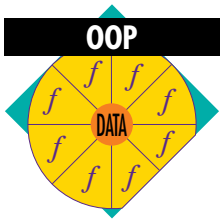
Building an OO Development Environment on AIX



By Michael J. Branson and Eric N. Herness

This article describes an Object-Oriented (OO) development tools environment for a large OO project in IBM's programming lab in Rochester, Minnesota. It outlines the findings and outcomes, the types of tools assembled, the source of those tools, customizations made, and feedback received from users. It describes what did and did not work, and the requirements that led to the tools environment described.

This article provides a behind-the-scenes view of the tools environment necessary to support a large OO development project. A complete Object-Oriented Programming (OOP) environment includes a methodology supported by processes, tools, and an OOP language. Constructing a tools environment for a large-scale OOP project requires a variety of resources and a unique set of skills. Effectively deploying object technology requires timely and effective delivery of scalable tools environments.

The OO Development Strategy

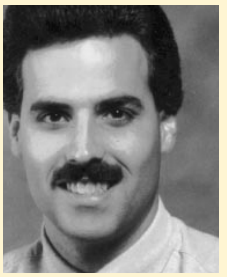
The OO development process must be integrated with a toolset that enables developers to be productive. The proper use of OO development tools is not as obvious as traditional tools because many are new, and there has not been a lot of experience with their use. Like traditional programming, developers depend on a process to guide them in using the tools.

Tools used in OO development must relate to the development methods and the development process used within a project. Tools by themselves do not replace a methodology or a

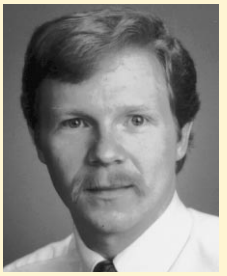
Project Description

The project that drove the development of this toolset was a large operating systems development endeavor. Because of the nature of this project, not all of our findings may be relevant to other projects. Here are the key characteristics that made our project unique.

- C++ was the implementation language for this project. Some tools are optimized to C++ and some problems may be unique to this development environment.
- It is a large-scale, object-oriented development project—150 direct programmers and dozens of affected programmers and supporting staff. The new C++ code written has more than 7,000 classes and more than one million lines of source code. About 12,000 compiles occur on the network each day.
- In addition, other smaller projects use the development environment, which adds hundreds of other users—making scalability a top priority.
- A legacy system is involved, so new C++ code had to interact with the old procedural code.
- There is no Graphical User Interface (GUI) component.



Michael J. Branson



Eric N. Herness

process. They are used to practice a methodology and to complete the deliverables of the development process.

Environment

The *development environment* includes the base hardware and software configurations used for development. It also encompasses the target execution environments of the software being developed. It is common to use a variety of hardware platforms with various operating systems to develop software products.

The environment and the tools are interdependent. For example, if the development workstations are a given, the tools must run on them. If, on the other hand, you are purchasing new equipment and base platform software, you should be influenced by where the OO development tools run best. The target execution environment also affects the requirements for compilers and build tools.

The development environment includes the base hardware and software configurations used for development.

Other Factors in the Transition to OO

Other elements essential to successful object-oriented development include education and training, the development process, and the development culture. Fundamental changes must occur in how software development is done when transitioning to object-orientation.

The development culture reflects changes experienced by the developers in the transition to OOP. Some workstation-based tools that accompany the OO development process are the most leading-edge tools available today—often motivating developers. On the other hand, tools used for OO development are so different from traditional tools that they require new thinking by developers. Discovering that you not only need new tools, but also a new toolbox, can be very frightening.

Integrating Tools with OO Development

There are many different OO methods. [1,2,3,4] Although each method takes a different approach to development, there are similarities between them. To define a toolset that supports the methodology choices of a diverse set of developers, a method-independent development process must be considered. A *method framework* lists the steps performed during development. [5,6] The following steps characterize the assumptions made by our team about the OO development process.

- **Derive candidate classes and objects**. This step encompasses domain analysis to produce candidate classes and objects. Refining and building a design model is more productive when candidate classes and objects are plentiful. Getting candidate classes and objects means understanding the problem domain and the requirements. It means carefully communicating with users of the system being built and with others having domain experience.
- **Build and refine a design model of those classes and objects**. The specific notation selected affects the exact name for this model; we call it the design model.
- **Build additional validation models to verify the evolving design**. Proceed with this step after a reasonable model of classes and objects has been formulated. These methodology-dependent models deal with more dynamic and functional elements of the design than the static model of the previous step. These models view the design model differently and help to validate and refine it.
- **Leverage existing classes, patterns, and frameworks**. Available class libraries are leveraged to describe the design and enable implementation. As details are added to the design model and other models are generated, the influx of components from the reusable class libraries continues.
- **Develop the class interfaces**. Creating the code-level interfaces to the classes can be an

[1]Booch, Grady. *Object-Oriented Analysis and Design with Applications*. Redwood City, California: Benjamin/Cummings Publishing Company, Inc. 1994.

[2]Rumbaugh, James; Blaha, Michael; Premerlani, William; Eddy, Frederick; and Lorensen, William. *Object-Oriented Modeling and Design*. Englewood Cliffs, New Jersey: Prentice-Hall. 1991.

[3]Jacobson, Ivar; Christerson, Magnus; Jonsson, Patrik; Overgaard, Gunnar. *Object-Oriented Software Engineering—A Use Case Driven Approach*. Wokingham, England: Addison-Wesley Publishing Company. 1992.

[4]Wirfs-Brock, Rebecca; Wilkerson, Brian; and Wiener, Lauren. *Designing Object-Oriented Software*. Englewood Cliffs, New Jersey: Prentice Hall. 1990.

[5]Branson, Michael and Herness, Eric. "The Object-Oriented Development Process." *Object Magazine*, 3(4), pp. 66-70. Nov-Dec 1993.

[6]Herness, Eric. "Object-Oriented Analysis and Design." *AIXpert*, pp. 40-44. May 1992.

additional verification step that may stimulate changes to the design done in previous steps.

- **Implement the classes**. Finally, probe the design decisions that have been encapsulated and left in the implementation. As implementations are written, additional small abstractions will be extracted. As additional uses for the component library are found, they should be reflected back into the design model.
- **Iterate through the steps until the design is robust**. Concurrency and iteration are inherent in the OO process. Each step can begin after a reasonable pass at the previous step has been made. A truly incremental and iterative process will take multiple passes through each step, producing workable code at the end of each pass. Second and succeeding passes will require less time in the earlier steps and more time in the latter, until a stable working system is attained.

The nature of the activities in an OO method framework require support from nontraditional tools. Figure 1 illustrates these activities and lists the kinds of tools that would be used to perform them. All of these tools must be available and integrated to support a large-scale object-oriented development project.

OO Tools

Many types of tools are needed to support OO development. Often, stand-alone products only work in isolation; many are not built to effectively support anything but small undertakings.

The following sections describe our tool choices and the customizations necessary to make the toolset both integrated and supportive of large-scale development.

CASE Tools

CASE tools are essential to deploy OO technology to a large project successfully. CASE tools that support analysis and design are helpful in the "Build Design Model" activity in Figure 1. They also support constructing additional models that validate the static model as shown in the "Build Validation Models" in Figure 1. Some CASE tools are optimized to one notation and method, while other tools can be used for several method and notation pairs.

The chosen CASE tool must not only support generating the appropriate models, but must also be integrated into the tools environment. Ideally, the CASE tool would accomplish the following objectives:

- Enable the developer to browse or edit the code that a class diagram models directly from the class diagram
- Load any code generated by the tool directly into the Configuration Management (CM) system
- Generate diagrams from the code in the CM system
- Enable multiple users to work on a set of design models simultaneously
- Control the design data by using the same CM and version control mechanism as the code

Besides integrating with the rest of the environment, the CASE tool should have a minimum set of additional characteristics:

- Scalable to support large projects in the thousands-of-classes range
- Have an underlying dictionary to allow the same entity to appear on multiple diagrams, but be defined only once
- Check for semantic inconsistencies between models and diagrams

CASE tools are essential to deploy OO technology to a large project successfully.

CASE tools must work with the documentation tools to easily generate the deliverables of development process activities. This may require some customization of the base CASE tool.

Variations on the Booch method were chosen for most of our development work. While methodology choice and application did vary, the Booch notation was agreed upon as a common documentation mechanism. When the project began in early 1992, no commercially available CASE tools supported Booch that met our requirements.

We used Cadre's® Teamwork® CASE tool that supported traditional development as an interim measure. Conventions were established and customizations were made to allow developers to build class diagrams and object diagrams using the traditional models available in Teamwork. When Rational ROSE™ became robust enough, developers moved to ROSE, which better supported object-orientation[7].

[7]Herness, Eric and Mitchell, Todd. "Using Cadre Teamwork for Booch Notation-based Design." 1993 Teamworkers Conference Proceedings (January 1993).

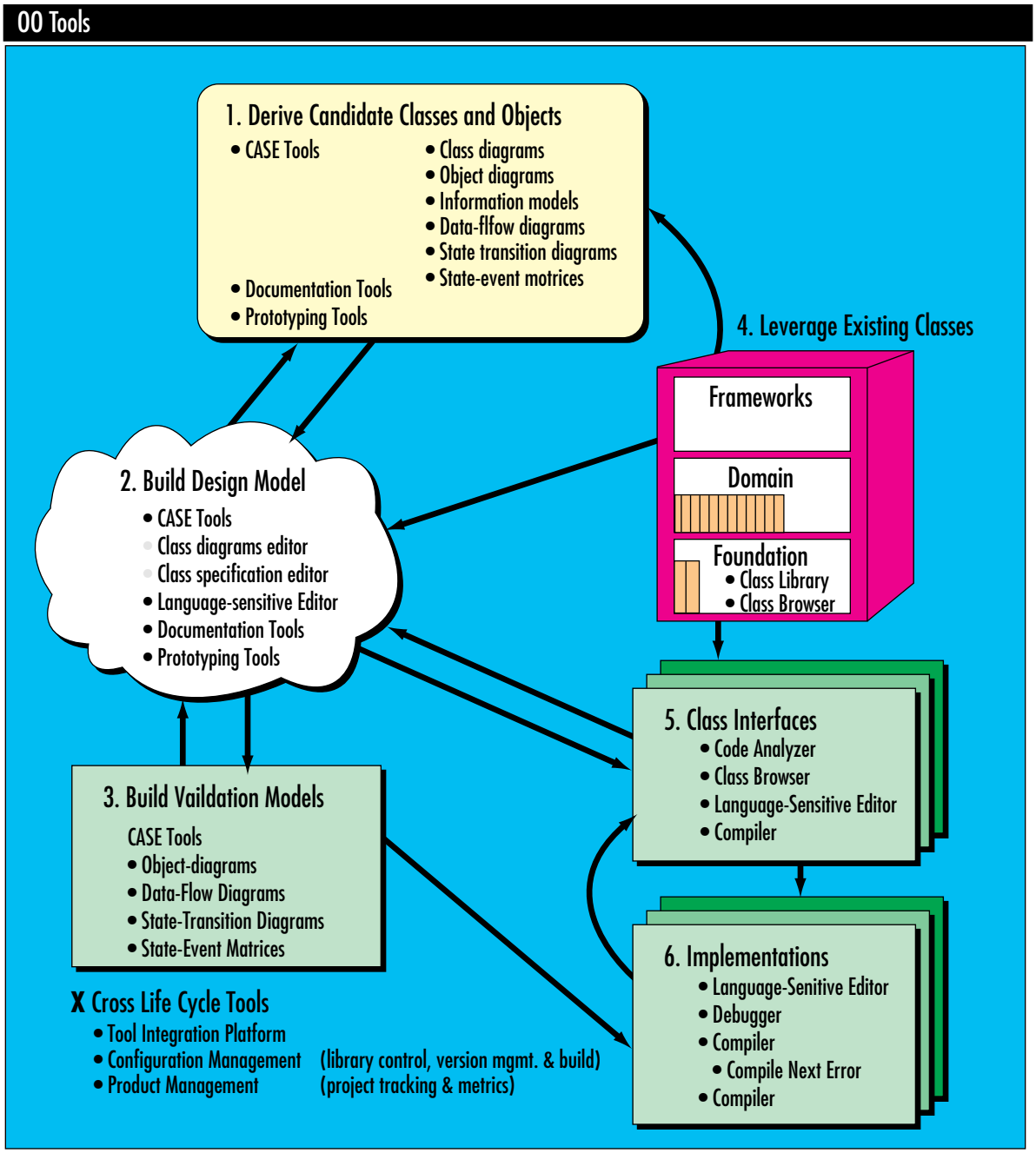


Figure 1. Object-oriented development—tool view

Our Customizations of CASE Tools

We made the following specific customizations to Teamwork:

- **Class specifications**. To tailor the Entity-Relationship Diagram (ERD) editor to our needs when working with class diagrams, we redefined the data dictionary to be used for Booch class specifications.
- **Information model**. We chose the ERD editor, also known as the Information Model (IM) editor to support class diagrams. Our users required an option for building Booch-like class diagrams. To make the data-dictionary editor more user-friendly, we added a pull-down menu that uses the SDE Workbench/6000-based editor to edit or add text to the data-dictionary entries.
- **Documentation**. To obtain documentation for a design, we added a pull-down menu to

retrieve all the notes and data-dictionary entries for a diagram, then send this information and the diagram to files that can be embedded into an inspection package. This enables the developer to obtain all documentation for a design with one menu command.

To translate class diagram information to C++ code files and to develop diagrams and code together in an iterative process, we added to both CASE tools the feature of bringing up an edit session on a file that corresponds to a class on the diagram. This is done by selecting a class on a class diagram, then selecting either Edit Source or Edit Header from a pull-down menu.

When this feature is integrated with SDE WorkBench/6000 and CMVC/6000, a search file (a special local customization made to the CMVC/6000 environment that will be described later) finds the file in CMVC/6000 when you select a class to edit its header or source code. The ability to check a file in and out of CMVC/6000 was added to the editor's pull-down menus.

Making the transition from Teamwork to ROSE was simple since we could reuse much of the customization. Developers demanded more support for the Booch notation and the semantic checking provided by Rational ROSE. To customize ROSE, we ported the customizations for editing source and header files. Supporting documentation required additional work because the open interfaces to ROSE are not the same as in Teamwork.

When we moved to Rational ROSE, we needed a mechanism to allow multiple users to access the same set of design diagrams. Diagrams were locked at the individual level with Teamwork. Because ROSE does semantic checks across diagrams, it locks at the project level. Since this was not acceptable for our purposes, we integrated ROSE with our configuration management system, allowing users to check out portions of a project they are working on.

It was considered too large an investment for us to add to ROSE the capability of generating code from design models. Although customizing CASE tools is difficult, it is not impossible, and yields benefits to the developers. If CASE tools are too hard to use, they will not be used, and that leads to a whole different set of problems.

Class Browser

Reuse is key to the OO development process. A class browser is used to understand the class library and the system that is being constructed. The "Leverage Existing Classes" in Figure 1 highlights the key role of the class browser: to assist in the search for existing reusable components and to analyze other classes for possible inclusion into the reuse library.

Integrating the class browser into the CM system is essential to finding reusable classes that emerge as the project proceeds.

Initially, we began using the C Set++ for AIX: Source Code Browser. Users rejected this browser after many attempts to adopt it for the following reasons:

- This browser requires classes to be compilable. It also requires a program to be written that uses any classes that will be browsed, because the compiler generates browser information. Although this requirement originally sounded harmless, we found that it made browsing code under development or browsing new code from an outside source very difficult.

- Since this browser offers a wealth of information and our project is large, a single browser session on all classes in the project is impossible to get running. Once it begins running, the performance is unacceptable.

- Our users did not like the GUI to this browser, so we developed our own browser. We produced a tool that did not exhibit the same problems as described above. The new browser was developed with considerable input from our user community. One key reason for the success of this browser is that it does not try to do everything that the C Set++ for AIX: Source Code Browser attempted to do. This yields much better performance and requires less from the user.

We may still use the C Set++ for AIX: Source Code Browser after the project is completed and in maintenance mode. It provides valuable information about the overall program structure, the use of classes, methods, and so on. This browser has also matured significantly since our project began.

Integrating the class browser into the CM system is essential to finding reusable classes that emerge as the project proceeds.

Code Analyzer

The code analyzer tool analyzes the interface and implementations written in the target language. It typically looks for inappropriate or high-risk use of language features. It should be tailored to enforce local coding conventions. The code analyzer must be able to do the following:

- Robustly parse and semantically understand standard C++ code
- Issue messages about source code that violates coding conventions
- Issue messages about source code that violate the rules of good semantic C++ programming, and warn about traps and pitfalls of the language
- Issue messages about syntax errors
- Tailor analysis to your project's standards and conventions
- Allow messages to be tailored
- Turn off messages about included files
- Suspend analysis of certain messages directly in the source code

Together with the CASE tools, the code analyzer is among the tools most closely integrated with the development process. Our OO development process shows that output from code analysis is required in addition to reviewing the models that are output from the CASE tools and the actual code. The code analyzer, therefore, must easily analyze a series of interfaces separately, or a series of interfaces and associated implementations. This output must be easily integrated with the documentation tools that are used to build the various design packages.

Flexibility is another required characteristic of the code analyzer. As coding conventions evolve and additional rules of analysis are determined, the analyzer must be easily modified to accommodate them.

Our project used a C++ code analyzer that we developed locally to enforce minor conventions such as naming, program structure, and so on. We also used the code analyzer to help programmers avoid some of the traps and pitfalls of C++. The analyzer was used to promote the use of C++ as an OOP language and to address considerations of the execution environment of the code being developed.

The code analyzer is a good educational tool for less-experienced C++ programmers. But some developers reject the "opinions" of such a tool until they are burned in a significant enough way to illustrate the importance of the analyzer.

Documentation Tools

Documentation tools must work together with the CASE and CM tools. Depending on the vehicle used to distribute and review the evolving object-oriented design, separate tools may be necessary. The key is simple design packages, versioning of design information, and flexibility in providing the necessary outputs. It is essential that developers focus on OO development rather than document creation.

Another consideration is the possibility of online verifications. Tools that make this easy will encourage additional inspection of material and can have a positive impact on the overall project.

Our project used traditional mainframe-based documentation tools and augmented them with diagrams from the CASE tools. Since generating review packages is still a manual task in our environment, each developer performed this task differently, often leveraging some of the CASE tool customizations that were described earlier. Although some online design and code inspections occurred, traditional review meetings with hardcopy material are still the norm.

Project Management

Project management tools assist in tracking inspections and collecting data related to the quality and productivity of the development project. Tight integration to the development process is required.

Dependency management capabilities are essential in project management tools. The incremental nature of the development process produces many dependencies. Understanding these dependencies is key to managing OO software development projects. The tools must provide management with the necessary information.

Some tools were effective when used as always, but others often required creativity in adapting them to the new methodology. We continue to use traditional project management tools because we considered supporting developers a higher priority than improved project management tools.

Prototyping Tools

Prototyping tools include tools for creating user interfaces, requirements validation, and design exploration.

Prototyping tools include tools for creating user interfaces, requirements validation, or design exploration. These tools should use the same language and class libraries as the final product, when possible.

No specific prototyping tool was available to our developers, but AIX (the target execution platform) has been used for prototypes. The software being developed will ultimately execute on another target, but AIX has been more convenient during prototypical stages. AIX has better debugging support than the project's execution

platform, and the tasks of linking and loading are much simpler.

Language-Sensitive Editors

A language-sensitive editor can provide assistance in reading and writing code and should be integrated with the CM, debugger, compiler, and CASE tools.

Our project uses the ez editor and related tools from the Andrew Toolkit. This editor provides a source view of C++ code that is knowledgeable of the language and its syntax. The editor formats different C++ constructs according to how the developer chooses to view the source. The source code editor is common between the CASE tools, CM tool, code analyzer, compiler, browser, and the debugger.

Class Library

A basic tool in the OO development environment is a foundation class library that contains types or classes that support basic data structures such as lists, queues, and stacks. The library should be optimized to the execution environments that are being targeted. Many foundation class libraries are available in the market today.

A GUI class library may also be necessary, depending on the domain of the system being built.

Our project has a foundation class library that was retargeted for the project's execution environment. The library supports the same interfaces for our prototyping and execution platforms. Applicable frameworks were not available at the start of the project. The development environment is currently being extended to better support development with frameworks.

Compiler

The compiler must be integrated into the CM and the build system. Integration with the editor and build system should support *compile/next-error*, a feature that places the user in the correct position within the source code when a compiler error message is selected.

Translators from C++ to C were used in the early stages of the project because no C++ compilers were available to support the execution target. Today, we use a common compiler front-end with support for multiple execution targets (to allow for prototyping).

Debugger

An integrated source-level debugger is essential for integrating the compiler, editor, and CM. The debugger must support object-orientation, not just offer what traditional debuggers have offered for years.

Our project used locally built debuggers to support our execution target. No commercially available debuggers support this target. Currently, the debugger on our prototyping platform is better than the debugger on the final execution platform because of the graphical way that the debugger on the prototyping platform displays objects. As debugger work continues, this difference in debuggers should diminish.

Most projects should be able to leverage commercially available debuggers, provided that they are open and can be integrated with the rest of the toolset.

Tool Integration Platform

An integrated source-level debugger is essential for integrating the compiler, editor, and CM.

The tool integration platform enables the integration of the rest of the tools described in this section. Tool-to-tool communication allows different tools to work together. Developers may look to a single source for all the tools at their disposal. The IBM AIX SDE WorkBench/6000 is an example of this type of tool, which becomes necessary because of the large number of tools and developers involved in a large-scale OO development project.

We integrated many tools into a single tool integration platform. Some tools required much work, while others offered integration with our platform as a feature.

We chose the IBM AIX SDE Workbench/6000 (Workbench/6000) to integrate tools. Each tool in the environment is registered with WorkBench/6000, which facilitates tool-to-tool communication and keeps developers focusing on the tasks of software development rather than the underlying tools that make up the environment. Developers do tasks necessary for development. Tools are launched by the system that support the tasks.

Three types of tools are involved in the WorkBench/6000 environment:

- **Tools shipped with SDE WorkBench/6000**: The basic tools shipped with the Workbench/6000 environment were modified to meet our requirements for function and our local preferences. First, we developed a replacement for the Workbench/6000 Development Manager (DM). This DM has more function than the product version and is more tightly coupled to our CM tool. Next, we upgraded our local editor to support the

necessary Editor Broadcast Message Server (BMS) messages. This gave our developers integrated support for the same editor that they were using before the release of the Workbench/6000-based environment.

- **Tools already integrated into SDE WorkBench/6000**: Integrating this type of tool is straightforward. Installation scripts normally accompany the tool and update the control files necessary to run the tool in an integrated mode. For example, control information found in the `.softinit` file and `softypes` directory is usually updated by these scripts. We use the Andrew File System (AFS®) and have a global version of these files that contain default values. We were often forced to modify these scripts to accommodate our distributed environment.
- **Tools locally or vendor-developed that are not integrated:** These tools required the most integration work. Vendor tools that were not yet integrated were a problem. We were usually unable to support all of the BMS messages required for a simple encapsulation because we did not have the source code for the tools. We defined a separate tool class for the class browser and Teamwork CASE to support starting the tool from either the Tool Manager or Development Manager.

Object-oriented projects put pressure on the CM system to support iteration and the structure of an object-oriented system.

It was easier to integrate locally owned tools because we controlled the source code. We integrated these to provide a consistent view of the tools development environment. Many developers were familiar with a consistent, integrated toolset on a mainframe they had used in previous projects. Having many tools with different user interfaces and usage semantics is very confusing. By integrating them, we could eliminate some confusion. Educating developers about these tools makes the transition easier.

Configuration Management

The CM system required for OO development is very important. Many previously described tools leverage the CM system to provide versioning and check-in/checkout capability—both essential in a large software project.

Object-oriented projects put pressure on the CM system to support iteration and the structure of an object-oriented system. This is necessary to effectively manage the source code, design the artifacts of the development process, and produce builds or drivers on a timely basis.

Our project initially used a Workbench/6000 integrated version of Revision Control System (RCS) called *softrcs*, but we quickly needed more capability than this entry-level tool provided. Migration to a customized CMVC/6000 gave us more functions and enabled us to control access to files and to structure our project so we could continue to support a growing number of developers. We combined CMVC/6000 with AFS and much add-on code to formulate our CM solution.

Before explaining the customization, a brief overview of the file tree is necessary. Our project worked with the hierarchical directory structure and used directory names to provide partitioning of the large amounts of data inherent in a project of this size. A strong relationship exists between directory names and the family—release and component qualifiers required for CMVC/6000. We architected the structure of the source file tree to match the high-level architecture of the system being constructed. This relationship enables developers to navigate, and easily, or at least methodically, find interfaces or implementations of the various classes that make up the system. There are architected places for interfaces that are exported from categories, highly reusable components, and local scoped classes. This facilitates reuse and provides a structure to understand the system.

Beyond this structure, the incremental development process requires that various levels of source code be worked on simultaneously. One driver can be in a test phase, while another is being coded, while yet a third may be in the design phase. All will have active code. Reducing developer confusion in this area and providing flexible compilation options were key goals.

Locating the right file (header file or implementation) is a complicated task that must be done efficiently. The compilation process and the CASE tools require this searching capability.

Once the basic structure was in place, and tools supported the movement and location of code within this structure, the `make`-based compilation model was modified to support the hierarchical structure and the various versions of source code that might be required for doing builds.

A complex set of new tools was written so developers did not have to deal directly with `make` files, while also providing fast and intelligent build capabilities that leveraged our distributed computing resources. Automated dependency analysis also eliminated build failures due to inaccurate information that might be supplied by developers in `make` files. This required that we add support to

Tool Evolution				
Tool Category	**Phase 1**	**Phase 2**	**Phase 3**	**Phase 4**
Editor	ATK's ez	ATK's ez	ATK's ez	ATK's ez
Compiler	Internal tool (Prototype platform)	Internal tool (Prototype platform) (Beta for target)	Internal tool (Prototype platform) (Target platform)	Internal tool (Prototype platform) (Target platform)
CASE	Teamwork (Not integrated)	Teamwork	Teamwork ROSE	Teamwork ROSE
Library Control	softrcs	CMVC/6000	CMVC/6000	CMVC/6000
Debugger		Internal tool (Prototype platform)	Internal tool (Prototype platform) (Beta for target)	Internal tool (Prototype platform) (Beta for target)
Class Library		Internal tool (Prototype platform)	Internal tool (Prototype platform) (Target platform)	Internal tool (Prototype platform) (Target platform)
Browser		C Set++ for AIX: Source Code Browser	C Set++ for AIX: Source Code Browser	C Set++ for AIX: Source Code Browser Internal browser
Code Analyzer		Internal tool (Not integrated)	Internal tool	Internal tool

Figure 2. Phases of tool evolution

the basic CMVC/6000 product. Once completed, we had a complicated, yet usable environment that supports many developers.

Customization was critical to allow us to structure the files for developers to use while also supporting large-scale compilation throughput requirements necessary for doing full-driver and user-oriented builds.

The Development Platform

The choice of the development platform—where these tools run—is a key part of the OO development paradigm. This choice is influenced by the execution environments for the system being built and by the availability of the various tools on particular platforms. Our experience shows that a high-powered workstation on each developer's desk supported by a high-speed LAN is a critical part of the infrastructure necessary for the OO development process to be successfully utilized. This environment and the associated tools needed to support the OO development process must be understood, planned for, budgeted for, and appropriately integrated.

The next section describes which tools were available as the toolset moved from a loosely bound set of independent tools to an integrated OO development environment.

Tools Environment Evolution

The development environment evolved over time. The resources to develop, acquire, and integrate these tools were limited, so tools support was offered on a just-in-time basis. The iterative nature of the process, however, required that the toolset become completely available very quickly. Figure 2 shows the development phases of the toolset.

Phase 1. This phase allowed developers to design and code in a flexible environment. Some developers were apprehensive about moving to object-orientation and could not begin work until there was some sign of a real compiler on the execution platform. We delivered this to help the early adopters of the project with their design and prototyping. They also needed help in getting more developers accustomed to OOP.

Phase 2. The second phase was based on experiences and feedback from users of the first toolset. Although more complete, this toolset still lacked robustness, integration, and target support. A compiler that worked on the final target platform was necessary to show that the toolset could support this project adequately.

Moving to CMVC/6000 was also key in this phase. It was important to reach the final code repository as quickly as possible to reduce the future costs of migration. The code analyzer, class library, and class browser gave developers a complete toolset. Although integration was still lacking, getting these types of tools to developers was important so we could assess their value to the project and make our support and customization investments wisely. Since this was our first very large C++ project, some tools were not demanded by the developers, probably because they were unaware of them. It was our responsibility to make them aware of as many tools as possible.

Phase 3. This phase delivered the Rational ROSE CASE tool. Teamwork, with conventions for Booch-notation diagrams, no longer met our needs. As more developers moved from analysis to design and code, we needed the additional notation and function provided by ROSE.

The rest of the toolset was enhanced and integrated based on problems we encountered as the toolset was used. Simplifying the workflow associated with the development activity was driving enhancements in this phase.

Phase 4. This is the toolset we use today. We are still working on the compiler's optimization capability. We have made performance improvements, but the number of developers is also increasing. Because of this, performance work continues.

The object-oriented paradigm has great potential for building high-quality software systems when it is effectively complemented with an industrial-strength tools environment.

Future Directions

Use of the environment results in many suggestions for improvement. As the focus shifts from creating components to reusing them, the tools environment will also have to evolve.

Incremental compilation. Technology exists for incremental compilation and the requirements for it are usually obvious. In an environment like ours, it is a challenge to manage the information model and data storage requirements for the incremental compile scenario. We believe this requires fine-grained object management capabilities that are not yet available in OO tools environments that support hundreds of users.

Frameworks. The use of frameworks is also increasing. Framework development places added requirements on many tools, such as browsers, and requires additional types of tools that are not in our existing environment.

CASE technology. A round-trip forward and reverse engineering capability, properly integrated into the CM environment, can give more power to developers.

Formal measurements and tools. The metrics required for OO development are still not crisply defined. Good progress, however, is being made in this area, and the tools challenge will soon be to gather and present these measurements effectively.

Tighter integration between design tools and the CM system. With the necessary tool interfaces, the overall design of the system as described in the design tools can be the basis for organizing data in the CM system. The CM tool, on the other hand, can also be the repository for the design data. Our future goal is to keep all relevant development data (design, code, metrics, and others) in the same logical location in the CM system. All must be versioned, found, reused, and fed into the system build process.

Conclusion

Constructing and successfully deploying a development tools environment for large-scale OO development is a major undertaking, and requires many different organizations to cooperate in order to build the appropriate infrastructure.

Object-oriented technology can effectively solve many small to medium-sized programming problems. The concepts of object-orientation can also be applied, with careful planning and investment, to the large-scale problems being solved by the large software development organizations prevalent in business today. The object-oriented paradigm has great potential for building high-quality software systems when it is effectively complemented with an industrial-strength tools environment. Customization for an organization's development needs will always be necessary, but hopefully products will evolve to allow for easy customization and integration.

Michael J. Branson, IBM Corporation, AS/400 Division, 3605 Highway 52 N., Rochester, MN 55901. Internet: mjb@rchvmw3.vnet.ibm.com. Mr. Branson is an advisory programmer, tools strategist, and object-oriented consultant to internal projects in Rochester. He has a BS in Computer Science from Purdue University.

Eric N. Herness, IBM Corporation, Personal Software Products Division, 3605 Highway 52 N., Rochester, MN 55901. Internet: herness@vnet.ibm.com. Mr. Herness is a senior programmer working in object-oriented systems development. He has a BS in Business Administration from the University of Wisconsin at Eau Claire and an MBA from the Carlson School of Management at the University of Minnesota.