dit
UPM

Programación de sistemas grandes

Juan Antonio de la Puente
DIT/UPM

Objetivos

- Repasaremos los elementos de los lenguajes de programación que facilitan la realización de sistemas grandes
- Veremos cómo usar módulos o paquetes para descomponer un sistema y abstraer los detalles de realización
- Veremos también cómo realizar componentes de software que se puedan reutilizar en varios sistemas

Contenido

- **Introducción**
- Ocultamiento de información
- Tipos abstractos de datos
- Compilación separada
- Objetos
- Reutilización
- Programación en C

Descomposición y abstracción

- Descomposición

  División de un sistema complejo en componentes más sencillos, hasta conseguir que cada componente pueda ser comprendido y realizado por una o varias personas

  – **diseño descendente**

- Abstracción

  Especificación de los aspectos esenciales de un componente, dejando para más adelante su diseño detallado

  – **diseño ascendente**

Módulos

- Un módulo es una colección de declaraciones de tipos, objetos y operaciones relacionados entre sí
- Los módulos permiten ***encapsular*** partes del sistema mediante interfaces bien definidas
- Los módulos permiten utilizar algunas técnicas que facilitan el desarrollo de sistemas grandes:
  - ocultamiento de información
  - tipos abstractos de datos
  - compilación separada

Contenido

- Introducción
- **Ocultamiento de información**
- Tipos abstractos de datos
- Compilación separada
- Objetos
- Reutilización
- Programación en C

Ocultamiento de información

- Los módulos permiten controlar la visibilidad de las declaraciones
- Para ello se distingue entre
  - interfaz visible (especificación)
  - cuerpo invisible (implementación)
    - » contiene los detalles que no son visibles desde fuera
- Es deseable poder compilar la especificación sin necesidad de escribir el cuerpo
- Ejemplos
  - Ada: paquetes
  - C: ficheros .h y .c

Paquetes en Ada

- Un paquete (*package) tiene dos partes:*
  - **especificación:** contiene únicamente declaraciones visibles desde otras partes del programa (la *interfaz* del paquete)
  - **cuerpo**: contiene los detalles invisibles desde fuera
- Todos los identificadores definidos en la especificación son visibles a la vez
- Hay una relación formal entre la especificación y el cuerpo

Ejemplo (1)

- Especificación de paquete (`greetings.ads`)

```
package Greetings is
   -- procedimientos para saludar
   procedure Hello;
   procedure Goodbye;
end Greetings;
```

- Contiene la especificación de dos procedimientos
- Los cuerpos van en el cuerpo del paquete

Ejemplo (2)

- Cuerpo del paquete (`greetings.adb`)

```
with Ada.text_IO;          use Ada.Text_IO;
package body Greetings is

   procedure Hello is
   begin
      Put_Line ("Hello);
   end Hello;

   procedure Goodbye is
   begin
      Put_Line ("Goodbye");
   end Goodbye;

end Greetings;
```

Uso de los paquetes

- Los identificadores declarados en la especificación van cualificados con el nombre del paquete

```
Greetings.Hello;
```

- Se puede evitar la cualificación con una cláusula ***use***:

```
with Greetings; use Greetings;
procedure Greet is
begin
   Hello;
   Goodbye;
end Greet;
```

Contenido

- Introducción
- Ocultamiento de información
- **Tipos abstractos de datos**
- Compilación separada
- Módulos genéricos
- Objetos
- Reutilización
- Programación en C

Tipos abstractos de datos

- Un TAD es un ***tipo de datos*** declarado conjuntamente con un conjunto de ***operaciones primitivas***
- Los detalles de representación del tipo se ocultan
  - se dice que el tipo es ***privado***
- Los detalles sobre la representación del tipo van en una *parte privada* de la especificación del paquete
  - la parte privada define la *interfaz física* del paquete y es invisible desde fuera del mismo
  - la utiliza el compilador para crear objetos del tipo abstracto
- Los tipos ***limitados*** no tienen operaciones predefinidas
- Los demás tipos privados tienen predefinida la asignación (":=") y la igualdad ("=")

Ejemplo: TAD cola (1)

```
package Queues is
   -- tipo abstracto
   type Queue is limited private;
   -- operaciones primtivas
   procedure Create (Q : in out Queue);
   function  Empty  (Q :        Queue) return Boolean;
   procedure Insert (Q : in out Queue; E :     Element);
   procedure Remove (Q : in out Queue; E : out Element);
```

Ejemplo: TAD cola (2)

```
private
  -- estas declaraciones no son visibles desde fuera
  type Queue_Node;
  type Queue_Node_Pointer is access Queue_Node;
  type Queue_Node is
    record
      Contents : Element;
      Next     : Queue_Node_Pointer;
    end record;

  type Queue is limited
    record
      Front    : Queue_Node_Pointer;
      Back     : Queue_Node_Pointer;
    end record;

end Queues;
```

Ejemplo: TAD cola (3)

```
package body Queues is
   ...
end Queues;
```

```
declare
  use Queues;
  Q1, Q2 : Queue;
  E : Element;
begin
  if not Empty (Q1) then
    Remove (Q1,E); Insert (Q2, E);
  end if;
end;
```

Contenido

- Introducción
- Ocultamiento de información
- Tipos abstractos de datos
- **Compilación separada**
- Módulos genéricos
- Objetos
- Reutilización
- Programación en C

Compilación separada

- Es interesante compilar por separado los módulos de un programa grande
- Conviene distinguir
  - ***Compilación independiente***
    - » los módulos se compilan por separado, pero el compilador no comprueba la interdependencias
    - » la resolución de las dependencias se hace al montar el programa
    - » los errores aparecen al ejecutar el programa
  - ***Compilación separada***
    - » el compilador comprueba el uso de las definiciones de unos módulos en otros
    - » se pueden detectar errores al compilar

Compilación separada en Ada

- Un programa grande se puede dividir en ***unidades de compilación***
- En Ada las unidades de compilación pueden ser:
  - especificaciones de paquetes
  - especificaciones de subprogramas
  - los cuerpos de paquetes y subprogramas son ***unidades secundarias***
- La dependencia entre unidades se indica con una cláusula ***with***

```
with Greetings; use Greetings;
procedure Greet is
begin
   Hello;
   Goodbye;
end Greet;
```

Compilación separada con GNAT

- Ahora hay tres ficheros:

```
greetings.ads        (especificación de Greetings)
greetings.adb        (cuerpo de Greetings)
greet.adb            (cuerpo del procedimiento Greet)
```

- Para compilar se hace:

```
$ gcc -c greet.adb
$ gcc -c greetings.adb
```

  - » ¡ no hace falta compilar greetings.ads !
  - » no importa el orden de compilación

- Montaje y enlace :

```
$ gnatbind greet.ali
$ gnatlink greet.ali
```

- Se puede hacer todo de una vez:

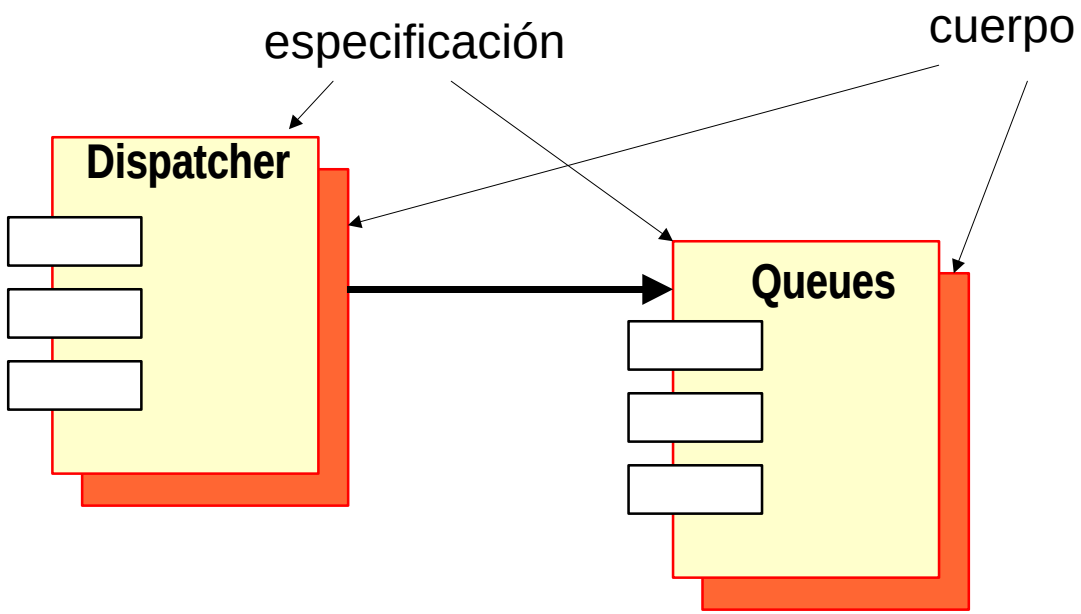
```
$ gnatmake greet
```

Biblioteca de compilación

- Las unidades que forman un programa se almacenan en una *biblioteca de compilación*
- La cláusula *with* establece una relación de dependencia entre unidades
- Se pueden escribir (y compilar) las especificaciones antes que los cuerpos
  - el compilador comprueba que las interfaces son consistentes
  - esto es adecuado para implementar hacia arriba *(bottom-up)*
  - es compatible con el diseño descendente *(top-down)*
  - los cuerpos dependen de las especificaciones

Diagramas de módulos

- Permiten representar gráficamente las dependencias entre unidades de compilación



Paquetes sin cuerpo

- Los paquetes cuya especificación está completa no tienen cuerpo
- Por ejemplo, si sólo se declaran tipos, constantes y variables
- Ejemplo:

```
package Elevation_Definitions is

  type Elevation_Angle is digits 5 range -30.0 .. 90.0;

  Maximum_Elevation : constant Elevation_Angle := 60.0;
  Current_Elevation :          Elevation_Angle;

end Elevation_Definitions;
```

Unidades separadas

- La cláusula ***separate*** facilita la implementación hacia abajo
  - se puede sustituir un cuerpo por un ***resguardo*** *(stub)* que se compila más tarde

```
procedure Main is
   type Reading is ...
   type Control_Value is ...
   procedure Convert (R : Reading; V : out Control_Value)
     is separate;
begin  -- Main
   loop
      Get (R); Convert (R,V); Put (V);
   end loop;
end Main;
```

```
separate (Main)
procedure Convert (R : Reading; V : out Control_Value) is
   -- cuerpo del procedimiento
end Convert;
```

Biblioteca jerárquica

- Cuando un programa se hace muy grande, puede haber conflictos de nombres entre unidades de programa
- Para evitar esto, Ada tiene una **biblioteca jerárquica**
  - Un paquete *P* puede tener uno o más ***«hijos»*** *P.Q*
  - La especificación de *P.Q* es una *extensión* de la especificación de *P*
    - » En la parte pública sólo se ve la *parte pública* de *P*
    - » En la parte privada de *P.Q* se ve también la parte privada de *P*
  - El cuerpo de *P.Q* implementa la extensión
    - » Aquí también se ve la parte privada de *P*
  - Cuando se hace *with P.Q* se importa también *P*
    - » Pero *use P.Q* no implica *use P*
  - La jerarquía se puede hacer tan profunda como sea necesario
  - La especificación de los hijos depende de la del padre

Ejemplo (1)

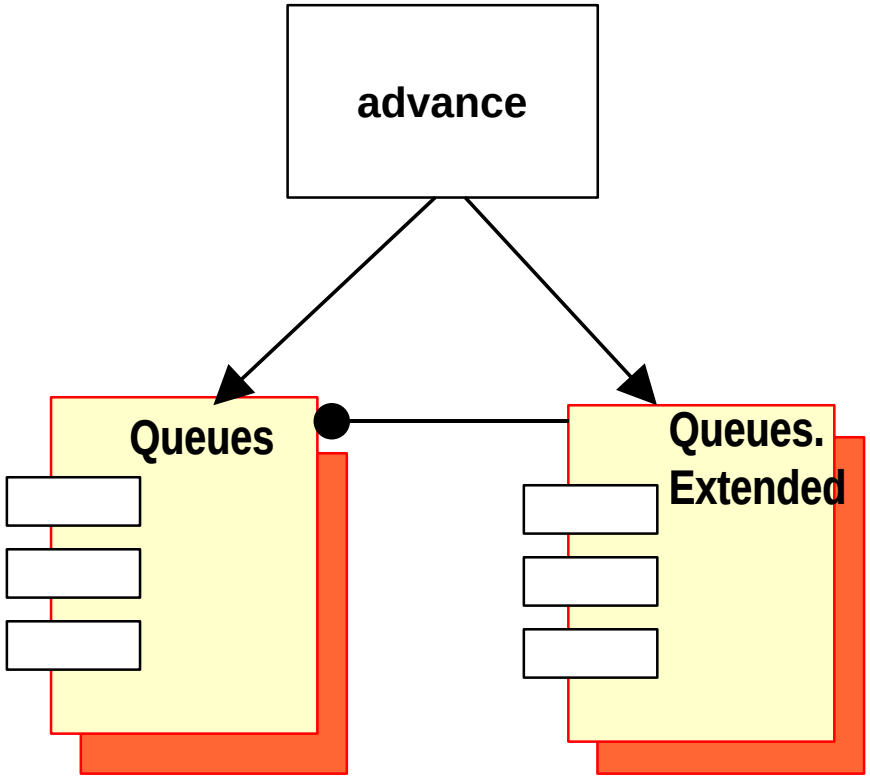
```
package Queues.Extended is
   function First (Q : in out Queue) return Element;
end Queues.Extended;
```

```
package body Queues.Extended is

   function First (Q : in out Queue) return Element
is
   begin
      return Q.Front.Contents;
   end First;

end Queues.Extended;
```

Ejemplo (2)

```
with Queues.Extended; -- se importa también Queues
use  Queues, Queues.Extended;
procedure Advance (Q : Queue) is
-- remove first element if equal to some value
   E : Element;
begin
   if First (Q) = Some_Value then
      Remove (Q,E);
   end if;
end Advance;
```

Diagramas de módulos



Hijos privados

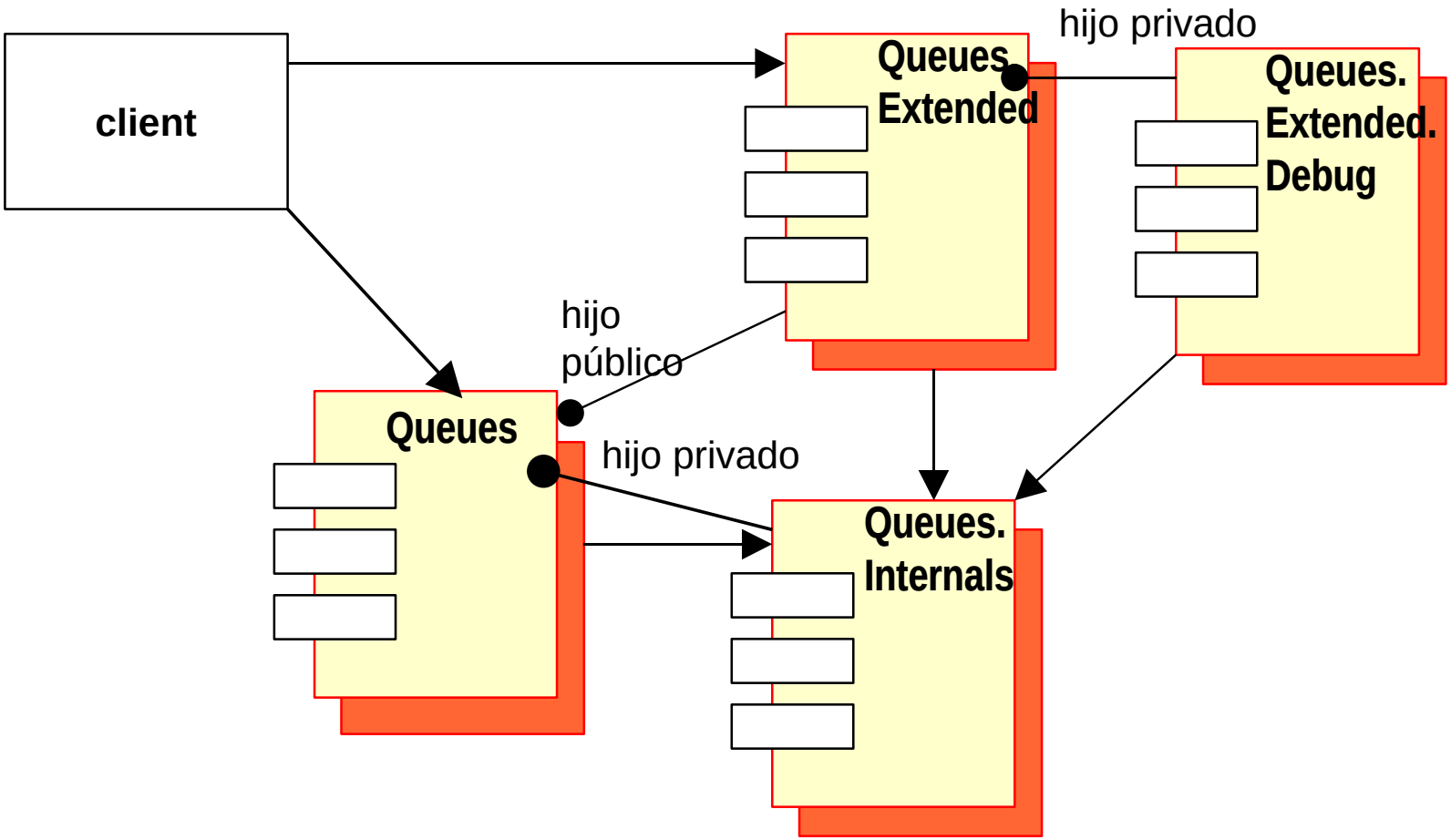
- Sirven para extender la implementación de un paquete sin necesidad de recompilar los clientes del padre
- La especificación de un hijo privado es una extensión de la parte privada del padre
  - En su especificación se ve la parte privada del padre
  - Sólo se puede ver en la parte privada de la jerarquía del padre
    - » cuerpo del padre
    - » cuerpo de los descendientes públicos del padre
    - » especificación y cuerpo de los descendientes privados del padre

Ejemplo (1)

```
private package Queues.Internals is
   procedure Copy (E : in Element;
                   Q : in out Queue_Node);
end Queues.Internals;
```

```
package body Queues.Extended is

   procedure Copy (E : in Element;
                   Q : in out Queue_Node) is
   begin
      ...
   end Copy;

end Queues.Extended;
```

Diagramas de módulos



Contenido

- Introducción
- Ocultamiento de información
- Tipos abstractos de datos
- Compilación separada
- **Objetos**
- Reutilización
- Programación en C++

Programación mediante objetos

- Un objeto es un elemento (constante o variable) de un tipo abstracto de datos
- Para realizar programas a partir de objetos es interesante disponer de
  - tipos extensibles con herencia
  - constructores de objetos con iniciación automática
  - destructores con terminación automática
  - selección de operaciones durante la ejecución *(dynamic dispatching)*

Tipos extensibles

- En Ada, un tipo derivado ***hereda*** las operaciones primitivas del progenitor
- Se pueden añadir o modificar las operaciones del tipo
- Si el tipo es un ***registro extensible*** *(tagged record)* se pueden añadir también componentes
- Una ***clase*** es la unión de todos los tipos que derivan de un antepasado común (la *raíz* de la clase)

Ejemplo

```
type Point is tagged
  record
    X, Y : Float;
  end record;

procedure Plot (P : Point);
```

```
type 3D_Point is new Point with
  record
    Z : Float;
  end record;

procedure Plot (P : 3D_Point);
```

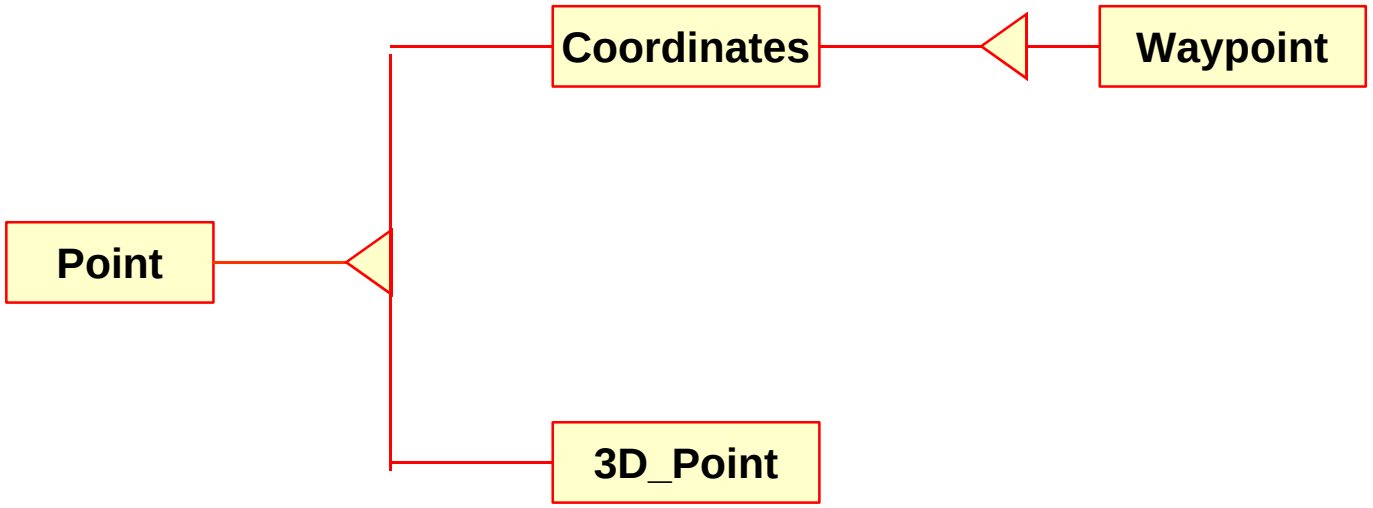
Clases y selección dinámica

- Cada tipo extensible *T* tiene asociado otro tipo *T'Class* que comprende los valores de *T* y sus descendientes
- Las operaciones sobre objetos de este tipo se seleccionan durante la ejecución

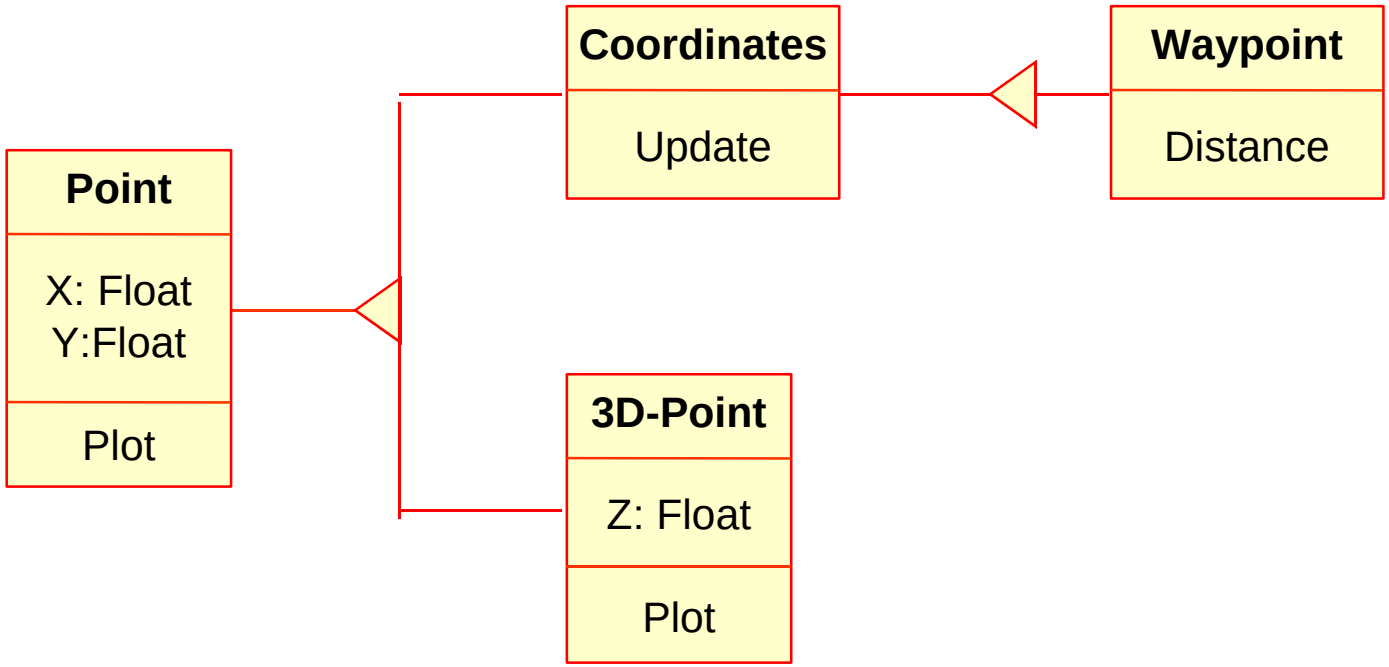
```
procedure General_Plot (P : Point'Class) is
begin
   -- otras operaciones
   Plot (P);
end General_Plot;
```

- En sistemas de tiempo real hace más difícil el análisis del tiempo de respuesta

Diagramas de clases (1)



Diagramas de clases (2)



Tipos controlados

- Son derivados de un tipo predefinido *Ada.Finalization.Controlled*
- Se pueden definir subprogramas que se ejecutan automáticamente al
  - crear un objeto — *Initialize*
  - destruir un objeto — *Finalize*
  - asignar un valor a un objeto — *Adjust*

Contenido

- Introducción
- Ocultamiento de información
- Tipos abstractos de datos
- Compilación separada
- Objetos
- **Reutilización**
- Programación en C

Reutilización de software

- La reutilización de componentes mejora la productividad y la calidad del software
- El tipado fuerte es un inconveniente
  - por ejemplo, habría que repetir el paquete *Queues* para distintos tipos de elementos
- Una solución es construir *plantillas* o *componentes genéricos* a partir de los cuales se pueden producir *ejemplares* concretos
- En Ada, los paquetes y subprogramas pueden ser genéricos

Ejemplo: colas genéricas

```
generic
  type Element is private;
  -- tiene al menos las operaciones ":=" y "="
package Generic_Queues is
   -- tipo abstracto
   type Queue is limited private;
   -- operaciones primtivas
   procedure Create (Q : in out Queue);
   function  Empty  (Q :        Queue) return Boolean;
   procedure Insert (Q : in out Queue; E :     Element);
   procedure Remove (Q : in out Queue; E : out Element);
private
  -- como antes
end Generic_Queues;
```

Ejemplares

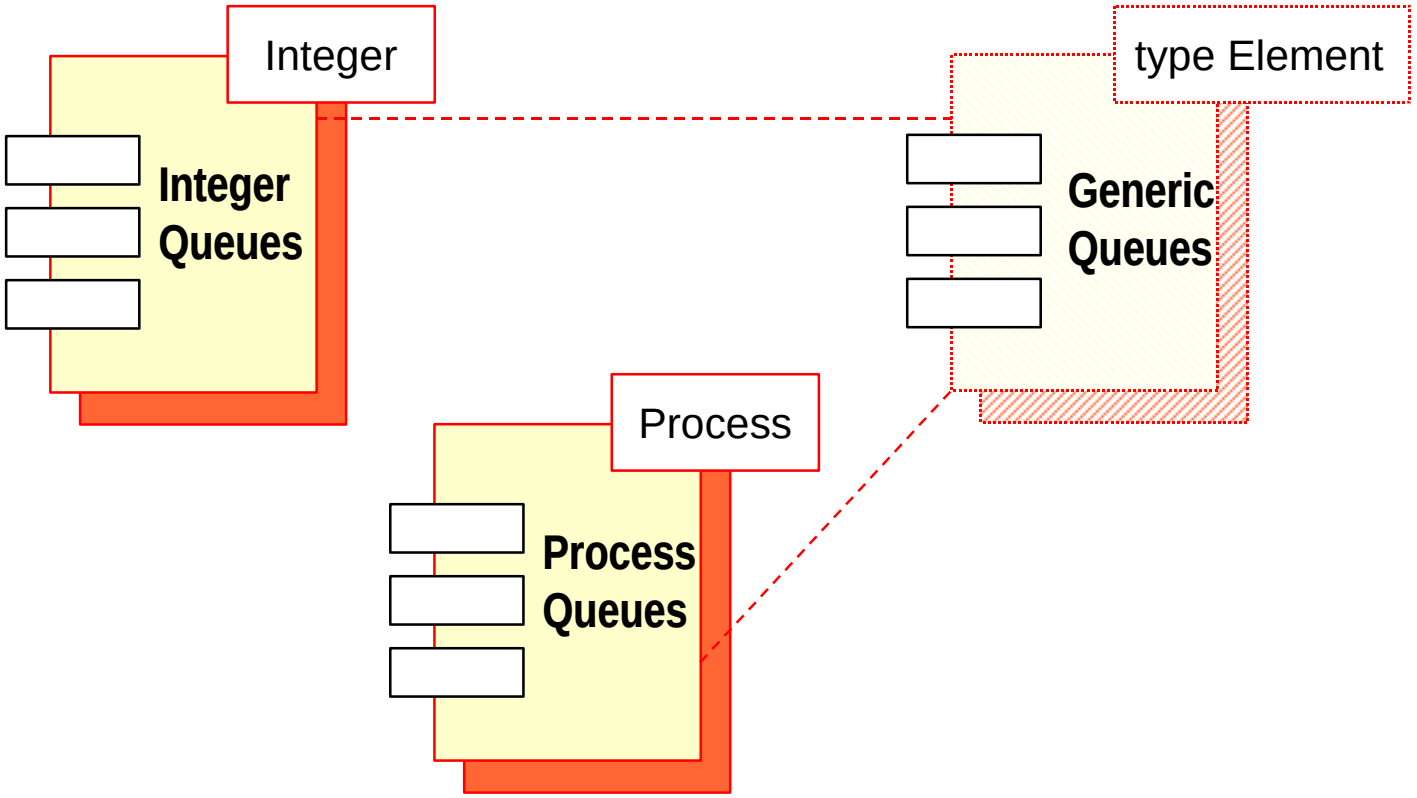
```
declare
  type Process_Id is ...;
  package Integer_Queues is new Generic_Queues (Integer);
  package Process_Queues is new Generic_Queues (Process_Id);
  use Integer_Queues;
  use Process_Queues;

  Value_Queue : Integer_Queues.Queue;
  Ready_Queue : Process_Queues.Queue;
  P  : Process_Id;
begin
  Create (Value_Queue); -- la operación se selecciona al
compilar
  Create (Ready_Queue);
  ...
  Insert (Ready_Queue, P);
  ...
end;
```

Parámetros genéricos

- Ada utiliza un *modelo de contrato* para los parámetros genéricos
  - Tipos
    - » discretos
    - » enteros
    - » reales
    - » formaciones
    - » derivados, extensibles, privados, privados limitados
  - Objetos
  - Subprogramas
  - Paquetes

```
type T is (<>);
type T is range  (<>);
type T is digits (<>);
type T is delta  (<>);
```

Diagramas de módulos



Contenido

- Introducción
- Ocultamiento de información
- Tipos abstractos de datos
- Compilación separada
- Objetos
- Reutilización
- **Programación en C**

Programación de sistemas grandes en C

- Se pueden construir módulos con ficheros .h y .c
- No hay relación formal entre ellos
- No hay tipado fuerte
- Los detalles de los tipos abstractos se ocultan mediante punteros
- No hay módulos genéricos

Ejemplo: cola (1)

```
/* queue.h -- especificación */

typedef int element;

struct queue_t;
typedef struct queue_t *queue_ptr_t;

queue_ptr_t create ();
int  empty  (queue_ptr_t Q);
void insert (queue_ptr_t Q, element E);
void remove (queue_ptr_t Q, element *E);
```

Ejemplo: cola (2)

```
/* queue.c -- cuerpo */
#include "queue.h”

struct queue_node_t {
  element contents;
  struct queue_node_t *next;
};
struct queue_t {
  struct queue_node_t *front;
  struct queue_node_t *back,
};

queue_ptr_t create () {
  queue_ptr_t Q;
  Q = (struct queue_t) malloc (sizeof(struct queue_t));
  Q->front = NULL;
  Q->back  = NULL;
  return Q;
};                                          /* etc */
```

Ejemplo: cola (3)

```
/* uso del módulo */
#include "queue.h”
main () {

  queue_ptr_t value_queue;
  element x, y;

  value_queue = create ();
  ...
  insert (value_queue, x);
  ...
  remove (value_queue, &y);
  ...
}
```

Resumen

- El mecanismo básico de descomposición y abstracción es el módulo
- Los módulos permiten trabajar con
  - ocultamiento de información
  - tipos abstractos de datos
  - compilación separada
- Otras técnicas asociadas a la programación basada en objetos son
  - extensión y herencia
  - iniciación, terminación y ajuste automáticos
  - selección dinámica de operaciones
- Los componentes genéricos o plantillas facilitan la reutilización del software