Object-oriented Software Considerations in Airborne Systems and Equipment Certification

Michael R. Elliott
The Boeing Company
2401 E. Wardlow Rd. MS 6052-0066
Long Beach, CA 90807, USA
+1 562 645-3355
Michael.R.Elliott2@boeing.com

Peter Heller
Airbus Operations GmbH
Kreetslag 10,
21129 Hamburg, Germany
+49 40 743 73098
Peter.Heller@airbus.com

Abstract

This is a practitioner's discussion of the production of software in airborne systems which operate in civil airspace and the changes impacting it with the introduction of DO-178C/ED-12C, the emerging standard for the development of safety-critical software in airborne systems. A focus is made on the impact of the object-oriented supplement to this document which establishes, for the first time, a standard for the use of object-oriented programming and design in this environment.

Discussion is made of the state of airworthiness certification where software is concerned, the existing standard DO-178B/ED-12B[1], its history, perceived shortcomings, existing practice and how that may change with the new standard. Additionally, an overview is given of how this supplement introduces a formal type theory basis for reducing the amount of verification an applicant for airworthiness must demonstrate in order to provide the necessary safety assurance for an airborne system.

Categories and Subject Descriptors D.1.5 [*Airborne Software*]

General Terms Airborne Software Certification, Safety-critical Software, Object-oriented programming

Keywords DO-178B, DO-178C, Safety-critical, airworthiness, DO-178, ED-12B, ED-12C

1. Introduction

The formal standard *Software Considerations in Airborne Systems and Equipment Certification*[1], known in the industry as DO-178B/ED-12B[1], is used as the means by which government certification authorities such as the FAA[1] and EASA[2] determine that aircraft and engines containing software as part of their operational capability can be granted the necessary airworthiness certification needed for operation in civil airspace. As such it is required reading by thousands of engineers worldwide who produce software for aircraft and the engines which go into those aircraft. It specifies the means by which such software is produced and verified so that airworthiness certification can be granted.

DO-178B/ED-12B[1] was produced by a group of industry practitioners whose goal was to be unknown and invisible; that is, it was fervently hoped that the flying public would go about their business – flying wherever they needed to go, whenever they wished to go there – without the slightest thought as to how safe the software on their airplane was or who had been involved in making it so. Those of us who are creating the new version of the standard strive to maintain that same goal of invisibility.

This most recent effort was completed in 1992 and therefore fails to reflect the great body of advancement in Software Engineering practice which has taken place since the mid 1980's. In late 2004 an effort was begun to produce its successor document, 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.

[1] Federal Aviation Administration (Washington, DC, USA),
`http://www.faa.gov`

[2] European Aviation Safety Agency (Cologne, Germany)
`http:// www.easa.europa.eu`

[3] RTCA, Inc. (Washington, DC, USA) is an organization which creates standards documents for the FAA
`http://www.rtca.org`

[4] The European Organization for Civil Aviation Equipment (Malakoff, France) is an organization which produces documents referred to as a means of compliance for European Technical Standard Orders
`http://www.eurocae.net`

1.1 Object-oriented Technology

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

> *To date, few airborne computer systems in civil aviation have been implemented using OOT. Although OOT is intended to promote productivity, increase reusability of software, and improve quality, uncertainty about how to comply with certification requirements has been a key obstacle to using OOT in airborne systems.*
>
> *OOTiA[2], 2004*

The mission of this subgroup was to address the needs of the software practitioner in creating object-oriented software for airborne systems, a practice widely seen in the community as being prohibitively difficult under the existing standard. To this end, changes were made to the core document of DO-178B/ED-12B[1] to help facilitate this effort and 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 the production of airborne software and the certification authorities in approving it as part of a certification effort.

1.2 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 October 29, 2009. It is singularly appropriate that the day the last ever OOPSLA 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.

2. Background

Initially, software was viewed as being a way of inexpensively extending 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.

2.1 DO-178

Created to provide a basis for communication between applicants and certification authorities, this initial effort at a standard for software development in airborne systems was a set of best practices. It required applicants to meet “the intent” of DO-178 without giving specific objectives to be achieved or any significant degree of guidance as to how to meet the intent of the document. It did, however, introduce a three tiered system of software criticality – critical, essential and non-essential – and set the level of verification to reflect the criticality level. Additionally, it provided a linkage between software verification and FAA documents such as Federal Aviation Regulations and Technical Standard Orders.

2.2 DO-178A

After the initial experience with certification using DO-178, there was a consensus that a revision was needed. SC-152 of RTCA created DO-178A in 1985, and it turned out to be quite different from DO-178. It introduced rigorous requirements on software process (based on the waterfall method), software production and quite stringent requirements to provide process documentation and history. Certification artifacts were, however, frequently misinterpreted by applicants and certification authorities sometimes causing entire software development efforts to be abandoned. In general, the knowledge of why the certification requirements existed and the purpose of the requirements failed to be understood or appreciated[3].

2.3 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, producing equipment subject to certification efforts. Lack of experience, documentation and understanding of the basis for satisfying DO-178A brought about a desire for an improved standard. In 1992 this became DO-178B[1], developed in cooperation with EUROCAE as ED-12B[1] by SC-167 and WG-12.

This updated document made many fundmental changes to its predecessor. Salient among these was the introduction of software criticality levels A through E replacing the critcal, essential and non-essential previously used. A strong emphasis was placed on requirements-based testing, which was seen as a more effective way of verification 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.

2.4 OOTiA

During the eight years after the release of DO-178B/ED-12B[1] and its adoption by industry, concern was expressed that more modern software practices were difficult to employ using that standard. In 2000, the FAA responded to this concern by contacting the representatives of several key companies, Boeing, BF Goodrich and others, to produce an analysis of what it would take to adapt object oriented software procedures to the needs of airworthiness certification.

This process was later opened up to industry in general and workshops were held in order to produce position papers which, it was hoped, would evolve into a best practices

[5] Information paper

guide, an FAA Advisory Circular or rolled into the not yet begun DO-178C effort. Meetings were held by the FAA and NASA[6] which eventually resulted in the FAA publishing the four volume *Handbook for Object-Oriented Technology in Aviation (OOTiA)*[2]. This document was never intended to contain objectives or guidance for practitioners and certification authorities, only to contain a set of suggestions as to best practices and warnings about problematic situations.

By 2005, the FAA had decided that it would no longer maintain sponsorship of OOTiA[2] 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[2] over to the nascent SC-205 to use as input to the creation of an object oriented supplement to the new standard.

2.5 Rationale

The views of a number of stakeholders, including certification authorities, airframe manufacturers and equipment suppliers, were taken into account in the creation of DO-178B/-ED-12B[1]. A basic tenet of this document was that it be written as much as possible to be requirements oriented; that is, to try to make the document about what has to be achieved rather than how to go about achieving it. A fundamental rationale for this was to try 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, to be utilized by the applicant in creating airborne software and the certification authorities in judging its 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.

2.6 Software certification

There is a perception among those entering this field, or wishing to enter this field, that software is somehow "certifiable" for airworthiness. This may come about through simple observation of the title of DO-178B/ED-12B[1] *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 UK, 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, etc., that is the subject of the airworthiness certification effort – not that of the actual software involved.

[6] National Aeronautics and Space Administration (Washington, DC, USA), `http://www.nasa.gov`

As a result of this, it is not possible to, for example, 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 that vendor may well do, and typically charge a substantial fee for, is to provide the requirements, source traceability, requirements-based tests and test results for a particular software component. This documentation can then be submitted to the certification authority as part of the applicant's request for airworthiness certification of an engine, aircraft, etc.

2.7 Software production process

DO-178B/ED-12B[1] is widely perceived as being process heavy; that is, that it imposes 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 which is to be considered as part of the certification effort. Although this has been widely seen as a very expensive activity, almost twenty 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 formal airworthiness certification effort as its flights through civil airspace fall under a different authority – but it serves to illustrate that spectacular disaster certainly can come about through software coding errors.

Nonetheless, it is still widely seen in the airborne software industry that DO-178B/ED-12B[1] makes using less process-heavy techniques such as model based development, formal methods and object-oriented programming difficult to use when certification aspects are 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 promised reduction in cost, errors and time to market.

3. Rationale for change

The answer to the question given above is that the cost of doing things the old fashioned way is becoming prohibitive. It now costs hundreds of millions of dollars to achieve airworthiness certification for a new large aircraft. 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 means by which 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 1980's, 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 an eventual certification effort. This, obviously, has an impact on cost.

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

In the 1980's, the desire was to use testing to achieve safety goals. In particular an objective is *"to demonstrate with a high degree of confidence that errors which could lead to unacceptable failure conditions, as determined by the safety assessment process, have been removed"*[1]. 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, that exhaustive testing of software 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 towards more realistic goals of reaching a reasonable level of confidence that the software is correct, safe and useful rather than that it is completely error free.

3.1 Terms of Reference

This thought process culminated in a desire to:

- Modify DO-178B/ED-12B[1] to become DO-178C/ED-12C with a minimum of changes to the core document.
- Consider the economic impact relative to system certification without compromising system safety.
- Address clear errors and inconsistencies in DO-18B/ED-12B[1].
- Provide recommendations (guidelines) that provide example solutions to expected problems to aid both practitioners and certification authorities in achieving the objectives.
- Provide supplements to amplify and expand objectives, guidance and guidelines for technology specific or methodology specific areas of interest.

Additionally, the desire to change the existing document was reinforced by the observation that the difference between the terms *guidelines* and *guidance* is not only not readily understood in English but apparently they don't translate at all into French[7] as different words. *Guidelines* then became *recommendations*.

[7] ED-12B, the EUROCAE-produced version of the document, contains both English and French language versions

4. What's new?

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 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, provide any new objectives which were deemed beneficial to airborne safety and to provide guidance and recommendations as to how to achieve those objectives.

4.1 OOTiA, CAST, FAA and EASA

One of the initial responsibilities of the subgroup was to address all issues raised in OOTiA[2] and either incorporate them into the supplement or decide they were either inapplicable or unfounded. IP 508 was produced by the subgroup to address each individual concern raised by OOTiA[2] and respond to that concern. Additionally, concerns about OO had been raised through CAST[8] papers, EASA CRIs[9] and FAA IPs. All these were also to be addressed by the subgroup.

4.2 Dead and deactivated code

DO-178B/ED-12B[1] disallows dead code, which is basically code which is never executed. Dead code is treated as a software error which should be eliminated. A variant on this is deactivated code which is code which might get executed for a particular configuration but for which that configuration is not the one 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-178B/ED-12B[1]. 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 at all out of place in a stack class. The particular airborne software using such a stack, however, might not actually use the *peek* method, for example. 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

[8] Certification Authority Software Team, a group of individuals representing several certification authorities, including EASA, FAA, JAA and Transport Canada

[9] Certification Review Item

standard relaxes restrictions on separately developed components and now allows this stack class to be used unmodified.

4.3 Type Theory

Early on, the subgroup decided to provide a type theoretical basis as a rationale for reducing the amount of redundant testing / 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 which involve class hierarchy design, as long as the type hierarchies in question share particular 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/ED-12B[1], so this was perceived by subgroup members as being at some risk of being rejected by the subcommittee as a whole but, in the end, it was accepted.

4.4 The Liskov Substitution Principle

This sort of type theoretical formulation initially manifested itself in the specification of the Liskov Substitution Principle[4] (LSP) as the basis for establishing that verification of the behavior of a superclass could be used as part of the verification compliance of a subclass of that superclass. The point was that only the additional behavior provided by the subclass needed to be verified for the subclass 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 succintness of this, the subgroup felt that a purely theoretical expression of this concept might place too great of a burden on the practitioners.

4.4.1 Explaining LSP

As the supplement neared completion, the feeling was expressed by members of the subgroup that we needed to provide a clearer understanding of LSP than simply providing the definition given above. The inclusion of a Frequently Asked Questions (FAQ) section in the supplement provided a less structured environment into which we could place a question (along with its answer) which we presume will be frequently asked, essentially: *What's the deal with the Liskov Substitution Principle and why should I care?*.

We could have reiterated the concept and continued to claim that it was a good thing – which is true – but probably doesn'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, we proposed showing how not following LSP could lead to problematic behavior. DO-178C/ED-12C is, fundamentally, a document about software safety, after all, so this approach was seen as reasonable.

4.4.2 Creating a counter example

For purposes of the supplement's FAQ, the following situation was proposed: There exists a conceptually abstract hardware speed controller which can be instantiated with the necessary behavior to reflect the actual hardware of multiple 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, it was felt that this example was something that practitioners would see as at least 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.

4.4.3 Preconditions, post conditions 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 to the hardware 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. The base class makes available a means to adjust the speed by giving a speed increment to an adjust speed method (to use the Java terminology). This adjust speed method would have as its post condition that when given a positive, non-zero argument, the speed attribute of the object has increased; that is, it must be non-zero.

4.4.4 Time to divide by zero

Based on this post condition and invariant, a method *time to go*, taking a distance argument, will return the time value (in whatever units are convenient) it takes to traverse that distance. This ultimately reduces to dividing the given distance by the object's current speed attribute then changing to whatever units are convenient.

The situation as outlined above represents a valid use of LSP. Any desired number of subclasses 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, we introduced an *auto controller*, a subclass of speed controller designed to control a fundamentally different type of hardware, one which is given a desired speed which it will then seek to reach and maintain.

Listing 1. Violation of LSP – Java

```java
/////////////////////////////////////////////////////////////////////
// Base class which implements a speed controller
class SpeedController {

   public int getSpeed() {
      return speed;
   }

   public void adjustSpeed( int increment ) {
      speed += increment;
      //code which tells controller hardware the speed increment
      //post condition: speed augmented by 'increment'
   }

   // Return the time to traverse the given distance at the current speed
   public int timeToGo( int distance ) {
      // here we expect that getSpeed() returns non zero value
      return distance / getSpeed();
   }
   protected int speed = 0;
}
/////////////////////////////////////////////////////////////////////
// Subclass of Controller which violates LSP.
// It uses setDesiredSpeed() rather than adjustSpeed() to change the speed.
// The now LSP-broken adjustSpeed() does nothing, ignoring its post condition.
class AutomaticSpeedController extends SpeedController {

   public void setDesiredSpeed( int val ) {
      desiredSpeed = val;
      // Code which tells hardware what the desired speed is
   }

   @Override
   public void adjustSpeed( int increment ) {
      //do nothing
      //post condition: speed doesn't change
   }
   private int desiredSpeed = 0;
}
/////////////////////////////////////////////////////////////////////
public class Main {
   public static void main( String[] args ) {
      final SpeedController controller =
         //  new SpeedController(); // This substitution would have been fine.
             new AutomaticSpeedController();  // This substitution violates LSP
      // Taking the traditional view, the speed is incremented by 2 units
      controller.adjustSpeed( 2 );
      // Expected post condition: speed is non-zero - since we just changed it
      System.out.println( "Time to go: " + controller.timeToGo( 5 ) );
      // Exception thrown only for instance of AutomaticSpeedController
   }
}
```

Listing 2. Violation of LSP – C++[10]

```
#include <stdio.h>
///////////////////////////////////////////////////////////////////
class Controller {
public:
    int Speed() {
        return speed;
        }
    virtual void adjustSpeed( int increment ) {
        if (speed + increment > 0)
            speed += increment;
        }
    //post condition: speed augmented by 'increment'
    Controller() {
        speed = 0;
        }
private:
    int speed;
};
///////////////////////////////////////////////////////////////////
//routine relying on adjustSpeed post condition
int computeTimeToGo(Controller* controller, int distance) {
    controller->adjustSpeed( 3 );
    // here we expect that controller.Speed() is non-zero
    return distance / controller->Speed();
}
///////////////////////////////////////////////////////////////////
//derived class violating LSP on adjustSpeed
class AutoController : public Controller {
public:
    int DesiredSpeed();

    void setDesiredSpeed( int val);

    virtual void adjustSpeed( int increment ) {} //does nothing
    //post condition: speed doesn't change

    AutoController() {
        desiredSpeed = 1;
    }
private:
    int desiredSpeed;
};
///////////////////////////////////////////////////////////////////
int main(int argc, char** argv) {
    Controller* controller = new AutoController();
    int time = computeTimeToGo(controller, 5);
    printf( "Time: %d\n", time );
    return 0;
}
```

[10] Contributed by Rob Morris and Thomas Bleichner

Listing 3. Violation of LSP – Ada (1 of 2)[11]

```ada
------------------------------------------------------------------------
--  Basic speed controller definition
package Speed1 is
   subtype Speed_Type  is Integer range 0 .. 200;
   subtype Speed_Delta is Integer range -5 .. +5;

   type Controller is tagged private;
   function Speed (This : Controller) return Speed_Type;
   procedure Adjust_Speed (This : in out Controller; Increment : Speed_Delta);
   pragma Postcondition (This.Speed = This'Old.Speed + Increment);
private
   type Controller is tagged record
      Actual_Speed : Speed_Type := 0;
   end record;
end Speed1;
------------------------------------------------------------------------
package body Speed1 is
   function Speed (This : Controller) return Speed_Type is
   begin
      return This.Actual_Speed;
   end Speed;

   procedure Adjust_Speed
     (This : in out Controller; Increment : Speed_Delta) is
   begin
      This.Actual_Speed := This.Actual_Speed + Increment;
   end Adjust_Speed;
end Speed1;
------------------------------------------------------------------------
--  routine relying on adjustSpeed post condition
with Speed1; use Speed1;
procedure Compute_Time_To_Go (C : in out Controller'Class;
   Distance : Integer; Time_To_Go : out Integer) is
begin
   C.Adjust_Speed (3);
   --  Controller's Adjust_Speed guarantees that C.Speed /= 0
   --  but we have a divide by 0 here if C is an Automatic_Speed_Controller
   Time_To_Go := Distance / C.Speed;
end Compute_Time_To_Go;
```

[11] Contributed by Cyrille Comar

Listing 4. Violation of LSP – Ada (2 of 2)

```
-----------------------------------------------------------------
-- Derived class violating LSP on adjustSpeed
with Speed1; use Speed1;
package Speed2 is
   type Auto_Controller is new Controller with private;

   function Desired_Speed (This : Auto_Controller) return Speed_Type;

   procedure Set_Desired_Speed
     (This : in out Auto_Controller; Val : Speed_Type);
   pragma Postcondition (This.Desired_Speed = Val);

   overriding procedure Adjust_Speed
     (This : in out Auto_Controller; Increment : Speed_Delta);
   pragma Postcondition (This'Old.Speed = This.Speed);

private
   type Auto_Controller is new Controller with record
      Desired_Speed : Speed_Type := 0;
   end record;
end Speed2;
-----------------------------------------------------------------
package body Speed2 is
   function Desired_Speed (This : Auto_Controller) return Speed_Type is
   begin
      return This.Desired_Speed;
   end Desired_Speed;

   procedure Set_Desired_Speed
     (This : in out Auto_Controller; Val : Speed_Type) is
   begin
      This.Desired_Speed := Val;
   end Set_Desired_Speed;

   procedure Adjust_Speed
     (This : in out Auto_Controller; Increment : Speed_Delta) is
   begin
      null;
   end Adjust_Speed;
end Speed2;
-----------------------------------------------------------------
-- program raises an exception due to violation of LSP
with Speed2; use Speed2;
with Compute_Time_To_Go;
procedure Main is
   Res : Integer;
   Ctrl : Auto_Controller;
begin
   Compute_Time_To_Go (Ctrl, 5, Res);
end Main;
```

4.4.5 Breaking LSP

Since this new auto controller class no longer needs (and, indeed, has no use for) the adjust speed by a speed increment method, its implementation of the method (which, of course, it must implement) did 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 we strove to make was that in having the auto controller's *adjust speed* method do nothing, the post condition would be violated, since invoking the *adjust speed* method on an object with zero speed would fail to make the actual speed attribute non-zero.

While this is a moderately contrived example, we did manage to create a situation where we could have an unexpected exception thrown due to violation of a post condition which came about through failure to ensure the Liskov Substitution Principle was maintained throughout the hierarchy. In order make this more concrete, example code was created in Java, C++ and Ada – see listings 1, 2, 3 and 4.

Originally the code examples we created used floating point values but once we actually ran the code we wrote, we were surprised to find that none of our examples actually threw the expected division by zero exception since all the implementations simply returned infinity when the division by zero occurred. We therefore had to increase the contrivance level a bit more by making all the values integers so we could actually cause the exception we wanted to have thrown, and possibly 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.

4.5 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, brought about 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, that developers could make type consistency determinations with well-defined boundaries facilitating the incorporation of separately (and often externally) developed class hierarchies.

4.6 Taxonomy of polymorphism

Although not really object-oriented in nature – the charter of the subgroup being essentially all coding issues – the notion of polymorphism was 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 was discussed as being divided into parametric and inclusion polymorphism, coercion and overloading, respectively. Again this was done by the subgroup with some apprehension but we felt that at least introducing the vocabulary would provide additional means of clarifying situations where polymorphism is used as well as a common vocabulary for practitioners and certification authorities.

A similar philosophy guided our decision to discuss closures as a means of specifying behavior; that is, if we introduce the terms in the supplement an applicant can use the concept with an expectation that the certification authority will at least be on the same page.

4.7 Resource management

One area in which the subgroup 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 permitted for the first time. Garbage collection in real-time systems is a subject where 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 we encountered with this view was the inability of any of its proponents (at least the ones with whom we communicated) to express just how complex "too complex" is or even how such complexity should be measured. We found it 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 – was unusable due to some unspecified and undefinable algorithmic complexity in all garbage collectors, we did indeed recognize the potential for heap memory exhaustion in an airborne system and provided 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.

5. Conclusions

Generally, the subgroup took the view that it should strive to remain language and technology neutral but to 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 really needed to be careful and conservative in the resulting document.

This was also done with the knowledge the real-time, avionics and safety-critical communities are quite reluctant to introduce new concepts (garbage collection and run-time polymorphism, for example) so the subgroup needed to provide a basis for acceptability of such ideas to that community by at least furnishing a theoretical base for discussion as well as an analysis of the perceived risks of a given approach as well as recommendations as to how to mitigate those risks.

5.1 DO-178C/ED-12C approval

Although the object-oriented supplement has been approved as part of the overall DO-178C/ED-12C document (along with supplements covering tool qualification and the use of formal methods), the overall core document along with one additional supplement have not yet completed the approval process. In particular, the aviation community has expressed a desire to see a supplement address the means by which model based development can be utilized in airborne software. Subgroup 4 of SC-205/WG-71 has been developing this supplement from the outset and it is hoped that it will be completed in time to become a part of DO-178C/ED-12C.

The approval of the entire DO-178C/ED-12C document (with or without the model based supplement) is expected in the second week of November, 2010, with actual publication by RTCA and EUROCAE shortly thereafter, at which time it will become available as the standard for the determination of airworthiness for airborne systems and and equipment containing software. As more and more practitioners consider this supplement as the means to achieve airworthiness consideration for their products, we – the members of subgroup 5 – will slowly, but eventually, become known as "those idiots" as in the phrase: *What were those idiots thinking of when they wrote* .... As long as we remain invisible idiots, we'll be OK with that.

Acknowledgments

The other active members of Subgroup 5 of SC-205/WG-71 who actually wrote and edited the OO supplement, 2005 – 2010:

Thomas Bleichner	Rohde & Schwarz
Jan-Hendrik Boelens	Eurocopter
Jim Chelini	Verocel
John Chilenski	Boeing
Cyrille Comar	AdaCore
Branimir Dulic	Transport Canada
Mark Gulick	Solers
James Hunt	aicas
John Jorgensen	Universal Avionics Systems
Varun Malik	Hamilton Sundstrand
Greg Millican	Honeywell
John Minihan	Resource Group
Rob Morris	CDL Systems
Sven Nordhoff	SQS
Bill St. Clair	LDRA
Robin Sova	FAA
Elroy Wiens	Cessna

References

[1] Special Committee 167 / Working Group 12 of RTCA and EUROCAE. *DO-178B/ED-12B – Software Considerations in Airborne Systems and Equipment Certification.* RTCA and EUROCAE, Washington, D.C., USA and Malakoff, France, Dec. 1992.

[2] Federal Aviation Administration (FAA). *Handbook for Object-Oriented Technology in Aviation (OOTiA).* Federal Aviation Administration (FAA), Washington, D.C., USA, Oct. 2004.

[3] L. Johnson. DO-178B, "Software Considerations in Airborne Systems and Equipment Certification", *Crosstalk, The Journal of Defense Software Engineering*, 11(10), Oct. 1998.

[4] B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6): 1811–1841, Nov. 1994.