Herencia Múltiple

Dpto. Lenguajes y Sistemas Informáticos
Universidad de Granada

Programación y Diseño Orientado a Objetos

Créditos

- Las siguientes imágenes e ilustraciones son libres y se han obtenido de:
  - Emojis, `https://pixabay.com/images/id-2074153/`
  -  `https://medias.maisonsdumonde.com/image/upload/q_auto,f_auto/w_2000/img/estanteria-de-metal-negro-y-abeto-con-reloj-1000-0-31-188836_1.jpg`
- El resto de imágenes e ilustraciones son de creación propia, al igual que los ejemplos de código

Objetivos

- Comprender en qué consiste la herencia múltiple
- Entender los problemas que puede ocasionar
- Conocer alternativas

Contenidos

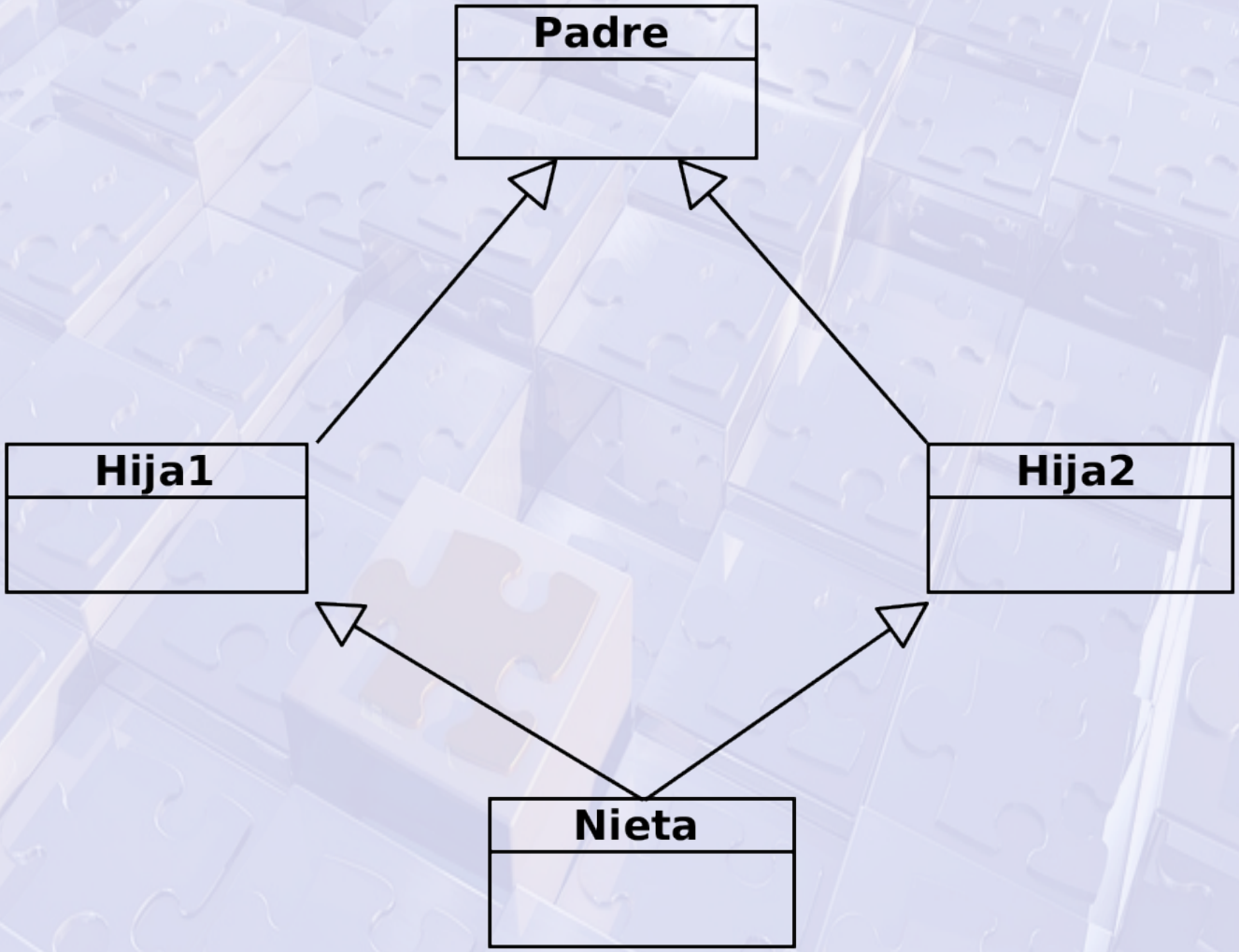
Herencia múltiple

- Se produce cuando una clase es descendiente de más de una superclase
- Permite representar problemas donde un objeto tiene propiedades según criterios distintos (clasificado según criterios distintos)
- Presenta problemas de implementación y pocos lenguajes la soportan (Ej. C++ y Python)
- Java y Ruby no tienen herencia múltiple

Problemas comunes

- Colisión de nombres de métodos y/o atributos
- Problema del diamante:
  - Provoca duplicidad en los elementos heredados
- No hay que olvidar que para que tenga sentido debe haber una relación *es-un* de la clase descendiente con todos sus ascendientes
  - La reutilización de código existente en varias clases no es por si solo un criterio para establecer relaciones de herencia múltiple

Problema del diamante



Ejemplo del problema del diamante

C++: Herencia múltiple. Problema del diamante

```
1  class A {
2          private:
3                  int a;
4          public:
5                  A ()  {std::cout << "Creado A SIN INICIALIZAR" << std::endl;}
6                  A (int i) {std::cout << "Creado A e inicializado" << std::endl; a = i;}
7                  int getA () {return a;}
8                  int setA (int i) {a = i;}
9  };
10
11 class B : public A {
12         public:
13                 B (int a) : A(a) {}
14                 void useAFromB () {std::cout << "Uso desde B " << getA() << std::endl;}
15 };
16
17 class C : public A {
18         public:
19                 C (int a) : A(a) {}
20                 void useAFromC () {std::cout << "Uso desde C " << getA() << std::endl;}
21 };
```

Ejemplo del problema del diamante

C++: Herencia múltiple. Problema del diamante

```
1 class D : public B, public C {
2         public:
3                 D (int a) : B(a), C(a) {}
4                 void useAFromD() {useAFromB(); useAFromC();}
5             //  void modifyA (int a) {setA(a);} // Error: Hay ambigüedad
6                 void modifyA (int a) {C::setA(a);} // evitamos que la referencia sea ambigua
7
8 };
9
10 int main(int argc,char **argv) {
11         std::cout << "Problema del diamante v0" << std::endl;
12         D *d = new D(33);
13         // Creando A e inicializando
14         // Creando A e inicializando
15         d->useAFromD ();
16         // Uso desde B 33
17         // Uso desde C 33
18
19         d->modifyA (66);
20
21         d->useAFromD ();
22         // Uso desde B 33
23         // Uso desde C 66  // Hay dos versiones del atributo A::a
24 }
```

Ejemplo del problema del diamante

C++: Herencia múltiple. Solución C++ a la duplicidad de atributos

```
1  class A {
2          private:
3                  int a;
4          public:
5                  A ()  {std::cout << "Creado A SIN INICIALIZAR" << std::endl;}
6                  A (int i) {std::cout << "Creado A e inicializado" << std::endl; a = i;}
7                  int getA () {return a;}
8                  int setA (int i) {a = i;}
9  };
10
11 class B : virtual public A {
12         public:
13                 B (int a) : A(a) {}
14                 void useAFromB () {std::cout << "Uso desde B " << getA() << std::endl;}
15 };
16
17 class C : virtual public A {
18         public:
19                 C (int a) : A(a) {}
20                 void useAFromC () {std::cout << "Uso desde C " << getA() << std::endl;}
21 };
```

Ejemplo del problema del diamante

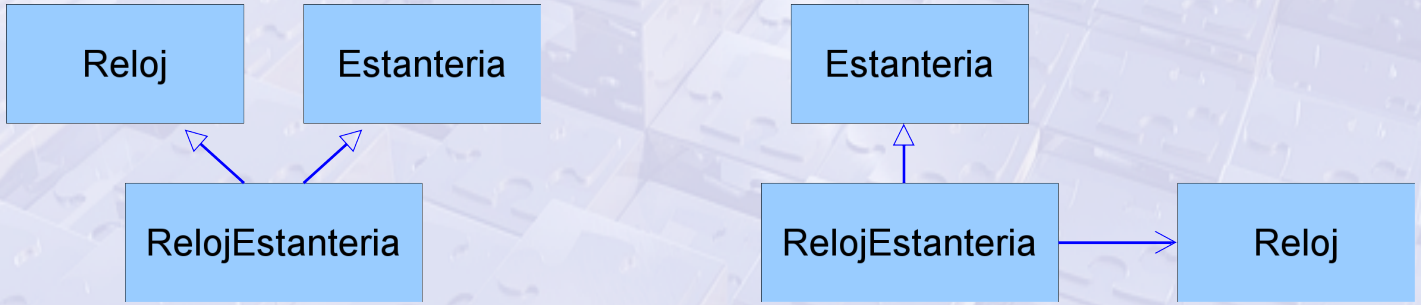
C++: Herencia múltiple. Solución C++ a la duplicidad de atributos

```
1  class D : public B, public C {
2          public:
3                  D (int a) : A(a), B(a), C(a) {}   // Debemos llamar al constructor de A
4                                                    // Si no, se llama al constructor por defecto
5                  void useAFromD() {useAFromB(); useAFromC();}
6              //  void modifyA (int a) {setA(a);} // Error: Hay ambigüedad
7                  void modifyA (int a) {C::setA(a);} // evitamos que la referencia sea ambigua
8
9  };
10
11 int main(int argc,char **argv) {
12         std::cout << "Problema del diamante v0" << std::endl;
13         D *d = new D(33);
14         // Creando A e inicializando
15         d->useAFromD ();
16         // Uso desde B 33
17         // Uso desde C 33
18
19         d->modifyA (66);
20
21         d->useAFromD ();
22         // Uso desde B 66
23         // Uso desde C 66  // Solo hay una versión del atributo A::a
24
25         B *b = new B(77);
26         b->useAFromB();
27         // Uso desde B 77  // Las instancias de B también tienen el atributo A::a
28 }
```

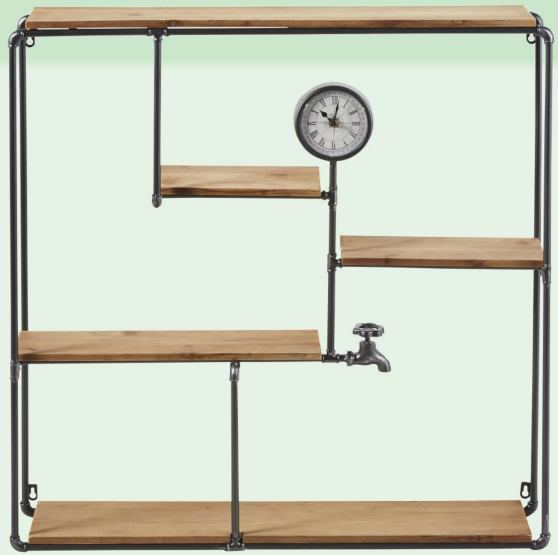
Alternativas

- Composición
  - Sustituir una o varias relaciones de herencia por composición
- Interfaces Java
  - Se pueden realizar varias interfaces Java y heredar de una superclase
- Mixins de Ruby
  - Permiten incluir código proveniente de varios módulos como parte de una clase

Ejemplo de composición

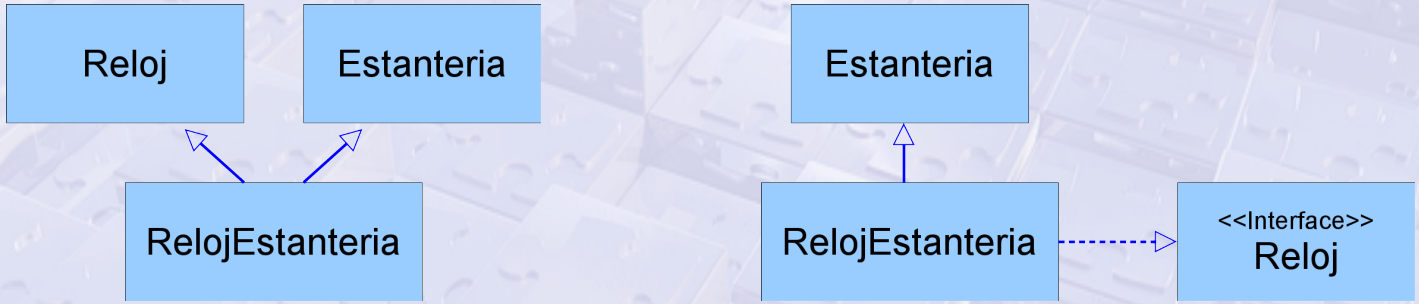


Java: Ejemplo de composicion



```
1  class Estanteria { . . . }
2
3  class Reloj { . . . }
4
5  class RelojEstanteria extends Estanteria {
6    private Reloj reloj;
7
8    RelojEstanteria () {
9      super();
10     reloj = new Reloj();
11   }
12
13   // Los métodos de Estanteria se heredan
14
15   void setHora (Hora h) {      // Los métodos de Reloj se definen
16     reloj.setHora (h);         // se implementan reenviando el mensaje al atributo
17   }
18 }
```

Ejemplo con interfaces Java



Java: Ejemplo con interfaces Java

```
1 class Estanteria { . . . }
2
3 interface Reloj { public void setHora (Hora h); public Hora getHora (); }
4
5 class RelojEstanteria extends Estanteria implements Reloj {
6
7   RelojEstanteria () {
8     super();
9   }
10
11  // Los métodos de Estanteria se heredan
12
13  // Se implementan los métodos de la interfaz
14
15  public void setHora (Hora h) { . . . }
16  public Hora getHora () { . . . }
17 }
```

Ejemplo de mixin de Ruby

Ruby: Ejemplo de mixin de Ruby

```ruby
module Volador
  def volar
    puts "Volando"
  end
end

module Nadador
  def nadar
    puts "Nadando"
  end
end

class Ejemplo
  def metodo
    puts "Método propio"
  end

  include Volador # Añadimos todo el módulo
  include Nadador # Añadimos todo el módulo
end

e=Ejemplo.new
e.metodo
e.volar
e.nadar
```

Herencia Múltiple

Dpto. Lenguajes y Sistemas Informáticos
Universidad de Granada

Programación y Diseño Orientado a Objetos