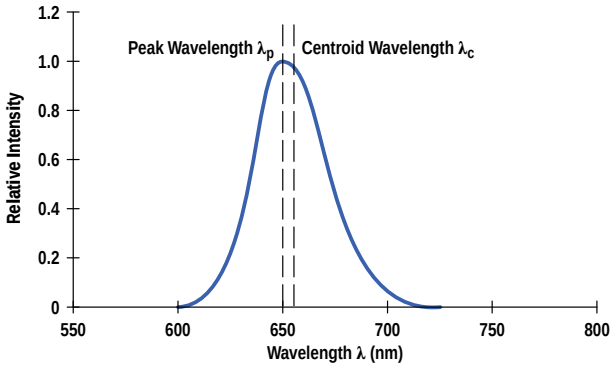


Fig. 20. *A typical LED intensity distribution. For SpO_2 measurements the centroid wavelength gives a better characterization than the peak wavelength.*

wavelength variation on a preselected wafer for red LEDs is in the range of ±5 nm. For the HP M1190A sensor in 1990, the HP Optoelectronics Division installed a selection process for a narrow, ±1-nm centroid wavelength variation.

For the new sensor family we chose for each sensor an LED pair with centroid wavelengths of 660 nm (red) and 890 nm (infrared). For the red LED a new high-efficiency AlGaAs technology was chosen. The maximum intensity for these LEDs is about a factor of four higher than for the older ones. This has the big advantage that the transmission values for both the red LEDs and the infrared LEDs are about the same. The average drive current for the LEDs, and therefore the heat dissipation, can be dramatically lowered.

The *transmission* Tr is defined as the ratio of photocurrent to LED current:

$$\mathrm{Tr} = \frac{I_{ph}}{I_{LED}}, \qquad (8)$$

where I_{ph} is in nanoamperes and I_{LED} is in milliamperes. Tr depends strongly on the absorption and extinction coefficients of the patient's tissue. Mean values are about 70 nA/mA over a large patient population. For thin absorbers like the earlobe, values of Tr as high as 300 nA/mA are possible. With new SpO_2 front-end hardware this would not have been a problem, but to be compatible with older pulse oximetry instruments we use a smaller active area of the photodiode for the HP 1194A ear sensor to get the same Tr values as the other sensors.

The LED supplier (not HP for the new sensors) guarantees a narrow centroid wavelength variation of less than ±2 nm. For LED qualification measurements, an optical spectrum analyzer with a wavelength resolution of 0.2 nm is used. All LED parameters are measured with a constant drive current of 20 mA. Because there is a wavelength shift over temperature of about 0.12 nm/K, the ambient temperature has to be held constant. Depending on the LED packaging, there is also a certain warmup time, which has to be held constant for LED qualification. In clinical practice, there can always be a temperature shift during SpO_2 measurements, but because of the definition of the ratio $\mathbb{R}$, with red intensity in the numerator and infrared intensity in the denominator (see equation 6), this effect is compensated within the specified operating temperature range of $15°C < T < 45°C$.

Another important factor is that some red LEDs have a low secondary emission (<4% of maximum intensity) at a wavelength of typically 800 to 850 nm (Fig. 21). For higher secondary intensities, interference with the infrared LED causes a ratio error and therefore an SpO_2 error, which must be eliminated. For the new high-efficiency LEDs the secondary emission is typically less than 0.1%.

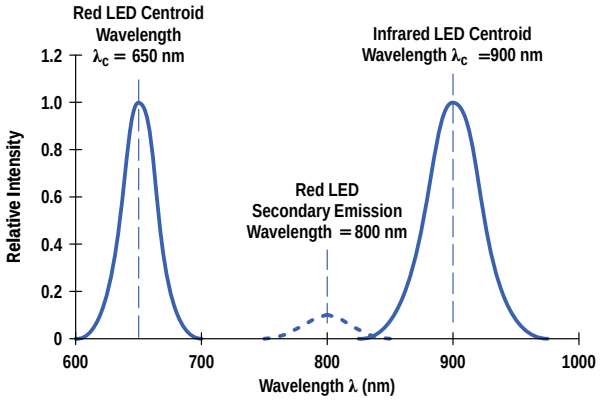


Fig. 21. *Typical red and infrared LED spectra for SpO_2 sensors. The spectral half-bandwidth for the red LED is about 20 nm and for the infrared LED about 40 nm. A secondary emission peak for the red LED is undesired and has to be lower than 4% of the maximum intensity.*

The receiver element is a standard silicon photodiode with peak sensitivity at 850 nm. The active area is approximately 2 mm square for the HP M1191/92/93A sensors and 1 mm square for the HP M1194A ear sensor. The die are mounted on a ceramic substrate with metalized layers for shielding.

The package for the LEDs in the HP M1190A sensor was a standard subminiature package. The emitter consisted of a red-infrared-red triplet in a longitudinal arrangement to make the apparent emission points for the red and infrared sources virtually identical. This is important for the ratio calculation, because both light paths have to be about the same length. One disadvantage is a possible malfunction when the patient's finger does not cover the entire light source. Then a part of the red light can cause an optical shunt that yields dc red levels that are too high (penumbra effect), causing false high readings. In the new sensor design, the two LEDs are very close together (<0.5 mm) on a common leadframe (see Fig. 22). This should eliminate the penumbra effect.

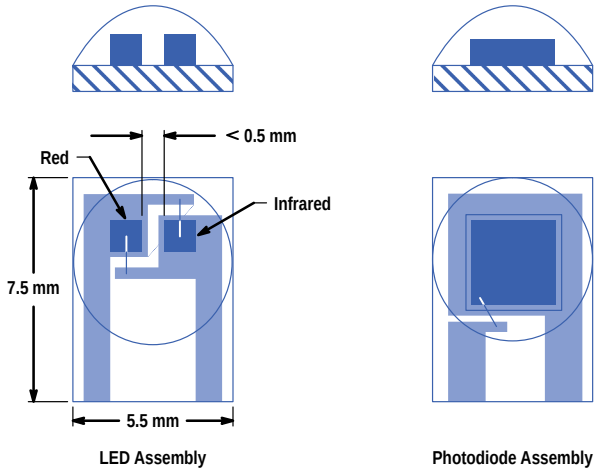


Fig. 22. *Transmitter and receiver assemblies for the new sensor family are on ceramic substrates. To avoid asymmetric optical shunting (penumbra effect) the two LED die are mounted as close as possible to each other. An epoxy coating is added before final packaging to protect the optical parts.*

The die are mounted on a ceramic substrate and covered with a transparent epoxy material. A design goal was to get a water and disinfectant resistant seal between the cable and the package. Immersion and disinfection tests show that this goal was achieved.

Materials

For the pediatric and neonatal sensors we chose silicone with a hardness of 35 ± 5 Shore A. The material is very robust and has good tensile strength compared to other silicones. Silicone is very often used in clinical areas and is very well-accepted. It is very resistant to chemicals and causes no skin irritations when used correctly.

For the clip sensor we chose a polyurethane with a hardness of 75 ± 5 Shore A, which gives the required clamping force (Fig. 23).

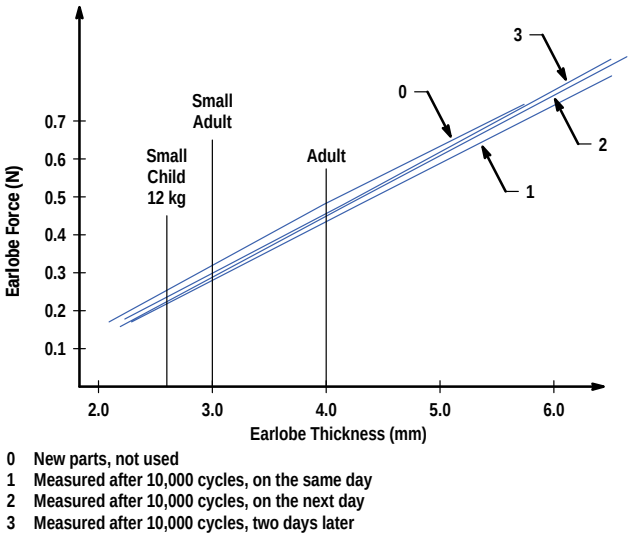


Fig. 23. *Spring forces in the clip sensor.*

Manufacturing Process

The manufacturing process for the new HP M1191A sensor is injection molding, the same as for the older HP M1190A. These sensors use only silicone rubber. For the HP M1192/ 93/94A sensors a different manufacturing process was necessary because these sensors use two different materials—silicon rubber and polyurethane, which do not combine well in the injection molding process. We also wanted to reduce the manufacturing costs and to gain more flexibility in choosing suppliers.

We decided to cast the premounted optical elements together with the cable in a special epoxy that combines very well with the cable including the Kevlar braid. We thus ensured watertightness, which means the sensors can be disinfected by immersion in solutions.

Reliability

To reach the reliability goals a few iterative changes were necessary and different tests installed. Many tests and customer visits were conducted to ensure that the sensors will not break. We tested several housing materials until we found the right one for the rough clinical environment. The tensile strength and robustness have been improved dramatically compared to the first samples. The method of anchoring the Kevlar braid in the ceramic substrate and connector was also improved several times. Every prototype was tested in the same way, by a combination of mechanical stress and cleaning by immersion in different solutions.

Technical Qualification

The most important factor for qualifying the new SpO_2 sensors has been how to determine test methods that are able to expose any weak points of the design. The qualification stress should be higher than the normal clinical application stress to provoke failures. The fulfillment of customer expectations concerning reliability was the overall guideline for prioritizing the test emphasis. Because of its customer orientation, the QFD methodology was an excellent tool for determining the main focus for testing. To make QFD more practical, we divided the sensor into three subelements, which made the specifics of the subassembly more visible. The three subelements were the interconnection, the sensor housings, and the optical assemblies.

The correlation matrix between the customer requirements and the technical specifications generated a relative importance ranking within the broad list of requested technical details. We could now determine which were the most important technical parameters. Their performance would have the greatest impact on the acceptance of the sensors in the market.

It was very important to assess the technical complexities and difficulties in the realization of technical specifications. This was the task of the engineers of a crossfunctional team chosen for their experience and ability to foresee potential problems. The correlation between expected technical difficulties and the importance of the parameters to the customer was an

essential input for further activities. We could now focus our efforts to reduce the risk potentials, which were clearly defined. High risk means high importance correlated with high technical difficulty ratings. These high-priority items were communicated to the project managers to give them an impression of the degree of technical maturity in this early project phase.

A critical assessment of design risk potential could now be made. This triggered a review of the importance of each customer requirement and gave the designers valuable inputs for design concepts. The results were also useful when considering strategies for accelerated stress testing.

The next step in the QFD process was to transfer the information on high-priority technical requirements into another matrix showing the relationship between parts characteristics and technical requirements. The key deliverables of this exercise were:

- Identification of key parts and their characteristics
- Preselection of parts characteristics to find critical parts for performing a design failure mode and effect analysis (FMEA)
- Information to aid in selecting between design alternatives to find the most competitive design concepts
- Inputs for stress testing using parts characteristic importance information.

The FMEA generates risk priority numbers (RPN). These numbers describe how often a failure will be occur, how easily it will be detected, and how severe the failure will be. Taking the interconnection as an example, the risk assessment was divided into three categories:

- High Risk: RPN > 200 and high parts importance
- Medium Risk: RPN > 100 and high parts importance
- Low Risk: RPN > 100 and low parts importance.

In this way, key customer needs were identified and test parameters selected. We also took into account the feedback from clinical trials.

Fig. 24 gives an overview of the qualification tests that were performed to get release approval for the sensors. A special machine was designed to simulate the cable stress that occurs in hospitals. We call this test the bending/torsion test. With a calculated number of cycles, equivalent to our reliability goals, we stressed the critical cable sections to ensure that the lifetime requirements were met.

Supplier Selection

The supplier chosen to manufacture the new SpO_2 sensor family had to meet a number of specific requirements. The supplier is responsible for the majority of the manufacturing process steps. This has a positive influence on production lead time, logistics, communication, and costs. To reach our quality goals with one supplier who is responsible for nearly all process steps is much easier than with a long chain of suppliers. The requirements covered technology, vertical integration, and costs.

Fourteen international suppliers were evaluated. Nine were not able to manufacture the sensors because they did not have the required technology. After considering cost aspects, only two suppliers fulfilled the selection criteria. For these two suppliers, we constructed supplier profiles derived from the QFD method.

To construct a profile, each customer need is listed along with an evaluation of how well the supplier fulfills that need in terms of technology and processes. The level of fulfillment is evaluated by an HP specialist team, which also evaluates the importance of each customer need. The profile shows the supplier's strengths and weaknesses and gives a point score. The supplier with the higher number of points is considered better qualified to manufacture these products.

To evaluate critical technology and processes, design and process failure mode and effect analyses (FMEAs) were conducted for both suppliers' products. To evaluate each manufacturer's capabilities, a quality and process audit was performed at the manufacturing site. The auditors reviewed the site and manufacturing processes for comparable products that were identified as critical for our sensor products.

Production Wavelength Measurements

The measurement of LEDs for the SpO_2 sensors at the manufacturing site is a critical and sensitive manufacturing process step. To guarantee the accuracy of HP SpO_2 measurements the wavelength of the red LED has to be within a very small range: between 657 and 661 nm. To measure the LED wavelength a very accurate optical spectrometer is used. To obtain repeatable measurement results, an integrating sphere is used to couple the light of the red LED into the spectrometer (see Fig. 25).

An integrating sphere is a ball with a highly reflective surface. The light is reflected many times on the surface and becomes diffuse. As a result, the spectrum and the intensity of an LED are the same at each point of the surface of the ball and can be coupled easily into the spectrometer. The main advantage of this method is that tolerances in the placement of the LED are

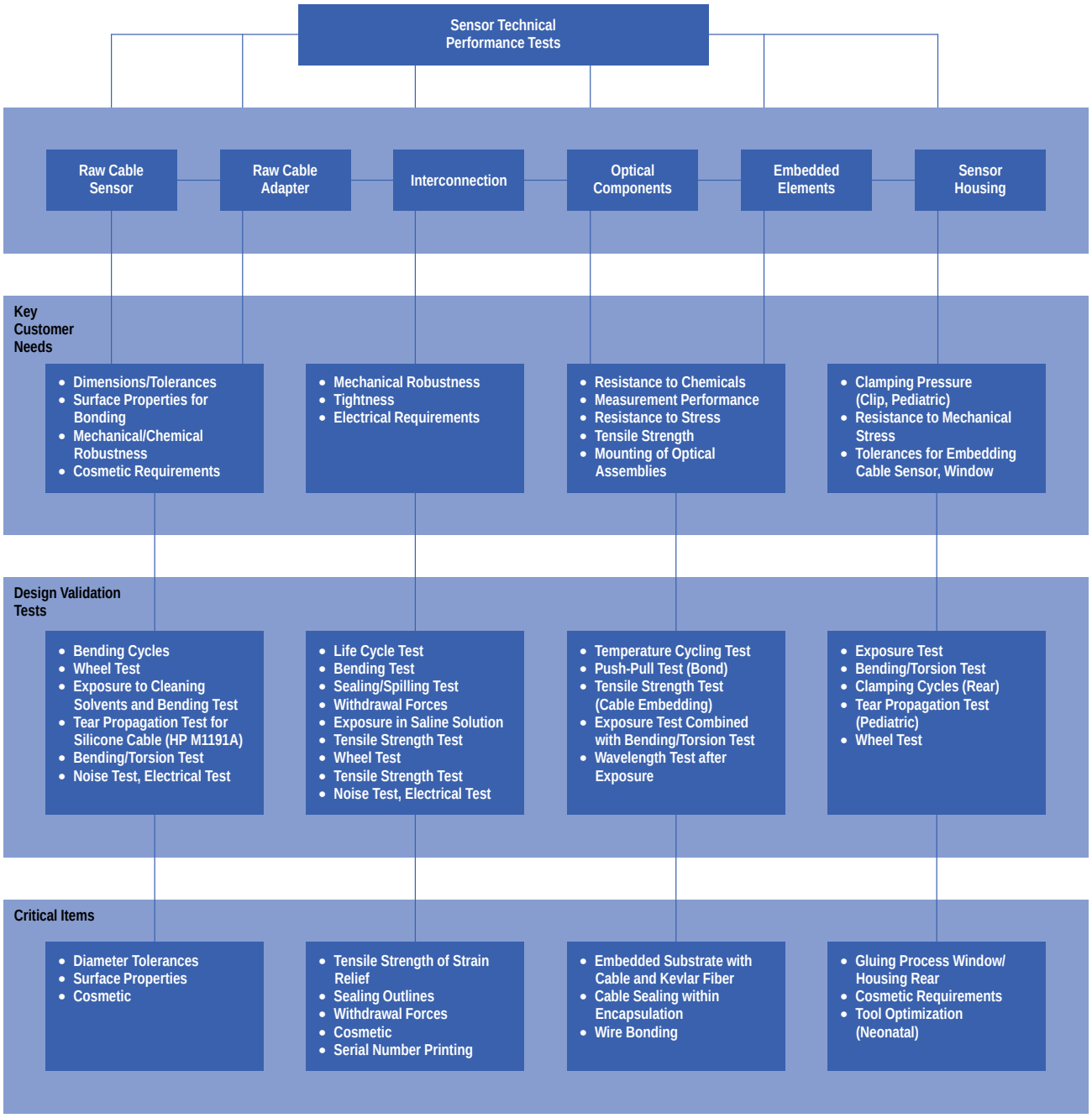


Fig. 24. *Qualification tests for the new sensor family.*

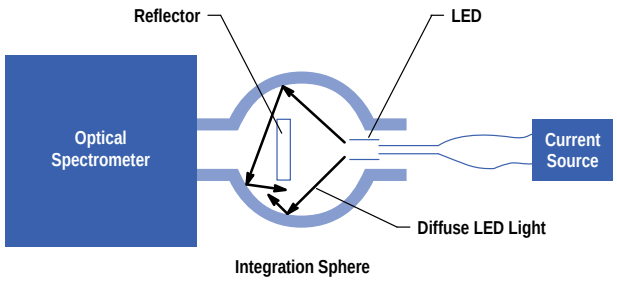


Fig. 25. *Setup for LED spectral measurements.*

not critical and the repeatability is very good compared to other methods. Fig. 26 shows a typical spectrum of a red LED measured with an integration sphere.

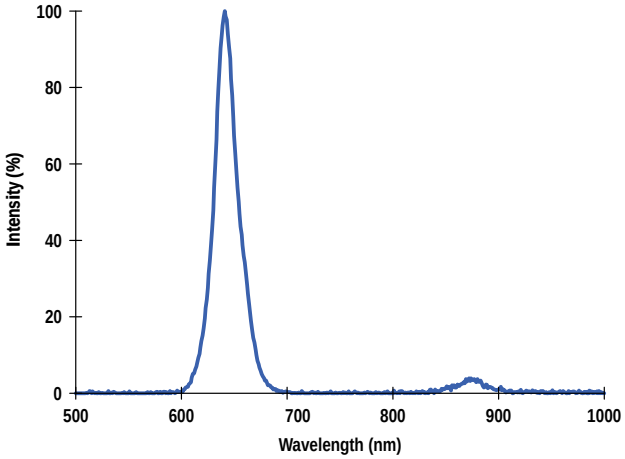


Fig. 26. *Spectrum of a red LED measured with an integration sphere.*

There are different ways to measure the wavelength of an LED. One is the peak wavelength, which is the highest point of the spectrum. The centroid wavelength, which is used in our measurements, calculates the center of the area under the spectrum. A secondary peak in the spectrum of the LED can have a large influence on the measurement results and has to be very small ($< 1\%$).

The temperature of the LED die has a large influence on the emitted wavelength—the higher the temperature the higher the wavelength (0.12 nm/K). Therefore, the LED must be in thermal equilibrium. In practice, the LED takes only a few seconds to reach thermal equilibrium. The ambient temperature must be monitored and if the temperature changes the spectrometer must be recalibrated.

Summary

A new family of reusable pulse oximetry sensors has been developed. Based on the HP M1190A, HP's first reusable SpO_2 sensor, these sensors can noninvasively monitor the blood oxygen levels of patients, a key vital sign. They are used primarily in operating rooms, recovery rooms, intensive-care units, and some general wards. The new sensor family covers all application areas and consists of the M1194A clip sensor (Fig. 27), the HP M1191A adult sensor with new wavelength (Fig. 28), the HP M1192A pediatric sensor (Fig. 12), and the HP M1193A neonatal sensor (Fig. 29).

Acknowledgments

Many people were involved in this project. The authors would especially like to thank Dietrich Rogler for the industrial design of the sensors, Willi Keim and Peter Jansen of materials engineering for their excellent support, Martin Guenther for performing all the optical characteristics measurements, Gerhard Klamser for verifying the algorithm, Gerhard Lenke for organizing all the regulation tasks, and Otto Gentner for managing the clinical trials. Special thanks to Professor Dr. J. W. Severinghaus of the University of California Hospital in San Francisco for performing volunteer studies.

Fig. 27. *HP M1194A clip sensor.*

Fig. 28. *HP M1191A adult sensor.*

Fig. 29. *HP M1193A neonatal sensor.*

References

1. T.J. Hayes and E.B. Merrick, "Continuous, Non-Invasive Measurements of Blood Oxygen Levels," *Hewlett-Packard Journal*, Vol. 28, no. 2, October 1976, pp. 2-10.
2. *Hewlett-Packard Journal*, Vol. 42, no. 4, October 1991, pp. 6-54.
3. D. Clausing, "The House of Quality," *Business Review*, May-June 1988.
4. L.P. Sullivan, "Quality Function Deployment," *Quality Progress*, June 1986, pp. 39-50.

Volunteer Study for Sensor Calibration

To calibrate the new SpO_2 sensor family it was necessary to adjust the relationship between the ratio measurements and the SpO_2 values using data based on real blood samples from volunteers.

Fig. 1 shows the measurement environment for the calibration study. The basic instrument is a special HP Component Monitoring System (CMS) with 16 SpO_2 channels. Sixteen sensors at different application sites could be used simultaneously. To get SpO_2 values over the entire specification range of $70\% < SpO_2 < 100\%$, the volunteers got air-nitrogen mixtures with lowered oxygen levels—less than 21%.

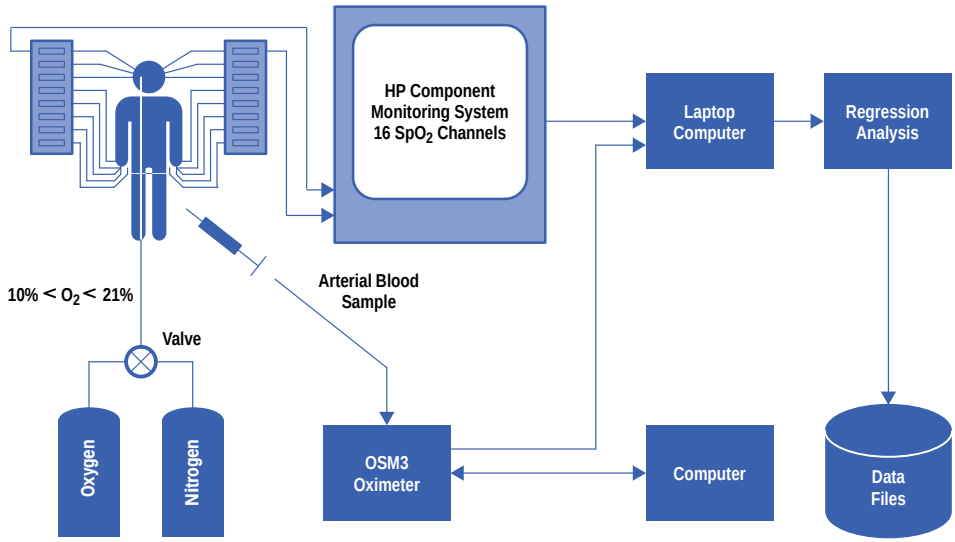


Fig. 1. *Sensor calibration using volunteers and SpO_2 data acquisition by a special HP Component Monitoring System (CMS) with a maximum of 16 channels. Different SpO_2 values are achieved by supplying different mixtures of oxygen and nitrogen. Arterial blood samples are analyzed by a Radiometer OSM3 oximeter. For each sensor and application site, regression analysis is done, and calibrating tables are derived from the results.*

Because of his great experience with such studies we used the method developed by Dr. J.W. Severinghaus of the University of California in San Francisco. For each volunteer a maximum of 16 sensors were applied at the fingers, earlobes, and nostrils. A catheter was placed in the left radial artery. Arterial O_2 saturation was reduced rapidly by a few breaths of 100% N_2. This was followed by a mixture of air and N_2 with about 4% CO_2 added while the subject voluntarily hyperventilated to speed the attainment of an alveolar gas hypoxic plateau and to provide end tidal samples for regression analysis. FiO_2 was adjusted to obtain plateaus for 30 to 60 seconds at different SpO_2 levels (Fig. 2). At the end of each plateau a 2-ml arterial blood sample was obtained and analyzed by a Radiometer OSM3 multiwavelength oximeter.

The regression analysis yielded three SpO_2-versus-ratio calibration curves: one for the HP M1190A adult sensor, a second for the HP M1191A adult sensor, the HP M1192A pediatric sensor, and the HP M1193A neonatal sensor, and a third for the HP M1194A ear sensor. The curves for the HP M1190A and M1191A are different because of their different LED wavelengths, while for the ear sensor the application site is different— the tissue constitution of the earlobe and nostril seems to be optically very different from the other application sites. Each calibration curve is the best least squares fit to the data points of a second-order polynomial.

Fig. 3 shows the good correlation with the reference ($R^2 = 0.95$) in the case of the HP M1191A adult sensor. Fig. 4 shows that the specified SpO_2 accuracy is reached within the range of $70\% < SpO_2 < 100\%$. Fig. 5 shows that the correlation for the HP M1194A ear sensor is not as good as for the HP M1191A. The data point distribution is also wider (Fig. 6). This is caused by a much poorer signal quality at the earlobe than at the finger. In general the perfusion index for the ear is only about a tenth of that for the finger. Therefore, in normal circumstances the preferred application site is the finger. In some cases, such as centralization (i.e., shock patients), the earlobe sometimes gives better results.

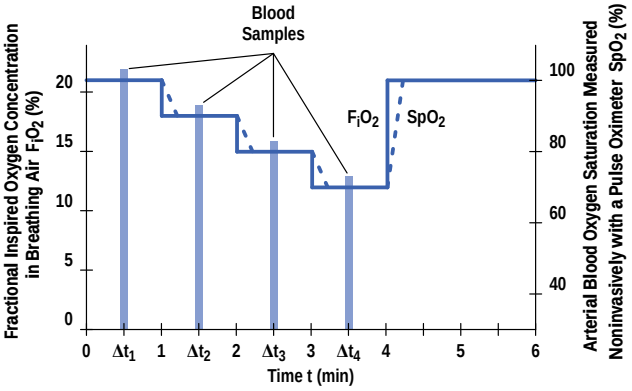


Fig. 2. *Stepwise desaturation by lowering oxygen levels leads to quasistable SpO_2 levels. This condition gives blood samples with correct blood gas values. The delay for the SpO_2 values compared to the oxygen values comes from the circulation time for the arterial blood from the lungs to the arm. Calibration tables are compiled by comparing the known SpO_2 values with the ratio data measured by the CMS.*

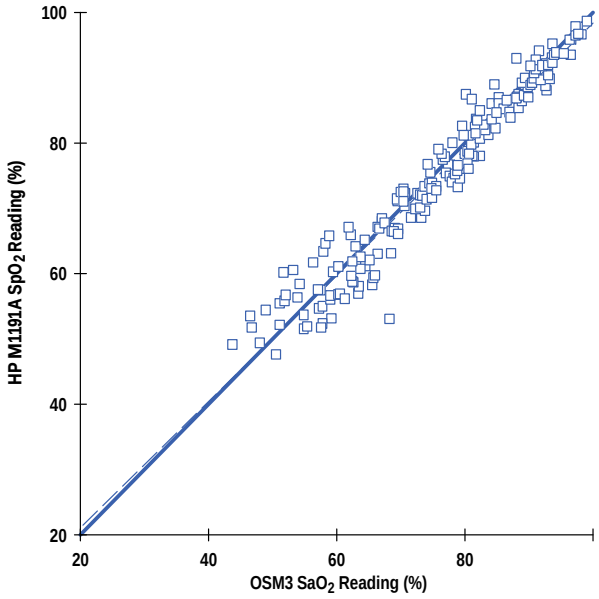


Fig. 3. *Regression analysis for HP M1191A adult sensor SpO_2 measurements after calibration. The measurements are plotted against arterial blood SaO_2 measurements from the OSM3 oximeter. The data (206 points) is from 12 volunteers with different oxygen saturation levels.*

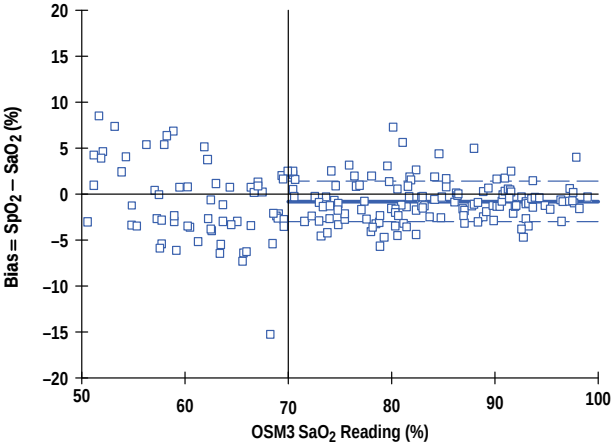


Fig. 4. *Bias and standard deviation for the HP M1191A over the specified range of 70% < SpO_2 < 100%.*

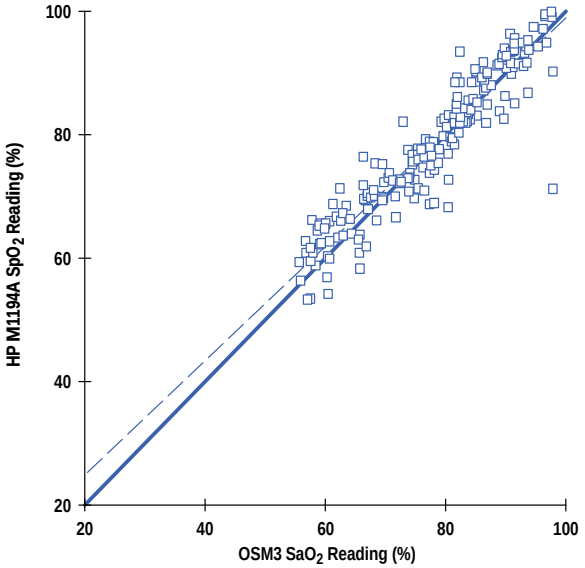


Fig. 5. *Regression analysis for HP M1194A ear sensor SpO_2 measurements after calibration. The measurements are plotted against arterial blood SaO_2 measurements from the OSM3 oximeter for 12 volunteers.*

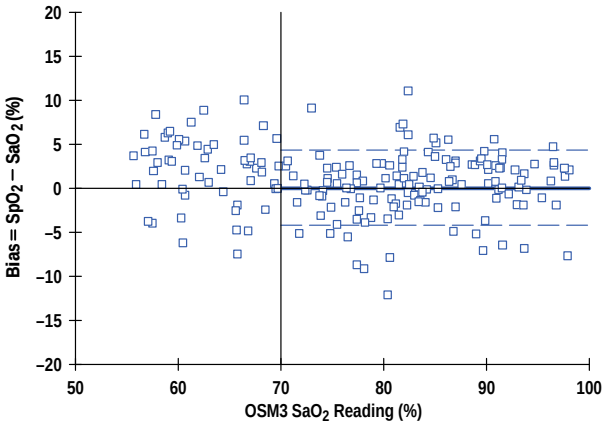


Fig. 6. *Bias and standard deviation for the HP M1194A. The standard deviation is larger than for the HP M1191A because of the smaller perfusion values in the ear.*

Neonatal Sensor Clinical Validation

In contrast to the volunteer study with adult subjects (page 1), a validation for the HP M1193A neonatal sensor had to be done with neonates in a clinical environment. Because blood sampling is very critical for sick neonates, only when an arterial line was already in place for therapy could we get blood sample values. Fig. 1 shows the regression line for 290 data points from 20 subjects. The correlation ($R^2 = 0.91$) is good considering that neonates often have oxygen saturation states that are unstable and changing rapidly. To eliminate these uncertainties, SpO_2 values with big differences before and after blood sampling ($\Delta SpO_2 > 5\%$) and with poor signal quality (perfusion index < 0.2) were not included. Fig. 2 shows that the specified accuracy of 3% SpO_2 standard deviation for the range $70\% < SpO_2 < 100\%$ has been reached for the HP M1193A sensor based on the clinical data from neonates.

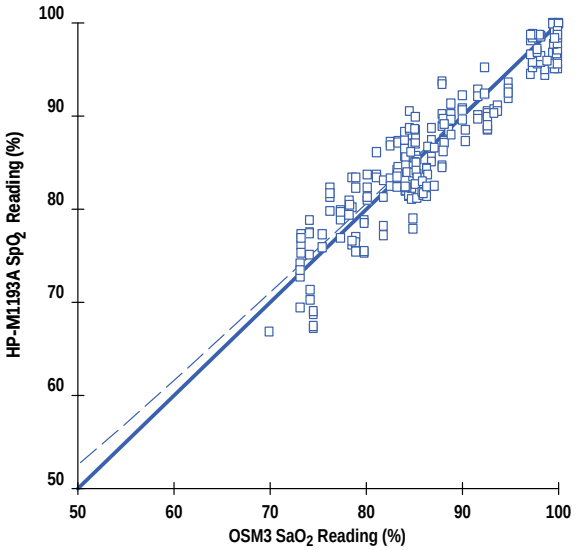


Fig. 1. *Regression analysis with data from clinical trials with the HP M1193A neonatal sensor. The 290 data points are derived from 20 subjects who already had an arterial line for blood sampling. The arterial SaO_2 values were measured by an OSM3 oximeter.*

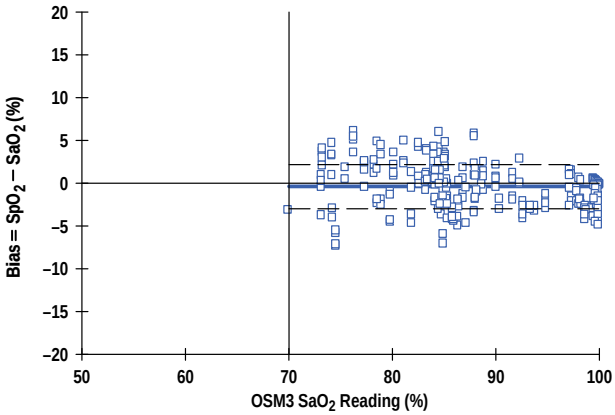


Fig. 2. *Bias and standard deviation for the HP M1193A neonatal sensor within the specification range of $70\% < SpO_2 < 100\%$, based on data from 20 neonates.*

Design of a 600-Pixel-per-Inch, 30-Bit Color Scanner

Simply sampling an image at higher resolution will not give the results a customer expects. Other optical parameters such as image sharpness, signal-to-noise ratio, and dark voltage correction must improve to see the benefits of 600 pixels per inch.

by Steven L. Webb, Kevin J. Youngers, Michael J. Steinle, and Joe A. Eccher

The objective of a scanner is to digitize exactly what is on the document that is being scanned. To do this perfectly would require a CCD (charge coupled device) detector with an infinite number of pixels and a lens with a modulation transfer function of 1.0, which does not exist. Modulation transfer function, or MTF, is a measure of the resolving power or image sharpness of the optical system. It is analogous to a visual test that an optometrist would use to measure a human eye's resolving power.

In the real world, the scanner user does not require a perfect reproduction of the original because the human eye does not have infinite resolving power. However, as originals are enlarged and as printers are able to print finer detail, the imaging requirements of the scanner are increased.

The HP ScanJet 3c/4c scanner, Fig. 1, is designed to obtain very finely detailed images for a variety of color and black and white documents and three-dimensional objects that are typically scanned. Its optical resolution is 600 pixels per inch, compared to 400 pixels per inch for the earlier HP ScanJet IIc. It produces 30-bit color scans compared to the ScanJet IIc's 24-bit scans, and its scanning speed is faster. The ScanJet 3c and 4c differ only in the software supplied with them.

Fig. 1. *HP ScanJet 4c 600-dpi, 30-bit color scanner.*

Optical Design

The HP ScanJet 3c/4c optical system is similar to that of the HP ScanJet IIc scanner,[1] with improvements to increase the optical resolution to 600 pixels per inch. Just sampling an image at higher resolution will not give the results a customer expects. Other optical parameters, such as MTF (i.e., image sharpness), signal-to-noise ratio, and dark voltage correction must improve to see the benefits of 600 pixels per inch.

The major optical components are:

- Two laminated dichroic composite assemblies used for color separation
- A fluorescent lamp with a custom mixture of phosphors
- A six-element double Gauss lens
- A three-row CCD sensor that has 5400 pixels per row
- Four front-surface mirrors.

The color separator composites, double Gauss lens, and CCD are shown in Fig. 2.

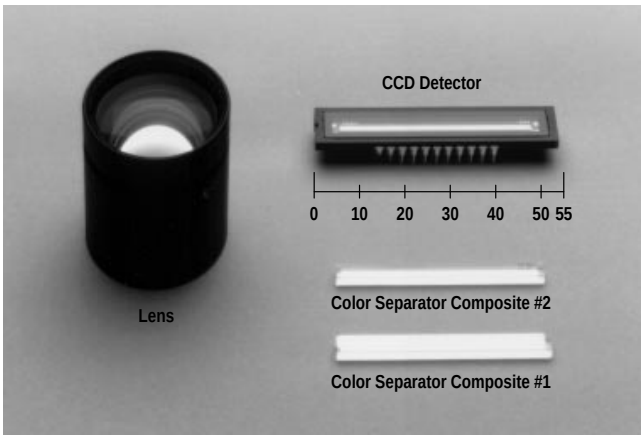


Fig. 2. *Lens, CCD (charge-coupled device) detector, and color separator composites.*

The color separation system (Fig. 3) consists of the two dichroic assemblies and the three-sensor-row CCD. With this method, red, green, and blue are scanned simultaneously, so only one pass is needed to scan all three colors. Each dichroic assembly is constructed of three glass plates that are bonded to each other with a thin layer of optical adhesive. Red, green, and blue reflective dichroic coatings are deposited onto the glass before lamination. The order of the coatings is reversed for the second dichroic assembly. The thickness of the glass plates between the color coatings and the flatness, tilt, and alignment are precisely controlled to ensure accurate color separation and image sharpness.

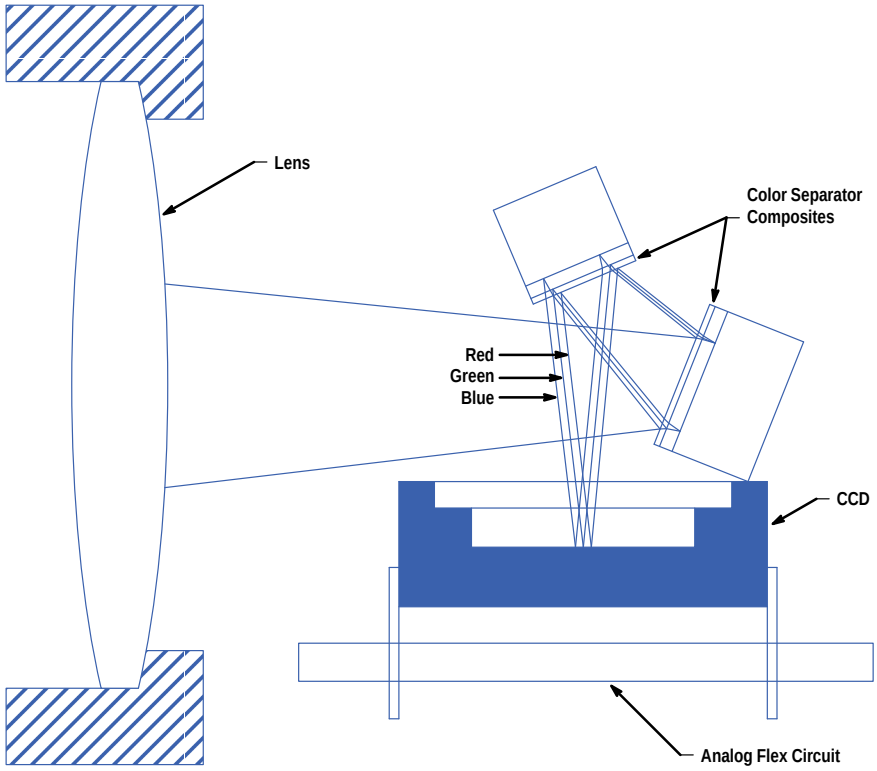


Fig. 3. *The color separation method uses two dichroic assemblies (composites) and a three-row CCD.*

Each color component is focused onto a CCD sensor row consisting of 5100 imaging pixels. Additional pixels are used for closed-loop dynamic light control, dark voltage correction, and reference mark location. By having all three rows integrated onto a single silicon chip, precise distances between the three rows are obtained. Production consistency is guaranteed by the integrated circuit process. Each CCD pixel generates a voltage signal that is proportional to the amount of light focused onto each pixel. The signal for each pixel is then processed and digitized. This data is sent to a computer or a printer.

Focus Optimization for Each Color

Two dichroic assemblies are used to equalize the path lengths of the three colors. A six-element double Gauss lens is used to focus the light onto the CCD sensors. However, the variation of the index of refraction of glass as a function of wavelength causes two of the three colors to obtain optimum focus at different locations. This phenomenon of differential refraction caused by wavelength dependence is best demonstrated by holding a prism up to a white light source and observing the colors. The light spectrum is separated because the shorter wavelengths (blue) are refracted or bent more than the longer wavelengths (red). Since lenses are made of glass that refract light of varying wavelengths at different angles, it is difficult to have all three colors focus at the same location.

To achieve simultaneous focus for all three colors there are several possible solutions. One is to design the focusing optics with curved front-surface mirrors only. However, these systems can be expensive, and it can be hard to correct other optical aberrations and difficult to image enough light onto the CCD. Another possible solution is to use an achromatic doublet. However, this type of lens can minimize chromatic aberration for only two of the three colors.

The ScanJet 3c/4c scanner optical design minimizes the chromatic aberration caused by the lens. An uncorrected optical system is shown in Fig. 4a, and a corrected optical system is shown in Fig. 4b. Lens chromatic aberration is corrected by adjusting the thickness of the dichroic coated plates. The path length of each color is adjusted to obtain optimum focus.

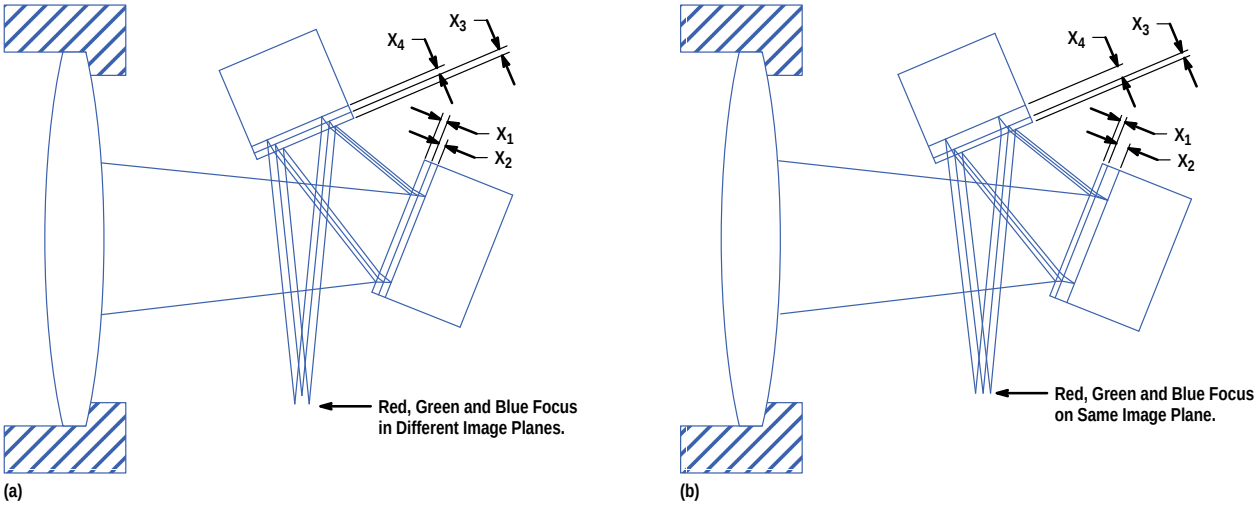


Fig. 4. *(a) Chromatic aberration of an uncorrected system. $X_1 = X_2 = X_3 = X_4$. (b) To ensure that simultaneous focus for red, green, and blue is achieved in the HP ScanJet 3c/4c scanner, unequal path lengths are used to compensate for the chromatic aberration of the lens. X1 = X3 and X2 = X4, but X2 ≠ X1.*

Unequal path lengths for red, green, and blue would cause color registration error across the scan region. To prevent this, the CCD sensor row lengths are adjusted as shown in Fig. 5. Each row has the same number of pixels. However, the center-to-center spacing (pixel pitch) is slightly larger for a small number of pixels in rows 1 and 3. The pixels with slightly larger pitch are strategically placed to correct for the lateral chromatic aberration of the lens. This eliminates any color registration error that would have been caused by the lens.

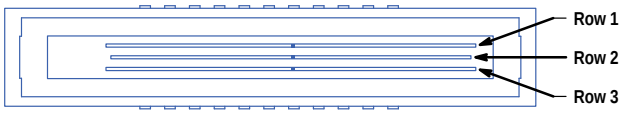


Fig. 5. *CCD row lengths are adjusted to compensate for color separator plate thicknesses.*

Optical System Layout

The lamp, lens, mirrors, color separators, and CCD are mounted into an aluminum carriage that is translated or scanned along the length of the document. The carriage is pulled underneath the glass platen by a belt connected to a stepper motor. The optical layout is shown in Figs. 6 and 7. Fig. 6 shows the mechanical design model of the carriage and light path. Fig. 7 shows part of the light path in more detail.

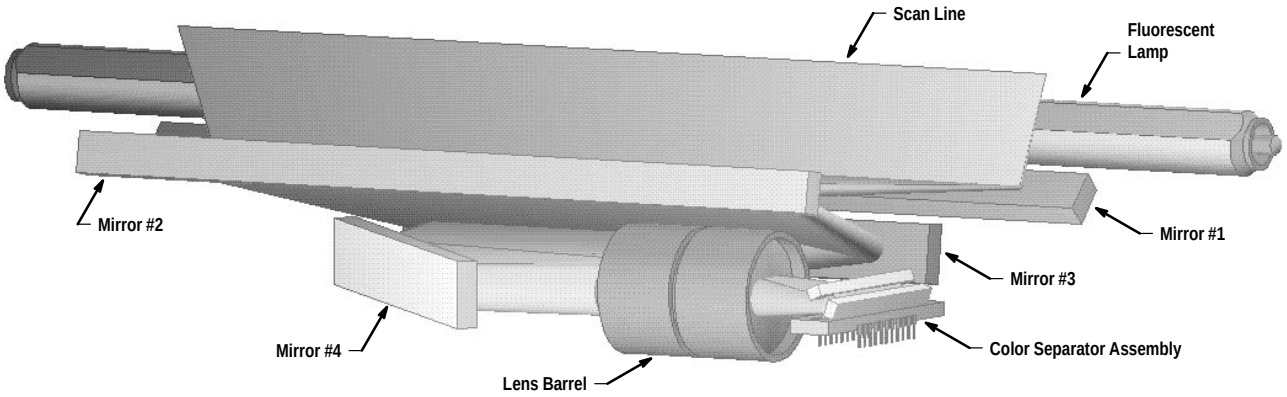


Fig. 6. *Mechanical design layout of the HP ScanJet 3c/4c optical path. The light path is from the scan line to mirror #1 to mirror #2 to mirror #3 to mirror #4 to the lens to the color separator to the CCD detector.*

The optical system was designed and evaluated using a commercially available optical design program. The sensitivity of optical tolerances such as lens centering, radii, thickness, and index of refraction were evaluated to determine the effects on image quality. The manufacturing assembly and mounting tolerances of key optical components in the carriage assembly were also evaluated. Image quality parameters such as MTF, color registration error, illumination uniformity, and distortion were emphasized.

To achieve precise optical alignment, custom assembly tooling was designed and implemented to meet production goals.

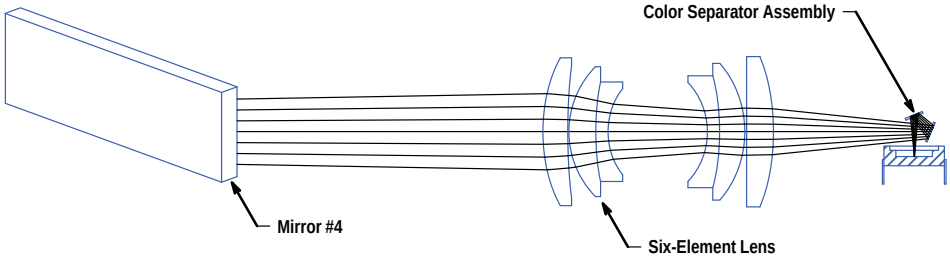


Fig. 7. *Ray trace of the optical path from mirror #4 to the color separator assembly (one color only).*

Fluorescent Lamp Driver

The fluorescent lamp is driven by a circuit that allows the lamp current to be varied over a range of 90 to 425 milliamperes. Since the lamp output is proportional to current, the lamp intensity is also varied.

A block diagram of the lamp driver circuit is shown in Fig. 8. The control inputs to the circuit provide the following functions:

- PREHEAT_L allows the filaments to be heated before the lamp is ignited.
- LAMP_PWM provides a pulse width modulated signal to set the desired current level.
- LAMPON_L turns the lamp on.

The filaments of the lamp are preheated for one second before lamp turn-on to reduce the amount of filament material that gets deposited on the insides of the glass. The deposits reduce light output, causing the light level to drop off near the ends of the lamp. This could create a lamp profile problem if preheating were not implemented.

The LAMP_PWM signal provides the desired current level plus a sync signal to the oscillator. The switching of the lamp driver power transistors occurs while the CCD (charged coupled device) is being reset. This helps keep switching noise from contaminating the CCD measurements. The lamp current command is derived from LAMP_PWM via the low-pass filter. The output of the low-pass filter is a voltage proportional to the amount of current desired.

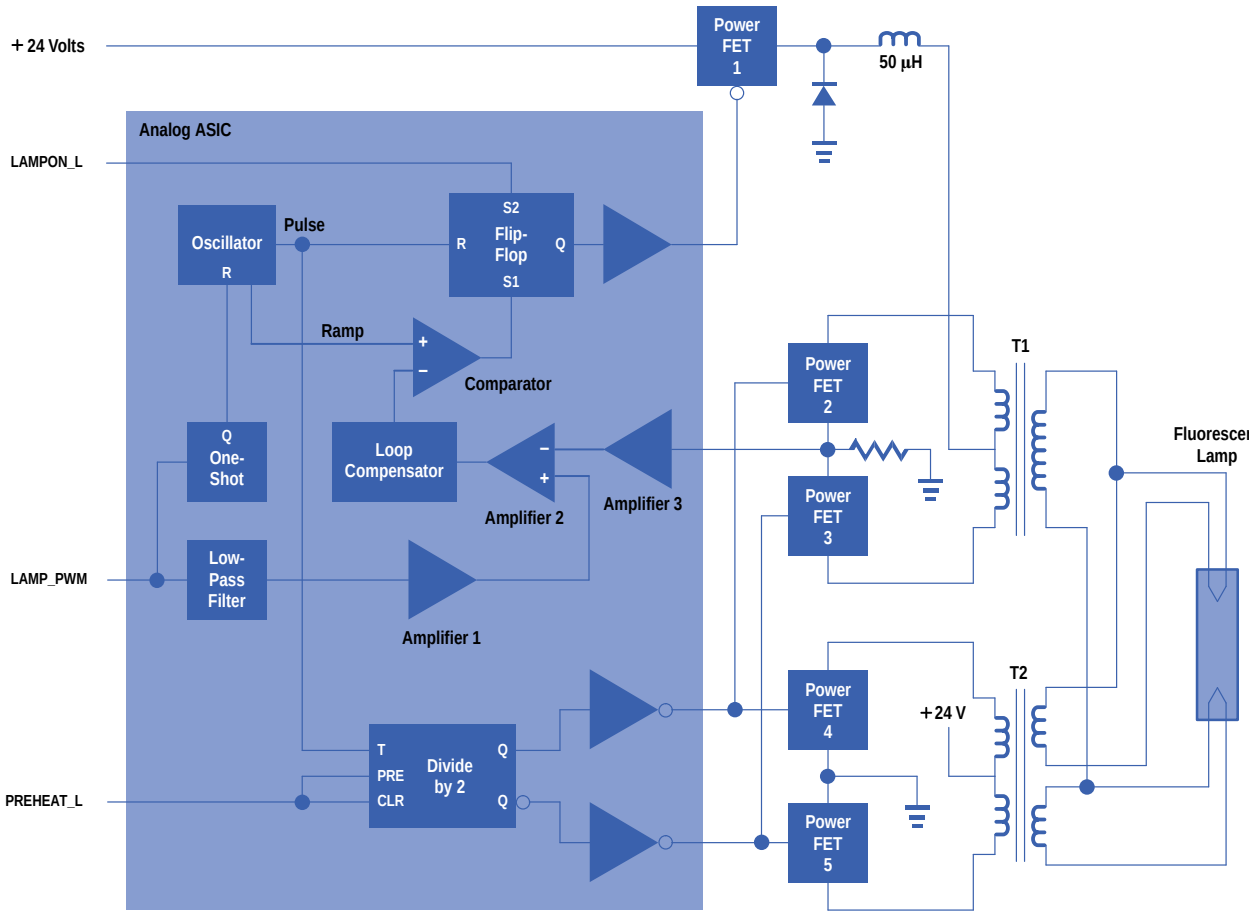


Fig. 8. *Block diagram of the fluorescent lamp driver.*

The LAMPON_L signal holds the flip-flop in the set mode until it is time to turn the lamp on. When the flip-flop is set, power FET 1 is held off via the buffer.

Operation of the lamp driver begins by taking PREHEAT_L to a logic zero. This allows the divide-by-2 circuit to begin toggling. When PREHEAT_L is high, both Q and $\overline{Q}$ are high, which turns off power FETs 4 and 5 via the inverting buffers. The toggling of the divide-by-2 circuit drives power FETs 4 and 5 out of phase. This provides a 24-volt square wave on the primary of T1 which is stepped down to 3.6V to drive the filaments. When LAMPON_L is activated, the flip-flop is reset on the next LAMP_PWM pulse, turning on power FET 1. The lamp appears as a high impedance in the off state, which results in power FETs 2 and 3 avalanching as a result of collapsing magnetic fields. The avalanche voltage of the power FETs is approximately 120 volts, half of which, or 60V, appears at the center tap of T1. This voltage is multiplied by the 1:6 turns ratio of T1 to produce 360V across the lamp. This voltage starts the lamp and the voltage drops to the low forty volt range. Current now flowing in the lamp is reflected back to the primary, where it is sensed. Amplifier 3 amplifies the voltage across the sense resistor and amplifier 2 subtracts it from the current command (output of amplifier 1).

The output of amplifier 2 is passed through the loop compensator (proportional plus integral) and applied to the comparator. The oscillator output is applied to the other input to the comparator. In the steady state, the loop compensator will stabilize at a voltage that produces the proper duty cycle on power FET 1 to maintain the commanded current. At this time the voltage across the 50-µH inductor will be in volt-second balance.*

All of the low-power analog and digital circuits are contained in an analog ASIC.

* The voltage across the inductor switches from positive to negative as the FET turns on and off. When the product of the positive voltage and its duration equals the product of the negative voltage and its duration, the inductor voltage is in volt-second balance.

Firmware Design

The firmware inside the ScanJet 3c/4c has many tasks. Two of the most critical (and most interesting to work on) were the start-stop algorithm and the light control algorithms.

Start-Stop. During some scans the host computer's I/O rate may not be able to keep up with the scanner's data generation rate. This will cause the internal buffer in the scanner to fill. When this occurs the scanner may need to stop and wait for the host to catch up (empty the internal buffer) before restarting the scan. This is called a start-stop. The scanner must restart the scan in the same place that it stopped or the user will see artifacts in the final image. If the scanner's drive system can start and stop within a fraction of the y-direction sampling size then no repositioning is needed. If the scanner's drive system cannot stop or start fast enough then it must back up and reposition the scan bar to be able to restart at the correct location (see Fig. 9).

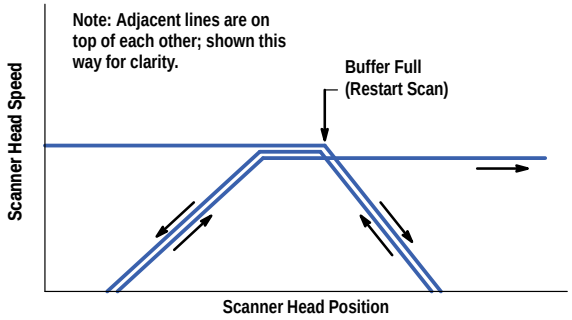


Fig. 9. *Start-stop profile.*

The ScanJet 3c/4c uses variable-speed scanning in the y-direction (along the length of the scan bed). Variable-speed scanning has two main advantages: better y-direction scaling and fast scan speeds at low resolution. The ScanJet3c/4c has a wide range of scan speeds (20 to 1), so the drive system needs some acceleration steps (of the stepper motor) to reach most of the final scanning speeds. This also means that the drive system cannot start or stop in one step. This dictated the need for a reposition movement for each start-stop.

There are three parts to a start-stop. First, when the internal buffer becomes full, the firmware marks the position and time of the last scan line and stops the drive system. Second, the firmware calculates how far to back up and then backs up and stops. Third, when there is enough space in the internal buffer the firmware accelerates the drive system up to the correct scanning speed and then restarts the scan line at the correct scan position.

The scanner firmware controls the step rate of the drive system. It uses its internal timer with a hardware interrupt to control the time between steps precisely. During acceleration, the firmware gets the next time interval from the acceleration table. Once at the proper scanning speed, the time interval is constant and the firmware just reloads the timer with the same interval. Deceleration uses the same table as acceleration in the reverse order. The firmware also keeps track of how many motor steps have occurred. Each motor step represents 1/1200 inch of travel for the scan head. This allows the firmware to keep track of the location of the scan head.

The scanner firmware also keeps track of when each scan line occurs (relative to a motor step). The scan lines are spaced 4.45 ms apart (for normal speed). A scan line may coincide with a motor step or may be between two motor steps, depending on the y-direction scan resolution). For example, for a 600-dpi scan there are exactly two motor steps for each scan line ($2 \times 1/1200 = 1/600$, so the scan head moves 1/600 inch in 4.45 ms). For a 500-dpi scan there would be 2.4 motor steps for each scan line.

When restarting the scan, the firmware must restart the CCD at least seven scan lines before putting scan data into the buffer. This is to allow the CCD to flush any extra charge in the system caused by restarting the CCD. The number of motor steps for seven scan lines depends on the y-direction scanning resolution. The number of steps to accelerate also depends on the y-direction scanning resolution. There is also a minimum number of steps that the drive system must be backed up to remove any mechanical backlash. These requirements determine the number of steps the scan head must be backed up (see Fig. 10). Once this is determined the firmware backs up the scan head and waits for the host to remove enough data from the internal buffer.

The internal buffer capacity inside the 3c/4c scanner is 256K bytes. Under the DOS operating system a typical receive block is 32K bytes (it can be larger). The ScanJet 3c/4c will restart a scan when the buffer is half full or holds less than twice the current receive block size, whichever is less.

Once there is enough space in the buffer the firmware restarts the scan. First, the scan head is accelerated up to the final scanning speed. A hardware interrupt is programmed to restart the CCD exactly seven scan lines before the position at which the last scan line was put into the buffer. Then, half a scan line away from the restart position, the buffer is reenabled such that the next line is put into the buffer. At this point the scan has been restarted and the start-stop is completed.

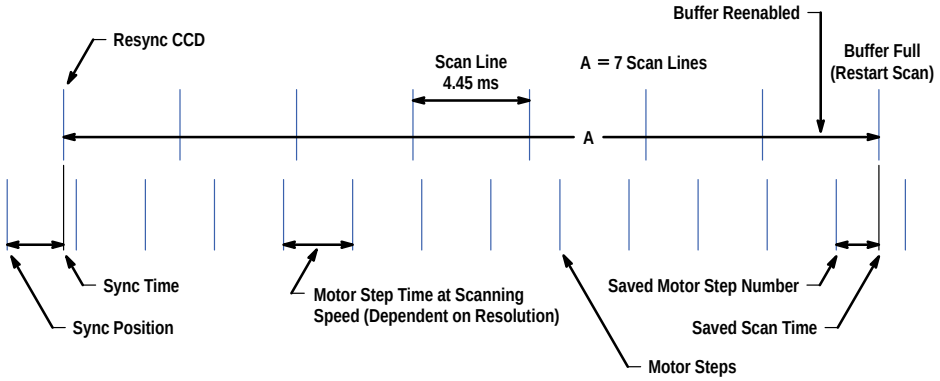


Fig. 10. *Start-stop timing.*

The start-stop accuracy of the ScanJet 3c/4c scanner is specified at half the y-direction scanning resolution. The typical resolution is between one-eighth and one-quarter pixel at the normal speed.

Light Control. The lamp in the ScanJet 3c/4c scanner is a special triphosphor fluorescent bulb. Using a fluorescent bulb has a number of trade-offs. The good news is that fluorescent bulbs have a range of phosphors to choose from. This allows the designer to balance the light spectrum with filters to give good colorimetric performance. The three phosphors in the ScanJet 3c/4c scanner give off red, green, and blue light. Florescent bulbs are also efficient, and give a reasonable amount of light for the energy used.

The bad news is that the intensity of the light is dependent on the bulb temperature. This means that as the bulb heats up the light gets brighter. If the bulb gets too hot, then the light gets dimmer again. What is worse, the bulb does not heat evenly across its length. The ends heat first and fastest and then the center of the bulb slowly heats up. The phosphors also have different efficiency-versus-temperature characteristics. This means that as the bulb heats up, it shifts color. At some nominal temperature, and only at that temperature, the phosphors are at their design efficiency, and the light is balanced with the filters. What makes this really bad is that the time it takes to complete a scan can vary between 15 seconds and 5 minutes. Fluorescent bulbs also have a long-term aging effect—a decrease in efficiency that affects performance—and the phosphors we have chosen age at different rates.

One solution to some of these problems is to leave the light on all the time. Then the bulb is at one stable temperature for the full scan. This solution has its own set of problems. For example, the bulb needs to be customer replaceable and the power consumption of the unit is high during idle time.

The ScanJet 3c/4c solves some of these problems with a real-time control system that controls the output of the light by modifying the power into the bulb during a scan. It also has separate red, green and blue system gains that are adjusted each time the light is turned on to help balance the overall color of the system. The light control system in the ScanJet 3c/4c uses the same CCD that is used for scanning. The CCD is wide enough so that it can look beyond the document being scanned at a white strip that runs along the length of the scan bed underneath the scanner top cover. This area of the CCD is called the *light monitor window.*

The light control algorithm for the ScanJet 3c/4c scanner has three parts. Part one turns on the power to the lamp and waits until some minimum level of light is detected. Part two tries to balance the output of the red, green, and blue channels by adjusting the independent system gains. Part three adjusts the power to the lamp to keep the green output at a fixed value during the scan. The purpose of part one of the lamp control is to turn the lamp on and make sure it is fluorescing at some minimum level. The goal for the startup algorithm (part two) is to have the lamp bright enough to scan with low system gains, which helps maximize the signal-to-noise ratio. The purpose of part three is to maintain the lamp at a given level for the entire scan.

Part one first sets the red, green, and blue gains to a low level. Then it turns on the preheaters (the coils at each end of the lamp) for about one second. It then turns on the lamp power, which is controlled by a pulse width modulation signal, to 20% for 4.5 ms and then to 80%. The first step at 20% is to help prevent the power supply from ringing. Once the lamp power is at 80% the control loop monitors the lamp output using the light monitor window. When the output of the lamp reaches or exceeds the minimum threshold, part two of the control algorithm starts. If the threshold is never reached the control loop will time out with an error (after about 5 minutes).

Part two of the algorithm waits about one second for the lamp to warm up (at 80% power). After the warmup delay the lamp power is lowered to 50% and the red, green, and blue system gains are adjusted. In the ScanJet 3c/4c there are two light monitor windows. One always reads the green channel's output, and the other reads either the red channel or the blue channel. The gain control loop adjusts the level of each system gain and tries to make the output of the light monitor window match a set value called the *desired value.* The window output is checked against the desired value on each end-of-scan-line interrupt, or every 4.45 ms. When the output of the green light monitor window matches its desired value (within some margin) 200 times in a row, the gains are considered stable and the green gain is fixed at its current value. If the control loop

is unable to match the desired values by adjusting the gains, that is, the gains are at maximum or minimum values, it times out. The green gain is then fixed at slightly above the minimum value or slightly below the maximum value (to give the red and blue gains some margin).

Once the green gain has been fixed, the control loop switches from controlling the gains to controlling the power to the lamp. This is part three of the light control algorithm. The lamp power control loop uses only the green channel. It uses an eight-line running average to damp the control loop. If the control loop sees a difference of one count for eight lines or eight counts for one line between the light monitor window and the desired value, it changes the lamp power by one count. When the control switches from the gains to the lamp power, there is a short delay to load the eight-line average used in the lamp power control loop. After the short delay, the output of the green light monitor window is compared to its desired value, and if they match (within some margin) 200 times in a row, the light is considered stable and the scan is allowed to start. During this stabilization period the red and blue gains are being controlled. Once the light is considered stable the red and blue gains are fixed. The control loop for the lamp power using the green channel continues to operate during the scan. If the light fails to match the desired output 200 times in a row, the scanner will time out with a lamp error. Once the scan has started, if the control loop is unable to keep the output of the green light monitor window within some tolerance of its desired value, a lamp error is issued.

RFI and ESD Design

The ScanJet 3c/4c color scanner was a challenging design with respect to RFI (radio frequency interference) and ESD (electrostatic discharge). To begin with, the mechanical design didn't lend itself to stellar RFI and ESD performance. In an attempt to lower cost and weight, the design specified a plastic chassis instead of a sheet-metal chassis. Secondly, the design spread key electrical systems throughout the scanner. For example, the controller board was positioned in the lower rear of the product. The controller board clock is derived from a 36-MHz crystal oscillator. It generates the CCD clocks, motor control signals, and lamp control signals, processes all of the image data, and controls the SCSI interface. It also controls the optional automatic document feeder or the optional transparency adapter. Not only is the controller board a source of a lot of RF energy, it also has multiple interconnections that increase the difficulty of containing that RF energy. The controller board connects to the power supply, to the carriage, to the SCSI interface, and to any optional accessory.

Another key electrical system is the power supply assembly. Besides generating $+5V$, $+24V$, $+12V$, and $-12V$, the power supply assembly also contains the lamp and motor drivers. It has a total of five cable connections including the ac power cord, the dc power cable to the controller board, the lamp cable, the motor cable, and the LED power-on indicator cable (see Fig. 11).

The third key electrical system is the carriage, which has characteristics that dominate the scanner's basic EMC (electromagnetic compatibility) performance. The carriage is a metal casting that rides on two steel guide rods. The steel guide rods are held in place by a sheet-metal plate in the rear and by the plastic chassis in the front. A fluorescent lamp is mounted on the carriage and is connected through its own dedicated cable, the lamp cable, to the lamp driver in the power supply. The lamp cable is about 15 inches long and travels along the right side of the scanner as the carriage moves under the glass window. The imaging flex circuit is a two-layer circuit that is wrapped around the outside of the CCD and is connected through the carriage cable to the controller. It is located in the left rear of the carriage. The carriage cable is a single-layer unshielded flexible cable that carries CCD clocks, which can run at speeds over 1 MHz, to the imaging circuit from the controller board. This cable also returns the resulting analog image data. The carriage cable, which is about 25 inches long, travels along the left side of the scanner as the carriage is in motion (see Fig. 11).

The carriage is a source of energy from the imaging circuit. It is also an antenna whose electrical length changes with the position of the carriage. At least three different electrical structures change as the carriage moves from the back of the scanner to the front. These include the carriage cable, the lamp cable, and the current path through the steel guide rods and the carriage. Because of this dynamic antenna structure, the radiating efficiency for any specific frequency will be optimized at one corresponding specific position of the carriage over its range of travel. One can think of it as a "self-tuning" antenna. Typical RFI control approaches that merely retune energy from one frequency to another simply do not work because the new frequency to which the RF energy is shifted will just correspond to a different carriage position at which the antenna efficiency is optimized for that frequency.

A number of RFI suppression techniques were considered. Putting a Faraday cage around the whole scanner was, of course, impossible because the top needed to be glass. Trying to enclose all the electronics and shield all of the cables also proved futile. Enclosing the controller board only seemed to make things worse. Using power and ground islands didn't help. Ferrites didn't seem to have a lot of impact, and extrapolating their performance, we estimated that RFI might only decrease by 5 dB if the box were completely filled with ferrite. Using capacitors to roll off clock or clock-like signals only seemed to increase emissions below 300 MHz.

We decided that the best approach to keeping RFI emissions down was to reduce all possible sources as much as possible. We needed to minimize the energy that got onto the carriage structure, because any energy that got there would be radiated efficiently at some point in the carriage's travel. We began to work on some new approaches that were guided by theory and that we later confirmed with experiment. First of all, we revisited the equation that describes the radiation from a current loop. Because this radiation is proportional to the product of the frequency squared, the current, and the loop area, we tried to minimize the areas of current loops and to minimize the current in those circuits with series impedance. Because we did

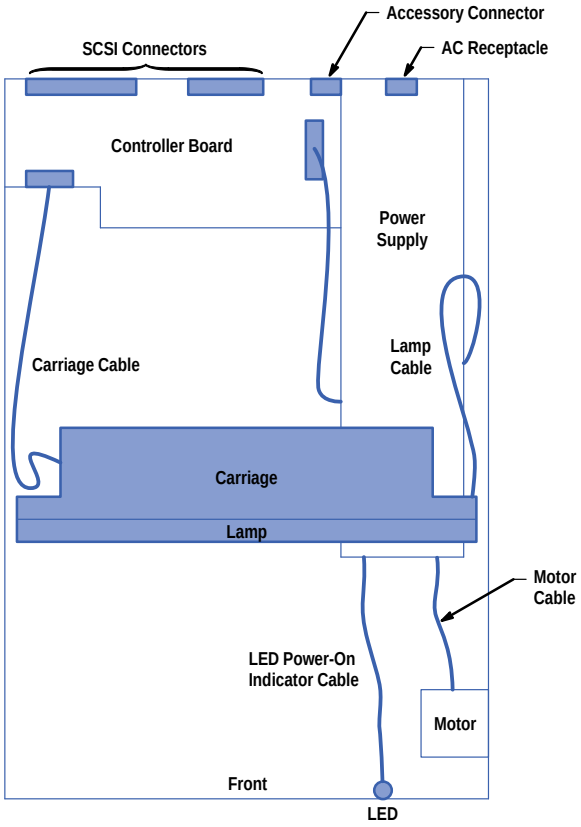


Fig. 11. *Scanner internal layout showing key components for RFI design.*

not want energy traveling onto the self-tuning antenna, we purposely tried to mismatch cable impedances so that most of the energy would be reflected back onto the controller board rather than traveling out onto the carriage cable. To do this, grounding and shielding needed to be minimized. This meant doing things that were just the opposite of what would normally be done. Instead of routing the carriage cable close to metal, it was raised away from any metal to increase its effective transmission line impedance. Although the carriage cable became a better antenna, far less high-frequency energy was able to get onto that antenna because of the impedance mismatch.

ESD also required an unusual approach. Initially, the scanner was highly susceptible to static discharges. An air discharge of only 1 kV would usually cause the SCSI bus to hang even if there was no data transfer in progress. This problem was ultimately improved by over an order of magnitude by the inclusion of a part affectionately known as the BMP or *big metal plate*. The BMP is simply the flat metal plate that is affixed to the bottom of the scanner. Its exact physical dimensions turn out to be relatively unimportant because it doesn't perform its function through any shielding or plane imaging phenomenon. It is attached to the SCSI cable shield and merely serves as a huge charge sink. The BMP could be connected to the SCSI shield without regard to three-dimensional position and it would always improve the ESD air discharge performance to over 10 kV, even while data was being transferred over the SCSI interface.

The ScanJet 3c/4c also inspired an interesting solution to a common ESD/RFI problem. Often, different methods of connecting the chassis to dc ground will have different effects on RFI and ESD. In the ScanJet 3c, if the chassis was connected directly to dc ground at the SCSI connectors, ESD performance was improved. However, if chassis ground wasn't connected at all to dc ground except in the power supply, RFI was improved. In the end, by connecting chassis ground to dc ground through parallel diodes oriented in opposite directions (see Fig. 12), good performance for both RFI and ESD was achieved.

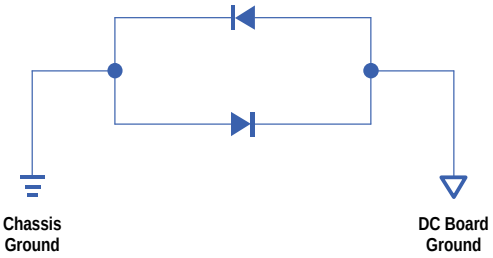


Fig. 12. *Diode connection for ESD and RFI suppression.*

Acknowledgments

The authors would like to acknowledge the contributions of the following individuals to the design of the HP ScanJet 3c/4c scanner: Bob Emmerich, Ray Kloess, Greg Degi, Irene Stein, Nancy Mundelius, Dave Boyd, Kent Vincent and Hans Neumann of HP Laboratories, project managers Gerry Meyer and Gordon Nuttall, section manager Jerry Bybee, and R&D manager Dean Buck.

Reference

1. K.D. Gennetten and M.J. Steinle, "Designing a Scanner with Color Vision," *Hewlett-Packard Journal*, Vol. 44, no. 4, August 1993, pp. 52-58.

Sing to Me

The HP ScanJet 3c/4c scanner uses variable y-direction scanning. This means that the scan head travels at different speeds dependent on the y resolution. This also means that the stepper motor runs at variable frequencies.

Musical notes are air vibrations at given frequencies. Play Tune (Esc*u0M) is an SCL (Scanner Control Language) command that can be used to make the scanner play any song downloaded into its buffer. The song can be loaded into the scanner's internal buffer using the SCSI write buffer command. The format for the song is: number of notes (2 bytes), note one, note two, etc. Each note is three bytes. All numbers are in hexadecimal format.

The first two bytes of each note specify the number of 3-MHz clock cycles between full motor steps for the desired speed. The third byte is the note duration in multiples of approximately 1/8 second. For example, middle C is 256 Hz. The clock frequency is 3 MHz, and the motor half-steps. For middle C, therefore, 3,000,000 clocks per second $\times$ 1/256 second per full step $\times$ 1/2 full step per half step = 5859 clocks per full step, which in hexadecimal is 16E3. For the third byte, a 4 would move the motor for 1/2 second (4/8 = 1/2). Thus, to get the scanner to play a 1/2-second middle C, the number to download is 16E3, 4.

For a rest between notes, set the frequency to zero and the duration to the desired length of the rest. When playing notes, the scan head always moves towards the center of the scanner and any frequency above the maximum scan rate of the scanner is truncated to the maximum scanning speed. This gives the ScanJet 3c/4c a three-octave range with the lowest note at about D below middle C.

Here is a well-known tune by Mozart (don't download the spaces or commas):

```
02f
16E3,6 16E3,6 0f47,6 0f47,6 0d9c,6 0d9c,6 0f47,9 00,2
1125,6 1125,6 122a,6 122a,6 1464,6 1464,6 16E3,9 00,2
0f47,6 0f47,6 1125,6 1125,6 122a,6 122a,6 1464,9 00,2
0f47,6 0f47,6 1125,6 1125,6 122a,6 122a,6 1464,9 00,2
16E3,6 16E3,6 0f47,6 0f47,6 0d9c,6 0d9c,6 0f47,9 00,2
1125,6 1125,6 122a,6 122a,6 1464,6 1464,6 16E3,9
```

Building Evolvable Systems: The ORBlite Project

One critical requirement that HP has learned over the years from building large systems is the need for the system and its components to be able to evolve over time. A distributed object communication framework is described that supports piecewise evolution of components, interfaces, communication protocols, and APIs and the integration of legacy components.

by Keith E. Moore and Evan R. Kirshenbaum

Hewlett-Packard has been building distributed and parallel systems for over two decades. Our experience in building manufacturing test systems, medical information systems, patient monitoring systems, and network management systems has exposed several requirements of system and component design that have historically been recognized only after a system has been deployed. The most critical of these requirements (especially for systems with any longevity) is the need for the system and system components to be able to evolve over time.

The ORBlite distributed object communication infrastructure was designed to meet this requirement and has been used successfully across HP to build systems that have evolved along several dimensions. The ORBlite framework supports the piecewise evolution of components, interfaces, communication protocols, and even programming APIs. This piecewise evolution enables the integration of legacy components and the introduction of new features, protocols, and components without requiring other components to be updated, ported, or rewritten.

A vertical slice through the ORBlite framework forms the basis of HP's ORB Plus product, a strict implementation of the CORBA 2.0 standard.

The Problem of Evolvability

By definition, a distributed system is one that contains components that need to communicate with one another. In most practical systems, however, many of these components will not be created from scratch. Components tend to have long lifetimes, be shared across systems, and be written by different developers, at different times, in different programming languages, with different tools. In addition, systems are not static—any large-scale system will have components that must be updated, and new components and capabilities will be added to the system at different stages in its lifetime. The choice of platform, the level of available technology, and current fashion in the programming community all conspire to create what is typically an integration and evolution nightmare.

The most common solution to this problem is to attempt to avoid it by declaring that all components in the system will be designed to a single distributed programming model and will use its underlying communication protocol. This tends not to work well for several reasons. First, by the time this decision is reached, which may be quite early in the life cycle of this system, there may already be existing components developers desire to use, but which do not support the selected model or protocol. Second, because of the availability of support for the model, the choice of model and protocol may severely restrict other choices, such as the language in which a component is to be written or the platform on which it is to be implemented.

Finally, such choices tend to be made in the belief that the ultimate model and protocol have finally been found, or at least that the current choice is sufficiently flexible to incorporate any future changes. This belief has historically been discovered to be unfounded, and there does not appear to be a reason to believe that the situation has changed. Invariably, a small number of years down the road (and often well within the life of the existing system), a new “latest-and-greatest” model is invented. When this happens, the system's owner is faced with the choice of either adhering to the old model, which may leave the system unable to communicate with other systems and restrict the capabilities of new components, or upgrading the entire system to the new model. This is always an expensive option and may in fact be intractable (e.g., one HP test system contains an investment of over 200 person-years in legacy source code) or even impossible (e.g., when the source code for a component is simply not available).

An alternative solution accepts the fact that a component or set of components may not speak the mandated “common protocol” and instead provides proxy services (protocol wrappers or gateways) between the communication protocols. Under this scheme, the communication is first sent to the gateway which translates it into the nonstandard protocol and forwards it on to the component. This technique typically gives rise to the following issues:

Issue	Typical Cause
Degraded performance	Message forwarding
Resource use	Multiple in-memory message representations
Reliability	The introduction of new messages and failure conditions
Security, location, configuration, and consistency	Disjoint mechanisms used by different communications protocols

It is tempting to think that the problem of evolvability is merely a temporary condition caused by the recent explosion in the number of protocols (and things will stabilize soon) or that the problem is just an artifact of poor design in legacy components (and won't be so bad next time). It appears, however, that this problem of protocol evolution is intrinsic in building practical distributed systems. There will always be protocols that are claimed to be better, domain-specific motivations to use them, and legacy components and protocols that must be supported. Indeed, we consider it a truism that nearly any real distributed system will have at least three models: those of legacy components, the current standard, and the emerging latest-and-greatest model. The contents of these categories shift with time—today's applications and standard protocols will be tomorrow's legacy.

Dimensions of Evolution

The ORBlite architecture is concerned with multiple dimensions of evolution.

Evolution of Component Interface. A component's interface may evolve to support new features. The danger is that this evolution will require all clients of the component to be updated. For reasons cited in the previous section, there must be a mechanism whereby old clients can continue to use the old interface and new clients can take advantage of the new features.

Evolution of Component Implementation. A component's implementation may evolve independently of the rest of the system. This may include the relocation of a component to a new hardware platform or the reimplementation of a component in a new programming language. There must be a mechanism that insulates other components from these changes in the implementation yet maintains the semantic guarantees promised by the interface.

Evolution of Intercomponent Protocol. It is generally intractable to choose a single communication protocol for all components in the system. Different protocols may be more attractive because of their performance, availability, security, and suitability to the application's needs. Each communication protocol has its own model of component location, component binding, and often data and parameter representation. It must be possible to change or add communication protocols without rendering existing components inaccessible.

Evolution of Intercomponent Communication Model. The programming models used to perform intercomponent communication continue to evolve. They change over time to support communication of new types of data and new version communication semantics. At the same time, new programming models are frequently developed. These models are attractive because of their applicability to a particular application, because of their familiarity to programmers on a particular platform, or because they are merely in fashion or in corporate favor. It must be possible to implement components to a new model or a new version of an existing model without limiting the choice of protocols to be used underneath. It must also be possible to do so without sacrificing interoperability with existing components written to other models or other versions of the same model (even when those components will reside in the same address space).

Contribution of Distributed Object Systems

Distributed object systems such as the Object Management Group's CORBA (Common Object Request Broker Architecture)[1,2] and Microsoft's® OLE (Object Linking and Embedding),[3] like the remote procedure call models that preceded them, address the issue of protocol evolution to a degree by separating the programming model from the details of the underlying protocol used to implement the communication. They do this by introducing a declarative Interface Definition Language (IDL) and a compiler that generates code that transforms the protocol-neutral API to the particular protocol supported by the model (see Fig. 1). As the protocol changes or new protocols become available, the compiler can be updated to generate new *protocol adapters* to track the protocol's evolution. These adapters are shown as *stubs* and *skeletons* in Fig. 1.

Another benefit of IDL is that it forces each component's interface to be documented and decouples a component's interface from its implementation. This allows an implementation to be updated without affecting the programming API of clients and simplifies the parallel development of multiple components.

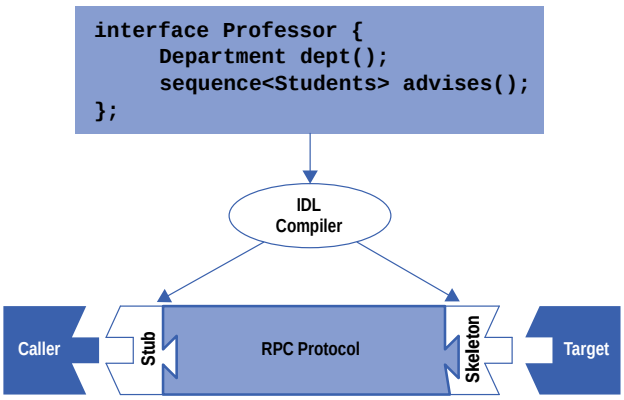


Fig. 1. *Generating stubs and skeletons from IDL. The stub and skeleton serve as software protocol adapters, which can be updated as a protocol evolves.*

In CORBA and OLE, interfaces are reflective—a client can ask an implementation object whether it supports a particular interface.* Using this dynamic mechanism, a client can be insulated from interface and implementation changes. Clients familiar with a new interface (or a new of an existing interface) ask about it, while old clients restrict themselves to using the old interface.

While such systems abstract the choice of communication protocol, none addresses the situation in which a system needs to be composed of components that cannot all share a single protocol or a single version of a protocol.** CORBA and OLE have each defined a protocol that they assert all components will eventually adopt. For reasons cited above, we feel that each is merely adding yet another (incompatible) protocol to the mix—a protocol that will continue to evolve.

Key Contributions of ORBlite

The ORBlite distributed object-oriented communication framework was designed with these concerns in mind. It takes the protocol abstraction provided by IDL a step further by allowing a single component to be accessed and to communicate over multiple protocols and multiple versions of the same protocol, simultaneously and transparently. Centered around the notion of the declarative interface, ORBlite also provides for different components to be written to different models, even when the components reside in the same process. The result is that programmers are presented with the illusion of the entire system adhering to their processing model regardless of whether this is true or in fact whether the component at the other end is even implemented using the ORBlite framework. It further enforces the notion that programming models and protocols have no knowledge of one another with respect to either existence or implementation, allowing the programmer complete freedom to mix and match.

ORBlite departs from the traditional client/server model by treating caller (client) and target (server) as merely roles relative to a particular call. Any process can contain objects that act as both callers and targets at different times or even simultaneously. Thus, ORBlite is fundamentally a peer-to-peer model even though a particular system may elect to follow a strict client/server distinction.

The main goal of the framework is to provide an efficient, thread-safe communication substrate that allows systems to be composed of components whose protocols, language mappings (i.e., object models), implementations, clients, interfaces, and even interface definition languages can evolve independently over time. It must be possible for protocols to evolve or be added without requiring recompilation of components, for object models to evolve without obsoleting existing components (or existing protocols), and for legacy components to be integrated without requiring reengineering. The reality of systems development is that components have different owners, different lifetimes, and different evolutionary time frames.

One further contribution of the ORBlite framework is that it treats local and remote objects identically. In most current systems, the syntax for a call to a remote object is quite different from a call to one located in the same process. As a result, once code has been written with the assumption that a particular object is local or remote, this decision becomes difficult to change. ORBlite, by contrast, encourages the programmer to talk in terms of *distributable references* (i.e. references to objects that may be local or remote), even when the referenced object is believed at coding time to be coresident. Application code that uses a distributable reference will not need to be changed if the referenced object is later moved to a remote process. The framework provides extremely efficient dispatching for calls when the object is detected to be coresident. The use of distributable references allows the assignment of objects to processes to be delayed well past coding time and to be adjusted based on performance or other requirements.

* In CORBA C++ this is a dynamic _narrow() mechanism. In OLE it is the IUnknown::QueryInterface() mechanism.

** The term protocol in this article refers to more than just the transport protocol. For example, the DCE protocol supports multiple string-binding handles so that objects can be accessible over connectionless and connection-based transports. However, programs based on the DCE RPC model cannot transparently communicate with programs based on the ONC RPC model.

The ORBlite Communication Framework

The ORBlite communication framework contains a core and three key abstraction layers: the language mapping abstraction layer, the protocol abstraction layer, and the thread abstraction layer (see Fig. 2). The core is responsible for behavior that is not specific to any particular protocol or language mapping. This includes the management of object references and the lifetime of target implementations, the selection of the protocol to use for a particular call, and the base data types used by the protocols and the language mappings to communicate.

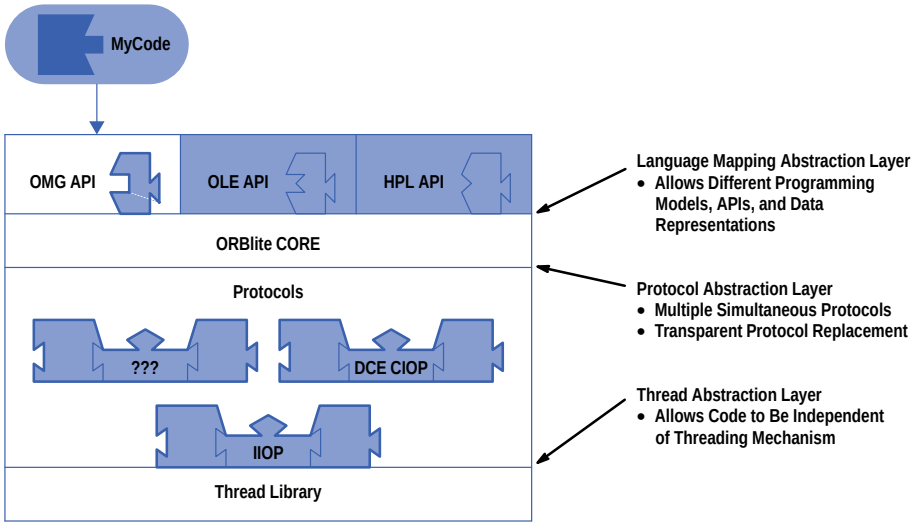


Fig. 2. *An overview of the ORBlite architecture.*

Language Mapping Abstraction Layer

This layer is designed to support evolution of the programming model presented to the application. Using the language mapping abstraction layer, each component views the rest of the system as if all other components (including legacy components) followed the same programming model. An OLE component, for example, views remote CORBA components as if they were OLE components, and a CORBA component views remote OLE components as if they were CORBA components.* This abstraction layer allows components to follow multiple programming models even when the components are located in the same address space.

Protocol Abstraction Layer

This abstraction layer is designed to support the evolution of protocols and the choice of protocol sets available in a particular system. In addition, it decouples the in-memory representation expected by a particular language mapping from the protocol used to communicate between components on a given call. For example, implementations of DCE RPC assume that the in-memory image for a structure has a particular memory alignment and member ordering. ONC RPC, on the other hand, has a different assumption about how memory should be layed out.[4,5] The protocol abstraction layer allows a given language mapping to transparently satisfy both without restricting its own layout decisions.

The protocol abstraction layer provides several features:

- Support for multiple simultaneous communication protocols—services can be shared across communication protocols and components can interact with objects simultaneously over multiple protocols.
- Support for transparent protocol replacement—one protocol can be replaced with another protocol without any change to application code. Available protocols are declared at link time or are dynamically loaded. No recompilation is necessary to change the available protocol set.
- Support for legacy integration—the framework does not need to be on both sides of the communication channel. Each protocol has full control over message representation, enabling a protocol to be used to communicate with non-ORBlite components.
- Support for multiple in-memory data representations—applications can choose the in-memory representation of data structures without incurring copy penalties from the protocols.

* The mapping between CORBA and OLE was standardized by OMG and is detailed in reference 3.

Thread Abstraction Layer

This layer is designed to provide a portability layer such that components can be written to be independent of platform-specific threading mechanisms. The thread abstraction layer also serves to coordinate the concurrency requirements of the various protocol stacks. When a protocol can be written in terms of the thread abstraction layer, it can coexist with other communication protocols in the same process. All parts of the ORBlite framework are written in a thread-safe

and thread-aware manner. The framework manages object lifetimes to ensure that multiple threads can be exploited and simultaneous calls can be executing safely in the infrastructure and in each object.

These three abstraction layers are strongly interrelated. A protocol that obeys the protocol abstraction layer will typically use the language mapping abstraction layer to marshal and unmarshal data structures. A language mapping, such as the OMG C++ mapping, will in turn use the protocol abstraction layer to allow the protocol to marshal the structure in the protocol's preferred representation.

Conceptual Overview of an ORBlite Call

In ORBlite, there are six major pieces involved in a distributed call. These pieces are shown in Fig. 3. In systems that include legacy components, two of these pieces might be purely conceptual. A legacy server might not have a discernible skeleton or an identifiable implementation, yet will honor the wire protocol. Likewise, a legacy client may not have a real stub.

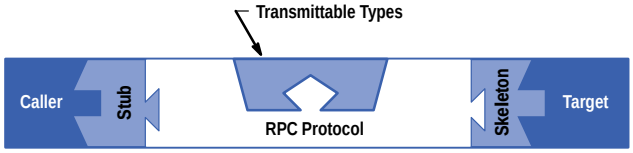


Fig. 3. *The pieces involved in a distributed call.*

The ORBlite model is similar to the CORBA and OLE models, except that in ORBlite an IDL compiler, for a given language mapping, emits stubs, skeletons, and types that are protocol-neutral. ORBlite further allows the caller and stub to follow a different language mapping from the skeleton and implementation.

Stub. The stub is responsible for turning a client-side, language-mapping-specific call of the form:**

```
result = object.foo(a,b,c);
```

into the protocol-neutral form:

```
ORBlite::apply(object, "foo", arglist);
```

Essentially, the stub is saying to the ORBlite core, "invoke the method named "foo" on the implementation associated with object using the list of arguments in arglist."

Skeleton. The skeleton is primarily responsible for the reciprocal role of turning a call of the form:

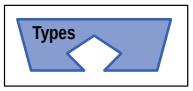
```
ORBlite::apply(object, "foo", arglist);
```

back into a call of the form:

```
result = impl.foo(a,b,c);
```

The stub can be viewed as the constructor of a generic call frame. The skeleton can be viewed as a call-frame dispatcher.

** The examples here use C++ syntax. The actual call syntax is a property of the language mapping. Also, note that the internal calls described here have been simplified.

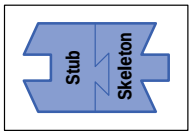


Transmittable Types. A language mapping defines one or more in-memory data representations or classes for each type (e.g., structure, union, interface, any,* etc.) describable in its IDL. For such data to be passed to a protocol, it must inherit from an ORBlite-provided base class TxType. Such classes are called *transmittable types* and support methods that allow protocols to request their instances to *marshal* themselves or to *unmarshal* themselves from a marshalling stream.** Occasionally, a language mapping may have a specification that precludes the types presented to the programmer from inheriting from TxType. In such cases, the IDL compiler often emits parallel transmittable classes that wrap the user-visible classes. These parallel classes are the ones presented to the core or to the protocols.

By convention, the marshalling methods are implemented in terms of requests on the stream to marshal the instance's immediate subcomponents. As an example, an object representing the mapping of an IDL sequence will marshal itself by first requesting the marshalling of its current length and then requesting the marshalling of each of its elements. ORBlite contains abstract transmittable base classes for each of the types specifiable in CORBA IDL, which implement the canonical marshalling behavior. Thus, the classes defined by a language mapping typically provide only methods that make a reference to or marshal the subcomponents

When a protocol's marshalling stream receives an instance of a transmittable type, it typically responds by simply turning around and asking that instance to marshal itself. Occasionally, however, a protocol may have special requirements for the wire representation (as with DCE's padding requirements for structures). Transmittable types provide type-safe accessors (foreshadowing C++'s recent dynamic_cast() mechanism) which allow a marshalling stream to ask, for example, "Are you a structure?," and take action accordingly, often calling the transmittable type's subcomponent marshalling methods directly.

The marshalling capability also provides transmittable types with the ability to convert from one language mapping's inmemory representation to another's (or between a single language mapping's distinct in-memory representations for the same type). As long as the two data types assert that they represent the same external IDL type, they can use a highly optimized in-memory marshalling stream to perform the conversion with the source object marshalling and the sink unmarshalling.



Local Bypass Optimization. When the stub and the skeleton exist in the same process space, the stub can directly invoke the skeleton's methods and bypass the transformation to and from the apply() call. In this case, the call:

```
result = object.foo(a,b,c);
```

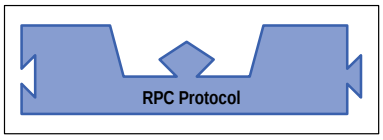
is directly forwarded through the skeleton using

```
result = impl.foo(a,b,c);
```

Note that the signatures for these two calls do not need to be identical.

An implementation object can disable this optimization. This is useful when an object wishes to ensure that a protocol has an opportunity to service every invocation, even those that are local. Certain logging, high-availability, and release-to-release binary compatibility mechanisms require this form of protocol intervention, even for the local case.

When the stub and skeleton reside in the same process but follow different language mappings, the stub may not know the target implementation object's calling conventions, or the argument data may not be in the appropriate form. When this happens, the local bypass is not taken. Instead, the call is routed through the protocol abstraction layer, which will use a very efficient local procedure call (LPC) protocol. This protocol behaves like a full RPC protocol (see below), but instead of marshalling its argument list, it merely tells the arguments to convert themselves from the caller's format to the target's.



RPC Protocol. The RPC protocol is primarily responsible for implementing a distributed apply() call. It works in cooperation with the transmittable types to migrate a call frame from one process space to another. ORBlite does not require that the protocol actually be an RPC protocol, only that it be capable of presenting the semantics of a thread-safe distributed apply( ) call. Asynchronous and synchronous protocols are supported, and it is common for more than one protocol to be simultaneously executing in the same process. The protocol may also be merely an adapter which is only capable of producing the wire protocol required for a particular remote interface but is not a full RPC implementation.

The separation between the *transmittable types' marshallers* and the RPC protocol means that transmittable types can be reused across different RPC protocols (see Fig. 4). An additional benefit is that adding a new custom protocol is fairly straightforward because almost all of the complex marshalling is handled outside of the protocol layer.

All RPC protocols have the same shape, meaning that each protocol obeys the protocol abstraction layer. There are well-defined interfaces for how a stub interacts with the protocol, how the protocol interacts with the marshallers, and how the protocol interacts with the skeleton.

* A self-describing type that can hold an instance of any IDL-describable type.

** Marshalling is the process of serializing a data structure into a buffer or onto a communication stream such that the resulting data stream is sufficient to recreate or initialize an equivalent object. Unmarshalling is the opposite process of reading the stream and creating or initializing the object.

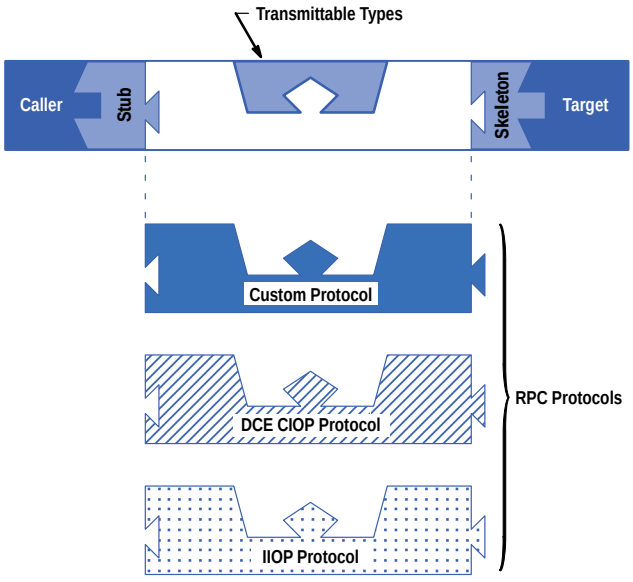


Fig. 4. *Alternate RPC protocols.*

These interfaces are, however, logically private in that they are not directly exposed to the client or to the implementation. Keeping these interfaces private means that the system can dynamically choose, based upon a variety of variables, which protocol should be used to connect a particular client to a particular implementation for a particular call. Examples of variables that may affect protocol selection would be the protocol's estimate of the time needed to bind to the implementation, a protocol's round-trip-time estimate for executing an apply( ) call, the security required on the communication, whether the channel should be rebound* on error, or the latency allowed for the call invocation.

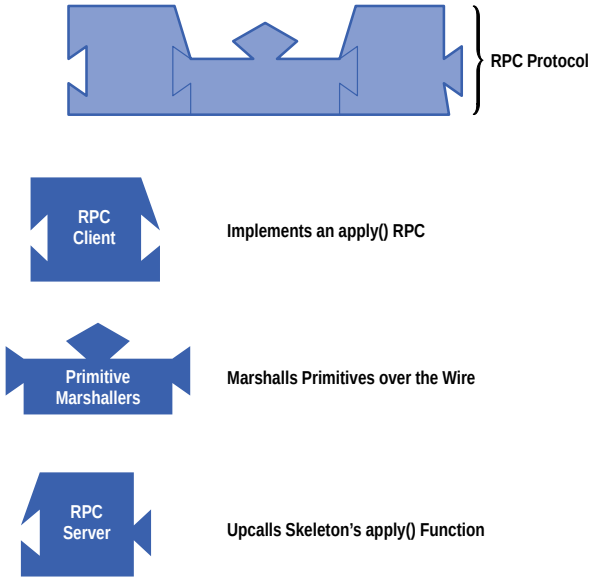


Fig. 5. *The major transport components associated with protocols.*

Internal Structure of an RPC Protocol. Only the external interfaces for the RPC protocols are defined by ORBlite. The internal structure may vary considerably between protocols. ORBlite makes no statement on whether a protocol is connection-based or connectionless, which marshalling format is used (NDR, XDR, ASCII, etc.), whether the protocol represents data in big-endian or little-endian format, or even what physical medium is used by the underlying communication mechanism. In general, however, protocols will have the three major components shown in Fig. 5:

- The RPC Client implements the client side of the apply( ) call and is responsible for locating the target's implementation.

* Rebound means to reestablish a connection between a caller and a callee if an error occurs.

- The primitive marshallers support the transmission and reception of primitive data types in a protocol-specific manner.
- The RPC Server is responsible for receiving the call over the wire, using the ORBlite core to find the skeleton associated with the target of the invocation and forwarding the RPC Client's apply( ) call to the target skeleton.

Logical Call Flow

Given these pieces of the puzzle, the logical flow of control for a remote method invocation is shown in Fig. 6.

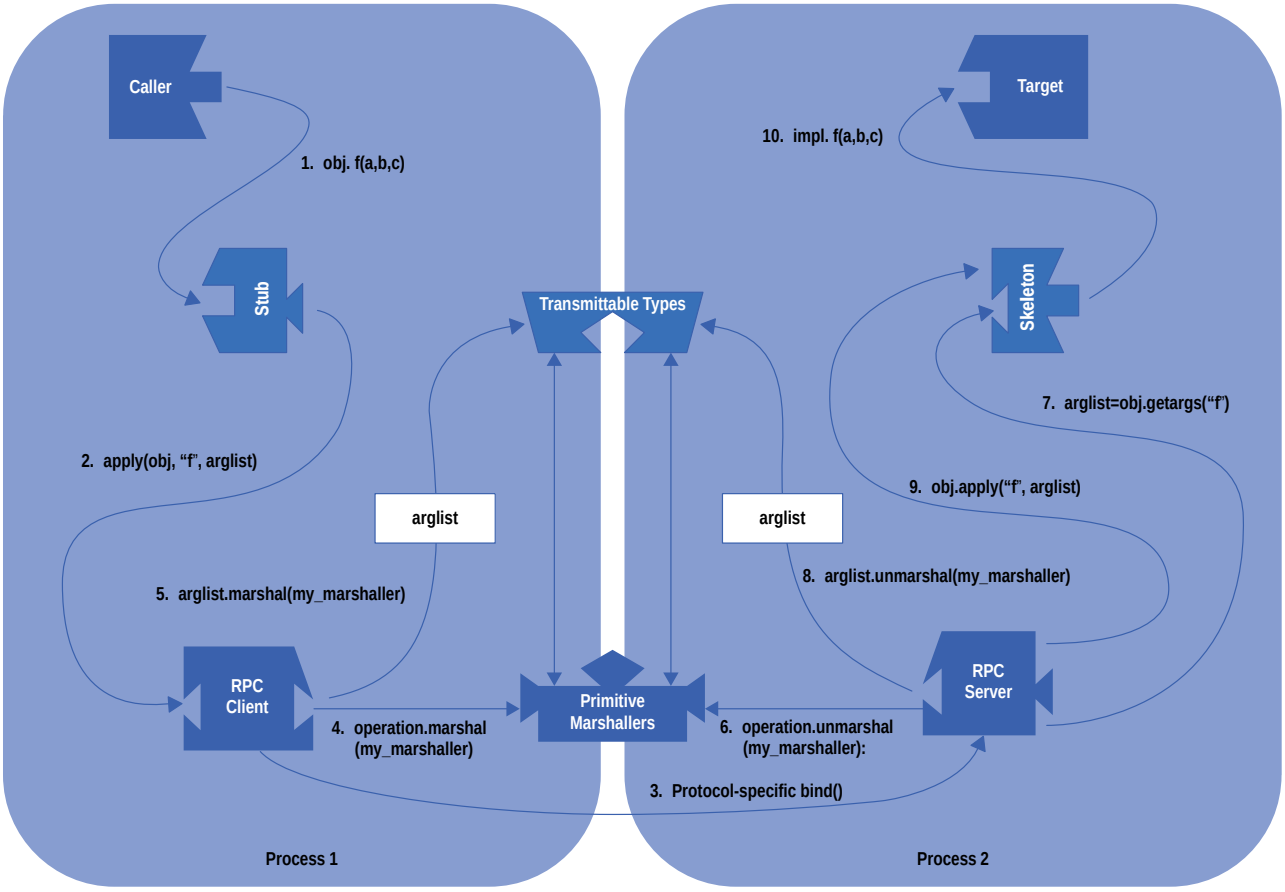


Fig. 6. *The logical flow of a remote method invocation.*

Step 1. The caller executes the method f(a,b,c) on the stub object.

Step 2. The stub creates an arglist and calls the RPC Client's apply( ) function.

Step 3. If necessary, the RPC Client binds with the target's RPC Server using a protocol-specific mechanism.

Step 4. The RPC Client marshals the identifier for the target skeleton and then marshals the name of the operation to perform.

Step 5. The RPC Client marshals an identifier for the target skeleton, then marshals the name of the operation to perform, and finally tells the arglist to marshal itself (handing in the transport's primitive marshallers). The arglist will use its transport independent marshallers to turn composite data structures into primitives which can be marshalled using the transport's primitive marshallers.

Step 6. The RPC Server unmarshals the identifier for the target skeleton and then unmarshals the name of the operation to perform.

Step 7. The RPC Server then upcalls the skeleton to get the server-side arglist for the specified operation. This upcall is a critical component in decoupling the language API from the underlying protocol. Without this upcall, the RPC Server component would have to know the memory format that the skeleton is anticipating and therefore would be tied to a particular memory mapping.

Step 8. The arglist returned from the upcall, which is operation-specific, is told to unmarshal its arguments. Each argument is a transmittable type and will use the protocol independent unmarshallers to construct the arglist contents from primitives unmarshalled using the protocol's unmarshalling stream.

Step 9. The skeleton is upcalled to apply the unmarshalled arglist to the desired operation.

Step 10. The skeleton takes apart the arglist and invokes the actual method on the implementation.When the call on the skeleton completes, the RPC Server will ask the arglist to marshal its output parameters back to the client process. The RPC Client will unmarshal the output parameters and the stub will return the values back to the caller.

Dimensions of Evolvability

In this section we discuss how the ORBlite framework addresses the various types of evolvability.

Evolution of Object Implementation

ORBlite uses the IDL specification and the language mappings defined by CORBA and OLE to decouple an object's implementation from its interface. In this manner, an object's implementation can be updated without affecting any other part of the system provided that the interface is considered to specify not only syntax but also semantics and behavior.

ORBlite is not tied to a particular IDL or even the set of data types describable by a particular IDL. ORBlite requires that isomorphic parts of different IDLs be mapped to the same base type constructs, but model and IDL designers are free to experiment with extensions. Such extensions may, of course, impact interoperability. For instance, a server whose interface uses a non-CORBA IDL type such as an asynchronous stream cannot easily be called by a client whose model does not map this type.

Evolution of Object Interface

In ORBlite, objects can support multiple interfaces simultaneously, and the language mapping abstraction layer allows clients to inquire of a target object whether the target supports a particular interface (in the OMG CORBA C++ mapping, this is presented as the _narrow() and is_a() methods and in OLE C++ this is presented as QueryInterface()).

If an ORBlite object supports new functionality (or changes the semantics behind an interface) the object should export a new interface. Old clients can query for the old interface, and new clients can query for the new one. In this manner, the target object can support old clients as well as new clients.

Of course, with a strongly typed object model such as CORBA, such dynamic queries are often unnecessary since the received object reference may already have been received as a strongly typed reference to the new interface.

Evolution of Programming Model

From the standpoint of evolution, there are two aspects of model evolution that must be anticipated: support for the introduction of new data types and support for new implementations of existing data types.

Evolution of Language Mapping Types. The ORBlite framework defines a set of basic data types from which the transmittable types used by each language mapping are derived. At the root of the tree is an abstract class TxType which requires the derived classes to support _marshal() and _unmarshal() methods. These methods take a primitive marshalling stream parameter supplied by the protocol being used for a particular call. Framework-provided subclasses of this root define more interfaces for each of the basic types describable by CORBA IDL (e.g., structures, sequences, or enumerations). These subclasses provide default marshalling behavior in terms of (abstract) methods for marshalling and unmarshalling the object's components.

A language mapping can evolve in two different ways. Since it is responsible for providing the actual types used by the programmer, it is free to define and modify their interfaces as emitted by the language mapping's IDL compiler. Canonically, these types will derive from the ORBlite-provided base classes shown in Fig. 7, so an OMG C++ structure or COM array will be seen by a protocol as merely a generic structure or array, regardless of its internal representation.

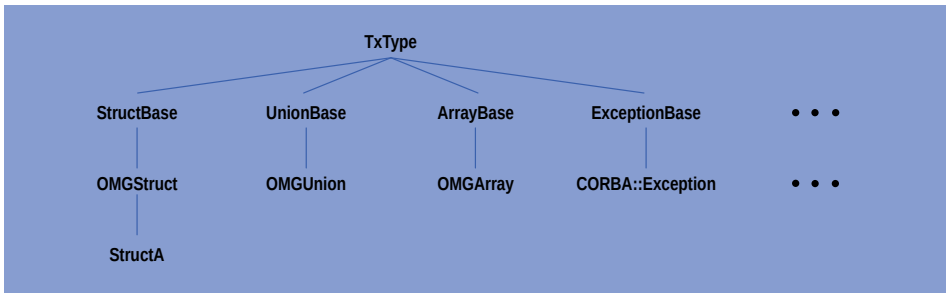


Fig. 7. *Data types derived from ORBlite's base class TxType.*

Note that there is no requirement that the actual types as presented to the programmer be transmittable. A language mapping merely has to guarantee transmittability of the data provided to a protocol. It is perfectly acceptable for a language mapping to use a transmittable wrapper class within argument lists and idiosyncratic classes (or even C++ primitives or arrays) in its API.

The other way that a language mapping can evolve is by adding types that are not directly supported by the ORBlite framework. The OLE mapping, for example, does this to create a VARIANT data type. The mapping can choose to implement the new data type in terms of one of the existing types (for instance, introducing a tree data type for use by the application but internally representing it using a sequence data type) and subclassing from a provided base. The language mapping can also choose a private representation for its contents and derive directly from TxType.

An additional attribute of ORBlite that supports a language mapping evolution is that the ORBlite framework makes no requirement that a language mapping have a unique class representing a particular IDL type. This allows a mapping to provide different representations of a type for different purposes. It also allows a later version of a language mapping to change to a new representation for a data type while remaining able to handle the old version's representation. For example, the ORBlite core uses two different mappings for strings: one optimized for equality comparison and the other for concatenation and modification. To the protocols, they behave identically.

Evolution of In-Memory Representation. There are two key issues involved in ensuring that the ORBlite core and the protocols are decoupled from the language mapping's data type representation. The first issue is ensuring that the RPC Client can marshal the parameters of a call, and the second is ensuring that the RPC Server can unmarshal the parameters without requiring excess buffering or parameter transformation. Essentially, we do not want to have to require that the language mapping translate from a protocol's in-memory data representation to its own.

The first issue is handled by the transmittable types' marshallers and accessors, which allow a protocol to marshal and retrieve composite data types without any knowledge of a language mapping's in-memory data representation.

The second issue is more complicated, and is shown as step 7 in Fig. 6, in which the RPC Server upcalls the skeleton to acquire the server-side default arglist. This upcall allows the RPC Server to offload memory management and in-memory representation for the incoming arguments to the portion of application code that actually knows the data type that is expected. A consequence of this is that the RPC Server can be reused across language mappings and is independent of the evolution of a particular language mapping.

The arglist returned from the upcall knows how to unmarshal itself. This means that the RPC Server does not need to buffer the incoming message and can allow the arglist to unmarshal its components directly into the language-mapping- specific memory representation. This is sometimes called zero-copy unmarshalling. The number of message copies is a major performance bottleneck in interprocess messaging.

Some language mappings, such as our experimental C++ mapping, allow an implementation to override the skeleton's default construction of the arguments. This is typically used when the implementation has a particular memory representation that is more convenient for the application than the default representation provided by the language mapping (e.g., the tree structure mentioned earlier). Overriding the construction of the default arguments removes the copy that would normally be required to switch representations. A language mapping can use this technique to support features not currently found in CORBA or OLE.*

The upcall is also used for two other features:

- Checking the per-object and per-method security policies
- Setting the thread-dispatch policy (e.g., thread priority and whether a new thread should be launched when executing the method).

A language mapping will typically allow the implementation to override the skeleton's default responses to the security policy or thread-dispatch mechanism.

Supporting Protocol Evolution

The principal obstacle to protocol evolution in most systems is the dependency of application code on protocol-specific APIs. In ORBlite, there are no references by the ORBlite core or by any of the language mapping components (i.e., the stub, the skeleton, and the transmittable types) to any specific protocol. Given this independence from a specific protocol, there is no need for visibility to the programmer.

This actually caused a rather interesting problem. It was not possible to just link a protocol into an ORBlite image as a normal C++ library. Since the core supports multiple protocols and there are no references by the language mapping or the core to any protocol, the linker does not have any unresolved symbols that would pull in a protocol built as a library. To overcome this obstacle we force the protocol to be loaded by creating an unresolved reference at link time.

* For instance, arbitrary graphs, migratable objects, or structures that support inheritance.

The protocols of a system evolve by dynamically or statically linking new protocols (or new versions of old protocols) into an ORBlite process. Updating or adding a protocol requires no change to the application code, the ORBlite core, or any language mapping.

To add a new protocol, the protocol developer derives from four abstract classes (the RPC Client, the RPC Server, the RPC primitive marshallers, and the RPC_Info class). The RPC_Info class registers the protocol with the ORBlite core and implements the bind() call for the protocol. The bind() call returns an instance of the RPC Client abstract interface that will be used to issue the apply() call for communication with a particular virtual process.

The RPC primitive marshallers will be used during the apply() call to choose the on-the-wire representation for the arguments of a call. They are called to marshal primitive data, such as integers and floating-point numbers, and are also given a chance to handle composite transmittable types. Normally, this last call merely hands marshalling responsibility back to the transmittable object, but the protocol can use this hook to satisfy special externally mandated padding, alignment, or ordering requirements as with DCE RPC's alignment requirements for structures and unions.

Managing Object References and Binding. Fig. 6 depicts the flow of a method invocation assuming an RPC Client has already been selected. In its simplest form, an RPC Client is selected when a client invokes a method on a stub. If the stub is not already bound to a suitable RPC Client, the stub asks the ORBlite infrastructure to find a protocol that can connect to the target object associated with an object reference. A bound RPC Client can become unsuitable if the client requires a particular quality of service (such as authentication or deadline-based scheduling). If the RPC Client is not suitable, a new RPC Client must be bound or an exception raised.

Each protocol registers with the ORBlite core a unique identifier and a binding interface. Each object reference contains a set of protocol tags and opaque, protocol-specific address information. The tags supplied in the object references are used by ORBlite to select a protocol that might be able to communicate with the target object.

If the target object is accessible over multiple protocols (i.e., both the client and the server support more than one protocol in common) then the protocol with the best quality of service is selected. The current selection criterion is based on a combination of the overhead involved for binding to the process associated with the reference plus the overhead for invoking the call. Assuming the process containing the object is activated, most RPC protocols have a 10-ms initial binding cost plus a 1-ms round-trip overhead per call. Protocols that can reuse connections across objects are generally selected in preference to connectionless protocols, which are selected in preference to protocols that require connection setup. The actual quality-of-service parameterization can get complicated. A named collection of collocated objects is called a *virtual process*. Fig. 8 shows the situation in which a process has exported two objects A and B in the virtual process VP1234. The virtual process is accessible over three protocols: IIOP (Internet Inter-ORB Protocol), ONC RPC, and the DCE-CIOP (DCE Common Inter-ORB Protocol).

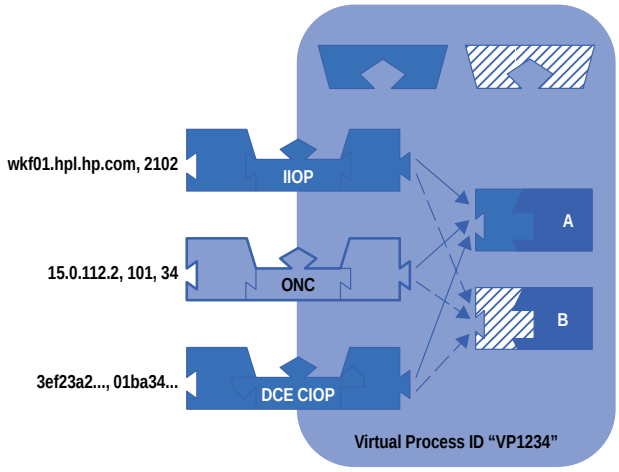


Fig. 8. *Using multiple profiles to locate object implementations.*

In ORBlite, protocols are encouraged to cache in the object reference the protocol-specific address of the last known location of the virtual process containing the object. While objects do move, the last known address is often correct and caching it can improve performance over using an external location mechanism.

Handling Common Scalability Issues. ORBlite was designed to support very large numbers of object references (more than 100,000) within a single process. To improve the scalability of location and per-object memory overhead, ORBlite provides support for protocols that wish to merge per-object cache information for objects located at the same address. In this model of object addressing, the address information held in an object reference is partitioned into two parts: an address associated with a virtual process identifier and an object identifier, which uniquely identifies the object within the virtual process.

In Fig. 8 the objects are named A@VP1234 and B@VP1234. A client that holds references to A and B can merge the cache information for the virtual process VP1234.

Often there are hundreds if not thousands of objects per process, and therefore, if location information for a protocol is based on a virtual process identifier, locating a single object in a process will have the side effect of refreshing the address information for all other objects at the same address. Some protocols will lose cache information for other protocols as the object reference is passed between processes. This is unfortunate because the cache information must be recreated if the object is to be accessible over other protocols. It is highly recommended that protocol designers allow object references to contain additional opaque information that may be used by other protocols.

ORBlite makes no requirement that a protocol use the virtual process abstraction, nor does it dictate how a protocol locates an object. ORBlite does expect, however, that the protocol's address information contained in an object reference is sufficient for that protocol to locate and, if necessary, activate the target object.

Supporting Legacy Protocols

In most cases, an object reference is created when an implementation is registered with the ORBlite infrastructure. When such an object reference leaves the process, the opaque, protocol-specific address information associated with each currently loaded protocol is marshalled along with it.

In the case of legacy components, it is likely that ORBlite is not in the server process. In this case, the binding information for the protocol must be added to the object reference via some other mechanism. Such ad hoc object references may be created by the legacy protocol, which obtains addressing information through an out-of-band mechanism. Alternatively, they may be acquired using normal protocols from a special-purpose server which creates the references from information kept in system configuration tables. However such constructed object references are obtained, they are indistinguishable from real object references and can subsequently be handed around in normal ORBlite calls.

When a stub attempts to bind the object reference, the protocol tag is matched to the protocols supported by the client process. If the process supports the protocol, an RPC Client is created that can interpret the request and communicate with the non-ORBlite server using the legacy protocol (see Fig. 9).

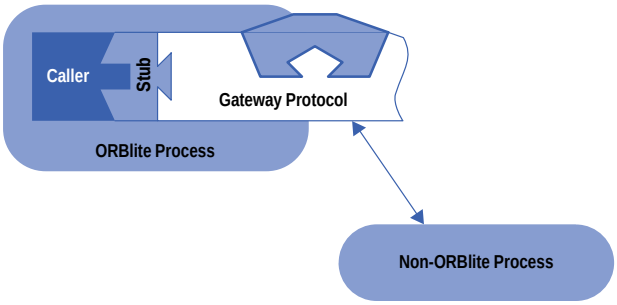


Fig. 9. *Using transport gateways.*

When ORBlite is not on both sides of the communication link, the protocol used is referred to as a *gateway protocol*. Note that gateway protocols are not only useful for communicating with legacy servers—an ORBlite process can publish itself on a legacy protocol so that it can be called by legacy, non-ORBlite clients. This form of publication is especially useful when a service needs to be accessible over both old protocols such as DCE RPC and new protocols such as IIOP.

Supporting Evolution of the ORBlite Core

In developing and deploying the ORBlite system, it became apparent that the typical owners of language mappings and protocols would not be the same as the typical owners of the ORBlite core. System developers from entities such as divisions building medical systems, test and measurement systems, or telecommunication systems were willing to own the portion that was particular to their domain, but each wanted the rest of the system to be someone else's responsibility.

This meant that the core itself needed to be able to evolve independently of the language mappings or protocols that plugged into it. It had to be simple to hook new protocols and mappings into old infrastructure and new infrastructure had to support old protocols and mappings.

The combination of the language mapping abstraction layer, the protocol abstraction layer, and the thread abstraction layer has made such independent evolution extremely straightforward.

Experience with the Framework

ORBlite was conceived in December, 1993 to support test and measurement systems. These systems contain computers and measurement instruments and are used in scientific experiments, manufacturing test, and environmental measurement. Analysis showed that the complexity of constructing the test and measurement system was the limiting factor in getting a product to market. Existing systems used a number of different communication mechanisms, and each component tended to have an idiosyncratic (and often undocumented) interface. Within HP, systems have used HP-IB, raw sockets, ONC/RPC, SNMP, NCS, and NFS.

At the time, there was a desire to move toward more stable, computer-industry-standard mechanisms, but it was unclear which proposed standard would win in the long run. The most likely contenders, CORBA and OLE, were still far from being well-specified. As we began publicizing our efforts within HP, we discovered that many others were facing a similar dilemma—notably those divisions responsible for medical systems and network management systems, each of which had its own set of legacy communication protocols.

The first version of ORBlite became operational in August of 1994. It supported the HyperDesk IDL/C++ language mapping[6] and two communication protocols: a thread-safe distribution protocol based on ONC RPC, and a gateway protocol designed to connect ORBlite services and clients to installed medical applications using the HP CareVue 9000 RPC protocol. The framework was extremely portable, thread-safe and reentrant, and because of the thread abstraction layer, it compiled without change on both UNIX® and Microsoft platforms. It was used in medical, test and measurement, analytical, financial, and telecommunication monitoring applications.

Over the past two years, dramatic changes have occurred in the specifications by OMG and in the OLE implementation by Microsoft. OMG has ratified a C++ language mapping,[7] two new standard communication protocols,[1] and recently an OLE language mapping for CORBA.[8] In addition, Microsoft has released a beta version of the DCOM (Distributed Component Object Model) protocol.[9]

In May, 1995, the ORBlite architecture began to make its way into external products. HP's Distributed Smalltalk was reimplemented to support the protocol abstraction layer, and the ORBlite code base was transferred to the Chelmsford Systems Software Laboratory to be turned into HP ORB Plus and released to external customers in April, 1996. HP ORB Plus, a strict implementation of CORBA 2.0, needed to support the new OMG standard C++ language mapping, which was previously unsupported by ORBlite. This pointed out the need for a well-defined language mapping abstraction layer and spurred its definition.

Since the transfer, the infrastructure has continued to evolve. We have experimented with new protocols to support high availability and legacy integration and new language mappings to support potential new IDL data types and to simplify the programmer's job. We are also investigating implementing an embeddable version of the architecture, which would have the same externally visible APIs but would be able to run in extremely memory-limited environments. Finally, we are looking into the declarative specification of protocol-neutral quality-of-service requirements and capabilities. This would assist in selecting the appropriate protocols to use and in guaranteeing the desired quality of service, where this interpreted to include performance, security, payment, concurrency, and many other dimensions. Following the ORBlite philosophy, we are attempting to design this mechanism in such a way that the set of available quality-of-service dimensions itself can evolve over time without impacting existing components.

The ORBlite infrastructure has allowed developers to build systems even as the standards evolve. The support of multiple language mappings, thread-safe distributed object communication, and multiple protocols has provided a unifying approach to building components and systems across the company. The key issues on the horizon will be ensuring that the standards from Microsoft, OMG, and others consider concurrency, streaming data types, and quality of service parameterization.

Acknowledgments

ORBlite would not have been possible without the input and feedback from our customer divisions and beta sites. We would especially like to thank Dean Thompson of the Network and System Management Division, Rob Seliger of the Medical Products Group, Horst Perner of the Böblingen Instrument Division, and Bill Fisher, Henrique Martins, and Paul Harry of HP Laboratories (Palo Alto and Bristol). We are also indebted to the ORB Plus team at the Chelmsford/Cupertino Systems Software Laboratory, especially Steve Vinoski, whose comments led to the idea of the language mapping abstraction layer, Mark Vincenzes, who was heavily involved in the design of the language mapping abstraction layer, Bob Kukura, who implemented the interoperable object references, and Bart Hanlon, who kept the product development on course. Others who assisted in the development include Kevin Chesney, who reimplemented HP Distributed Smalltalk according to the ORBlite framework, Walter Underwood and Lance Smith, who developed the thread abstraction layer, and Mark Morwood, who implemented the HP CareVue 9000 gateway protocol. We also wish to thank Mary Loomis, Joe Sventek, Jeff Burch, and Randy Coverstone for running interference for us, Harold Brown and Walter Underwood for picking up customer support when it got out of hand, Shane Fazzio for the NT build environment, and Dick Allen for general Microsoft expertise. And, of course, Lisa and Susan for putting up with all the long hours.

References

1. *The Common Object Request Broker: Architecture and Specification, Revision 2.0*, Object Management Group, July 1995.
2. *The Common Object Request Broker: Architecture and Specification*, Object Management Group, Document Number 91.8.1, August 1991 (Draft).
3. *OLE 2 Programmer's Reference: Volume 1 & 2*, Microsoft Press, Redmond Washington, 1994.
4. *OSF DCE 1.0 Application Development Guide*, Technical Report, Open Software Foundation, December 1991.
5. *Network Programming Guide, Revision A*, Sun Microsystems Inc., March 27, 1990.
6. *Second Revised Submission in Response to the OMG RFP for C++ Language Mapping*, OMG Document Number 93-11-5, HyperDesk Corporation, November 1993 (Draft).
7. S. Vinoski, editor, *C++ Mapping 1.1 Revision*, OMG Document Number TC.96-01-13, January 1996.
8. J. Mischkinsky, editor, *COM/CORBA Part A Corrected Revised Submission*, OMG Document Number ORB.96-01-05, January 1996.
9. C. Kindel, *Microsoft Component Object Model Specification*, OMG Document Number 95-10-15, October 1995 (Draft).

Developing Fusion Objects for Instruments

The successful application of object-oriented technology to real-world problems is a nontrivial task. This is particularly true for developers transitioning from nonobject-oriented methods to object-oriented methods. Key factors that improve the probability of success in applying object-oriented methods are selecting an object-oriented method, developing a process definition, and continually improving the process.

by Antonio A. Dicolen and Jerry J. Liu

Object-oriented technology is fast approaching mainstream status in the software community. Many software developers are interested in becoming object-oriented practitioners. Managers, once skeptical of its value, are considering its use in their business enterprises. This technology is old enough not to be a fad and new enough to be recognized by customers as high technology.

Within the embedded community (i.e., microprocessor-based instrumentation) at HP, there is significant interest in adopting object-oriented technology for the development of new products. However, the adoption rate of object-oriented technology at HP has been hampered by earlier negative experiences. Attempts to use object-oriented technology in instruments occurred as early as the mid 1980s. At that time the technology was in its infancy. The methods for employing the technology were immature and the development tools necessary for its effective use were nonexistent. Application of the technology at that time resulted in unmet product requirements.

These experiences hindered further development using object-oriented technology. Object-oriented technology became synonymous with slow speed, high risk, and failure. This perception imprinted itself on the culture of HP divisions using embedded software technology. It was not until the early 1990s that this perception began to change. As engineering productivity became an issue for management, software reuse emerged as a possible solution. With reuse as a business goal, an object-oriented approach was once again considered as a means of achieving that goal.

It is important to recognize that reuse and object-oriented technology are not synonymous since it is possible to achieve reuse without an object-oriented approach. Software math libraries are a prime example of this fact. This type of reuse is called *library* reuse. It is the most common and the oldest form of software reuse. *Generative* reuse, such as that provided by tools like lex and yacc, is another form of software reuse. In general these tools use a common implementation of a state machine and allow the user to modify its behavior when certain states are reached.

Another type of reuse is *framework* reuse. Microsoft® Windows' user interface is an example of framework reuse. In framework reuse, the interaction among the system components is reused in the different implementations of the system. There may be certain common code components that some, but not necessarily all, of the implementations use. However, the framework is what all these systems have in common. Microsoft foundation classes are an example of common code components. Menu bars, icon locations, and pop-up windows are examples of elements in the framework. The framework specifies their behaviors and responsibilities.

One reuse project based on this approach was a firmware platform for instruments developed at our division. The goal was to design an object-oriented firmware framework that could be reused for different instruments. With this project, we hoped to use object-oriented technology to address reuse through framework reuse. We chose to use Fusion,[1,2] an object-oriented analysis and design methodology developed at HP Laboratories, to develop our instrument framework.

In this article, we first describe the firmware framework and our use of the Fusion process. Next we present our additions to the analysis phase of the Fusion process, such as object identification and hierarchical decomposition. A discussion of the modifications to the design phase of Fusion then follows, including such topics as threads and patterns. We conclude with the lessons we learned using Fusion.

Firmware Framework

The new firmware framework is an application framework. An application framework provides the environment in which a collection of objects collaborate. The framework provides the infrastructure by defining the interface of the abstract classes, the interactions among the objects, and some instantiable components. A software component, or simply a component, is an

atomic collection of source code used to achieve a function. In many situations, a component will have a one-to-one correspondence with a C++ object. At other times, a component may be made up of multiple objects implemented in C++ or C source code.

Users of the firmware framework contribute their own customized versions of the derived classes for their specific applications. Note that the framework approach is very different from the traditional library approach. With the library approach, the reusable components are the library routines, and users generate the code that invoke these routines. With the framework approach, the reusable artifacts are the abstractions. It is their relationships to one another, together with the components, that make up the solution to the problem.

The firmware framework contains a number of application objects. These are different kinds of applications that handle different kinds of responsibilities. The responsibilities of these application objects are well-defined and focused. For example, there is a spectrum analyzer application that handles the measurement aspects of an instrument and also generates data, a display application that is responsible for formatting display data, and a file system application that knows how to format data for the file system.

There is always a root application in the system, which is responsible for creating and destroying other applications and directing inputs to them. Other components of the application framework include the instrument network layer and the hardware layer. The applications communicate with each other via the instrument network layer. The hardware layer contains the hardware device driver objects, which the applications use through a hardware resource manager. Fig. 1 shows an overview of the firmware framework.

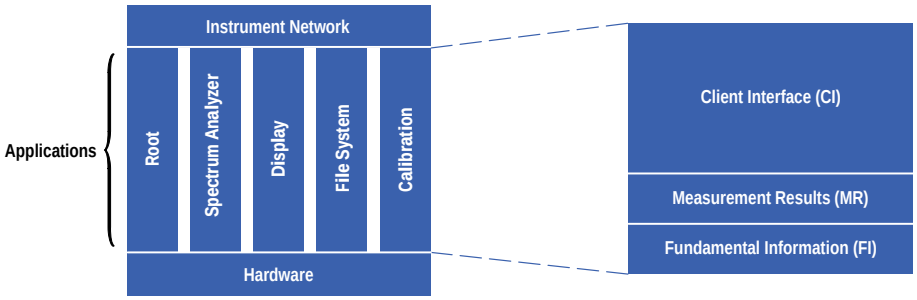


Fig. 1. *An overview of the new firmware framework.*

Application Layers

An application in the firmware framework is a collection of objects organized into three layers: client interface, measurement results, and fundamental information. These layers deal with information at different levels of semantics. The semantics at the client interface layer deal with instrument functionality while the semantics at the fundamental information layer are more related to areas such as hardware control.

Client Interface Layer. This layer represents an abstraction containing user-selectable parameters, the interface for setting these parameters, the results, and the sequence for generating the results. Thus, the client interface layer defines the features and the capabilities of an application. It is responsible for maintaining application state information and creating the requested results. This layer also contains a collection of application parameter objects that store the state of the application, and a dependency manager that manages the parameter limiting and coupling dependencies. The dependency manager also triggers events on state changes. These state changes cause the selection of the correct MeasurementResult to use to satisfy the user's request.

Take, for example, a simplified multimeter instrument. It could be an ohmmeter, a voltmeter, or a current meter. To select the voltmeter mode, the instrument software must deselect the ohmmeter or current meter mode and then select the voltmeter mode. The user interface simply turns on voltmeter mode. The dependency manager knows that these three modes are mutually exclusive and automatically sets the current meter and ohmmeter modes to off. In addition, the user could set the measured voltage to be the average value or the rms (root mean square) value. This corresponds to the selection of a specific MeasurementResult that provides the information the customer is interested in.

Measurement Result Layer. This layer is made up of objects derived from a base class called MeasurementResult. These objects contain the measurement algorithms that specify the methods for combining raw data into meaningful data.

MeasurementResult objects subscribe to and respond to events in the client interface layer and in other MeasurementResult objects. Complex measurement results contain simple MeasurementResult objects. Examples of MeasurementResult objects in an instrument application are SweepMR, MarkerMR, and LimitLineMR. These could be be measured values from a spectrum analyzer. An example of a MeasurementResult object in a display application could be a TraceDisplayItem that knows how to read a MarkerMR and generate marker information for the display.

The measurement result layer has no knowledge of how or where its input data is generated. Its input can come either from other MeasurementResults or from the fundamental information layer. It is thus free of any hardware dependencies. This layer uses the fundamental information layer to coordinate the hardware activity.

Fundamental Information Layer. This layer performs the specific activities that orchestrate the hardware components to achieve a desired result. The objects in the fundamental information layer know about specific hardware capabilities. They keep the hardware objects isolated from each other and also generate self-describing, hardware-independent data. The fundamental information layer applies hardware corrections (e.g., compensations for hardware nonlinearities) to the measured results.

The fundamental information layer contains three major components: a state machine with sequencing information that controls the objects in the layer, a production object that is responsible for orchestrating the hardware components, and a result object that is responsible for postprocessing data. Examples of fundamental information layer objects include SweepFI, which is responsible for measuring frequency spectra in a spectrum analyzer application, and the display list interpreter in the display application, which is responsible for controlling the instrument display.

Instrument Network

The instrument network contains the objects that facilitate interapplication communication, including an ApplicationArchive object, which is responsible for naming and providing information to applications, and an ApplicationScheduler object that schedules the threads that make up the applications.

Hardware Layer

The hardware layer contains the objects that control the instrument hardware. These objects contain very little context information. There are two types of hardware objects: device objects, which drive a simple piece of hardware, and assembly objects, which are collections of hardware objects. Hardware components are organized in a hierarchy much like the composite pattern found in design patterns.* Hardware objects are accessed through handles like the proxy pattern described in the patterns book.[3] Handles can have either read permission or read-write permission. Read permission means that the client can retrieve data from the object but is not able to change any of the parameters or issue commands. Read-write permission allows both. Permissions are controlled through the hardware resource manager.

Communication Mechanisms

Two main communication mechanisms glue the architecture together: agents and events. Agents translate the language of the user (client) into the language of the server (application). Different kinds of agents apply different kinds of translations. For instance, a client may enter information in the form of a text string, while its target application may expect a C++ method invocation. Thus, the client would use a specialized agent to translate the input information into messages for the target application (the server).

Events are mechanisms used to notify *subscribers* (objects that want to be notified about a particular event) about state changes. We decided to use events because we wanted to have third-party notification, meaning that we did not want the *publishers* (objects that cause an event) to have to know about the subscribers.

There are two types of events: active and passive. Active events poll the subject, whereas passive events wait for the subject to initiate the action. Our event mechanisms and the concepts of subscribers and publishers are described in more detail later in this paper.

Use of Fusion

In selecting an object-oriented method to guide our development, we were looking for a method that would be easy to learn and lightweight, and would not add too much overhead to our existing development process. We were a newly formed team with experience in our problem domain and in embedded software development, but little experience in object-oriented design. We wanted to minimize the time and resources invested in working with and learning the new technology until we were fairly certain that it would work for us. At the same time, we wanted to have a formal process for designing our system, rather than approach the problem in an ad hoc manner.

Fusion (Fig. 2) met these requirements. It is a second-generation object-oriented methodology that is fairly lightweight and easy to use.[4]

For the most part, our use of Fusion was very straightforward. We started with the system requirements, and then generated a draft of the system object model and the system operations of the interface model. We also generated data dictionary entries that defined our objects and their interrelationships. These documents made up the analysis documents. We did not develop the life cycle model because we did not see how it contributed to our understanding of the system. As time went on, we discovered that we really did not need it.

* Design patterns are based on the concept that there are certain repeated problems in software engineering that appear at the component interaction level. Design patterns are described in more detail later in this article.

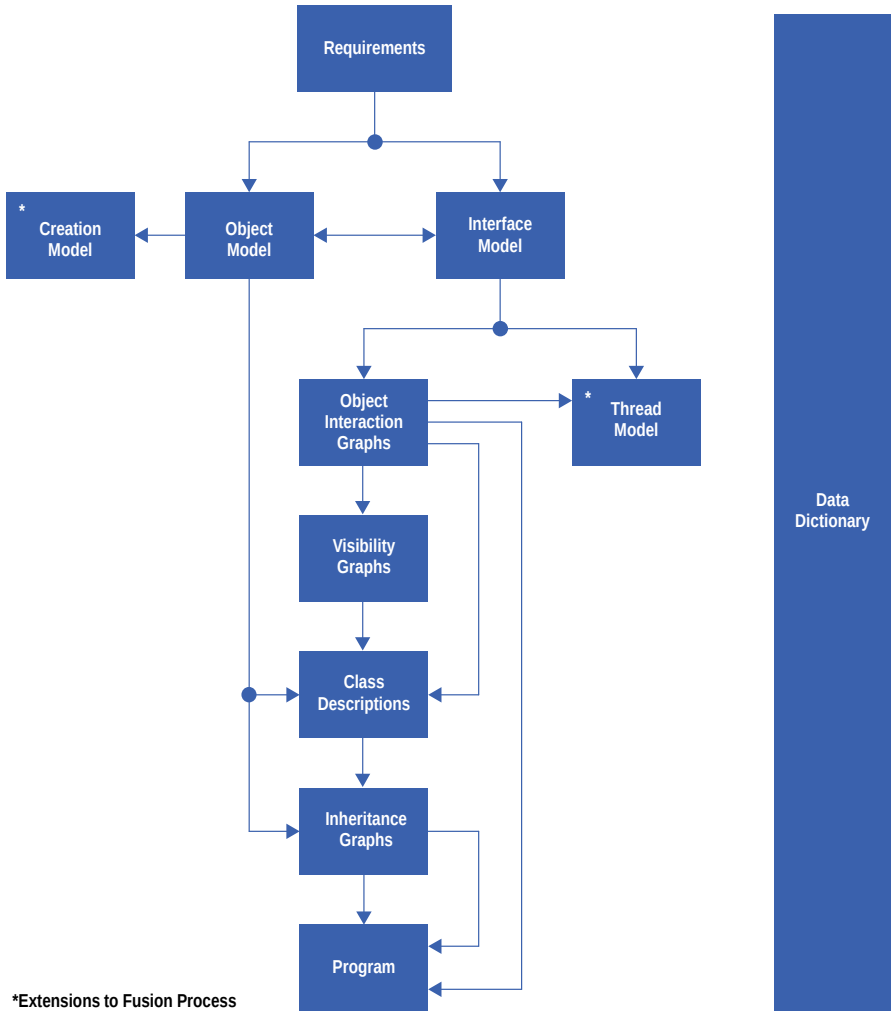


Fig. 2. *The Fusion process for software development.*

From the analysis model, we mapped the analysis onto a design and generated the object interaction graphs to show the interactions between the objects. We then generated the visibility graphs and derived the class descriptions. These were straightforward processes.

By no means did we go through this entire process in one pass. For us, using Fusion was an iterative process. Our system was clearly too large to analyze and design in one pass. If we had tried, we would have been overwhelmed with the details. Instead, we made a first pass to identify the primary objects. We then divided the system into subsystems and recursively applied the Fusion method to each subsystem level to discover the higher-order objects at that level.

For instance, at the topmost level we identified the major components of the firmware framework: the client interface layer, the measurement result layer, and the fundamental information layer (see Fig. 3). We then sketched out the interactions between these components, repeated the process for each of the subsystems, and explored the details within each of the components of the subsystems.

We did not apply the iterative process simply to find details. It was also a way to check the top-level analysis and design and feed back into the process anything that we had overlooked in the higher-level passes. These checks helped to make our system implementable. Through external project reviews with object-oriented experts, we also discovered other ways to look at our abstractions. For instance, with our original analysis, our focus was on the subsystem that performed the measurement functionalities of the instruments. Thus, we ended up with an architecture that was focused on measurement. We had layers in the system that handled the different aspects of obtaining a measurement, but few layers that supported the instrument firmware. It was not until later, with outside help, that we saw how the patterns and rules for decomposing the instrument functionality into layers applied equally well to subsystems that were not measurement related, such as the display or the file system. We were also able to abstract the different functionalities into the concept of an application and use the same rules and patterns to decide how the responsibilities within an application ought to be distributed.

We found Fusion to be an easy-to-use and useful methodology. This method provided a clear separation between the analysis and the design phases, so that we were able to generate system analyses that were not linked to implementation details.

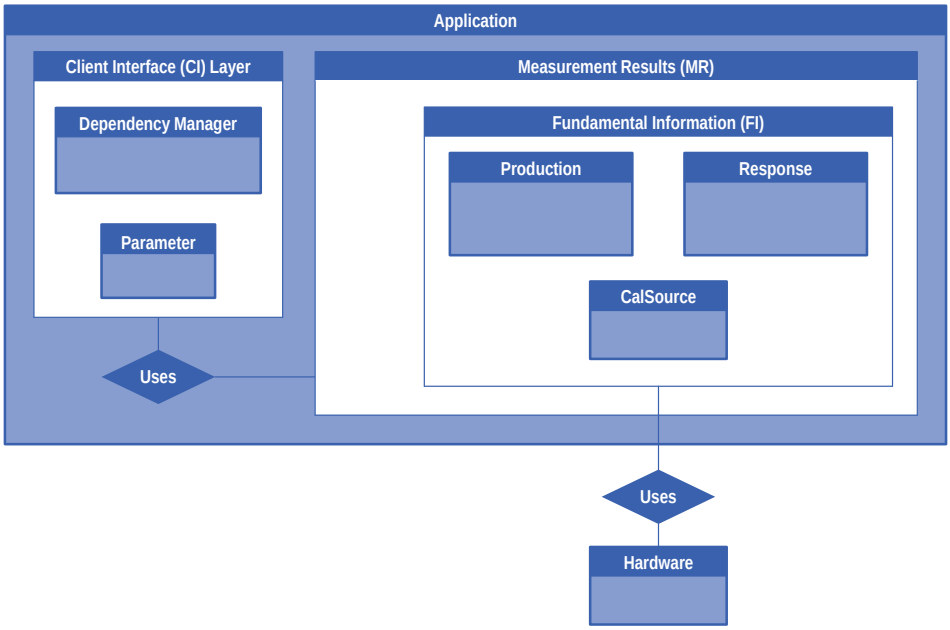


Fig. 3. *The object model for the client interface, measurement results, and fundamental information objects.*

Of course, no methodology is perfect for every situation. We made some minor modifications to the method along the way, as well as some extensions (see Fig. 2), which will be described later. For instance, we omitted the life cycle models. Since we knew that we were going to implement our system in C++, we used C++ syntax to label our messages in the object graphs and C++ class declarations when we generated the C++ classes. We also did not use the state diagram portions of Fusion to generate states for our state machines. We felt that we did not need this state machine facility and thus freed the staff from having to learn yet another notation.

Extensions to Fusion—Analysis Phase

In our desire to perform object analysis more consistently, our team developed extensions to Fusion that helped non-object-oriented practitioners make the paradigm shift to the object-oriented mind-set much more easily.

Many developers and managers naively assume that a one-week class on object-oriented technology is sufficient to launch a team into developing object-oriented software. While this may be a necessary condition, it is not sufficient for the successful acquisition and application of object-oriented technology.

Many texts and courses on object-oriented methods treat the analysis phase as merely the identification of nouns that are objects and their relationships with one another. Having conveyed this, the analysis sections of these books then focus on method notation rather than helping the novice overcome the biggest obstacle in object-oriented analysis, the identification of objects.

Without sufficient help, novices produce analysis diagrams that conform to the object notation, but merely recast ideas from either the structured analysis paradigm or from some home-grown paradigm. The circles of structured analysis and design are turned into boxes and, voila, an object diagram is born.

Our team was not spared this experience. Fortunately, we consulted object-oriented experts who taught us what to do. Thus, we developed an analysis technique that could be consistently applied project-wide to help the developers transition from structured to object-oriented analysis. This was critical to our facilitating software reuse, the primary goal of the project.

Object Identification

Successful object-oriented analysis begins with identifying a model that captures the essence of the system being created. This model is made up of behaviors and attributes that are *abstractions* of what is contained in the system to accomplish its task.

What makes a good abstraction? The answer to this question is critical to the effective use of object-oriented technology. Unfortunately, identifying the wrong abstraction encourages a process known as “garbage in, garbage out.” Furthermore, the right abstraction is critical to the ease with which a developer can implement the object model. It is possible to generate a proper object model that cannot be implemented. The key is in the choice of the abstraction.

What makes an abstraction reusable? The answer to this question is critical to achieving the value-added feature of object-oriented technology that is needed to achieve software reuse. Understanding the context in which reuse can occur is important.

An analysis framework exists that can be used to guide the identification of abstractions. This framework has the added benefit of guaranteeing that the resultant object model derived from its use is realizable. Furthermore, its foundation is based on the premise that software reuse is the ultimate goal.

In developing our analysis, we noted the questions the experts would ask when presented with our work. Fundamentally, their questions focused on understanding the responsibilities of the abstractions that we had identified. Responsibility, it turns out, gives rise to the state and behavior of an object. Previous research on this topic yielded an article[5] that discusses responsibility-based design, and describes an object-oriented design method that takes a responsibility-driven approach. We synthesized this knowledge into what can be described as *responsibility-based analysis*.

This new analysis technique is based on a pattern of three interacting abstractions: the *client*, the *policy*, and the *mechanism*. Fig. 4 illustrates the object model for the client-policy-mechanism framework.

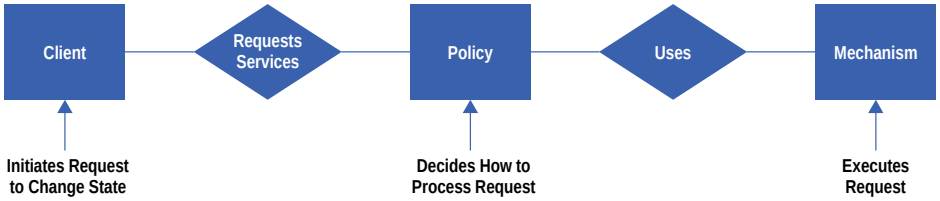


Fig. 4. *The object model for the client-policy-mechanism framework.*

The client abstraction requests services, initiates activities that change the system state, and queries for request-specific status within the system.

The policy abstraction decides when and how a request will be acted upon. It accepts the client request and, based on the responsibility given to it by the analyst, chooses the appropriate way in which the work will be done. In performing this responsibility it sets the context for the relationships between the system components.

The mechanism abstraction actually performs the change to the system state. If the operation is a state query, it returns the desired information. It does not return context information related to the operation being discussed. The mechanism abstraction makes no decision as to whether it is appropriate for it to perform an operation. It just does it.

As an example, consider creating a software application to read the current market value of HP stock. The client-policy-mechanism analysis of the problem, at a very high level, yields at the minimum three abstractions: an abstraction representing the user (the client), an abstraction that represents when and how the HP stock information is to be acquired (the policy), and lastly, an abstraction that knows about the value of HP stock (the mechanism). The mechanism abstraction, when implemented, becomes the software driver for acquiring the stock price. In one instance, the mechanism object reads the value of HP stock from a server on the Internet via telnet. In another instance, the mechanism acquires the stock value via http. (Telnet and http are two internet communication protocols.) The policy abstraction determines how often to access the mechanism. In our case it determined how often, that is, the time interval used, to poll the mechanism. The client object receives the resultant information.

From a software reuse perspective, mechanism abstractions are the most reusable components in a system. Mechanisms tend to be drivers, that is, the components that do the work. Since the responsibility of a mechanism is context-free, the work that it does has a high probability of reuse in other contexts. Being context-free means that it does not know about the conditions required for it to perform its task. It simply acts on the message to perform its task. In the example above, the mechanism for acquiring the stock price can be used in any application requiring knowledge of the HP stock price.

Though not as obvious, using the client-policy-mechanism framework gives rise to policy abstractions that are reusable. In the example above, the policy abstraction identified can be reused by other applications that require that a mechanism be polled at specific time intervals. Making this happen, however, is more difficult because the implementer must craft the policy abstractions with reuse in mind.

The analysis technique described above attempts to identify client, policy, mechanism, and the contexts in which they exhibit their assigned behaviors. When policy roles are trivial, they are merged into the client role, producing the familiar client/server model. This reduction is counterintuitive, since most client/server model implementations imbed policy in the server. However, from a software reuse point of view, it is important to keep the server a pure mechanism. On the other hand, it is also important to resist the temptation to reduce the analysis to a client/server relationship. Doing so reduces both the quality of the abstractions and the opportunity for reusing policy abstractions.

These three abstractions together define the context of the problem space. Experience has shown that to produce a clean architecture, it is important for each abstraction to have one responsibility per context. That is, a policy abstraction should be responsible for only one policy, and a mechanism abstraction should be responsible for doing only one thing.

On the other hand, abstractions can play multiple roles. In one context an abstraction may play the role of mechanism and in another context be responsible for policy. An example illustrates this point more clearly. Consider the roles in a family unit. A young child performs requests made by a parent who in turn may have been asked by a grandparent for a specific activity. In a different context, for example, when the child grows up, it plays the role of parent for its children and its parents, who are now grandparents. In this latter setting, the parents are the policy makers, the grandparents are the clients, and the children (of the child now grown up) are the mechanisms (see Fig. 5).

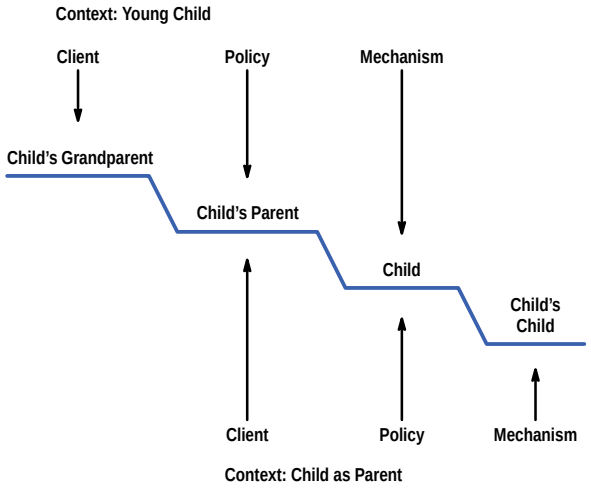


Fig. 5. *The client-policy-mechanism model as applied to a family unit.*

Just as, depending on context, a specific individual plays different roles, so it is true with abstractions. In one context an abstraction may be a mechanism and in another, a policy. The critical rule to keep in mind when using the client-policy-mechanism framework is that there should only be one responsibility per abstraction role.

Hierarchical Decomposition

Another example of systems that illustrate the single-role- per-context rule is found in the hierarchy of the military forces. In the United States, the commander in chief issues a command, the joint chiefs respond to the command and determine the methods and the timing for executing the command, and the military forces complete the task. In a different context, the joint chiefs may act as clients to the admiral of the navy who determines the methods and timing for the subordinates who execute the task (see Fig. 6).

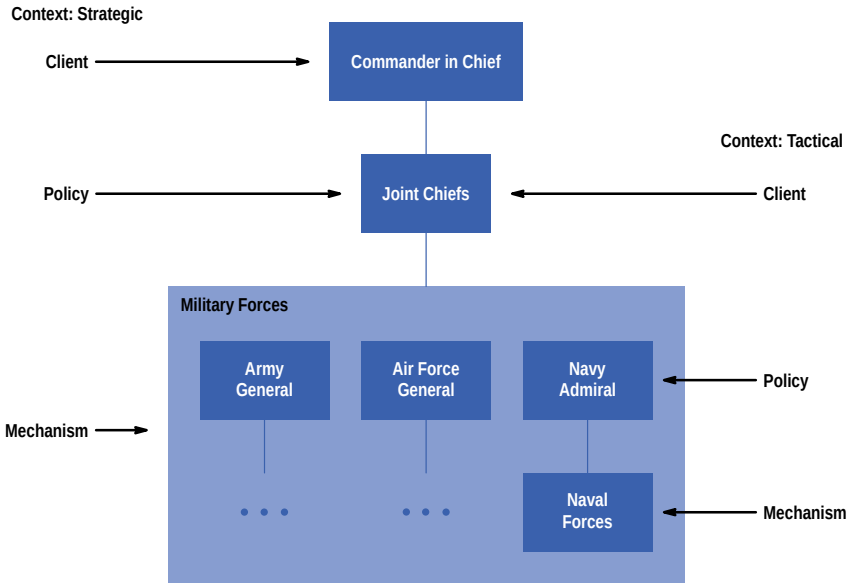


Fig. 6. *The client-policy-mechanism model as applied to a military hierarchy.*

In each of these examples there is a client, a policy, and a mechanism. In one context, a person is responsible for policy decisions. In another, the same person is responsible for mechanism or client activities. It is this concept that gives rise to the use of the client-policy-mechanism framework in helping to perform hierarchical decomposition of a problem domain. The repetitive identification of roles, contexts, and responsibilities at ever finer levels of granularity helps identify the solution space for the problem domain.

The firmware framework team performed hierarchical decomposition by identifying roles, contexts, and responsibilities. These responsibilities defined abstractions that produced objects and groups of objects during the implementation phase. In the early phases of our novice object-oriented project, it was expedient to use the words object and abstraction interchangeably. As the team gained experience and became comfortable with object-oriented technology and its implementation, the distinction between the abstraction and its resulting objects became much better appreciated.

The analysis technique based on the client-policy-mechanism framework resulted in a hierarchical decomposition that yielded layers and objects as shown in Fig. 7. Layers are groups of objects that interact with one another to perform a specific responsibility. Layers have no interfaces. Their main function is to hold together objects by responsibility during analysis to facilitate system generation. For example, many software systems include a user interface abstraction. However, upon problem decomposition, the user interface abstraction typically decomposes into groups of objects that collaborate and use one another to satisfy the responsibilities of the user interface. When the abstraction is implemented, it usually does not produce a single user interface object with one unique interface.

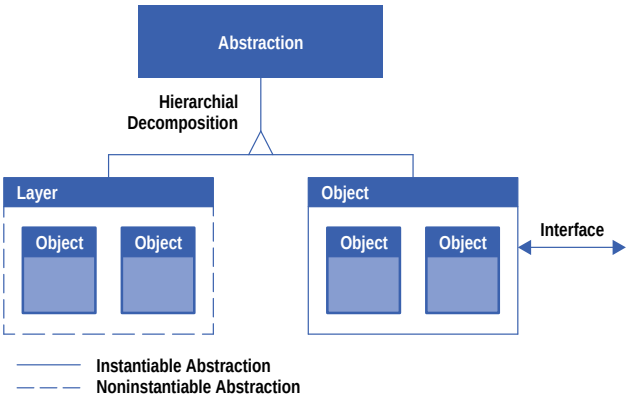


Fig. 7. *An abstraction decomposition.*

Much of this may not be discovered or decided until the design phase. However, knowing about it in the analysis phase maximizes the identification of abstractions and the completion of the analysis.

Creation Model

Many times discussions about abstractions resulted in intangibles that were difficult to grasp. To alleviate this problem, the team supplemented Fusion with a dependency model showing object dependencies and indicating when objects should be created. This provided developers with a concrete picture of which objects needed to be available first.

Consider again the HP stock price application. Let the mechanism object be represented by object A and let the policy object be represented by object B. Fig. 8 represents a creation model for the objects under discussion. It shows that object A has to be created before object B. This means that the mechanism for acquiring the HP stock price is created first. The object that determines how often to acquire HP stock price can only be created after object A. This example creation model is one of several that were discussed during the analysis phase to clarify the roles of the abstractions.

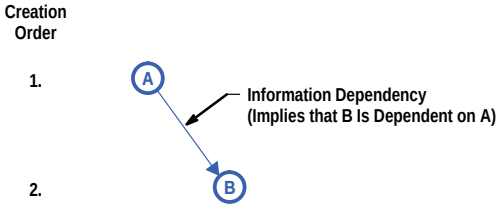


Fig. 8. *A creation model.*

Extensions to Fusion—Design Phase

We made extensions to the Fusion process with threads, design patterns, and reuse.

Threads

Our most extensive modifications to Fusion in the design phase were in the area of threads. Our real-time instrument firmware systems, which are very complex, must deal with asynchronous events that interrupt the system as well as send control commands to the measurement hardware. For example, measurement data from the analog-to-digital converter must be read within a certain time period before it disappears, and there may also be measurement hardware that needs to be adjusted based on these dynamic data readings.

There are also many activities going on in the system that may or may not be related. For example, we may want to have a continuous measurement running at the same time that we run some small routine periodically to keep the measurement hardware calibrated. Traditionally, a monolithic code construct performs all of these tasks. However, since some of these activities may only be peripherally related, it makes more sense to place these tasks in different threads of execution. Each thread of execution can be thought of as a path through the code. These threads of execution may be either regular processes or lightweight processes, and they may or may not share resources. In this paper, the term thread is used to mean a thread of execution, not necessarily to denote the preference of a lightweight process over a regular one. For instance, it would make sense to keep the task that performs the measurements separate from the task that checks the front panel for user input.

Fusion provides us with information on how to divide the behavior of the system into objects, but Fusion does not address the needs of our real-time multitasking system. It does not address how the system of objects can be mapped into different threads of execution, nor does it address the issues of interprocess communication with messages or semaphores. Lastly, no notation in Fusion can be used to denote the threading issues in the design documents.

Thread Factoring

We extended Fusion thread support in two ways. First, in the area of design we tried to determine how to break the system into different threads of execution or tasks. Second, in the area of notations we wanted to be able to be able to denote these thread design decisions in the design documents.

Our main emphasis was on keeping these extensions lightweight and easy to learn and keeping our modifications to the minimum needed to do the job. We wanted a simple system that would be easy to learn, rather than a powerful one that only a few people could understand.

We adopted portions of Buhr and Casselman's work on time-thread maps to deal with thread design issues such as the identification and discovery of threads of control within the system.[6,7,8] In our design, a time-thread map is essentially a collection of paths that are superimposed on a system object model (see Fig. 9). These paths represent a sequence of responsibilities to be performed throughout the system. These responsibility sequences are above the level of actual data or control flows, allowing us to focus on the responsibility flow without getting involved in the details of how the exact control flow takes place. We then applied the process of thread factoring, as described by Buhr and Casselman, where we brought our domain knowledge to bear on decomposing a single responsibility path into multiple paths. These paths were then mapped into threads of execution throughout our system.

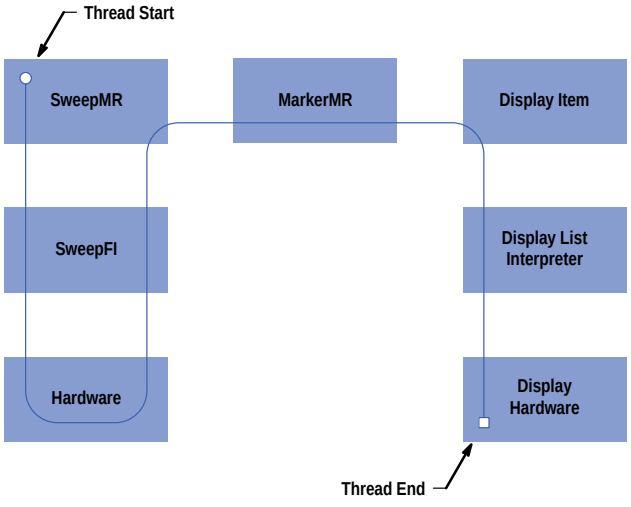


Fig. 9. *An example of a time-thread map.*

With the Fusion method, we had already identified the areas of responsibility. We then used this thread heuristic at the beginning of our design phase in those places where we had already identified the objects in the system, but where we had not yet designed the interaction among the objects. We dealt with the concurrency issues at the same time that we dealt with the object interaction graphs shown in Fig. 10. We also performed thread factoring and divided the system into multiple threads.

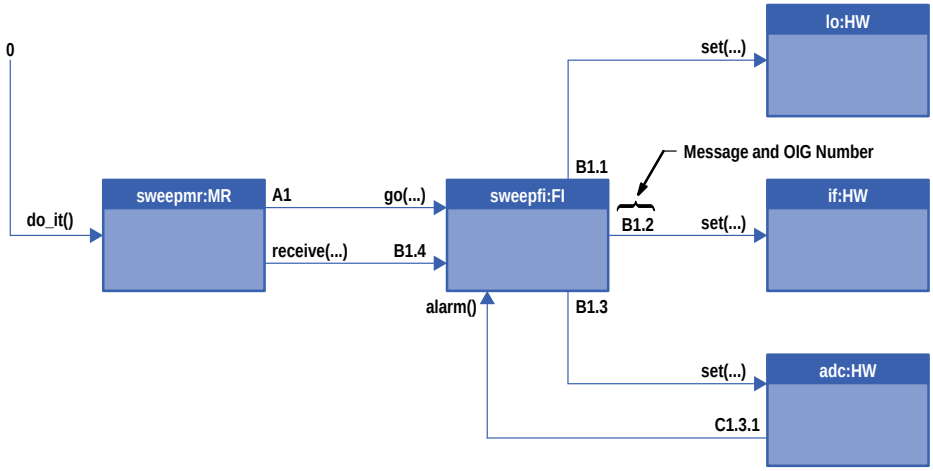


Fig. 10. *An object interaction graph (OIG). This representation is an extension of a Fusion object interaction graph. The letters in front of the OIG numbers associate a thread of execution with a particular message.*

The thread map in Fig. 11 depicts an example of thread factoring an application in our system. Using Fusion, we identified a path of responsibility through the objects CI, MR, and FI (client interface, measurement results, and fundamental information). Inputs enter the system through CI, and the responsibility for handling the input goes through the various layers of abstraction of MR and FI. Since information from the measurement hardware enters the system through FI, FI may have to wait for information. The information then flows goes back up fundamental information to MR and then possibly to other applications.

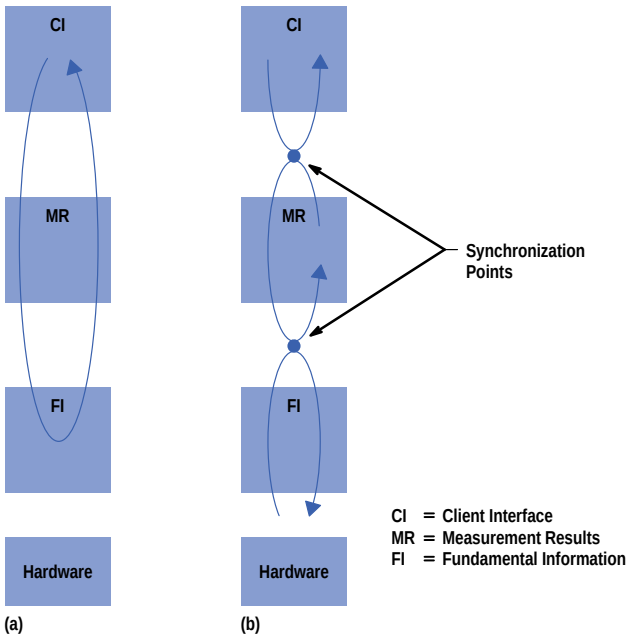


Fig. 11. *A thread map showing an example of thread factoring. (a) Before factoring. (b) After factoring.*

Clearly, the system worked fine as it was. However, we wanted to find where we could break the thread of execution and perform thread factoring. Many issues, such as questions about performance, were raised at this point. For example, if the thread is executing in part A of the system, it may not be available to perform services in part B of the system. Thus, in our system, we could have a thread pick up a user request to change the measurement hardware settings and then traverse the

code in the hardware setup routines to perform the hardware adjustments. However, while it was doing so, the thread would not be available to respond to user requests. This might impact the rate at which the system was able to service these requests. Therefore, we broke the user thread at the CI object boundary and gave that layer its own thread.

Next, we tried to find a place where we could break the thread that goes through MR and FI. Clearly, the place to break was between MR and FI. Making the break at this point gave us several flexibilities. First, we would be able to wait at the FI thread for data and not have to be concerned with starving MR. Second, developing components that were all active objects allowed us to mix and match components much more easily.

Mapping a system onto threads is a design-time activity. Thinking about the thread mapping at this stage allowed us to consider concurrency and the behavioral issues at the same time.

Thread Priorities

After we had identified the threads of execution, we needed to assign priorities to the threads. Note that this is mostly a uniprocessor issue, since priorities only provide hints to the operating system as to how to allocate CPU resources among threads.

In the firmware framework project, we took a problem decomposition approach. We reduced the architecture of our system to a pipeline of simple consumer/producer patterns (see Fig. 12). At the data source we modeled the analog-to-digital converter (ADC) interrupts as a thread producer generating data that entered the system with FI as consumer. FI, in turn, served as the producer to MR, and so forth. Inputs may also enter the system at the user input level via either the front panel or a remote device.

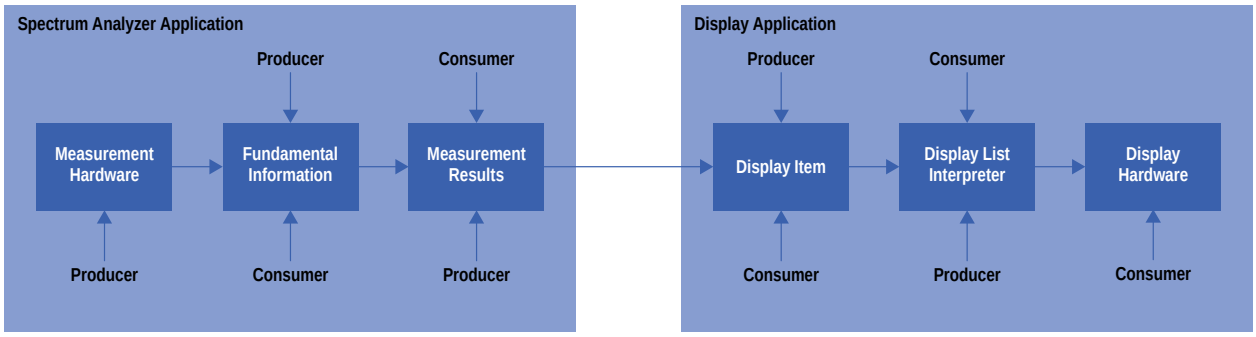


Fig. 12. *An example showing some of the producer/consumer chains used in the firmware framework project.*

We decided to give the highest priority to those threads that introduced data into the system from the physical environment so that they could respond to events in the environment quickly. Those threads included the user input threads and the ADC interrupt thread.

For thread priorities in the rest of the system, we considered three possibilities: that the producer priority was higher than that of the consumer, that the two priorities were equal, or that the consumer priority was higher than the producer priority. We ruled out setting the priorities to be equal because that would be equivalent to having no policy and would just let the systems run without any direction.

Making the producer priority higher than that of the consumer made sure that data was generated as quickly as possible. Unfortunately, since we continuously acquired data in our system, our data generation could go on forever unless we explicitly stopped the process and handed control to the higher level.

Alternatively, if we gave the consumer thread the higher priority, it would have priority over the producers with regard to CPU time. However, without the data generated from the producers, the consumers would block and be unable to run. Thus, if the data consuming chain had a higher priority than the data producers, the threads would run when data was available for them to process. This eliminated the necessity for the consumers to give up the CPU explicitly.

Threads and Synchronization

Another thread issue we considered was how to present the thread communication and synchronization operating system primitives to our system. We saw two alternatives. We could either expose the system level operating system calls to the system or encapsulate the operating system primitives inside objects so that the rest of the objects in the system could talk to these objects. For other system objects, it would be like communicating with nonoperating system objects.

We chose the latter approach. We created operating system objects such as tasks and semaphores to encapsulate operating system functionalities. This approach allowed us to model the operating system primitives as objects so that they would fit in well with the Fusion process and give us a clean model and good code reuse. This approach also had the side affect of

isolating our system from the operating system API. There were drawbacks with this approach, but they were not major. **Reference 7** contains more details about both of these approaches.

Thread Notation

We used thread notations within our Fusion diagrams in two ways. First, we used the thread map notations to show sketches of thread flows (Fig. 11). These simple notations gave us a general idea of the thread activities in the system. We adopted only a few of the notations that Buhr and Casselman use, employing the paths and waiting places that show the behavior of the system. We did not use their notation to handle the different types of synchronizations because we did not feel that this was the level of detail appropriate for what we needed. This method gave us an overview of what the system looked like without bogging us down in the details of how communication and synchronization were implemented.

For our second method of using thread notations, we extended the Fusion object interaction graph (OIG) notations to describe threads more formally (Fig. 10). We added letters in front of the OIG numbers to associate a thread of execution with a particular message. We also experimented with coloring the threads.

Design Patterns

Design patterns have become popular in the object-oriented world only recently. Design patterns evolved from the realization that certain software engineering patterns are repeated. These patterns are not at the implementation level, but at the level of component interactions. The idea here is to look at software design problems at a higher level so that recurring patterns can be identified and a common solution applied.

For instance, there is often a need in software systems for one or more objects to be notified if the state changes in another object. For example, if the value in a database changes, the spreadsheet and the word processor currently displaying that value need to change their displays. The *observer pattern*, described in the design patterns book,[3] shows how to set up the relationship among these objects. It describes when a pattern may be appropriate for solving the notification problem and some implementation hints and potential pitfalls.

Design patterns came to our attention a year or so into the project. By then, we had already completed most of the design. Therefore, we did not use them as templates to build the system from scratch. Instead, we used the design pattern catalog to validate designs. In looking through our system, we found potential applications for over half the patterns in the design patterns book. We then compared our design with those patterns.

We found patterns to be useful for design validation. In some places, they helped us improve the design. For instance, the hardware components are accessed through hardware handles, which are very similar to the protection proxies described in the patterns book. The hardware architecture itself is an example of a composite pattern. A composite pattern is an organization of objects in which the objects are arranged in a tree-like hierarchy in which a client can use the same mechanism to access either one object or a collection of objects. The descriptions of composite patterns in the design patterns book also helped us to identify and clarify some of the issues related to building composites.

In other areas in the system, we found our analysis to be more detailed because of our extensions to identify objects using the client-policy-mechanism framework. We have an event mechanism in the system to inform some component when an action has occurred. This mechanism is very similar to that of the observer pattern mentioned earlier. The observer pattern describes two components: a publisher and a subscriber, which define a methodology for handling events.

Our event pattern is slightly more sophisticated. We placed the event policies into a third object, so we have three components in our event pattern: a subscriber, an actor (publisher), and the event itself. Actors perform actions, and subscribers want to know when one or more actors have performed some action. The subscriber may want to be notified only when all of the actors have completed their actions. Thus, we encapsulated policies for client notification into the event objects. An actor is only responsible for telling events that it has performed some action. Events maintain the policy that determine when to notify a subscriber.

This arrangement gives more flexibility to the system because the design-patterns approach allows the policy for notification to be embedded in the actor. In our case, we also have the freedom to customize the policy for different instances of the same actor under different situations.

We feel that the main advantage of not using the patterns until the system design is done is that the developer will not fall into the trap of forcing a pattern that resembles the problem domain into the solution. Comparing our problem domain with those described in the patterns book helped us to understand more about our context and gave us a better understanding of our system. Also, as many other object-oriented practitioners have reported, we also found patterns to be a good way to talk about component interaction design. We were able to exchange design ideas within the team in a few words rather than having to explain the same details over and over again.

Scenarios

Part of our system requirements included developing scenarios describing the behavior of the system. Scenarios describe the system output behavior given a certain input. These scenarios are similar to the use cases described in reference 1 and are part of the Fusion analysis phase. However, for people not conversant in object-oriented methods, these scenarios often do

not have much meaning because the descriptions are far above the implementation level. Whenever we presented our analysis and design models, our colleagues consistently asked for details on what was happening in the system at the design level. Although Fusion documents provided good overviews of the system as well as excellent dynamic models for what happened in each subsystem, people still wanted to see the dynamics of the entire system.

To explain how our system works, we developed scenarios during the design phase. These scenarios were a collection of object interaction graphs placed together to show how the system would work, not at an architectural level but at a design and implementation level. We used the feedback we received from presenting the scenarios to iterate the design.

The Fusion model is event-driven, in that an event enters the system and causes a number of interactions to occur. However we had a real-time system in which events happen asynchronously. We needed scenarios that were richer than what the object interaction graph syntax could provide.

For example, our instrument user interface allows the user to modify a selected parameter simply by turning a knob, called the RPG (rotary pulse generator). One attribute by which our customers judge the speed of our instruments is how quickly the system responds to RPG input. The user expects to get real-time visual feedback from the graphics display. The empirical data suggests that real-time means at least 24 updates per second. As the layers were integrated, we looked at the scenario in which the user wanted to tune the instrument by changing a parameter (e.g., center frequency). This scenario led to questions such as: How would the system's layers behave? What objects were involved? What were the key interfaces being exercised? Were the interfaces sufficient? Could the interfaces sustain the rate of change being requested? What performance would each of the layers need to deliver to achieve a real-time response from the user's point of view? The answer to these questions led to a refinement of both the design and the implementation.

These design-level scenarios provided a better idea of what would happen in the system and presented a more dynamic picture. Since the scenarios encompassed the entire system, they gave the readers a better view of system behavior. We found them to be good teaching tools for people seeking to understand the system.

We also found that instance diagrams of the system objects helped us to visualize the system behavior. A diagram of the instantiated objects in the system provided a picture of the state information that exists in the system at run time.

Reuse

To build reuse into a system, the development method has to support and make explicit the opportunities for reuse. The analysis extensions described earlier serve to facilitate the discussion of reuse potential in the system. The design is driven by the biases encoded into the analysis.

At the end of the first analysis and design pass, an entity relationship diagram will exist and a rudimentary class hierarchy will be known. The more mature the team in both object-oriented technology and the domain, the earlier the class hierarchy will be identified in the development method. Additional information can be gathered about the level of reuse in the class hierarchy during the analysis and design phase. These levels of reuse are:

- Interface reuse
- Partial implementation reuse
- Complete implementation reuse.

The ability to note the level of reuse in the work products of the development method is valuable to the users of the object model. A technique developed in this project was to color code the object model. Fig. 13 shows two of these classes.

Except for defect fixes, complete implementation classes cannot be modified once they are implemented. This type of color coding aids developers to know which components of the system can be directly reused just by looking at the object model.

Process Definition

The pursuit of object-oriented technology by a team necessitates the adoption of formal processes to establish a minimum standard for development work. This is especially true if the team is new to object-oriented technology. Various members of the team will develop their understanding of the technology at different rates. The adoption of standards enables continuous improvements to the process while shortening the learn time for the whole team.

In the firmware framework project, we adopted processes to address issues like communication, quality, and schedule. We customized processes like inspections and evolutionary delivery to meet our needs. It is important to keep in mind that processes described in the literature as good practices need to be evaluated and customized to fit the goals of a particular team. The return on investment has to be obvious and the startup time short for the process to have any positive impact on the project.

Coding standards, for example, can help the team learn a new language quickly. They can also be used to filter out programming practices that put the source code at risk during the maintenance phase of the project. They also facilitate the establishment of what to expect when reading source code.

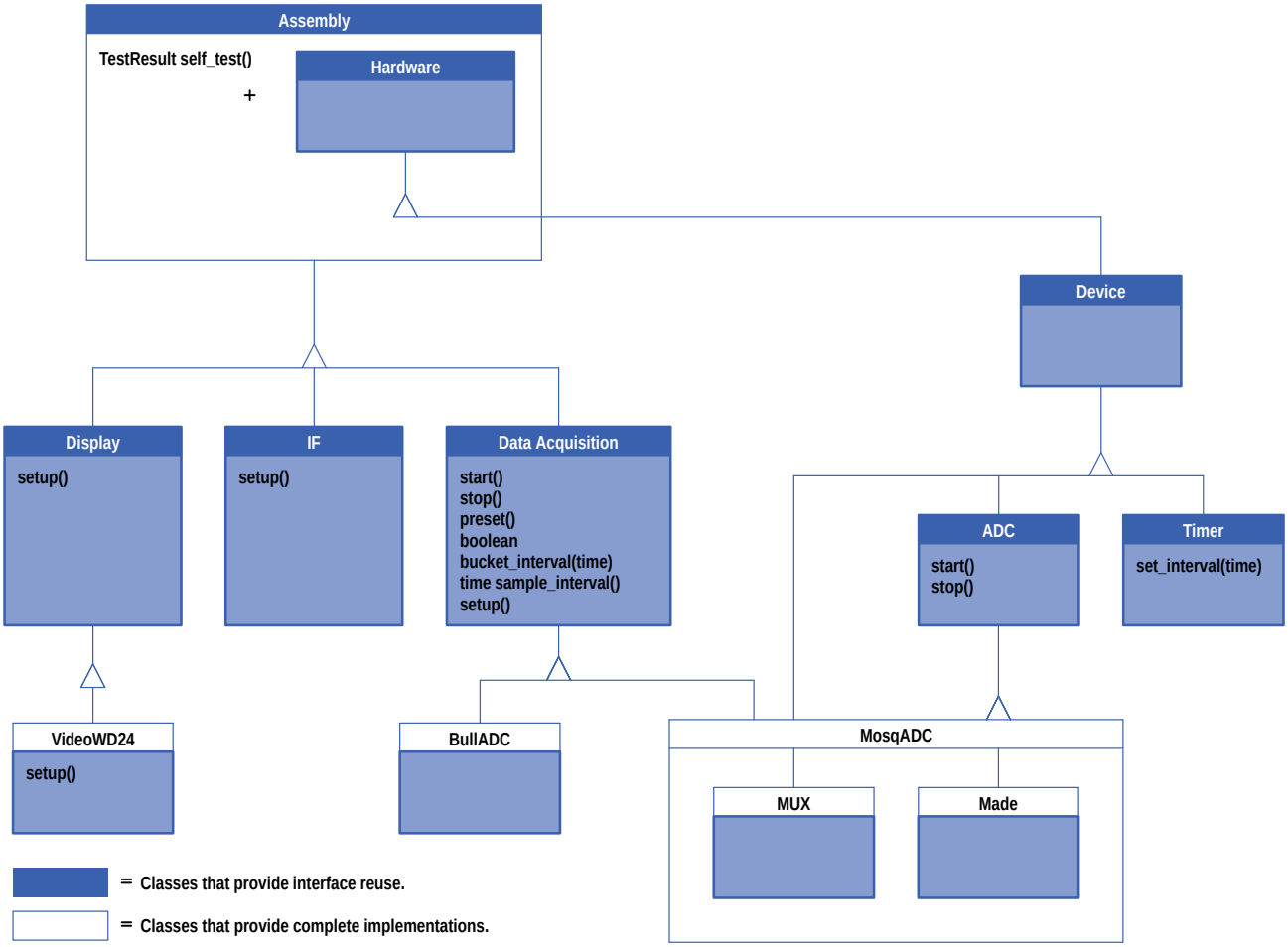


Fig. 13. *An example of an object model of the hardware layer that is coded to show the reuse status of the various objects.*

Evolutionary Delivery

We partnered with HP's Software Initiative program to develop what is now known as EVO Fusion.[9,10] EVO is a management process that constructs a product in small increments. After each increment is completed, the development process is examined for improvements that might contribute towards the successful completion of the next increment.

Each increment can have an analysis, design, code, and test phase. The product evolves over time into the desired product through repeated execution of small development cycles that add greater and greater functionality. This process helps to focus the team and increases the probability that schedules will be met.

Inspections

Much has been written about the value of inspections to software development. Though much of the literature focuses on product quality, the inspection process also identifies (that is, makes explicit) other issues. Once identified, these issues can be quantified into high, medium, and low risk factors. Their impact on the success of the project can be ascertained and the appropriate action can be taken to manage their resolution in a timely manner. Institution of an inspection process thus provides the project manager and the project team with an additional means by which to gather information pertinent to project completion.

In a project, the use of a development method like EVO Fusion, coupled with an inspection process, facilitates the discussion of issues that relate to requirements, software architecture, software integration, and code development. The benefits to the team are significant because these processes enable members to understand the product and its functionality long before the debug phase begins.

Legacy Systems

In many cases, it is not possible to generate a system completely from scratch without using legacy code. The firmware framework project was no exception.

We found that the most straightforward approach is to encapsulate the legacy code inside objects. This works for systems that provide services to client objects. It also works for legacy subsystems that act as clients, such as language parsers. These parser components are not good framework citizens because they already have their own definition of the server interface they expect, which may not coincide with the object-oriented design.

We feel that the proper approach is to perform the object-oriented analysis and design without regard for the legacy system first, and then encapsulate the legacy code inside the proper objects. There is a strong temptation to design the object-oriented system around the existing legacy code, but in our experience the legacy system may not have been designed with the appropriate object-oriented principles. Thus, allowing it to affect the analysis may lead to a faulty design.

Summary

Fusion is the result of the evolution of a long line of software development processes. Like its predecessors, Fusion has its benefits, problems, and areas for improvement.

Benefits. The benefits we derived using Fusion include:

- Lightweight and easy to use. We found Fusion to be easy to learn. There is lot of guidance in the process that leads the user from step to step. It is not mechanical, but the user will not be wondering how to get from one step to the next.
- Enforces a common vocabulary. Often in architecting systems, the different domain experts on the team will have their own definitions of what certain terms mean. Generating data dictionary entries at the analysis phase forces everyone to state their definitions and ensures that misunderstandings are cleared up before design and implementation.
- Good documentation tool. We found that the documents generated from the Fusion process served as excellent documentation tools. It is all too easy, without the rigor of a process, to jump right in and start coding and do the documentation later. What often happens is that schedule pressure does not allow the engineer to go back and document the design after the coding is done.
- Hides complexity. Fusion allows a project to denote areas of responsibility clearly. This feature enables the team to talk about the bigger picture without being bogged down in the details.
- Good separation between analysis and design. Fusion enforces a separation between analysis and design and helps in differentiating between architectural and implementation decisions.
- Visibility graphs very useful. The visibility graphs are very useful in thinking about the lifetime of the server objects. Simply examining the code all too often gives one a static picture and one does not think about the dynamic nature of the objects.

Problems. The problems we encountered with the Fusion method included:

- Thread support. Although the Fusion method models the system with a series of concurrent subsystems, this approach does not always work. The threads section of this article describes our problems with thread support.
- Complex details not handled well. This is a corollary to Fusion's ability to hide details. Do not expect Fusion to be able to handle every last detail in the system. In instrument control, there are a lot of complex data generation algorithms and interactions. Although in theory it is possible to decompose the system into smaller subsystems to capture the design, in practice there is a point of diminishing returns. It is not often feasible to capture all the details of the design.

Areas for Improvement. The following are some of the areas in which the Fusion method could be improved:

- Concurrency support. We would like to see a process integrated with the current Fusion method to handle asynchronous interactions, multitasking systems, and distributed systems.
- CASE support. We went through the Fusion process and generated our documentation on a variety of word processing and drawing tools. It would have been very helpful to work with a mature CASE tool that understands Fusion. Some of the functionalities needed in such a tool include: guidance for new Fusion users, automatic generation of design documents, and automatic checking for inconsistencies in different parts of the system. Throughout the course of our project we evaluated several Fusion CASE tools, but none were mature enough to meet our needs.

Acknowledgments

The authors wish to thank the other members of the firmware framework team: David Del Castillo, Manuel Marroquin, Steve Punte, Tony Reis, Tosya Shore, Bob Buck, Ron Yamada, Andrea Hall, Brian Amkraut, Vasantha Badari, and Caroline Lucas. They lived the experiences and contributed to the knowledge described in this paper. We'd like to also recognize Todd Cotton from the HP Software Initiative (SWI) team who, as a part-time team member, helped us develop our EVO process. Our gratitude also goes to the rest of the HP SWI team for the support they gave us during the project. Thanks to Derek Coleman for helping us use Fusion. Finally, we would like to express our appreciation to Ruth Malan; without her encouragement this paper would not have been possible.

References

1. I. Jacobson, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, 1992.
2. D. Coleman et al., *Object-Oriented Development: the Fusion Method*, Prentice Hall, 1994.
3. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*, Addison-Wesley, 1995
4. R. Malan, R. Letsinger, and D. Coleman, *Object-Oriented Development at Work: Fusion in the Real World*, Prentice Hall, 1996.
5. R. Wirfs-Brock, "Object-Oriented Design: A Responsibility-Driven Approach," *OOPSLA '89 Conference Proceedings*, pp. 71-75.
6. R.J.A. Buhr and R.S. Casselman, "Timethread-Role Maps for Object-Oriented Design of Real-Time and Distributed Systems," *OOPSLA '94 Conference Proceedings*, pp. 301-316.
7. R.J.A. Buhr and R.S. Casselman, *Use of CASE Maps for Object-Oriented Systems*, Prentice Hall, 1996.
8. R.S. Casselman et al., *Notation for Threads*, SCE-92-07, Department of Systems and Computer Engineering, Carleton University, September 1992.
9. T. Cotton, "Evolutionary Fusion: A Customer-Oriented Incremental Life Cycle for Fusion," *Hewlett-Packard Journal*, Vol. 47, no. 4, August 1996, pp. 25-38.
10. T. Gilb, *Principles of Software Engineering Management*, Addison-Wesley, 1988.

An Approach to Architecting Enterprise Solutions

A frequently mentioned ailment in healthcare information management is the lack of compatibility among information systems. To address this problem, HP's Medical Products Group has created a high-level model that defines the major architectural elements required for a complete healthcare enterprise information system.

by Robert A. Seliger

HP's Medical Products Group (MPG) produces medical devices such as patient monitors and ultrasound imaging systems, which obtain physiological data from patients, and clinical information systems, which document, retrieve, and analyze patient data.

In December 1994, MPG directed its architects to define and drive the implementation of an open, standards-based MPG application system architecture that would enable:

- Improved application development productivity
- Faster times to market
- Seamless integration of applications developed by MPG and its partners
- Integration with contemporary and legacy systems in an open standards-based environment

To meet these objectives and to help establish MPG as a leader in healthcare information systems, the Concert architecture was conceived. Concert is a software platform for component-based, enterprise-capable healthcare information systems.

The primary objective of Concert is to enable the decomposition of healthcare applications and systems of applications into sets of interconnectable collaborative components. Each component implements important aspects of a complete healthcare application or system of applications. The components work together to realize fully functional applications and systems of applications.

A component-based approach was pursued to leverage the fundamental precepts of good software engineering: *decomposition*, *abstraction*, and *modularity*. We reasoned that an architecture that facilitated decomposing large complex systems into modular components and abstracted the details of their implementation would contribute to development productivity. The ability to use these components in a variety of applications would expedite time to market.

Carefully specified component interfaces would enable flexible integration of components in a seamless manner. Openly publishing these interfaces would enable components developed by MPG's partners to interoperate with MPG's components. The judicious use of healthcare and computing standards would enable integration with systems based upon other architectures.

Concert was developed by MPG in conjunction with HP Laboratories and the Mayo Clinic, a strategic MPG partner. It serves as the technical cornerstone for MPG's group-wide initiative to provide better enterprise solutions for its customers. Key aspects of the architecture have also been applied by HP Laboratories and the Mayo Clinic to develop a prototype electronic medical record system.

Concert also serves as the foundation for the technical development effort of the Andover Working Group for Open Healthcare Interoperability. This MPG-led healthcare industry initiative was been formed to achieve enterprise-wide multivendor interoperability (see **Subarticle 2**).

Concert currently consists of the following elements:

- A general reference model that organizes the architecture of healthcare enterprise information systems into a key set of architectural ingredients
- A model for software components that can be implemented using CORBA-based[1] or Microsoft® OLE-based[2] technologies
- An initial set of Concert components including their interfaces and the policies that govern the patterns of interaction between the components
- An approach for organizing Concert component interfaces to represent component application development, system integration, and system management capabilities

- An initial information model that provides an object-oriented description of healthcare terms, concepts, entities, and relationships to establish a common clinical basis for Concert components and the applications developed from them.

Concert Components

To facilitate the description of the Concert component model, an example of one of the components that MPG has developed will be used. The component, called an *enterprise communicator*, is at the heart of the enterprise communication framework (ECF) that MPG is developing in conjunction with other healthcare vendors and providers that form the Andover Working Group.

An enterprise communicator is a software component that facilitates healthcare standards-based data interchange between healthcare systems and applications within a healthcare enterprise. Different types of communicators encapsulate different healthcare standards. The particular communicator that MPG is currently developing encapsulates the Health Level 7 (HL7) 2.2 data interchange standard.[3] Fig. 1 shows a system based on enterprise communicators.

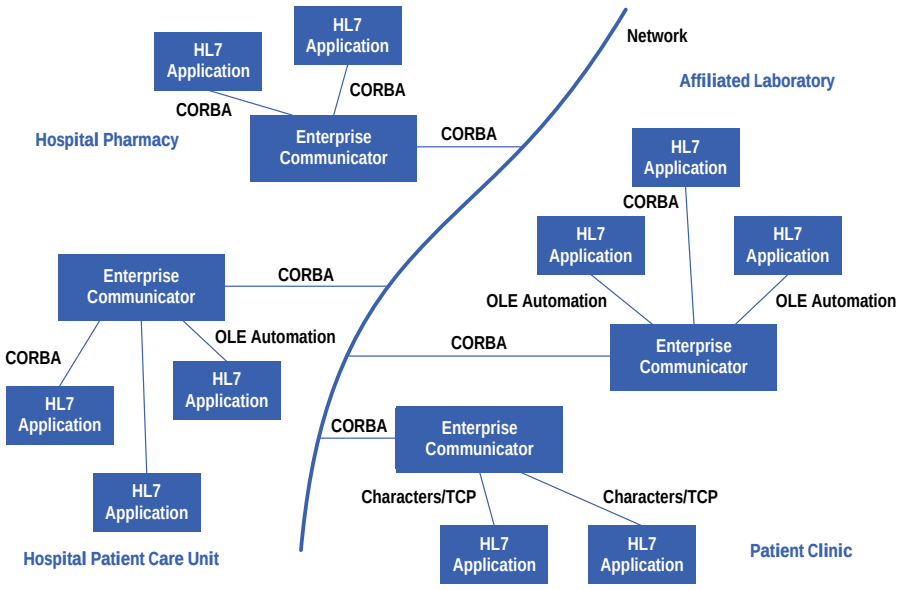


Fig. 1. *A healthcare system based on enterprise communicators and the HL7 data interchange standard.*

HL7 is a widely used healthcare electronic data interchange standard. Its primary contribution is the specification of a set of messages that healthcare systems can exchange to share clinically relevant data. Examples include messages that enable applications to obtain the results of laboratory tests from the applications that have access to this data.

The HL7 standard is not intended to be particularly prescriptive in terms of messaging technology or how messaging services should be implemented. This has led to a variety of custom HL7 implementations based on a range of technologies. A typical implementation employs specially formatted character-encoded messages and point-to-point network or serial-line connections. An example of a character-encoded HL7 message is shown in Fig. 2.

In the Concert-based model, applications employ enterprise communicators to broker their HL7 data interchange needs. Enterprise communicators provide applications with the necessary messaging capabilities, such as guaranteed message delivery and multicasting (i.e., sending several messages at once). Enterprise communicators also present HL7 messages as object-oriented abstractions using both CORBA and OLE automation technologies. This eliminates the need for applications to parse the messages to extract the encoded data.

In addition, for legacy integration, communicators support TCP/IP interfaces through which applications that are not object-oriented can send and receive character-encoded HL7 messages.

Why Components?

The concept of component-based systems has become increasingly popular over the last several years. There are currently many definitions of components and a variety of tools and technologies have emerged to facilitate developing component-based systems. Many of the general concepts about what a component is are similar across all of these definitions. However, there appears to be little agreement on the granularity of a component. Granularity depends on how much functionality a component represents and how much code and complexity are embodied within a component implementation.

```
MSH|^~\&|HP|HOSPITAL|AWG|GENERAL CLINIC|199608271353|
ADT^AO1|56844_1_AA|T|2.2|<cr>
EVN|AO1|199608271353|<cr>
PID|l| |102983|3106|DOE^JOHN^^SR^^ | |19450112|M|
AWG^TEST^^^^~ANDOVER^GROUP^^^^ |H|1400 MAIN ST^ ROOM 6263^ANDOVER^
MA^10810| |(508)555-1022|862-1022| |M|CAT|14563|838-29-4938|
<13>NK1|1|DOE JOAN B |CHD|101 MAIN STREET APT^2A^BOSTON^MA^05404|
(508)555-0000|(508)555-0000|<cr>
NK1|2|DOE JANE^^^^|CHD|434 NORTH STREET^APARTMENT 5B^CHELMSFORD^MA^
05401|(508)555-1111| |<Cr>
PV1|1|I|1N^107 A|ELE| | |36^HEART^THOMAS^MD^^^| | |IMX| | | |1| |Y|
36^HEART^THOMAS^MD^^^|0|14563| | | | | | | | | | | | | | | | | |
| | 1W^102^W| |199608191015|<cr>
PV2| |s| |<cr>
```

basically means ...

Admit patient John Doe Sr., whose wife is Joan and next-of-kin is Jane, and whose physician is Dr. Heart, to General Clinic

Fig. 2. *A character-encoded HL7 admit patient message.*

In Concert, components tend to be medium-to-large-grained objects.[4] For example, a Concert component might be implemented by what is traditionally thought of as an executable program, as is the case for an enterprise communicator. Alternatively, a group of Concert components might be packaged within a library. However, a Concert component is rarely as small as a single C++ or Smalltalk object.

In general, a Concert component is a portion of an application system that:

- Implements a substantial portion of the overall application system's capabilities
- Represents its capabilities via one or more modularly defined binary interfaces
- Can be developed independently of other components
- Is capable of efficiently communicating with other components over a network
- Is the fundamental unit of configurability, extensibility, replaceability, and distribution
- Is the basis for an open system through the publication of its interfaces.

In other words, a Concert component represents a significant portion of an overall application system, but is small enough to enable efficient and flexible composition with other components to form full-fledged applications and application systems.

A key motivation for a component-based architecture is that it makes accomplishing the following architectural objectives much easier.

- Simplification. Components can make the approach to decomposing a complex application system into smaller simpler pieces tangible and precise.
- Replaceability. Existing components can be readily replaced with new implementations as long as the new component supports the same interfaces as the component it replaces.
- Configurability. Components provide a modular, precise, and manageable basis for configuring a system.
- Extensibility. New components with new capabilities can be added to an existing system in a modular and organized manner. The risk of breaking existing capabilities that are well-encapsulated in existing components is minimized. In addition, new capabilities that are added to existing components can be represented by new component interfaces that represent the new capabilities without requiring changes to existing interfaces.
- Independence. The interfaces between components define the "contract" between components that can enable independent development as long as the contracts are respected.
- Scalability. Components can be physically distributed or alternatively collocated depending upon the computing infrastructure available and desired price/performance profile. The component interfaces define what components communicate about, and this communication can be realized using same-machine or network-based mechanisms.
- Stability. A variety of tools, technologies, and design methods can be employed to implement the components, thereby enabling evolution of the implementation technology, tools, and methods without violation of the architecture.
- Business-Centeredness. The efficient and timely realization of the architectural objectives listed above is the basis for a significant competitive advantage.

To achieve these objectives, Concert specifications primarily emphasize how application systems are assembled from components. This approach provides a great deal of latitude for application developers to define what capabilities their application systems will actually provide. Perhaps most important, the architecture also enables product teams to put more focus on developing the content of their applications because they can leverage a standard approach to constructing their application systems.

Component Interfaces

Concert components implement object-oriented interfaces. An object-oriented interface is a named grouping of semantically related operations that can be performed on a component. A component that implements a particular interface implicitly supports the functionality contract implied by the interface.

For example, among the interfaces that an enterprise communicator implements is the ApplicationConnect interface. This interface enables an application to connect to and disconnect from a communicator. Only connected applications can send and receive HL7 messages.

Components that implement similar capabilities represent these capabilities via the same interface. For example, any Concert component that requires its client applications to explicitly connect and disconnect might implement the ApplicationConnect interface. The effect of connecting and disconnecting would depend upon the type of component, but the policies governing when and how to use the interface would be the same.

Each object-oriented interface enables a subset of a component's overall capabilities to be accessed and applied. A component's full set of object-oriented interfaces enables a component's full array of capabilities to be accessed and applied. For example, another interface that is implemented by an enterprise communicator is MessageManager. This interface enables a connected application either to create a new message that can be populated with data and sent or to obtain a message that the communicator has received from another application.

Many of the details of the Concert model for software components come from the OMG's Object Management Architecture (OMA). The most notable OMA ingredient is the use of the OMG Interface Definition Language (OMG IDL) for specifying a Concert component's object-oriented interfaces independent of the technology used to implement the component and its interfaces.

OMG IDL serves as the software equivalent of the schematic symbols that electrical engineers use to diagram circuits. For example, the symbol for an AND gate clearly conveys its role without relying on descriptions of the underlying circuitry or fabrication technology (e.g., CMOS, TTL, etc.).

OMG IDL provides a standard and formal way to describe software component interfaces. Further, when applied within the context of an overall component-based architecture, formally specified interfaces can be used to create a level of precision that helps ensure that important architectural features and principles are reflected in products that are eventually developed. For example, components that constitute a particular product can be examined to see if they correctly implement the necessary interfaces. The role that interfaces play in adding precision to a software architecture is illustrated in Fig. 3.

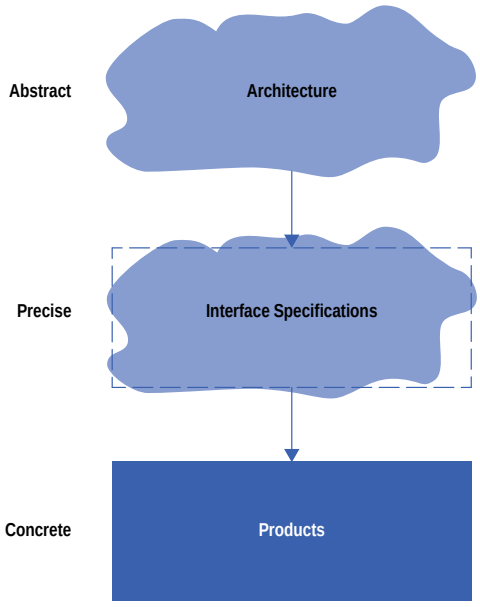


Fig. 3. *An illustration of the role that interfaces play in adding precision to a software architecture.*

Another advantage of defining interfaces is that they can provide a shorthand for describing components. The Concert specification currently consists of less than forty interfaces. Just the name of the interfaces that a component implements is often all one needs to understand how to use the component.

For example, the enterprise communicator interface ImplementationInformation allows access to implementation information about a communicator, including its product number, software revision, and when it was installed on its current host. The interface HostInformation provides access to information about the computer that is hosting an enterprise communicator, including the host's network name and the type of operating system it supports.

A simplified OMG IDL specification for an enterprise communicator's ApplicationConnect interface is shown in Fig. 4. This specification conveys the following information about the interface:

- The name of the interface is ApplicationConnect.
- The interface supports two operations: connect and disconnect. An application that wants to connect to an enterprise communicator performs the connect operation on the communicator's ApplicationConnect interface. An application that wants to disconnect performs the disconnect operation. In either case, the application identifies itself by setting an appropriate value for the input parameter which_application.
- Under normal conditions, neither operation returns any data. However, they can raise exceptions. An operation that has encountered an abnormal condition can communicate this fact to its client by raising an exception. When an operation completes, its client is able to determine whether or not the operation completed normally or has raised an exception.

```
interface ApplicationConnect : Composable {
 exception UnknownApplication {};
 exception AlreadyConnected {};
 exception NotConnected {};
 void connect (in ApplicationIdentifier which_application)
   raises (UnknownApplication, AlreadyConnected);
 void disconnect (in ApplicationIdentifier which_application)
   raises (UnknownApplication, NotConnected);
} ;
```

Fig. 4. *An example of an interface definition.*

Different types of exceptions can be defined, each of which represents a different abnormal condition. The exception UnknownApplication indicates that the application identified by the parameter which_application is not known to the enterprise communicator. The exception AlreadyConnected indicates that the application that is trying to connect is already connected to the enterprise communicator. The exception NotConnected indicates that an application that is trying to disconnect is not currently connected to the enterprise communicator.

Another important characteristic of the ApplicationConnect interface is that it inherits the definition specified for the interface Composable, which is described in the next section.

Multiple Interfaces. Additional ingredients of the Concert model for components were leveraged from Microsoft's Component Object Model (COM). While much of COM describes the low-level conventions for performing operation invocations on objects, COM also motivates the concept of representing a component through multiple distinct interfaces.

In COM, a client must explicitly ask a component whether it supports a particular interface before it can access the interface. If the component does indeed support the interface, the client can use it. Otherwise, the client must seek another interface, or try to make do with the interfaces that are supported. See Subarticle 3, "***Multiple Interfaces in COM***."

Although typically described by Microsoft as a way to evolve component functionality through the addition of new interfaces and as a way to simplify perceived problems with object-oriented inheritance, the real strength of multiple interfaces is the ability to model complexity.

For example, in Concert, components represent significant subsets of the overall functionality of an application or system of applications. It would be unwieldy to try to represent a component's complete set of capabilities through a single interface. It would be unnatural in many cases to impart modularity by organizing these interfaces using inheritance.

As a simple real-world example of multiple interfaces, consider the interfaces that might represent an employee who is also a father and a baseball fan. It is unnatural to model this employee's interfaces using an inheritance relationship because the interfaces are semantically unrelated. It would be awkward to define a single employee-specific interface because the

advantages of developing distinct models for the concepts of the employee as father and baseball fan become obscured. However, modeling the employee as supporting multiple interfaces is essentially how things work in the real world.

Concert's adaptation of the COM concept of multiple interfaces is referred to as *interface composition*. This is because a component's functionality is represented by a composition of distinct interfaces. The interfaces that the component chooses to include in this composition can vary over time as a function of the component's internal state or because its underlying implementation has changed.

The interfaces in a Concert interface composition are referred to as *composable* interfaces. In Concert, all composable interfaces are derived from the base interface Composable. The interface Composable provides functionality similar to COM's IUnknown:. It supports a method similar to QueryInterface which enables a component's client to determine whether the component implements a particular interface, and if so, to obtain a reference to the interface. For convenience, this querying capability is available via any Composable interface.

In addition, every component implements the interface Principal, which is also derived from Composable. In addition to providing a way for a component's clients to interrogate a component about the interfaces it supports, Principal also enables clients to obtain a list of all of the interfaces that the component currently implements. For some clients the ability to obtain a list of available interfaces is preferred to the technique of interrogating for interfaces one at a time.

The primary difference between COM and Concert in terms of support for multiple interfaces is that in COM, the concept only applies to components implemented using COM-based technology. In Concert, the concept has been easily layered on top of a variety of technologies, including COM, CORBA, and even Smalltalk and C++. This enables Concert to apply a powerful architectural notion in a technologically flexible manner. Fig. 5 shows an enterprise communicator's implementation of multiple interfaces.

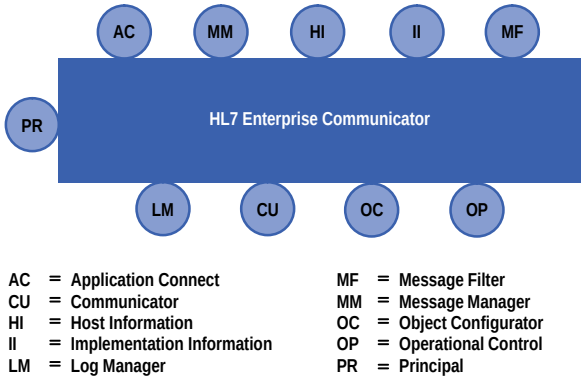


Fig. 5. *An enterprise communicator's multiple interfaces.*

Channels. Concert's object-oriented component interfaces are primarily intended to be implemented using CORBA-based or OLE-based technologies. However, certain component capabilities are better suited for other representations that are not necessarily object-oriented or for implementations using technologies other than CORBA or OLE. For example, backwards compatibility with existing standards or stringent performance constraints might dictate the use of other technologies.

In Concert, a component can have interfaces that are not object-oriented. These interfaces are referred to as *channels*. Channels generally do not offer access to the full set of component capabilities that are represented by a component's object-oriented interfaces, but they do provide an architectural basis for representing alternative communication mechanisms.

For example, an enterprise communicator implements TCP/IP channels over which it can send and receive character-formatted HL7 messages. Contemporary applications use a communicator's object-oriented interfaces to send and receive messages, but legacy applications can use a communicator's TCP/IP channels.

Interface Perspectives. During the early development of Concert, most of the emphasis was on the interfaces that represented a component's application capabilities. These interfaces support the ability to use the component to construct healthcare applications.

For example, the application capabilities of an enterprise communicator are represented by the following three interfaces:

- The application connect interface enables an application to connect to and disconnect from a communicator. When connected, an application can send or receive HL7 messages. When disconnected, messages will be buffered for the application until the next time it connects.
- The message manager interface enables an application to create new empty messages that it can fill with data and send and also receive messages that have been sent by other applications.

- The message filter interface enables an application to instruct an enterprise communicator to filter messages based upon their data content. Messages that are filtered are not delivered to the application. For example, an application might only want to receive messages that pertain to a particular patient. The enterprise communicator will send to the application only those messages that pertain to the indicated patient.

It was soon recognized that the application construction interfaces represented only one perspective for defining a component's interfaces and that there were other perspectives that needed to be represented. Specifically, within a healthcare enterprise, there are at least two other perspectives that need to be considered:

- System integration perspective, which is concerned with interconnections within and between systems for the purpose of establishing interoperation (typically based upon relevant standards).
- System management perspective, which is concerned with how systems are configured, monitored, administered, and maintained to preserve desired availability and performance levels.

These perspectives turn out to be extremely important as soon as one starts to address basic issues such as how a component is started or halted, or how data within a component is accessed by systems and applications that are not component-based.

For example, with an enterprise communicator, there are two system integration interfaces. One is an object-oriented interface that enables a communicator to send and receive binary-encoded HL7 messages. The other is a TCP channel that enables a communicator to send and receive ASCII-encoded HL7 messages.

For system management purposes, a communicator supports seven object-oriented interfaces and one SNMP-based channel. The breadth of functionality needed to manage a communicator exceeds the functionality needed to use it for application purposes. While this situation was surprising at first, it is consistent with the notion that enterprise-capable components must be inherently manageable. For example, it would not be practical to deploy communicators throughout an enterprise if there were no way to monitor their performance and intervene from a central location when problems occur.

The concept of organizing a component's interfaces in terms of application construction, system integration, and system management perspectives is one of the cornerstones of Concert. It is this way of thinking about components that has enabled Concert to provide the basis for components that are truly capable of enterprise-wide deployment and use.

In general, the interfaces that make up these three perspectives can be thought of as providing an architectural foundation for component use. Well-defined interfaces organized in a useful way lower the obstacles to using components in a black-box manner to construct systems.

There is, however, a fourth perspective defined in Concert. The component customization perspective represents the concept that a component may have internal interfaces that are similar to traditional application programming interfaces. These interfaces can be used to modify a component's functionality. The important distinction from an architectural perspective is that the customization interfaces offer access to a component's implementation and should not be confused with the external view offered by the interfaces for the other perspectives.

Hardware analogies for software systems are often a stretch, but the following analogy for a Concert component and its various interface perspectives has proven to be effective. A Concert component has sophistication that is roughly analogous to a printed circuit board, such as a sound card that one might plug into a personal computer. The sound card provides application construction interfaces for programs that enable the user to create and control sounds.

The sound card also provides:

- System integration interfaces so that the sound card can be used in conjunction with external MIDI-based instruments (i.e., instruments that support the Musical Instrument Digital Interface) or with an audio speaker
- System management interfaces, often in the form of LEDs that indicate the card's status and DIP switches that enable configuring the card (e.g., setting interrupt vectors or resetting the card's processor)
- Customization interfaces, such as sockets for additional memory chips, which enable changing the functionality of the card, and unlike the card's other interfaces, expose aspects of the card's implementation.

These key component-interface perspectives are illustrated in Fig. 6.

The principles for organizing and defining interfaces for Concert components in terms of these perspectives has proven to be productive and straightforward to implement using both CORBA and OLE automation technology. The work to conceive, specify, and then hone the definition of each interface can be considerable, but the rewards can be substantial.

A well-thought-out and stable set of interface definitions has enabled component design and implementation to proceed at a brisk pace. Further, the interface definitions form a rich basis for an interesting form of reuse referred to as *specification reuse*. Some of the behaviors for new types of components can be reused from the set of interfaces and associated patterns of use that have already been defined.

The use of a common and relatively constrained set of component interfaces across MPG and its partners will enable components to be developed by a single MPG team, by teams in different MPG organizations, and by one or more of MPG's

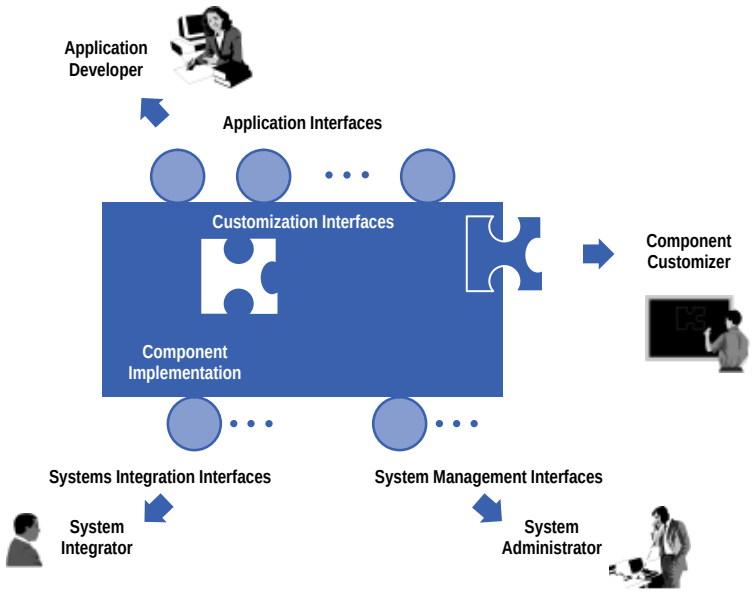


Fig. 6. *Component interface perspectives.*

partners. These interfaces also serve as the basis for open MPG systems. The interface definitions are the key points at which the systems can be opened.

Components and the Architecture Reference Model

The truly important dimension of Concert is not the underlying component model, which is a hybrid of COM and OMA concepts, but the actual components that have been conceived and specified. The first step towards conceiving Concert components was not the development of the component model, but rather the development of a high-level model for healthcare enterprise information systems.

This model, referred to as the MPG architecture reference model (ARM), identifies the key architectural ingredients for healthcare enterprise-capable applications and systems of applications.[5] These ingredients do not prescribe particular system features or technologies. Instead, they organize the architectural content of a software system into ten major groupings.

Each group describes a broad, but nevertheless partitionable, subset of an overall system architecture. The structure of a system is represented by seven facets, shown on the front of the cube shown in Fig. 7. The characteristics of the system that are transitive across all of the facets are illustrated as three horizontal layers that are stacked behind the facets.

The technique for graphically depicting these characteristics as slices was adapted from work on open distributed processing developed by HP's former Network Systems Architecture group.

The alignment of the boxes that represent the facets is important. The facets that represent system features that are most readily perceived by the end user are located towards the top of the illustration. Adjoining facets have significant interrelationships and influences on each other.

An application in the traditional, intuitive sense is also illustrated as a slice, but this slice only cuts through the three inner facets. In an actual system, the software that corresponds to these facets typically implements application-specific behaviors.

In contrast, the four outer facets represent the functional elements of an application system required to relate applications to each other in a coherent and consistent manner. These outer facets also represent the functional elements of an application system needed to relate the overall system to the healthcare enterprise.

The final element of the architecture reference model is the recognition that an application system is designed, developed, implemented, and supported using tools. The degree to which the design, development, implementation, and support activities are productive is a direct function of the degree to which complementary tools are employed.

Further, for each of these activities, the degree to which insightful knowledge of the healthcare enterprise is applied governs the degree to which the resulting application system meets the business needs of the system's supplier and satisfies the requirements of the clinical, operational, and business customers in the healthcare enterprise.

Outer Facets (in Fig. 7). The enterprise communication facet represents the capability for a system to interchange data with other systems in the enterprise based upon relevant healthcare standards. Important elements of this facet are the data formats and communication profiles that make up the interchange standards.

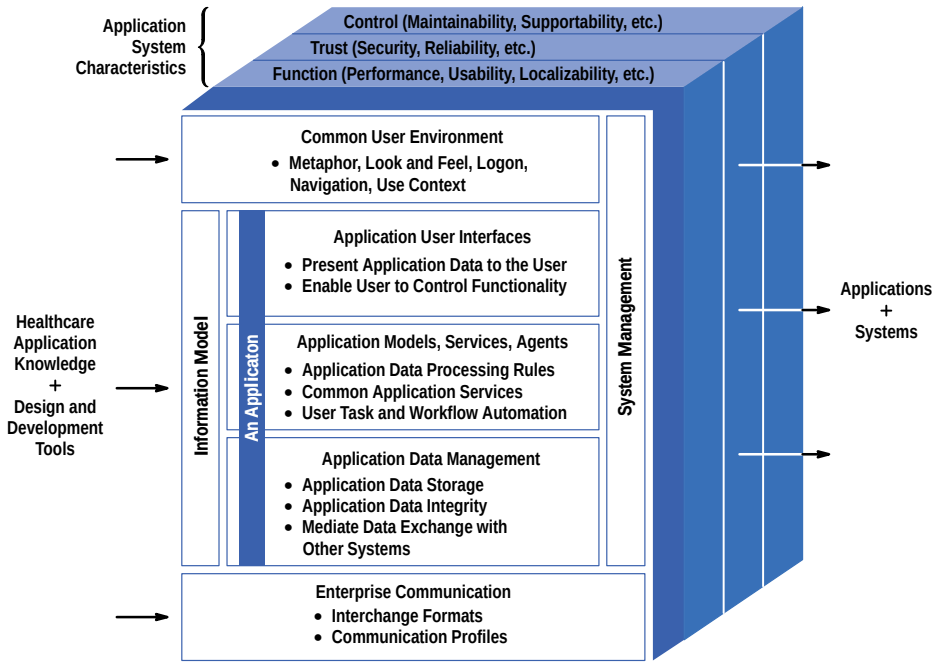


Fig. 7. *Concert reference model.*

The information model facet represents the "conceptual glue" that is essential for deploying an application system within an enterprise. The information model identifies and defines the entities and concepts that are important in the domain of the healthcare enterprise. The information model also helps ensure that these entities, concepts, terminology, and clinical processes have a consistent interpretation across all parts of an application system and the enterprise as well as between different but related applications.

The system management facet represents the "operational glue" that enables the uniform and consistent management of the system. This includes capabilities to:

- Turn the system on and off
- Assign passwords for users
- Install new software revisions
- Configure the functionality of the software
- Detect, enunciate, and log faults
- Intervene to correct faults
- Adjust performance parameters and resource utilization levels
- Provide end-user help-desk functionality.

The common user environment facet defines a unifying metaphor that governs user interactions with the underlying applications. For example, for an electronic medical record application system, the metaphor might represent patient data as sections in a virtual three-ring binder.

Under the umbrella of the metaphor, the common user environment also defines the healthcare-specific approach to application user interface look and feel (e.g., clinically appropriate colors, fonts, terminology for common menu selections, etc.), and it provides the highest-level controls, which enable the user to navigate to and between applications.

In addition to these specification-oriented elements, the common user environment includes capabilities that enable the user to log on once to an application system and to establish and manage a *use context* which is applicable to any of the underlying applications. The use context can include settings that identify the user and describe the user's clinical role, characterize the user's physical location, and indicate the user's natural language preference and default preferences for application appearance and control settings.

For example, a physician's use context might include the list of patients that the physician is responsible for. This list resides within the implementation of the common user environment but is accessible to any application in the system.

As the physician switches between applications, applications are provided with information about the patients on the list without requiring the physician to reestablish the list. The continuity provided by the use context enables applications to achieve a high degree of coordination and cooperation. These qualities benefit the physician by providing a simpler and more efficient user interface.

Inner Facets. The outer facets of the architecture reference model define an enterprise environment within which an application participates. An application is described in terms of three basic application facets. Representing an application in this manner makes it possible to factor the responsibilities of an overall application into more granular categories.

While reminiscent of the increasingly popular multitier client/server systems (in which application processing is distributed across a client and a hierarchy of servers), the three application facets are not about client/server computing. Instead, they are about decomposing application software into three distinct sets of responsibilities. This decomposition serves as the basis for scalable and extensible application implementations that can be deployed on a single computer or on a two-tier or N-tier client/server network.

The application user interface facet is responsible for presenting application data to the user and for providing mechanisms that enable the user to interact with and control the application. In this regard, the fundamental role of the user interface is to transform computer-based data into tangible entities that a user can perceive and manipulate. While this is clearly the overall responsibility of an application, the user interface portion of an application is focused on the ergonomic and human-factor aspects of this transformation.

The models, services, and agents facet is responsible for:

- Models
  - Validating user inputs before performing significant application data processing tasks and then performing these tasks
  - Mediating the transformation of application data into concepts and organizations that facilitate populating a user interface with application data
- Services. Providing application-level facilities that are common among but independent of any particular application
- Agents. Automating individual user tasks and multiuser workflows.

The models, services, and agents facet represents a substantial subset of an application's overall responsibilities. However, this facet is notably devoid of any responsibilities pertaining to the direct interaction with the user or with underlying data sources. This facet is neither responsible for the "face" put on the application data, nor is it responsible for the application data. Instead, this facet serves as the bridge in the transformation of data into entities that are tangible to the user.

The application data management facet is responsible for:

- Storing application data that is important to the user and the enterprise
- Mediating the exchange of application data with other systems in the enterprise
- Enforcing the information-model-based rules that ensure the semantic integrity of the application data over time.

This facet is easily confused with a database. However, a database is a particular technology, while application data management represents a set of related responsibilities. For example, application data could be stored in a file or come from a real-time feed (e.g., a patient-connected instrument) as well as from a database.

Further, one of the key responsibilities of this facet is to enforce fundamental data integrity rules (often referred to as business rules). This includes rules based upon the semantics of the data as identified in the information model (e.g., the valid set of operations that can be performed on a medication order) and enforcement of more basic consistency rules (e.g., ensuring that updates that affect multiple data items are reliably performed on all of the data items).

Application System Characteristics. The final part of the architecture reference model describes various characteristics of an enterprise application system that requires the participation of all of the architecture reference model facets.

Functionality is the characterization of an application system in terms of user-perceived qualities that are independent of any one application but must be adequately supported by all applications. These qualities include performance, usability, and localizability.

Trust is the characterization of an application system in terms of its responsibilities to provide users with a system that is secure, reliable, and available when needed.

Control is the characterization of an application system in terms of its capabilities to be administered, managed, supported, and serviced.

Status and Conclusions

Concert was first applied in a deployable prototype electronic medical record (EMR) system that was developed by HP Laboratories and the Mayo Clinic for use at Mayo's Rochester, Minnesota site. Protoypes based upon four types of Concert application components were developed for this project.

The architecture was subsequently applied by MPG to the development of the enterprise communication framework (ECF). An implementation of the enterprise communication framework has been provided to the core members of the Andover Working Group.

For both of these projects CORBA and OLE technologies were employed and development proceeded on HP-UX* and Windows®-NT platforms. Substantial practical experience was obtained, and several important architectural refinements were introduced. Most notably, however, the key concepts described in this paper were exercised and validated.

More recently, Concert has served as the basis for a variety of information system product development activities within MPG. The specifications, experiences, and some of the software developed for the EMR and the ECF are being applied. It typically takes an object-oriented software developer about two weeks to become familiar enough with the architecture to begin productive development of Concert-based software. Indications are that once this investment is made, the specifications provide a solid, self-consistent basis for system development.

The next challenge is to further optimize development productivity through the creation of Concert component development frameworks. These frameworks would provide code skeletons for partially implemented components. Armed with an appropriate set of productivity tools, application developers would be able to add the necessary features to the skeletons to create fully functional components. Tools would also help the developer "wire" the components together to form an application or a system of applications.

Acknowledgments

A substantial number of people were involved in the conceptualization and specification of Concert. It is through the efforts of all of these people, working together in concert, that we were able to develop a comprehensive architecture in a relatively short amount of time. The participants include: Don Goodnature, Mike Higgins, Jeff Perry, Jaap Suermondt, and Charles Young from HP Laboratories Analytical and Medical Lab, Rafi Ahmed, Philippe De Smedt, Louis Goldstein, Jon Gustafson, Pierre Huyn, Joe Martinka, James Navarro, Tracy Seinknecht, and Joe Sventek from HP Laboratories Software Technology Lab, Tom Bartz and Dean Thompson from the HP Network and Systems Management Division, Robin Fletcher, David Fusari, Jack Harrington, Nico Haenisch, and Geoff Pascoe from MPG R&D, Bob Anders, Steve Fiedler, Peter Kelley, Anthony Nowlan, and Mike Stern from MPG HealthCare Information Management Division R&D, Rick Martin and John Silva from the MPG Customer Services Division, and Alfred Anderson, Pat Cahill, Calvin Beebe, Woody Beeler, Tom Fisk, and Bruce Kaskubar from the Mayo Clinic. In addition, the author would like to thank Mark Halloran, HP MPG R&D manager, for his enthusiastic support and executive sponsorship of this work.

References

1. *Common Object Request Broker: Architecture and Specification, Revision 1.2*, Object Management Group, 1993.
2. K. Brockschmidt, *Inside OLE, Second Edition*, Microsoft Press, 1995.
3. *Version 2.2, Final Standard, Health Level Seven*, December 1, 1994.
4. *Concert Component Architecture: Component Concepts and Base Specification, Version 1.0*, Concert Document 95-11-1, Rev. A, November, 1995.
5. *MPG Architecture Reference Model, Version 1.0, Rev. A*, MPG Architecture Document 95-9-3, last revised September 21, 1995.

Components and Objects

Objects enable concepts to be developed using abstractions that represent real-world and computing concepts. The objects are interconnected to form programs that perform useful tasks. Components are also objects. However, components have the added dimension that they represent an economically and technologically practical way to organize and package an object-oriented system. A component system can be developed, marketed, licensed, maintained, and enhanced on a component basis.

In an informal sense, components are just "bigger" objects. With this bigness comes the need, and fortunately, the technical feasibility to support computing capabilities that are impractical for traditional "small" objects. For example, most component development technologies enable a component's external interfaces to be accessed through several different programming languages, and these accesses can often be performed across a network. It would be overwhelming to support these capabilities for every small object. However, supporting the capabilities becomes practical when objects are organized into bigger components.

Components can also be more cost-effective to develop and maintain than small objects. This is because components do more. Similarly, components can be more efficient to develop and maintain than traditional monolithic programs. This is because components don't try to do everything.

In a well-architected system, each component will provide enough functionality to warrant development as a standalone entity that can nevertheless be combined with other components to form fully functional applications. In a well-architected system, each component will be a candidate for being catalogued as a product and marketed as an essential building block for an overall system.

Examples of healthcare-related software components include a component that describes and correlates medical terms based upon standard schemes for encoding medical terminology, a component that checks whether medications being ordered for a patient might interact in an adverse manner, a component that enables viewing physiological waveforms in a manner that preserves aspect ratios and display size even when viewed on different display devices, and a component that enables applications to send and receive patient data based upon healthcare electronic data interchange standards.

The Andover Working Group

To establish a common implementation of data interchange standards in healthcare, in 1996 HP's Medical Products Group led the formation of the Andover Working Group (AWG) for open healthcare interoperability. This program is an industry-wide effort to accelerate plug-and-play interoperability between healthcare computing systems. The lack of compatibility among information systems is one of the most frequently cited information technology problems facing the healthcare industry today.

In 1996, the core membership of AWG included fifteen healthcare vendors and three healthcare providers. Each of these organizations contributed engineering resources to work on defining the enterprise communication framework (ECF) for HL7. In addition, in 1996, the AWG supporting membership included over one hundred additional vendors and providers. These organizations attended early review meetings of the ECF and provided the AWG with feedback and guidance about its processes, technology, and future directions.

The objective of the AWG is not to define new standards for interoperability. Instead, the AWG seeks to increase the commonality among the implementations of relevant healthcare computing standards. Standards such as HL7 walk a fine line between being prescriptive enough to be useful and being flexible enough to be widely accepted in the industry. However, inherent in this flexibility is the opportunity for implementers of the standard to make different implementation decisions. Different and often incompatible implementation decisions reduce the likelihood that systems will interoperate.

To overcome these problems, the AWG has developed an implementation of HL7. This implementation consists of detailed message profiles in which the specific HL7 messages that ECF-based applications can send and receive are described. The software that enables applications to use these messages easily is also provided in the implementation. The core of this implementation is a software component called an *enterprise communicator*.

The derivation of ECF message profiles involved the iterative refinement of an elaborate object-oriented information model by the AWG representatives. The enterprise communication framework software follows the component architecture described in this article. The result is a high degree of interoperability in the form of data interchange between healthcare systems without the usual system integration costs.

The first example of ECF-based interoperability was demonstrated in October 1996 when twelve applications developed by six different vendors, running on three different computing platforms, were modified to use the ECF software. The applications were able to participate in a detailed scenario that simulated a patient's admission to a hospital, ordering of a series of laboratory tests and reporting of the corresponding results, and an eventual discharge from the hospital. This level of interoperation was the first concrete proof of the effectiveness of the AWG as an organization and of the ECF as truly enabling software.

Multiple Interfaces in COM

In Microsoft's Common Object Model (COM), all components implement the interface IUnknown. This interface specifies only a few methods (method is the COM term for operation), including the method QueryInterface. A client of a component uses QueryInterface to interrogate the component to determine if it implements an interface of interest to the client.

QueryInterface accepts a single input parameter, which is used to indicate the interface of interest. If the component supports the indicated interface, a reference to the interface is returned. This reference can then be used by the client to access the component via the interface. If the interface is not supported, then the special value NULL is returned.

Conceptually, within a running instance of each COM component, a table of the interfaces implemented by the component is maintained. When the component instance is initialized, its table is populated with the names of all the interfaces that the component implements. Associated with each name is a pointer to the code that implements the particular interface.

When QueryInterface is called by a client of the component, the component consults its table of implemented interfaces. If the interface being queried about is implemented, then the reference that is returned contains data that essentially points to the implementation of the interface. This data enables the component's client to access the interface and therefore its underlying implementation.

In COM, the interfaces that are supported by a component cannot change once the component has been initialized.

Object-Oriented Customer Education

As customers require more trusted advice to solve their business problems, the choice of education solutions has become a strategic issue that often precedes and directs the choice of technologies.

by Wulf Rehder

Whether you buy a laptop computer or a lawn mower, you expect to learn how best to use it. For some products it is enough to skim the user's manual. For others you need to attend a class. In the past, product training was considered an attribute of product support. It came bundled with the product and was an expected feature like a power cord or the certificate of warranty. This situation has changed. When you buy a toy, batteries are no longer included. Similarly, education is no longer automatically included and free with the large, enterprise-wide solution purchases customers make today.

In such enterprise projects even laptop computers must be designed to work together with many other products that are often distributed in networks over large entities or even different countries. In these environments, training on how to use a single, standalone product is no longer sufficient. Customers now expect more comprehensive services, ranging from training in soft skills such as design methodologies and project management to proficiency in hands-on implementation and online troubleshooting.

The complexity of solutions, the size of customer projects, and the fact that computer systems are increasingly mission-critical for most businesses have led to the unbundling of product training and to the creation of entirely new product lines for professional consulting and education. Training has changed from being a product accessory to being a product itself. Customer education has grown from under the umbrella of product support to becoming a large and profitable industry by itself. In this paper, I will focus on the way HP's customer education, as part of the HP Professional Services Organization, is meeting the new challenges of developing and delivering to customers a cohesive suite of object-oriented education products.

Managing the Transition

It is a truism that every act of learning is a passage from knowing less to knowing more. However, customer education is more ambitious. This ambition shows itself in three ways. First, it is not enough to fashion data and information into a consistent, meaningful body of knowledge. While in a training class, customers must be led from "knowing what" to "knowing how" and being able to apply the new learning to their real-life problems. More knowledge must be transformed into more skills. However, there is a second, complementary aspect to learning: learning means not only acquiring more skills, but also acquiring different skills for new tasks in a changing environment.

The successful management of adapting to this change and the transition to higher levels of knowledge are the objective of customer education for all job roles as shown in Fig. 1.

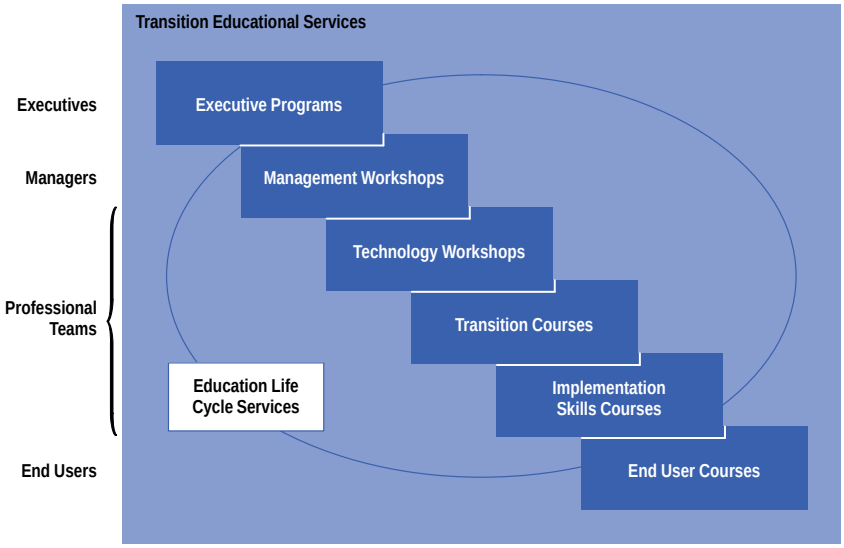


Fig. 1. *Enterprise-wide approach to managing transitions.*

Executives must be made aware of the risks and benefits the change to new technologies and processes entails for the entire company. With this awareness they will acquire the confidence, authority, and credibility to lead their business into previously uncharted terrain. Managers obtain the understanding and expertise to make the right technical decisions for their teams to be successful. Designers and developers master new professional crafts that help them apply the lessons learned for the creation of new products. Finally, end users realize the concrete benefits of the new technologies and processes.

The third defining component for contemporary customer education is its comprehensiveness. Fig. 2 specifies the three branches of a company's assets that need to transition together in a balanced way: its people, processes, and technology. All three are centered on serving common business objectives.

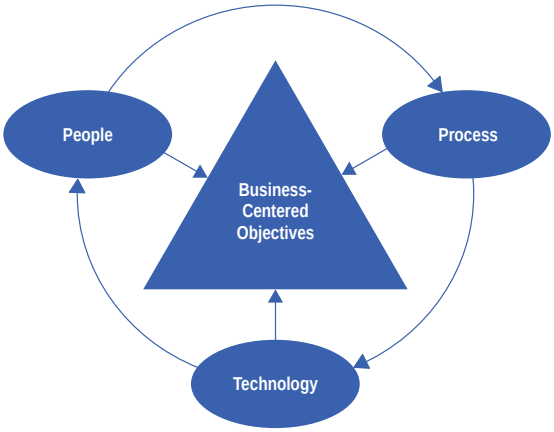


Fig. 2. *HP's people, process, and technology approach.*

This brief sketch has far-reaching practical consequences for the positioning, development, and delivery of customer education solutions. They do not merely add value to a product, but create their own suite of added values. Fig. 3 shows this value chain from the point where the actual interaction with the customer occurs (for instance, course content research and development phases are left out). As appropriate, some phases will be traversed repeatedly, depending on the results obtained so far and on the quality measures (e.g., completeness, level of detail) applied in the particular phase. Therefore, the links of the chain need to be interpreted as cycles. Under the name of education life cycle services, this simplified framework articulates the fact that customer education teaches how to manage change and how to evolve new skills.

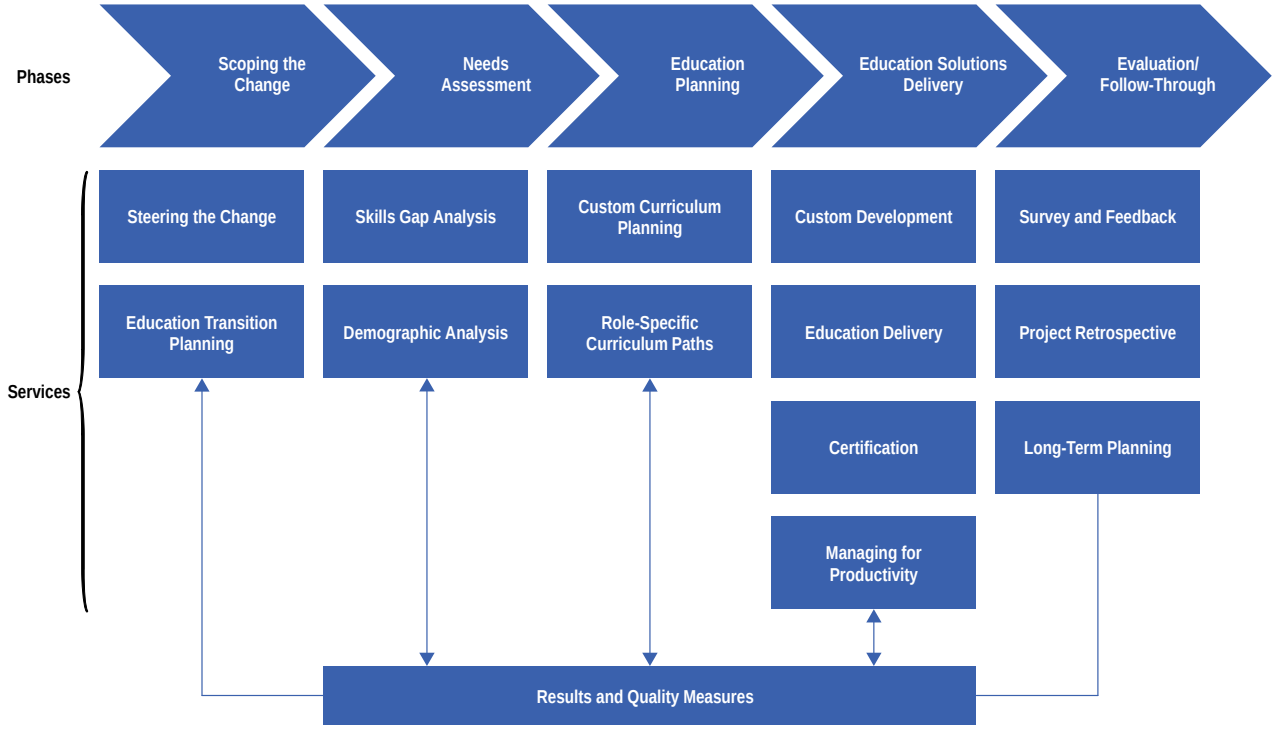


Fig. 3. *Education life cycle services.*

Customer education has become the industry of facilitating the transition from Tennyson's "blind and naked ignorance" to St. Thomas Aquinas' skill of man "to know what he ought to do."

Know Thyself

Before answers about the right path to object technology can be given, the right questions about the starting point, the path itself, and the goals have to be asked. To evaluate the starting point, HP's customer education services have developed a workshop called skills gap analysis.[1] Fig. 4 shows a step-by-step outline of this course. During the analysis, which is done jointly with the customer, the following documents are created to serve as the basis for the next transition steps:

- A written statement about business needs
- An inventory of current skills
- A list of additional skills to close the gap between current skills and identified needs
- Validation of findings and determination of action items.

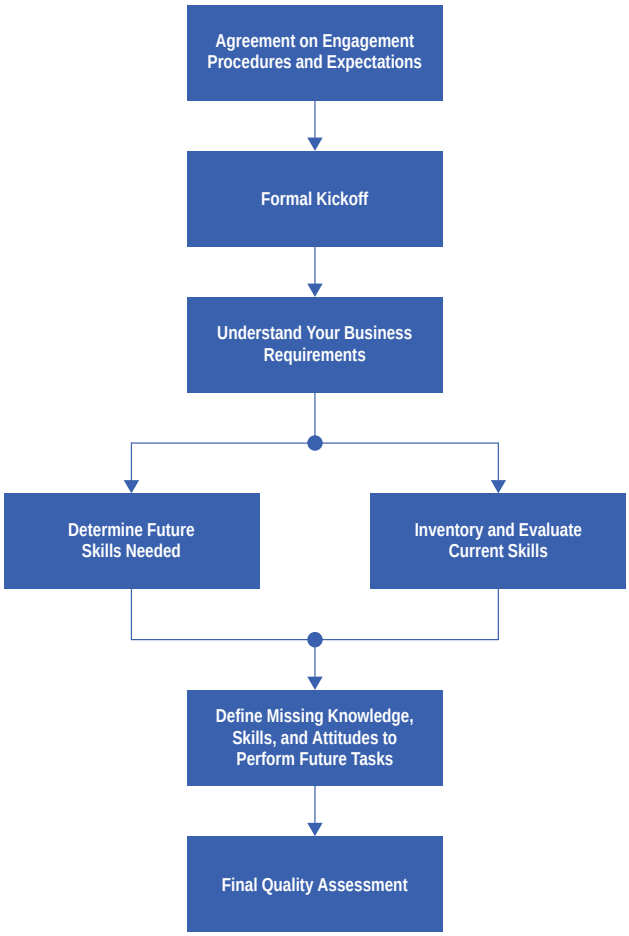


Fig. 4. *Skills gap analysis methodology.*

A skills gap analysis addresses a company's overall training needs and by itself does not result in a detailed training plan. To be more relevant to the discussion, we will focus on objects. The customized, object-specific version of a skills gap analysis is the object-oriented transition assessment workshop.[2] Similar to the skills gap analysis, the customer and at least two of HP's educational and technical consultants work through three sets of questions, assessing:

- The goals of a transition to objects
- The present skill level and object exposure
- The customer's current software development process.

A selection of some of these questions is enumerated on this page. In the transition assessment workshop, the skills gap analysis culminates in the preparation of a list of the ten biggest obstacles for a successful move to objects, jointly agreed upon by the customer and HP's consultants. These obstacles are different from company to company, but they typically fall into the categories of management commitment, organized barriers, fear of change, scarcity of resources, and loyalty to

legacy systems. Rarely are the inhibitors purely technical; the switch to new object-oriented tools and products is less problematic than overcoming the "soft" issues just mentioned.

This list of obstacles is the document upon which HP's team bases its recommendations for a concrete object adoption agenda, including job-specific curriculum paths. Such a detailed plan is the final outcome of the object-oriented transition assessment workshop.

After the workshop, with the enthusiasm usually quite high, many software development teams want to start their first object-oriented development project without delay. At this juncture, the HP consultant assumes the role of a mentor and monitors the speed, direction, and results of the transition that is now under way. See Subarticle 12b "***Starting an Object-Oriented Project,***" which summarizes a few caveats collected from many mentoring sessions.

Four Pillars of Soft Skills

A glance at the life cycle in Fig. 3 shows that the next phase is education planning. Based on the skills needs, curriculum paths are created that match specific jobs and roles designed to fill the needs. If, for example, system modeling skills are missing, the joint HP-customer team may define the new role of a system architect and recommend a series of courses to retrain designers to become architects. Once the roles are identified, the solutions will be designed and implemented. Experience has shown that this is not yet the place to select technologies (such as tools and implementation languages) or products (middleware, databases). Instead, the success of a transition to object technology appears, as our case studies with customers have shown, to be determined by the mastery of four soft skills: software architecture,[3] analysis and design methodology,[4] project management,[5] and systematic reuse.[6]

Software Architecture. Of the four skills mentioned above, architecting a software system is perhaps the most difficult, yet the most important and least well-understood skill. For the sake of brevity, three of the most important aspects of this difficulty are discussed here. First, there are at least four different views of a system architecture that emphasize different but overlapping concerns of high-level system design (see Fig. 5).

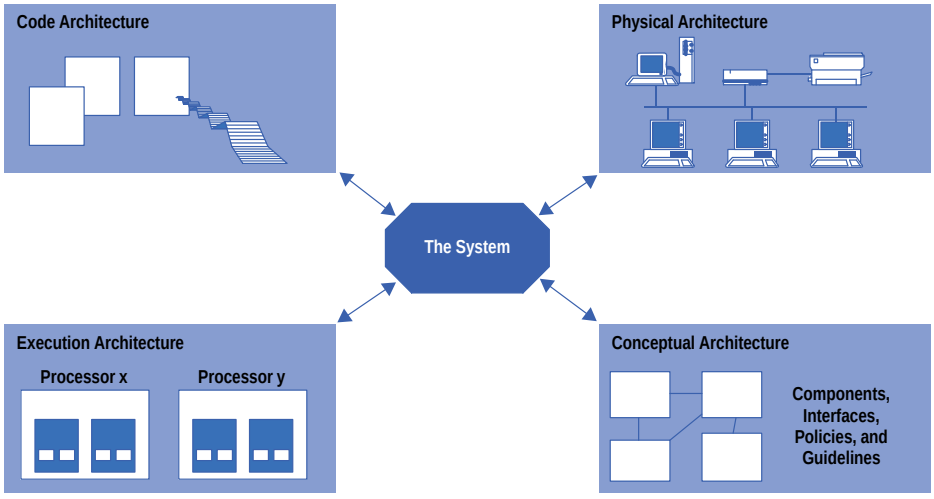


Fig. 5. *Four architectural views.*

Second, there is the choice of a viable reference architecture for an enterprise, which is a blueprint realization of an architecture that best fits a given business purpose.

One of the most successful frameworks for such a reference architecture is the so-called three-tier architecture (see Fig. 6). Once the tiers with their subsystems and interface specifications have been defined, it is possible to map products into the framework. For example:

- VisualBasic, Powerbuilder, or VisualWorks for the presentation layer
- C++ to build application programs whose components may be running on distributed servers
- A database or data warehouse like HP's Depot/J for the data management system
- Softbench for the development environment
- The Object Request Broker (ORB) software for the infrastructure logic that manages the communication among the distributed software components.

However, it is advisable to postpone these technology choices until after a thorough analysis and design methodology has been applied based on the particular customer requirements and the anticipated use cases of the planned system. See ***Article 10*** for a definition of use cases.

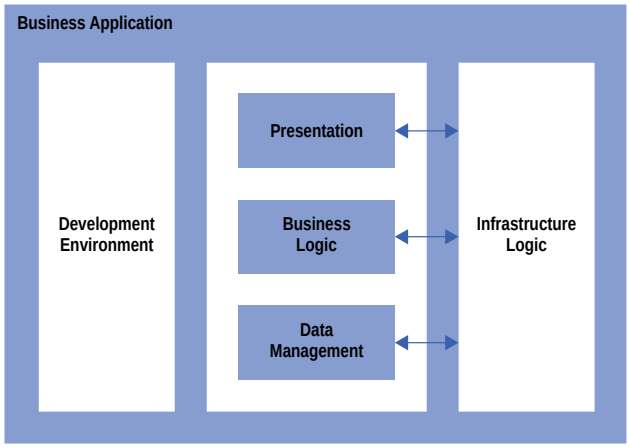


Fig. 6. *The three-tier architecture framework.*

The third difficulty is the lack of a generally accepted notation that is simple to apply and learn and yet rich enough to express the complex semantics of objects and their interactions in the different layers of a software system. HP and its partners are working together in committees chartered by the Object Management group (OMG) to formulate such a unified architectural language.

Analysis and Design. Better known and more mature than architectural models are the software analysis and design methods. They are often called methodologies, to distinguish them from the methods (i.e., the procedures or functions) owned by objects. A methodology defines a process that allows the division of work into distinct phases, each of which has well-defined exit criteria (e.g., finishing a graphic object model, drawing all dependency diagrams, and agreeing on design documents). The goal is to translate informal customer requirements into a more formal structure that then guides the implementation. Besides structured analysis and structured design other methodologies include the waterfall life cycle model of software development and HP Fusion.[7]

Project Management. Once the software architecture has been chosen (e.g., a three-tier reference model) and a methodology team has gone through the phases of system requirements, analysis, and design, a project team needs to be chosen to implement the design that realizes the architecture and solves the business problem. At this point of the transition, thinking about the peculiarities of object-oriented project management becomes important. Because of the inherent modularity of object-oriented design and the ensuing independence and autonomy of subteams, team building and communication may become an issue. New roles and responsibilities, such as framework architect, pattern designer, and class librarian need to be integrated. Since object-oriented design favors the implementer who postpones coding and (re)uses components as much as possible, performance evaluation and reward systems need to be reconsidered. This is opposed to the model of rewarding the implementer who "hacks" out the most code.

Reuse. The fourth of the recommended soft skills essential for a successful move to object-oriented software development is the incorporation and long-term management of systematic reuse. This course combines a discussion of

reuse technology (frameworks, patterns, software kits, components, and standards), and tools and processes with organizational and management issues. These latter nontechnical concerns often have the biggest impact on change management and the success of the transition to objects.[8] In the spirit of hands-on skill development, the second part of this course simulates the steps of systematically building reuse into a software organization. Fig. 7 shows the incremental steps from no reuse to systematic reuse through stages that mirror the phases of the Capability Maturity Model (CMM), which is widely used in the assessment of software skills.[9,10,11]

Projects versus High Volume

From the discussion above it should be obvious that the approach to customer education requirements for the transition to objects is not simply a matter of technology and product training. Just as an information technology department is much more than a random collection of computers and wires, so is today's customer education more than a collection of training courses. It has become an industry with finely tuned product lines that match the requirements of job groups by providing comprehensive training paths, from introductory courses to in-depth specialized skills.

However, in addition to these task-oriented, individualized curriculum paths, increasing emphasis is being put on integrated curricula for project teams, departments, and entire organizations. This latter trend has led to two distinct, but collaborating branches within the customer education business. One branch addresses the difficult, unique custom software project or the transition of, say, a COBOL programming team to Smalltalk proficiency. Efforts like these are resource-intensive, of high complexity, and more often than not also low-volume affairs. (They are the human learning system equivalents of highly sophisticated hardware and software systems, which usually need to be custom-made.)

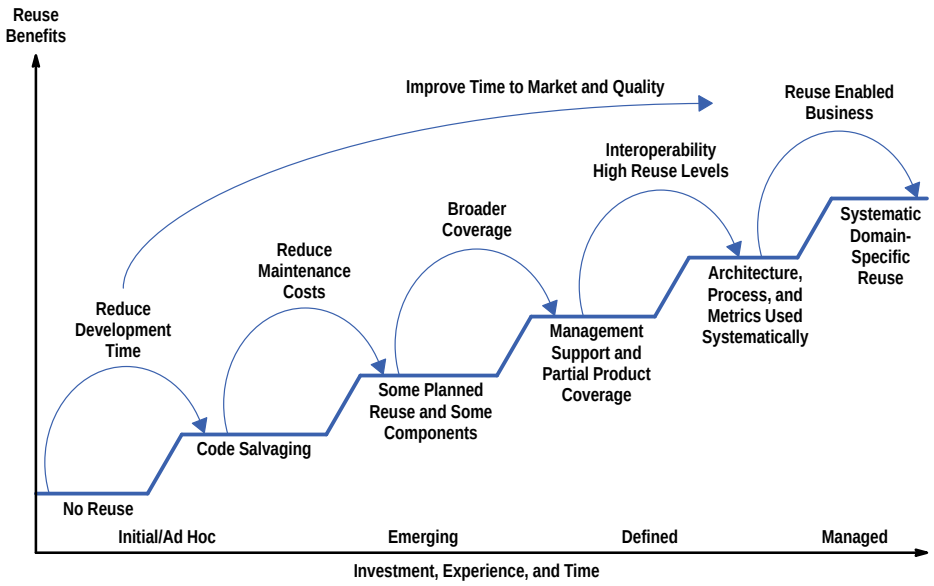


Fig. 7. *Incremental approach to reuse and the resulting benefits.*

For custom-made education solutions to be affordable, such highly complex offerings need to be created in a repeatable and modular manner. Examples of custom courses are the total immersion programs. In these programs, which are variously known in the industry as residency programs or boot camps, entire teams are led through a four-to six-week customized curriculum to object-oriented literacy.

The other, complementary branch of customer education addresses the high-volume, lower-complexity demands. These are requests for standard programming language courses, fundamentals of operating systems, system administration, networking, and relational databases—all of which figure prominently in most two-or-three-tier business application developments.

These conditions of serving widely diverging interests are posing challenges for the development, sales, and delivery of education solutions in general and object-oriented education in particular. The challenges are similar to the ones known in traditional product development:

- Primary and secondary research explore the market conditions
- Investigations define product possibilities
- Curriculum creation involves outsourcing, partnerships, and collaborations with product divisions and the field
- After going through the typical lab cycles, prereleased material is validated in alpha and beta tests.

In parallel, marketing collateral is being prepared, data sheets, sales briefs, and advertising copy are written, catalogs appear worldwide, and indirect and direct sales are made. To be successful, a well-managed and diverse team of course designers, business developers, solution architects, education advisors, technology specialists, consultants, and instructors needs to be trained and deployed worldwide. Issues of localization, government regulations, copyright protection, postrelease support, updates, and pricing (for instance, discounts, volume buying, specials) are again not different from the rollout of major hardware and software products.

In light of these considerable complexities, training vendors may be tempted to define their solutions by offering a variety of topics for which they have in-house technology expertise and then to reshape the customer needs along the lines of these topics. The true challenge consists, however, in basing education solutions on the transition assessment workshops and education plans that have been crafted and agreed upon jointly by the customer and education consultants. Only such solutions have the strategic impact of preceding and guiding the choice of implementation technologies.

Point Solutions and Product Training

Supporting the larger picture of education solutions outlined above are several training offerings that are more specialized, narrower in scope, or tool and technology related. Here, training usually tracks the release, purchase, and installation of products. As a consequence, training courses have to be updated in a rhythm following the product updates. This especially includes languages converging towards standards, such as ANSI C++, different implementations of new languages, such as Java, and products that bridge evolving de facto standards, such as those for distributed computing. Examples of the latter are the Object Request Broker (ORB) implementations which adhere to the Common Object Request Broker Architecture (CORBA) standards and serve as interoperability middleware between CORBA objects and the emerging Microsoft® OLE automation product suite. Such software has to be supported by several operating systems and communication protocols. In the case of HP's ORB Plus 2.0 these are the HP-UX*, SunSoft Solaris, and Microsoft NT platforms and the IIOP (Internet

Inter-ORB Protocol, platform independent) and DCE CIOP (Common Inter-ORB Protocol, HP-UX only) standards. Using the IIOP protocol, ORB Plus 2.0 will interoperate, for instance, with Distributed Smalltalk software from ParcPlace-Digitalk.

From these typical examples it becomes obvious that narrow, specialized point solutions and product training can be as labor-intensive as the solutions centered around the care for people and processes. Since the competitive pressure for training on shrink-wrapped products is fierce (you can learn C++ in community colleges almost free), larger education providers have surrounded themselves with satellites of smaller, agile partners, who can, in the analogy used before, be compared to suppliers of hardware and software parts.

Challenges and New Directions

One of the most exciting events in the emergence of object technology is the recent promise of its convergence with internet technology. To begin with, Java is a C++-like object-oriented language that allows the objects (for instance, the ones created in its applets) to be shared over the net in a platform independent way. Java has spawned several new customer education offerings, including ones on web security and on how to use the web for commercial transactions.

Furthermore, with the web becoming more familiar as a medium for information exchange, it is also fast becoming a candidate for alternative training delivery, complementing computer-based training (CBT), CD ROMs, and the traditional lecture and lab format. Such a departure from copyrighted class material to an essentially open, public forum creates new challenges, but these challenges are no more severe than the ones faced by software distribution and publishing on the net. This is especially true in the high-volume, point-solution, and product-training market where the material is rapidly becoming part of a commodity business with small differentiating value and practically no proprietary lock on content. Instead, as Tim O'Reilly[12] suggests (and practices for his own on-line business of computer books), more important than copyright is the development of a brand identity that represents a consistent, trusted selection of high quality. This is where high-volume customer education may be headed in the future.

Acknowledgments

My view of customer education as an autonomous business has evolved in discussions with Patricia Gill-Thielen, Brian McDowell, Tom Ormseth, Morris Wallack, and Ann Wittbrodt. While they are not to blame for my opinions, I hope that they'll accept my thanks for mentoring me when I joined their team.

References

1. *Skills Gap Analysis Workshop*, Product Number H6230A.
2. *Object-Oriented Transition Assessment Workshop*, Product Number H6290X+002.
3. *Software Architecture Workshop*, Product Number H6290X+009.
4. *Analysis and Design Methodology Workshop*, Product Number H5851S.
5. *Project Management Workshop*, Product Number H6516S.
6. *Systematic Reuse Workshop*, Product Number H6514S.
7. T. Cotton, "Evolutionary Fusion: A Customer-Oriented Incremental Life Cycle for Fusion," *Hewlett-Packard Journal*, Vol. 47, no. 4, August 1996, pp. 25-38.
8. M. Griss, I. Jacobson, and P. Jonsson, *Software Reuse: Architecture, Process, and Organization for Business Success*, Addison-Wesley, January 1997.
9. W.S. Humphrey, *Managing the Software Process*, Addison-Wesley, 1989.
10. M.C. Paulk, B. Curtis, M.B. Chrissis, and C.V. Weber, *Capability Maturity Model for Software, Version 1.1*, Software Engineering Institute, CMU/SEI-93-TR-24, February 1993.
11. D. Lowe and G. Cox, "Implementing the Capability Maturity Model for Software Development," *Hewlett-Packard Journal*, Vol. 47, no. 4, August 1996, pp. 6-14.
12. T. O'Reilley, "Publishing Models for Internet Commerce," *Communications of the ACM*, Vol. 39, no. 6, June 1996, pp. 79-86.

Questions about Using Objects[1]

What are the goals for your transition to objects?

- Following the general evolution of the software industry
- Benefitting from external libraries of reusable components
- Making the resulting software easier to modify
- Improving time-to-market for new products
- Decreasing software development costs.

What is your company's current exposure to object technology?

- Novice level
- Some people have general knowledge
- Some people have used object-like technology, for example by programming in Ada
- The company has successfully completed a small object-oriented project
- The company has successfully completed a substantial object-oriented project (more than 300 classes) using a hybrid language like C++, CLOS, or a purely object-oriented language like Eiffel or Smalltalk.

What is your company's current software development process?

- It develops most of its software in-house
- It outsources its software development
- It has a recommended software development process (such as the waterfall model, the spiral model, prototype based, etc.).

Reference

1. B. Meyer, *Object Success—A Manager's Guide to Object Orientation, Its Impact on the Corporation and its Use for Reengineering the Software Process*, Prentice Hall, 1995.

Starting an Object-Oriented Project

Reading books and magazines will not always guide you through your first object-oriented project. A recent issue of a trade magazine had 24 advertisements for CASE tools, 26 advertisements for products, and 23 advertisements for object-oriented consulting services. Add to this the lack of standards, and the process of adopting object technology looks truly daunting. However, there are great rewards as long as you are ready to follow a well-defined process and make a longer-term commitment.

You and your managers may think that the success of the move to objects depends on the size of the projects undertaken, the number of people involved, and the tools and techniques used. However, in reality these factors have little impact on the transition's success. As a rule of thumb, the transition of a single software development team to object technology takes at least a year.

You will most likely find your organization in one of two stages of adoption: the investigation phase or an early adoption phase. If in the investigation phase, your company is ready to make some investments, but is not sure yet if object technology is the right choice. The objective of your project, which should be important but not mission-critical, is to provide a feasibility proof and show the measurable benefits. If in the early adoption phase, higher management has probably made a strategic decision in favor of object technology, and it is expected that your project will make a significant contribution to the business and provide a competitive advantage.

There are several questionnaires that help in assessing where your company is in the transition process. Here are some questions that I have found useful:

- Can you formulate a business case for your project that will yield a measurable, positive net present value (NPV) for your organization?
- What are the investments necessary to fill the skill gaps found in your skills gap analysis?
- What are the specific success factors and possible risks for this project?
- What object architecture will you pick and why?
- What outcome of your project shows the feasibility of object technology for your organization or your whole company?
- What will you do with the existing legacy systems?
- Does it make sense to connect your project to the potential of the intranet and internet?

The object-oriented transition assessment workshop includes these and other questions. They have proven helpful for customers and, despite their simplicity, are surprisingly hard to answer.

Ramesh Balasubramanian
HP Professional Services Organization
Objects Consultant