12 June 2012

Source Code as the Key Artifact in Requirement-Based Development: The Case of Ada 2012

José F. Ruiz
Senior Software Engineer
AdaCore

Cyrille Comar
Managing Director
AdaCore

Yannick Moy
Senior Software Engineer
AdaCore

Ada-Europe 2012, Stockholm

Outline

- **Safety-critical standards**
  - The case for DO-178
- **Artifacts to produce**
  - How to manage them easily
  - How to verify them
- **How to handle their interrelationship**
- **Conclusion**

Safety-Critical Software

- **What is “safety critical” software?**
  - Failure can cause loss of human life or have other catastrophic consequences
- **How does safety criticality affect software development?**
  - Regulatory agencies require compliance with certification requirements
  - Safety-related standards may apply to the finished product, to the development process, or both

DO-178: the civil avionics standard

- **"Just" reasonable development process…**
  - Planning
  - Specify requirements
  - Implement only requirements
  - Test
  - Verify tests
  - Reviews
  - Control the development process

  but …

  … now this process is checked and validated (objectives are met)

  … and all development artifact must be traceable

- **That's the certification process**

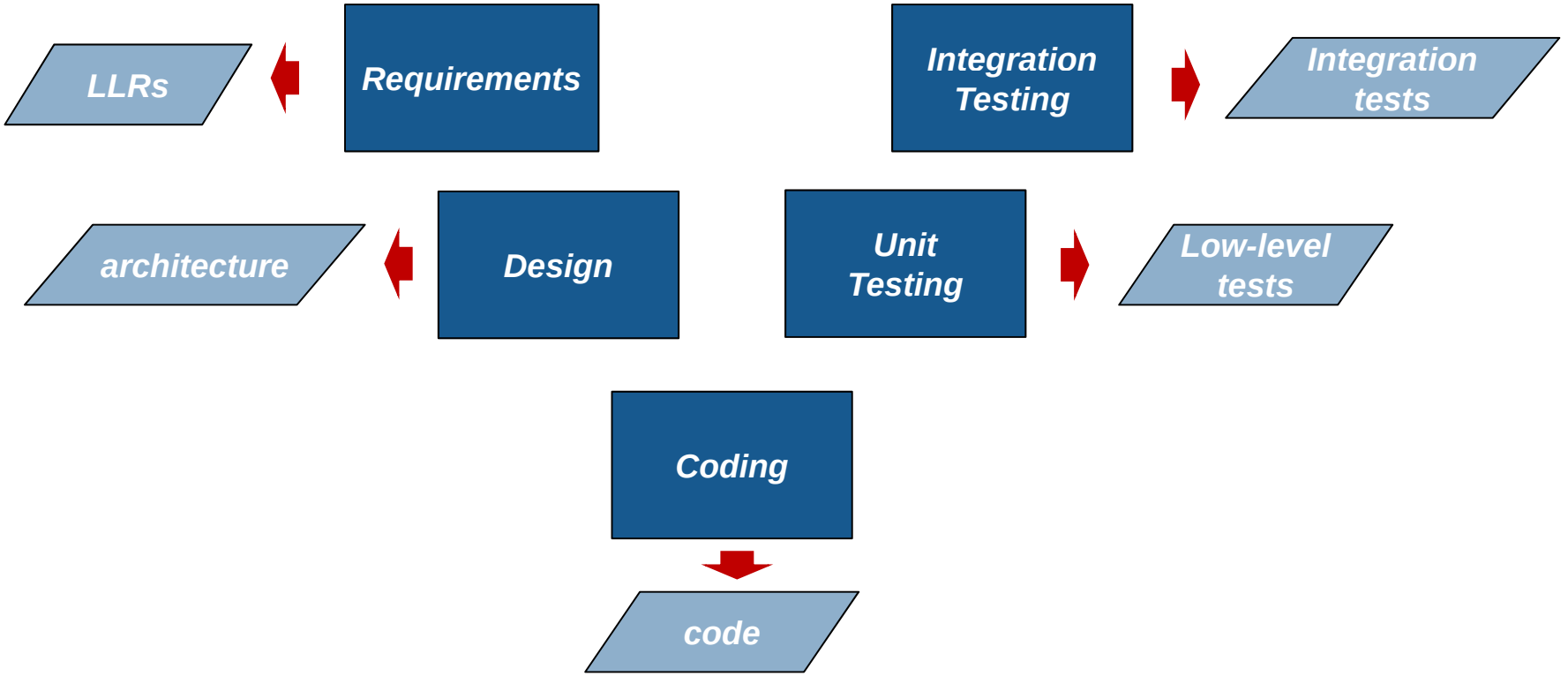
- **Certification authorities check that the process is followed**
  - All software plane flying in the civil air space must have software certified

Agility

- **Approaches to handle the evidences**
  - Traditional activity-centric

    Temporal and causal dependencies among activities
  - Artifact-centric

    Focus on what the activities produce



Goals of this presentation: to take-away

- **The goal is to centralize in Ada 2012 code the artifacts generated during development and verification**
  - Requirements
  - Architecture
  - Code
  - Test cases
  - Test procedures
  - Test results

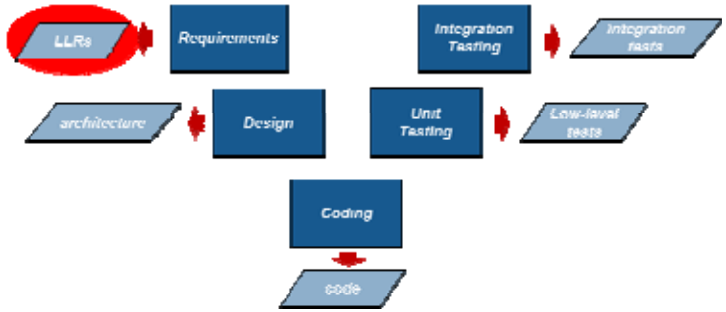
- **Traceability made easy**

Requirements

- **Software requirement process produces**
  - High-level requirements (HLRs)

    What to implement
  - Low-level requirements (LLRs)

    How to implement



- **What we propose to represent LLRs**
  - Ada 2012 pre- post- conditions

    Design-by-Contract approach
  - Informal part of the requirement can also be captured

```
package Arith is
   procedure Double (X : in out Integer) with
      Pre  => X >= Integer'First / 2 and then
              X <= Integer'Last / 2,
      Post => X = 2 * X'Old;
end Arith;
```

DO-178 objectives for requirements

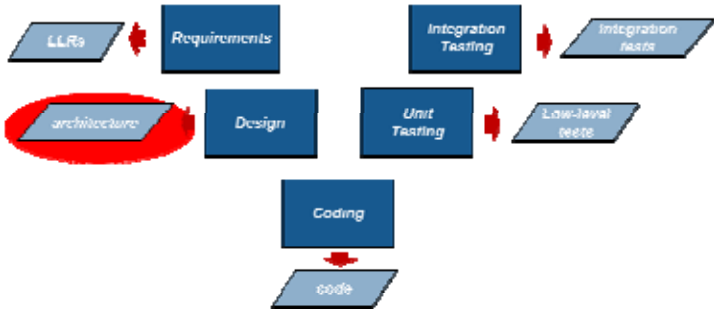
- **Accuracy and consistency**
  - Contracts defined by the static and dynamic semantics of Ada
    - Use coding standard avoiding ambiguities

      Rules such as: "*Use only short-circuit boolean operators*"

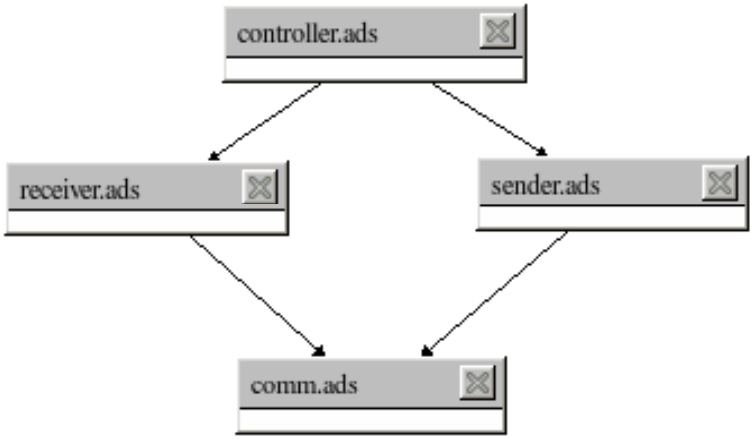
- **Verifiability**
  - Formal prove
    - Contracts translated into logical formulas that can be proved
    - Subprograms proved in isolation using callee's contracts
  - Testing
    - Contracts translated into assertions checked at execution time
  - Or a mixed approach
    - Proving what is easy to prove and test the rest

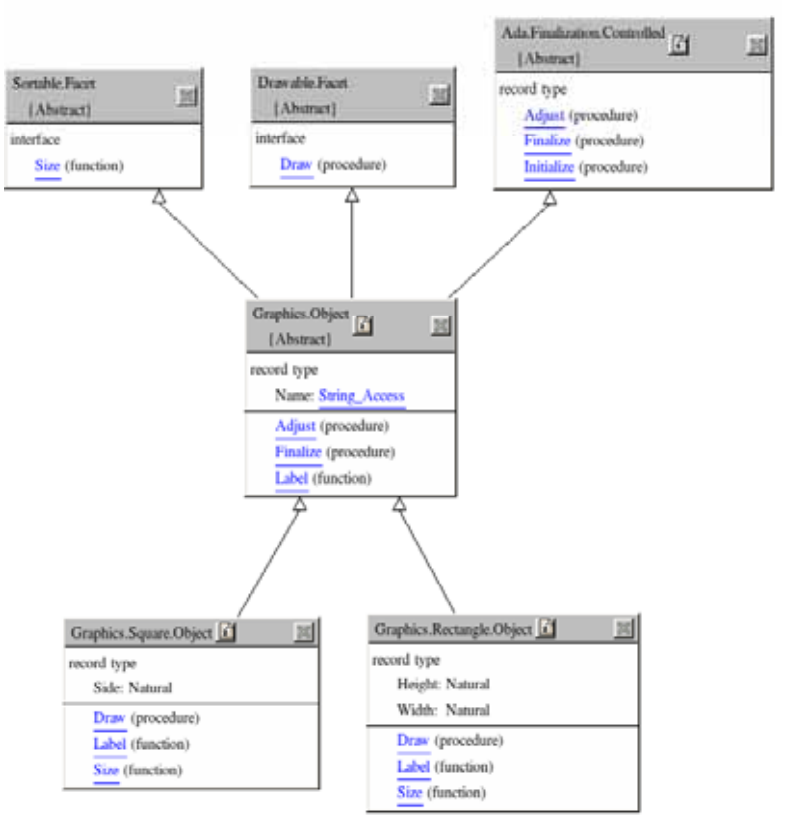
Software architecture

- **Architecture implementing the requirements**



- **We propose to use**
  - Ada package specs
    - Encapsulates components and subsystems
    - Shows their interfaces
  - With clauses and hierarchical dependencies
    - Relationships





DO-178 objectives for the software architecture

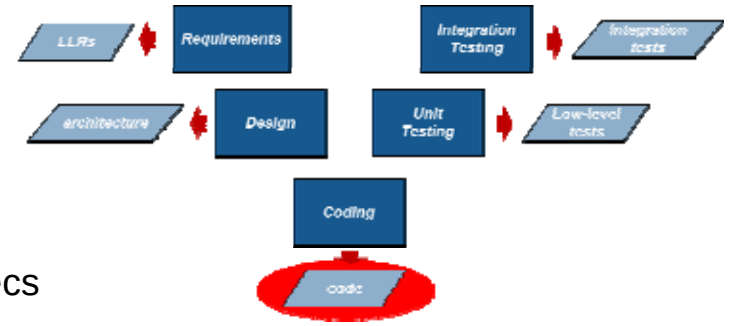
- **Consistency**
  - Data and control flow analysis
    - Ada helps
      - Visibility rules limit the scope of the analysis
      - Coding standards may restrict data and control coupling
      - Flow information in parameter mode
    - SPARK can take you much further

```
procedure Process
   (Output         : out T;
    Input1, Input2 : in  T);

--# global out Global_Output;
--#        in  Global_Input;
--# derives Output        from Input1, Input2 &
--#         Global_Output from Global_Input, Input2;
--# pre  Input1 /= 0;
--# post Output = Input2 / Input1;
```

Code

- **Code is produced by the software coding process, from**
  - Low-level requirements
  - Architecture



- **We propose to**
  - Implement the Ada bodies corresponding to the specs
    - Compliant with LLRs (contracts)

```
package Arith is
   procedure Double (X : in out Integer) with
      Pre  => X >= Integer'First / 2 and then
              X <= Integer'Last / 2,
      Post => X = 2 * X'Old;
end Arith;
```

```
package body Arith is
   procedure Double (X : in out Integer) is
   begin
      X := 2 * X;
   end Double;
end Arith;
```

Code (II)

Robustness as part of the requirements

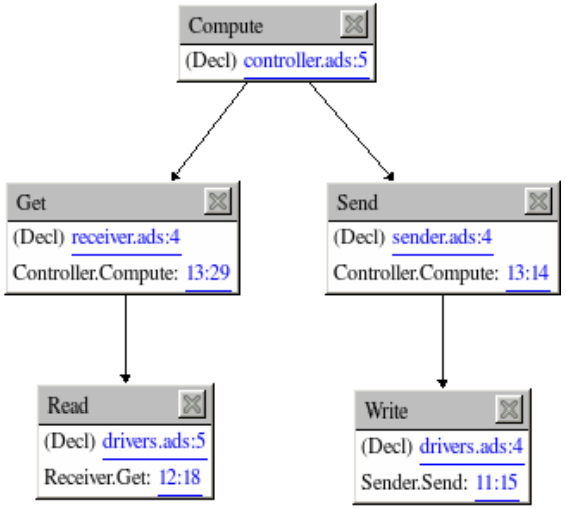
```
package Arith is
   procedure Double (X : in out Integer) with
      Post => (if X < Integer'First / 2 then
                  --  Underflow
                  X = Integer'First
               elsif X > Integer'Last / 2 then
                  --  Overflow
                  X = Integer'Last
               else
                  --  Nominal
                  X = 2 * X'Old)
end Arith;
```

```
package body Arith is
   procedure Double (X : in out Integer) is
   begin
      if X < Integer'First / 2 then
         X := Integer'First;
      elsif X > Integer'Last / 2 then
         X := Integer'Last;
      else
         X := 2 * X;
      end if;
   end Double;
end Arith;
```

DO-178 objectives for the code (I)

- **Compliance with LLRs**
  1. Implement the required functionality …
     - Testing or contract proving
  2. … and only that
     - This is more difficult but:
       - You can do manual code review, or
       - You can rely on exhaustive coverage analysis, or
         Also symbolic execution
       - Use SPARK flow analysis
         Detection of ineffective statements

- **Compliance with software architecture**
  - Match desired data and control flow
  - Ada already helps
    - With visibility control and parameter modes
  - You can visualize control-flow with tools
    - Compiler, GPS, …
  - Tools can help data-flow analysis showing who uses the data
  - Or define data and information flow with SPARK

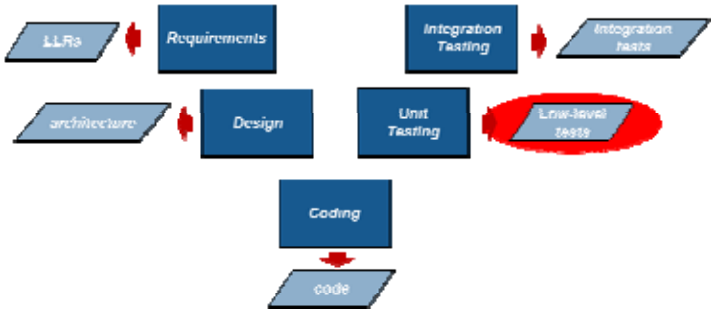


DO-178 objectives for the code (II)

- **Verifiability**
  - Avoid statements and structures that cannot be verified
  - Everything accessible from the spec is easy
  - Private parts with child units
  - Everything hidden in package bodies must be used through the spec
- **Conformance to coding standard**
  - Ada provides **pragma Restrictions** and **pragma Profile**
  - There are tools such as GNATcheck, AdaControl, … for extended and fine-grain checking
- **Traceability to LLRs**
  - Straightforward: implementation linked to the contracts
- **Accuracy and consistency**
  - It is about correctness and consistency of the code
  - Ada reliability underpinnings
  - You can go a step further with mathematical analysis
    - SPARK, CodePeer

Testing

- **The goal is to**
  - Demonstrate code satisfies the requirements
  - Potential sources of errors have been removed



- **Three kinds of tests**
  - Hardware/software integration
  - Software integration
  - Low-level testing

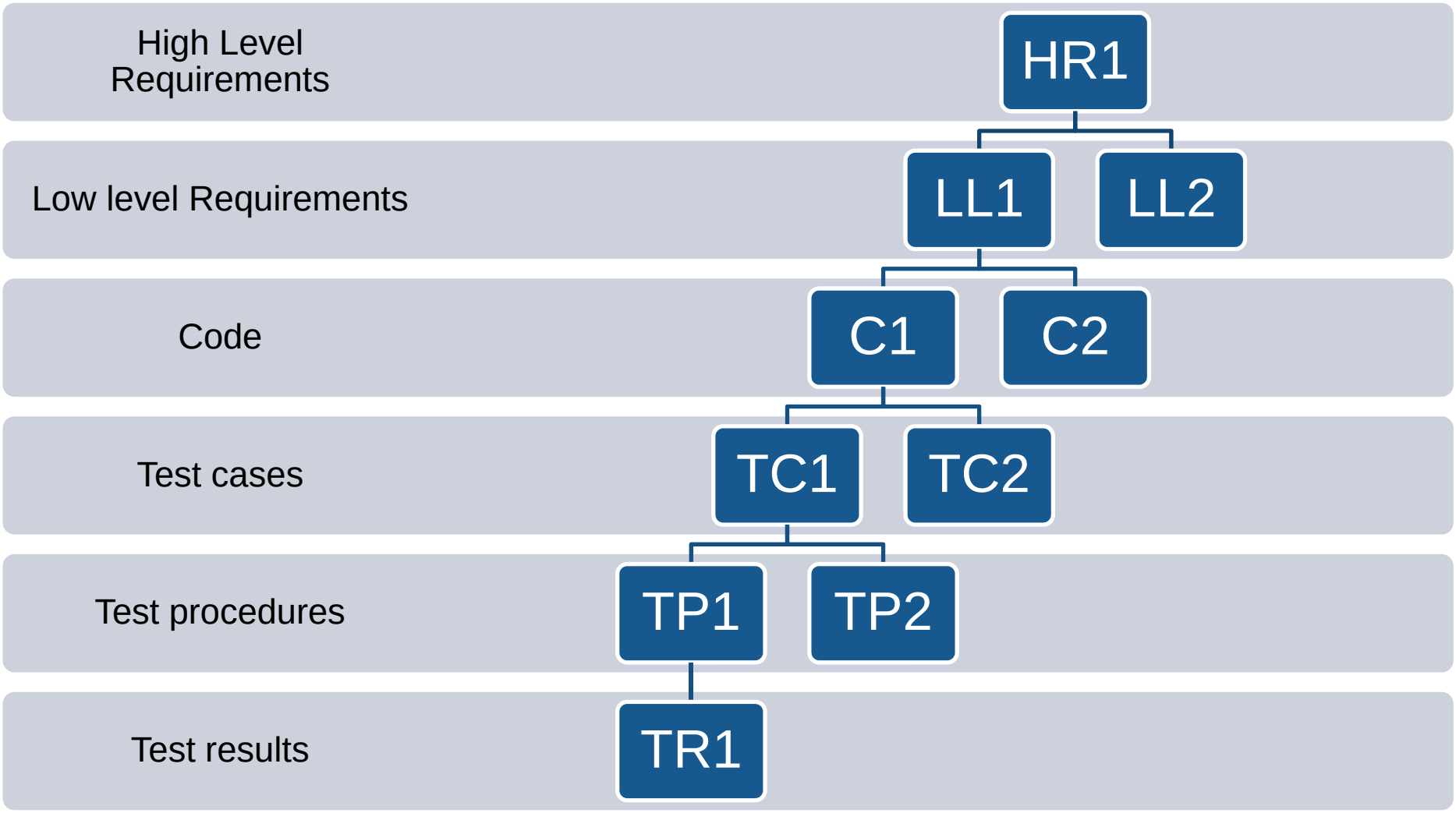
- **What we propose for low-level testing is**
  - Follow the DO-178C Formal Method Supplement, with mix of
    - Automated formal verification
    - Testing
      - Translate contract into run-time checks, and
      - Create a test aspect

Test aspect

```
package Arith is
   procedure Double (X : in out Integer) with
      Pre  => X >= Integer'First / 2 and then
              X <= Integer'Last / 2,
      Post => X  = 2 * X'Old,
      Test_Case => (Name     => "positive",
                    Mode     => Nominal,
                    Requires => X >= 0,
                    Ensures  => X >= 0),
      Test_Case => (Name     => "lower-bound",
                    Mode     => Nominal,
                    Requires => X = Integer'First / 2,
                    Ensures  => X = Integer'First),
      Test_Case => (Name     => "off-by-one-positive",
                    Mode     => Robustness,
                    Requires => X = Integer'Last / 2 + 1,
                    Ensures  => X = Integer'Last),
      Test_Case => (Name     => "off-by-one-negative",
                    Mode     => Robustness,
                    Requires => X = Integer'First / 2 - 1,
                    Ensures  => X = Integer'First);
end Arith;
```

Traceability

- **Every single artifact must be traceable**
  - Modifications applied to any artifact must be traceable too



Traceability – How to

Architecture

Requirements

```
package Arith is
   procedure Double (X : in out Integer) with
      Pre  => …,
      Post => …,
      Test_Case => …;
end Arith;
```

Test cases

Code

```
package body Arith is
   procedure Double (X : in out Integer) is
   begin
      X := 2 * X;
   end Double;
end Arith;
```

GNATtest

```
procedure Test_Double_XXX (T : in out Test) is
   procedure Double (X : in out Integer) renames
      Wrap_Test_Double_XXX;
begin
   …
   AUnit.Assertions.Assert (…);
end Test_Double_XXX;
```

Test procedures

AUnit

Test results

```
OK Operators.Test_Data.Tests : Test_Double_XXX
…

Total Tests Run:   N
Successful Tests:  N
Failed Assertions: 0
Unexpected Errors: 0
```

Conclusion

- **Ada 2012 very helpful in a DO-178 context**
  - Contracts for the requirements
  - Modularity, encapsulation, visibility control for the architecture
  - Aspect programming for testing
    - Automatic generation of test procedures and test results
- **Traceability links are there by construction**
  - Tools help automating generation of artifacts
- **Facilitates hybrid approach for verification**
  - Formal proofs plus testing
- **Reviews are more effective**
  - The context is clear
- **Maintainability and evolution easier**
  - More Agile