TECHNICAL CHALLENGES IN USING OBJECT ORIENTED TECHNOLOGY IN AVIATION APPLICATIONS

Kelly J. Hayhurst, NASA Langley Research Center, Hampton Virginia

Abstract

When it comes to software engineering technologies, enthusiastic boasts about production efficiencies are more common than sober discussions about technical difficulties—that’s the nature of marketing. However, understanding potential technical difficulties in new technologies is essential, particularly when it comes to using these technologies in safety-critical systems. Such is the case with object oriented technology (OOT).

In 2001, the Federal Aviation Administration (FAA) and the National Aeronautics and Space Administration (NASA) started the Object Oriented Technology in Aviation (OOTiA) project to facilitate safe use of OOT in aviation applications. A primary objective of the project was to identify and document safety and certification concerns about using OOT in compliance with RTCA/DO-178B, Software Considerations in Aircraft Equipment and System Certification. This paper provides a brief survey of the technical challenges that have been identified so far.

Introduction

Within the broad software engineering community, developers have made widespread use of object-oriented technology, including object oriented modeling, design, programming, and analysis, for well over a decade. New technologies, however, tend to make their way more slowly into a regulated environment. This is certainly true of OOT in commercial aviation applications. To date, few airborne computer systems in civil aviation have been implemented using OOT.

In the software engineering world, perhaps more so than many others, there is a constant stream of new development technologies. In a recent article, Robert Glass makes the following observation about acceptance of new software engineering concepts:

"It is high time that we in the software field questioned some of its old wives’ and husbands’ tales. That we question the hype spewing from the hype purveyors, especially vendors. That we question the advocacy spewing from all too many computer science researchers, who seem to feel that research papers should "conceive a concept, advocate the concept, and scold practitioners who refuse to use the concept."

Imagine the outcome of such questioning. We could have books with titles such as

- Questioning Structured Programming
- Questioning CASE Tools
- Questioning Object Orientation
- Question Web Services
- Questioning Formal Methods

And we could engage in honest-to-goodness, objective debates about the merits of all those new concepts that keeping flowing down the software engineering pipeline. [1]"

In the spirit of Glass’s observation, the FAA and NASA started the Object Oriented Technology in Aviation (OOTiA) project to debate the merits of OOT in commercial aviation applications. This project built on initial work by Rierson [2] and the Aerospace Vehicle Systems Institute [3] to identify certification issues with OOT, and opened the debate to the aviation software industry at large. The goal of the OOTiA project included identifying concerns and questions about OOT relevant to safety and certification and developing recommendations for its safe use and compliance with RTCA/DO-178B, Software Considerations in Airborne Systems and Equipment Certification [4]. Compliance with DO-178B is the primary means of securing approval of software in civil transport aviation products. A similar standard, DO-278 [5], applies to software used in communications, navigation, and surveillance applications for air traffic management. Neither DO-178B nor DO-278 explicitly mentions OOT. Object-oriented (OO)
programs that have sought regulatory approval, so far, have been required to formulate issue papers to respond to certification concerns.

To foster dialogue and debate within the aviation software community about OOT, the OOTIA project accomplished the following:

- Established a web site dedicated to collecting data on safety and certification concerns, http://shemesh.larc.nasa.gov/oot/
- Held public workshops to which the aviation software community was invited to discuss concerns
- Documented key concern and questions raised either through the web site or the workshops
- Developed guidelines to address the concerns
- Drafted an OOTIA handbook.

The OOTIA handbook records the results of the above activities, and comprises 4 volumes:

Volume 2 of the handbook covers the concerns and questions about OOT documented through special workshop breakout sessions and through the issue list on the OOTIA web site. Some of the results of the special workshop breakout sessions are reported in [6]. This paper focuses expressly on the part of Volume 2 that deals with the technical challenges to safe use of OOT in compliance with DO-178B, as identified in the issue list. The purpose of this paper is not to provide in-depth coverage of the technical detail, but to raise general awareness of the subject matters that are a concern to safety and certification. The paper is structured as follows. First, the OOTIA issue list is briefly described. Next, the technical challenges to using OOT in a DO-178B context are presented. Finally, brief concluding remarks are made.

Organizing the OOTIA Issue List

A total of 103 separate concerns about various aspects of OOT were collected on an issue list through the OOTIA web site. That list is available at http://shemesh.larc.nasa.gov/ootissues.html. To help consolidate the concerns on the list, entries that expressed similar concerns were grouped together. The groups were sorted according to the DO-178B life cycle process that is most influenced by that group to help provide a direct link to the guidance in DO-178B. The software life cycle processes specified in DO-178B are

- Planning Process
- Development Processes (requirements, design, code, and integration)
- Integral Processes (verification, configuration management, quality assurance, and certification liaison)

In addition to these life cycle processes, section 12 of DO-178B, titled Additional Considerations, provides guidance for ancillary topics such as tools, previously developed software, and formal methods. Because OO tools were the main topic of a number of the entries, Addition Considerations seemed like a reasonable fourth category for organizing the list.

Each comment recorded on the issue list was mapped into one of the categories (Planning Process, Development Processes, Integral Processes, or Additional Considerations). For each category, technical challenges relevant to DO-178B were derived from the issues mapped to that category. The following sections describe the technical challenges within each category.

To support the discussion, entries from the issue list are cited and denoted by their tracking number from the OOTIA web site. For example, IL 1 is the first entry to the issue list, so (IL 1) will

---

1 The current expectation is that the FAA will release the OOTIA Handbook in the Fall 2004. After it has been released, the handbook will also be available through the OOTIA web site.

2 There are actually 107 entries to the list, but four of them are duplicates.

3 Some concerns span multiple life cycle processes. Determining which process is most influenced is necessarily subjective.
appear directly after the citation of that issue. It is important to note that mentioning an entry from the issue list does not imply its validity; some readers likely will dispute the validity of individual entries. The purpose of citing the entries is to give an account of the data that was collected, and show the basis for the challenges discussed within that category. The technical challenges, not the individual issues, are what is important.

**Challenges for the Planning Process**

The planning process in DO-178B specifies the development and integral process activities, environment, and standards for a project. Planning decisions about how OOT will be used to meet the DO-178B objectives are recorded in documents such as the Plan for Software Aspects of Certification, Software Development Plan, and integral process plans. The challenges identified for the planning process involve life cycle data, requirements, and standards.

**Defining Life Cycle Data**

OOT introduces new notations and models, such as behavior and implementation diagrams (use cases, class, sequence, component, deployment, activity, and statechart diagrams), that do not map directly to the data indicated for requirements, design, and code in DO-178B. A description of the software life cycle data is typically included in the Plan for Software Aspects of Certification, as discussed in section 11.1e of DO-178B. Some of the entries from the issues list (IL 77 and 87) specifically questioned how low level requirements, design, and source code would be defined for an OO program. Determining how the life cycle data from an OO development process maps to the life cycle data specified in DO-178B section 11 is a non-trivial challenge.

**Requirements Methods and Notations**

Another challenge for planning relates specifically to requirements methods and notations. Standards for software requirements, as described in 11.6 of DO-178B, define the methods, rules, notations, and tools for developing high-level requirements. The challenge here lies in determining whether OO approaches to requirements definition (Unified Modeling Language, UML, for example) are adequate for all types of requirements. Issue list entries covered two specific areas: (1) concern about adequately capturing non-functional requirements (IL 75), including performance requirements (IL 80); and (2) concern about the tendency in OOT to group requirements in a graphical format—possibly making identification of low-level requirements and derived requirements difficult, and complicating safety assessment (IL 79).

**Restrictions**

The final topic relevant to planning deals with restrictions. Section 4.5c of DO-178B states, "the software development standards should disallow the use of constructs or methods that produce outputs that cannot be verified or that are not compatible with safety-related requirements" [4]. Some OO languages have features that can make it extremely difficult or impossible to satisfy the objectives of DO-178B. In such cases, coding standards might be necessary to specify a well-defined subset of the programming language that will allow compliance with objectives. Multiple inheritance is one controversial feature discussed in the issue list (IL 38) that may require restriction. The challenge is to determine what restrictions and rules on language features, such as multiple inheritance, are warranted for a given software level.

**Challenges for Development Processes**

Over 40 of the issues on the list are related to or affect development processes (requirements, design, code, and integration). Many of these comments describe ways that OO features promote complexity and ambiguity in the requirements, design, and code, potentially making the two assurance principles in DO-178B (assurance of intended functionality and assurance of no unintended functionality) difficult to meet. Sections 6.3.1 and 6.3.2 of DO-178B describe objectives for ensuring that high and low-level requirements are accurate, unambiguous (written in terms that only allow a single interpretation), and consistent.
Challenges related to subtypes, subclasses, memory management and initialization, and dead and deactivated code were identified.

**Subtypes**

Inheritance allows different objects to be treated in the same general way. Through inheritance, programmers can develop types by basing new type definitions on existing ones. In a type hierarchy, supertypes should capture the behavior that all of the descendent subtypes have in common. A clear understanding of how subtypes and supertypes are related is essential to having a correct type hierarchy. Type substitutability and inconsistent type use are two issues here.

**Type Substitutability**

The substitutability issue involves the suitability of various subtypes that are possible at the point of a call. Whenever a system expects a value of type T, a value of type T', where T' is a subtype of T, can be substituted. Improper subtyping can result in unintended functionality. The challenge is to prevent or detect improper subtyping that can result in unintended functionality.

That is, how do we know, or provide assurance, that this substitution is always appropriate? (IL 42) The Liskov Substitution Principle (LSP) proposes a theoretical means to mitigate improper subtyping by constraining the behavior of subtypes [7]:

*Subtype Requirement:* Let $\phi(x)$ be a property provable about objects $x$ of type $T$. Then $\phi(y)$ should be true for objects $y$ of type $S$ where $S$ is a subtype of $T$.

In theory, conformance to this principle would satisfy the assurance dilemma. However, in practice, this is difficult because the semantics of real programming languages differ considerably from the simple model used in [7], and establishing the subtype relation may require verification (formal proof) beyond traditional (DO-178B-type) testing.

**Inconsistent type use**

Another subtyping problem is inconsistent type use. When a descendant class does not override any inherited method (that is, there is no polymorphic behavior), anomalous behavior can occur if the descendant class has extension methods resulting in an inconsistent inherited state. (IL 91)

**Subclasses**

Just as with types, hierarchies of classes can be constructed where subclasses are created from more abstract superclasses. A subclass automatically inherits all of the visible attributes and operations of the superclass; but can override inherited operations and add new attributes and operations. Ambiguity is a major concern.

**Unclear Intent**

With multiple inheritance, a subclass may have more than one superclass, so the same operation may be inherited from multiple sources. In these cases, the intent of the operation at the subclass level might not be clear. Cuthill notes that overuse of inheritance, particularly multiple inheritance, can lead to unintended connections among classes [8]. The key concern is that the original intent for a subclass or operation may not be clear. Examples of specific concerns about unclear intent from the issue list include:

- When the same operation is inherited by an interface via more than one path through the interface hierarchy (repeated inheritance), it may be unclear whether this should result in a single operation in the subinterface, or in multiple operations. (IL 27)
- A subclass may be incorrectly located in a hierarchy because the complete definition/intention of a class may not be clear. (IL 21)
- When a subinterface inherits different definitions of the same operation (as a result of redefinition along separate paths), it may be unclear whether/how they should be combined in the resulting subinterface. (IL 28)
- Top-heavy multiple inheritance and very deep hierarchies (six or more subclasses) are error-prone, even when they conform to good design practice. The wrong variable type, variable, or method may be inherited, for example, due to confusion about a multiple inheritance structure. (IL 24)
Overriding

Overriding is the redefinition of an operation or method in a subclass. Unintentionally overriding an operation is easy in some OO languages because of the lack of restrictions on name overloading (the use of the same name for different operators or behavioral features, operations or methods, visible within the same scope). The consequence is that a method of the expected name but of a different type might be called in a program. It is important that the overriding of one operation by another and the joining of operations inherited from different sources always be intentional rather than accidental. (IL 32)

Offutt has identified five classes of errors associated with overriding [9]. Each class was submitted as a concern to the issue list:

- **State Defined Incorrectly**: If a computation performed by an overriding method is not semantically equivalent to the computation of the overridden method with respect to a variable, a behavior anomaly can result. (IL 94)
- **Anomalous Construction Behavior 1**: If a descendant class provides an overriding definition of a method which uses variables defined in the descendant's state space, a data flow anomaly can occur. (IL 96)
- **Anomalous Construction Behavior 2**: If a descendant class provides an overriding definition of a method which uses variables defined in the ancestor's state space, a data flow anomaly can occur. (IL 97)
- **State Definition Anomaly**: If refining methods do not provide definitions for inherited state variables that are consistent with definitions in an overridden method, a data flow anomaly can occur. (IL 92)
- **State Visibility Anomaly**: When private state variables exist, if every overriding method in a descendant class doesn't call the overridden method in the ancestor class, a data flow anomaly can exist. (IL 99)

It is also important to note that overriding is affected by language features such as overloading. In theory, overloading enhances readability when the overloaded operators, operations, or methods are semantically consistent. (IL 60) However, overloaded operators, operations, and methods contribute to confusion and human error when they introduce methods that have the same name but different semantics.

Memory Management and Initialization

Memory management involves making accessible the memory needed for a program's objects and data structures from the limited resources available, and recycling that memory for reuse when it is no longer required. According to section 11.10 DO-178B, the software design data should discuss limitations for memory and the strategy for managing memory and its limitations. "The basic problem in managing memory is knowing when to keep the data it contains, and when to throw it away so that the memory can be reused. This sounds easy, but is, in fact, such a hard problem that it is an entire field of study in its own right" [10].

Indeterminate Execution Profiles

One problem with memory allocators is that regardless of the allocation algorithm used, allocating and deallocating memory repeatedly leads to fragmentation. Some algorithms, of course, lead to more fragmentation than others. In any case, periodic reorganization of memory is needed to reduce fragmentation. The problem is that many traditional allocation and deallocation algorithms are unpredictable in terms of their worst-case memory use and execution times, resulting in indeterminate execution profiles (IL 66). In a DO-178B context, understanding the worst-case is important.

Initialization

In OO programs, class hierarchies (deep hierarchies in particular) may lead to initialization problems. Sections 6.3.4 and 6.4.3 of DO-178 address incorrect initialization of variables and constants. A potential problem is that a subclass method might be called (via dynamic dispatch) by a higher level constructor before the attributes associated with the subclass have been initialized. (IL 19) Inadequate initialization can lead to the
incomplete (failed) construction problem identified by Offutt [9]. According to Offutt, there are two possible faults, depending on programming language:

"First, the construction process may have assigned an initial value to a particular state variable, but it is the wrong value. That is, the computation used to determine the initial value is in error. Second, the initialization of a particular state variable may have been overlooked. In this case, there is a data flow anomaly between the constructor and each of the methods that will first use the variable after construction (and any other uses until a definition occurs)" [9].

Dead or Deactivated Code

The final area of concern for development processes is dead and deactivated code. The glossary of DO-178B describes 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." [4].

"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" [4].

Section 6.4.4.3 of DO-178B requires that dead code be removed and analysis performed to assess the effect and need for reverification, and Section 5.4.3 of DO-178B requires deactivated code to be verified (analysis and test) to prove that it cannot be inadvertently activated.

In OO programs, dead and deactivated code may be more common than in other style programs due to reuse of objects. In a reusable component, the requirements, design, and code might cover more functionality than required by the system being certified. In a DO-178B context, there is a trade-off between the benefit of reuse and the cost of additional verification of deactivated code or removal of dead code.

Identifying Dead and Deactivated Code

The difference between dead and deactivated code is not always clear when using OOT. Without good traceability, identifying dead versus deactivated code may be difficult or impossible. Dead or deactivated code can result when (a) methods of a class are not called in a particular application; (b) methods of a class are overridden in all subclasses; or (c) attributes of a class are not accessed in a particular application [11]. For example, no instances of a superclass might be used if all of the subclasses override a particular method.

Libraries and Frameworks

Dependence on libraries is also a concern for safety-critical systems because it is often unclear 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 [12]. Deactivated code will be found in any application that uses general purposed libraries or object-oriented frameworks. (IL 1)—the challenge is to readily identify it.

This challenge applies equally well to non-OO systems. However, some might argue that dependence on libraries and frameworks may be more extensive in an OO system. In any case, use of libraries must be carefully considered and verified for proper functionality.

Challenges for Integral Processes

Complexity and ambiguity raise challenges for the integral processes, just as they do for development processes. The entries from the issue list dealing with integral processes were partitioned into three categories: verification, configuration management, and traceability. Although traceability is not called out as an integral process in DO-178B, it was categorized under integral processes because traceability data is often collected in conjunction with verification activities. The key challenges within each of the three categories are discussed below.


**Verification - Analysis**

Verification, in the DO-178B context, examines the relationship between the software product and the requirements. The goal is to use reviews, analyses, and tests to detect and report errors introduced during the development processes. OO techniques, especially dynamic dispatch and polymorphism, can complicate various verification activities required by DO-178B (IL 9). Program control flow can be difficult to predict (if it can be predicted at all) because polymorphism forces binding to be delayed until execution time. Execution-time circumstances can cause a single line of source code to mean many different things depending upon specific data values that the program sees. For example, given a function \( f(x) \), which \( f() \) to call depends on which class \( x \) belongs to, which might be multiple classes depending on the run-time state of the system. This is not a problem of doubt about a conditional statement—it is doubt about what a specific function call means because of dynamic dispatch \[13\]. This particular problem complicates testing and many different types of analyses including flow and coupling analysis, structural coverage analysis, and source to object traceability as discussed below.

**Flow Analysis**

Data and control flow analysis must be performed for software levels A-C to confirm data and control coupling between code components. OO features tend to make data and control coupling relationships more complicated and obscure than in software developed using procedural languages. Dynamic dispatch complicates flow analysis, as described above, because it might be unclear which method in the inheritance hierarchy is going to be called. OO design, in general, encourages the development of many small, simple methods to perform the services provided by a class. Determining the correctness of control flow decisions requires analysis of how individual data objects that control the execution flow of the software are created and maintained: where and when are the objects created or destroyed, where and when are the values of the objects set, and how are any potential "shared data" conflicts controlled. The key concern here is that decision points can use data objects with values that are maintained in other parts of the software that might be remote from the proximate path of execution—making flow analysis difficult \[14\].

OOT also encourages hiding the details of the data representation (that is, attributes) behind an abstract class interface. The commonly suggested best practice is that attributes of an object should be private, and access to them only provided through the methods appropriate to the class of the object. Being able to access attributes only through methods makes the interaction between two or more objects implicit in the code, complicating analysis.

Many of the relevant entries from the issue list expressed questions about how to do this analysis.

- How can we meet the control and data flow analysis requirements of DO-178B with respect to dynamic dispatch? (IL 56)
- Flow analysis, recommended for Levels A-C, is complicated by dynamic dispatch (just which method in the inheritance hierarchy is going to be called?). (IL 2)
- Flow analysis and structural coverage analysis, recommended for Levels A-C, are complicated by multiple implementation inheritance (just which of the inherited implementations of a method is going to be called and which of the inherited implementations of an attribute is going to be referenced?). The situation is complicated by the fact that inherited elements may reference one another and interact in subtle ways which directly affect the behavior of the resulting system. (IL 16)

OO language features such as inlining can also complicate flow analysis because inlining can cause substantial differences between the flow apparent in the source code and the actual flow in the object code.

**Structural Coverage Analysis**

Structural coverage analysis is required in DO-178B for software levels A-C. The intent of structural coverage analysis, in the DO-178B context, is to complement requirements-based testing by: (1) providing evidence that the code structure was verified to the degree required for the...
applicable software level; (2) providing a means to support demonstration of absence of unintended functions; and, (3) establishing the thoroughness of requirements-based testing [15].

Structural coverage analysis is complicated by dynamic dispatch because structural coverage changes when going from subclass to superclass. Structural coverage in an OO program is not meaningful unless coverage measurements are context dependent; that is, based on the class of the specific object on which the methods were executed. “Coverage achieved in the context of one derived class should not be taken as evidence that the method has been fully tested in the context of another derived class” [16]. Inlining, templates, and code sharing were all identified in the issue list as impacting structural coverage analysis. A more detailed look at the effect of OOT on structural coverage is available in [17].

Timing and Stack Analysis

Timing analysis, worst-case execution time in particular, and stack usage are both part of review and analysis of source code in DO-178B section 6.3.4f. Stack overflow errors are listed in section 6.4.3f of DO-178B as errors that are typically found in requirements-based hardware/software integration testing. Timing and stack analysis are complicated by certain implementations of dynamic dispatch. With some implementations of dynamic dispatch, it is difficult to know just how much time will be expended determining which method to call. (IL 3) If polymorphism and dynamic binding are implemented, stack size can grow, making analysis of the optimal stack size difficult. (IL 107)

Timing and stack analysis are also affected by inlining, templates, and macro-expansion. Inline expansion can eliminate parameter passing, which can affect the amount of information pushed on the stack as well as the total amount of code generated. This, in turn, can affect the stack usage and timing analysis.

Source to Object Trace

Source to object code traceability tends to be a controversial issue; object orientation does not improve the situation. As discussed in DO-248B, for level A software, it is necessary to establish whether the object code is directly traceable to the source code. If the object code is not directly traceable to the source code, then additional verification should be performed [4]. Dynamic dispatch complicates source to object code traceability because it might be difficult to determine how the dynamically dispatched call is represented in the object code. (IL 6) In addition, source to object code correspondence will vary between compilers for inheritance and polymorphism, along with constructors/destructors and other language helper functions. (IL 12)

Some OO language features, such as inlining and implicit type conversion, can also complicate source to object code traceability. Inline expansion may not be handled identically at different points of expansion. This can be especially true when inlined code is optimized. (IL 46) Implicit type conversion raises certification issues related to source to object code traceability, the potential loss of data or precision, and the ability to perform various forms of analysis called for by DO-178B including structural coverage analysis and data and control flow analysis. It may also introduce significant hidden overheads that affect the performance and timing of the application. (IL 59)

Verification - Testing

Testing in DO-178B has two high-level objectives tied to the two fundamental assurance principles: demonstrating that the software satisfies its requirements, and demonstrating (with a high degree of confidence) that errors that could lead to unacceptable failure conditions have been removed (section 6.4 of DO-178B). According to Alexander, “object-oriented programs are generally more complex than their procedural counterparts. This added complexity results from inheritance, polymorphism, and the complex data interactions tied to their use. Although these features provide power and flexibility, they increase complexity and require more testing” [18]. The topic of testing OO programs is huge in scope with many books and considerable research devoted to the subject.

Issue list entries identified two general challenges with respect to testing: requirements testing and test case reuse.

Some of the books themselves are huge; for example Binder’s Testing Object-Oriented Systems, Models, Patterns, and Tools [19] is over 1000 pages!
Requirements Testing
The first challenge goes back to the difference between the functional and object perspective. Three levels of tests are called out in DO-178B testing activities: low-level tests, software integration tests, and hardware/software integration tests. The challenge for requirements testing is that the mapping of function-oriented test cases to an object-oriented implementation might not be obvious, because the basic unit of testing in an OO program is not a function or a subroutine, but an object or a class. Also, test coverage of high-level and low-level requirements will likely require different testing strategies and tactics from the traditional structured approach because information hiding and abstraction techniques decrease or complicate the observability of low-level functions.

Test Case Reuse
The second challenge area deals with the reuse of test cases or, more specifically, determining the appropriate reuse of test cases. Inheritance, dynamic dispatch, and overriding complicate matters because it might be difficult to determine how much testing at a superclass level can be reused for its subclasses. (IL 4)

- Inheritance and overriding raise a number of issues with respect to testing: “Should you retest inherited methods? Can you reuse superclass tests for inherited and overridden methods? To what extent should you exercise interaction among methods of all superclasses and of the subclass under test?” (IL 18)

Configuration Management
Although configuration management is addressed in only a few entries from the issue list, the comments are worth noting here because configuration management is an essential element of achieving regulatory approval. In DO-178B, configuration identification and control involve defining what constitutes a configuration item, and defining processes for controlling (baselining and changing) those items.

Configuration Identification and Control
Concerns were raised in entries from the issue list regarding what constitutes a configuration item. Section 7.2.1 of DO-178B discusses the need for unambiguous labeling of each configuration item so that a basis is established for control. Configuration management may be difficult in OO systems, causing traceability problems. If the objects and classes are considered configuration items, they can be difficult to trace, when used multiple times in slightly different manners. (IL 76)

OO tools and modeling languages such as UML also might affect the way configuration items are managed and changed. Change impact analysis may be difficult or impossible due to difficulty in tracing functional requirements through implementation. (IL 74) Uniquely identifying and controlling configuration items can be a significant challenge.

Traceability
Although traceability is not called out as a life cycle process in DO-178B, traceability plays a large role in providing assurance of intended functionality and of the absence of unintended functionality. DO-178B guidelines require traceability between (a) system requirements and software requirements to enable verification of the complete implementation of the system requirements and give visibility to derived requirements; (b) low-level and high level requirements to verify the architectural design decisions, give visibility to derived requirements, and demonstrate complete implementation of high-level requirements; and (c) source code and low-level requirements to enable verification of the absence of undocumented source code and verification of the complete implementation of the low-level requirements (section 5.5 of DO-178B). Four areas of concern about traceability were identified in the issue list.

Function versus Object Tracing
Traceability of functional requirements through implementation might be lost or difficult with an OO program because of mismatches between function-oriented requirements and an object-oriented implementation (IL 61). The use of OO methods typically leads to the creation of many small methods which are physically distributed over a large number of classes. This, and the use of dynamic dispatch, can make it difficult for developers to trace critical paths through the application during design and coding reviews. (IL
Inheritance, polymorphism, and overloading exacerbate the problem by increasing the complexity of interaction among distributed objects.

Complexity
Another key concern for traceability is that class hierarchies, especially those constructed through multiple inheritance, can become overly complex. This complexity makes tracing functional requirements difficult, and can impact safety analysis. Current safety analysis is often based on determining that a function, as implemented, is both correct and safe. OOT complicates this analysis because the operations related to a function can be widely distributed throughout the objects, making the function difficult to trace.

Tracing Through OO Views
Traceability can also be affected by different graphical representations used in OOT. OO requirements, design, and implementation might have multiple “views”; for example, the class diagram in the logical view. Traceability may get lost, or be made more difficult, because behavior of the classes and how they interact together to provide the required function might not be visible in any single view. Dynamic information concerning control flow or data flow might not be visible [19].

Unfortunately, many current OO tools do not currently provide a mechanism to trace requirements through multiple views and notations. IL 72 notes that traceability is made more difficult because there is often a lack of OO methods or tools for the full software lifecycle.

Iterative Development
Some OO programs are developed through iteration. The risk of losing traceability may increase when using an iterative development process because of increased changes to development artifacts (IL 107). Although this is certainly a traceability issue, this could also be considered when planning for the development process.

Challenges for Tools
Ten of the 103 issues on the issue list were categorized under Additional Considerations, with most of these being tool issues. Tool issues are a non-trivial part of compliance with DO-178B. DO-178B is concerned with identifying the tools that are used throughout the software life cycle, controlling those tools, and qualifying them.

Part of the claimed appeal of OOT is the wide range of tools that support development and verification. Some OO tools provide support for developing design and code through framework libraries of patterns, templates, generics, and classes, and also a framework to automatically generate source code from models. A fundamental question about OO tools is whether the introduction of these tools in the development process contributes to the integrity of the software products or adds additional burden for verification. Two particular areas of concern regarding tools are: (1) long-term maintainability and maturity of tools and tool environments and (2) qualification.

Tool Environments
DO-178B provides guidelines on software life cycle environment control in section 7.2.9 to ensure that tools that are used are properly identified, controlled, and retrievable. An important concern about OO tools is that the rapid rate of evolution of these tools may conflict with the needs of aviation software developers to support tools for relatively long periods of time. Concern was also expressed about anticipating new tool types. Specific entries in the issue list pertaining to tool environments include:

- Maturity/long-term support of tools. Tool manufacturers may not realize the long-life need of tools. Is this a higher risk in the OO environment? Education for both the tool and aviation communities to understand the specific needs for tool manufacturers and aircraft manufacturers. Not necessarily OO-specific, but might be more prevalent with OO. (IL 85)

- Maintaining tool environment, archives, ... when licenses are involved is not clear. May need to have some kind of “permanent license” to support safety and continued airworthiness of the aircraft. OO more dependent on tools, but not necessarily an OO-specific issue. (IL 84)

- Are there other types of OO tools that need to be addressed? Need to
anticipate other classes of tools that may come onto the scene; e.g., traceability tool for OO, transformation tools, CM tools, refactoring tools (tool to restructure source code to meet new requirements), (IL 86)

**Tool Qualification**

According to section 12.2 of DO-178B, tool qualification is needed when processes in DO-178B are eliminated, reduced, or automated by the use of a software tool without its output being verified. Qualification requirements differ depending on whether a tool is a development tool (whose output is part of the software) or a verification tool (that cannot introduce errors, but might fail to detect them). Tools that automate the software development process are subject to more rigorous qualification requirements than those that automate verification processes.

The distinction between development tool and verification tool may not always be clear for OO tools, especially model-based tools. A challenge for qualification is whether new or different qualification criteria are needed for OO specific tools such as visual modeling tools or OO frameworks that serve as the basis for multiple data items.

The following entries from the issue list deal with these concerns:

- **Auto-test and code generation tools** – what are the concerns when a single tool generates code and test from the same model? The concern is with the independence – same input and same tool. (IL 83)
- **When using OO tools to develop software requirements, design and implementation**, it is beneficial to work at the visual model level, especially when using UML. When working with OO tools, configuration management might be done at the modeling level (i.e., diagrams). This may cause a concern when the OO tools can introduce subtle errors into the diagrams. (IL 100)

**Summary**

While new software technologies may offer the promise of exciting development efficiencies and capabilities, those technologies also may present serious challenges to safety and certification. The existence of these challenges, however, may easily get lost under a deluge of promotional hype. For safety-critical systems, recognizing and solving these challenges is essential; otherwise, using the new technologies may inadvertently introduce safety hazards. As Glass has said, "[t]here is a cost to learning new ideas. We must come to understand the new idea, see how it fits into what we do, decide how to apply it, and consider when it should and shouldn't be used" [20].

To this end, this paper presented an overview of a number of technical challenges to safely using OOT in an aviation application. These challenges span the life cycle processes specified in the RTCA/DO-178B guidelines. The list of challenges is certainly incomplete; however it provides a starting point for constructive debate about the true merits of OOT and for developing guidelines for how OOT should and shouldn't be used in aviation applications.

**Acknowledgement**

The technical challenges raised in this paper were taken from the list of issues about OOT that were submitted to the OOTIA web site. I thank all of the contributors to this list for their candor, insight, and diligence in raising concerns, and to the OOTIA workshop participants for debating and refining these concerns into discernible challenges.

**References**


with Embedded Object-Oriented Software, version
1.2.

Considerations in Airborne Systems and Equipment
Certification, RTCA/DO-178B, Washington, D. C.

Communication, Navigation, Surveillance, and Air
Traffic Management (CNS/ATM) Systems
Software Integrity Assurance, RTCA/DO-278,
Washington, D. C.

[6] Hayhurst, Kelly J., and C. Michael Holloway,
12-17 October 2003, Considering Object Oriented
Technology in Aviation Applications, 22nd Digital
Avionics Systems Conference, Indianapolis, IN, pp.

1994, A Behavioral Notion of Subtyping, ACM
Transactions on Programming Languages and

[8] Cuthill, Barbara, 1993, “Applicability of Object-
Oriented Design Methods and C++ to Safety-
Systems Reliability and Safety Workshop.

[9] Offutt, Jeff, Roger Alexander, Ye Wu,
Quansheng Xiao, Chuck Hutchinson, November
2001, “A Fault Model for Subtype Inheritance and
Polymorphism,” The 12th IEEE International
Symposium on Software Reliability Engineering,
Hong Kong, PRC, pp. 84-95.

[10] The Memory Management Reference
Beginner's Guide Overview, archived at
http://www.memorymanagement.org/articles/begin.

(CAST), January 2002, “Use of the C++
Programming Language,” Position Paper CAST-8,
available at
http://www.faa.gov/certification/aircraft/ay-
info/software/CAST_Papers.htm. Visited on 6 July
2004.

[12] Certification Authorities Software Team
(CAST), January 2000, “Object-Oriented
Technology (OOT) in Civil Aviation Projects:
Certification Concerns,” Position Paper CAST-4,
available at
http://www.faa.gov/certification/aircraft/ay-
info/software/CAST_Papers.htm. Visited on 6 July
2004.

[13] Knight, J.; D. Evans, and J Offutt, Object
Oriented Programming in Safety-Critical Software.
A white paper.

Safety Code Analysis of an Embedded C++
Application, Proceedings of the 20th International
System Safety Conference, Denver, CO, pp. 422-
429.

Clarification of DO-178B “Software Considerations
in Airborne Systems and Equipment Certification”,
RTCA/DO-248B, Washington, D. C.

[16] Information Processing Ltd., “Advanced
Coverage Metrics for Object-Oriented Software”,
Visited on 6 July 2004.

[17] Chilenski, John Joseph, Thomas C.
Timberlake, and John M. Masalskis, November
2002, Issues Concerning the Structural Coverage of
Object Oriented Software, DOT/FAA/AR-02/113.

[18] Alexander, Roger T., September/ October
2001, “Improving the Quality of Object-Oriented

[19] Binder, Robert V., 2000, Testing Object-
Oriented Systems, Addison-Wesley, Reading, MA.

Software Engineering, Addison-Wesley, Boston,
MA.