A simple solution to the medical instrumentation software problem

Robert C. Leif[1], S. B. Leif[1], S. H. Leif[1], and E. Bingue[2]

[1]Ada_Med, a Division of Newport Instruments, San Diego, CA 92115.
e-mail: rleif@rleif.com

[2]Ada Software Engineering Education Training (ASEET) Team
e-mail:binguee@sw-eng.falls-church.va.us

ABSTRACT

Medical devices now include a substantial software component, which is both difficult and expensive to produce and maintain. Medical software must be developed according to "Good Manufacturing Practices", GMP. Good Manufacturing Practices as specified by the FDA and ISO require the definition and compliance to a software processes which ensures quality products by specifying a detailed method of software construction. The software process should be based on accepted standards. US Department of Defense software standards and technology can both facilitate the development and improve the quality of medical systems. We describe the advantages of employing Mil-Std-498, Software Development and Documentation, and the Ada programming language. Ada provides the very broad range of functionalities, from embedded real-time to management information systems required by many medical devices. It also includes advanced facilities for object oriented programming and software engineering.

Keywords: Software, Mil-Std-498, Ada, Medical Device, Regulated, Object Oriented, Compiler, Good Manufacturing Practices, GMP

1. INTRODUCTION

Medical Device Development and manufacture is a large regulated industry. Clinical Diagnostic equipment and other medical devices now include a large software component, which must now be developed using GMP. GMP requires the definition and enforcement of software processes which ensure quality products by defining a detailed method of software construction. This is very different from the past commercial model for personal computer software which was to get the product to market in the fastest way possible.

Traditionally, large groups of Beta testers, often more than the total number of users of a medical device, attempted to find problems. Frequent software releases were made to fix the problems discovered during Beta testing and after formal product release. This model is inappropriate for medical device specific software because users cannot safely rely on the data from the Beta device nor can they revalidate their procedures for using this device. The initially released software must satisfy the same quality constraints as the final product. In spite of the long lifetimes of medical devices, their limited market and regulatory requirements often preclude complete revisions of the software. A complete revision would require a new filing with the FDA. The situation with military, aeronautical, and space systems is similar. Extreme reliability[1] is required throughout the long lifetimes of systems where the operating conditions may change. The result of a computer error could be a plane crash or a dead patient.

The key elements that regulatory agencies[2,3] look for in software development are:

1. A documented and adhered to Software Development Process with emphasis on specifying requirements.
2. Written software requirements, with hazard and safety concerns in the requirements.
3. Hazard and safety traceability through design code and testing.
4. Formalized testing with evidence through traceability that hazards and safety issues were addressed.

A Software development process describes how to create software. This definition is at once self-evident and unsatisfactory. While software development processes have been compared to cook-books and road maps, we will address them as a series of "to do" steps with several overall guiding principles.

Our Solution is to apply the techniques developed for producing military software to medical software. These techniques are Mil-Std-498[5] and the Ada programming language[6]. Mil-Std-498 describes the process; while, Ada provides the means.

The process for medical software development should be based on accepted standards. The de novo creation of an in house description of a software process is expensive both in its implementation and its justification to regulatory agencies. Recently, we suggested[7] the use of IEEE standards. Mil-Std-498 is similar in content to the present IEEE standards and is slated to evolve into a joint IEEE and Mil-Std. A careful analysis of the Mil-Std-498 process shows that all elements of software system development which are required by the FDA and ISO 9000 are covered providing a better solution than using IEEE standards. Mil-Std-498 includes a series of Data Item Descriptions (DIDs). Each of these DIDs represents one piece of the overall system development process.

Mil-Std-498 helps the user follow a spiral development process[8]. Many of the early software processes, such as Mil-Std-2167A[9], encouraged the waterfall method. Interestingly, many of the complaints about Ada were really directed towards 2167A. In developing new devices, requiring each step to be completed before the next becomes a forced system where one of two things usually happened. Either the development team ignored the waterfall in order to do the needed regression/ progression development or the product outcome was forced into the waterfall and was isolated from information generated during the development period. This lack of response to feedback and inflexibility during development often had catastrophic results. The specifications should evolve with the product and prototypes constructed as the system evolves. This is identical to the real world of clinical device software[10]. Most clinical systems require prototype software in order to optimize both the hardware and chemistries. This optimization will often result in an improvement in the quality of the data, which will in turn result in improvements to the algorithms and often to other parts of the software. An appropriate term would be the onion method of development.

2. PRODUCT DEVELOPMENT

The present software battle cry of object oriented analysis, object oriented design, etc. is very similar to standard operating procedure for hardware, both electronic and mechanical. The major systems are described as black boxes with well defined interfaces, and then as much as possible is constructed of off the shelf components. In fact, the similarity of hardware and software design process is so great that Hines[11] has employed a common incremental method for both the software and hardware for a military satellite communication system. His recursive methodology is essentially MIL-STD-498. MIL-STD-498 like its predecessor, 2167A, and the IEEE standards that we have previously recommended[7] addresses Software Development and Delivery. The total process that Hines and we propose encompasses a system. This means hardware, software, and reagents as well as their integration and functioning.

The development process can be separated into two stages. The research leading to the development of the initial prototype, which motivates the whole process and the second stage, which starts when there is a formal commitment to develop a marketable system. The first stage, most often, consists of crude prototype hardware or a simple instrument mock-up; offline data analysis software and, if necessary, simple data acquisition software; and manual chemistry. Companies seldom specify requirements without any tangible technology. This first stage is not subject to strict regulatory requirements. However, there are significant savings, if the code developed during research can be reused for development of the ultimate marketable system, which is under the control of the regulatory authorities.

The first step in the second stage of development is a System Requirements Analysis Document. It should be a fusion of market needs, technological (engineering and science) feasibility, and regulatory acceptability. One of the major questions is: can the device be approved by the FDA 510K mechanism rather than the very expensive Premarket Approval Process (PMA). The proposed device must meet or beat the current levels of precision and accuracy of similar pre-existing products.

One of the virtues of Mil-STD- 498 is the associated DID's which come with the standard. They are easy to follow and can serve as the basis for the development of the software procedures and documents. The DIDs provide very useful details that can serve as a checklist as to the scope and content of each document. The way

that the items are numbered within the documents also aides in setting up the associated testing which needs to be done.

The 22 DIDs are comprehensive. They are: Computer Operation Manual, Computer Programming Manual, Database Design Description, Firmware Support Manual, Interface Design Description, Interface Requirements Specification, Operational Concept Description, Software Center Operator Manual, Software Design Description, Software Development Plan, Software Input/Output Manual, Software Installation Plan, Software Product Specification, Software Requirements Specification, System/Subsystem Design Description, System/Subsystem Specification, Software Test Description, Software Test Plan, Software Test Report, Software Transition Plan, Software User Manual, and Software Version Description.

Each of the DIDs starts with a title page which includes a Description/Purpose. For example, below is section 3.1 of the Software Product Specification DID.

> "3.1 The Software Product Specification (SPS) contains or references the executable software, source files, and software support information, including "as built" design information and compilation, build and modification procedures, for a Computer Software Configuration Items (CSCI)."

All of the DIDs employ essentially the same text to describe how to prepare the DID. This is followed by the text that is specific for the DID. This text often with minor modification can serve as the basis for one's own documentation. For instance, Section 6 of the Software Product Specification:

> "6. Requirements traceability. This section shall provide:
>
> a. Traceability from each CSCI source file to the software unit(s) that it implements.
>
> b. Traceability from each software unit to the source files that implement it."

Medical software development must maintain traceability. There must be requirements, design, code and testing results for each requirement.

The DID entitled, "The Software Development Plan" contains a detailed outline which is a list of all the required items. It even includes schedules and activity network.

The Software Requirements Specification is a good example of a DID appropriate for medical software development. The instructions provide a tutorial for writing requirements. For example, 3.3.1 Interface Identification and Diagrams

> "3.3.1 Interface identification and diagrams. This paragraph shall identify the required external interfaces of the CSCI (that is, relationships with other entities that involve sharing or exchange of data). The identification of each interface shall include a project-unique identifier and shall designate the interfacing entities (systems, configuration items, manual operations, etc.) by name, number, version, and documentation references. The identification shall state which items already exist (and therefore impose interface requirements on interfacing items) and which are being developed or modified (thus having interface requirements imposed on them). One or more interface diagrams shall be provided to depict the interfaces."

This DID goes on to require both safety and functional requirements. The outline provided ensure that operation concerns, including quality, safety, and ease of use are met.

Another example is the Interface Requirements Specification DID. Section 4 on Qualification Provisions denotes how each CSCI is to be tested (demonstration, test, analysis, inspection, or special qualification methods). This document also specifies the traceability from each CSCI requirement to the system or subsystem.

A key to all deliverable software is adequate and sufficient testing. The Software Test Plan and the Software Test Descriptions DIDs provide all the organization needed to create test plans and descriptions that continue to follow the same scheme of traceability. At every stage in development, the completed product is fully documented, and tested.

The DIDs and the body of Mil-STD-498 provide both a detailed outline and check list that will greatly facilitate development of both the documentation and the processes required by the regulatory authorities.

Three critical aspects of medical software are planning, configuration management and a corrective action system. Project planning and oversight is the core of all projects. This becomes management's way of guiding and monitoring a project. 498 specifies that planning be performed for each build. This planning includes planning for the contract, the current build, and future expected builds. The 498 planning DID covers all major aspects for overseeing a project. It satisfies both the ISO and FDA in terms of specifying what controls will be in place and how they will be used.

Mil-Std-498 provides strict criteria for configuration management. Developers must provide levels of control for each entity in a project. There must be records which show the status of all entities for the lifetime of the project. The status includes the history of the entity. In terms of medical devices, it is important to retain these records for the lifetime of the product.

A key FDA requirement for medical devices is a corrective action system. Mil-Std-498 presents requirements for such a system:

"a. Inputs to the system shall consist of problem/change reports.

b. The system shall be closed-loop, ensuring that all detected problems are promptly reported and entered into the system, action is initiated on them, resolution is achieved, status is tracked, and records of the problems are maintained for the life of the contract.

c. Each problem shall be classified by category and priority, using the categories and priorities in Appendix C or approved alternatives.

d. Analysis shall be performed to detect trends in the problems reported.

e. corrective actions shall be evaluated to determine whether problems have been resolved, adverse trends have been reversed, and changes have been correctly implemented without introducing additional problems."

Quoted from 5.17.2 Corrective action system.

Mil-Std-498 in the Software Requirement Specification DID provides the organization of both safety and functional requirements. The outline provided ensures that operational concerns, including quality, safety and ease of use are met.

A key to all deliverable software is adequate and sufficient testing. The testing aspect for a project is first addressed throughout the requirements. Mil-Std-498 also provides DIDs on Software Test Plans and Reports.

Requirements must be written in a manner in which they can be tested. Marketing people love to tell the engineers that they want the system 'fast', 'inexpensive', or 'easy to use'. When the engineers deliver the product, Marketing is unhappy because it is not fast enough, or it is too expensive because expensive components were employed to achieve high through-put. Testing is also impossible when there are no criteria to test against. Criteria can be timing, specific outputs or accuracy levels or other discrete comparisons.

There is a major apparent difference between the DOD (Mil-STD-498) and Medical Instrument models of software development. This is the term "Customer". For the DOD, this means the procurement branch and the software manufacturer are separate entities. These entities are mostly Defense contractors; however, they can include a different DOD organization. Thus, the rules have been formulated for an arms length type of relationship. Most of the time, the medical instrument corporation either produces its own software or has a very close relationship with the software provider. However, the initial "customer" for a medical device is <u>not</u> the traditional customer who purchases or leases it. The device has to be sold first to the regulatory agencies. This relationship is also arms length.

3. ADA PROGRAMING LANGUAGE

The Ada language is the major descendant of the Algol/Pascal family of programming languages. It is sufficiently syntactically similar to Pascal; that, it could have been named Pascal++. However, it was quite correctly named after the mother of software, Countess Augusta Ada Lovelace, a mathematician and the daughter of the poet, Lord Byron. While in her twenties, she worked with Charles Babbage on his Difference Engine and thus is considered the world's first computer programmer. Ada is a modern, object oriented, general purpose pro-

gramming language, which was designed to facilitate the application of software engineering concepts, maximize reliability, and provide a breath of capabilities suitable for large software projects. The software required for a medical device has to perform many disparate functions. These range from real-time instrument control to billing. Ada is the only high level language which provides specific syntax to directly implement all of these functions. The latest version of Ada as described in the Language Reference Manuel[6], includes the core language and the following annexes: Systems Programming, Real-Time Systems, Distributed Systems, Information Systems, Numerics, Safety and Security, Interface to Other Languages (C, COBOL, and Fortran), and Obsolescent Features. The use of annexes permits the compiler vendors to implement only those annexes required by their specific markets. For instance, the Package Text_IO.PICTURES, which permits COBOL type formatted output, is irrelevant to a real-time embedded system. Ada has been successfully employed in systems of even greater criticality than medical devices[1].

3.1. Standardization & portability

Ada is the first object oriented language to be described by an international standard, ISO\IEC 8652: 1995. Ada is also a Federal Information Processing Standard (FIPS 119), and an American National Standards Institute (ANSI) standard (ANSI-1815A-1983). Ada code can be made to be extremely portable because portability is one of the of the software engineering goals and principles that was designed into the language. The United States Defense Department, DOD, supported the development of a very detailed language specification, which is described in a very precise language Reference Manual. The DOD requires that any Ada compiler used internally or on a Government contract be validated. For this purpose, the DOD and the international community have supported and maintained an extensive validation suite of over 3,000 tests. Validation performed by Ada Validation Facilities world wide assures with some degree of confidence that compilers operate and that the software successfully compiled by any validated Ada compiler will operate on different hardware configurations. Since Ada includes the ability to specify how objects are represented, even, changes to code that interacts directly with hardware can be minimized.

3.2. Software engineering

Ada is not merely a programming language; it is a vehicle for new software practices and methods for specification, program structuring, development and maintenance. Ada also includes advanced facilities for object oriented programming and multitasking. Ada code is constructed out of packages, which only interact through their specifications. The actual implementation of the methods (functions and procedures) described in the specifications is contained in separately compilable bodies. A special “with” statement must be included in a package to make specifications of other packages visible. These interactions are the side effects that plague software development and testing. This deliberate control of visibility guarantees the scope of the interactions between packages. The separate definition of interfaces (specifications) and their implementations (bodies) in Ada allows an orderly integration, testing, fault isolation, and design review of the software components, and reduces the time to market because of improved integration process.

Since the specifications are compiled before the bodies, it is possible to employ the compiler to test the overall design before any detailed implementation. Design problems that would adversely effect system integration often result in errors during the compilation of the specifications.

The package structure, provides the modularity required to partition the software into manageable components, which permits groups to develop and integrate software. The breakdown of a system into packages facilitates the management of a project[12]. The software project schedule throughout the development life-cycle can be monitored in terms of the Ada packages. The package structure also facilitates tracing the software requirements. The package bodies, which contain the code that actually performs the operations, are invisible to all other packages except their own package body child library units. Hardware dependent changes are made whenever possible in the bodies. Since Ada is a general purpose language which can be employed for both procedural and object oriented programming, packages are not limited to just encapsulating a class. Packages can also encapsulate other logically related collections of: types, objects, operations, exceptions, etc.

Ada, as are all the members of the Pascal family of languages, was designed to be readable. Readability has been enhanced by permitting the replacement of positional notation in procedures and functions by an arrow notation where the formal parameter points to the actual parameter.

```
-- Specification
package Soda_Machine is
        type Coins is (Nickel, Dime, Quarter);
        type Sodas is (Coke, Pepsi, Seven_UP);
        type Options (Insufficient, Purchase, Illegal_Coin);
        function Correct_Amount (Coins) return Options;
        function Select_Soda return Sodas;
        procedure Dispense (Beverage: in Sodas; Option: in Options);
.............
end Soda_Machine;

-- Main Routine
with Soda_Machine;
procedure ABC_Soda_Machine is
begin  --Main procedure "Driver"
Soda_Machine.Dispense (Beverage => Soda_Machine.Select_Soda,
                       Option => Soda_Machine.Correct_Amount);
.......
end ABC_Soda_Machine;
```

Following the principles of information hiding, data types can be private and are only visible by the operations specified by the procedures and functions which employ them.

3.3. Regulatory and legal

The FDA regulations[2] require that:

"Subpart E-Purchasing Controls

820.50 Purchasing Controls

Each manufacturer shall establish and maintain procedures to ensure that all components, manufacturing materials, and finished devices that are manufactured, processed, labeled, or packaged by other persons or held by other persons under contract conform to specifications..."

"(a) Assessment of suppliers and contractors. Each manufacturer shall establish and maintain assessment criteria for suppliers and contractors that specify the requirements, including quality requirements that suppliers and contractors must meet..."

"(b) Purchasing forms. Each manufacturer shall establish and maintain purchasing forms that clearly describe or reference specifications, including quality requirements, for components, manufacturing materials, finished devices, or services ordered or contracted for. Purchasing forms shall include an agreement that the suppliers agree to notify the manufacturers of any changes in the product or service so that manufacturers may determine whether the change may affect the quality of a finished device..."

Since a compiler manufactures code, then presumably, it would <u>not</u> be exempt from the proposed rules. We believe that validated Ada compilers meet the FDA's requirements. The Alsys Ada compiler comes with a warranty which states, "Alsys warrants that the Compiler Programs and/or Executive Programs perform substantially as described in the Documentation and the Reference Manual for the Ada Programming Language."

There is no time limitation and the cited document, which is controlled by the US Government, can only be changed after a long and careful revision process. Only Ada compilers pass a Unites States government approved validation suite.

The use of a validated compiler and documented software engineering practices should be of some value as

evidence in a product negligence suite.

3.4. Real-Time

Real-time constructs based on tasking and protected records are included in the language. The Ada 95 Real-Time Systems Annex includes the capability of producing portable code which adheres to critical real-time requirements and follows specified scheduling policies, such as rate-monotonic. Expensive, proprietary run-time packages should no longer be required.

3.5. Established methodology and software engineering culture

The military, NASA and FAA have developed the technology to maximize both safety and productivity. Well engineered code is available from software repositories and on two CDROMS[13]. Since the Department of Defense has mandated the use of Ada for all new software including management information services, a large selection of software tools is available to assist in producing software documentation. Most importantly, from a management perspective, Ada is a product of the software engineering establishment. The culture of Ada practitioners considers it a professional obligation to attempt to achieve maximum reliability by producing code with zero defects, handlers for unexpected events, and employing risk reduction to minimize irreversible system missteps. This engineering approach to software development reduces loss of resources (i.e. money, time, life).

3.6. Hazard analysis

Ada includes a very powerful exception construct, which is employed to detect type incompatibilities, other errors, or user specified abnormal situations, and permit responding to these events. Many of the hazards can have a one to one correlation with an exception.

3.7. Testing

The white box testing that occurs during verification and validation includes checking the behavior of the system by entering a range of values for each of the variables entered by the user or that could be corrupted by the system. Since Ada compilers permit data types to have specific ranges and the initialization of variables when they are declared, coverage for a variable can be achieved with fewer cases than for a weakly typed languages, such as C or C++. The early detection of internal inconsistencies by the compiler expedites development and finds defects that might not otherwise be detected during testing. This significantly increases the likelihood of fielding a properly functioning system.

The presently published data indicates that Ada is still safer[1] and more cost effective than any third generation language now known[14].

3.8. Ada vs. C & C++

The Systems and Interface Annexes now provide all of the low level operations of C except the direct use of registers. However the Systems Programming Annex states that, “The implementation shall support machine code insertions or intrinsic subprograms (or both), and all machine operations shall be accessible through one or the other.” The Package Interfaces supports shifts and rotates. The only significant perceived advantage of C++ was that it was “Object Oriented.” However as shown in the discussion below of object oriented programming Ada has now rendered C++ obsolete.

Although C++ has achieved significant popularity, at this time January 1995, its most vocal proponents appear to have either significant delivery or financial problems. Borland International has just had massive layoffs and removed its founder from being chief executive officer and Microsoft’s Windows 95 is very late. An experiment[15] at the State University of New York, Plattsburgh indicates that for a real-time project the students were able to complete it in Ada but not in C.

Fast single pass compilers please programmers. In this regard, Ada will probably always loose. Ada compilers are written to please the customer for the software, who very often is the United States Defense Department. Ada compilers are slow, since they can have 8 passes and even more when generics are involved. However, this permits them to very efficiently eliminate dead code and very often to produce very efficient optimized code. A description of the inner workings of an Ada compiler has recently been published[16].

It should be noted, that general comparisons between languages are fraught with difficulties because the

quality of the compiler technology and libraries is a language independent variable. For instance, some vendors who make compilers for multiple languages employ common libraries for numerics, code generation, and other componants. One very good test of the quality of a compiler, is whether it is self hosted. Since Ada compilers are large programs, the capacity to compile itself indicates significant functionality.

Tartan Corp[17], which has great expertise in optimizing technology, sells both an Ada and a C++ compiler; their tests demonstrate that their Ada is 10% faster than their C/C++ in real-time. The explanation of Mr. John Stare of Tartan is, “The Ada language provides a better environment for optimization technology to take advantage of a target architecture. A detailed description of how the Tartan Ada compiler performs its optimizations and takes advantage of the TI C30 DSP has been published by Birus and Syiek[18]. As an example of the quality of Ada code, they describe the Army's Wide Area Mine project[18], which employs the C30 for target identification and processing, signal processing of acoustic signals, and control functions. The Ada was faster than the C implementation for mission-critical portions of the code that must execute in less than 500 microseconds. The Ada code met stringent memory requirements. It was small enough to leave 50% reserve capacity in memory and made optimal use of a mix of memory that included types with differing wait states.

Ada outperformed C on a complex system of multiple 'C40 processors for Itek Optical Systems[19], a division of Litton Industries. What was important to Itek was latency--the delay between receipt of incoming data and output of its result. Ada beat C in both of the cases studied. The processing of two algorithms was compared. The first and second processed new data respectively 500 and 6,000 times a second. The results were that for the 500 and 6,000 Hz cases the percentage that the Ada was of the C processing time was respectively 61.2 and 97.4%. Eighty-four percent of the improvement of Ada versus C was ascribed to compiler optimization and the rest to the fast math functions.

P.K. Lawlis and T.W. Elam have reported[20] an instance of Ada doing better than assembly on a TI DSP. “With only minor source code variations, one version of the compiled code was much smaller while executing at approximately the same speed, and a second version was approximately the same size but much faster than the corresponding assembly code.” The authors explanation was that a compiler in a short period of time can iterate until there is no further change through a large number of interacting optimizing techniques. Human iteration through various assembly code optimizations is much slower and not cost effective.

However, Ada speed tests with run-time range and other checking turned off are only useful for convincing skeptics. For medical devices, the real comparison should be with all checks on and the equivalent checks coded in the other language. As Ada compilers have matured, the technology to optimize away the majority of the run-time checks, which are redundant has matured. For instance, the compile-time consistency checks do not have to be repeated at run-time. Thus, the probability of another language winning a competition with run-checking on is very small.

3.9. Object oriented programming:

3.9.1. Types and objects:

Objects in Ada are always of a specific type. The selection of types is very rich and the construction of user defined types is very flexible, precise, and strongly encouraged. The Pascal family of languages has a concept of range, which is absent from the C family and other languages. This is consistent with standard engineering practice, where objects are assigned dimensions and tolerances. All methods (procedures and functions) are strictly defined in terms of the types upon which they operate. In Ada 95 class wide operations are permitted, see below.

3.9.2. Classes

It was not as critical to build a class hierarchy in Ada 83 as C++ because the variant record structure that Ada inherited from Pascal provided similar functionality. The Ada 95 tagged record combined with the child library units provides improved cohesion and permits new data types to be added. The tagged records are extensible, since the variant part of the Ada 83 variant record has been separated from the constant part. Instead of employing a case statement to select the correct path, each variant can now be separately located, usually in its own package. These separate packages are usually child packages of the parent. They are compiled after the parent, yet maintain visibility to the parent package’s contents. These child packages inherit but can override all the

methods (procedures and functions) including abstract (place holder ones) that operate on the parent tagged record. The naming of the child packages conforms to a hierarchical tree structure, similar to a DOS path, but employs a period as the separator. This naming system will significantly facilitate tracing the inheritance tree of object based code.

According to Taft, the Ada 95 (Ada 9X) language Architect,[21]: a class in Ada 95 is a "set of types consisting of a type and all of its derivatives, direct and indirect."

> "The distinction Ada 9X makes between types and classes (= set of types) carries over into the semantic model, and allows some interesting capabilities not present in C++. In particular, in Ada 9X one can declare a "class-wide" object initialized by copy from a "class-wide" formal parameter, with the new object carrying over the underlying type of the actual parameter. For example:

```
procedure Print_In_Bold (X: T'Class) is
   -- Copy X, make it bold face, and then print it.
   Copy_Of_X: T'Class := X;
   begin
     Make_Bold (Copy_Of_X);
     Print (Copy_Of_X);
   end Print_In_Bold;
```

> "In C++, when you declare an object, you must specify the "exact" class of the object -- it cannot be determined by the underlying class of the initializing value. Implementing the above procedure in a general way in C++ would be slightly more tedious."
>
> "Similarly, in Ada 9X one can define an access type that designates only one specific type, or alternatively, one can define one that can designate objects of any type in a class (a "class-wide" access type)."
>
> For example:

```
type Fancy_Window_Ptr is access Fancy_Window;
 -- Only points at Fancy Windows -- no derivatives allowed
type Any_Window_Ptr is access Window'Class;
 -- Points at Windows, and any derivatives thereof.
```

> "In C++, all pointers/references are "class-wide" in this sense; you can't restrict them to point at only one "specific" type. In other words, C++ makes the distinction between "specific" and "class-wide" based on pointer/reference versus object/value, whereas in Ada 9X, this distinction is explicit, and corresponds to the distinction between "type" (one specific type) and "class" (set of types)."
>
> "The Ada 9X approach we believe (hope;-) gives somewhat better control over static versus dynamic binding, and is less error prone since it is type-based, rather than being based on reference vs. value."

Testing and validation of code which includes run-time dispatching is sufficiently difficult, that this new addition to Ada should be avoided for any critical code in a medical device.

3.9.3. Multiple inheritance

Eckel in a recent article[22] describes the complexity and ambiguity of multiple inheritance in C++. For instance he states concerning persistent objects, "However, if your member function needs to know the true starting address of the object, multiple inheritance causes problems." Eckel also notes, "Multiple inheritance is a minor but advanced feature of C++ designed to solve problems that arise in special situations." These situations are when one is deriving from more than one inheritance tree. In Ada 95 lacks a specific construct for multiple inher-

itance, but provides similar functionality by the combination of single inheritance and the use of a generic (template).

3.9.4. Availablility of objects

There is also an upcoming release of the Booch Components for Ada that will be released under the GNU Library General Public License (LGPL). These can be included without cost or obligation in applications. Contact dweller@neosoft.com for more details.

Objective Interface Systems, Inc. (OIS), MITRE, and DISA have been working on a mapping from the Common Object Request Broker Architecture, CORBA, Interface Description Language to Ada 95. OC Systems, Rational, and Objective Interface Systems are planning on selling CORBA. According to one of the developers[23], "The CORBA IDL to Ada 95 mapping specifies a mapping, not a binding. This will put Ada 95 on equal footing with the C++ and Smalltalk products (except, of course, the Ada mapping is cleaner;)." The CORBA technology provides the possibility of portable code that will interoperate on both Microsoft and X Windows platforms.

3.10. Windows

For the present Paul Pukite writes[24], "It should come as no surprise that large, well-integrated, reliable, maintainable, and reusable Ada applications can be developed for Windows. The development process becomes more manageable and less obscure by using standard Ada constructs such as package specifications, generics, tasking, and exceptions." The Windows NT Ada compiler from Thompson Software Group[25], has excellent features, such as a GUI builder, symbolic debugger, and context sensitive editor. Ada tasks are implemented as NT treads and full bindings are provided to Win32. The same compiler and tools should work with Microsoft Windows 95, when it becomes available.

3.11. Mathematics

The Numerics Annex includes Complex Types, their functions including trigonometric operations, and formatted input and output of complex values. The Ada language includes direct representation of multidimensional arrays and a pragma (command to the compiler) Convention(Fortran, ...) which permits operations to be performed column-major order rather than row-major. This permits the use of libraries and algorithms that were optimized for Fortran numerics. Even, bit ordering to specify both big and little endian is provided.

3.12. Experience

The successful use of Ada for a medium range hematology instrument, the Coulter® ONYX[12], and a clinical chemistry[26] have been reported. In the case of the ONYX, the Ada effect was observed, "The software was completed before the hardware." The quality of the Alsys 386 DOS compiler was demonstrated by the functioning ONYX with its initial hardware, an Intel 386SX which employed a 16 bit bus and lacked a floating point processor.

The COBAS® INTEGRA, an integrated clinical analytical system, was also developed with Alsys Ada. However, the computer employed was a HP 9000 series 400 and 700 running under UNIX. Ten persons produced 220,000 lines of code in three years. The throughput is of up to 750 tests per hour. The system employs absorbance and fluorescence photometry, and ion selective electrodes. The system functionality comprises a user interface, order & result processing and calculation, data storage and retrieval, printing, real-time instrument control, scheduling of tests, event tracking, server interface to laboratory information systems (LIS), quality control, system configuration, system maintenance and system diagnostics.

According to Martin Burri, "Ada was initially chosen because it preserves our knowledge of programming languages like Pascal and Modula-2, it is more reliable than other languages because of the required validation of the compiler, and it proved after comparisons with C to provide significant advantages in terms of software engineering". He also stated, "use of Ada is a key asset" to achieve compliance with ISO 9001[3] and ISO 9000-3[4], which imply quality and certification for medical instruments. During development, Ada also demonstrated the advantages of its strong typing. "Let's say if you have compiled and linked successfully and something goes wrong, it is a 95% a matter of logic and/or design error but never something like a pointer mismatch" acknowledges Mr. Burri. Also, Ada has been very easy to learn and understand by the engineers because of their Pascal

and Modula-2 background.

3.13. Future

Because of its portability, built-in support for software engineering, and unique features; Ada would be an excellent choice for the language for the Human Genome project. Its principle sponsors, the US, other Defense Departments and the aerospace industry will be around for a very long time and have to maintain systems for many years. Ada can express arrays of three bit entities (Adenine, Guanine, Cytosine, Thymidine, Uracil, Unknown, Space). Since Ada is strongly typed, DNA will not equal RNA. Ada has very sophisticated array manipulation including the ability to start and stop anywhere in an array and to know the size of the array. The elimination of the check for nulls required in C, should result in a significant improvement in performance.

4. References

1. I. C. Pyle, Developing Safety Systems, A Guide Using Ada, Prentice Hall ISBN 0-13-204298-3, 1990.

2 Request for Comments on the Proposed Revisions to the Medical Devices; Current Good Manufacturing Practice (CGMP) Regulations; Proposed Rule, published in the Federal Register, 58, No. 224, pp. 61952-61986 (November 23, 1993) by the Department of Health and Human Services, Food and Drug Administration

3. ISO 9001 “Quality systems, model for quality assurance in design/development, production and installation”.

4. ISO 9000-3 “Guidelines on the application of ISO 9001 to the development, supply and maintenance of software”

5. MIL-STD-498, Software Development and Documentation, US Department of Defense, 1994.

6. “Ada 9X Reference Manual The Language, The Standard Libraries,” Ada 9X Mapping/Revision Team Intermetrics, Inc. 733 Concord Ave. Cambridge, MS. 02138

7. S. B. Leif, S. H. Leif (Aha), and R. C. Leif; “Setting Up a Pre-production Quality Management Process in the Medical Device Industry”. in Software Quality Management II Vol. 1: Managing Quality Systems Ed. M. Ross, C. A. Brebbia, G. Staples, J. Stapleton, Computational Mechanics Publications, Southampton, Boston pp. 63-78 (1994).

8. B. W. Boehm, “A Spiral Model of Software Development and Enhancement,” *Computer,* May, p.61, 1988.

9. Mil-Std-2167A, Department Of Defense, Washington, D.C. 20301, Defense System Software Development

10. S. B. Leif and R. C. Leif; “Producing Quality Software According to Medical Regulations for Devices”. Computer Based Medical Systems, Proceedings of the Fifth Annual IEEE Symposium 265-272 (1992).

11. K. M. Hines, “An Object-Oriented System Design For A Satellite Communication System,” Tri-Ada ‘94, 494-498, 1994.

12. R. C. Leif, J. Sara, I. Burgess, M. Kelly, S. B. Leif, and T. Daly, “The Development of Software in the Ada Language for a Mid-Range Hematology Analyzer”. Tri-Ada ‘93 340-346, 1993.

13. Ada CDROM November 1994. Organized by R. Conn, Published by Walnut Creek CDROM, 1547 Palos Verdes Mall, Suite 260, Walnut Creek CA 94596, E-mail info@cdrom.com

14. Mosemann, L. K. “New is Good--Is Ada Too Old”, CrossTalk, The Journal of Defense Software Engineering, 40 8, 1993.

15. J. McCormick, Computer Science Department, Stare University of New York, Plattsburgh, NY 12901; mccormjw@splava.cc.plattsburgh.edu; ftp ajpo.sei.cmu.edu in directory: /public/ada9x, document: 9x_cplus.hlp

16. E. Schonberg and B. Banner, “The GNAT Project: A GNU-Ada 9X Compiler”, TRI-Ada ‘94 Proceedings, C. Engle, Jr. Editor, Association for Computing Machinery, Inc. New York, NY, 48-57, 1994.

17. Tartan Corp. 300 Oxford Drive, Monroeville, PA 15146. Tel. 412.856.3600

18. T. Birus and D. Syiek, “Optimizing Ada Compilers for DSP Avionics Applications”, DSP Applications 2, Number 4, 21-32, 1993.

19. J. Stare, “Growing Complexity in Military DSP Applications Requires High-Level Language Development Systems”, DSP Applications 2, Number 10, 9-17, 1993.

20. P.K. Lawlis and T.W. Elam, “Ada Outperforms Assembly: A Case Study.” Proceedings of TRI-Ada ‘92, 334-337, 1992.

21. T. Taft, Section 5.1 in “DRAFT ADA PROGRAMMER'S FREQUENTLY ASKED QUESTIONS (FAQ), Compiled by Magnus Kempe, 4 January 1995.” (URL http://lglwww.epfl.ch/Ada/FAQ/programming.html)

22. B. Eckel, “Multiple Inheritance”, Embedded Systems Programming 7 Number 12, 54-71, 1994.

23. Bill Beckwith, Objective Interface Systems, Inc. Bill.Beckwith@ois.com

24. Pukite, Paul. “Ada for Windows,” Software Development Magazine, February 1995, pp. 59-68.] publisher, Miller-Freeman (Phone: 800/829-5537).

25. Thompson Software Products (formerly Alsys) 10251 Vista Sorrento Pkwy, Suite 300, San Diego, California 92121-2706. Tel. 619.457.2700

26. Form U125-0394 MEDANAL.TXT, Ada Information Clearinghouse, 1-800-AdaIC-11 (232-4211), 703/685-1477, Ada Used to Develop Medical Analytical Systems; The developer (Tegimenta). For further information, please contact: Ann Tr|b Alsys GMBH & Co.KG Kleinoberfeld 7 D-76135 Karlsruhe Germany Tel: + 49 721 986 530 Fax: + 49 49 721 986 5398

End of Cited References

28. S. J. Andriole, “Rapid Application Protyping, The Storyboard Approach to User Requirements Analysis, Second Edition”, QED Technical Publishing Group, Wellesley, MA, 1992

29.

30. Advanced Systems, Nov 94”. “C++: The COBOL of the 90's”, under “The Nightmare of C++”.

31. ZiffWire via Fulfillment by INDIVIDUAL, Inc. > DATE: December 13, 1994 > INDEX: [1] > ORDER NO: 404094# >

32. Embedded Systems Programming, December 1993, “Using Ada with Embedded DSPs,” by Erich Pfeiffer and Jason Disch.

33. DSP Applications, April 1993, “Optimizing Ada Compilers for DSP Avionics Applications,” by Timothy Birus and David Syiek.

34... “A Comparison of the OO features of Ada9x and C++” in Springer Lecture

Notes in CS: “Ada Europe 93" pp.125-141 (short paper, good reading,

enlightens idioms)

35 J. McCormick, ftp ajpo.sei.cmu.edu in directory: /public/ada9x, document: 9x_cplus.hlp

State University of New York, Plattsburgh, (Ref. [30] > though, “Professor John McCormick has assigned the same project to each of his classes for about nine years, but switched languages at mid-decade. Working in teams of three or four, McCormick's real-time-programming students must write 15,000 lines of code to control a system that would need about 150 switches to operate using hardware alone. In the five years students used C, no team completed the project -- even when more than half of the code was provided. With Ada, however, half of the teams completed the project before any support code had even been written. With some support code now provided, three out of four teams finish the project. Specific factors in this improvement, according to McCormick, include both syntax and semantics. Ada leaves less room for single-keystroke errors, such as the common C error of using = (assignment) instead of == (comparison); its type-abstraction facilities reduce the need for error- prone pointer manipulation; and its modular facilities improve teams' coordination of effort.”

Check out Nov 94 issue of “Advanced Systems”. They have an article titled: “C++: The COBOL of the 90's”, under “The Nightmare of C++”.

Quotes like:

“...other books can tell you how using any of dozens of OO languages

can make programmers more productive, make code more robust, and

reduce maintenance costs. Don't expect to see any of these advantages

in C++.

That's because C++ misses the point of what being OO was all about. Instead

of simplifying things, C++ sets a new world record for complexity.

Like Unix, C++ was never designed, it 'mutated' as one goofy mistake after

another became obvious.There is no grammar specifying the

language (something practically all other languages have,) so you

can't even tell when a given line of code is legitimate or not.”

... and more goodies.