SAFRAN ENGINEERING SERVICES
C++ PROGRAMMING UNDER DO178 - 332

January 2017

C++ programming under DO178 / DO332
Course organization

Planning			Teatcher	Content	Room
12/12/2017	10h15	12h15	Manon	Presentation of DO178 / DO 332 Practical work on landing gear monitoring system	Caud07
12/12/2017	13h15	15h15	Manon	Practical work on landing gear monitoring system specification	D202
19/12/2017	13h15	15h15	Dupouy	Recall on C++ basics Zoom on DO 332 (review of main FAQ in DO332)	D202
19/12/2017	15h30	17h30	Dupouy	Practical work on landing gear monitoring system class modelling / C++ coding / testing	D202
10/01/2018	10h15	12h15	Dupouy	Practical work on landing gear monitoring system class modelling / C++ coding / testing	D203
10/01/2018	13h15	15h15	Dupouy	Practical work on landing gear monitoring system class modelling / C++ coding / testing	D203
17/01/2018	10h15	12h15	Dupouy	Practical work on landing gear monitoring system class modelling / C++ coding / testing	D205
17/01/2018	13h15	15h15	Dupouy	Practical work on landing gear monitoring system class modelling / C++ coding / testing	D205
22/01/2018	10h15	12h15	Dupouy	Practical work on landing gear monitoring system class modelling / C++ coding / testing	D205
22/01/2018	13h15	15h15	Dupouy	Exam	D205

Objectives

Main pedagogical objectives:

- To get acquainted with the basis of object coding for critical software
- To be able to realize simple coding using C++

Detailed pedagogical objectives:

- To get acquainted with critical software
- To get acquainted with DO178C norm and related refinements
- To get acquainted with DO178C process
- To get acquainted with DO332
- To get acquainted with the concept of Configuration management and Modification management
- To get acquainted with the use of manual coding versus automatic coding

DO178C : Synopsis of the course on DO178

- Difference between software and critical software
- Different DO norms for different purpose
- DO178C: Purpose
- DO178C: Historic and refinements
- DO178C: Different levels of criticism (DAL A to E)
- DO178C: Process and Life Cycle
- Manual coding versus Automatic coding
- Overview on modification management
- Overview on configuration management
- DO178C : Objectives
- DO178C : Zoom on DO332
- DO332: main principles
- DO332: FAQ

Difference between software and critical software

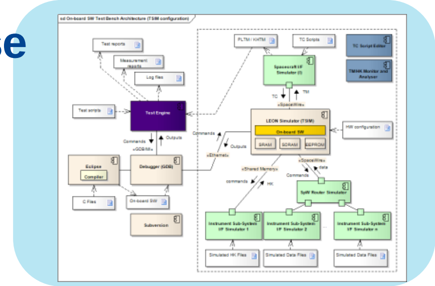
Definition:

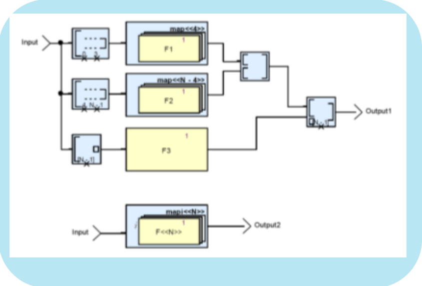
• Software: the programs used to direct the operation of a computer, as well as documentation giving instructions on how to use them

• Critical software:
→ No failures accepted, i.e. initial safety analysis is led to define the acceptable level of failures for the system (measured as probability $< 10^{x}$)
→ Definition of a process (full life cycle) to get no failures
→ Definition of a norm precognizing objectives and activities to be performed

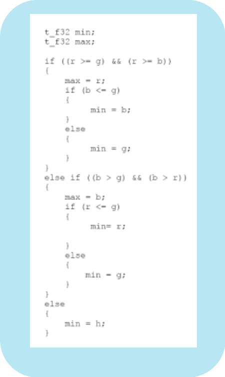
SAFRAN

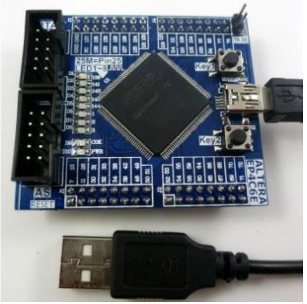
Different DO norms for different purpose



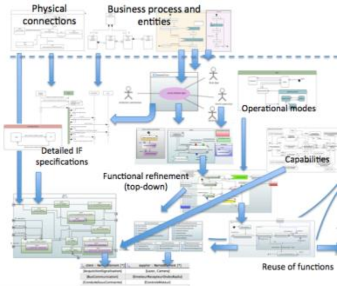


DO178 and all refinements (cf. next slide)
→ Software Architecture, Development





DO254 / DO160
→VLSI Design and Verification for FPGA, ASIC and SoC Components



ARP4754
→ System-level Design, Verification and Validation

DO178C : Purpose

DO-178C, Software Considerations in Airborne Systems and Equipment Certification provides guidance for developing airborne software

DO-178C is the primary document by which the certification authorities such as FAA or EASA approve all commercial software-based aerospace systems

DO-178C was published by RTCA (Radio Technical Commission for Aeronautics) Incorporated, in a joint effort with EUROCAE (The European Organisation for Civil Aviation Equipment)

DO-178C/ED-12C was completed in November 2011

DO178C : Historic and refinements

Evolution of DO-178 and Related Documents

Document	Year Published	Content
DO-178	1982	Provides very basic information for developing airborne software.
DO-178A	1985	Includes stronger software engineering principles than DO-178. Includes both verification and validation of requirements.
DO-178B	1992	Significantly longer than DO-178A. Provides guidance in form of *what* (objectives), rather than *how*. Provides visibility into life cycle processes and data. Does not include requirements validation.
DO-248B	2001	Includes errata for typographical errors in DO-178B. Also provides FAQs and DPs to clarify DO-178B. Was preceded by DO-248 in 1999 and DO-248A in 2000. Is not considered to be *guidance*—it is only clarification.
DO-278	2002	Applies DO-178B to CNS/ATM software. Adds some CNS/ATM-specific terminology and guidance.

Basis

Add V&V principles

Guidance instead of rules

Clarifications on DO178B

Dedicated to CNS/ATM Context (communication, navigation systems and air traffic management)

DO178C : Historic and refinements

DO-178C	2011	Content is very similar to DO-178B; however, it clarifies several areas, adds guidance for parameter data items, and references DO-330 for tool qualification.	Clarifications + dedicated DO330 for tool qulification
DO-278A	2011	Stands alone from DO-178C, unlike DO-278 which made direct references to DO-178B. Very similar to DO-178C with a few terminology changes and additional guidance needed for CNS/ATM software.	Evolutions
DO-248C	2011	Updates DO-248B to align FAQs and DPs with DO-178C updates. Also expanded to address DO-278A topics, to clarify additional topics that came about since DO-248B, and to add rationale for DO-178C objectives and supplements.	Evolutions
DO-330	2011	Provides guidance on tool qualification. It is a stand-alone document. DO-178C and DO-278A reference DO-330.	Dedicated to tool qulification
DO-331	2011	A technology supplement to DO-178C and DO-278A. Provides guidance on model-based development and verification.	Dedicated to model based development
DO-332	2011	A technology supplement to DO-178C and DO-278A. Provides guidance on OOT&RT.	Dedicated to OOT
DO-333	2011	A technology supplement to DO-178C and DO-278A. Provides guidance on formal methods.	Dedicated to formal methods

SAFRAN

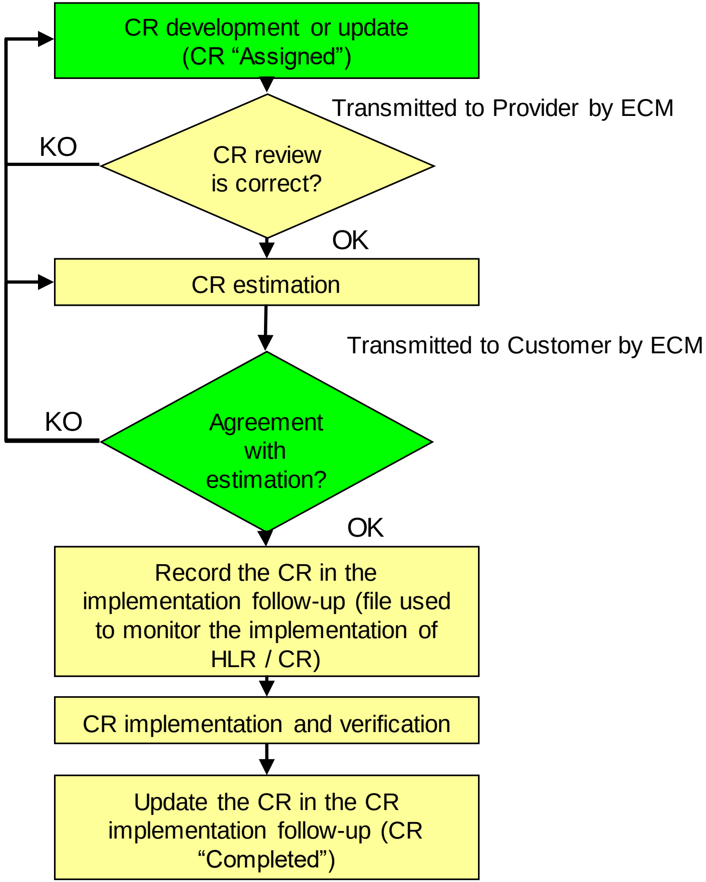
DO178C : Different levels of criticism (DAL A to E)

Design Assurance Level (DAL) is determined by examining the effects of a failure condition in the system.

Catastrophic - Failure may cause multiple fatalities, usually with loss of the airplane.
Hazardous - Failure has a large negative impact on safety or performance, or reduces the ability of the crew to operate the aircraft due to physical distress or a higher workload, or causes serious or fatal injuries among the passengers.
Major - Failure significantly reduces the safety margin or significantly increases crew workload. May result in passenger discomfort (or even minor injuries).
Minor - Failure slightly reduces the safety margin or slightly increases crew workload. Examples might include causing passenger inconvenience or a routine flight plan change.
No Effect - Failure has no impact on safety, aircraft operation, or crew workload.

Level	Failure condition	Objectives	With independence
A	Catastrophic	71	33
B	Hazardous	69	21
C	Major	62	8
D	Minor	26	5
E	No Safety Effect	0	0

Overview on modification management



Provider

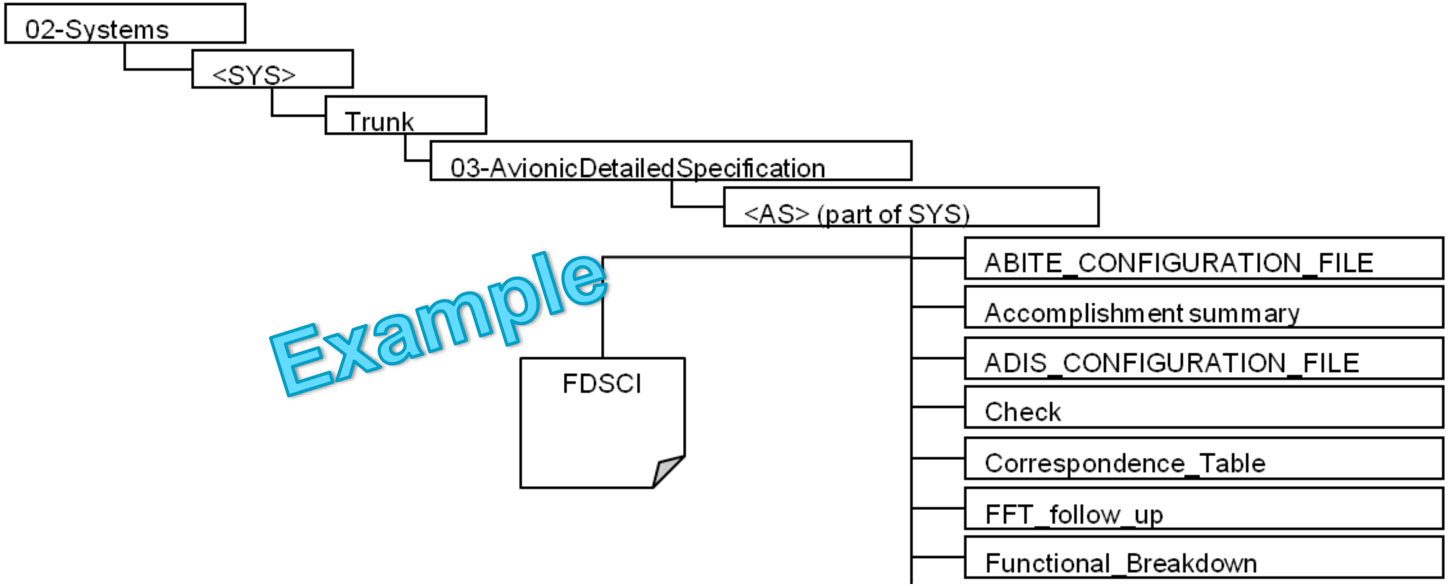
Customer

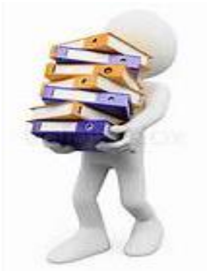
Overview on configuration management and modification management

Configuration management

The configuration management process aims to identify all the items of the project in order to control their lifecycle, the following of modifications and problem corrections.

Organization of project directory





Overview on configuration management and modification management

Identification rule of project documents and tag delivery

The formal tag name is defined as follows, according to DA4 at §*3.2.2.2*:
DELIVERY_<Partition name>_<M>_<DATE>_<Version>
Where:
Partition name is defined in §2.1.3.2
M is the maturity of the delivery, [STIx, SCADE100] where x is the sprint number
The date format is yyyymmdd
Version is the current developpement standard, [S0A, S0B, S1A, S1B, S2A, S2B, ...]

Example

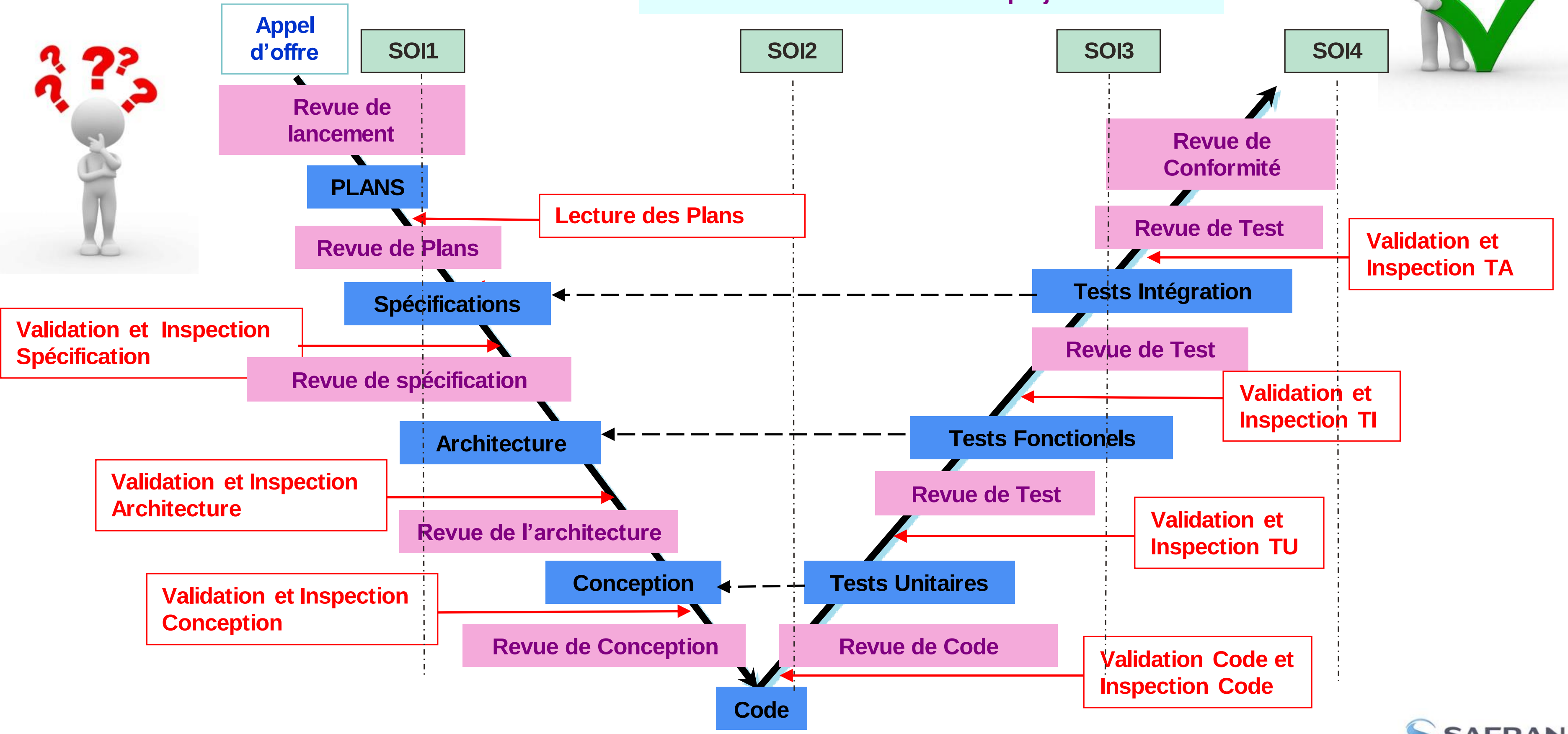
Rule of document versioning

For SENS plans tag, the rule consists to use identification (see 2.3.2) and to add version as follows:
•Vxy, x∈[A…Z] and y∈[1…N] : for intermediary versions
•x, x∈[A…Z] for validated versions

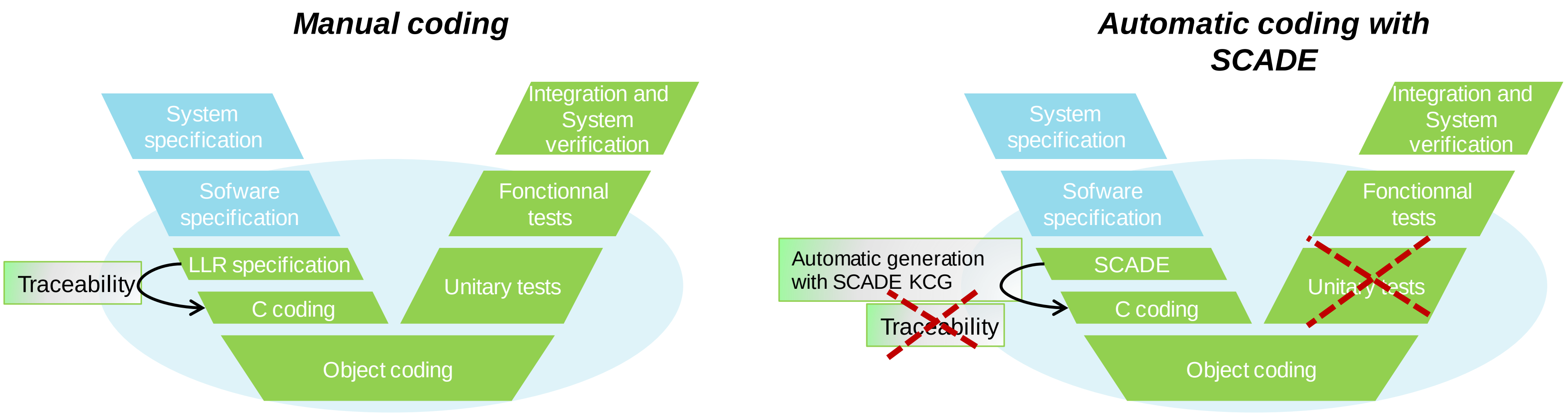
Example

DO178C : Process

Tout au long du projet : Audit processus
Suivi des Actions
Suivi des PR
Soutien DO178B au projet



Manual coding versus Automatic coding



Automatic coding : applicative software

DO178C : Coding Process description

Activity: To verify CSUs code for a QACG CSC	
Objectives: To verify the Source code	
Responsibility: SVVL is responsible for this activity	
Entry criteria: The inputs are released and managed under CM The CSC/CSUs Models are reviewed by technical and quality team (with no open blocking issue) The development tool Matlab/Simulink code generator is qualified	
	DAL
Inputs	**B**
- CSC Matlab/Simulink code generation procedure	M
- CSC Matlab/Simulink code generation report	M
Outputs	**B**
- TC on CSC Matlab/Simulink code generation procedure	I
- TC Matlab/Simulink code generation	M
Roles/Tasks	
Software verification team	**B**
- To perform the CSC Matlab/Simulink code generation procedure TC	I
- To update the Matlab/Simulink code generator tool errata analysis	M
- To analyze the CSC Matlab/Simulink code generation report	M
- To perform the TC Matlab/Simulink code generation	M
Closure criteria: TS-TRR accepted, CDR accepted	
Methods/Rules/Guidelines - Technical check - Tool errata analysis - Tool warning analysis	

SAFRAN

DO178C : Coding Process verification

1.To verify the code (MCP process)" activity

Prerequisites

The inputs are configuration managed.

SEnS starts the activity after SDDD maturity assessment performed with **§0**: To verify the LLR.

Planning baseline is approved.

Activity

To perform Code technical check:

Different points to check are:

- Traceability with upstream document(s)
- Conformity with standard
- Consistency
- Accuracy
- Compliance with upstream document(s)
- Testability, Verifiability

These checks are done using the checklist related to this activity and automatic check (results of automatic check are configuration managed).

Input data	Output data
PMP SDDD Source code files (.c, .h) Building procedures Products files (.o, .a) Traceability LLR <-> code document Standard: C language programming rules **[DA09]**	code CL

DO178C : Objectives

Objectif			Applicabilité par niveau logiciel				Produit		Catégorie contrôle par niveau logiciel			
	Description	Réf.	A	B	C	D	Description	Réf.	A	B	C	D
1	Les exigences de haut niveau sont développées.	5.1.1a	❍	❍	❍	❍	Description de Spécification	11.9	①	①	①	①
2	Les exigences de haut niveau dérivées sont définies.	5.1.1b	❍	❍	❍	❍	Description de Spécification	11.9	①	①	①	①
3	L'architecture du logiciel est développée.	5.2.1a	❍	❍	❍	❍	Description de Conc					
4	Les exigences de bas niveau sont développées.	5.2.1a	❍	❍	❍	❍	Description de Conc					
5	Les exigences de bas niveau dérivées sont définies.	5.2.1b	❍	❍	❍	❍	Description de Conc					
6	Le Code Source est développé.	5.3.1a	❍	❍	❍	❍	Code Source	11.11	①	①	①	①
7	Le Code Objet Exécutable est produit et intégré dans la machine cible.	5.4.1a	❍	❍	❍	❍	Code Objet Exécutable	11.12	①	①	①	①

Process design

Objectif			Applicabilité par niveau logiciel				Produit		Catégorie de contrôle par niveau logiciel			
	Description	Réf.	A	B	C	D	Description	Réf.	A	B	C	D
1	Les exigences de haut niveau sont en conformité avec les spécifications du système.	6.3.1a	●	●	❍	❍	Résultats de Vérification du Logiciel	11.14	②	②	②	②
2	Les exigences de haut	6.3.1b	●	●	❍	❍	Résultats de Vérification du	11.14	②	②	②	②
									②	②		
									②	②	②	
	règles.								②	②	②	
6	Les exigences de haut niveau sont traçables vers les spécifications au système.	6.3.1f	❍	❍	❍	❍	Résultats de Vérification du Logiciel	11.14	②	②	②	②
7	Les algorithmes sont précis.	6.3.1g	●	●	❍		Résultats de Vérification du Logiciel	11.14	②	②	②	

Process verification

What does it mean → 2 soft engineer will write the same code?
How can it be measured? What are the criterion?
→Need more explanations concerning Classes and Objects, encapsulation, polymorphism …

Some of the objectives are too general to be in line with the problematic encountered when coding with OOC → refinement realized in DO332

DO178C : Zoom on DO332

Modified objectives include:

• Software architecture is compatible with high-level requirements (section OO.6.3.3.a).
• Software architecture is consistent (section OO.6.3.3.b).
• Source Code complies with software architecture (section OO.6.3.4.b).
• Source Code is accurate and consistent (section OO.6.3.4.f).

Additional objectives include:

• Verify local type consistency (section OO.6.7.1).
• Verify the use of dynamic memory management (section OO.6.8.1)
➔ No instantiation of objects allowed when running the program (C++, Java, ADA)
➔Creation of segregation using the concept of hypervisors and several virtual machines, in case of crash, only one virtual machine is impacted

Additional activities include:

• Executable Object Code complies with high-level requirements (section OO.6.4.2.1.e).
• Executable Object Code complies with low-level requirements (section OO.6.4.2.1.e).

DO178C : Example of modified objectives in DO332

1.Memory allocation

OO languages brings dynamic memory allocation.
But on critical system, the memory allocation has to be controlled to prevent memory leaks.
In DO332, dynamic memory is not forbidden, but the choice depends on the effort on resources analysis (memory analysis).

- if the runtime brings a garbage collector, so the effort is on the runtime certification
- If not (this is the case of C++), you have to make memory analysis.

→ to reduce this, we often use static allocation.
→ Another way, is to make object creation at the start of the software

2.Inheritance issue

In certification, dead code is not allowed. All the functions have to be used,tested and documented.
In OO languages, the inheritances principles allows to have subclasses which get all the methods and properties of the superclass, even if the subclass don't use it. Moreover, it can become less readable.
One of the good practice in DO332 is the « Liskov principle » :

- The subclass can be substitute by it superclass and so on.
- It means that the whole properties and methods have the same behavior.

Example FAQ#21 and 22 in DO332

DO178C : Example of modified objectives in DO332

OO.B.1 General Questions B-1

FAQ #1: What is covered by this supplement? B-1

FAQ #2: What is the relationship between this supplement and the OOTiA Handbook? B-2

FAQ #3: What additional information is required for the software planning process in an OO related system? B-2

FAQ #4: What additional measures may be taken when reusing OO components? B-2

FAQ #5: What additional objectives and activities have been added to support OOT&RT in the software development processes? B-3

FAQ #6: What additional objectives may be met and activities may be performed when reviewing and analyzing software using OO components? B-3

FAQ #7: What are the vulnerabilities required to be addressed in section OO.11.1? B-4

FAQ #8: Are there any special concerns for using closures? B-4

DO178C : Example of modified objectives in DO332

OO.B.2 Requirements Considerations B-5
FAQ #9: What is the relationship between low-level requirements and classes? B-5
FAQ #10: What role do classes play in the design process? B-5
FAQ #11: When should a requirement be expressed as invariants as opposed to preconditions and postconditions? B-6
FAQ #12: Do all methods, including constructor, accessor, mutator, and destructor, need requirements? B-6

DO178C : Example of modified objectives in DO332

OO.B.3 Design Considerations B-6

FAQ #13: What is typing? B-6

FAQ #14: Why is the Liskov Substitution Principle (LSP) important? B-7

FAQ #15: Must all classes in an application be type consistent? B-8

FAQ #16: What impact does the Liskov Substitution Principle (LSP) have on architecture and low-level requirements? B-8

FAQ #17: Are there garbage collection techniques that should be avoided? ... B-9

FAQ #18: Should inheritance be used for code sharing? B-9

FAQ #19: What techniques can be used to minimize potential problems with the use of multiple inheritance? B-9

FAQ #20: Is static dispatch safer than dynamic dispatch? B-10

FAQ #21: How can Liskov Substitution Principle (LSP) be violated in a strongly-typed language, such as Ada or Java? B-11

FAQ #22: What would need to be done to make the examples in FAQ #21 conform to the Liskov Substitution Principle (LSP)? B-15

FAQ #23: Does the discussion on preconditions, postconditions, and invariants imply the use of a particular design methodology? B-15

SAFRAN

DO178C : Example of modified objectives in DO332

OO.B.4 Programming Language Considerations B-15

FAQ #24: What is an example of static dispatch with method overriding? ... B-15

FAQ #25: Is it acceptable to use a COTS library, such as the C++ Standard Template Library, without explicitly specifying high-level and low-level software requirements, tests, and documents for this library?B-17

FAQ #26: What considerations are important when using garbage collection?B-17

FAQ #27: What considerations are important when using a virtual machine?B-18

DO178C : Example of modified objectives in DO332

OO.B.5 Verification Considerations B-19

FAQ #28: What additional objectives and activities have been added to support OOT&RT in the verification process? B-19

FAQ #29: In section OO.5.5.d, what is meant by "a requirement which traces to a method"? B-19

FAQ #30: How does the formal verification activity for ensuring local type consistency differ from a low-level requirements review? B-20

FAQ #31: Does the guidance related to dynamic dispatch also apply to the use of function pointers (for example, function pointer tables and callback functions)? B-20

FAQ #32: What is the difference between fulfilling type consistency locally and globally? B-21

FAQ # 33: How can local type consistency be verified? B-21

FAQ # 34: How can local type consistency with an abstract superclass be verified by testing without using pessimistic testing? B-21

DO178C : Example of modified objectives in DO332

FAQ #35: How can classes that include unencapsulated data be verified for local type consistency? B-22

FAQ #36: Does this supplement require tracing of low-level requirements to preconditions, postconditions, and invariants? B-23

FAQ #37: Is "flattened class testing" a way of verifying local type consistency? B-23

FAQ #38: When preconditions, post conditions, and invariants are checked at run time, what verification must be performed on the additional Source Code? B-24

FAQ #39: Is there a problem with covariant collections? B-24

DO178C : To go further

DO332 is available on the intranet of ENAC

To go forward, look at the FAQ and ask explanation if any need

ect-Oriented Technology and Related Techniques.pdf

To learn more about DO178: http://www.do178site.com/do178b_questions.php

Course organization

Fundamentals of C++

C++ Basics : What is C++ programming

- C++ can be viewed as a procedural language with some additional constructs, some of which are added for object oriented programming and some for improved procedural syntax.

- A well written C++ program will reflect elements of both object oriented programming style and classic procedural programming.

- C++ is actually an extensible language. We can define new types in such a way that they act just like the predefined types which are part of the standard language.

- C++ is designed for large scale software development.

→ The goal of C++ programming language was to add features that support data abstraction and object oriented concepts to C language.

C++ Basics : Object Oriented Approach

Object oriented programming (OOP)

Model real-world objects with software counterparts

Attributes (state) - properties of objects

- Size, shape, color, weight, etc.

Behaviors (operations) - actions

- A ball rolls, bounces, inflates and deflates
- Objects can perform actions as well

Inheritance

- New classes of objects absorb characteristics from existing classes

Objects

- Encapsulate data and functions
- Information hiding
  - Communicate across well-defined interfaces

C++ Basics : Object Oriented Approach

User-defined types (classes, components)

<u>Data members</u>

- Data components of class

<u>Member functions</u>

- Function components of class

<u>Association</u>

<u>Reuse classes</u>

C++ Basics : Object Oriented Approach

Unified Modeling Language (UML)

2001: Object Management Group (OMG)

Model object-oriented systems

Complex, feature-rich graphical language

An example

```
#include<iostream.h>
int main()
{
        int i=0;
        double x=2.3;
        char s[ ] ="Hello";
        cout<<i<<endl;
        cout<<x<<endl;
        cout<<s<<endl;
        return 0;
}
```

1.cpp

Lexical elements

Identifiers: case sensitive

nCount, strName, Strname

Reservered words

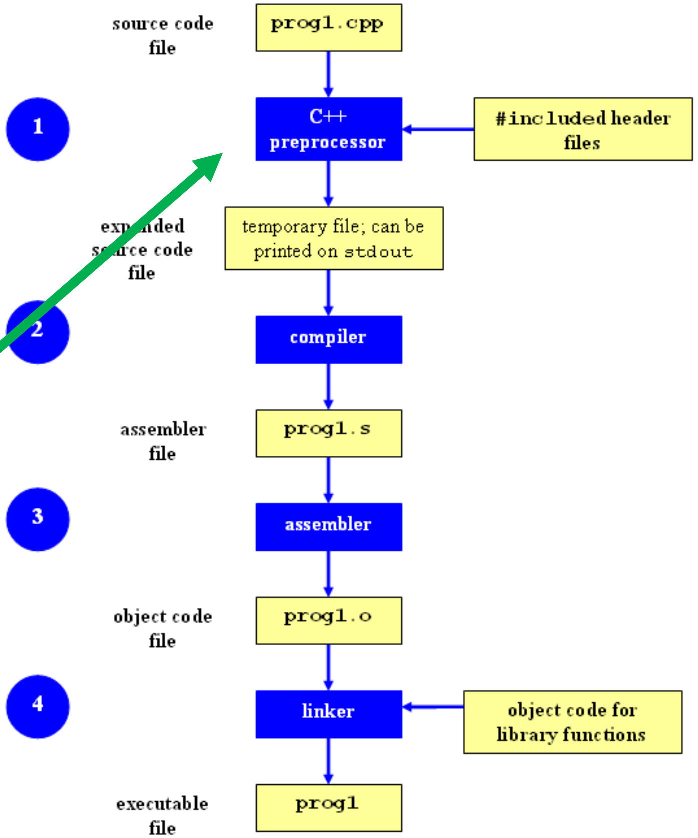
if, else, while

Operators

+, ==, &, &&, ‘? :’

Preprocessor Directives

#include, #if,



Primitive Data Types

Name	Size (bytes)	Description	Range
char	1	character or eight bit integer	signed: -128..127 unsigned: 0..255
short	2	sixteen bit integer	signed: -32768..32767 unsigned: 0..65535
long	4	thirty-two bit integer	signed: -2^{31} .. $2^{31}-1$ unsigned: 0 .. 2^{32}
int	* (4)	system dependent, likely four bytes or thirty-two bits	signed: -32768..32767 unsigned: 0..65535
float	4	floating point number	3.4e +/- 38 (7 digits)
double	8	double precision floating point	1.7e +/- 308 (15 digits)
long double	10	long double precision floating point	1.2e +/- 4932 (19 digits)
bool	1	boolean value false → 0, true → 1	{0,1}

Variables declaration & assignments

```
#include<iostream>
using namespace std;
int main()
{
        int i,j,k;
        int l;
        i=10;
        j=k=l=20;  //j=(k=(i=20))
        cout<<"i="<<i<<endl;
        cout<<"k="<<k<<endl;
        cout<<"l="<<l<<endl;
        i+=10;     //i = i + 10;
        i++;       //i = i + 1;
        cout << "i="<<i<<endl;
}
```

2.cpp

Expressions

Boolean expressions

== , != , >, >=, < , <=, …

&& , || …

Arithmetic expression

+ , - , *, / ,% …

&, | …

Assignment

=

? :

Expressions have values

Example of expressions

```
7 && 8
7 & 8
7 / 8
7 % 8
7 >> 1

(i > 127 ? true : false)
(i > 127 ? i-127 : i)
```

3.cpp 4.cpp

Operator	Priority	Description	Order
()	1	Function call operator	from left
[]	1	Subscript operator	from left
− >	1	Element selector	from left
!	2	Boolean NOT	from right
~	2	Binary NOT	from right
++	2	Post-/Preincrement	from right
−−	2	Post-/Predecrement	from right
−	2	Unary minus	from right
(*type*)	2	Type cast	from right
*	2	Derefence operator	from right
&	2	Address operator	from right
sizeof	2	Size-of operator	from right
*	3	Multiplication operator	from left
/	3	Division operator	from left
%	3	Modulo operator	from left
+	4	Addition operator	from left
−	4	Subtraction operator	from left
<<	5	Left shift operator	from left
>>	5	Right shift operator	from left
<	6	Lower-than operator	from left
<=	6	Lower-or-equal operator	from left
>	6	Greater-than operator	from left
>=	6	Greater-or-equal operator	from left
==	7	Equal operator	from left
!=	7	Not-equal operator	from left
&	8	Binary AND	from left
^	9	Binary XOR	from left
|	10	Binary OR	from left
&&	11	Boolean AND	from left
||	12	Boolean OR	from left
?:	13	Conditional operator	from right
=	14	Assignment operator	from right
op=	14	Operator assignment operator	from right
,	15	Comma operator	from left

Statements

Assignments

Conditional

Loop

Goto,break,continue

Compound statement

Conditional

if A B ;

if A B else C

If (I > 10) {cout<<“ > 10”;} else {cout<<“ < 10”;}

5.cpp

Loop, for

for (A;B;C) D

1 execute A

2 execute B

3 if the value of B is false(==0), exit to D

4 execute C, goto 2

for(i=0; i<n; i++){cout << A[i]<<endl;}

for(;;) {…}

Loop, while & do while

while A B

While (i>10) { x-=4;i--;}

do A while B

do {x -=4;i--} while (i>10);

Goto,break,continue

```
For (; ;){
…
If (a==b) break;
…
}
C
---------------------------------------
---------------------------------------

For (;;){
{B}
If (a==b) continue;
{A}
}
```

switch

```
switch (grade){
case 'A':++nACount;break;
case 'B':++nBCount;break;
case 'C':++nCCount;break;
case 'D':++nDCount;break;
default: cout<<"Something wrong\n";break;
}
```

Try: write a program using the code segment. Then remove several of the 'break's and see the difference

functions

Can not define a function within another function*

Parameters passed by value or reference

Example

```
#include<iostream>
using namespace std;

int square (int);

int main ()
 {
int z = 4;
cout << square(z);
}

int square (int x)
{               x = (x*x); return x; }
```

6.cpp

Pass by value

```
void swap1(int x,int y)
{
        int temp=x;
        x = y;
        y=temp;
}
```

Pass by reference

```
void swap2(int& x,int& y)
{
        int temp=x;
        x = y;
        y=temp;
}
```

7.cpp

Array in C/C++

Definition

Int a[10];   //int[10] a;

Char b[12];

No bounds checking

The cause of many problems in C/C++

Array

Int x[7];

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]

Int score[3][3]={{1,2,3},{2,3,4}{3,5,6}};

Array: confusing

What is the result of the program.

So, array is passed by reference?

8.cpp

Pointer

Example

int *p, char * s;

The value of a pointer is just an address.

Why pointers?

Dereferencing (*)

Get the content

Referencing (&)

Get the address of

Examples of pointer

```
int *p;
Int a;
a=10;
p=&a;
*p=7;
Int b=*p;
```

You must initialize a pointer before you use it

81.cpp 82.cpp

Array and pointer

9.cpp

arithmetic of pointer

Suppose n is an integer and p1 and p2 are pointers

p1+n

p1-n

p1-p2

- 91.cpp

Strings

C

A string is an array of chars end with ‘\0’

char name[]=“ABC”;

char school_name[]={‘N’,’Y’,’U};

C++ library: string class

- 10.cpp 101.cpp

Dynamic allocating memory

new , delete
int *p=new int;
int *p=new int [12];
delete p;
delete []p;
malloc,…

- 11.cpp (difference between different implementations)

Course organization

C++ programming basics

4 h	Course on C++	
2h		Fundamentals of C++
1h10min		Class & inheritance
40min		Overloading & overriding
10min		Error handling,…

Struct:

```
struct person
{   long nId;
    char strName[30];
    int nAge;
    float fSalary;
    char strAddress[100];
    char strPhone[20];  };

struct person a ,  b, c;
struct person *p;
```

12.cpp

union

```
union num
{
   int x;
   float y;
}
```

13.cpp

More in a stucture: operations

```
struct box
{

    double dLength,dWidth,dHeight;
    double dVolume;

    double get_vol()
      {
        return dLength * dWidth * dHeight;
      }
}
```

14.cpp   141.cpp

Class

```
class box
{
   double dLength,dWidth,dHeight;
   double dVolume;
public:
   double vol(){return dLength * dWidth * dHeight;}

}
```

15.cpp

Public vs. private

Public functions and variables are accessible from anywhere the object is visible

Private functions and variable are only accessible from the members of the same class and “friend”

Protected

class

```
class box
{
   double dLength,dWidth,dHeight;
   double dVolume;
public:
   double vol() ;
}

double box::vol()
{
return dLength * dWidth * dHeight;}
}
```

16.cpp

Constructors

A special member function with the same name of the class

No return type (not void)

Executed when an instance of the class is the created

17.cpp

Deconstructors

A special member function with no parameters

Executed when the class is destroyed

- 18.cpp

Tricky

What is the result of the program

19.cpp

How many times the constructor executed?

How many times the deconstructor executed

Examples 20.cpp 21.cpp

Empty constructor & Copy constructor

Empty constructor

The default constuctor with no parameters when an object is created

Do nothing: e.g. Examp::Examp(){}

Copy constructor

Copy an object (shallow copy)

The default constructor when an object is copied (call by value, return an object, initialized to be the copy of another object)

- 22.cpp {try not to pass an object by value)

Inheritance

Base class

Derived class

- 23.cpp

Protected members

- 24.cpp

Constructors

28.cpp

Inheritance

	base	derived
Public inheritance	public	public
	protected	protected
	private	N/A
Private inheritance	public	private
	protected	private
	private	N/A
Protected inheritance	public	protected
	protected	protected
	private	N/A

Static members in class

Static variables

Shared by all objects

Static functions

Have access to static members only

Static members can be accessed by the class name

29.cpp

Friend functions

Have access to the private members of a class.

Must be declared as friend in that class.

Why friend functions?

efficiency

- 30.cpp 31.cpp 32.cpp

Friend class

A class can be declared as the friend of another class.

Course organization

C++ programming basics

4 h	Course on C++	
2h		Fundamentals of C++
1h10min		Class & inheritance
40min		Overloading & overriding
10min		Error handling,…

Function overloading

Define several functions of the same name, differ by parameters.

```
void Show()

void Show(char *str)

Void show(int x)
```

33.cpp

Function overloading

Must have different parameters

```
int func1(int a, int b);

double func1(int a, int b);

void func(int value);

void func(int &value);
```

Static binding

The compilers determine which function is called.

(Often used for the multiple constructors)

Operator overloading

Define new operations for operators (enable them to work with class objects).

+ - * / = < > += -= *= /= << >> <<= >>= == != <= >= ++ -- % & ^ ! | ~ &= ^= |= && || %= [] () new delete

Class date x ,y

x+y x-y x>y, x&y

Special member functions

Ret_type class_name::operator<>(arg_list)

34.cpp

Overloading summary

Same name

Different parameters

Static binding (compile time)

Anywhere

Virtual function & overriding

Define a member function to be virtual

Use pointer/reference/member functions to call virtual functions

Dynamic binding

Time consuming

The constructor cannot be virtual

Must be a member function

Virtual functions examples

By pointers 42.cpp

By reference

By member function of the base class

Overloading & overriding

Polymorphism

Static and dynamic

Compile time and running time

Parameters

Anywhere / between the base and derived class

Pure virtual functions & abstract class

Pure virtual functions

A function declared without definition

virtual ret_type func_name(arg_list)= 0;

Abstract class

A class contains one or more pure functions

Can not be instantiated

Can be used to define pointers

Course organization

C++ programming basics

4 h	Course on C++	
2h		Fundamentals of C++
1h10min		Class & inheritance
40min		Overloading & overriding
10min		Error handling,…

Exceptions

```
try
{ // code to be tried
  throw exception; }
 catch (type  exception)
 { // code to be executed in case of exception }
```

Examples of exception

54.cpp

throw out an object.

55.cpp

Uncaught exception

Uncaught exceptions will be thrown into outer scope.

If no catch, usually program will terminate.

Void terminate();

POWERED
BY TRUST

Powered by trust : la confiance est notre moteur