DO-332/ED-217 Using Modern Software Practice in Airborne Systems

Michael R. Elliott

Abstract. In civil airspace, the methods needed to produce software compliant with airworthiness have been considered overly burdensome, expensive and process-heavy. This view is based largely on experience with DO-178B/ED12B[5], which was the standard for software development in civil airspace beginning in 1992. Finalized in 2011, DO-178C/ED12C[6] was created to address these concerns and to apply more modern software practice to issues of software production and verification. Of special interest is DO-332/ED-217[7], the *Object–oriented Technology and Related Techniques* supplement to DO-178C/ED-12C, which addresses modern software practice and how it can be utilized in the production of software in systems that need airworthiness certification. This article addresses the traditional view of software development in this area and the significant cost and schedule reduction which can be realized by using the guidance and recommendations described in DO-332/ED-217 over those practices based on DO-178B/ED-12B.

I. Introduction

The formal standard *Software Considerations in Airborne Systems and Equipment Certification*, known in the industry as DO-178B or more recently DO-178C, is the means by which certification authorities, such as the FAA[1] and EASA[2], determine whether aircraft and engines containing software as part of their operational capability can be granted airworthiness certification for operation in civil airspace. As such, it is required reading for thousands of engineers worldwide who produce software for aircraft and aircraft engines. It specifies the means by which such software is produced and verified so that airworthiness certification can be granted.

Military aircraft, such as the USAF C-17, have made use of DO-178B for guidance in airworthiness even though not formally required to do so. An effort began in late 2004 to produce a successor document to DO-178B/ED-12B to be known as DO-178C/ED-12C. Special Committee 205 (SC-205) of the RTCA[3] and Working Group 71 (WG-71) of EUROCAE[4] were formed to address the perceived shortcomings of the existing standard from a viewpoint more attuned to modern software practice.

The mission of this subgroup was to address the needs of software practitioners in creating object-oriented software for airborne systems, which was a practice widely viewed as prohibitively difficult under the existing standard. The core document of DO178B/ED-12B was changed to help facilitate this effort. A supplement to the emerging DO-178C/ED-12C was produced, providing additional (and sometimes alternative) objectives, guidance and recommendations to aid the practitioner in airborne software production and the certification authorities in approval processes.

A. Object-Oriented Technology (OOT)

To date, few airborne computer systems in civil aviation have been implemented using OOT. Although OOT is intended to promote productivity, increase software reusability, and improve quality, uncertainty about how to comply with certification requirements has been a key obstacle to OOT use in airborne systems. (OOTiA[1], 2004.)

The importance of object-oriented technology was recognized as a key element to be addressed. One of the three subgroups formed to address software development practice was Subgroup 5 – Object-oriented and Related Technologies.

B. Object-oriented Technology Supplement

This supplement, as IP[5] 500, was formally approved at the SC-205/WG-71 plenary session in Paris, France, on Oct. 29, 2009. It is singularly appropriate that the day that the last-ever OOPSLA[6] ended – the day that object-oriented programming was considered so mainstream that it was no longer worthy of a special conference – is the day that marked the first formal acceptance of the use of object-oriented programming in the international standards for safety-critical airborne software.

II. Background

Initially, software was viewed as a way to inexpensively extend the versatility of analog avionics. However, software in the system did not fit easily into the safety and reliability analysis based on mean time between failure and other service history based techniques.

A. DO-178

This initial effort at a standard for software development in airborne systems was a set of best practices that was created to provide a basis for communication between applicants and certification authorities. It required applicants to meet "the intent" of DO-178 without giving specific objectives or significant guidance on how to do so. It did, however, introduce a three-tiered system of software criticality – critical, essential and nonessential – and set the level of verification to reflect the criticality level. Additionally, it provided a link between software verification and FAA documents, such as Federal Aviation Regulations and Technical Standard Orders.

B. DO-178A

After the initial experience with certification using DO-178, there was a consensus that it needed revision. SC-152 of RTCA created DO-178A in 1985, and it turned out to be quite different from DO-178. It introduced rigorous requirements for software processes (based on the waterfall method), software production, and process documentation and history. Applicants and certification authorities

frequently misinterpreted certification artifacts, sometimes causing entire software development efforts to be abandoned. In general, knowledge of why the certification requirements existed and their purpose failed to be understood or appreciated. [2]

C. DO-178B

The avionics industry became more and more software-oriented during the time DO-178A was in use. Many new companies entered the field and produced equipment subject to certification. Lack of experience, documentation, and understanding of the reasons for satisfying DO-178A brought about a need for an improved standard. In 1992, this became DO-178B, which was developed in cooperation with EUROCAE as ED12B by SC-167 and WG-12. This updated document made many fundamental changes to its predecessor. Salient among these was the introduction of software criticality levels A through E, which replaced the "critical, essential and non-essential" designations that were used previously. It placed a strong emphasis on requirements-based testing, which was seen as a more effective verification strategy than traditional white-box testing. It also required that these tests and their related artifacts be made available to certification authorities for use as part of their approval process.

D. OOTiA

During the eight years following the release of DO-178B/ED12B and its adoption by the industry, some people expressed concern that more modern software practices were difficult to employ using that standard. In 2000, the FAA responded by contacting the representatives of several key companies, including Boeing, BF Goodrich and others, to produce an analysis of how object-oriented software procedures could be adapted to the needs of airworthiness certification.

This process was later available to the industry in general, and workshops were held in order to produce position papers that would hopefully evolve into a best practices guide, become an FAA Advisory Circular, or be rolled into the not-yet-begun DO-178C effort. The FAA and NASA[7] held meetings that eventually resulted in the FAA publishing the four-volume *Handbook for Object-Oriented Technology in Aviation* (OOTiA). This document was never intended to contain objectives or guidance for practitioners and certification authorities. It was only meant to contain a set of suggestions for best practices and warnings about problematic situations.

By 2005, the FAA had decided that it would no longer maintain sponsorship of OOTiA or facilitate any updates or corrections to it. SC-205 of the RTCA was under consideration as a means to upgrade DO-178B [1], and it was considered best to turn OOTiA over to the nascent SC-205 to use as input for the creation of an object-oriented supplement to the new standard.

E. Rationale

The views of a number of stakeholders, including certification authorities, airframe manufacturers, and equipment suppliers, were taken into account in the creation of DO178B/ED-12B. A basic tenet of this document was that it should be written, as much as possible, to be requirements-oriented; that is, the document should be about objectives rather than processes. This was fundamentally meant to minimize the impact of technological evolution, as long diatribes such as the best use of blank COMMON blocks in FORTRAN were considered inappropriate in the long term. This brought about the philosophy of creating the document in terms of objectives, guidance, and guidelines so that applicants could use it in creating airborne software and certification authorities could use it to judge software's suitability in an airworthiness determination.

A large part of the document is concerned with how software is produced, how source and object code is traced to requirements, how the requirements trace to source and object code, and how the software is tested and shown to have been adequately tested.

F. Software Certification

There is a perception among those new to this field that software is somehow "certifiable" for airworthiness. This may come from a simple reading of the title of DO-178C/ED-12C *Software Considerations in Airborne Systems and Equipment Certification.* However, software is not actually "certifiable." Entities for which airworthiness certification can be granted are aircraft, engines, propellers, and, in the U.K., auxiliary power units. This means that the effort expended on achieving the "certifiability" of software is in actual practice expended on ensuring the certifiability of the aircraft, engine, or something else that is subject to the airworthiness certification effort — not the software involved.

For example, it is not possible to produce a "certified" version of a real-time executive or garbage collector, regardless of any statements in the marketing material of a particular vendor. What software vendors may do — and typically charge a substantial fee for — is provide the requirements, source traceability, requirements-based tests and test results for a particular software component that it provides. This documentation can then be submitted to the certification authority as part of the applicant's request for airworthiness certification of an engine, aircraft, or something else.

G. Software Production Process

The DO-178 series of documents are widely perceived as process-heavy; that is, they impose a substantial burden on the applicant to show that a particular process has been followed in the production of and verification of the airborne software that is being considered for certification. Although this has been widely considered a very expensive activity, over 35 years of airborne operations have not revealed any major safety flaws. Contrast this with, for example, the maiden flight of the Ariane 5 (Flight 501, June 4, 1996), which was destroyed 37 seconds after launch due to a software coding flaw — the failure to handle an exception raised during the initial boost phase. The Ariane 5 was never subjected to a civil airworthiness certification effort as its flights through civil airspace fall under a different authority, but it serves to illustrate that software coding errors can cause spectacular disasters.

Nevertheless, many people in the airborne software industry still believe that DO-178B/ED-12B made using less process-heavy techniques, such as model-based development, formal

methods and object-oriented programming, difficult when certification aspects were considered. The attitude is often one of "We already know how to create certifiable software the old-fashioned way. Why should we change now?" There is, therefore, a substantial perceived risk to adopting more modern techniques, regardless of the reduction in cost, errors, and time to market.

III. Rationale for Change

The answer to the question above is that the cost of doing things the old-fashioned way is becoming prohibitive. It now costs hundreds of millions of dollars for a new large aircraft to achieve airworthiness certification. That makes even small increases in efficiency lead to a competitive edge for airframe manufacturers and their equipment suppliers, who are able to be more efficient in their software production. Object-oriented programming is one way substantial increases in efficiency can be achieved, if only it can be used in an approved airworthiness certification effort.

Additionally, the software world has changed. Back in the 1980s, almost all airborne software was written from scratch to run on a single processor. This was a big problem for "commercial off the shelf" (COTS) software, as it was almost certainly not developed in an airborne software environment and therefore didn't have all the traceability and requirements-based test artifacts needed for eventual certification. This, obviously, has an impact on cost.

As far as safety is concerned, there's a real benefit to investing substantial resources into doing certain things right in a project-independent manner. Consider the wisdom of using a memory management system written by a specialist in real-time garbage collectors and used by thousands of developers rather than a pooled memory system written by a specialist in terrain avoidance and used by fifteen developers.

In the 1980s, people wanted to achieve safety goals through testing. One particular objective of software testing is "to demonstrate with a high degree of confidence that errors which could lead to unacceptable failure conditions, as determined by the system safety assessment process, have been removed." [5] The realization that this objective is, by and large, unobtainable in modern software systems has gained substantial consensus. It is widely felt that this view does not scale to the complex systems of current airborne software — let alone future systems — due to both hardware and software complexity. That is, exhaustive software testing will not reach the desired conclusion that all necessary errors "have been removed." This, in turn, has brought about a refocusing of the testing effort toward more realistic goals like reaching a reasonable level of confidence that the software is correct, safe, and useful rather than completely error-free.

IV. Changes with DO-178C/ED-12C

The creation of DO-178C/ED-12C brought about five auxiliary documents:

- DO-278A/ED-109A: *DO-278A Software Integrity Assurance Considerations for Communication, Navigation, Surveillance and Air Traffic Management (CNS/ATM) Systems.*
- DO-330/ED-215: *Software Tool Qualification Considerations.*
- DO-331/ED-218: *Model-Based Development and Verification Supplement to DO-178C and DO-278A.*
- DO-332/ED-217: *Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A.*
- DO-333/ED-216: *Formal Methods Supplement to DO178C and DO-278A.*

The remainder of this article will focus on only one of these: DO-332/ED-217[7], which was meant to address coding and verification issues.

Subgroup 5 — Object-Oriented and Related Technologies — took on the challenge of addressing, to a large extent, all coding issues. While issues such as dead and deactivated code, inlining, ad-hoc and parametric polymorphism are not particularly object-oriented, subgroup 5 addressed those issues along with more obviously Object-oriented (OO) topics such as inheritance, class hierarchy consistency, and run-time polymorphism.

The overall aim was to provide clarification of objectives from an OO viewpoint, to provide any new objectives that were deemed beneficial to airborne safety, and to provide guidance and recommendations for achieving those objectives.

A. OOTiA, CAST, FAA and EASA

One of the initial responsibilities of the subgroup was to address all issues raised in OOTiA and either incorporate them into the supplement or deem them inapplicable or unfounded. IP 508 was produced by the subgroup to respond to each individual concern raised by OOTiA. Additionally, concerns about OO had been raised through CAST papers, EASA CRIs[9] and FAA IPs. The subgroup was to also address all of these.

B. Dead and Deactivated Code

DO-178C/ED-12C disallows dead code, which is basically code that can never be executed. Dead code is treated as a software error that should be eliminated. A variant on this is deactivated code, which might be executed for a particular configuration not used in flight. An example of this might be a software-controlled radio, which includes code to control a military hardware encryption/decryption device but which would not be selected for a purely civilian application. This is already addressed by DO-178C/ED-12C. However, when reusing software components — especially externally developed software components such as class libraries — this comes into play as the abstraction for a component may include more behavior than is actually exercised by the airborne software.

Consider a stack class which is used as a previously developed component and which contains methods for "push", "peek" and "pop." All of these methods fit the abstraction for how a stack should work and are not out of place in a stack class. The particular airborne software using such a stack, however, might not actually use the "peek" method. The previous standard would have forced the practitioners to actually remove the code for the peek method before certification as it would be considered dead code. The new standard relaxes restrictions on separately developed components and allows this stack class to be used unmodified.

C. Type Theory

Early on, the subgroup decided to provide a type theoretical basis as a rationale for reducing the amount of redundant testing and verification that involved base classes and their derived subclasses. A great deal of this testing and verification can be shown to be redundant and therefore unnecessary through type-theoretical arguments that involve class hierarchy design, as long as the type hierarchies in question share certain properties. There is a notable absence of type theory — or, for that matter, any sort of formal computer science — as a basis for decision-making in DO-178B/ED12B. Subgroup members perceived this as being at some risk of being rejected by the subcommittee as a whole, but it was accepted in the end.

D. The Liskov Substitution Principle

This sort of type-theoretical formulation initially manifested itself in the specification of the Liskov Substitution Principle (LSP) [3] as the basis for establishing that superclass behavior verification could be used as part of the verification compliance of the subclass of that superclass. The point was that only the additional behavior provided by the subclass needed to be verified if that subclass conformed to LSP. Consider the formulation LSP which appears in the supplement:

Let *q(x)* be a property provable about objects *x* of type *T*.

Then *q(y)* should be true for objects *y* of type *S* where *S* is a subtype of *T*.

Regardless of the succinctness of this, the subgroup felt that a purely theoretical expression of this concept might place too great of a burden on the practitioners.

1) Explaining LSP: As the supplement neared completion, members of subgroup expressed that the definition given above needed to be explained more clearly. The inclusion of a Frequently Asked Questions (FAQ) section in the supplement provided a less structured environment into which the subgroup could place questions and answers that were presumed to be destined to be frequently asked. These essentially addressed the question "What's the deal with the Liskov Substitution Principle and why should I care?" The subgroup could have simply reiterated the concept and continued to claim that it was a good thing — which is true — but that probably wouldn't get the point across. Based on the idea that seeing a car crash is more conducive to reminding drivers why safety is important than listening to safety lectures, it was decided to show how failure to follow LSP could lead to problematic behavior. DO-178C/ED-12C is fundamentally a document about software safety, so this approach was considered reasonable.

2) Creating a Counterexample: For purposes of the supplement's FAQ, the following situation was proposed: There exists a conceptually abstract hardware speed controller that can be instantiated with the necessary behavior to reflect the hardware of many different manufacturers. This provides the necessary basis for creating a base class so that concrete subclasses could be created for each manufacturer's particular version with whatever device-specific low-level hardware interface was necessary. Additionally, some members felt that this example was something that practitioners would see as vaguely similar to the sort of software they were developing — software to control a pump for a fuel control system, maybe. A stretch, perhaps, but not an outrageous one.

3) Preconditions, Postconditions and Invariants: The argument is made that a number of different manufacturer's speed controllers would be substitutable for the base class as long as they correctly implemented the adjust speed method to communicate the desired increase in speed through whatever hardware-specific means necessary. A class invariant for the speed controller is that an instance's speed attribute is the magnitude of the velocity and therefore can never be less than zero. To use the Java terminology, the base class creates a means to adjust the speed by giving a speed increment to an adjust speed method. This adjust speed method's postcondition is that when given a positive, nonzero argument, the speed attribute of the object has increased. Therefore, it must be nonzero.

4) Time to Divide by Zero: Based on this postcondition and invariant, a method "time to go," taking a distance argument, will return in whatever units are convenient the time value it takes to traverse that distance. This ultimately reduces to dividing the given distance by the object's current speed attribute, then converting it to the correct units. The situation as outlined above represents a valid use of LSP. Any desired number of subclasseses of the speed controller can be created, each of which tailors its behavior to what is required by the underlying hardware. In order to demonstrate the failure of LSP, the subgroup introduced an "auto controller," a subclass of speed controller designed to control a fundamentally different type of hardware that is given a desired speed that it seeks to reach and maintain.

5) Breaking LSP: Since this new auto controller class no longer needs the adjust speed by a speed increment method, its necessary implementation of the method does nothing. Additionally, a "set desired speed" method would need to be introduced to address the new abstraction of this type of speed controller. The point of all this is that by having the auto controllers adjust speed method do nothing, the postcondition is violated, since invoking the adjust speed method on an object with zero speed would fail to make the actual speed attribute nonzero. This, in turn, would cause division by zero when the "time to go" method was invoked, causing a division by zero exception to propagate through the system. This should leave the reader with an image of the smoke and debris cloud ultimately resulting from that unhandled exception on the Ariane 5's maiden flight.

A. Local and Global Class Hierarchies

The supplement includes a brief explanation of the concept of hierarchical encapsulation so that it could form the basis for a discussion of class hierarchies which, in turn, brings about a discussion of type consistency for local and global type hierarchies. The supplement uses the term "local type consistency" to provide a means to determine type consistency in a component, independent of the type consistency of code which might utilize that component. That is, developers could make type consistency determinations with well-defined boundaries, facilitating the incorporation of separately (and often externally) developed class hierarchies.

B. Taxonomy of Polymorphism

Although not really object-oriented in nature — the charter of the subgroup being essentially all coding issues — the notion of polymorphism is approached from a type-theoretical basis as well. With a brief description of the forms of polymorphism as being universal polymorphism and ad-hoc polymorphism, each of these is discussed as being divided into parametric and inclusion polymorphism, coercion and overloading, respectively. Again, the subgroup did this with some apprehension but felt that at least introducing the vocabulary would provide additional means of clarifying situations where polymorphism is used and provide a common vocabulary for practitioners and certification authorities. A similar philosophy guided the decision to discuss closures as a means of specifying behavior; that is, if the terms are introduced in the supplement, an applicant can use the concept with an expectation that the certification authority will at least be on the same page.

1) Resource Management: One area in which DO-332 expects to have a large impact on software design in airborne systems is the provision of a section on resource management, especially heap management, where automatic garbage collection is explicitly permitted for the first time. Garbage collection in real-time systems is a subject on which a great deal of religious fervor has been expressed in the software safety community, especially the ongoing theme that garbage collectors are somehow too complex and therefore should not be allowed in a real-time or safety-critical situation.

A consistent problem encountered with this view is the inability of any of its proponents — or at least the ones with whom the subgroup communicated — to express just how complex "too complex" is, or even how such complexity should be measured. It was found to be especially curious that the notion was expressed — and fiercely defended — that garbage collectors were inherently too complex to be used in aviation but that high-bypass turbofan jet engines somehow were not.

While rejecting the notion that garbage collection — now and, presumably, forever — is unusable due to some unspecified and undefinable algorithmic complexity in all garbage collectors, the supplement recognizes the potential for heap memory exhaustion in an airborne system and gives guidance to detect it and provide a degraded mode into which the subsystem can transition if such a situation becomes imminent. The idea of throwing an unhandled out-of-memory exception is still possible, just as is the throwing of an unhandled division by zero exception. But the guidance and recommendations give developers and certification authorities a specific set of criteria to verify.

2) Functional Programming: There is an expectation that functional programming will be an increasingly important technique in future embedded systems, including airborne systems. As the future of processors seems to be in more cores rather than in more speed per core, a widely accepted technique for handling additional throughput is to move to a computation model involving an increased use of concurrency.

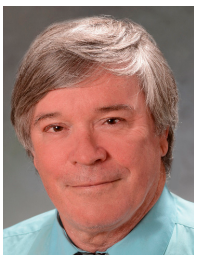
Handling concurrency will be increasingly important, and a well-established method for handling massive concurrency is the use of immutable data and functional programming. This was part of the driving force behind providing guidance for polymorphism, closures and garbage collection, all of which are heavily used in functional programming. The subgroup wanted to ensure that it left a path to functional programming for future practitioners.

V. Conclusion

Generally, the subgroup took the view that it should strive to remain language- and technology-neutral, but that it should use real languages (Ada, C++ and Java, in particular) and technology to provide examples and illustrations of problem areas (for example, static dispatch and violation of the Liskov Substitution Principle). The subgroup also subscribed to the view that this supplement will be the foundation for perhaps two decades of future safety-critical software implementation, so the subgroup needed to be careful and conservative in the resulting document. This was also done with the knowledge that the real-time, avionics and safety-critical communities are reluctant to introduce new concepts (garbage collection and runtime polymorphism, for example), so the subgroup needed to provide a basis for acceptability of such ideas to that community by furnishing a theoretical base for discussion as well as an analysis of the perceived risks of a given approach and recommendations for mitigating those risks.

In general, the subgroup feels that DO-332/ED-217 provides a sound collection of techniques to mitigate the difficulty and expense involved with creating and validating airborne software, now and in the future.

ABOUT THE AUTHOR



Michael R. Elliott is a software engineer with a deep passion for modern software practice, embedded systems architecture and safety- and security-critical software. Elliott was a member of SC-205/WG-71 Subgroup 5, which developed DO-178C/ED-12C and DO-332/ED-217. He has a bachelor's degree in information and computer science from the University of California, Irvine and a master's degree in software engineering from Edinburgh University, Edinburgh, Scotland.

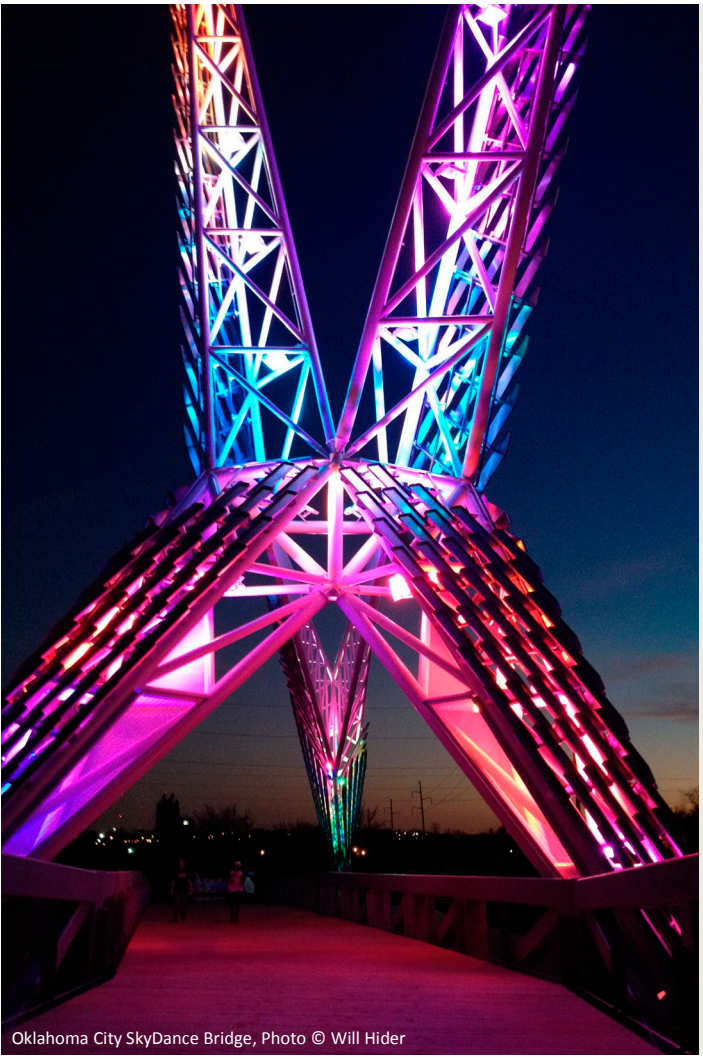
aicas GmbH
Haid-und-Neu Straße 18
76131 Karlsruhe Germany
elliott@aicas.de

NOTES

1. Federal Aviation Administration. Washington, D.C., USA. http://www.faa.gov.
2. European Aviation Safety Agency. Cologne, Germany. http://www.easa.europa.eu.
3. RTCA, Inc. (Washington, D.C., USA) is an organization that creates standards documents for the FAA. http://www.rtca.org.
4. The European Organization for Civil Aviation Equipment (Malakoff, France) is an organization that produces documents referred to as a means of compliance for European Technical Standard Orders. http://www.eurocae.net.
5. Information Paper.
6. Object-Oriented Programming Systems, Languages and Applications conference of the Association for Computing Machinery.
7. National Aeronautics and Space Administration. Washington, D.C., USA. http://www.nasa.gov.
8. Certification Authority Software Team, a group of individuals representing several certification authorities, including EASA, FAA, JAA and Transport Canada.
9. Certification Review Items.

REFERENCES

1. Federal Aviation Administration (FAA). (Oct. 2004.) "Handbook for Object-Oriented Technology in Aviation (OOTiA)." Washington, D.C., USA.
2. Johnson, L. (Oct. 1998.) DO-178B, "Software Considerations in Airborne Systems and Equipment Certification." Crosstalk, The Journal of Defense Software Engineering, 11 (10).
3. Liskov, B. & Wing, J. (Nov. 1994.) "A Behavioral Notion of Subtyping." ACM Transactions on Programming Languages and Systems, 16(6): 1811–1841.
4. Special Committee 152 of RTCA. (March 1985.) "DO-178A/ED-12A – Software Considerations in Airborne Systems and Equipment Certification." RTCA and EUROCAE. Washington, D.C., USA and Paris, France.
5. Special Committee 167 / Working Group 12 of RTCA and EUROCAE. (Dec. 1992.) "DO-178B/ED-12B – Software Considerations in Airborne Systems and Equipment Certification." RTCA and EUROCAE. Washington, D.C., USA and Paris, France.
6. Special Committee 205 / Working Group 71 of RTCA and EUROCAE. (Dec. 2011.) "DO-178C/ED-12C – Software Considerations in Airborne Systems and Equipment Certification." RTCA and EUROCAE. Washington, D.C., USA and Malakoff, France.
7. Subgroup 5 of Special Committee 205 / Working Group 71 of RTCA and EUROCAE. (Dec. 2011.) "DO-332/ED-217 – Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A." RTCA and EUROCAE. Washington, D.C., USA, and Malakoff, France.



Oklahoma City SkyDance Bridge, Photo © Will Hider