Experience Report: Using RESOLVE/C++ for Commercial Software

Joseph E. Hollingsworth
Holly Software, Inc.
PO Box 480
Floyds Knobs, IN 47119 USA
+1 812 923 1927
jholly@hollysoftware.com

Lori Blankenship
Holly Software, Inc.
PO Box 480
Floyds Knobs, IN 47119 USA
+1 812 923 1927
blankens@cis.ohio-state.edu

Bruce W. Weide
Computer and Information Science
The Ohio State University
Columbus, OH 43210 USA
+1 614 292 1517
weide@cis.ohio-state.edu

ABSTRACT

Academic research sometimes suffers from the "ivory tower" problem: some ideas that sound good in theory do not necessarily work well in practice. An example of research that potentially could impact practice over the next few years is a novel set of component-based software engineering design principles, known as the *RESOLVE discipline*. This discipline has been taught to students for several years [23], and previous papers (e.g., [24]) have reported on student-sized software projects constructed using it. Here, we report on a substantial commercial product family that was engineered using the same principles — applications that we designed, built, and continue to maintain for profit, not as part of a research project. We discuss the impact of adhering to a very prescriptive set of design principles and explain our experience with the resulting applications. Lessons learned should benefit others who might be considering adopting such a component-based software engineering discipline in the future.

Keywords

Component-based software engineering, design-by-contract, design discipline, generic, software reuse.

1. INTRODUCTION

The Reusable Software Research Group (RSRG), a university-based group, has conducted research on the technical aspects of component-based software engineering and reuse since the 1980's [1, 6, 8, 14, 16, 22, 24]. One of us (JEH) contributed to the development of RSRG's RESOLVE discipline for component-based software engineering [6], and believing in its potential for impacting software engineering practice, in the fall of 1993 began developing a commercial application using the RESOLVE discipline. Another (LB) joined this venture later. In addition to commercial interests, we hoped to gauge how well research into the technical design aspects of component-based software engineering would hold up under real-world, commercial application stresses. This experience report discusses lessons learned from this effort.

The commercial product family discussed in this paper was developed for the Microsoft Windows environment using C++ and a specialized version of the RESOLVE discipline known as the RESOLVE/C++ discipline [14, 22, 23]. When we say "the system" or "the application", we really mean a suite of related products consisting of a central application and several unbundled add-ons. The central application and these add-ons have gone through several upgrades and versions over the past 7 years.

Application domain details are proprietary and irrelevant to the rest of this report. But the product can be classified as an information system that serves as a decision making aid through the use of a graphical user interface, real-time interaction with external devices attached to the PC, and extensive reporting capabilities. Users interact with data entry screens using a mouse and keyboard, and can view and print report results and diagrams. Over 100 reports are available through add-on capabilities.

This is not mission-critical software in the usual sense, but it is considered very important to our clients. The application utilizes 250 components consisting of about 100,000 lines of C++ code. Since its introduction 7 years ago, the system has served over 2000 customers. They have generated only two bug reports. The sources of and solutions to these problems, along with the solution to a performance concern, are discussed in Section 3.

Although the application does not fly planes or in any other way involve life-critical processes, it does confront most of the issues common to commercial software systems in vertical markets and which are vital to its commercial success. For example, because it involves interaction with external devices, and there are domain-specific soft-real-time requirements, response time is a concern. Other issues include the usual data storage and portability issues, user friendliness, etc.

Competitive systems have been built using Visual Basic and some of the many common off-the-shelf components (COTS) available for VB. Their clear weakness has been in matching their "system image" to their users' existing mental models of the domain [11]. We were able to capture considerable domain knowledge (in over 200 of the 250 components in the system; see Section 4) and thus bring the application closer to the user, rather than forcing the user to come to the application. This meant that in addition to using the RESOLVE/C++ COTS catalog, we had to design many new components; hence the need for a disciplined approach to component design.

Section 2 briefly reviews RESOLVE-style software component engineering as manifested in this application. Section 3 discusses

our experiences and lessons learned from using this design approach. Section 4 summarizes some metrics about the code, and Section 5 presents our conclusions.

2. HOW WE DESIGNED THE SYSTEM

The system was designed by a team of four people. One (JEH) was already an expert on the RESOLVE discipline and on RESOLVE/C++. One (LB) learned both during the project. The other two contributed domain knowledge and user interface design expertise. The third author of this paper (BWW) and the rest of the RESOLVE academic research group were not involved in the project, even to provide technical support, but were kept apprised of the project's status.

2.1. High-Level Design

The system's high-level design has the following features that come from principles of the RESOLVE discipline:

- The unit of modularity is a generic component (i.e., a template).
- The system is heavily (deeply) layered.
- Dynamically linked structures are layered on a single bottom-level component that hides all pointers and references, so the entire application uses "value semantics" for complex user-defined types as well as scalar types.

The RESOLVE discipline can be applied with any of a number of vehicles to provide the basic computational infrastructure, e.g., with different programming languages such as Ada, Ada-95, C++, Java, and/or with different component middleware, e.g., DLLs, COM, CORBA, etc. (see also Section 3.5). We chose to use DLLs for unbundled add-ons. We did not use COM when it became available because it is unnecessary in our vertical market to communicate with third-party applications where COM might be more useful. There still seems to be no reason to use COM *within* our application because the RESOLVE/C++ discipline provides the requisite composition mechanism at that level.

2.1.1. Unit of Modularity: Generic Component

What is the composition mechanism? Only one is needed: template instantiation. With RESOLVE/C++, the unit of modularity is a generic component, i.e., a C++ class template. Except for scalar types, every object is declared to be of a type provided by an instance of a class template. Thus, each component in the system has a very uniform look and feel to the programmer, which is one of the goals of RESOLVE/C++.

2.1.2. Highly Layered System and Scaling Up

The main function declares an object from the component Main_Window. Main_Window is layered on a number of different objects: Input_Window, Report_Window, Report, etc. Each of these objects is layered on other objects and so on down to the smallest components in the system. From Main_Window down to the deepest non-layered component is over 10 layers.

This approach to high level system design seems to have avoided some of the problems faced by other approaches with respect to scaling-up [17]. No special language constructs are needed to build something bigger than a component, i.e., we have no need for a *subsystem* construct. In this system, the Input_Window and Report_Window components can be viewed as subsystems, if you like. Components that appear higher in the hierarchy are just layered on components at lower layers. Those components are layered on components from lower layers. Eventually, if one keeps going down to the next lower layer, one finds components that are implemented using *raw* C++ representations (e.g., pointers, arrays, built-in ints, etc.). In a controlled setting, there was already evidence to indicate that such a layering approach can lead to significantly improved quality and lower development costs [24].

2.1.3. Dynamically Linked Structures are Layered

Since it is unknown ahead of time how much data is to be processed, most of the container components use dynamic storage allocation to build dynamically linked structures. The traditional way to implement this type of component is to directly represent it using a data structure involving raw C++ pointers. An alternative approach taken by RESOLVE is to use component layering so that all components requiring standard linked structures are layered on another component called Chain_Position [5]. Component layering is not novel, but the Chain_Position component is unusual, and layering all of these other unbounded container components on it is (we believe) new. The benefits are that Chain_Position encapsulates in one component all the raw C++ pointer tricks (e.g., efficient free-storage management) commonly used for building linked structures; and that it provides through its interface an abstract mental model for client reasoning purposes. The implementer of a component layered on Chain_Position uses this abstract mental model for reasoning rather than reasoning in terms of raw C++ pointers. Furthermore, the direct manipulation of raw C++ pointers — in our prior experience, one of the biggest sources of bugs — is localized in the Chain_Position component, where there are fewer than 40 lines of executable code. This code has been rigorously and thoroughly reviewed, and by virtue of the heavy layering, it has been well-tested under fire.

To summarize, we benefit in three ways from this component:

1) The explicit manipulation of pointers is encapsulated in one place, once and for all, and we can leverage off of that work each time we layer a component on Chain_Position.
2) Storage management for linked structures is done here too, so the integrity of dynamic storage allocation is relatively easy to establish. Tricks of the trade that lead to order-of-magnitude efficiency gains in storage deallocation are all encapsulated here.
3) This approach supports modular (i.e., component-wise) reasoning [18, 21, 24] about system behavior, since all pointers are hidden "under the hood" and there is *no aliasing in any of the software we wrote* except that which is carefully controlled inside this component's implementation [4, 19].

2.2. Component-Level Design

Highlights of the system's component-level design are:

- All component design and use is based on design-by-contract, an approach popularized by Bertrand Meyer.
- Movement of data is performed via swapping [4]. There is very limited use of copying.
- Automatic use of constructors and destructors — as opposed to explicit manual use of **new** and **delete** — eliminates storage leaks.
- Components are "fully parameterized" [1] to permit easy reconfiguration and performance tuning.
- Special *checking components* are used during testing and debugging to detect and track down defects in a component's

client code [3]. Consequently, delivered code is not made unnecessarily complex or slow by the addition of precondition checking code inside the reusable components.

Each of these component-level decisions except the first — which is becoming standard practice at this point — is discussed in the sections that follow. These decisions come from a very specific component-design discipline, which appears in [6].

2.2.1. Data Movement Using Swapping

Out of the approximately 250 components (classes, hence types in C++) comprising this system, only seven permit using the assignment operator. Assignment and its cousin, parameter passing by-value, are prohibited for the others by making the C++ `operator =` and the copy constructor private member functions of the class. Instead of using assignment to move objects, the system uses swapping [4]. Each RESOLVE/C++ component is designed and implemented so that it exports a swap operator. The built-in C++ scalar types (e.g., int, char, etc.) have been augmented with a swap operator so that they seamlessly fit in with all the RESOLVE/C++ components.

The seven RESOLVE/C++ components in the system that permit assignment perform deep copying of the value stored in the object being assigned. These domain-specific objects turn out to be the ones that store domain-specific data, which it turns out consists of tuples of integers. Therefore, assignment/copying of this data amounts to copying one tuple of integers to another, which is a fairly inexpensive operation.

To software engineers unfamiliar with the swapping paradigm, the entire idea of getting rid of assignment seems to be a radical approach for the movement of data within a program. Moreover, when we explain that an entire system can be implemented with almost *no* use of assignment and copying, some say it cannot be done, period. However, our application has been designed and implemented based on the swapping paradigm, and it works fine.

A more detailed technical explanation of the rationale for the swapping paradigm, and how it affects component and application design, is contained in the Appendix.

2.2.2. Use of Constructor/Destructor

Each component defines a constructor and destructor for objects of its type. From the client programmer's point of view, the constructor provides the object with its initial conceptual value. Internally, the constructor performs all necessary operations to initialize the internal representation. The destructor is implemented so that all resources allocated to the object are reclaimed.

One goal of any product that can run indefinitely is careful management of "recyclable" resources. Especially important is the absence of storage leaks. Because of the consistent use of the constructor and destructor, and because raw pointer manipulation is concentrated within the Chain_Position component, this goal was fairly easily achieved using code walk-throughs and system monitoring during early development.

2.2.3. Reduced Component Coupling Via Fully Parameterized Components

In general, a component built following the RESOLVE/C++ discipline is generic, i.e., it is a C++ template. Template parameters are of two kinds: conceptual parameters and realization parameters (see [6]). Conceptual parameters are familiar, e.g., for a generic Queue component, the item type enqueued and dequeued is a conceptual parameter. This standard use of templates addresses the problem of *generalization*.

Through the use of realization parameters, concrete-to-concrete component coupling can be eliminated. For example, suppose a Queue component is implemented by layering it on a List component (instead of directly implementing a Queue object with raw C++ pointers and nodes). By deciding to layer, we have made a commitment to use a List in the Queue's representation, i.e., we have bound Queue's implementation to List. But at the time we design and code this implementation of the Queue component, we do *not* have to make a commitment to a particular List implementation. That is, if we have multiple implementations of the same List abstraction, we can delay binding a Queue implementation to any particular one of those List implementations. The traditional approach to layering not only binds the Queue code to the abstract List behavior, it also binds it to a particular implementation of List by using #include to mention a specific pre-existing List implementation (e.g., explicitly #include "List_1.h"). For example, the Standard Template Library (STL) follows this traditional early-binding approach [13].

The fully-parameterized approach in RESOLVE delays that binding by making the List implementation class a template parameter to the Queue implementation, i.e., a realization parameter. By doing this, we permit the client programmer of Queue to choose which List implementation to supply as a template parameter to Queue at template instantiation time — not earlier, i.e., at component design time. The details of how this is performed are outlined in [6, 8, 22, 23].

In this application, almost all components are fully parameterized. This allows the system developer to fine-tune easily the performance of any fully-parameterized component because the developer gets to choose, at component instantiation time, which underlying implementations of other components a given component is layered on. Fully-parameterized components thereby solve the same problems as "abstract factory" components used in some commonly-practiced design patterns, but without introducing any extraneous components that complicate the programmer's life.

2.2.4. Non-defensive Components and Checking Components

Anyone who designs a component to be reused must face the *defensiveness dilemma,* which is simply, "should a component be bulletproof?" Traditional wisdom says "yes", construct the component so that it is bulletproof, i.e., be defensive in order to protect the client from himself. Examples of this approach include component implementations in [2, 10, 12], among many others.

The RESOLVE discipline, on the other hand, says "no", do not make the underlying component bulletproof. This is fully consistent with design-by-contract and offers several benefits :

1) The code that implements the component is more elegant and easier to understand, and hence easier to get right.

2) The performance of the component's operations is better, because when an operation is bulletproof, significant time can be spent checking to make sure that the operation was called appropriately.

3) Once this step has been taken, it is easy to use cleaner methods of providing this protection where it is desirable. In fact,

we have encapsulated defensiveness in another kind of "wrapper" component called a *checking component* [3, 6].

To many software engineers, building components so that they are non-defensive also seems to be a very radical approach. This is not to say that there is no defensiveness in the system, because there is. However, the RESOLVE design discipline recommends that it appear in very precise locations, so it is not spread throughout the system or done on an ad hoc basis. Most of the defensiveness appears in components that are responsible for interaction with the end user. For example, when the system obtains input from the end user, he or she is presented with a finite menu of choices in which currently inappropriate choices are either dimmed or do not appear. This approach has long been standard practice under some user-interface guidelines, e.g., those for the Macintosh. It is becoming more commonplace with Windows applications, too, to the point that it is now relatively rare to find an application program that brings up an alert window with a message such as, "Sorry, this command is not allowed now." So when we say components are not designed to be defensive, we mean that operations' implementations do not defend themselves against inappropriate calls from *other* components *within* the system.

For an operation exported by a component that has no precondition (i.e., one that can be called under any circumstances) this non-defensiveness has no bearing. It is only for an operation that has a precondition that being non-defensive comes into play. For such an operation, it must be possible for the client programmer to find out what the precondition is, and to be able to check whether it holds. Otherwise, the client programmer cannot hope to guarantee that all preconditions are met prior to calling any operation. For example, a precondition for the Dequeue operation of a Queue component might state that a Queue object must not be empty upon calling the Dequeue operation. In order to check this precondition, RESOLVE design principles require that the Queue component must export an operation that permits the client program to check the precondition, e.g., a Size or Is_Empty operation.

Continuing with the Queue example, we can construct a *checking version* of the Queue component called, say, Queue_Checking. Queue_Checking is a fully-parameterized generic component that is layered on the Queue component. Queue_Checking's syntactic interface is exactly the same as Queue's (except for the name). During development of a system, i.e., during testing and debugging, a client programmer needing a Queue type instantiates Queue_Checking instead of Queue. Queue_Checking is implemented so that when the client program calls an operation that has a precondition (e.g., Dequeue), it first checks to see if the size of the queue is greater than zero. If it is, then Queue_Checking merely calls through to the Dequeue operation from the Queue component at the next lower layer. If the size of the queue is zero, then Queue_Checking takes action to notify the programmer (or tester) that a precondition was violated. A violation of a precondition by a client program is viewed as an erroneous client program. We want to expose and eliminate these defects as soon as possible during development and/or testing. Checking components can be more sophisticated than the "one-way" checking components that we used; see [3] for more information.

The beauty of this approach is that when testing has been completed, the checking version of a component can be replaced by the non-checking version and the program will behave exactly as it did with the checking version, except that it will have better performance. The replacement of the checking version by a non-checking version requires no more than a change in the instance declaration section of the system, i.e., no other change in the source code is required. Furthermore, the checking version for a particular abstract component (e.g., a Queue component) can be used with any implementation of it, so long as all Queue implementations satisfy the same external interface (which they will assuming you follow the RESOLVE component discipline).

3. WHAT WE EXPERIENCED

3.1. Defects

Throughout the several-year lifetime of the original application and several upgrades, there have been only two cases where post-release defects have been reported:

- One array was declared too small by one column. Under certain unusual user input conditions, the reports generated by the software were missing one piece of information. It took approximately 20 minutes to track down this defect and change the array size to correct it.
- Output to some printers was problematic. This was not really a defect in the software itself; but a problem with our understanding of how to interact with the operating system when printing. By consulting various resources, eventually we were able to determine the (effectively undocumented) preconditions that had to be satisfied in order to successfully send output to any printer.

We once accidentally delivered an upgrade to some customers that was missing a required DLL, and needless to say this generated several calls. But we do not consider that to be a software defect because the error was not in the application itself, like the other two problems noted above.

If our testing regime had been especially rigorous compared to normal practice — which it was not (see Section 3.3) — this excellent defect record might be attributable to other factors than our strict observance of the RESOLVE/C++ discipline. But we know of nothing else that can explain it.

3.2. Performance Problems

Common wisdom often suggests that heavily layered applications with deep nesting of operation calls might have performance problems. We did not experience this at all. However, a different and more interesting performance issue was encountered. On older equipment, the software at times would take an unacceptably long time to start up. Through careful analysis and monitoring, we determined that most of this time was being spent initializing the objects in the system.

We had observed a similar problem in student-size projects designed using the swapping paradigm, and adopted a similar solution. We simply replaced the implementations of some key abstract components with new implementations based on lazy initialization. Objects whose representation data structures are never needed are simply never allocated or initialized, and those whose representation data structures are needed have their initialization times amortized over the different user commands that cause them to be needed. Since the system was heavily layered, we needed to introduce this technique only in a very few low-level components. This achieved the desired performance improvement in startup time with no user-perceptible penalty during later operation.

3.3. Testing and Debugging

Section 2.2.4 introduced the notion of non-defensive components and their corresponding checking versions. We found out during the development of this system just how helpful checking components can be. To use a checking component requires one additional component instantiation at the point where you create the instance of the non-defensive component. We used these checking components during unit development, unit testing, system integration, and system testing. At the end of system testing and prior to release, we removed the checking components from the system, recompiled, performed additional system testing ("just in case") and then released the system, with no change in system behavior, except that its performance improved a bit.

Recall that the sole purpose of a checking component is to check that when a client calls one of the non-defensive component's operations (e.g., Op_1), that its precondition is satisfied at the time of the call. If the precondition is satisfied, the checking component calls through to the non-defensive component's Op_1 to get the requested work performed. If it is not satisfied, then the checking component notifies the developer/tester with a meaningful message, and halts the application. This approach detected almost all of the client's original defects during *unit* testing and system integration. Very few defects were revealed during *system* testing, and only two were detected post-release.

We also found that it can be helpful to allow checking components to have multiple implementations. For example, an implementation can be specially-crafted to work with a particular development environment's symbolic debugger. (We have changed IDEs during the past 7 years.) If a precondition is not satisfied, then if the IDE supports this, control can be passed to its symbolic debugger, permitting the software developer to examine various object values, the state of the call stack, etc. This can greatly aid the developer in identifying the source of the error and the location of the defective code.

3.4. Compilation Problems

One of the problems faced by users of C++ templates is the method by which they are handled by the compiler and linker. It would be helpful to have an easy way for instances of a template to be separately compiled, and not have to be recompiled when changes are made to other unrelated parts of the system. It appears that the need for separately compiled instances was not given enough attention when templates were added to C++, even though this problem had been addressed by other languages of the day, in particular, Ada. If one is not careful about the actual file organization of a template, then it becomes difficult, if not impossible, to have separately compiled instances.

An instance of a template may be needed in multiple locations of a system for type checking purposes only, while the code for the instance needs to only be generated once. C++ compilers and linkers have handled the generation of code so that it gets generated only once, but that does not eliminate the problem of the compiler having to parse the same code multiple times. Multiple passes of the compiler over the same code will happen if one puts the class specification and the member function bodies all in one include file. We have examined several STL libraries available over the Internet and have found that component-specification and member function bodies are collocated in one file [13]. We have not been able to find any published reports concerning a different organization of the files comprising such components.

Our method of organizing template source code supports type checking anywhere the component instance is used, without forcing the compiler to make multiple passes through component's member function bodies. The compiler makes only one pass through a component's member function bodies when it is asked to generate code for that component instance. This has saved literally hours of compile time throughout the lifetime of the project. The organization puts the template-class specification in a file called .HS (S for Specification). No member function code is present in the .HS file. Member function bodies containing their code are put in a separate .HC (C for Code/implementation) file (see Figure 1). Neither of these files #includes the other.

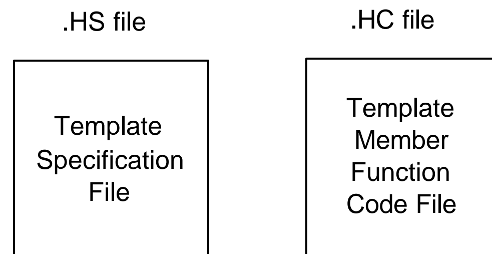

Figure 1

Instantiation of the component is done in a third .H file by using #include of the .HS file only. This instantiation is created to support type checking in other parts of the system (see Figure 2). No object code is generated because only the .HS (template Specification) file was included, not the .HC (member function Code file). Furthermore, if the C++ environment being used supports precompiled headers, the .H file described here will end up being compiled into the precompiled header file, saving even more on compile time.

Finally a .CPP file is created that #includes the .HS and .HC files and declares an instance of the component using C++'s explicit template instantiation construct (see Figure 3). The compiler generates an object code file for the component's member functions.

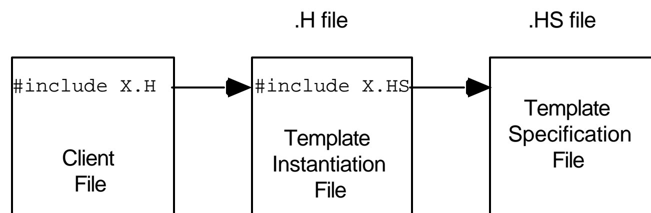

Figure 2

Note that this file organization is a mechanical process that improves compilation performance, and is orthogonal to component design issues such as those addressed by other researchers (e.g., [15]). All our component designs were done by following the RESOLVE discipline and principles.

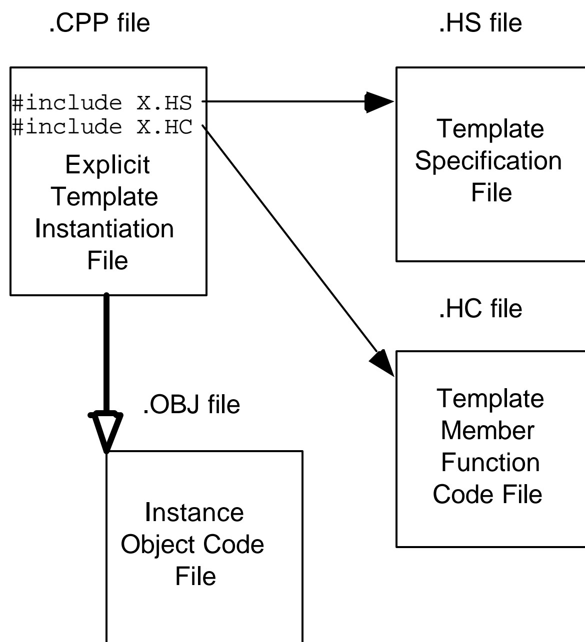

Figure 3

3.5. Compilers/Linkers and Templates

Initially this project for Windows was started using Ada83. Unfortunately, the Ada compilers available at that time (1993-1994) for the PC Windows environment could only handle trivial uses of Ada generics. This forced the switch to C++. Since then, Ada compilers have become more sophisticated and Ada95 has been released. We have not tested more recent compilers to see whether the project could now be done using Ada.

C++ compilers handle most uses of templates pretty smoothly at this point, probably because they are effectively required to do so in order to support the STL. But some linkers still do not handle template instances very well because of naming problems. Fully parameterized components are templates with (in general) many template parameters. The lengths of instance names — as known to C++ and to the linker — grow exponentially with the level of nesting of template instantiation. In a deeply-layered system like ours, this causes the names that must be recognized by the linker to exceed some artificially-imposed limit that apparently works fine in the presence of shallow layering and/or few template parameters. Working around this kind of problem cost us many hours.

3.6. Evolving the System

At the outset of the project, there was the question, "How will the RESOLVE discipline support the evolution of the system?" Our experience with one case of performance tuning, which is one aspect of system evolution, was discussed in Section 3.2, and this definitely was made easier because of the RESOLVE-based system architecture.

The addition of new functionality, another aspect of evolution, was also supported well because the system possessed the modular reasoning property and was highly layered (see Sections 2.1.2 and 2.1.3). We found that software developers joining the project after initial release, as well as veteran project developers, were able to "jump right in the middle" of a module and quickly understand the context — which is explicitly and intentionally restricted to be small by the RESOLVE discipline. They could almost immediately begin to design and implement the changes required to bring about the new functionality. Testing and debugging of new functionality was enhanced during this phase through the use of the discipline's checking components (see Section 3.3).

The key feature supporting easy evolution was our *confidence* that the modular reasoning property would hold. When replacing one component with another, we *knew* that simply observing the discipline would prevent back-door interactions that were unexpected, so we did not waste time looking for them in the code or later testing for their absence. As a result of this confidence, we felt no need to do systematic regression testing at the system level after changes, which saved us an enormous amount of time and money. We did only systematic unit testing of enhancements.

4. METRICS

We did not systematically collect effort data, etc., because this was a commercial venture by a small firm, not a research project. However, we can easily count the artifacts created and used, and classify them along a few key dimensions:

- Total number of components (i.e., templates) in the system: about 250.
- Number of components implemented using raw C++ (e.g., using pointers) or making direct calls to the operating system: about 25.
- Number of layered components: approximately 225. (By layered, we mean components that are *not* implemented by using raw C++ or by making calls to the operating system, but are implemented on top of other components.)
- Number of "horizontal", i.e., general-purpose and potentially cross-application components: 10. These include mainstays from the standard RESOLVE/C++ component catalog, such as List, Stack, Queue, Partial Map, etc. One component of particular interest and utility is Sorting Machine [20]. Each of these components is instantiated multiple times with different template parameters.
- Number of horizontal MS Windows components: 17. These components wrap up MS Windows objects, e.g., Push_Button, Check_Box, Menu, etc. Like those mentioned in the previous bullet item, these components have been reused in other Windows applications. Because their component-level design is based on the RESOLVE/C++ discipline, larger interesting and more useful components can be constructed by composition. For example, if a dialog contains multiple check boxes, one can easily create a list of check boxes by supplying the Check_Box component as a template parameter to the List component. Then the dialog code can iterate through the list of Check_Boxes to dim/undim, check/uncheck, etc.
- Number of domain-specific components: about 220. As noted before, these components contain the domain knowledge and present users with a familiar system conceptual model that helps distinguish the product from its competitors.

5. CONCLUSIONS

Our conclusions about using the RESOLVE/C++ discipline for a commercial software project are as follows:

- The swapping paradigm [4] works. It made it not only possible, but remarkably easy, to address the data movement dilemma in a way that preserved modular reasoning without sacrificing performance. Copying was only occasionally required in our application — and we think this would be the rule in many applications. But when copying was required we always did deep copying. We attribute the remarkably clean bug report history of this product family to the relatively simple reasoning about behavior that resulted from this single most-important design decision.

- Lazy initialization is probably mandatory when using the swapping paradigm. The only fundamentally new programming idiom needed with the swapping paradigm involves "catalyst" objects that serve as temporary holding places for data values that are "in transit". These objects generally don't need to have their representations initialized (although conceptually they must *appear* to be initialized, for reasoning purposes).

- Hiding pointers and references [5] works. Except when implementing cosntructors and destructors for classes that wrap Windows functionality that we had to use (e.g., buttons, printers, etc.), we never worried about storage management. Yet we experienced no storage leaks or other common pointer problems. Compared to other systems we have been involved with, where there were visible pointers/references and aliases almost everywhere, this application has been far easier to maintain.

- Checking components [3] work. Unit testing and system integration with checking components in place dramatically reduced the effort needed for system testing compared to a more traditional approach.

- Fully parameterized components were less important than we thought they might be. In principle, they allow certain kinds of changes to be made easily. For example, substituting lazy initialization for eager initialization in a few components solved a performance problem with relatively little trouble. However, the headaches we faced because of the desire for separate compilation of templates, and especially because of linkers that have trouble with deeply-nested template instances, more than offset the advantages of making most components fully parameterized. We probably wouldn't do that again.

6. ACKNOWLEDGEMENTS

Many colleagues, especially Steve Edwards and Sergey Zhupanov, contributed to the RESOLVE/C++ discipline that we used. We also gratefully acknowledge the reviewers for many helpful suggestions that improved the paper.

7. REFERENCES

[1] Bucci, P., Hollingsworth, J.E., Krone, J., and Weide, B.W., "Implementing Components in RESOLVE", *Software Engineering Notes*, Vol. 19, No. 4, pp. 40 - 51, October, 1994.

[2] Budd, T.A., *Classic Data Structures in C++*, Addison-Wesley, Menlo Park, California, 1994.

[3] Edwards, S.H., Weide, B.W., Hollingsworth, J.E., "A Framework for Detecting Interface Violations in Component-Based Software", *Fifth International Conference on Software Reuse*, IEEE CS Press, June 1998.

[4] Harms, D.E., and Weide, B.W., "Copying and Swapping: Influences on the Design of Resuable Components", *IEEE Transactions on Software Engineering*, Vol. 17, No. 5, pp 424-435, May 1991.

[5] Hollingsworth, J.E. and Weide, B.W., "Engineering 'Unbounded' Reusable Ada Generics", *Proceedings of 10th Annual National Conference on Ada Technology,* Arlington, VA, February 1992.

[6] Hollingsworth, J.E., *Software Component Design-for-Reuse: A Language Independent Discipline Applied to Ada*, Ph.D. thesis, Dept. of Computer & Information Science, The Ohio State University, Columbus, OH, 1992. Available at http://www.cis.ohio-state.edu/rsrg/.

[7] Hollingsworth, J.E. "Uncontrolled Reference Semantics Thwart Local Certifiability", *Proceedings of the Sixth Annual Workshop on Software Reuse*, November 1993. Available at: http://www.umcs.maine.edu/~ftp/wisr/wisr.html.

[8] Hollingsworth, J.E., Sreerama, S., Weide, B.W., and Zhupanov, S., "RESOLVE Components in Ada and C++", *Software Engineering Notes*, Vol. 19, No. 4, pp. 52 - 63, October, 1994.

[9] Hollingsworth, J.E., and Weide, B.W., "Micro-Architecture vs. Macro-Architecture", *Proceedings of the Seventh Annual Workshop on Software Reuse*, August 1995. Available at: http://www.umcs.maine.edu/~ftp/wisr/wisr.html.

[10] Murray, R.B., *C++ Strategies and Tactics*, Addison-Wesley, Reading, Massachusettes, 1993.

[11] Norman, D.A., *The Design of Everyday Things*, Currency Doubleday, New York, 1988.

[12] Sengupta, S., and Korobkin, C.P., *C++: Object-Oriented Data Structures*, Springer-Verlag, New York, 1994.

[13] STL — Hewlett-Packard's downloadable Standard Template Library, from ftp://butler.hpl.hp.com/stl SGI's STL — Silicon Graphics Computer Systems, Inc, downloadable Standard Template Library, from http://www.sgi.com/Technology/STL; Rogue Wave Software, Inc., on-line documentation, from http://www.roguewave.com/support/docs/stdref/index.cfm.

[14] Sitaraman, M., and Weide, B.W., eds., "Component-Based Software Using RESOLVE", *Software Engineering Notes*, Vol. 19, No. 4, pp. 21 - 67*,* October, 1994.

[15] van Hilst, M, and Notkin, D., "Decoupling Change from Design", *ACM SIGSOFT '96*, ACM, 1996, pp. 58-69.

[16] Weide, B.W., Ogden, W.F., and Zweben, S.H., "Reusable Software Components", M. C. Yovits, editor, *Advances in Computers*, Vol.33, Academic Press, 1991, pp. 1-65.

[17] Weide, B.W., and Hollingsworth, J.E., "Scalability of Reuse Technology to Large Systems Requires Local Certifiability", *Proceedings of the Fifth Annual Workshop on Software Reuse*, October 1992.

[18] Weide, B., and Hollingsworth, J. *On Local Certifiability of Software Components*, OSU-CISRC-1/94-TR04, Dept. of Computer and Information Science, Ohio State Univ., Co-

lumbus, OH, January, 1994.

[19] Weide, B.W., Edwards, S.H., Harms, D.E. and Lamb, D.A., "Design and Specification of Iterators Using the Swapping Paradigm", *IEEE Transactions on Software Engineering*, Vol. 20, No. 8, pp. 631 - 643, August 1994.

[20] Weide, B.W., Ogden, W.F. and Sitaraman, M. "Recasting Algorithms to Encourage Reuse", *IEEE Software*, Vol. 11, No. 5, pp. 80 - 88, September 1994.

[21] Weide, B.W., Hollingsworth, J.E. and Heym, W.D., "Reverse Engineering of Legacy Code Exposed", *Proceedings 17th International Conference on Software Engineering,* ACM, April 1995, pp. 327-331.

[22] Weide, B.W., *Software Component Engineering*, OSU Reprographics, Columbus, OH, 1997.

[23] Weide, B.W., and Long, T.J. Software Component Engineering Course Sequence Home Page, from http://www.cis.ohio-state.edu/~weide/sce/now.

[24] Zweben, S.H., Edwards, S.H., Hollingsworth, J.E., and Weide, B.W., "The Effects of Layering and Encapsulation on Software Development Cost and Quality," *IEEE Transactions on Software Engineering*, Vol. 21, No. 3, March 1995.

APPENDIX: THE SWAPPING PARADIGM

A key part of the RESOLVE discipline is what we call the *swapping paradigm.* How to achieve "movement" of data values between variables is a technical problem that needs to be faced by all imperative-language software engineering disciplines that concern themselves with component-level design details. Because there is no way to avoid this problem, and because traditional approaches to dealing with it make trade-offs that introduce serious technical difficulties, we call it the *data movement dilemma*.

To explain why there is a dilemma, we start with a simple question: How does one make some variable (say, *x*) get the value of another variable (say, *y*)? For example, suppose *x* and *y* are variables of type Integer, a type whose mathematical model is a mathematical integer. This means we want to think of the value of *x* as being something like 17 (i.e., not as a string of 32 bits), and the value of *y* as being something like 42. The obvious answer is that we use an assignment statement, like this:

```
x = y;
```

This works fine for Fortran and C programs and for C++ programs with no user-defined types. A problem arises, though, in modern object-oriented software systems. What if *x* and *y* are variables of a type T, where T's mathematical model is relatively complex so its representation is potentially large? Suppose, for example, that T is Set <Integer>; we wish to think of the value of *x* as being something like {1, 34, 16, 13} and the value of *y* as being something like {2, -9, 45, 67, 15, 16, 942, 0}. There are now two options, neither of which is especially attractive:

- Consider the assignment operator and copy constructor for Set to perform "deep" copy, so that after the assignment statement we can think of both *x* and *y* as having the same abstract value. Logically, *x* and *y* must behave independently, too, so changes to *x* do not side-effect the value of *y* and vice versa. This can be terribly inefficient, because without using fancy data-structure tricks that frequently do not apply, both the assignment operator and copy constructor take time linear in the size of *y*'s representation. Big sets simply take a long time to copy and hence to assign.
- Retreat from viewing *x* and *y* as having values that are mathematical sets of mathematical integers, and consider their values to be pointers or references to sets of integers. This certainly fixes the efficiency problem. But it introduces a distressing non-uniformity in reasoning about program behavior: some variables denote values and others denote references. It also means that the assignment operator and copy constructor create aliases, which complicates reasoning about program behavior.

The latter approach has been codified into some modern languages, notably Java. We observe, however, that it is actually far worse than the former from the software engineering standpoint. One reason is that the programmer now must be aware that variables of some types have ordinary values while variables of other types hold object references (it's the objects that have the values). For template components this creates a serious problem. Inside a component that is parameterized by a type (say, Item), there is no way to know *before instantiation time* whether an assignment of one Item to another will assign a value or an object reference. Of course, this can be "fixed" as it is in Java: introduce otherwise-redundant object types such as Integer to shadow value types such as int. We could then require that all actual template parameters be object types. This is not a language-enforced rule in C++, however, so some discipline is required to make it a reality there.

A more serious problem is that the "reference semantics" approach makes aliases run rampant, and aliasing breaks modular reasoning [7]. As noted in Sections 3.3 and 3.6, the ability to do modular reasoning — and to *know* that we could rely on it — was extremely valuable to us in terms of both minimizing testing effort and in terms of the resulting product quality.

Figure 4 summarizes the data movement dilemma faced by someone who wants efficient software about whose behavior it is easy to reason. The conclusion seems to be that this is only attainable by sticking to built-in scalar types — not incidentally, the only types available when the assignment operator was introduced into programming languages — or, at best, by inventing only new user-defined types that admit "small" representations.

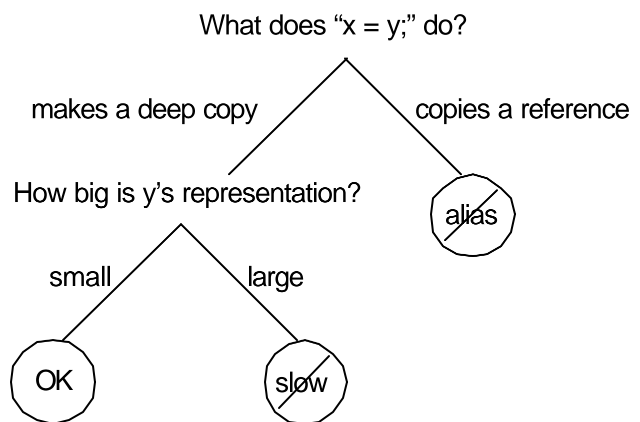

Figure 4

Rather than confronting the dilemma directly, one way out is to revisit the original question and to consider alternatives to the assignment statement as the "obvious" answer to the question: How does one make some variable (say, *x*) get the value of another variable (say, *y*)? There is no inherent requirement that the value of *y* must not change as a result of the data movement process. Realizing this opens the door to many other possibilities. The new value of *x* must be the old value of *y*, but the new value of *y* might be:

- the old value of *y* (to get this behavior we just use assignment, which works well if *y*'s representation is small); or
- undefined; or
- a defined, but arbitrary and unknown value of its type; or
- some particular value of its type, e.g., an initial value; or
- the old value of *x*.

It is beyond the scope of this paper to analyze the pros and cons of all the other possibilities beyond the first one, which is definitely unsatisfactory as a general approach to data movement. Suffice to say that leaving *y* undefined complicates reasoning, although not nearly as much as allowing aliasing; and that leaving *y* with either an arbitrary or a distinguished value of its type is actually quite a reasonable thing to do. However, we have found that the last approach — *swapping* the values of *x* and *y* — is both efficient and safe with respect to modular reasoning, and it results in remarkably few changes to how most programmers write imperative code [4].

One needs to get used to a few new idioms when adopting the swapping paradigm, e.g., for iterating through a collection [19]. The biggest effect of the swapping paradigm, however, is on the design of component interfaces. Consider, for example, a Queue component with operations Enqueue, Dequeue, and Length. What should Enqueue(x) do to the value of *x*? The analysis of this question parallels the analysis of the data movement dilemma as the question was phrased above. The result of analysis is that Enqueue should *consume x*, i.e., it should leave *x* with an initial value of its type.

How can this be accomplished? A raw C++ implementation of the Queue component declares a new variable of the parametric type Item in the body of Enqueue, e.g., the "data" field in a new "node" that is to be inserted in a linked list of nodes. This variable is then swapped with *x*. Swapping simultaneously puts the old value of *x* into the Queue's representation data structure, where it needs to be; and sets the new value of *x* to the initial value for its type that was originally in the data field of the node.

In an implementation of the Queue component that is layered on top of a List implementation, the Enqueue operation simply Inserts *x* at the appropriate place into the List that represents the Queue. If the Insert operation for List is designed using the swapping paradigm, so it consumes its argument's value just like Enqueue does, then this call does exactly what is needed.

In other words, in both these situations, the code that one would have written if using assignment for data movement is changed in at most one respect: assignment of *x* to its place in the Queue's representation is replaced by swapping *x* with its place in the Queue's representation.

Our experience is that a group of components such as those in the RESOLVE/C++ component catalog [23] can be designed according to the swapping paradigm to work together in such a way that programming within the RESOLVE discipline is substantially identical to programming with assignment statements. But the resulting components offer efficiency and/or reasoning advantages over similar components designed in a traditional fashion.

We should be clear that we still do use the assignment operator with built-in scalar types. There is nothing wrong with the following statement from either the efficiency or reasoning standpoints, assuming that *x* and *y* are variables of some built-in scalar type:

```
x = y;
```

The interesting and possibly surprising empirical observation that has been substantiated by our application development is that there is rarely a need for such a statement when *x* and *y* have user-defined types.

Recapping these and other advantages of using the swapping paradigm:

- All variables have uniform value semantics, which allows the modular reasoning that is impossible if reference semantics creep in.
- All pointers and references are hidden deep within the bowels of a few low-level components and are invisible to a client programmer layering on top of them.
- If these low-level components have no storage leaks, then client programs have no storage leaks, and client programmers do not have to worry about where to invoke **new** and **delete** because they simply never do.
- The swapping paradigm is easy for imperative-language programmers to learn and apply.

Other questions often asked about the interactions between the swapping paradigm and other programming language and software engineering issues, such as the role of function operations, assignment of function results to variables, parameter passing, etc., are discussed in [4].