Lenguajes de programación orientados a objetos y basados en prototipos

Grupo IMO
Área de Lenguajes y Sistemas Informáticos
Departamento de Informática

Conferencia en la *Escuela Superior de Informática*,
Universidad de Valladolid

J. Baltasar García Perez-Schofield
http://webs.uvigo.es/jbgarcia/

El modelo de orientación a objetos basado en prototipos

<u>Características básicas</u>

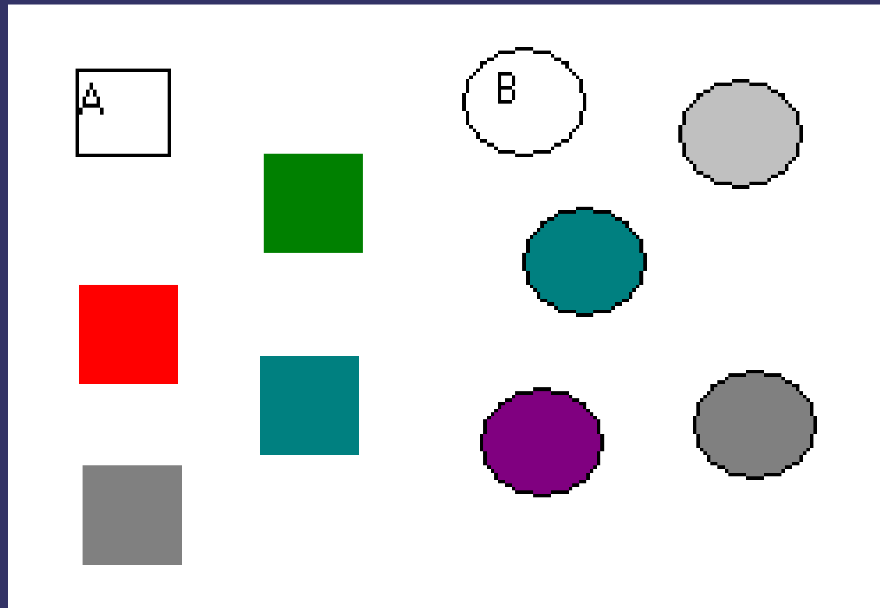
Orientación a objetos basada en prototipos

- Existen dos corrientes principales:
  - Lenguajes orientados a objetos basados en clases: C++, Object Pascal, Java, Eiffel ... son los más utilizados por la industria.
  - Lenguajes orientados a objetos basados en prototipos: Self, Kevo, Poet/Mica, Cecil ... son todos ellos experimentales, es decir, no se utilizan en la industria.

Terminología

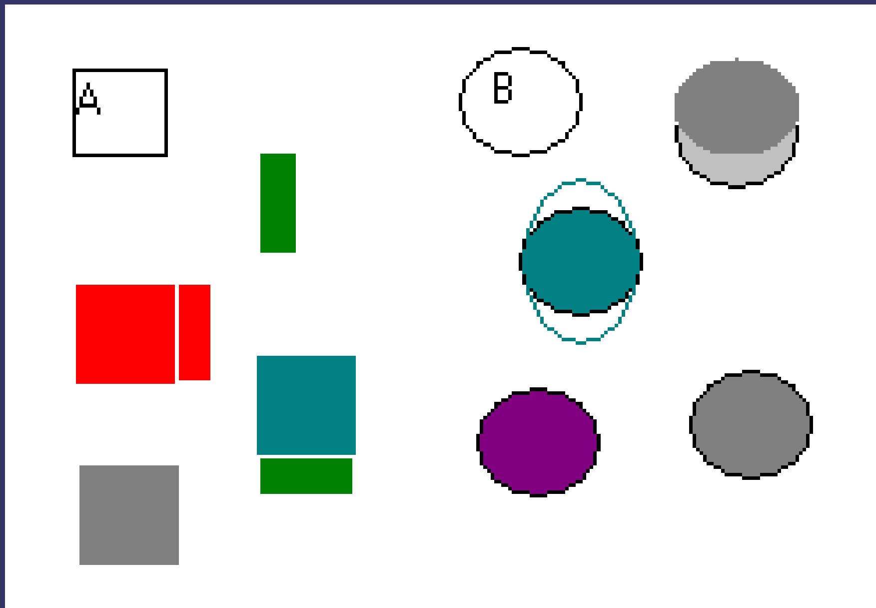
- ***Estado***: los *atributos* (terminología SmallTalk) o *datos miembro (terminología C++)* de un objeto. En el caso de un coche, su color, su longitud, cilindrada, ...
- ***Comportamiento***: los *métodos* (SmallTalk) o *funciones miembro* (C++) de un objeto. En el caso de un coche, arrancar, acelerar, frenar, apagar.
- ***Mensaje***: ejecución de un método de un objeto. Si un objeto tiene un método f(), mandarle a O el mensaje f es lo mismo que ejecutar O.f()

Orientación a objetos basada en clases



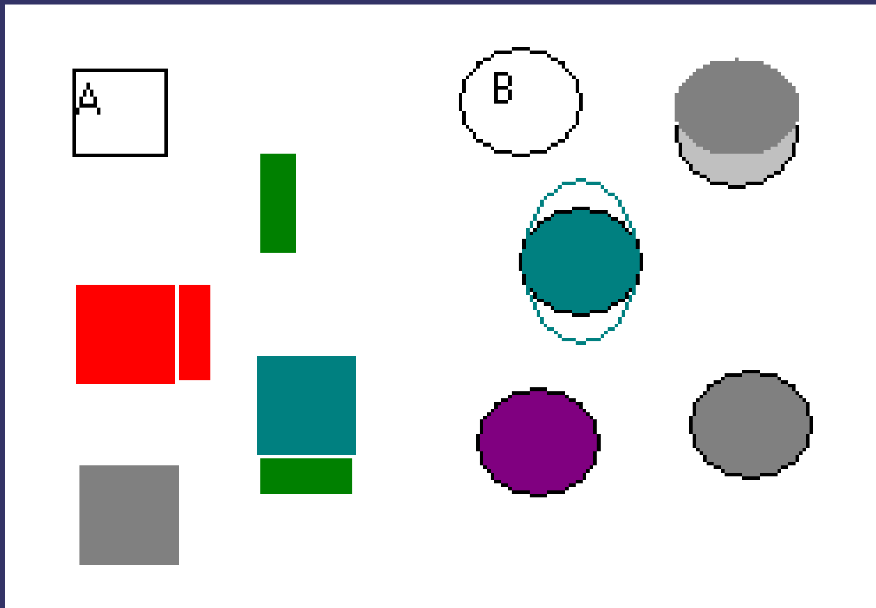
- Una clase es un "tipo" de objetos, es decir, un molde del que se obtienen nuevos objetos, que comparten similar comportamiento, cambiando el estado de los mismos.

Orientación a objetos basada en prototipos



- No existen las **clases**. De hecho, todos los objetos son iguales en cuanto a categoría.
- Los nuevos objetos se copian de otros ya existetntes. Algunos de ellos son **prototipos**.

Orientación a objetos basada en prototipos

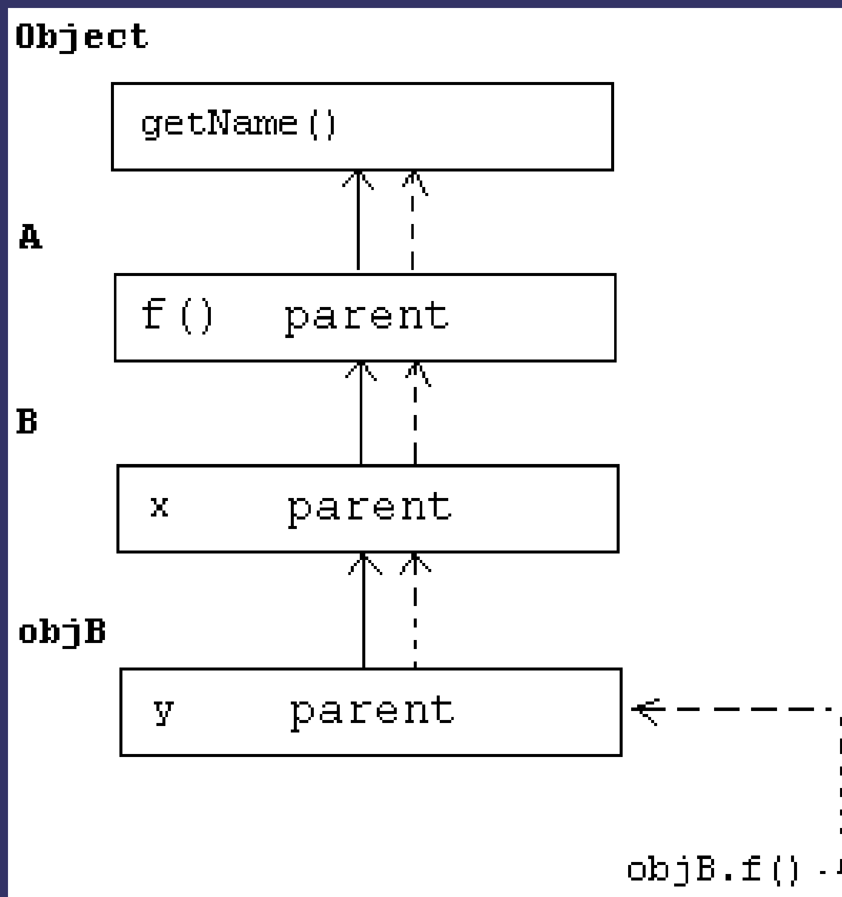


- Normalmente, en este tipo de lenguajes los objetos pueden modificarse, añadiendo o borrando métodos y atributos.
- Cada objeto es independiente, no necesitando información extra de ningún tipo.

Herencia

- La herencia en lenguajes basados en prototipos suele ser por delegación.
  - El objeto tiene uno o más aributos *parent*, de forma que cuando no puede responder a un mensaje, le reenvía éste a su padre.
- En el caso de los lenguajes basados en clases, ésta suele presentarse como concatenación
  - El objeto está compuesto por las partes que define cada una de las clases de las que hereda.

Herencia mediante delegación



- Existe una cadena de objetos apuntando a sus padres, hasta llegar a un objeto padre de todos.

Comparativa: el modelo basado en clases respecto al modelo basado en prototipos

Creación de objetos

Creación de Objetos

- Al crearse los nuevos objetos mediante copia, no es necesario que existan los constructores de los lenguajes orientados a objetos basados en clases.
- Los objetos no sólo definen los tipos de datos de los atributos, como en las clases, sino que además ya tienen un valor asociado.

Creación de Objetos

- Por ejemplo:

```
object Persona
  attribute + nombre = “Juan”;
  attribute + apellidos = “Nadie”;
  attribute + telefono = “906414141”;
  attribute + edad = 18;
  attribute + direccion;

  method + toString()
  {
    return nombre.concat( apellidos );
  }
endObject
```

Creación de objetos

- El objeto *Persona* es un prototipo que servirá para crear nuevos objetos, aunque no existe ninguna diferencia entre un prototipo y cualquier otro objeto.
- Por ejemplo:

```
Persona.copy( “paulaMarquez” );
```

- Si se le envía el mensaje *copy* al objeto *Persona*, entonces se creará un nuevo objeto copia exacta de *Persona*, con el nombre “PaulaMarquez”, cuyos atributos deberán ser modificados convenientemente.

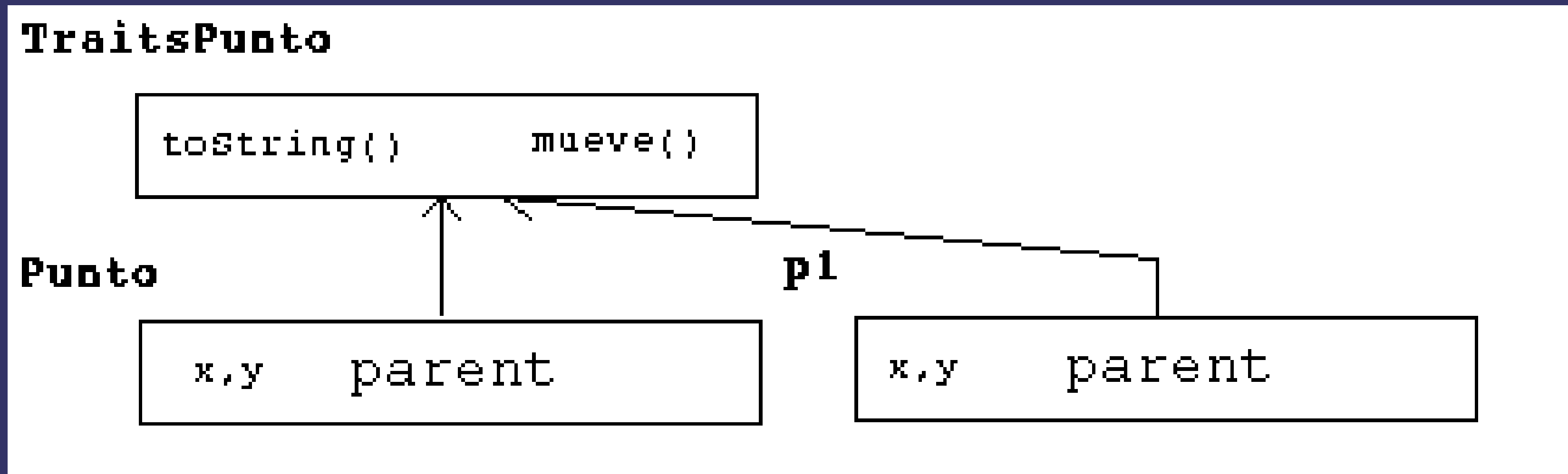
Creación de objetos

- Crear un objeto de *Persona*:

```
reference paula;

paula = Persona.copy( “paulaMarquez” );
paula.ponNombre( “Paula” );
paula.ponApellidos( “Márquez Márquez” );
paula.ponTelefono( “988353535” );
...
```

La copia puede ser costosa



- El prototipo (separado en rasgos y estado) es el padre de los objetos que son *instancias* de él.
- Se emplea la herencia de manera conveniente.

Ejemplo de traducción en C++

- El siguiente programa:

```
class Coche {
public:
  int numRuedas;
  int color;
  int combustible;
  static void encontrarGasolinera();
  void arranca();
};
//...
int main(void) {
    Coche micoche;

    miCoche.color = 1; /* BLANCO */
    micoche.arranca();
}
```

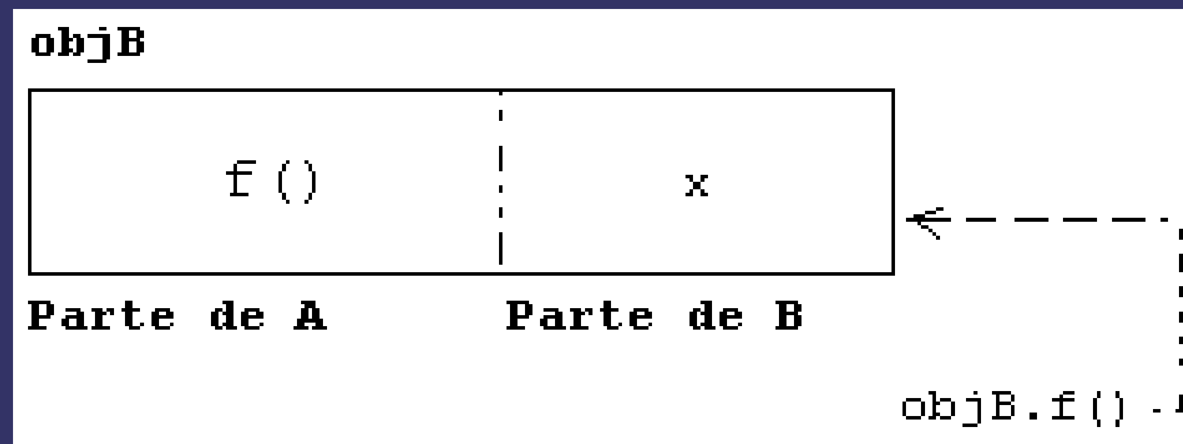
Ejemplo de traducción en C++

- Sería traducido como:

```
struct Coche {
    int numRuedas;
    int color;
}
void Coche_encontrarGasolinera() {
  // ...
}
void Coche_arranca(struct Coche &this) {
 // ...
}
int main(void)
{
  struct Coche miCoche;
  miCoche.color = 1; /* BLANCO */
  Coche_arranca(&miCoche);
}
```

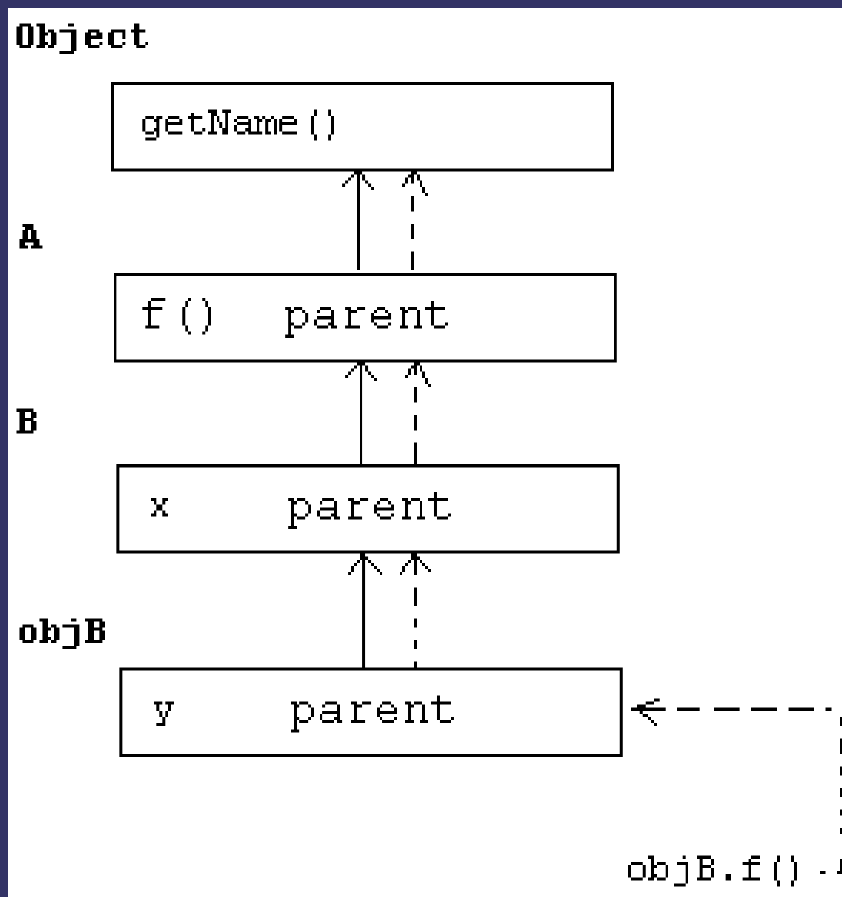
Comparativa: Herencia.

Herencia por concatenación



- Todos los atributos y métodos heredados están disponibles en el mismo objeto (si bien se necesita la clase para poder interpretarlos).

Herencia mediante delegación



- Existe una cadena de objetos apuntando a sus padres, hasta llegar a un objeto padre de todos.

Respuesta a mensajes

- Cuando un objeto no puede responder un mensaje, porque no posee el miembro (a-tributo o método), que se le pide, reenvía el mensaje al objeto que marca su atributo *parent*.
- Las relaciones de herencia, en lugar de ser un caso aparte, pasan a ser un caso particu-lar de las relaciones de composición.

Respuesta a mensajes

- Ejemplo. Dados los objetos:

```
object A
  method + foo()
  {
    System.console.write( “foo” );
    return;
  }
endObject

object B: A
endObject
```

Respuesta a mensajes

- El mensaje:

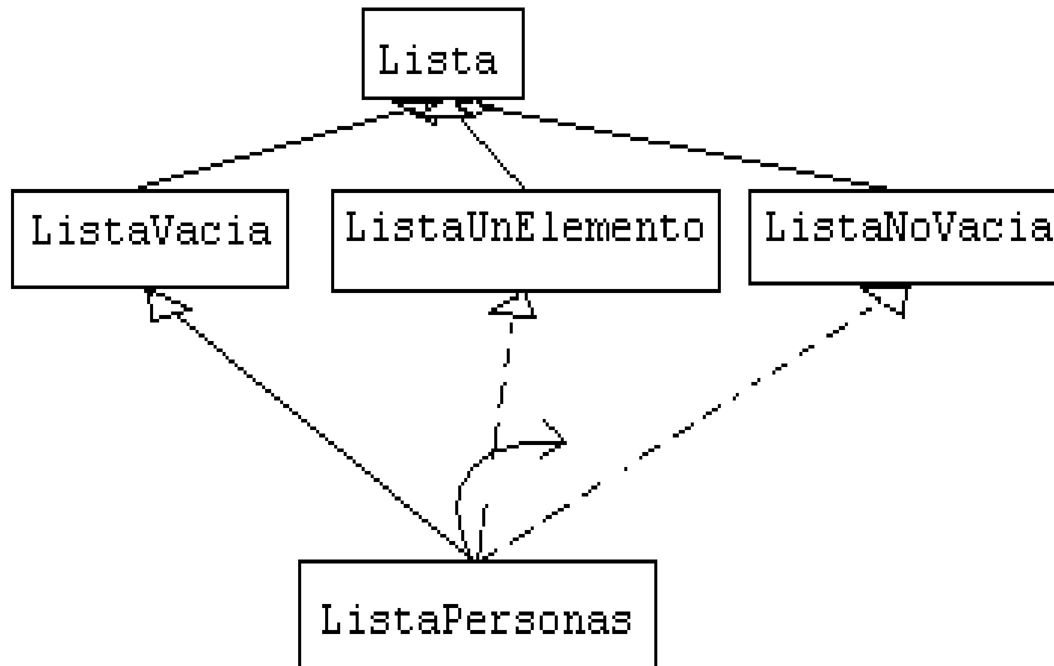
```
B.foo();     // MSG B foo
```

- No encuentra el método *foo*() en el objeto *B*, así que se sigue el atributo parent, que apunta a *A*, que sí tiene ese método, y es ejecutado.
- Si no se encontrara, entonces se produciría un error, que normalmente se traduce en una excepción. En este caso, la excepción producida sería “Método no encontrado”.

Herencia dinámica

- En el caso de estar implementada por delegación, se abre una nueva posibilidad: el hecho de poder cambiar el atributo (ya que, normalmente, es un atributo más) que señala al *padre* del objeto, hace que un objeto pueda ser "hijo" de varios objetos, dependiendo del momento de la ejecución.
- El aprovechamiento de esta característica requiere cambiar ligeramente el tipo de programación.

Herencia dinámica



- Es posible cambiar, en tiempo de ejecución, al "padre" de un objeto.
- Es totalmente contrario a la corriente actual, que trata de detectar todos los errores posibles en tiempo de compilación.

Herencia dinámica

- En el método insertar de ListaUnElemento:

```
object ListaUnElemento: Lista
  method + insertar(n, obj)
  {
    super( 1, obj );
    parent = ListaNoVacia;
    return;
  }
  method + getElementoNumero(n)
  {
    return parent.getPrimerElemento();
  }
endObject
```

Herencia dinámica

- En el método borrar de ListaNoVacia:

```
object ListaNoVacia: Lista
  method + borrar(n)
  {
    super( n );

    if ( this.getNumeroElementos()
           == 1 ){
      parent = ListaUnElemento;
    }

    return;
  }
endObject
```

Herencia dinámica

- En el lenguaje Prowl:

```
object ListaPersonas:
   ListaVacia( this.size() == 0 ),
   ListaUnElemento( this.size() == 1 ),
   ListaNoVacia( this.size() > 1 )
endObject
```

Herencia dinámica

- Su principal ventaja reside en que los métodos pueden escribirse según el tipo del objeto. En ListaVacia, no es necesario que getNumeroElementos() consulte el tamaño de la lista, sólo debe devolver cero. Puede ayudar a solucionar errores y hacer el código más simple.
- Su principal desventaja es que precise coordinar varios tipos para realizar una serie de tareas. Ésto puede conllevar errores y puede hacer las modificaciones de código más simples o más complicadas..

Conclusiones de la comparativa: El modelo de prototipos incluye al de clases

- Los objetos que sirven de prototipos son equivalentes a las clases de aquellos lenguajes orientados a objetos basados en clases.
- La diferencia es que este modelo es mucho más flexible que el de clases.
- Incluso una "*clase*" en este modelo puede modificarse, al no ser más que un objeto.
- La delegación es un mecanismo altamente flexible, separando a los objetos de sus prototipos, como en los lenguajes basados en clases, pero no al *comportamiento* del *estado*.

Conclusiones de la comparativa: ¿Cuál es el papel de la clase?

- La clase no desaparece, sino que se transforma:
  - Sigue siendo necesaria.
  - ... pero es un objeto más en el sistema.
  - ... manipulable y flexible, en lugar de rígida e inmodificable.

Patrones de diseño y prototipos

Prototype

- Copiar una instancia prototípica.
  - Un ejemplo podría ser el de las fichas del juego de las Damas.
- Es una característica inherente a los lenguajes basados en prototipos.
  - No se pueden crear objetos de ninguna otra forma.
- A veces, la copia no es suficiente.
  - Copiar todo un objeto (comportamiento incluído), puede ser demasiado costoso. Ésto se puede paliar.

Bridge

- Una jerarquía de clases presenta abstracciones, mientras otra presenta implementaciones.
  - El objetivo es permitir varias implementaciones o distinguir entre funcionalidades.
- En un lenguaje basado en prototipos, es posible emplear la herencia dinámica.
  - La abstracción cambia de implementación cuando sea necesario.
- Se puede, sin embargo, emplear el mismo diseño que en el modelo basado en clases.

Decorator

- Añadir más funcionalidades.
  - El objetivo es complementar (“decorar”) objetos con nueva funcionalidad.
- En un lenguaje basado en prototipos, se emplea la misma técnica que en clases.
  - Aunque sólo se aporta la decoración en sí.
- Se puede emplear herencia.

Proxy

- "Traer" el objeto/clase real sólo cuando es necesario hacerlo.
  - Algunas clases puede ser necesario traerlas desde el disco o por una conexión de red.
- En un lenguaje basado en prototipos, se emplea la misma técnica que en el patrón *decorator*.
  - Se duplican los métodos en el decorador (objeto hijo), que antes de remandar la petición al objeto padre, recuperan el objeto.

Observer

- La implementación es la misma que en el modelo basado en clases.
  - No existe ninguna diferencia teórica entre ambos modelos.
- En el lenguaje Self, no es posible distinguir entre un método y una variable, sólo hay *slots*.
  - Facilita el envolver el atributo para señalar el momento en el que ha cambiado a sus observadores.

Lenguajes que siguen el modelo de orientación a objetos basado en prototipos

Lenguajes que siguen el modelo de orientación a objetos basado en prototipos

- Self
  - Fue creado en los laboratorios de Sun, y muchas partes de su sistema son las precursoras de Java.
  - Fue el primero en implementar el modelo de prototipos, que también fue inventado por ellos. Self trata de ser todo lo dinámico que sea posible, haciendo el mínimo chequeo en tiempo de compilación posible.
  - http://research.sun.com/research/self/
- Io
  - Fue creado por uno de los desarrolladores de Self, siguiendo sus directrices principales.
  - http://www.iolanguage.com/

Lenguajes que siguen el modelo de orientación a objetos basado en prototipos

- Kevo
  - Kevo fue desarrollado para una tesis doctoral, como una demostración de que un lenguaje orientado a objetos y basado en prototipos podía incorporar técnicas modernas como la comprobación de errores que realiza *C*++, por ejemplo, en tiempo de compilación.
  - Implementa herencia por concatenación.
  - Es menos flexible en tiempo de ejecución que *Self* o *Io*.
  - ftp://cs.uta.fi/pub/kevo

Lenguajes que siguen el modelo de orientación a objetos basado en prototipos

- Cecil
  - Cecil fue desarrollado en la universida de Washington. Incorpora el concepto de objetos predicados, es decir, que mediante una condición inherente al objeto, y la herencia dinámica, es posible llevar el concepto de programación por contrato al nivel del objeto.
  - Es un lenguaje basado en prototipos.
  - http://www.cs.washington.edu/research/projects/cecil/www/cecil.html

Lenguajes que siguen el modelo de orientación a objetos basado en prototipos

- Python
  - Soporta el modelo basado en prototipos de man-era camuflada.
  - Es necesario crear una clase, que sin embargo se comporta como un objeto más.
  - La clase es modificable, se le pueden añadir nuevos métodos.

Lenguajes que siguen el modelo de orientación a objetos basado en prototipos

- Sistema Zero. Aún en desarrollo, en la Universidad de Vigo.
  - El lenguaje principal es Prowl.
  - Simple, basado en prototipos.
  - Su característica más novedosa es que incorpora persistencia implícita.
  - http://trevinca.ei.uvigo.es/~jgarcia/TO/zero/

Lenguajes que siguen el modelo de orientación a objetos basado en prototipos

- Otros lenguajes
  - http://www.programming-x.com/programming/prototype-based.html

Conclusiones:
Escenarios donde estos lengua-jes son más apropiados

Escenarios donde estos lenguajes son más apropiados

- Prototipado rápido.
  - Puede hacerse un *prototipo* de aplicación con gran rapidez utilizando estos lenguajes.
- Problemas que precisen de gran flexibilidad, variabilidad de características.
  - Guardar toda la información para todas las instancias de una clase puede redundar en gran desperdicio de espacio y tiempo.
- Los lenguajes en la industria empiezan a incorporar características de estos lenguajes.

C# 3.0

- Inferencia de tipos:

```
string toString(IList list)
{
    string toret = "";

    foreach( var elem in list ) {
        toret += elem.toString() + '\n';
    }

    return toret;
}
```

No todo son ventajas ...

- Eficiencia.
  - El *method dispatching* de C++ es incomparablemente rápido, aunque como siempre se pueden establecer mecanismos paliativos.
- Puede ser más difícil encontrar errores.
  - Los errores no son detectados en tiempo de compilación.
- Experiencia.
  - Si bien es cierto que es más sencillo de entender para un principiante, la detección de errores en tiempo de compilación es una gran ventaja.

Bibliografía

Bibliografía

- Sobre Self y el modelo de prototipos en general:

  - Cuesta, P., García Perez-Schofield, B., Cota, M. (1999). “Desarrollo de sistemas orientados a objetos basados en prototipos”. Actas del Congreso CICC' 99. Q. Roo, México.
  - Smith & Ungar (1995). “Programming as an experience, the inspiration for Self”. European Congress on Object-Oriented Programming, 1995.
  - Ungar & Smith. (1987). “Self: The power of simplicity”. Actas del OOPSLA.

Bibliografía

- Sobre Self y el modelo de prototipos. Cuestiones prácticas sobre desarrollo de aplicaciones:

  - Ungar, Chambers *et al*. (1991). "Organizing programs without classes". *Lisp and Symbolic Computation* 4(3), Kluwer Academic Publishers, June, 1991
  - Chambers, Ungar, Chang y Hözle. (1991). "Parents are Shared Parts: Inheritance and Encapsulation in Self". Lisp and Symbolic Computation 4(3), Kluwer Academic Publishers, June, 1991

Bibliografía

- Sobre Kevo y el modelo de prototipos:

  - Taivalsaari, Antero (1996). *Classes Versus Prototypes: Some Philosophical and Historical Observations*. ResearchIndex, The NECI Scientific Literature Digital Library: http://citeseer.nj.nec.com/taivalsaari96classes.html
  - Antero Taivalsaari (1996): On the Notion of Inheritance. *ACM Comput. Surv. 28*(3): 438-479
  - Antero Taivalsaari: Delegation versus Concatenation or Cloning is Inheritance too. *OOPS Messenger 6*(3): 20-49 (1995)
  - Taivalsa, A., Kevo - a prototype-based object-oriented language based on concatenation and module operations. University of Victoria Technical Report DCS-197-1R, Victoria, B.C., Canada, June 1992

Lenguajes de programación orientados a objetos y basados en prototipos

Grupo IMO
Área de Lenguajes y Sistemas Informáticos
Departamento de Informática

Conferencia en la *Escuela Superior de Informática*,
Universidad de Valladolid

J. Baltasar García Perez-Schofield
http://webs.uvigo.es/jbgarcia/