# SOFTWARE DEVELOPMENT

Franco Gasperoni

gasperon@act-europe.fr

http://libre.act-europe.fr

# Copyright Notice

# Object Oriented Programming

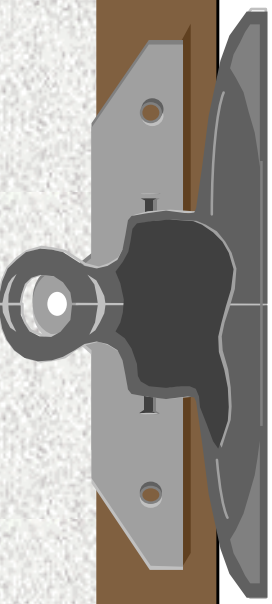
# When creating a new system you must identify its ...

- **Data types**
  (what kind of data will be manipulated)
- **Functionalities**
  (what kind of manipulations are allowed)

# Software System Organization

- Around its **functionalities**
  (functionality-oriented / structured programming)

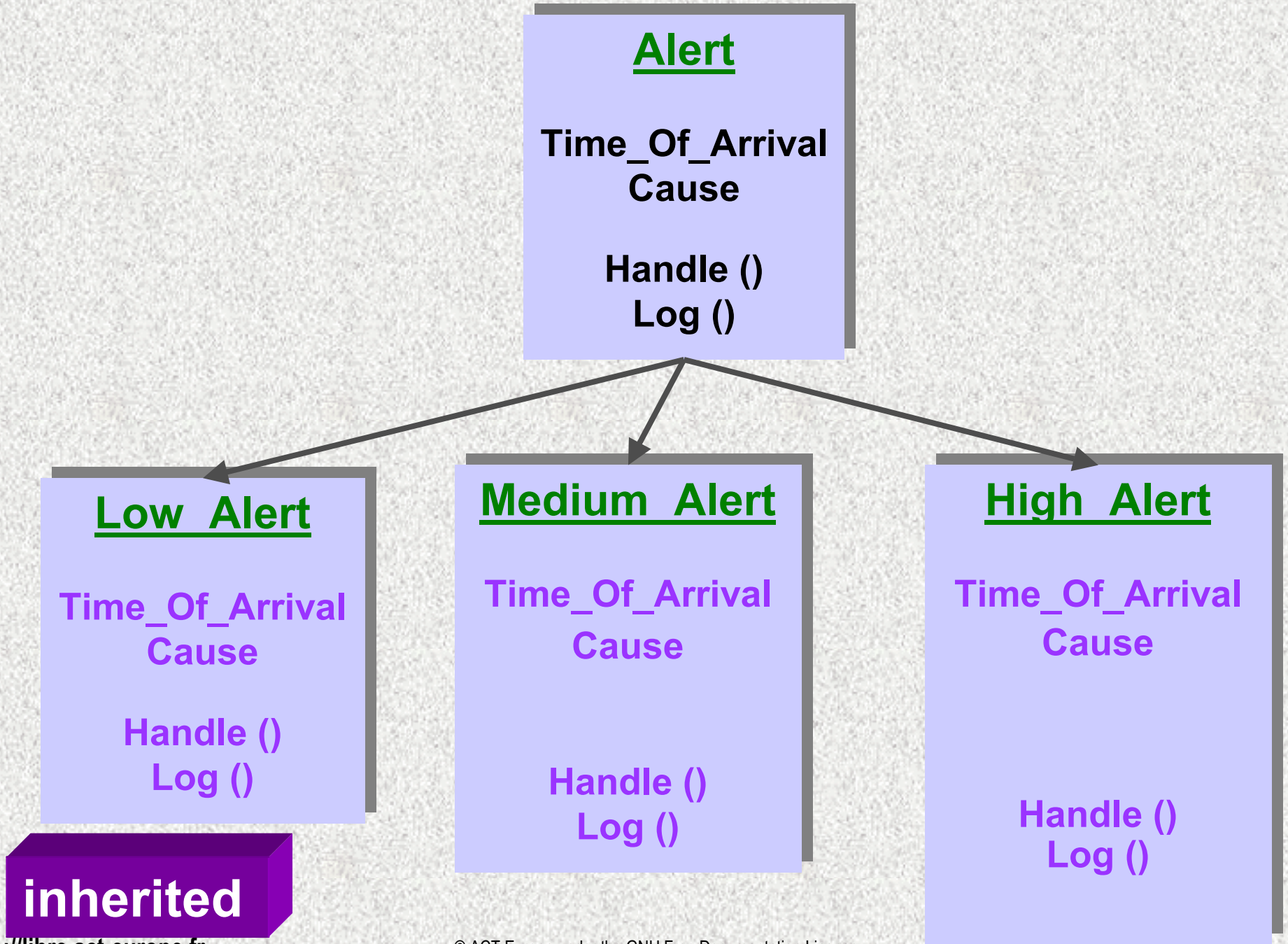
- around its **data types**
  (object-oriented programming)

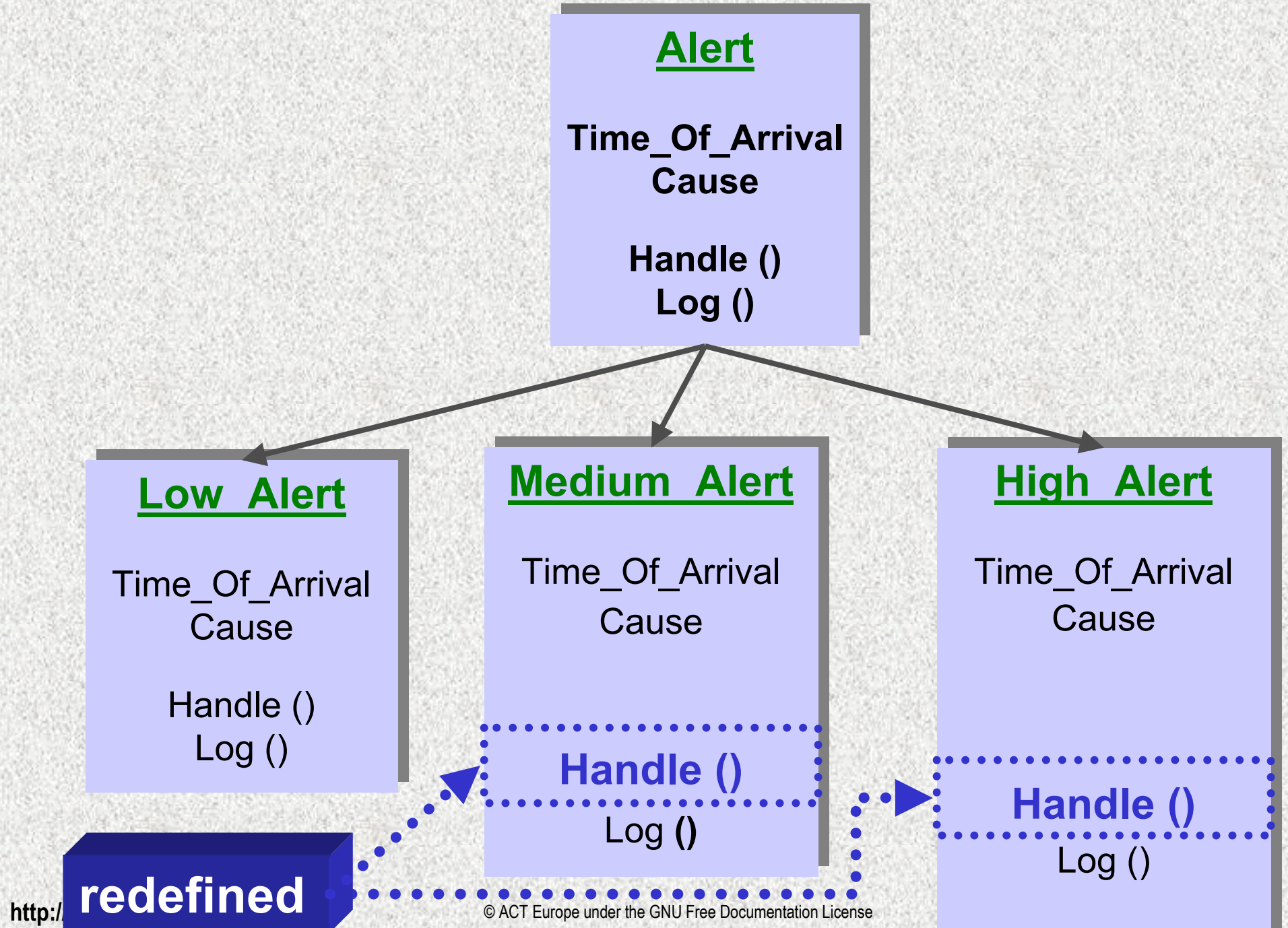


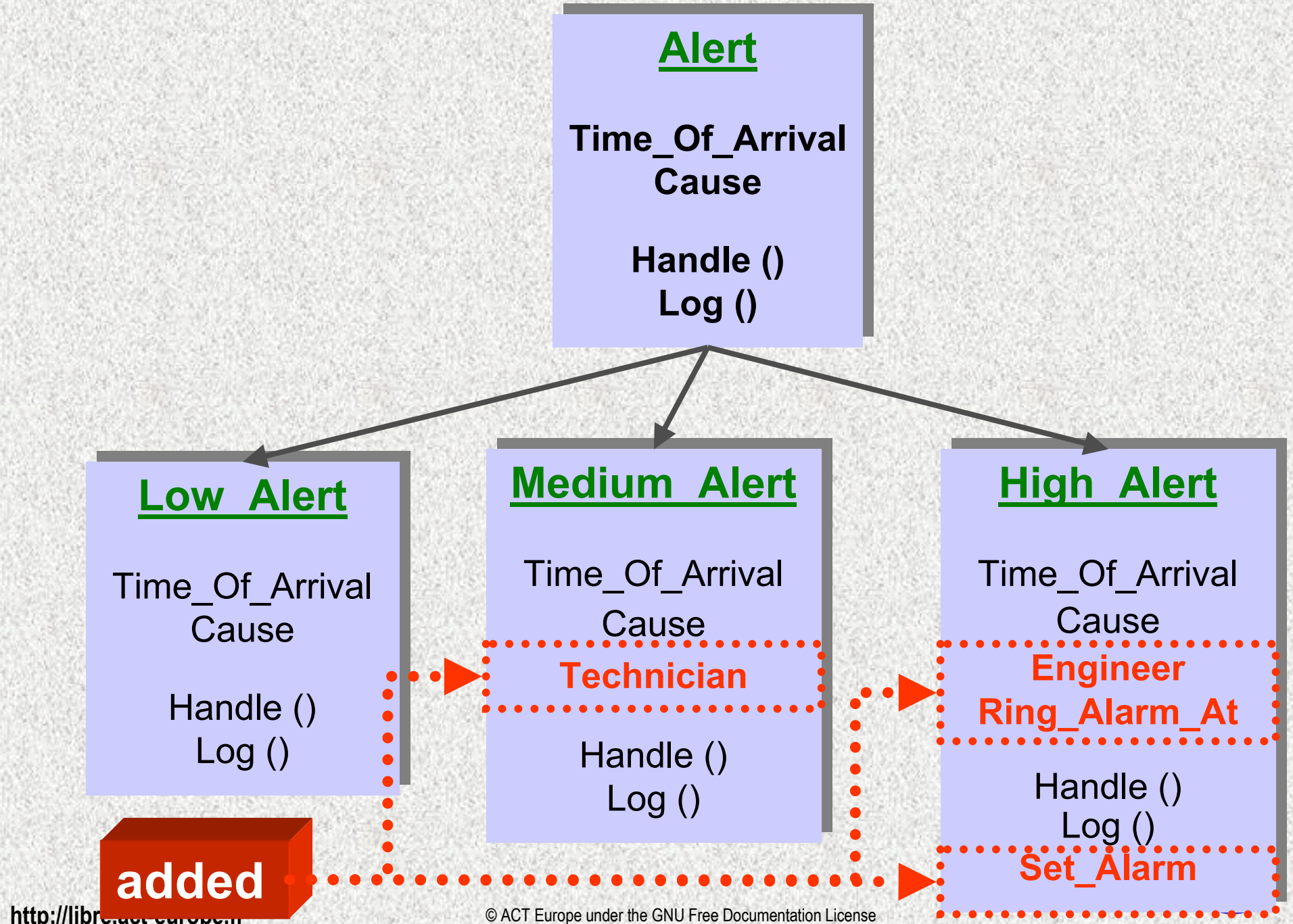
- **Object-Oriented Organization**
  - *inheritance (simple)*
  - polymorphism
  - abstract types & subprograms
  - modifying an OO system
  - when to use OO organization

# Often types have some but not all properties in common...

- Create completely different types
- Use variant programming to factor commonalties
- **Use inheritance**







- Alert
- Low_Alert
- Medium_Alert
- High_Alert

# Are 4 *Different* Types

```
with …;

package Alerts is

   type Alert is tagged record
      Time_Of_Arrival : Calendar.Time;
      Cause           : String (1 .. 200);
   end record;

   procedure Handle (A : in out Alert);
   procedure Log    (A : Alert);

   ...
end Alerts;
```

Alert is a tagged type

Primitive operations (methods)

```
package Alerts is

   type Alert is tagged record
      Time_Of_Arrival : Calendar.Time;
      Cause           : String (1 .. 200);
   end record;

   procedure Handle (A : in out Alert);
   procedure Log    (A : Alert);

   type Low_Alert is new Alert with null record;

   ...
end Alerts;
```

inherited

Low_Alert is a tagged type derived from Alert

Derived type inherits everything by default

```
package Alerts is

   type Alert is tagged record
      Time_Of_Arrival : Calendar.Time;
      Cause           : String (1 .. 200);
   end record;

   procedure Handle (A : in out Alert);
   procedure Log    (A : Alert);

   type Medium_Alert is new Alert with record
      Technician : Person;
   end record;

   procedure Handle (A : in out Medium_Alert);

   ...
end Alerts;
```

inherited

added

redefined

```
package Alerts is

   type Alert is tagged record
      Time_Of_Arrival : Calendar.Time;
      Cause           : String (1 .. 200);
   end record;

   procedure Handle (A : in out Alert);
   procedure Log    (A : Alert);

   type High_Alert is new Alert with record
      Engineer : Person;
      Ring_Alarm_At : Calendar.Time;
   end record;
   procedure Set_Alarm (A : in out Alert; Wait : Duration);
   procedure Handle (A : in out High_Alert);
end Alerts;
```



# Attributes (record fields)

- Are **always inherited**
- Can never be redefined or deleted
- You can **add** new attributes

# Inherited Attributes

```
with Alerts; use Alerts;
procedure Client is
    A   : Alert;
    A_L : Low_Alert;
    A_M : Medium_Alert;
    A_H : High_Alert;

begin
    A   . Time_Of_Arrival := ...;
    A_L . Time_Of_Arrival := ...;
    A_M . Time_Of_Arrival := ...;
    A_H . Time_Of_Arrival := ...;
end Client;
```

# Added Attributes

```
with Alerts; use Alerts;
procedure Client is
    A   : Alert;
    A_L : Low_Alert;
    A_M : Medium_Alert;
    A_H : High_Alert;

begin
    A   . Engineer := …;
    A_L . Engineer := …;
    A_M . Engineer := …;

    A_H . Engineer := …;
end Client;
```

# Operations (methods)

- **Inherited** operation has exactly the same code as the original
- **Redefined** (or overridden) operations have new code (can never delete an operation)
- **Added** operations are new operations

# Inherited & Redefined Operations

```
with Alerts; use Alerts;
procedure Client is
    A   : Alert;
    A_L : Low_Alert;
    A_M : Medium_Alert;
    A_H : High_Alert;

begin
    Handle (A);
    Handle (A_L);
    Handle (A_M);
    Handle (A_H);
end Client;
```

```
type Alert is tagged record ... end record;
procedure Handle (A : in out Alert);

type Low_Alert is tagged record ... end record;
-- procedure Handle (A : in out Low_Alert);

type Medium_Alert is new Alert with ... end record;
procedure Handle (A : in out Medium_Alert);

type High_Alert is new Alert with ... end record;
procedure Handle (A : in out High_Alert);
```

# Added Operations

```
with Alerts; use Alerts;
procedure Client is
    A   : Alert;
    A_L : Low_Alert;
    A_M : Medium_Alert;
    A_H : High_Alert;

begin
    Set_Alarm (A, 1800);
    Set_Alarm (A_L, 1800);
    Set_Alarm (A_M, 1800);

    Set_Alarm (A_H, 1800);
end Client;
```

# Variant Programming

```
procedure Handle (A : in out Alert) is
begin
   A.Time_Of_Arrival := Calendar.Clock;
   A.Cause           := Get_Cause (A);
   Log (A);
   case A.P is
      when Low    =>
         null;
      when Medium =>
         A.Technician := Assign_Technician;
      when High   =>
         A.Engineer := Assign_Engineer;
         Set_Alarm (A, Wait => 1800);
      end case;
end Handle;
```

# Programming with Inheritance

```
procedure Handle (A : in out Alert) is
begin
   A.Time_Of_Arrival := Calendar.Clock;
   A.Cause           := Get_Cause (A);
   Log (A);
end Handle;
```

```
procedure Handle (A : in out Medium_Alert) is
begin
   Handle (Alert (A)); --  First handle as plain Alert

   A.Technician := Assign_Technician;
end Handle;
```

```
procedure Handle (...) is
begin
   A.Time_Of_Arrival := ...;
   A.Cause           := ...;
   Log (A);
   case A.P is
      when Low    =>
         null;
      when Medium =>
         A.Technician := ...;
      when High   =>
         A.Engineer := ...;
         ...);
```

```
procedure Handle (A : in out Alert) is
begin
   A.Time_Of_Arrival := Calendar.Clock;
   A.Cause           := Get_Cause (A);
   Log (A);
end Handle;
```

```
procedure Handle (A : in out High_Alert) is
begin
   Handle (Alert (A)); --  First handle as plain Alert

   A.Engineer := Assign_Engineer;
   Set_Alarm (A, Wait => 1800);
end Handle;
```

```
procedure Handle (...) is
begin
   A.Time_Of_Arrival := ...;
   A.Cause           := ...;
   Log (A);
   case A.P is
      when Low    =>
         null;
      when Medium =>
         A.Technician := ...;
      when High   =>
         A.Engineer := ...;
         Set_Alarm (A, ...);
   end case;
end Handle;
```

# Centralized vs Distributed Code

- The code which is **centralized** in the same routine in the *functionality-oriented* version
- is now **distributed** across 3 different routines in the *object-oriented* version

- **Object-Oriented Organization**
  - inheritance (simple)
    - ***encapsulation & inheritance***
  - polymorphism
  - abstract types & subprograms
  - modifying an OO system
  - when to use OO organization

```
with …;

package Alerts is
      type Alert is tagged private;

      procedure Handle (A : in out Alert);
      procedure Log      (A : Alert);

 private
      type Alert is tagged record
          Time_Of_Arrival : Calendar.Time;
          Cause                : String (1 .. 200);
      end record;
end Alerts;
```

# Two possibilities to extend Alert

- **Child package** to access fields
  - **Time_Of_Arrival**
  - **Cause**
- **Normal package** if you do not need to access
  - **Time_Of_Arrival**
  - **Cause**

# Child Package

```
with Alerts; use Alerts;
with …;
package Alerts.Medium is
      type Medium_Alert is new Alert with private;
      procedure Handle (A : in out Medium_Alert);
 private
      type Medium_Alert is new Alert with record
           Technician  : Person;
      end record;
end Alerts.Medium;
```

```
package body Alerts.Medium is


end Alerts.Medium;
```

Can access fields
- Time_Of_Arrival
- Cause

# Regular Package

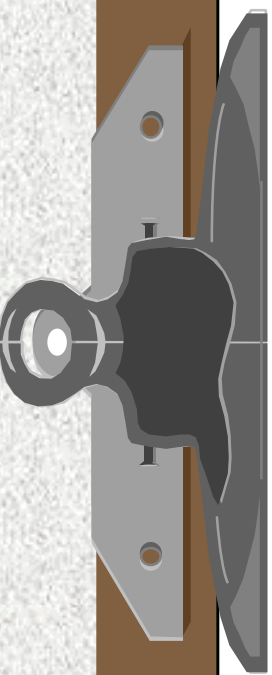
```
with Alerts; use Alerts;
package High_Importance is
      type High_Alert is new Alert with private;
      procedure Handle (A : in out High_Alert);
      procedure Set_Alarm (A : in out High_Alert; W : Duration);
 private
      type High_Alert is new Alert with record
           Engineer  : Person;
           Ring_Alarm_At : Calendar.Time;
      end record;
end High_Importance;
```

```
package body High_Importance is


nd High_Importance;
```

Cannot access fields
- Time_Of_Arrival
- Cause

# Important Remark

- Adding a new type derived from Alert
  - No need to **modify** what is working already
  - No need to retest what you did already
- Just add the data type in a separate package (regular or child package)



- **Object-Oriented Organization**
  - inheritance (simple)
  - ***polymorphism***
  - abstract types & subprograms
  - modifying an OO system
  - when to use OO organization

# Handling an Alert

- You have a **Get_Alert** routine
- Connected to the sensors in the factory
- Collects the alerts

```
with Alerts; use Alerts;
function Get_Alert return ???;
```

# Objective

- Be able to mimic the code used in the variant programming case

```
with Alerts; use Alerts;
with Get_Alert;
procedure Process_Alerts is
begin
    loop  --  infinite loop
        Handle (Get_Alert);
    end loop;
end Process_Alerts;
```

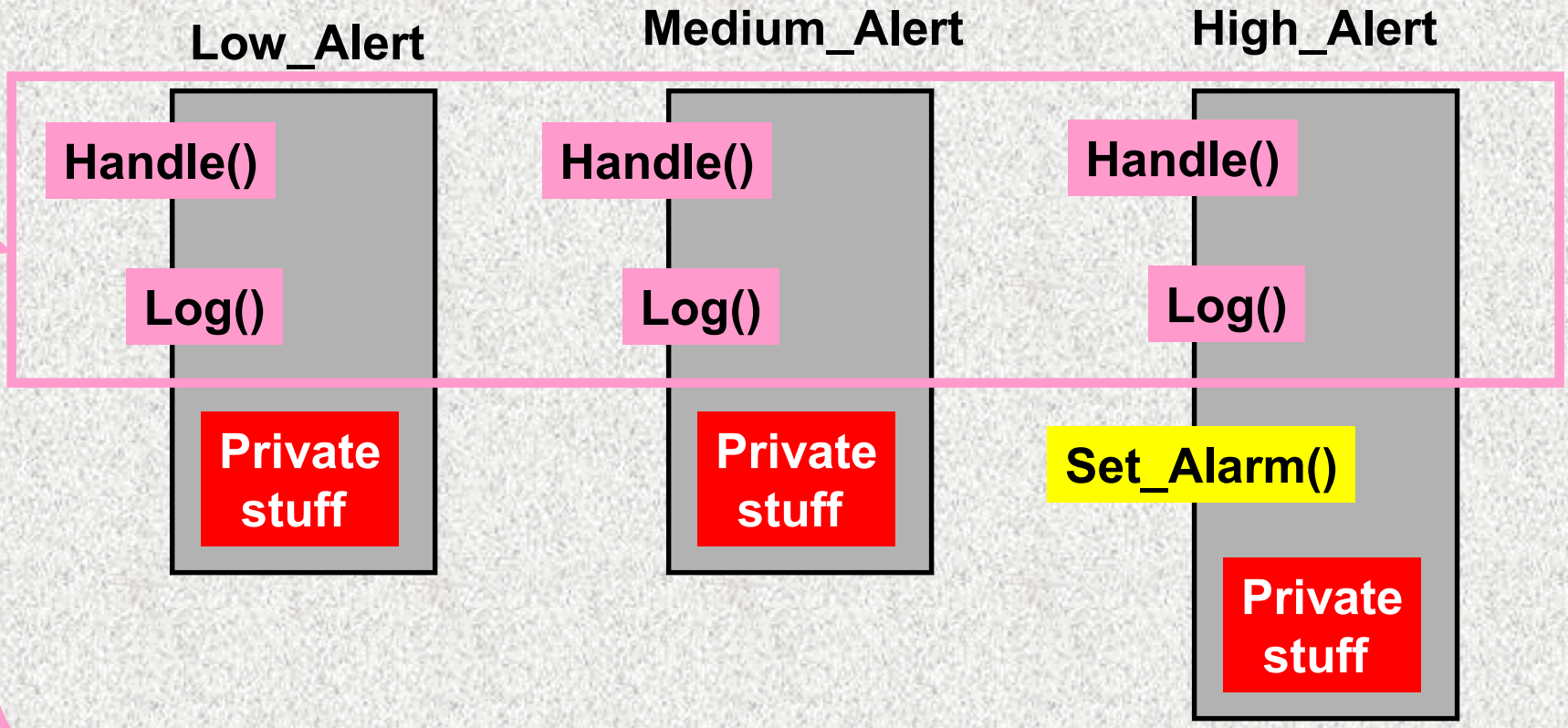
# HOW ?

- 4 different **Handle** routines depending on the type of the alert object returned
- How can **Get_Alert** return objects of different types ?

```
type Alert is tagged record  ...  end record;
procedure Handle (A : in out Alert);

type Low_Alert is tagged record  ...  end record;
--  procedure Handle (A : in out Low_Alert);

type Medium_Alert is new Alert with ...  end record;
procedure Handle (A : in out Medium_Alert);

type High_Alert is new Alert with ... end record;
procedure Handle (A : in out High_Alert);
```

# Polymorphism

- Variables can name objects with common properties but ***different types***
- Can ***select dynamically*** the right ***operation*** for the underlying object




**The exact same interface because they all derive from type** Alert

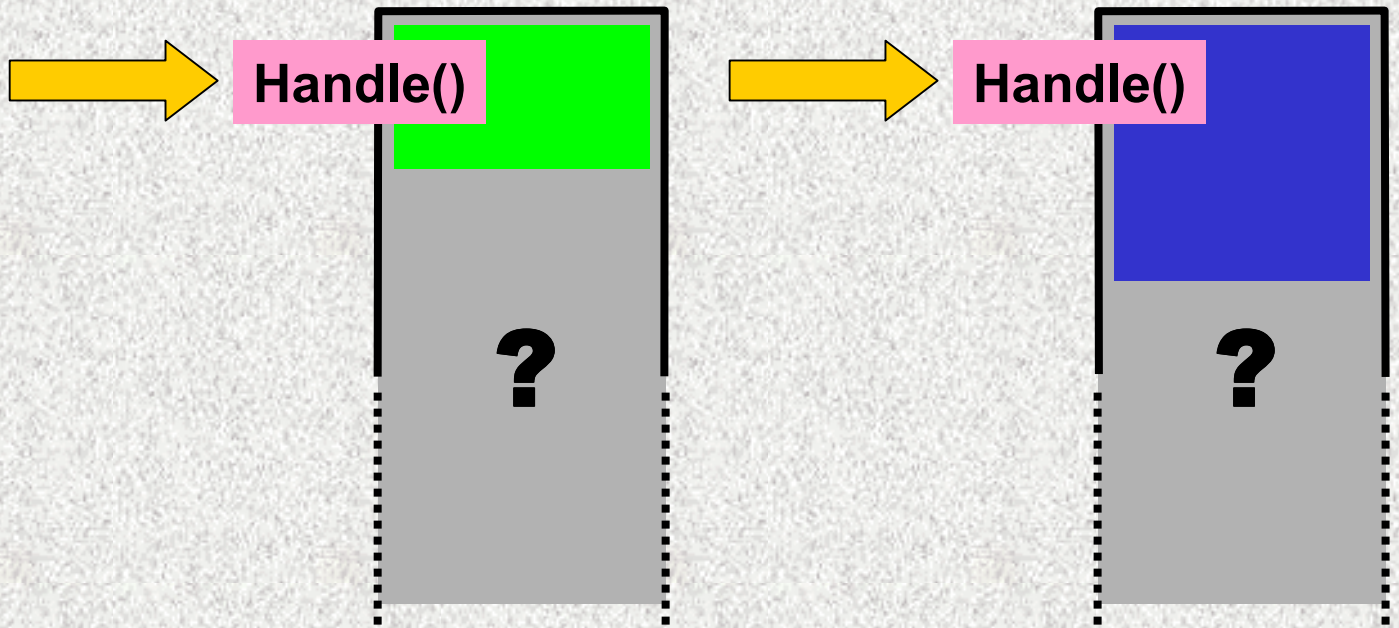
# Inheritance & Interfaces

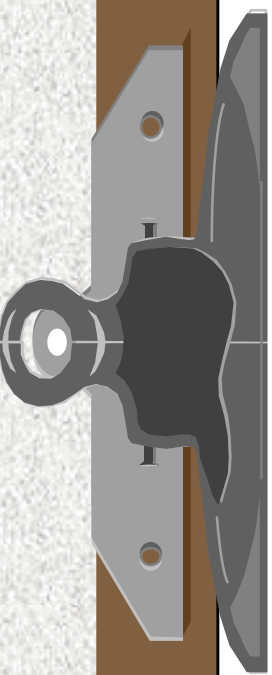
- All type **T** derived from **Alert** must implement or inherit:
  - **procedure** Handle (A : T);
  - **procedure** Log (A : T);
- Cannot remove inherited operations, you can only redefine their implementation

# Idea: select the operation dynamically

**Obj** : *some unknown type derived from* Alert;

**Handle (Obj);**





- **Object-Oriented Organization**
  - inheritance (simple)
  - polymorphism
    - *tags & class wide types*
    - dynamic dispatching
    - using access parameters
    - redispatching
  - abstract types & subprograms
  - modifying an OO system
  - when to use OO organization

# Generally Speaking ...

*For any tagged type* ***T***

***T'Class***

*denotes ANY type* ***D*** *derived from* ***T***

$$\forall T$$

$$T'Class = \{D \mid D \xrightarrow{\text{derives from}} T\}$$

$$\forall T$$

$$Values(T'Class) = \bigcup_{D \to T} Values(D)$$

# Inheritance Theorem

*For all type* ***D*** *derived from* ***T***

*set of operations implemented for objects of type* ***T***

$\subseteq$

*set of operations implemented for objects of type* ***D***

$$\forall\, D \in T'Class$$

$$\exists\ \mathrm{Op}(\mathrm{X}:\mathrm{T};\ldots) \Rightarrow \exists\ \mathrm{Op}(\mathrm{X}:\mathrm{D};\ldots)$$

$$Operations(T'Class) = Operations(T)$$

CLASS = Values
+ Operations
+ Inheritance

# T'Class in Ada

- Conversions:
  - A value of any type derived from T can be implicitly converted to type T'Class
  - conversion from T'Class to a type derived from T checks the tag
- T'Class is treated like an unconstrained type
- A variable of type T'Class must be initialized

# Class-Wide Objects

```
A_L1 : Low_Alert;
A_L2 : Low_Alert;
A_M  : Medium_Alert;
A_H  : High_Alert;

V1 : Alert'Class := A_L1;
V2 : Alert'Class := A_M;
V3 : Alert'Class := A_H;

V1 := A_L2;

A_L1 := Low_Alert (V1);
```

OK - the type of the object named by V1 and A_L2 is the same (a Low_Alert)

OK - V1 names an object of type Low_Alert

```
A_L1 : Low_Alert;
A_L2 : Low_Alert;
A_M  : Medium_Alert;
A_H  : High_Alert;

V1 : Alert'Class := A_L1;
V2 : Alert'Class := A_M;
V3 : Alert'Class := A_H;

V3 := A_L1;

A_L1 := Low_Alert (V2);
```

**Constraint_Error**
objects named by the 2 variables have different types

V1 : Alert'Class;

# Class-Wide Access

```
type All_Alert_Ptr is access all Alert'Class;

Ptr_1 :  All_Alert_Ptr;  --  no need to default initialize

Ptr_1 := new Alert;
Ptr_1 := new Low_Alert'(Get_Time, Get_Cause);
Ptr_1 := new Medium_Alert;
Ptr_1 := new High_Alert;
```

General access type to all conversions

```
type All_Alert_Ptr  is access all Alert'Class;
type Low_Alert_Ptr is access all Low_Alert'Class;

Ptr_M :  All_Alert_Ptr;
Ptr_L  :  Low_Alert_Ptr := new Low_Alert;

Ptr_M := new High_Alert;


Ptr_L := Low_Alert_Ptr (Ptr_M);
```

**Constraint_Error**
object pointed by Ptr_M must be in Low_Alert'Class

# Class-Wide Membership

Given a class-wide variable

```
V : T'Class := ...;
```

you can ask whether

```
V  in  D'Class
```

for all type D derived from T

```
A_L  : Low_Alert;

V1 : Alert'Class := A_L;

B : Boolean;


B := (V1 in Alert ' Class);
```
True

```
B := (V1 in Low_Alert ' Class);
```
True

```
B := (V1 in Medium_Alert ' Class);
```
False

# Run Time Type Identification

- Tag = ID of a tagged type
- For every type T
  - **T'Tag** returns the tag of T
- For every class wide variable V
  - **V'Tag** returns the tag of the type of the object named by V

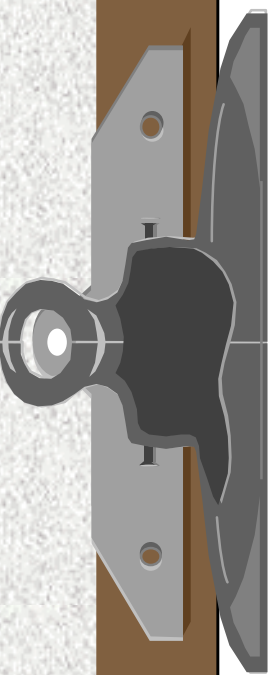
# 'Tag Attribute

```
with Ada.Tags;  use Ada.Tags;
procedure Client is
   A_L1 : Low_Alert;
   A_L2 : Low_Alert;
   A_M  : Medium_Alert;

   V1 : Alert'Class := A_L1;
   V2 : Alert'Class := A_L2;
   V3 : Alert'Class := A_M;

   B : Boolean;
begin
   B := (V1 ' Tag = V2 ' Tag);
   B := (V2 ' Tag = V3 ' Tag);
   B := (V1 ' Tag = Low_Alert ' Tag);
```

True

False

True



- **Object-Oriented Organization**
  - inheritance (simple)
  - polymorphism
    - tags & class wide types
    - ***dynamic dispatching***
    - using access parameters
    - redispatching
  - abstract types & subprograms
  - modifying an OO system
  - when to use OO organization

# Handling an Alert

- You have a **Get_Alert** routine
- Connected to the sensors in the factory
- Collects the alerts

```
with Alerts; use Alerts;
function Get_Alert return Alert'Class;
```

# Dynamic Dispatching

```
with Alerts; use Alerts;
with Get_Alert;
procedure Process_Alerts is
begin
    loop  --  infinite loop
        declare
            A : Alert'Class := Get_Alert;
        begin
            Handle (A);  --  could have written Handle (Get_Alert);
        end;
    end loop;
end Process_Alerts;
```

# Which Handle () is Called ?

- A'Tag = Alert'Tag

  Handle (A: Alert)

- A'Tag = Low_Alert'Tag

  Handle (A: Low_Alert)

- A'Tag = Medium_Alert'Tag

  Handle (A: Medium_Alert)

- A'Tag = High_Alert'Tag

  Handle (A: High_Alert)

# Static vs Dynamic Binding

***STATIC BINDING*** = call known at **compile time**

***DYNAMIC BINDING*** = call known only at **run time**

# How do you know if call Op (V, …) is dispatching ?

- Type of **V** is **T'Class** for some tagged type **T**

- **Op** is a *primitive operation* of **T**

```
type T is … end record;

procedure Op (P : T; …)                    (or function)
procedure Op (P : in out T; …)
procedure Op (P : access T; …)             (or function)
```

```
AL : Low_Alert;

Handle (AL);
```

Static Binding

```
A  : Alert'Class := Get_Alert;

Handle (A);
```

Dynamic Binding

```
A  : High_Alert'Class := ...;

Handle (A);
```

Dynamic Binding

```
type Low_Alert_Ptr is access all Low_Alert;
Ptr   : Low_Alert_Ptr := …;

Handle (Ptr.all);
```

Static Binding

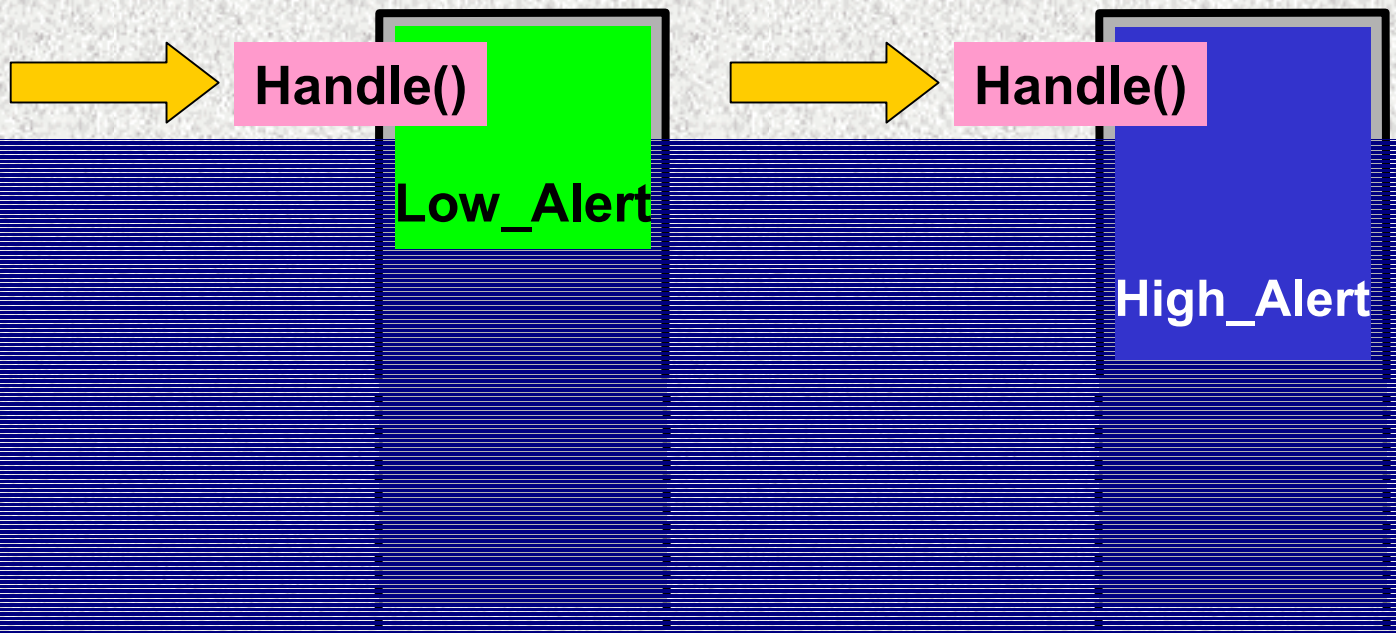
```
type Low_Alert_Class_Ptr is access all Low_Alert'Class;
Ptr   : Low_Alert_Class_Ptr := …;

Handle (Ptr.all);
```

Dynamic Binding

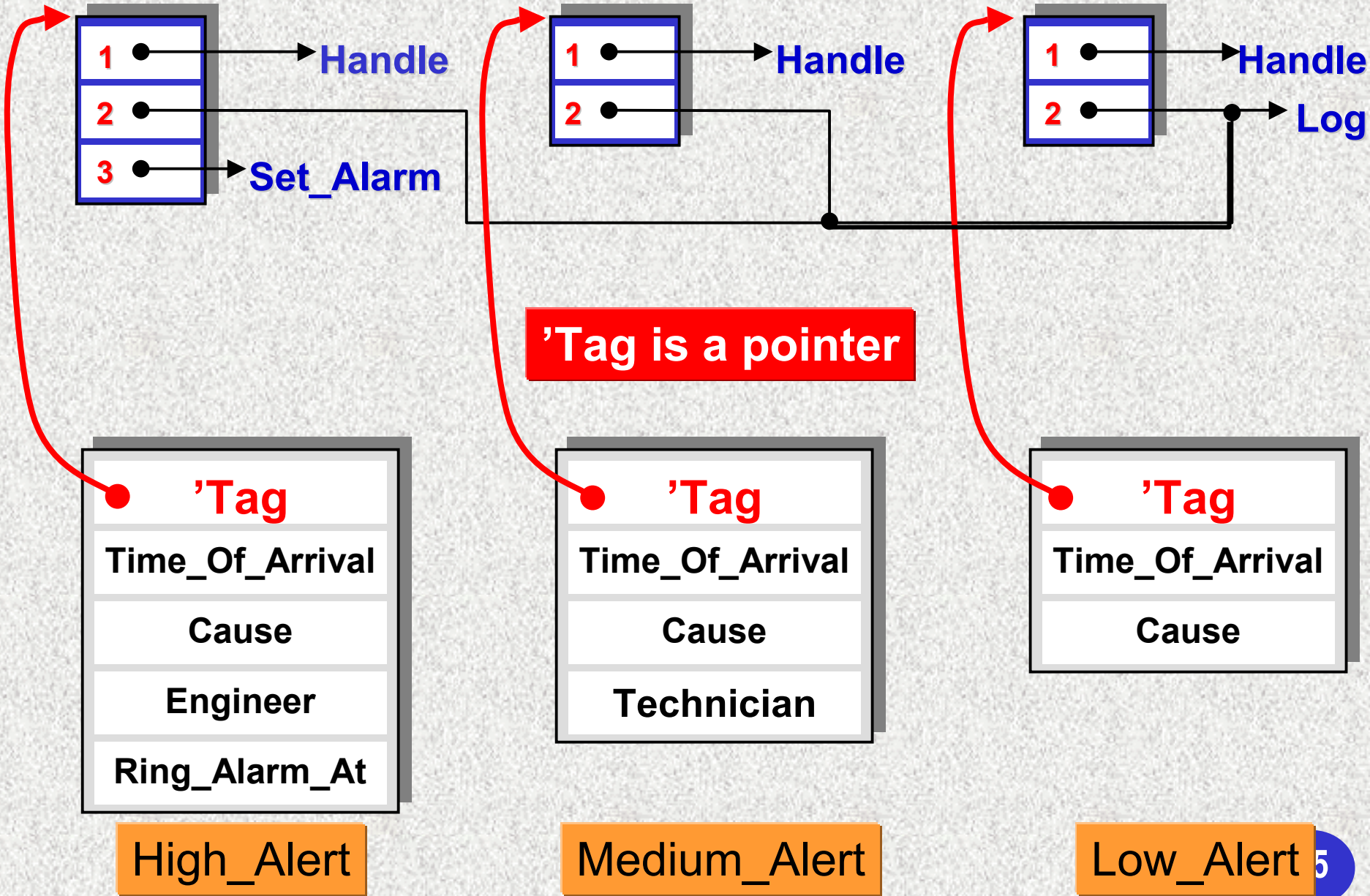
# Where is the magic ?

```
A : Alert'Class := Get_Alert;

Handle (A);
```

Dynamic Binding



# Tables of pointers to primitive operations



```
A : Alert'Class := Get_Alert;

Handle (A);
```

indirect call to operation pointed by A'Tag (1)

```
A : Alert'Class := Get_Alert;

Log (A);
```

indirect call to operation pointed by A'Tag (2)

# Can we DOUBLE dispatch ?

```
type Base is tagged null record;

procedure Op (X : Base; Y : Base);
```

```
type Deriv is new Base with null record;

procedure Op (X : Deriv; Y : Deriv);
```

```
V1 : Base'Class := ...;
V2 : Base'Class := ...;

Op (V1, V2);
```

Dynamic Binding

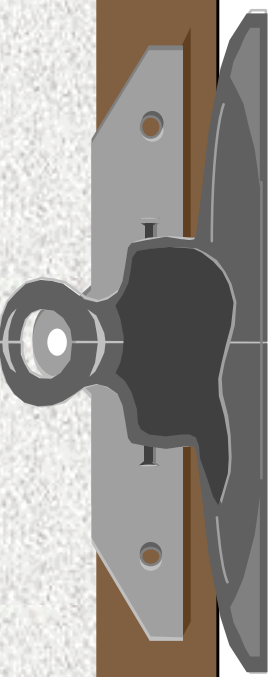
If V1'Tag /= V2'Tag raises Constraint_Error

# What about ...

```
type T1 is tagged null record;
type T2 is tagged null record;

procedure Op (X : T1; Y : T2);
```


**operation can be dispatching in only one type**

- **Object-Oriented Organization**
  - inheritance (simple)
  - polymorphism
    - tags & class wide types
    - dynamic dispatching
    - *using access parameters*
    - redispatching
  - abstract types & subprograms
  - modifying an OO system
  - when to use OO organization



skip

# Sometime it is convenient to use pointers:

## access parameters

```
package Alerts is
   type Alert is tagged private;

   procedure Handle (A : access Alert);
   procedure Log    (A : access Alert);

private
   type Alert is tagged record
      Time_Of_Arrival : Calendar.Time;
      Cause           : String (1 .. 200);
   end record;
end Alerts;
```

```
package Alerts.Medium is
   type Medium_Alert is new Alert with private;

   procedure Handle (A : access Alert);

private
   type Medium_Alert is new Alert with record
      Technician  : Person;
   end record;
end Alerts.Medium;
```

# Handling Alerts using access parameters

```
type Alert_Class_Ptr is access all Alert ' Class;

function Get_Alert return Alert_Class_Ptr;
```

```
declare
    A : Alert_Class_Ptr := Get_Alert;
begin
    Handle (A);  --  could have written Handle (Get_Alert);
```

Dynamic Binding

# Polymorphism is powerful

- Alerts are buffered in a linked list

```
type Alert_Node;
type Alert_List is access Alert_Node;

type Alert_Node is record
    An_Alert : Alert_Class_Ptr;
    Next     : Alert_List;
end record;

function Get_Alerts return Alert_List;
```

- All the alerts are processed uniformly

```
Ptr : Alert_List := Get_Alerts;

while Ptr /= null loop
    Handle (Ptr.An_Alert);
    Ptr := Ptr.Next;
end loop;
```

# Rules for access parameter conversions

```
type T is tagged … end record;
type T_Ptr is access all T;

procedure Op (X : access T);
```

T is some tagged type

```
P : T_Ptr := …;

Op (P);
```

Static Binding

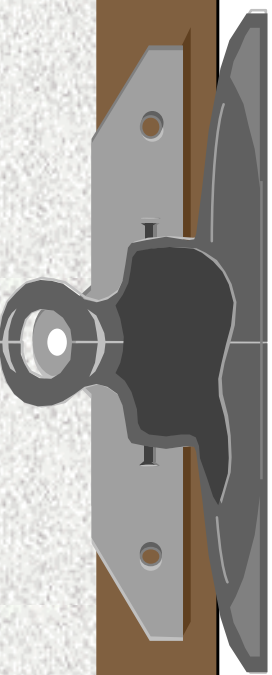
implicit conversion

```
type T is … end record;
type T_Class_Ptr is access all T'Class;

procedure Op (X : access T);
```

```
PC : T_Class_Ptr := ...;

Op (PC);
```

Dynamic Binding


```
procedure Op (X : access T) is

   P    : T_Ptr          := T_Ptr (X);

   PC : T_Class_Ptr := T_Class_Ptr (X);

begin
   …
end Op;
```



- **Object-Oriented Organization**
  - inheritance (simple)
  - polymorphism
    - tags & class wide types
    - dynamic dispatching
    - using access parameters
    - ***redispatching***
  - abstract types & subprograms
  - modifying an OO system
  - when to use OO organization

```
procedure Handle (A : in out Alert) is
begin
   A.Time_Of_Arrival := Calendar.Clock;
   A.Cause           := Get_Cause (A);
   Log (A);
```

Static Binding

always calls: **procedure** Log (A : Alert);

```
procedure Handle (A : in out Medium_Alert) is
begin
   Handle (Alert (A)); --  First handle as plain Alert

   A.Technician := Assign_Technician;
end Handle;
```

# What if …
# … we override Log

```
package Alerts is
   type Alert is tagged private;
   procedure Handle (A : in out Alert);
   procedure Log    (A : Alert);

   type Medium_Alert is new Alert with private;
   procedure Handle    (A : in out Medium_Alert);
   procedure Log (A : Medium_Alert);
private
   ....
end Alerts;
```

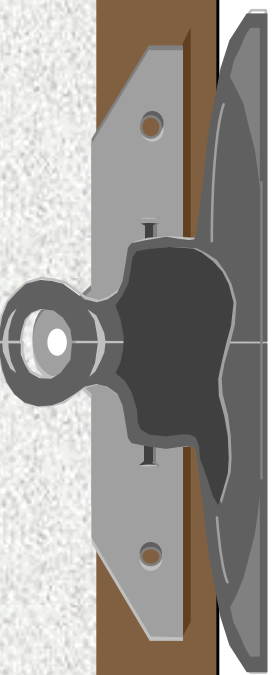
```
procedure Handle (A : in out Alert) is
begin
   A.Time_Of_Arrival := Calendar.Clock;
   A.Cause           := Get_Cause (A);
   Log (Alert'Class (A));
end Handle;
```

Dynamic Binding
Redispatching

```
procedure Handle (A : in out Medium_Alert) is
begin
   Handle (Alert (A)); --  First handle as plain Alert

   A.Technician := Assign_Technician;
end Handle;
```

# Dispatching Philosophy

- **Ada:**
  - All primitive operations are potentially dispatching
  - Decide when to have a dispatching call
- **C++:**
  - Decide which methods are dispatching (virtual methods)
  - All calls to these functions are dispatching by default
- **Java:**
  - All primitive operations are dispatching
  - all calls are dispatching



- **Object-Oriented Organization**
  - inheritance (simple)
  - polymorphism
  - ***abstract types & subprograms***
  - modifying an OO system
  - when to use OO organization

# In the Alert example ...

- One could create objects of type **Alert** rather than
  - **Low_Alert**, **Medium_Alert**, **High_Alert**
- Undesirable if plain **Alert** has no significance but is used only to transmit:
  - *Fields*: **Time_Of_Arrival** & **Cause**
  - *Methods*: **Handle** & **Log**

# Make Alert an abstract type

```
package Alerts is
   type Alert is abstract tagged private;

   procedure Handle (A : in out Alert);
   procedure Log    (A : Alert);

private
   type Alert is tagged record
      Time_Of_Arrival : Calendar.Time;
      Cause           : String (1 .. 200);
   end record;
end Alerts;
```

# Cannot create objects of an abstract type

```
type Alert is abstract tagged private;

A : Alert;
```

# Can have abstract operations

```
package Alerts is
   type Alert is abstract tagged private;

   procedure Handle (A : in out Alert);
   procedure Log    (A : Alert) is abstract;

private
   type Alert is tagged record
      Time_Of_Arrival : Calendar.Time;
      Cause           : String (1 .. 200);
   end record;
end Alerts;
```

# Rules for abstract operations

- Do not provide the body of an abstract operation
- Every non abstract type derived from an abstract type must provide the body of all abstract operations

```
package Alerts is
   type Alert is abstract tagged private;
   procedure Handle (A : in out Alert);
   procedure Log    (A : Alert) is abstract;

   type Low_Alert is new Alert with private;
   procedure Log (A : Low_Alert);
```

**Must provide Log or make Low_Alert abstract**

- Only dispatching calls to abstract routines allowed

```
procedure Handle (A : in out Alert) is
begin
   . . .
   Log (A);
end Handle;
```

Compilation error

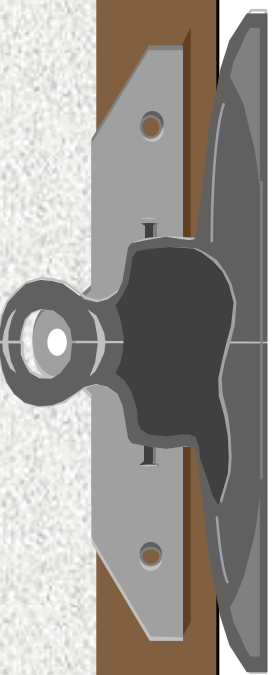
**procedure** Log (A : Alert);
does not exist

```
procedure Handle (A : in out Alert) is
begin
   . . .
   Log (Alert'Class (A));
en
```

OK

Dispatching Call

- **Object-Oriented Organization**
  - inheritance (simple)
  - polymorphism
  - abstract types & subprograms
  - ***modifying an OO system***
  - when to use OO organization



# Adding a NEW Type...

- **Do not modify** what is working already
  - No need to retest what you already did since you do not need to touch it
- Just add the data type  in a separate package (regular or child package)

```
package Alerts is
   type Alert is abstract tagged private;
   procedure Handle (A : in out Alert);
   procedure Log    (A : Alert);
private
   . . .
end Alerts;
```

```
with Alerts; use Alerts;
package Alerts.Medium is
   type Medium_Alert is new Alert with private;
   procedure Handle (A : in out Alert);
   procedure Log    (A : Alert);
private
   . . .
end Alerts.Medium;
```

# Adding NEW Functionality…

- **Have to modify** the spec containing tagged type T to which we add the functionality
- **Have to modify** all the packages containing types derived from T to implement the new functionality
  - **Error Prone** & **labor intensive**
  - need to **retest everything** for regressions

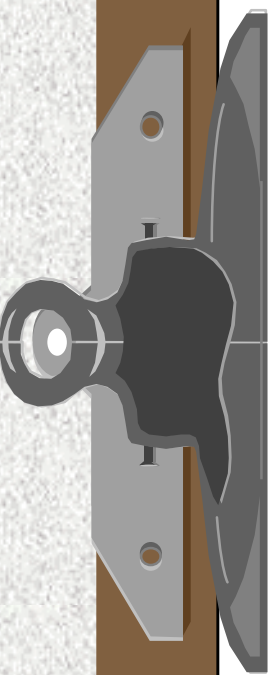
# Example

- Suppose you want to add a new functionality
- that behaves **DIFFERENTLY** for all alert types

```
package Alerts is
   type Alert is abstract tagged private;
   procedure New_Functionality (A : Alert);
   . . .
```

```
with Alerts; use Alerts;
package Alerts.Medium is
   type Medium_Alert is new Alert with private;
   procedure New_Functionality (A : Medium_Alert);
   . . .
```

```
with Alerts; use Alerts;
package Alerts.High is
   type High_Alert is new Alert with private;
   procedure New_Functionality (A : High_Alert);
   . . .
```



- **Object-Oriented Organization**
  - inheritance (simple)
  - polymorphism
  - abstract types & subprograms
  - modifying an OO system
  - ***when to use OO organization***

- System Functionalities are well understood before starting the design
- Adding new functionality will happen infrequently
- Will add lots of new data types with the same functionality over the life time of the system