Thomas R. Arnold and William A. Fuson

"In A Perfect World"

The pivotal scene in the film *A Perfect World* sums up the problems surrounding testing object-oriented (OO) applications in a development environment. In the film, the authorities in charge of a manhunt say "IAPW (In A Perfect World) we would link arms and march across Texas until we find the culprits," while the sociologist involved says "IAPW we wouldn't have to hunt for anyone." In the first case, infinite resources solve the problem; in the second, infinite cultural change avoids the problem. We might add to this "neither of which corresponds with large-project reality."

In the "real world" of large projects, involving legacy systems, non-OO interfaces, noninfinite resources, non-infinitely applicable tools, competing and clashing features, and noninfinite delivery windows, testing is thrown back into the trade-off world along with all of the other trade-offs involved in engineering and business.

This article presents observations and lessons learned from several OO projects at Mead Data Central (MDC). The projects occurred over a span of approximately two years and were not related in terms of project teams, production objectives, or project management. While MDC conducted several OO projects prior to the ones summarized here, these are the most recent and the ones most

carefully analyzed for objective lessons, metric-based assessment, and production readiness. (For more information about the business and development organization of MDC, see the sidebar "Mead Data Central's Product and Development Organization".)

Testing OO Systems

We begin by summarizing the lessons learned in designing, developing, and testing several object-oriented applications packaged as Unix servers and cooperating client-server pairs, designed and developed by several teams of programmers and engineers, working iteratively. Several tools and tool suites were used during the development to analyze either the source code or the executables for compliance issues, test coverages, performance profiling, and memory-leak detection. In-house development environment support tools were created to facilitate testing.

Our object-oriented applications tend to be a mix of "pure" OO code, in our case written using C++, and legacy and third-party vendor code, typically written in C (e.g., RPC, GUI, database front-ends). Typically they exist as client-server applications, and span processing platforms (PC, Unix, MVS). Our experiences show that testing in these heterogeneous language and system environments is much more complicated than in a homogeneous language environment—techniques that work in the small isolation don't scale up well.

Testing is a multidimensional domain. In a perfect world we would subject our code to tools that analyze the source code to comment on structure, complexity, and/or violation of development criteria. We would instrument the code to provide code coverage, and to check for memory leaks, using third-party tools as well as traditional Unix tools such as gprof and tcov. None of these approaches is successful in all cases, as will be discussed.

Levels of testing. Testing of our OO software components exists at several levels:

- unit level: C++ classes as the unit of test.
- subsystem level: Classes composed of others as the unit of test. This level added nothing of significance and will not be discussed further.
- process level: Each Unix executable.
- domain level: Collaborating executables in a domain, in both client-server and peer-peer relationships.
- cross-domain level: For example, PC to Unix to MVS.

Problems encountered. One of C++'s strengths is that it can (mostly) coexist with legacy and third-party C applications. That coexistence leads to conflicts. We have encountered several examples of feature clash in heterogeneous applications that hindered testing:

- non-thread-safe C++ libraries vs. DCE/RPC, which is inherently multithreaded.
- non-thread-safe memory management routines, interacting with DCE implementations and colliding with third-party vendor tools, rendering useless the tools most necessary to test for memory leaks in the class of programs (daemons) most susceptible to leaks and most mission-critical in operation.
- lack of multiplatform multithread debuggers.
- different exception models, again in the DCE versus C++ realm.

Aspects of objects tested. There are several aspects to testing objects as characterized by C++ classes

- behavior: The comparatively easy part, since this is the specification
- good citizenship: How well the object coexists with others in the application (addressed later in the sub-section "Code analysis tools for good citizenship")
- consistency: A large project requires a consistent approach to automating testing to the extent reasonable to avoid anarchy during integration (even under iterative development)
- processes: Client-server computing only superficially matches the OO model as implemented by C++.

Well-designed objects tend to be easy to test for expected behavior, since the behavior is an integral part of the object's specification. Furthermore, hierarchies derived from abstract base classes tend to reuse more test driver code, simplifying the testing setup software.

The encapsulation inherent in objects also led to the ability to simplify the testing of the applications built around communicating objects. In our case, the various applications (Unix processes) communicate through the abstraction of "queues." These queues separated naturally into client and server stubs that hid the underlying communication mechanism. Early in development the communication mechanism chosen for a simple implementation was Unix files, which allowed us to automate the input and output of the processes by building process-based test drivers and using existing tools to compare expected output to actual output among the "queues."

A side effect of this approach was that we could successfully run a memory leak detection tool against this version of the software. This particular tool failed when run under some implementations of DCE/RPC, due to reimplementation of the memory management interfaces to ensure thread safety.

When multiple processes were to be tested, the shared queues were replaced with named pipes (FIFOs), and with no changes in the rest of the source code we could test the system-level executables. This allowed earlier "integration" testing, since part of our development environment is built around a "private workspace/public baseline" scheme, so access to delivered software is always available for testing, even in an individual developer's workspace.

Architecturally, our ultimate "queue" implementation was based on RPC calls, rather than FIFOs. Once the simple integration testing had proceeded enough to verify in the rather simple FIFO model that the interfaces were reliable, we were able to replace the FIFO implementation with an equivalent RPC one, since the implementation details were completely encapsulated in the member functions and private functions of the "queue" classes. Errors introduced when the queue implementation was changed were clearly localized to the implementation, since the

Mead Data Central's Product and Development Organization

Mead Data Central (MDC) offers an on-line, computer-aided research product that customers use to conduct research on a variety of legal, financial, medical, news, and business topics. The service is international and offers access to literally hundreds of information sources from an international collection of providers (the data owners). Customers accessing MDC's data warehouse have immediate access to nearly one terabyte of on-line information that is updated and expanded throughout the day. Some of this information, such as statutory law, is public domain, while other types, such as periodicals, are proprietary to information providers.

Like most commercial on-line services, MDC's software must operate in a very complex and changing environment. High performance, reliability, repeatability, and accuracy are essential attributes for developed software. Production software must operate almost continuously (24 hours per day, 7 days a week, with a short daily downtime window), providing rapid and consistent response times. Customers must have uninterrupted access to research services and once access has been gained, their sessions must be protected against interruption or loss.

The impact of development errors that cause system failures or interruption of customer sessions is substantial, since our systems routinely support over 2,000 concurrent users, over 75,000 discrete user sessions per day, and in excess of 400 database updates daily. Supporting the on-line services, MDC also has an extensive set of internal applications to receive, prepare for production, update production databases, and archive a continuous flow of new data. These support systems directly impact the reliability of MDC's on-line services and are therefore subject to the same requirements for testing and product quality. The development projects summarized in this article address different system needs: data update services to MDC's on-line databases, and system management services for Unix-based distributed system components directly supporting on-line customer sessions.

Implementation Directions

MDC's architectural directions include the implementation of a distributed, client-server system executing in a heterogeneous hardware and software environment. On the commercial product side we are relying on the availability of Open System Foundation products to set a base facilitating the addition of new vendor products and environments. On the internal development side we are relying on the use of object-oriented technologies to improve initial development, increase code reuse, improve software testing and quality, and generally improve our ability to adapt to changing environmental and functional requirements.

Cost-effective creation of affordable, reliable, and high-quality software is an objective that industry constantly strives to achieve, and MDC is no exception. Our migration to a Unix-based development environment is but one step toward improving our development productivity. We have also selected a set of development tools for the Unix environment and are instituting new processes and development methodologies to further enhance our productivity. The development phases we are addressing include:

- product/project design;
- design, development, maintenance, and end-user documentation;
- implementation;
- project/product testing (includes unit, subsystem, integration, validation, and certification); and
- product release and support.

We have seen a wide range of products and methodologies introduced by researchers, practitioners, and vendors that profess to address some or all of these phases. Like many other development organizations, MDC has procured some of these tools for software design, source code control, debugging, documentation and the like—all providing some incremental improvements to our development processes. However, even when properly understood and religiously used on a project-wide scale, they have failed to provide the necessary productivity improvements. We continue to see industry reports that most software projects exceed budget, fail to meet original functional specifications, are usually late, and fail to meet minimum quality standards. These reports together with our own experiences reinforce the assertion that even our best tools and techniques for software development have failed to provide the necessary life-cycle improvements.

Clearly, we needed more than tools and process improvements to meet productivity objectives. In this light MDC has decided to pursue an object-oriented (OO) approach for major new Unix- and PC-based development efforts. The decision does not apply to incremental improvements to legacy systems (predominantly MVS applications written in IBM BAL, Cobol, Pascal, or PL/1). Very few if any new development efforts at MDC will be entirely independent and able to adhere to the OO approach for every component. Consequently, we must design our own applications that take advantage of the underlying OO technology and effectively interoperate with the legacy systems. With these constraints even new OO development efforts will not realize all the benefits of a pure OO application.

The adoption of the OO paradigm has not been an easy one. We have been encumbered by the scarcity of industry knowledge (as it relates to large-scale production applications), experience base, overly zealous vendor claims, a general lack of the integrated development tools and training necessary for development of major, large-scale mission-critical applications. The actual commitment to an OO approach has been several years in the making, with multiple false starts and learning difficulties. As an organization we have encountered and met challenges in recruiting, training, tool development, vendor support, and other problems characteristic of early adoption of technology. Although the journey has not been easy we continue to experience new success stories as developers become more comfortable with the environment and their skills.

Over the past few years we have increasingly benefitted from the growth in commercially available (and substantially improved) training, tools for design and implementation, and a general maturing of the OO technology. While many of the products, particularly compilers, code analyzers, and design tools are substantially better than the

"best in class" of three years ago, they still fall short in terms of a complete development solution. Specifically, the available tools are still substantially lacking in the analysis, test and maintenance capabilities. This has proved particularly true in MDC's world of legacy systems incrementally moving to the distributed OO, and client-server products.

The primary deficiency for object-oriented software development (at present) is the lack of standard tools and supported environments, integrated development environments (design through testing and release), and adequate training for development staffs. The fact is that the technology is new to the majority of developers and vendors alike, and there are almost as many variations of the OO concept as there are vendors selling products. Certainly there are many products that profess to offer object or object-like capabilities, but for the most part we have found them lacking in functionality, robustness, scalability, compatibility, and in some cases reliability. This has proved to be even more true when we consider the testing-related phases of software development.

Improvements in testing techniques and the necessary support tools continue to lag development tools such as design aids, compilers, and code analyzers, a fact that has hindered our transition to an OO development base. Our projects suffered from this lack of testing tools and techniques. Our testing environments, strategies, and tools are discussed in more detail in this article but some general assertions continue to hold true:

- Unit and subsystem testing were conducted using handcrafted tools, often with a test behavior provided in the basic class designs and implementation.
- Most testing was done through total application functional testing.
- Testing was manually intensive and required substantial effort.
- Testing was not automated, and attempts to reuse testing procedures were not very successful.

Ideally, the tools to facilitate testing of object components would be available off the shelf from a variety of vendors. Unfortunately, that is not presently the case. Users must still understand their problems, the type of testing required, basic testing concepts, and then possess the capability to design and implement appropriate testing classes, behaviors, and data. MDC has had to adapt existing tools and techniques as well as create new internal processes to allow even minimally acceptable testing of OO applications.

Testing Strategy

MDC's testing strategy is to provide an environment supporting automated testing, results evaluation, and report generation. This strategy includes unit and integration testing, certification, verification, stress, and performance testing. Ultimately this environment will be capable of testing load-related characteristics of a dynamic, distributed, client-server application and will be fully automated wherever possible. This automation must, as a minimum, provide for automated test execution and results evaluation.

In addition to supporting test execution and results analysis this target environment will support definition, planning and preparation of test cases and their associated standard results sets at the unit-, subsystem-, or component-, and complete-product level. It will generate reports of code complexity, inheritance trees, test coverage, memory management, naming conflicts, and thread safety. We have found this information to be critical and often suspect areas when integrating purchased products, multiple class libraries (regardless of origin), and internally developed products. We have also proved that it is extremely difficult to adequately test and certify these aspects once the code is in test in a distributed, client-server environment.

A further complication for the on-line portions of the system is the fact that the new development components must integrate with a combination of existing host-centric yet distributed applications and more recently completed distributed system components. The legacy host systems are comprised of several million lines of IBM Assembler and Pascal code and C and/or C++ code in the newer development distributed systems. MDC's legacy host system is a tightly coupled, multihost system in its own right.

These existing systems are a combination of:

- MVS-based IBM assembler, and Pascal system and application components
- Unix-based OO, and C++
- Unix-based structured analysis and coding with C++ and C
- Unix-based SQL and RDBMS applications
- Unix- and MVS-based various third-party applications (predominately non-OO at present)
- a variety of vendor-provided Unix and MVS system and application packages

Staff Characteristics

We have diverse levels of expertise on the development teams, from little OO testing experience to very experienced. Our resources were also limited, so that the OO-experienced staff didn't have time to mentor the less experienced staff unless they were working on the same component together. Experience in producing high-performance and reliable production quality products was also highly variable among the staff, so many lessons were learned during certification and actual production. This was particularly true since the staff members most familiar with object technology were the least likely to have substantial production development experience.

As these projects progressed we noticed that each developer thoroughly knew their piece of the system and its functionality. However, they typically were not as 'tuned' to worrying about how to test to determine whether their piece integrated and worked correctly. This seems characteristic of a disturbing tendency in OO and highly modularized development. Specifically, it is very easy for development staff to concentrate only on their portion of the product, and do an excellent job in the process. They may miss the overall product perspective, however, particularly in areas such as performance, reliability, successful integration and functional testing. Left unchecked this is a potentially fatal problem for production software, particularly when high performance and reliability are critical. In the best case, this problem can cause significant rework during the certification and early production phases of a product, in the worst case it can result in a failed development effort.

interfaces had already been tested. (Note: this obviously assumes the RPC implementation is based on a message-passing scheme, rather than a blocking-sequential scheme; i.e., the "result" of the RPC call is roughly "OK, I have it.")

In another project, the client-server interfaces were dealt with in an equivalent way using "proxy" objects as stand-ins for the server objects. The proxy objects, developed by the server developers, were responsible for the implementation details of the communication. The mechanics of the testing and the ease of integration testing across the project were equivalent to the queue model discussed previously.

Development environment support. Objects represented by C++ classes lend themselves to a clean testing setup, since the public interfaces are usually defined early, leading to early generation of test drivers that are delivered along with the classes themselves as an integral part of our development/delivery cycle, appropriate test input and expected output, and a clean way to compare the expected output with the actual output.

Unit and subsystem-level test support was replicated across our development on a per-project basis. Our approach was to deliver the test drivers when a class was declared, and to "grow" the test drivers as the class hierarchy grew. Once the mechanism was in place to automate the generation of regression-test runs for an iteration of development, implemented in our development environment through a combination of Bourne shell scripts and Makefile targets, testing of software became a natural part of the development process. The basics of this approach are summarized in the remainder of this subsection. Representative source listing fragments are included in the Appendix to this article.

In what follows, assume the developer has a class called **myObj,** that is implemented by the files **myObj.h** and **myObj.C.** The development environment was built by the integration team to contain the following components:

- **test_Driver.sh:** The global driver script that automates much of the comparison/logging/cleanup services for the user test drivers.
- Makefile support for "reg_test" targets for all directories, as well as Makefile include files to automate said support in the development directories under developer control.
- Prototype main() and shell scripts to allow developers to cut-and-paste their basic setups.

Each developer was required to produce the following components in support of testing.

For each development directory, a companion test directory containing the following files (for each object such as X):

- **test_myObj.C,** the main program that exercises the class and its interfaces.
- **test_myObj.sh,** which usually follows a boilerplate to configure the environment before invoking several of the methods of the main test driver script, provided with the environment.
- **test_myObj.out.reg,** expected console output of test_myObj.sh, which consists both of output done by the class and the driver, and file comparisons done by the main test support script, test_Driver.sh.
- (optional) alternate input files that are to be read to run the test. This is typically something like **SGML_file.test_myObj.**

Users could regression test the entire suite of software under development, or their particular parts, at their option.

This "black-box" testing was done at each delivery, for each class represented in the system under test. "White-box" testing was the responsibility of the developer and was not carried through the delivery cycle.

Our approach rejected, for the most part, separate compilations with DEBUG delimited code or special test code embedded within the classes as part of the delivered software that was formally subject to acceptance/certification testing, although the implementers were free to use either or both early in their own development. The DEBUG approach, while effective, has the side effect that what you test is not what you deliver, and for large-enough systems, it at least doubles the build times.

The DEBUG option has been found effective in tracking obscure errors and system behavior. However, as noted earlier the DEBUG-option increases compilation time and changes execution characteristics to some extent. The tester and developer must be aware of these realities prior to deciding to follow this path for problem analysis. These concerns rapidly become more significant as system or produce size grows.

Support tool for instrumentation and feature clash. Support tools tend to fall into two areas: source code parsers/analyzers and run-time analyzers. The source code parsers tend to prevent errors and rule out class tests that are otherwise required. (See the following subsection, "Code analysis tools for 'good citzenship,'" for that class of testing.) The run-time analyzers instrumented the executables and generated reports on path coverages, memory usage, leak detections, and so forth.

We built into our environments support for those tools available on our platforms. If there was one constant throughout, it was that large-enough applications, using large-enough class libraries, would defeat virtually every tool in some way. Complexity-analysis tools tended to become swamped when building applications wrapped around subsystems implemented with possibly competing/clashing class libraries. Instrumentation tools failed for one of two reasons:

- scale: too many classes, too many libraries; and
- feature class, mostly between C++ and DCE/multithreading and reimplemented memory management routines.

Code analysis tools for "good citzenship." Testing objects for their expected behaviors is easy. Objects as *good citizens* are much more difficult to test. Such "good citzenship" is represented by such things as correctly written copy constructors and assignment operators, so that third parties, using collection classes not envisioned by the designers will be able to count on instantiating, copying and deleting copies of the objects without

Glossary

Behavior: specification of the functionality, or behavior that the object provides. The action or set of actions completed by the object when it is invoked.

Black-box testing: Testing performed at the system or product-level and is focused on the verification of the complete system. See *Certification testing* and *Validation.*

Certification testing: Testing to determine the consistency, completeness (in functionality) and correctness of the product.

Code bloat: Uncontrolled growth of executables due to code reuse and the inclusion of additional class libraries that may include class definitions that are either extraneous or duplicate. Providing code that will never be used by the including application but which is included by virtue of the reused libraries.

Cyclomatic complexity: A reference or measure of a component's complexity. Helpful in the analysis of code quality, cost to maintain and enhance.

DCE: Distributed computing environment. The specification of the Open Systems Foundation selected technology for distributed computing support. This technology is available from multiple sources and is key to effective creation of distributed applications across heterogeneous platforms.

Domain-level testing: Collection of collaborating processes, without restriction by locality of execution.

Exception models: The methods by which an environment manages exceptions. Key considerations here include how errors are managed. For example, does the environment attempt to recover and return status information and control to the application for subsequent processing, or does it intercept the error condition and terminate the offending application?

Inheritance trees: The concept of a tree structure applied to inheritance. The inheritance tree builds from the initial base class and "grows" branches as new classes are derived. A branch grows when a new class is derived from a previously existing class.

Inheritance: The object-oriented concept by which a class possesses all the methods and variables of the classes above them which are on the same branch of the inheritance tree.

Instrumentation data: A reference to data programmatically collected. The data (in the applications referenced by this article) includes performance, capacity, status, error conditions, type activity, and certain defined events about which specific data are collected.

Integration testing: Testing to ensure that the tested entities integrate or communicate correctly. Tests the application interfaces to ensure that components can communicate with one another. Is not directly concerned about the accuracy of the processing results. It is concerned with consistency of the interface and the ability of the participating components to correctly send and receive data.

Life cycle: In the context of software components or products this includes the period of time from initial product conceptualization until the time it is retired from service and support/maintenance of the component is terminated. Most commonly used in reference to the "life-cycle costs" of the product: a reference to the costs incurred in product definition, design, implementation, release, maintenance, and enhancement. The life cycle ends when the product is retired from production, support, and enhancement.

Makefile: A text file containing the complete specification of the actions, code modules, and dependencies necessary to build or "make" an entity. Within the context of this article the "made" entities are most commonly object modules or executable files. However, the entity could just as easily be a document or book.

Memory leaks: A reference to memory that is allocated during execution and that is not released when the requesting process completes execution.

Naming conflicts: Instances where the same name is used to refer to different code elements. This could apply to variables, procedures, and code modules, for example. Commonly found when code modules from multiple development efforts are merged.

Process-level testing: The unit of testing is a single Unix executable.

RPC: Remote procedure call. A programming technique that allows applications to invoke remote and local procedures in similar manners. This technique is well proved and allows developers to create distributed applications without concern for the physical location of called procedures.

Standard results set: The definition of expected test results. This set defines the results of successful completion of a defined test suite.

Subsystem testing: Testing of a collection of units, leading to a major collection that performs a well-defined function or set of functions. Within the OO projects described in this article, the subsystem is a collection of classes.

Test coverage: A description of the amount of product code that is actually tested. Commonly referenced in terms of percentage of source code actually tested by the testing process.

Thread safety: A reference to an application's ability to safely execute multiple threads (see *Threads*). An application can be thread-safe even though it does not actually execute in a multithreaded environment.

Threads: A technique allowing concurrent processing of multiple sequential execution paths. Allows applications to more efficiently exploit the computing power available in a distributed computing environment.

Unit testing: Reference to the smallest "unit" being tested. This testing is normally completed by the unit developer and would be done without integration with the rest of the product. This testing uses a developer-defined set of test cases and is measured against a defined standard results set. This testing should also measure code coverage and attempt to execute all possible code branches. Within the OO projects described in this article, C++ classes are the "units."

Validation: Determination that the final product is correct in terms of functionality and accuracy of results. This occurs at the end of each phase of the development cycle.

White-box testing: Testing performed at the unit- or process-level and is focused on the actual code testing. See *Unit Testing.*

the slicing/dicing problems described by Meyers [2].

This is typified by the failure at some point to correctly implement objects as fully functional concrete data types, in the style characterized by Coplien [1] as "Orthodox Canonical Form." Form is the key word here, since in principle it can be detected by an analysis of the code, whether mechanical or visual. Source code analyzers are applicable here.

Lacking automated support for such static code analysis, we "tested" for good citizenship at code review time. In retrospect I believe the object test drivers should have attempted to do this at test execution time, but we did not do so.

Reuse. An interesting side effect of the relative ease of reuse in C++ is worth mentioning. At one point in our development cycle we decided to evaluate the impact of different third-party C++ libraries on which our core processes depended. Although typical string and collection class interfaces differ somewhat, one programmer, in two days of effort, was able to swap out one string and collection set of classes for another that resulted in cutting our execution wall clock time by a factor of 17.

Instrumented runs in other tests demonstrated that the ratio of objects we constructed to total objects constructed in the application was on the order of 1:2,000. The class ratios

Appendix. Sample Drivers and Scripts

Makefile segment to create/run the test executable:

```
test_%:   test_%.o ../$(DIRLIBTGT)
          $$(CC) $@.o -o $@ $(FLAGS) $(LIB_PATH) $(LIBRARIES) $(LOCAL_LIBS) \
                  $(INC_PATH)
          @echo "   $@ is now ready"
          @echo ""

reg_test:
          @if [ "$(REG_PGMS)"x = ""x ]; then \
                    echo "ERROR: There are no reg tests defined in `pwd`" ;\
          fi
          @for i in $(REG_PGMS) ; do \
                    $(MAKE) reg_$$i ;\
          done

reg_test_%:test_% ../$(DIRLIBTGT)
          -$<.sh
```

Shell script boiler-plate to support reg_test target:

```
#! /bin/sh

#         test_FiltPoss.sh:sets up environment for special SGML file

SGMLFILE=SGML_file.test_FiltPoss;export SGMLFILE
 TESTPGM=`basename $0 .sh`;      export TESTPGM
# RM_PGM=yes;                   export RM_PGM
testname=$TESTPGM

     DBX=${DBX:-dbxtool}

while [ $# -ne 0 ]; do
          case $1 in
                    -dbx)       DO_DBX=$DBX; export DO_DBX;
                                shift;;
                    -*|*)       break;;
          esac
done

test_Driver.sh run_test         $testname  $*
test_Driver.sh comp_out_to_exp $testname  $*

if [ $? -eq 0 ]; then
          test_Driver.sh reg_test_ok     $testname $*
else
          test_Driver.sh reg_test_failed $testname $*
fi
```

Excerpts from driver script for test support.

```
#! /bin/sh

#         test_Driver.sh

# ........

#######
usage()
{         cat << END-OF-USAGE | more
test_Driver.sh.usage:  test_Driver.sh [-h] function args, e.g.
          test_Driver.sh run_test        testname
          test_Driver.sh comp_out_to_exp testname
          test_Driver.sh comp_out_to_exp testname outfile expected_outfile
          test_Driver.sh comp_in_to_out  infile   outfile
          test_Driver.sh reg_test_ok     testname
          test_Driver.sh reg_test_failed testname
END-OF-USAGE
}

######
help()
{         cat << END-OF-HELP | more
```

Summary:

```
          test_Driver.sh [-h] arglist

          test_Driver.sh is the test driver shell. The normal mode of invocation
          is for each test_???.sh script to set up the environment, then
          call this script in one of several ways.

                    NOTE: test_??? stands for whatever program test driver
                    is to be run; e.g. test_Token, test_FiltDate, etc.

          This proc encapsulates a variety of utilities shared by various
          functions performed under its control. The various functions
          which this proc supports are:
                    run_test            - executes the test driver program
                    comp_out_to_exp     - compares output to expected output
                    comp_in_to_out      - compares input to output
                    reg_test_ok         - cleans up after successful test
                    reg_test_failed     - cleans up after unsuccessful test

#         .......

END-OF-HELP
}

# ........


############### Input processing ####################

# ........


case $1 in
          run_test)           $*;;      # $0 $1 testname
          comp_out_to_exp)    $*;;      # $0 $1 testnameOR
                                        # $0 $1 testname outfile exp_outfile
          comp_in_to_out)     $*;;      # $0 $1 infile   outfile
          reg_test_ok)        $*;;      # $0 $1 testname
          reg_test_failed)    $*;;      # $0 $1 testname

          *)                  usage;
                              echo "" >&2; echo "You entered \"$0 $*\"" >&2;
                              exit;;
esac
exit
```

were approximately 1:2. There was a lot of execution time spent in code that we did not have to write/debug/test.

A side effect we noticed during integration testing, confirmed in studies such as [3], was code bloat. Reusing significant parts of services developed by other groups for multiplatform, multiproject use that used different class libraries than those chosen for our application led to significant code bloat. Several of the analysis and instrumentation packages had tremendous difficulty functioning—correctly or at all—in this environment.

Conclusions

Testing is difficult. Good objects are more difficult to write well for several reasons:

- Their behaviors and component parts are sometimes complex.
- They are likely to be reused in contexts outside of those envisioned by the designer, particularly as reuse grows within an organization or the use of purchased packages grows.
- They are likely to depend on non-OO software for their implementation, with all of the potentially nonencapsulated side effects inherent in some of that software.
- The C++ object model does not expand well across the process boundaries in a client-server or peer-peer applications environment without care at design time.

Test design is easier:

- Hierarchies tend to reuse test code.
- Public interfaces are defined early, allowing early development of test-driver interfaces, which have more similarities than differences, leading to easier automation of the generation and execution of regression tests.

Feature clash is making analysis/testing more difficult:

- Different exception models between DCE and C++.
- Extant non-thread-safe libraries are in the majority.
- Reuse of existing software usually contains the non-thread-safe code.
- Thread support tools are practically nonexistent, except in the latest versions of some vendor announcements for specific platforms.

The tools to support testing in the heterogeneous environments are lagging far behind the applications being developed and the underlying technologies on which they are based.

Recommendations:

- Use code analysis tools to ease the paper review process.
- Construct a development environment that encourages and facilitates consistent testing projectwide.
- Use self-instrumenting tools to assist in coverage, complexity, and memory leak detection. This is particularly necessary for multithreaded and distributed client-server applications.
- Be prepared to develop in-house tools to overcome the lag time between the leading-edge features and the robust development/debugging support for same. ◼

References

1. Coplien, J. *Advanced C++*. Addison-Wesley, Reading, Mass.
2. Meyers, S. *Effective C++*. Addison-Wesley, Reading, Mass.
3. Srivastava, A. Unreachable procedures in object-oriented programming. *ACM Letters on Program. Languages and Syst. 1*, 4 (Dec. 1992).

About the Authors:

THOMAS R. ARNOLD is a software engineer at Mead Data Central. Current research interests include mapping (the promise of) object-oriented development into the open systems, client-server, and cooperating process models.

WILLIAM A. FUSON is a director of system services development at Mead Data Central. Current research interests include effective adaptation of object-oriented development for large-scale production systems, and integration of object-oriented development into heterogeneous hardware and software systems.

Authors' Present Address: Mead Data Central, P.O. Box 933, Dayton, OH 45401; email: {toma,billf}@meaddata.com