Tema 1
Programación Imperativa de Computadores

Pablo Sánchez

Dpto. Matemáticas, Estadística y Computación
Universidad de Cantabria
Santander (Cantabria, España)
p.sanchez@unican.es

Bibliografía Básica


Francisco Durán, Francisco Gutiérrez, and Ernesto Pimentel.
Programación Orientada a Objetos con Java.
Thomson Paraninfo, 2007.


Bertrand Meyer.
Construcción de Software Orientada a Objetos.
Prentice Hall, 2000.

Objetivos

Objetivos

1. Repasar los conceptos aprendidos en el curso anterior sobre Programación de Computadores.
2. Enfatizar aspectos y cuestiones importantes de Programación de Computadores en lo referente a Estructuras de Datos y Algoritmos.
3. Conocer y saber manipular el lenguaje de programación de pseudocódigo que se usará durante el curso.

Lenguajes de Pseudocódigo

Objetivos

1. Aprender conceptos y técnicas con independencia de lenguajes de programación específicos.
2. Formar y entrenar buenos hábitos de programación.
3. Resaltar conceptos importantes para el aprendizaje mientras que los detalles irrelevantes permanecen ocultos (abstracción).

`http://splashcon.org/index.php?option=com_content&view=article&id=173&Itemid=66`

Pseudosintaxis del Lenguaje de Pseudocódigo

Disponible en:

`http://personales.unican.es/sanchezbp/teaching/faqs/pseudoCodeSyntax.html`

NOTA: Debido al escaso tiempo con el que se ha desarrollado, ésta sintaxis podría contener errores. en aso de detectarse, se ruega se comuniquen a través del sistema de notificaciones de erratas y errores. En caso de ambigüedad, lo importante es preservar el espíritu de la norma.

Detalles Importantes

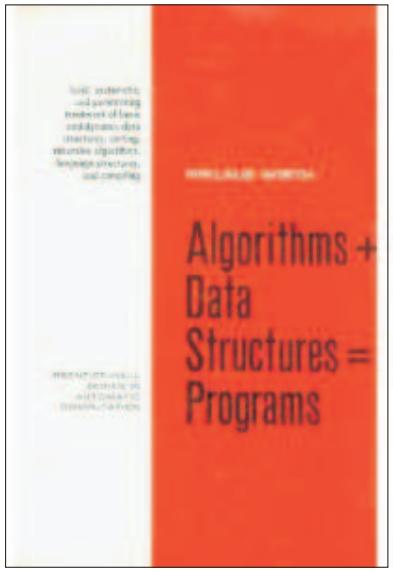
1. Hay que inicializar siempre las variables.
2. Definir tipos adecuados para los parámetros.
3. El lenguaje es fuerte y estáticamente tipado.
4. Vectores comienzan en cero.
5. No existen sentencias `break` o `continue`.
6. Sólo se puede poner un `DEVOLVER` al final de una función.

Módulos Software

Niklaus Wirth (1976)

Programación = Estructuras de Datos + Algoritmos

"This is the book every beginning programmer must read. Modern programmers know a lot about specific technical details, but they know nothing about effective algorithms. We're hiring new programmers now. We've tested a lot of folks with 5-8 years experience in Java. The result is nobody can implement basic operations with B-tree or combinatory algorithm. It's terrible. So read Niklaus Wirth if you want to be a serious programmer."



Módulos Software

Módulo Software

Un *módulo software* es un conjunto de datos y operaciones que ofrecen una funcionalidad bien determinada a través de una cierta *interfaz*.

Principio de Ocultación [3]

Cada módulo software debe proporcionar los detalles relevantes para su utilización a la vez que oculta aquellos detalles y decisiones que sean irrelevantes para su utilización.

Propiedades Ideales de una Descomposición Modular

Cohesión

Medida del grado de relación existentes entre los diversos pertenecientes a un módulo software.

Acoplamiento

Medida del grado de dependencia de un módulo software respectos a otros módulos software.

Un buen módulo software:

1. Está débilmente acoplado a otros módulos (cambios en otros módulos no le afectan).
2. Posee una alta cohesión (sirve a un único propósito).
3. Tiene una *granularidad adecuada*.

Beneficios de las Descomposiciones Modulares [3]

1. Desarrollo paralelo e independiente de cada módulo.
2. Sustitución de módulos (e.g. interfaz de escritorio por web).
3. Posibilidad de pruebas independientes.
4. Desarrollo de aplicaciones por ensamblado de módulos.

Punteros

Variable Puntero

Un variable de tipo puntero es una variable que contiene el valor de la dirección de memoria donde se aloja un dato de interés.

- Un puntero es siempre un puntero a *algo*. `PtrReal ES PUNTERO A REAL`.
- El *algo* define el tipo del puntero.
- Por tanto, `PUNTERO A ENTERO` $\neq$ `PUNTERO A REAL`.

Operaciones sobre Punteros

1. Acceder al valor apuntado (*dereferenciación*): `->VariablePuntero`.
2. Crear una variable dinámica: `miPtr := NUEVO PUNTERO A REAL;`
3. Borrar una variable dinámica: `ELIMINA PUNTERO(miPtr)`
4. Asignación: `miPtr := otroPtr`, `->miPtr := valorReal`.
5. Comparación: `puntero1 == puntero2`, `puntero1 != puntero2`
6. ¡OJO!: Se comprueba si ambos punteros apuntan al mismo sitio, no si los contenidos son iguales
7. `NULO` es un valor constante que representa puntero a ninguna parte.
8. `->NULO` carece de sentido y genera una excepción en tiempo de ejecución.

Utilidades de los Punteros

1. Imprescindibles para la gestión de *variables dinámicas*.
2. Permite *apuntar* a datos almacenados en memoria y evitar su replicación.
3. En lenguajes como C se usa para el paso de parámetros por referencia.

Peligros de los Punteros

1. Puedo acceder a posiciones de memoria inválidas.
2. *Memory leaks*: Memoria reservada a la cual no apunta ningún puntero, y que es por tanto irrecuperable. Solucionable mediante el uso de recolectores automáticos de basura (*garbage collector*).

Equivalencia Prog. Modular y Prog. Orientada a Objetos

Programación Modular	Prog. Orientada a Objetos
Módulo	Clase
Registro/Tupla	Atributos de una clase
Campo de Registro	Atributo de una clase
Puntero a Registro	Referencia a objeto
Función sobre registro	Método de una clase
funcion(obj : PtrRegistro, args)	obj.funcion(args)
NUEVO PUNTERO A Registro	NUEVO Clase
ELIMINAR ptrRegistro	DESTRUIR obj

Limitaciones de la Programación Modular

- Sea un tipo T_1 al que se puede aplicar un conjunto de operaciones op_1.
- Sea un tipo T_2 que especializa T_1 por adición de nuevas propiedades, operaciones o restricciones.
- Sea $op_{com} \subseteq op_1$ tal que op_{com}, con sus mismas especificaciones, es aplicable a T_2.
- Usando programación modular sin orientación a objetos, tengo que reescribir todas las op_1, incluyendo op_{com}, para que funcionen sobre datos de tipo T_2.
- Esto crea redundancias, que aumentan el tiempo de desarrollo y dificultan el mantenimiento y la adaptabilidad de los programas.
- La reutilización también se ve comprometida, porque las operaciones para un tipo de datos sólo son aplicables a ese tipo de datos.

Concepto de Clase y Herencia

Clase

Una *clase* es un módulo software que:

1. Encapsula datos y funciones;
2. Define un tipo;
3. Soporta relaciones de herencia.

Relación de herencia

Una relación de herencia es una *relación de especialización* entre dos clases A y B, tale que si A hereda de B:

1. A posee los mismos atributos, operaciones y restricciones que B;
2. A puede definir nuevos atributos, operaciones y restricciones, siempre que no entren en conflicto con los de B.
3. Las instancias de la clase A pertenecen al tipo A y al tipo B.

Polimorfismo de Datos y Principio de Sustitución

Polimorfismo de Datos

Propiedad por la cual una instancia de una clase A que herede de una clase B puede ser usada en cualquier lugar donde se exija un elemento de tipo B

Principio de Sustitución de Liskov [1]

Sea $q(x)$ un propiedad verificable sobre objetos x de un tipo T. Entonces $q(y)$ debería ser verdad para cualquier objeto y de un tipo S, donde S es un subtipo (hereda) de T.

No basta con "ser" hay que "comportarse como tal".

Vinculación Dinámica

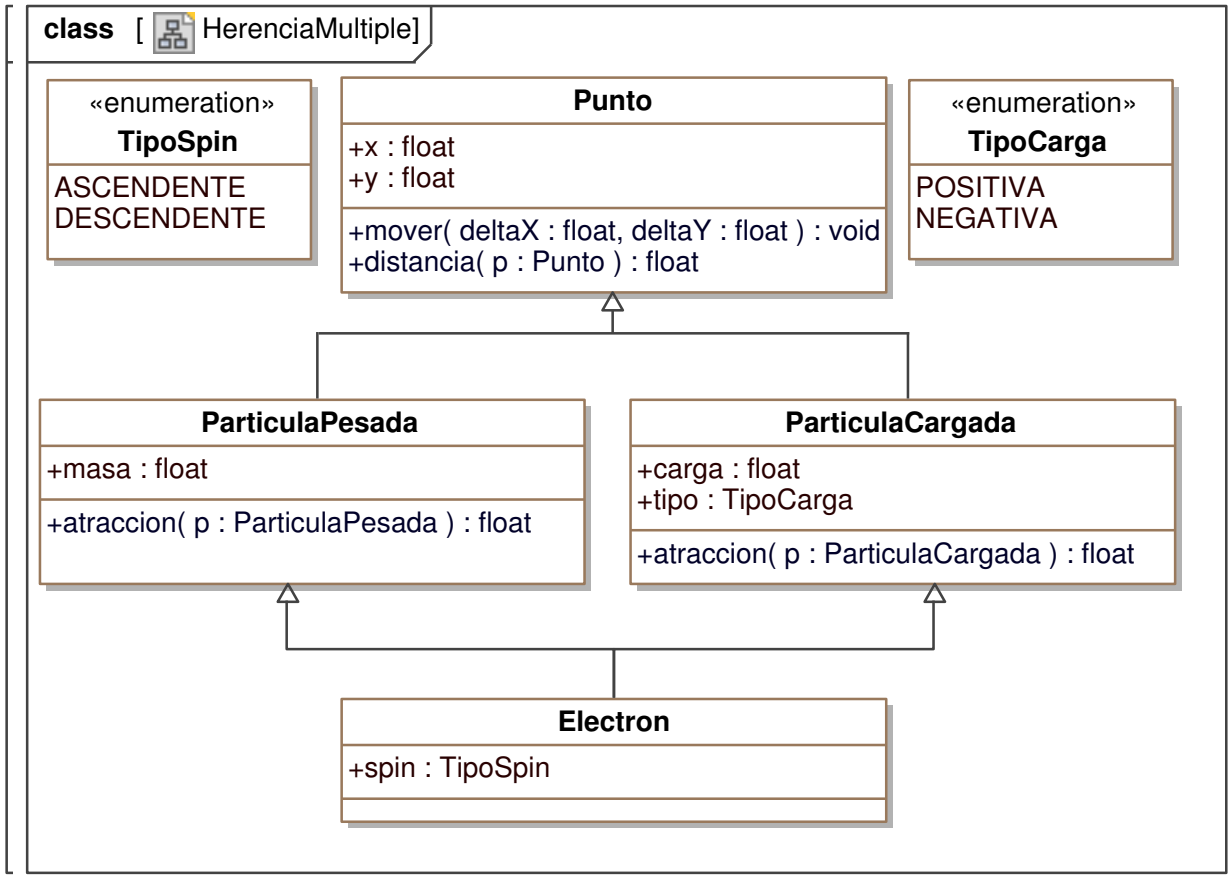
Vinculación Dinámica

Característica de un lenguaje de programación orientado a objetos por la cual el método que se invoca sobre un objeto se determina en tiempo de ejecución en función del *tipo real* de dicha instancia, y no en tiempo de compilación, en función del *tipo declarado* para dicho objeto.

Tipos de Herencia

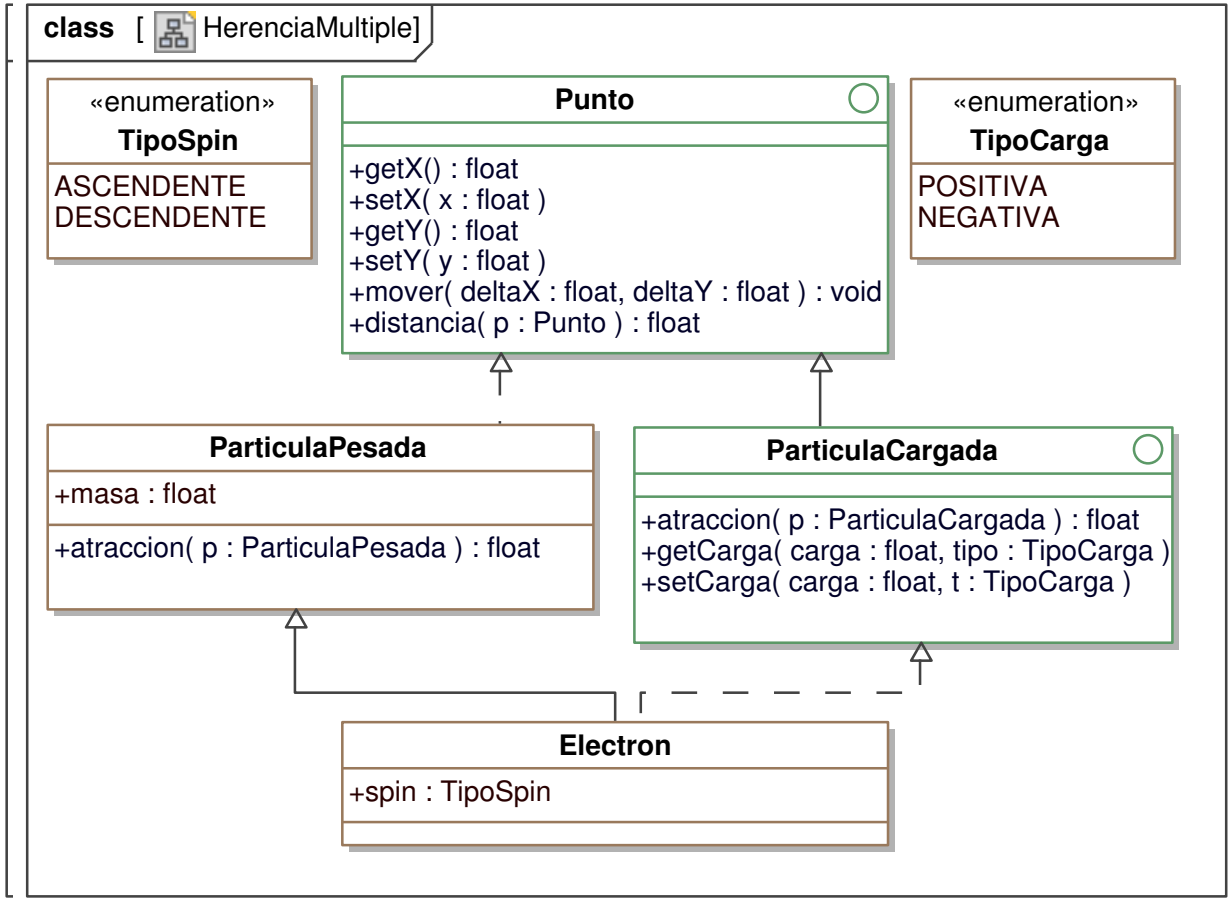
Selectiva	Puedo elegir que heredo.
No Selectiva	Heredo siempre todo lo del padre (y ascendientes).
Estricta	No puedo redefinir lo que heredo.
No Estricta	Puedo redefinir lo que heredo.

Herencia Múltiple

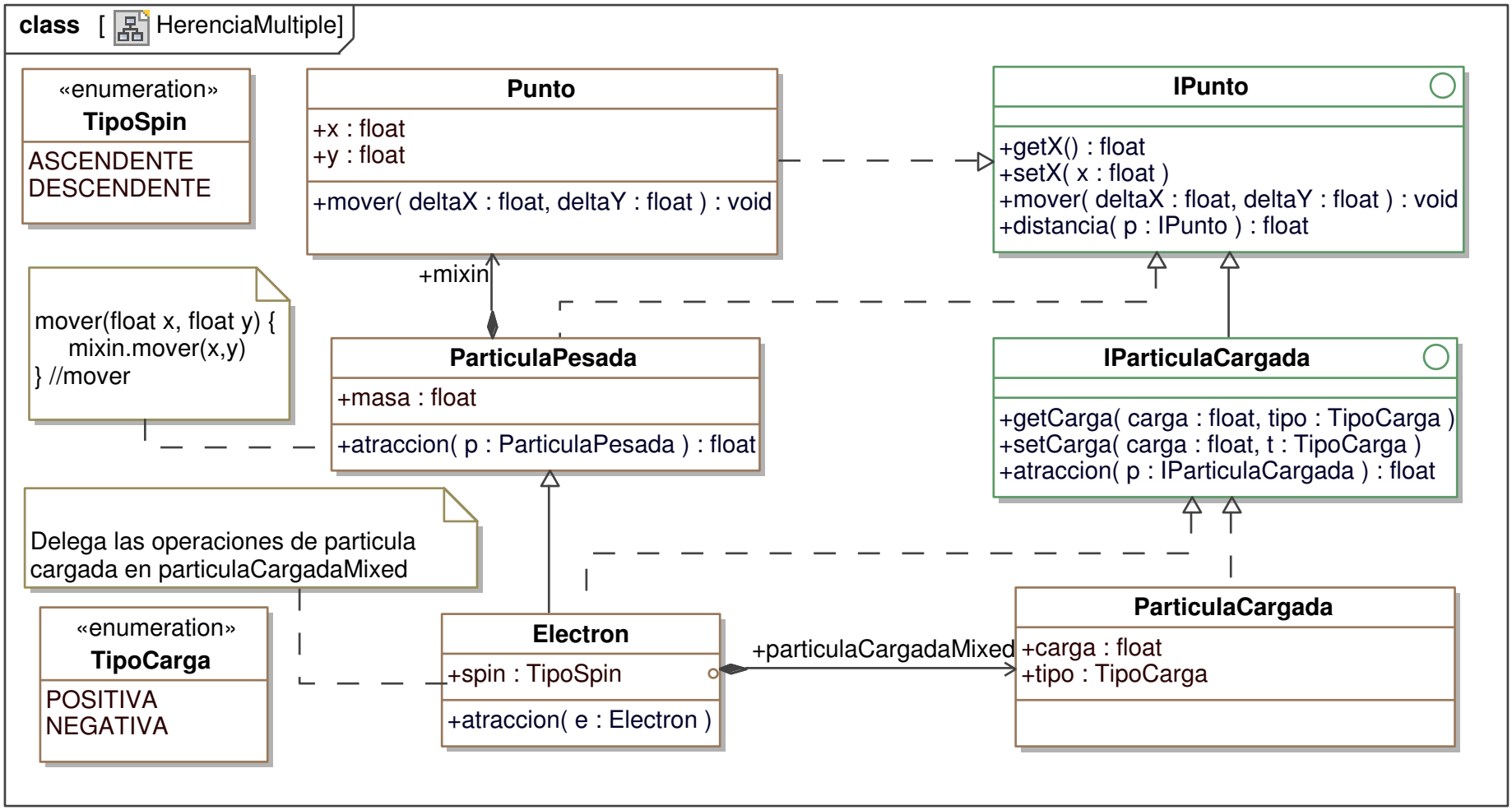


¿ `electron.atraccion(otroElectron)` ?

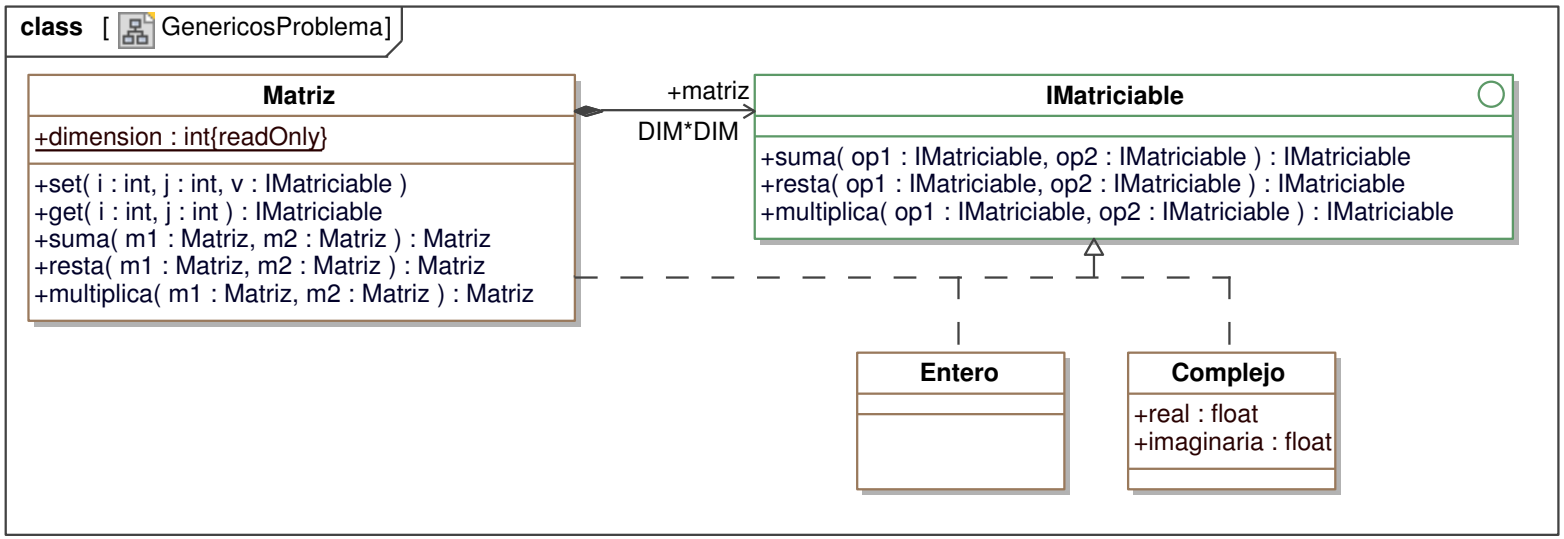
Herencia de Interfaces



Herencia Múltiple Mediante Patrón Mixin [2]



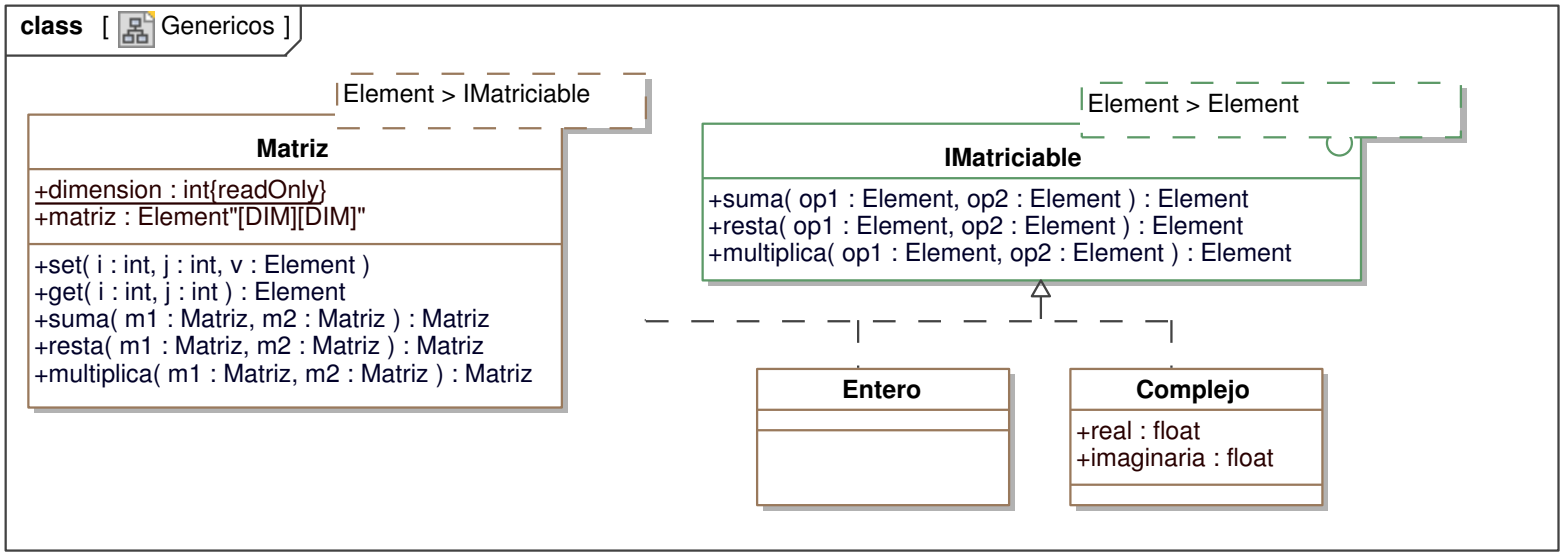
Limitaciones de la Herencia



Limitaciones de la Herencia

```
m : Matriz; c : Complejo; m2: Matriz;

m := NUEVO Matriz(); c := NUEVO Complejo()
m.set(0,0,c);

// Da error porque el valor devuelto
// no es de tipo Complejo
c := m.get(0,0);

// Obliga por tanto a hacer casting
c := (Complejo) m.get(0,0);

// Funciona (y no debería) porque las matrices
// son elementos matriciables
m.set(0,1,m2);
```

Programación Genérica



Ventajas del Uso de Genéricos

```
m : Matriz<Complejo>; c : Complejo; m2: Matriz<Entero>;

m := NUEVO Matriz<Complejo>(); c := NUEVO Complejo()
m.set(0,0,c);

// Funciona porque el valor devuelto es de tipo Complejo
c := m.get(0,0);

// No funciona porque el valor a asignar no es de tipo
// complejo
m.set(0,1,m2);
```

¿Qué tengo que saber de todo esto?

1. Saber usar los mecanismos básicos de la programación estructurada.
2. Entender y saber usar el concepto de módulo e interfaz.
3. Entender y saber usar el concepto de puntero.
4. Entender y saber usar los concepto de clase, herencia y polimorfismo.
5. Entender y saber usar la vinculación dinámica.
6. Entender y saber usar la herencia múltiple.
7. Entender y saber usar genéricos.

Referencias

Barbara Liskov and Stephen N. Zilles.
Programming with Abstract Data Types.
SIGPLAN Notices, 9(4):50–59, 1974.

David A. Moon.
Object-Oriented Programming with Flavors.
In *Proc. of the 1st Int. Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 1–8, November 1986.

David L. Parnas.
On the Criteria To Be Used in Decomposing Systems into Modules.
Communications of the ACM, 15(12):1053–1058, 1972.