Safety, Security, and Object-Oriented Programming

Franco Gasperoni
gasperoni@adacore.com
AdaCore, 8 rue de Milan, 75009 Paris, France

Abstract

When safety-critical software malfunctions people lives are in danger. When security-critical software is cracked national security or economic activity may be at risk.

As more and more software embraces object-oriented programming (OOP) safety-critical and security-critical projects feel compelled to use object-orientation. But what are the guarantees of OOP in terms of safety and security? Are the design goals of OOP aligned with those of safe and secure software (S^3) systems?

In the following sections we look at key OOP aspects and analyze some of the hazards they introduce with respect to S^3 and outline a possible way of addressing these vulnerabilities. Specifically, after a quick overview of OOP in section 2, section 3 deals with inheritance and shows some of its hazards in terms of S^3 along with possible remedies. Section 4 focuses on dynamic binding and suggests a safer and more secure implementation than what is conventionally done. Finally, section 5 looks at testing programs with dynamic binding.

1 Introduction

Software is typically organized in one of two ways: structured or object-oriented (OO). In the structured organization the program is broken down and organized around its subprograms. In the OO organization the application is subdivided and arranged around its data types (also known as classes).

Even though the DO-178B safety-critical standard used for airborne software [DO-178B] does not impose a specific way of organizing computer software, in practice, most DO-178B software has used a structured organization. Likewise, the Common Criteria security standard [ISO15408] does not make specific recommendations on how a secure software program should be organized (structured vs. OO).

One of the objectives [DO-178C], the future revision of the DO-178B standard, is to address the use of object-oriented techniques and their associated development processes in the avionics industry. A preliminary document [OOTiA] provides a comprehensive analysis on the safety concerns associated with OO techniques in the context of DO-178B.

2 Object-Oriented Programming (OOP)

Today all modern programming languages such as C++, Java, or Ada (in its 95 or 2005 incarnation [Ada2005]), allow developers to make use of OOP. To increase its use in the

construction of S^3 programs, the use of OOP must be balanced with the need to retain existing confidence in software safety and security.

The key shift from structured programming (SP) techniques to OOP is the ability to break up SP's centralized code organization into OOP's distributed one. SP's data and code is centralized around a small set of types and subprograms while OOP's data and code are distributed around a forest of incrementally built data types. OOP's organization simplifies code maintenance and evolutions when new data types are built incrementally from existing ones. In addition, OOP provides a good framework for libraries where software components can be adapted and modified.

To achieve its distributed organization OOP rests on three important concepts:

(1) Field inheritance & extension: the ability to create a new class from existing ones (the parents) and extend this new class with additional data fields.

(2) Method inheritance & extension: the inheritance for a class of the methods declared for its parents with the ability to override them and add new ones.

(3) Dynamic binding: the ability to link at runtime a method call with a subprogram based on the class of the object on which the method is invoked.

We have excluded from this list important concepts such as encapsulation and privacy which are shared by both SP and OOP, as demonstrated by structured programming languages such as Ada 83 and Modula 2.

3 Inheritance and S^3

3.1 Attribute Inheritance & Extension

When extending a class with data fields the question arises of name clashes, that is what if the added data field has the same name as a field in a parent class? In some OOP languages, such as Ada 95 and Ada 2005, this problem cannot arise because this is forbidden at the language level. In other languages such as C++ or Java this problem does arise and is a potential S^3 hazard as shown in the following example.

<table>
<tr><th>C++</th><th>Java</th></tr>
<tr><td><pre>class A {
  protected: int key;
  public: A() { key = 99; }
};
class B : public A { };
class C : public B {
  public: void use_key() {
    cout << "key=" << key << "\n";
  }
};</pre></td><td><pre>class A {
  protected int key;
  public A() { key = 99; }
}
class B extends A { }
class C extends B {
  public void use_key() {
    System.out.println ("key=" + key);
  }

  public</pre></td></tr>
</table>

`int main () {`
`  C* object = new C();`
`  object->use_key();`
`  return 1;`
`}`
	`  static void main (String[] args) {`
	`    C object = new C();`
	`    object.use_key();`
	`  }`
	`}`

The output from both programs is "key=99". If later on another developer adds a field also named **key** to class **B** as shown below:

C++	Java
`class B: public A {protected: int key;};`	`class B extends A {protected int key; }`

then the behavior of methods such as **use_key()** changes implicitly. This implicit change is dangerous since it is legal in both C++ and Java and no warnings are typically emitted by the compiler. If the derivation chain between **A**, **B**, and **C** is long the side-effect caused by the addition of **key** in **B** will go undetected.

One may wonder why such a rule was put in C++ and Java. The reason stems from the need to provide freedom to OO library manufacturers in such languages. This is a legitimate concern. If the language did not allow the same field name in a parent class and a child class, adding a new field to an existing parent class could potentially break any child class already using those names.

This is a good example of a legitimate situation in a day-to-day OO context which turns into an S^3 hazard.

As we previously mentioned the rules of Ada 95 or Ada 2005 do not allow the above scenario because Ada's first target is S^3 applications. A possible fix in other languages would be to create qualified tools (in the DO-178B sense of the term) that would check for such a hazard in an S^3 context.

3.2 Method Inheritance and Extension

As mentioned in 2(2), fundamental to all OOP languages is the ability to override inherited methods and add new ones. But what happens if in doing so the programmer misspells the name of the method to override as shown below?

C++	Ada 95 / Ada 2005

`class A {`
`public: virtual`
`  void finalize_xyz();`
`};`
`class B : public A {`
`public: virtual`
`  void finalise_xyz();`
`};`
	`type A is tagged`
	`  null record;`
	`procedure finalize_xyz (Obj: A);`
	
	`type B is new A with`
	`  null record;`
	`procedure finalise_xyz (Obj: B);`

In the above example the developer of class **A** has used US-spelling for the word "finalize", whereas the developer of class **B** has used British-spelling and written "finalise". This means that method **finalize_xyz** (US spelling) has not been overridden in class **B** and all dynamically bound calls to such a method for objects of class **B** will invoke **A**'s implementation.

One may claim that the above typo would be caught as part of the testing procedures of certification protocols such as DO-178B since method **finalise_xyz** (British spelling) will come out as never being invoked during testing. Perhaps. But may be the programmer of class **B** needed to make explicit calls to **finalise_xyz** (British spelling) thereby making the above typo invisible to conventional 100% source-coverage (and MC/DC) testing. This is true, in particular, if the testing plan did not include a test case with a dynamic binding call to **finalize_xyz** (US spelling) with an object of class **B**.

There are several lines of defense against this type of hazard. We will mention the one recently introduced by Ada 2005. In Ada 2005 the developer can prefix each method with the keywords **overriding** or **not overriding**. These signal to the compiler the programmer's intention and allow the compiler to double check that these intentions match reality.

For instance, the code below on the left hand side compiles since the programmer has clearly stated that the method with the British spelling should not override the US spelled one. On the other hand the code on the right hand side will not compile since the British spelled method does not override any method in **A** and all that is left to do is to fix the spelling typo.

<table>
<tr><th>Ada 2005 – correct code</th><th>Ada 2005 – incorrect code</th></tr>
<tr><td><pre>type A is tagged null record;
procedure finalize_xyz (Obj: A);

type B is new A with null record;
not overriding
procedure finalise_xyz (Obj: B);</pre></td><td><pre>type A is tagged null record;
procedure finalize_xyz (Obj: A);

type B is new A with null record;
overriding
procedure finalise_xyz (Obj: B);</pre></td></tr>
</table>

By requesting at project level that all methods in subclasses be prefixed with the keywords **overriding** or **not overriding** (which can be checked by a qualified tool - in the DO-178B sense of the term) we defend against this potential hazard.

4 Dynamic Binding and OOP

A tricky and crucial element of OOP is dynamic binding. The use of dynamic binding is so far discouraged in safety-critical applications that have to be certified according to DO-178B. This is detailed in FAQ #32 of the DO-178B clarification document [DO-248B]. This is partly due to dynamic dispatching, the technique used to implement dynamic binding in compiled OO languages. In the following sections we look at dynamic

dispatching, its S^3 downsides and suggest a safer and more secure alternative to implement dynamic binding.

4.1 Dynamic Dispatching

Consider the following:

<table>
<tr><th>C++</th><th>Ada 95 / Ada 2005</th></tr>
<tr><td><pre>class A {
public:
  virtual void method ();
};
class B : public A {
public:
  void method ();
};
class C : public B {
public:
  void method ();
};
void dynamic
  (A *object)
{
  object->method ();
}</pre></td><td><pre>type A is tagged
  null record;
procedure method (Obj: A);

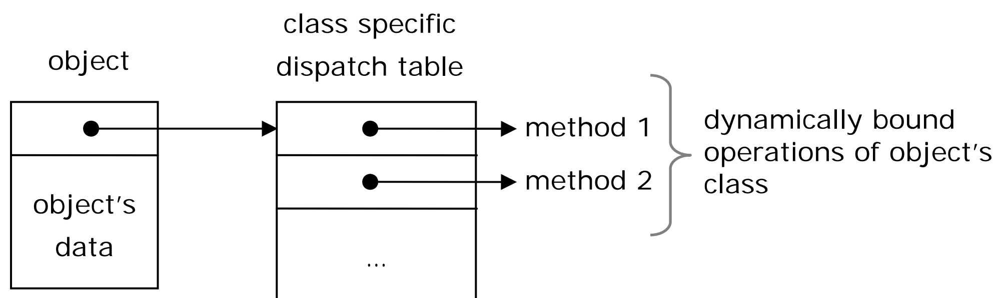
type B is new A with
  null record;
procedure method (Obj: B);

type C is new B with
  null record;
procedure method (Obj: C);

procedure dynamic
  (object : A'class) is
begin
  object.method; -- Ada 2005 only
  method (object);-- Ada 95 / Ada 2005
end call;</pre></td></tr>
</table>

In procedure **dynamic** the call to **method** has dynamic binding: the routine called at runtime will depend on **object**'s exact class. Specifically, the call to **method** will call **A**'s method, **B**'s method, or **C**'s method, according to **object**'s class.

The way compilers typically implement the above is with dynamic dispatching. This consists, in the case of simple inheritance, in adding a new field known as the vtable pointer in C++ or the tag in Ada 95/2005 to the object. This new field references a table (the dispatch table) of pointers to the dynamically bound operations defined for the object's class (see picture below).



To make a dispatching call one follows the tag/vtable pointer, indexes the resulting table by the offset corresponding to the dynamically bound method (a constant known at compile time) and makes an indirect call to the resulting method pointer.

Note that this way of implementing dynamic binding is very flexible as it allows the addition of further classes on the outskirts of the inheritance graph without the need to recompile existing classes and their clients.

4.2 Problems with Dynamic Dispatching

Dynamic dispatching has several safety and security problems, namely:

(1) Initialization: how can we prove that dispatch tables and tag/vtable fields are initialized correctly?

(2) Modification: how can we prove that dispatch table and tag/vtable values are not updated maliciously or unintentionally during the execution of a program?

Because the implementation of dynamic dispatching is invisible at source level a further practical issue arises:

(3) Tools: how can we use source-based tools in the presence of dynamic dispatching for tasks such as code coverage?

Demonstrating correct dispatch-table initialization at object-level is akin to the problem of showing that the linker produces a correct executable from the object files it links. This problem is part of the control coupling objective in DO-178B parlance and is addressed by either verifying the correctness of the final result by hand of by employing a qualified tool that performs such verification [VerOLink].

If one can ROM dispatch tables or place them in OS-guarded read-only memory the need to verify that dispatch tables are unchanged during a program's execution disappears. Unfortunately, an object's tag/vtable field cannot typically be placed into read-only memory and the costs of demonstrating at object-code level that these fields are unchanged during a program's execution remain. Such modifications could occur because of a rogue pointer or buffer overflow in assembly or C/C++ code that may be part of the S^3 application or by other accidental or malicious means.

4.3 Fixing the Problems of Dynamic Dispatching

A possible solution to address the problems described in the previous section would be to use a special pre and post-processor integrated in the compile/bind/link cycle to transform every dynamically bound call into case statements. This transformation, done partly at compile-time and partly just before link-time, is illustrated below for C++ and Ada 95 / Ada 2005 for the code example given in section 4.1. Items in the source added during the pre-processing stage (prior to compilation) have a gray background. Items removed from the source during pre-processing are commented out, italicized, and have a gray background.

C++	Ada 95 / Ada 2005
	`pragma Restrictions (No_Dispatching_Calls);`

`typedef int Class_Id;`
`extern const Class_Id a_id;`
	`type Class_Id is new Integer;`
	`a_id : constant Class_Id;`
	`pragma import (Ada, a_id);`

`class A {`
`public: Class_Id ID;`
`  /* virtual */`
`  void method();`
`};`
	`type A is tagged record`
	`  ID : Class_Id; --could be a discriminant`
	`end record;`
	`procedure method (Obj: A);`

`extern const Class_Id b_id;`	`b_id : constant Class_Id;` `pragma import (Ada, b_id);`

`class B : public A {`
`public: void method ();`
`};`
	`type B is new A with null record;`
	`procedure method (Obj: B);`

`extern const Class_Id c_id;`	`c_id : constant Class_Id;` `pragma import (Ada, c_id);`

`class C : public B {`
`public: void method ();`
`};`
	`type C is new B with null record;`
	`procedure method (Obj: C);`

`B some_object;` `some_object.ID = b_id;`	`some_object : B;` `some_object.ID := b_id;`

The pre-processing part of the code transformation involves telling the compiler that there are no more dispatching calls. This is achieved by removing the keyword **virtual** in C++ and adding a restrictions pragma in Ada. The objective of this transformation is to request to the compiler that it no longer generate dynamic dispatching machinery, such as dispatch tables, implicit tag/vtable pointers inside objects, and their initialization since this would be dead data/dead code in the final executable. The pre-processing stage also introduces an explicit **ID** field used to contain the type of the object. When created, objects of type **A**, **B**, or **C**, have their **ID** field initialized by the pre-processor to **a_id**, **b_id**, or **c_id** respectively.

The second part of the pre-processing code transformation involves replacing every dynamically bound call with a statically bound one as shown below.

C++	Ada 2005

`void dynamic (A *object)`
`{`
`  find_method (object, object->ID);`
`  //  was  object->method ();`
`}`
	`procedure dynamic (object: A'class) is`
	`begin`
	`  find_method (object, object.ID);`
	`  --  was  object.method;`
	`  --  or   method (object);`
	`end call;`

The post-processing part of the code transformation is performed just before link time and involves generating the values for constants **a_id**, **b_id**, and **c_id**, as well as the body for routine **find_method** which implements dynamic binding with an explicit case statement as shown below (code for Ada 95 is the same as for Ada 2005 except for the difference in method call notation).

C++	Ada 2005

`void find_method`
`  (A *object,`
`   Class_Id ID)`
`{`
`  switch (ID) {`
`    case a : object->A::method(); break;`
`    case b : object->B::method(); break;`
`    case c : object->C::method(); break;`
`    default:`
`      find2_method (object, ID);  break;`
`  }`
`}`
	`procedure find_method`
	`  (object : A'class;`
	`   ID     : Class_Id) is`
	`begin`
	`  case ID is`
	`    when a => A (object).method;`
	`    when b => B (object).method;`
	`    when c => C (object).method;`
	`    when others =>`
	`      find2_method (object, ID);`
	`  end case;`
	`end find_method;`

In the above, the **find2_method** routine invoked in the default case would take some remedial action since it is called when the type of the object cannot be determined.

The above code transformation has several advantages:

- It is completely transparent to the programmer who can keep using dynamic binding as usual.
- It makes control flow explicit in the post-processed source without the danger of accidental or malicious modifications to dispatch tables or vtable pointers.
- It allows the use of source-based tools based on static control flow.
- If the compiler implements sibling call optimization (e.g. GCC) then the performance of the case statement implementation of dynamic binding is practically the same as for dynamic dispatching.
- The **find_method** routine can be generated on an individual call basis or on a global **method** basis thereby allowing different levels of testing granularity (more on this in section 5).

These advantages come at the cost of having to do a complete post-processing before link every time a new class is added to the program. This is unattractive in the case of general-purpose OO applications but is desirable for S^3 applications where knowing and checking the flow of control statically is important.

Note that the above scheme allows a certain degree of flexibility since unforeseen events (such as the later addition of a new class derived from **A**) could be daisy-chained through routine **find2_method** without having to recompile existing code.

5 Testing

Testing is a very important aspect of safety-critical software. In DO-178B, for instance, requirements-based testing is mandated to verify that the software behaves as expected. Requirements-based testing translates into various levels of code coverage and can be done at either source or object-code level.

One important issue when it comes to testing OO applications is how to ensure that programs containing dynamic binding are properly tested in the context of S^3 applications. In particular, should the testing campaign provide test cases exerting all possible operations at every dynamically bound call (100% dispatching coverage) or is it sufficient to test a subset of these operations, as long as all operations are called at least once in some dispatching call (100% method coverage)?

If the number of "dispatching" methods in the program is **m** and the number of dynamically bound calling points is **d** then 100% dispatching coverage could require as many as O(**m** x **d**) tests, as opposed to the O(**m** + **d**) tests needed to achieve 100% method coverage. This is a significant difference for OO programs making heavy use of dynamic binding.

5.1 When 100% Method Coverage Is Not Enough

To gain an insight on the 100% dispatching vs. 100% method coverage issue let's go back at the difference between structured programming (SP) and OOP.

In this section we look at the SP and OOP implementations of a simple alert system. The example shows that certain bugs involving crosscutting of concerns [ASPECT] would be detected by conventional testing in the SP version but not in the OOP version unless we test all method calls at certain dispatching points. This example does not prove that 100% dispatching coverage is systematically needed for S^3 applications, but it does show that 100% method coverage isn't enough.

For the purpose of the example we are to implement an alert system which is to handle alerts in a system coming from different devices (flaps, rudder, cabin pressure, etc). All handled alerts must be logged. A partial sketch of the SP and OOP versions in Ada 2005 of the core alert structures follows. The problem shown below arises in similar terms had the system been coded in C++ or Ada 95.

SP version (Ada 83/Ada 95/Ada 2005)	OOP version (Ada 2005)

<pre>**type** Device_Kind **is** (Flaps, Rudder);
type Alert (Device: Device_Kind) **is**
record
  … -- *Fields common to all alerts*
  case Device **is**
    when Flaps =>
      … -- *Flaps specific data*
    when Rudder =>
      … -- *Rudder specific data*
  end case;
end record;
 	
procedure Handle (A: Alert) **is**
begin
  CH; -- *Code common to all alerts*
  Log (A); -- *alerts are logged here*
  case A.Device **is**
    when Flaps =>
      FH; -- *Flaps specific handling*
    when Rudder =>
      RH; -- *Rudder specific handling*
  end case;
end Handle;
 	
function Get_Alert **return** Alert;
-- *Builds the alert object for the*
-- *device that issued the alert*
-- *interrupt.*</pre>
	<pre>**type** Alert **is abstract tagged record**
	  … -- *Fields common to all alerts*
	end record;
	procedure Handle (A: Alert);
	
	type Flaps_Alert **is new** Alert **with record**
	  … -- *Flaps specific data*
	end record;
	overriding procedure Handle (A: Flaps_Alert);
	
	type Rudder_Alert **is new** Alert **with record**
	  … -- *Rudder specific data*
	end record;
	overriding procedure Handle (A: Rudder_Alert);
	
	procedure Handle (A: Alert) **is**
	begin
	  CH; -- *Code common to all alerts*
	  Log (A) -- *alerts are logged here*
	end Handle;
	
	procedure Handle (A: Flaps_Alert) **is**
	begin
	  Alert (A).Handle; -- *do common processing*
	  FH; -- *Flaps specific handling*
	end Handle;
	
	procedure Handle (A: Rudder_Alert) **is**
	begin
	  -- ***BUG** forgot to call:* Alert (A).Handle;
	  RH; -- *Rudder specific handling*
	end Handle;
	
	function Get_Alert **return** Alert'Class;
	-- *Same comment as in SP version*</pre>

Function **Get_Alert** in the above code may be called by a routine which is invoked by the system when detecting an alert interrupt. Let's assume that this routine, which we will call **Process_Alert** also computes the time interval between two consecutive alerts and does something based on this time interval as shown in the following code snippet which is common to both the SP and OOP version.

Code common to both SP and OOP versions (Ada 83/Ada 95/Ada 2005)

```
function Time_Last_Alert_Handled return Ada.Real_Time.Time;
--  Time in which the last alert was handled or zero if no such alert exists

procedure Process_Alert is
  T1, T2 : Ada.Real_Time.Time;
begin
  T1 := Time_Last_Alert_Handled;
  Handle (Get_Alert); -- Can be written Get_Alert.Handle in Ada 2005
                      -- Dynamically bound call in OOP version.
                      -- Statically bound cal in SP version.
  T2 := Time_Last_Alert_Handled;
  --  Compute T2 - T1 and do something according to the time span
  …
end Process_Alert;
```

In the SP version the code to log an alert is shared by the alerts for all device kinds. As a result, had we forgotten the log alerts inside the SP version of procedure **Handle**, testing **Process_Alert** with any type of alert would demonstrate the error. This is not so in the OOP version because the common code to log alerts, which is factored inside the **Handle** operation of the root **Alert** type, needs to be called by each **Handle** routines for each different type of alert. If, as we did in this example, forgot to call the **Handle** of **Alert** in the implementation of **Handle** for **Rudder_Alert** we have a bug because we do not log rudder alerts.

Now if we look at the **Process_Alert** routine, the dynamically bound call to **Handle**:

```
Handle (Get_Alert);  -- Can be written Get_Alert.Handle in Ada 2005
```

will show the bug if and only if it is called with an **Rudder_Alert** object, thereby showing that to detect this type of error by testing we may need to test all **Handle** calls at this calling point in the OOP version.

The problem shown above is related to the issue of concerns that cut across class types [ASPECT], specifically the fact that all alerts need to be logged when they are handled. This concern can be located in a single subprogram in the SP version but not conveniently in the OOP version. Generally speaking, the style of coding where a method in a child class needs to invoke the method in the parent class is fairly frequent. It happens with constructors, destructors, finalization routines, etc. In these instances a simple 100% method coverage testing approach may not produce the same level of safety and security confidence than in the SP case.

5.2 *Dynamic Binding, Testing, and Case Statements*

In 4.3 we have shown how a dynamically bound call can be converted into a case statement to increase the safety and security of OO applications. Section 4.3 left open the issue of whether a single case statement should be generated for a whole class of dispatching operations, or whether a case statement should be generated on a call by call basis. Whatever the answer, it is worth pointing out that the conversion of dynamically bound calls to case statements allows for both 100% dispatching coverage and 100% method coverage testing.

What the previous example seems to indicate is that in certain instances it may be useful to generate case statements on a call by call basis. What are those instances is still an open question.

6 Conclusion

This paper looks at some of the issues when using OOP in S^3 applications. The paper just scratches the surface. For airborne safety-critical software the effort on DO-178C [DO-178C] is undertaking a thorough investigation of the issues. It would be beneficial if efforts from members of all S^3 communities were pulled together.

7 References

[Ada2005] Programming in Ada 2005, by John Barnes. Addison-Wesley, 2006.

[ASPECT] Special Issue on Aspect-Oriented Programming, Communications of the ACM, Volume 44 Issue 10, October 2001.

[DO-178B] Software Consideration in Airborne Systems and Equipment Certification. RTCA/DO-178B or EUROCAE/ED-12B, 1992.

[DO-178C] Forum on Software Considerations in Airborne Systems, RTCA/SC-205 and EUROCAE/WG-71. Available at http://forum.pr.erau.edu/SCAS

[DO-248B] Final Report for Clarification of DO-178B. RTCA/DO-248B or EUROCAE/ED-94B, 2001.

[ISO15408] Common Criteria for Information Technology Security Evaluation. ISO/IEC 15408:2005.

[OOTiA] Handbook for Object-Oriented Technology in Aviation (OOTiA). FAA, 2004. Available at : http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/oot

[VerOLink] http://www.verocel.com/verolink.htm