

OBJECT-ORIENTED TECHNOLOGY (OOT) IN CIVIL AVIATION PROJECTS: *CERTIFICATION CONCERNS (1999)*

Leanna K. Rierson, Federal Aviation Administration, Washington, D.C.

Abstract

Object-Oriented Technology (OOT) has been extensively utilized throughout the non-safety rated software development community (e.g., Windows 98 and Internet software). Although the adoption of OOT has been limited in the airborne civil aviation community, OOT is being considered by an increasing number of manufacturers of airborne software. Application of RTCA DO-178B/EUROCAE ED-12B ("Software Considerations in Airborne systems and Equipment Certification") is the primary means of approving software within these airborne systems. However, DO-178B/ED-12B was developed prior to the widespread use of OOT. The application of DO-178B/ED-12B to systems developed using OOT is not well understood in the industry. This paper will provide an introduction to the issues surrounding the development of aviation software using OOT within the context of DO-178B/ED-12B assessments. This introduction will demonstrate that DO-178B/ED-12B is compatible with OOT but that there are significant issues that need to be addressed. Other papers will be developed to address these issues in more depth.

Introduction

Object-Oriented Technology (OOT) is seen by many in the mainstream software community as the "silver bullet" that will take us into the new millennium of software development. OOT is appealing because of the number of available tools, the emphasis on reuse, and the appeal to software designers. It is touted as a technology that saves money, improves quality, and saves time.

However, to date, few airborne computer systems in civil aviation have implemented OOT. Safety-critical designers tend to use proven technologies and, as a result lag, a few years behind the mainstream designers of non-safety software. Since OOT has proven to be cost-effective and technically sound for many projects, manufacturers of safety-critical systems are now considering its use.

There are some concerns when using OOT that must be carefully considered. This paper will provide an overview of OOT, an overview of RTCA DO-178B/EUROCAE ED-12B ("Software Considerations in Airborne Systems and Equipment Certification"), and some of the concerns of using OOT in airborne aviation software. This is merely the beginning of a more in-depth study and will likely be followed by other papers.

Overview of OOT

OOT is a software development technique that is centered around "objects." IEEE refers to OOT as "a software development technique in which a system or component is expressed in terms of objects and connections between those objects" [4]. An object can be compared to a "black box" at the software level – it sends and receives messages. The object contains both code (functions) and data (structures). The user does not have insight into the internal details of the object, thus giving it the comparison to a black box. An object can model real world entities, such as a sensor or hardware controller, as separate software components with defined behaviors.

A major concept in OOT is the "class." Grady Booch, a champion in OOT methodology, defines a "class" as "a set of objects that share a common structure and a common behavior" [2]. A class contains the attributes and operations that are required to describe the characteristics and behavior of a real world entity. Figure 1 illustrates a representation of a class definition for an object.

Principles of OOT

There are seven principles that form the foundation for OOT: abstraction, encapsulation, modularity, hierarchy, typing, concurrency, and persistence [2]. Not all of these principles are unique to OOT, but OOT is the only development methodology that embodies all seven as a consistent model.

Abstraction, modularity, concurrency, and persistence are principles that are commonly used in other development methodologies. However, encapsulation (using a technique called *information hiding*), hierarchy (using a technique called *inheritance*), and typing (using a concept called *polymorphism*) are relatively unique to OOT. Each of the seven principles is described below.

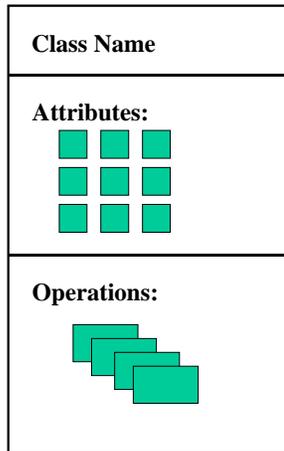


FIGURE 1 – Object-Oriented Class Representation

Abstraction is one of the fundamental ways that complexity is addressed in software development. “An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer” [2].

Encapsulation is the process of hiding the design details in the object implementation. Encapsulation can be described as “the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse” [11]. Encapsulation is generally achieved through *information hiding*, which is the process of hiding the aspects of an object that are not essential for the user to see. Typically, both the structure and the implementation methods of the object are hidden [2].

Modularity is the process of partitioning a program into logically separated and defined components that possess defined interactions and limited access to data. Booch writes that modularity is

a “property of a system that has been decomposed into a set of cohesive and loosely coupled modules” [2].

Hierarchy is simply the ordering of abstractions. Examples of hierarchy are *single inheritance* and *multiple inheritance*. In OOT, when a sub-class is created, this new class “inherits” all of the existing attributes and operations of the original class, called the “parent” or “superclass” [8]. Inheritance is a relationship between classes where one class is the “parent” (also called “base,” “superclass,” or “ancestor”) class of another [6]. One author puts it this way, “Inheritance is a relationship among classes where a child class can share the structure and operations of a parent class and adapt it for its own use” [5].

Inheritance is one of the key differences between OOT and conventional software development. There are two types of inheritance: *single inheritance* and *multiple inheritance*. In *single inheritance*, the sub-class inherits the attributes and operations from a single superclass. In *multiple inheritance*, the sub-class inherits some attributes from one class and others from another class. *Multiple inheritance* is controversial, because it complicates the class hierarchy and configuration control [9].

Typing is a principle that is used in OOT that has many definitions. Booch presents one of the most clear and concise definitions by stating, “Typing is the enforcement of the class of an object, such that objects of different types may not be interchanged, or at the most, they may be interchanged only in very restricted ways” [2]. Examples of OOT typing are strong typing, weak typing, static typing, and dynamic typing. Each OOT programming language varies in its implementation of typing.

Another OOT concept closely related to typing is *polymorphism*. *Polymorphism* comes from the Greek meaning “many forms.” It allows one name to be used for two or more related but different purposes [11]. It is the ability of an object to assume or become many different forms of object. *Polymorphism* specifies slightly different or additional structure or behavior for an object, when assuming or becoming an object [6]. This allows different underlying implementations for the same command. For example, assume there exists a vehicle class that includes a steer-left command. If a boat object was created from the vehicle class, the steer-left command would be implemented by a push to the right on a tiller. However, if a car object was created from the same class, it might use a counter-clockwise rotation to achieve the same command.

Concurrency is the process of carrying out several events simultaneously.

Persistence is “the property of an object through which its existence transcends time (i.e., the object continues to exist after its creator ceases to exist) and/or space (i.e., the object’s locations moves from the address space in which it was created)” [2].

OOT Methodology

Everyone seems to have a slightly different perspective of what OOT actually entails. OOT can be described in four phases: Object-Oriented Analysis (OOA), Object-Oriented Design (OOD), Object-Oriented Programming (OOP), and Object-Oriented Verification/Test (OOV/T). The implementation of these phases is typically iterative or evolutionary. An overview of each phase will be addressed below.

OOA is the process of defining all classes that are relevant to solve the problem and the relationships and behavior associated with them [9]. A number of tasks occur to carry out the OOA as shown in Figure 2. The tasks are reapplied until the model is completed. As shown in Figure 2, use cases, class-responsibility-collaborator (CRC) models, object-relationship (OR) models, and object-behavior (OB) models are methods typically used to carry out the OOA. The use case is a method utilized to identify the user’s requirements. The CRC model is used to identify the class attributes, operations, and hierarchy. The OR model is used to illustrate the relationship between the numerous objects. And, the OB model is used to model the behavior of each object.

OOD transforms the OOA into a blueprint for software construction. Four layers of design are usually defined: subsystem layer, class and object layer, message layer, and responsibilities layer. The *subsystem design layer* represents each subsystem that enables software to achieve the requirements. The *class and object design layer* contains class hierarchies and object designs. The *message design layer* contains the internal and external interfaces to communicate between objects. The *responsibilities design layer* contains the algorithm design and data structures for attributes and operations of each object.

OOP is the coding phase of the design project, using an object-oriented (OO) language. There are dozens of OO languages. Three of the most well known are C++, Smalltalk, and Java. C++ and Java are of particular interest for designers of embedded software. Java’s platform independence and C++’s tool support make these two languages very appealing to the developers of airborne systems.

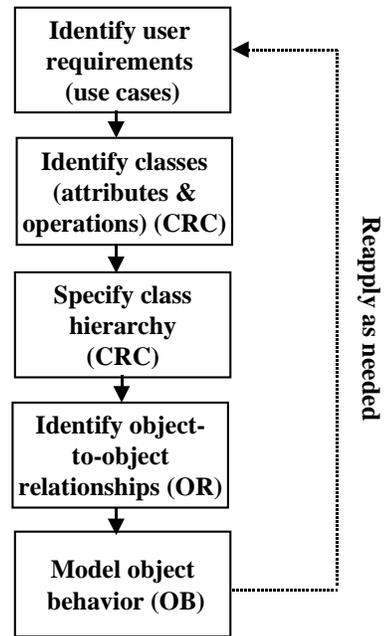


FIGURE 2 – OOA Tasks

There have been a few C++ applications on aviation products—mostly in less safety-critical systems. One area of great concern for these programs was the use of certain built-in C++ functions. For example, the use of “new” and “delete” functions was prohibited because of non-deterministic behavior due to dynamic memory management functions. Assuring the lack of memory leakage was another issue that was addressed. The developers on these programs found that the use of C++ functions and built-in libraries required extensive verification and testing in order to understand their behavior.

In 1996 David Binkley of the National Institute of Standards and Technology (NIST) published a paper entitled “C++ in Safety Critical Systems.” The paper outlined guidelines for using C++ to create safe software. Adherence to these guidelines can lead to safer and more maintainable C++ programs. The paper also outlined a series of techniques and examples for creating safer C++ programs. The paper is available on the world wide web [1].

An article in Computer Design stated, “For embedded systems, a language generating a great deal of interest today is Embedded C++, a subset of ANSI C++ that offers certain advantages for real-time development.” Embedded C++ (EC++) omits some of the “problematic” features of ANSI C++. For example, multiple inheritance, virtual base classes,

run-time identification, templates, exceptions, and namespaces are deleted [13].

Brian Wichmann recently conducted a website discussion entitled “Moderated Discussion on C++ and Safety.” This discussion attracted worldwide input about C++ applicability to safety-critical projects. The discussion demonstrated how controversial the use of C++ is for safety-critical applications.

Wichmann began the discussion with this thought: *“Although the major problem with safety-critical software is getting the requirements correct, the impact of the language is significant... The main problems I see with C++ arise from its 'high-level' nature. For instance, it is hard to show that there is no storage leak or bound the storage requirements statically. Another problem is that in several cases, the order in which an execution is performed is not defined, making it effectively impossible to guarantee predictable execution”* [14].

The discussion to Wichmann’s question goes on for eight pages. Some of the more enlightening and relevant discussions are included below in order to illustrate the controversy of this subject. Peter Fenelon stated the following regarding sub-sets of C++: *“I’d be highly reluctant to see C++ in any safety-related or critical environment. By the time “unsafe” or “difficult” features are ruled out -- I’m referring particularly to exception handling, the use of templates and Standard Template Library (STL), multiple inheritance, and so on -- what’s left isn’t much more than ANSI C (a language I have far fewer quibbles with, if sensible guidelines are followed)...”* [14].

Another interesting comment by Bob Gorman endorsed the possibility of using C++ on safety-critical systems: *“It seems to me that many people here are evaluating C++ only as a sum of its features. Of course we can pick apart any language if we only focus its features or lack of them. However, it’s the real world implementation of the features that makes the application safe and robust...”* [14].

Jim Jaskol wrote: *“... The tools, knowledge base, and experience surrounding C/C++ gives it tremendous advantages in many areas over other languages--advantages that can translate into safer systems. C++ has too much of a following to be ignored...”* [14].

Because of its platform independence and widespread use by the mainstream software development community, Java has recently become a desirable language for developers of safety-critical systems. Java was originally developed for embedded

systems on cable television boxes. Since most safety-critical systems are embedded, Java has some potential. Some aviation companies are currently performing studies to determine the feasibility of using Java in safety-critical systems.

There are currently many concerns regarding the use of this language. Java is an extremely powerful language; however, it lacks robustness. There are new releases of Java every few months; this fast production of the language does not allow it to become robust. One software developer who uses Java for browsers claims that the browsers “crash” a lot. Frequent crashing may be tolerable for non-safety applications; however, it is not acceptable for a safety-critical system, such as an airplane or a nuclear power plant.

An article in Computer Design explored the use of Java in safety-critical systems. A positive aspect of the language is that the absence of pointers and automatic checking for common errors reduces the potential for memory errors. However, Java’s use of garbage collection is problematic, because it is a non-deterministic memory management technique [13].

Java has matured rapidly in the last two years. Despite a few problems, the power and potential of Java is impressive. With improvements and further investigation, it could become a language appropriate for use in safety-critical systems.

OOV/T is the process of detecting errors and verifying correctness of the OOA, OOD, and OOP. OOV/T includes reviews, analyses, and tests of the software design and implementation. OOV/T requires slightly different strategies and tactics than the traditional structured approach. The variance in the approach is driven by characteristics like inheritance, encapsulation, and polymorphism. Most developers use a “design for testability” approach to begin addressing any verification/test issues early in the program.

Overview of DO-178B/ED-12B

DO-178B/ED-12B is the guidance document that most civil aviation manufacturers use for certification approval of their airborne software. In order to assess how DO-178B/ED-12B applies to OOT, it is important to understand the background and basics of the document.

DO-178/ED-12 (no revision) was first developed by the international civil aviation community in 1982. It was revised in 1985 to add more detail. In 1992, DO-178B/ED-12B was completed and has become the software “standard” for

airborne software in civil aviation products. The DO-178/ED-12 document and all of its revisions were sponsored by RTCA and EUROCAE, with the involvement of aviation, software, and certification experts from across the world.

DO-178B/ED-12B focuses on the software aspects of system development. As part of the systems engineering task, a system safety assessment must be performed before DO-178B/ED-12B can be applied to the software development effort. A system safety assessment is a process to identify the hazards, failure conditions leading to these hazards, and the effects of mitigation strategies. The safety assessment task determines a software level based upon the contribution of the software to the potential failure conditions defined in the system safety assessment process. The five software levels, A to E, are summarized in Table 1 [10].

These software levels define differing degrees of rigor for the software development process. Annex A in DO-178B/ED-12B lists the objectives that must be met for each specific software level. These software levels define a number of desirable attributes for the software development and verification processes. The differences in rigor are determined by the number of objectives which need to be satisfied, whether a specific objective is satisfied with independence, and the formality of configuration control of the software data produced during development. For example, the number of objectives for each software level is listed below:

- Level A: 66 objectives
- Level B: 65 objectives
- Level C: 58 objectives
- Level D: 28 objectives
- Level E: 0 objectives

DO-178B/ED-12B is divided into development activities and integral processes. The development activities include planning, requirements, design, code, and integration. The integral processes include verification, configuration management, quality assurance, and certification liaison. The integral processes are overlaid on each of the development activities (i.e., verification, configuration management, quality assurance, and certification liaison are applied to each development activity).

Failure Condition Category	Description	SW Level
Catastrophic	Failure conditions which would prevent continued safe flight and landing of the aircraft.	A
Hazardous	Failure condition which would reduce the capability of the aircraft or the ability of the crew to cope with adverse operation conditions to the extent that there would be: <ol style="list-style-type: none"> (1) a large reduction in safety margins or functional capabilities, (2) physical distress or higher workload such that the flight crew could not be relied on to perform their tasks accurately or completely, or (3) adverse effects on occupants including serious or potential fatal injuries to a small number of occupants. 	B
Major	Failure conditions which would reduce the capability of the aircraft or the ability of the crew to cope with adverse operation conditions to the extent that there would be, for example, a significant reduction in safety margins or functional capabilities, as significant increase in crew workload or in conditions impairing crew efficiency, or discomfort to occupants, possibly including injuries.	C
Minor	Failure conditions which would not significantly reduce aircraft safety, and which would involve crew actions that are well within their capabilities.	D
No Effect	Failure conditions which do not affect the operational capability of the aircraft or increase crew workload.	E

Table 1 – DO-178B/ED-12B Software Levels

The objectives of DO-178B/ED-12B are listed in Annex A of the document and are organized around the development activities and integral processes previously described. There are ten tables in Annex A with objectives—the subject of each table is listed below:

- Table A-1: Software Planning Process
- Table A-2: Software Development Processes
- Table A-3: Verification of Outputs of Software Requirements Process
- Table A-4: Verification of Outputs of Software Design Process
- Table A-5: Verification of Outputs of Software Coding & Integration Processes
- Table A-6: Testing of Outputs of Integration Process
- Table A-7: Verification of Verification Process Results
- Table A-8: Software Configuration Management Process
- Table A-9: Software Quality Assurance Process

- Table A-10: Certification Liaison Process

Table A-4 objective 1 is used in Figure 3 to illustrate the Annex A table layout and structure. The first set of columns contains information about the DO-178B/ED-12B **objectives**: objective number, description, and reference to DO-178B/ED-12B paragraph where that objective is further detailed. The next set of columns with headers A, B, C, D show the applicability of that particular objective to the software level. For example, objective 1 is applicable for levels A, B, and C; however, it does not need to be satisfied for software level D. If the circle indicating applicability is filled in, then that objective must be satisfied with independence. The next series of columns describe the **outputs** produced as evidence that the objective is satisfied. The “Description” column lists where that data is found. The “Ref.” Column identifies the paragraph within Chapter 11 of DO-178B/ED-12B that details the attributes of that software data. The last 4 columns correlate the rigor of configuration management of the particular output with the associated software level. Control category 1 requires more configuration management activities than control category 2. For instance, control category 1 requires problem reporting and change control, where as control category 2 requires only change control.

	Objective		Applicability by SW Level				Output		Control Category by SW level			
	Description	Ref.	A	B	C	D	Description	Ref.	A	B	C	D
1	Low-level requirements comply with high-level requirements.	6.3.2a	I	I	m		Software Verification Results	11.14	2	2	2	

Figure 3 – Portion of Table A-4 in DO-178B

Assessment to DO-178B/ED-12B is performed through on-site reviews and/or desk-top (data) reviews by FAA personnel, Designated Engineering Representatives, and/or software developer’s team members. The assessment evaluates the data to determine if the objectives listed in Annex A of DO-178B/ED-12B are met. In June of 1998, the FAA released a job aid entitled, “Conducting Software Review Prior to Certification”. The job aid outlines a process for assuring compliance to the objectives of DO-178B/ED-12B. The job aid is available electronically and is designed to be tailored to meet the specific needs of the evaluator or project.

This section has provided a very high level overview of DO-178B/ED-12B. More information may be obtained by reading DO-178B/ED-12B itself,

by participating in related RTCA and EUROCAE activities, and by reviewing the FAA job aid.

Concerns in Use of OOT in Airborne Software

For most software projects seeking FAA approval, the objectives of DO-178B/ED-12B should be satisfied, as appropriate for the software level. This section will look at some of the issues to be addressed by a software development team using OOT in airborne civil aviation software in order to meet the objectives of DO-178B/ED-12B. This should not be considered a comprehensive study of the issues – there are likely additional issue to be addressed, depending on the specific project details (for example, each OO language and/or compiler may have different certification issues).

Planning. The OOT software development process should be carefully planned and documented. In particular, the Plan for Software Aspects of Certification document should address any special certification issues in order to get the certification authority’s “buy-in.” Additionally, the development standards should address any special limitations for the development team to consider (e.g., no multiple inheritance, etc.).

Traceability. DO-178B/ED-12B requires traceability from requirements to design to code to test cases/results. When inheritance is used in the design, special care must be taken to maintain traceability. This is particularly a concern, if multiple inheritance is used. Overall, multiple inheritance is a concern to certification authorities. If used, it should be very carefully applied and addressed in the development standards for the project.

Traceability is made more difficult because there is often a lack of OO methods or tools for the full software lifecycle. For example, tools/methods often cover OOA or OOD but not both. New tools are beginning to address this gap [3].

Target Compatibility. A number of DO-178B/ED-12B objectives address the topic of target compatibility. Using classes, instantiation, and automatic memory management typically implies the use of dynamic memory allocation. In typical implementations, dynamic memory algorithms require periodic reorganization of the memory to reduce the inevitable fragmentation. This leads to indeterminate execution profiles. As an alternative to the typical implementation provided by most OO languages, the developer might consider the feasibility of designing a deterministic memory allocation subsystem. Another

approach which might be feasible, is to pre-allocate objects during program initialization and avoid creating or deleting them after that. Dynamic memory allocation must be verified in terms of both space (available memory) and execution time in order to determine compatibility with the target.

Structural Coverage. DO-178B/ED-12B has three forms of structural coverage, which are applicable depending on the software level: statement coverage (Levels A, B, & C); decision coverage (Levels A & B); and modified condition/decision coverage (MC/DC) (Level A only). The use of inheritance and polymorphism might cause difficulties in obtaining structural coverage, particularly decision coverage and MC/DC. Source to object code correspondence will vary between compilers for inheritance and polymorphism.

Dead/Deactivated Code. DO-178B/ED-12B defines dead and deactivated code as follows:

“Dead code - Executable object code (or data) which, as a result of a design error cannot be executed (code) or used (data) in a operational configuration of the target computer environment and is not traceable to a system or software requirement. An exception is embedded identifiers” [10].

“Deactivated code - Executable object code (or data) which by design is either (a) not intended to be executed (code) or used (data), for example, a part of a previously developed software component, or (b) is only executed (code) or used (data) in certain configurations of the target computer environment, for example, code that is enabled by a hardware pin selection or software programmed options” [10].

DO-178B/ED-12B basically requires any dead code to be removed and deactivated code to be analyzed to prove that it is not dead.

When superclass methods are replaced by sub-class methods (i.e., overridden methods), there is a possibility that dead or deactivated code could be introduced. Structural coverage analysis is intended to address the dead/deactivated code. However, any such occurrences would need to be addressed.

Verification/Testing. Test coverage of high-level and low-level OO requirements will likely require different testing strategies and tactics than the traditional structured approach. The characteristics of inheritance, encapsulation, and polymorphism drive the need for the different strategies and tactics. Most developers are using a “design for testability” approach to begin addressing any test issues early in the program.

Overuse of Inheritance. Overuse of inheritance, particularly multiple inheritance, can lead

to unintended connections among classes [3]. This could lead to difficulty in meeting the DO-178B/ED-12B objective of data and control coupling.

Ambiguity. Inheritance, polymorphism, and operator overloading through dynamic or run-time linkage can lead to ambiguity. Polymorphic and overloaded functions may make tracing and verifying the code difficult [3]. Since DO-178B/ED-12B requires that the source code be verifiable, attention should be paid to such issues.

Coding Issues. Some OO languages have “features” that could make it extremely difficult or impossible to satisfy the objectives of DO-178B/ED-12B. In many cases, a well-defined sub-set of the language may be identified and documented in the coding standards that will allow compliance to objectives for a given software level. As an example, ANSI C++ has some “features” that might make meeting the objectives of DO-178B/ED-12B impossible. These obstacles might be addressed by including the following restrictions in the coding standards:

- ◆ Minimize dynamic binding
- ◆ Minimize operator overloading
- ◆ Minimize control flow complexity
- ◆ Use “new” only at initialization
- ◆ Avoid using “delete”
- ◆ Avoid use of exception handling
- ◆ Avoid multiple inheritance
- ◆ Avoid type-cast pointers

Library Dependence. The dependence on libraries is a concern for safety-critical systems—it is often unclear as to what is happening in the object libraries. Libraries may not have been developed with safety-critical applications in mind and may not have the integrity required for such applications. Use of libraries must be carefully considered and verified for proper functionality.

Conclusions

An article in Computer Design, entitled “Building Tomorrow's Embedded Software,” stated, “Size, complexity, and time-to-market issues are causing fundamental changes in how embedded software is written. While there are solutions to borrow from desktop development, they have to be chosen judiciously” [12]. Such is the case with safety-critical systems’ development. The use of OOT in aviation systems is being considered by many developers of airborne software. The jury is still out with respect to its use. There are advantages and

disadvantages for software all development methods – OOT is no exception.

Developers should carefully weigh their program needs with the benefits and risks of OOT. There are a number of potential certification concerns, as discussed in this paper. The author intends to further investigate each of the certification concerns in more depth in order determine potential risk mitigation strategies.

References

- [1] Binkley, David. “C++ in Safety Critical Systems,” National Institute of Standards and Technology, February 29, 1996. Web-site: http://hissa.ncsl.nist.gov/sw_develop/ir5769/ir579.1.html.
- [2] Booch, Grady. Object-Oriented Analysis and Design. Addison-Wesley, 2nd edition, 1994.
- [3] Cuthill, Barbara. “Applicability of Object-Oriented Design Methods and C++ to Safety-Critical Systems” from Proceedings of the Digital Systems Reliability and Safety Workshop (1993).
- [4] “Glossary of Software Engineering Terminology.” ANSI/IEEE Standard, 1983.
- [5] Gomaa, Hassan. Software Design Methods for Concurrent and Real-time Systems. Addison-Wesley, 1993.
- [6] Hathaway, Bob. “Frequently Asked Questions on Object-Oriented.” Web-site: <http://www.cs.cmu.edu/Groups/AI/html/faqs/lang/ooop/faq-doc-0.html>.
- [7] Meyer, Bertrand. Object-Oriented Software Construction. Prentice Hall, 2nd edition, 1997.
- [8] Montlick, Terry. “What is Object-Oriented Software?” Web-site: <http://www.soft-design.com/softinfo/objects.html>
- [9] Pressman, Roger. Software Engineering: A Practitioner’s Approach. McGraw Hill, 4th edition, 1997.
- [10] RTCA, document DO-178B/ED-12B, “Software Considerations in Airborne Systems and Equipment Certification”, dated December 1, 1992.
- [11] Schildt, Herbert. Teach Yourself C++. McGraw Hill, 1998.
- [12] Varhol, Peter. “Building Tomorrow’s Embedded Software,” Computer Design, April, 1998. Web-site: <http://www.computer-design.com/Editorial/1998/04/Embedded/498DESRP.HTM>.
- [13] Varhol, Peter, “Software Design for Life-Critical Systems”, Computer Design, July, 1998. Web-site: <http://www.computer-design.com/Editorial/1998/07/Wang/798JULFEAT.htm>.

[14] Wichmann, Brian. “Moderated Discussion on C++ and Safety,” March 20, 1998. Web-site: <http://www.cs.york.ac.uk/~jdm/cplussafety.html>.

About the Author

Leanna K. Rierson is a Software Program Manager and Technical Specialist for the FAA Aircraft Certification Service’s Avionics branch in Washington, D.C. She has previous experience as a software engineer at NCR, Cessna Aircraft Company, and the Wichita Aircraft Certification Office. Leanna graduated sume cum laude, with a bachelor’s degree in electrical engineering. She is currently completing a Master’s in software engineering and plans to continue work toward a PhD. Leanna is the leader for FAA’s Software Grand Design program, Streamlining Software Aspects of Certification program, Flight Critical Digital Systems Research effort, and Technical ReUsable Software Team. She is also the chair of the international Certification Authorities Software Team and is the editorial leader of RTCA’s Special Committee #190.

Acknowledgements

Much thanks to Michael DeWalt of Certification Services Incorporated and Uma Ferrell of MITRE for their review and input on this paper.