Conceptos de Orientacion a Objetos y Lenguajes Orientados a Objetos

Gustavo G. Larriera
Grupo de Proyecto OO *
Informe interno 890900

Setiembre de 1989
Revisado: Enero de 1990

Resumen

El objetivo de este trabajo es explicar los principales conceptos y caracteristicas de los lenguajes de programacion orientados a objetos. Se describen conceptos tales como objetos, tipos, clases y herencia, y se muestra un panorama de algunos de los lenguajes orientados a objetos disponibles. Se pretende que este trabajo sirva para introducir las nociones basicas de los sistemas orientados a objetos.

Este trabajo fue usado como texto de un curso preliminar interno del grupo de proyecto.

1 Introduccion

Podemos decir que la *Programacion Orientada a Objetos (PO2)* es en principio una herramienta poderosa para la construccion de sistemas, apoyada en la idea de *reusabilidad del software* [Cox86]. Este concepto es la forma natural de construir nuevas componentes de un sistema a partir de otras ya construidas. Cada componente constituye una *capsula* de datos y programacion, que es independiente y puede ser reutilizada en trabajos futuros. De esta forma, el desarrollo de nuevas componentes a partir de componentes basicas puede hacerse *especializando* las componentes basicas, describiendo los aspectos en que las nuevas componentes difieren de las componentes basicas originales.

La Programacion Orientada a Objetos es, en pocas palabras, una tecnologia de integracion o empaquetado de software. Dicha tecnologia de integracion permite a los proveedores de software fabricar unidades funcionales que puedan ser utilizadas por los consumidores como material componente para construir

*en desarrollo en: Escuela de Informatica, Cuareim 1522, Montevideo-Uruguay.

sus aplicaciones especificas. Es importante destacar que la PO2 es simplemente *una* de las formas de lograr software empaquetado. Se puede mencionar como ejemplo de estos conceptos los *pipes y filtros* de UNIX, que permiten reutilizar y componer programas para construir nuevos programas mas complejos. Otros ejemplos son los *packages* de Ada y los *modulos* de Modula.

2 La Orientacion a Objetos

Para describir caracteristicas de los lenguajes *basados en objetos*, los *basados en clases* y los *orientados a objetos* consideraremos los siguientes conceptos referentes a la orientacion a objetos:

- objetos y mensajes
- tipos
- clases e instancias
- herencia
- abstraccion de datos
- persistencia

2.1 Objetos y mensajes

Los *objetos* son entidades autonomas representadas por estructuras de datos, con un *estado interno* y equipados con un conjunto de *operaciones*. El estado y las operaciones modelan el *comportamiento* de una entidad del mundo real representada por el objeto. El estado de un objeto "recuerda" el efecto de las operaciones aplicadas, a diferencia de una funcion que no tiene memoria y los valores que retorna dependen exclusivamente de sus argumentos. Es por esto, que el valor retornado por una operacion aplicada sobre un objeto depende de su estado interno ademas de los argumentos de la operacion.

Este encapsulamiento de estado y operaciones aplicables al estado facilitan una abstraccion de datos analoga a los *tipos abstractos de datos*. La diferencia fundamental entre ambos conceptos radica en los mecanismos de *herencia* disponibles en el mundo de los objetos, segun se vera mas adelante.

Los objetos que van siendo creados durante la ejecucion de un programa, tienen una *identidad* que se mantiene en el tiempo, aun cuando el estado del objeto cambie. En el momento de que el objeto es creado por una operacion de construccion, recibe una identidad que le es propia y lo individualiza de cualquier otro objeto. De esta forma, dos objetos pueden ser iguales pero nunca identicos, en el sentido de que sus estados internos pueden tener los mismos valores en un momento dado, aunque los objetos son entidades diferentes.

Cuando un objeto es creado, debe especificarse la *clase* del nuevo objeto. Esta clase puede tener asociado un tipo, que sirve a los efectos de "molde". La clase establece la estructura de los datos internos o *variables de instancia* y las operaciones que podran aplicarse a los objetos que sean creados en ella, a las que se denominan *mensajes*.

La informacion privada de un objeto solamente puede ser accedida enviando *mensajes* a los que el objeto responde de alguna manera. Esto puede asimilarse al concepto de llamada a un procedimiento. La respuesta del objeto puede ser de diversa indole: simplemente un cambio en su estado interno (la informacion privada del objeto), envio de mensajes a otros objetos, creacion de nuevos objetos, etc. El conjunto de sentencias que forman el cuerpo de un mensaje -y que por ende, especifican su semantica- se denomina *metodo del mensaje*. Cuando un mensaje es enviado a un objeto *receptor*, este decide el metodo adecuado para responder. El conjunto de mensajes a los que un objeto es capaz de responder se denomina su *protocolo*. El protocolo actua como interface entre el objeto y el mundo exterior. Esta interface consiste en una lista de nombres de las operaciones junto con el tipo de sus argumentos y recorridos. A estas declaraciones se las denominan *signaturas*. Las signaturas son utilizadas por el sistema con el fin de validar los tipos de argumentos y recorridos en las expresiones del lenguaje.

2.2 Tipos

En los sistemas orientados a objetos, el concepto de *tipo* describe aspectos *intensionales* o *estructurales* de los objetos. Podemos decir que un tipo es un conjunto de (posiblemente infinitos) valores dotados de un conjunto de operaciones primitivas que se aplican a tales valores y generan resultados del tipo. Normalmente cada lenguaje dispone de varios tipos primitivos con sus operaciones respectivas, mas operaciones *constructoras de tipos* que permiten definir nuevos tipos a partir de los disponibles. La construccion de tipos nuevos puede hacerse a partir de los tipos primitivos, a partir de tipos nuevos o una mezcla de ambos casos.

El concepto de tipo en un sistema orientado a objetos se corresponde al concepto de *tipo abstracto de datos*, en el sentido de que dispone de una interface al objeto, mas la implementacion del objeto que aparece oculta a los ojos del usuario del tipo [Ban88]. La unica parte visible al exterior es la interface.

La implementacion del tipo especifica su estructura interna o *representacion del tipo* junto con los programas que implementan las operaciones. Las operaciones son totalmente internas al tipo y y no son visibles desde el exterior.

La nocion de tipo facilita el modelado de datos y fuerza al programador a escribir expresiones correctas, ayudado por el sistema que controla la validez de cualquier operacion de asignacion y manipulacion de objetos. Un lenguaje se considera *fuertemente tipeado* si su sistema de tipos puede determinar la compatibilidad entre los tipos de cualquier expresion del lenguaje. En muchos casos, la compatibilidad de tipos se hace a partir de la representacion *estatica* del programa (es decir, en tiempo de compilacion), garantizandose que durante la ejecucion del programa no van a haber errores por expresiones mal tipeadas. A esto se lo denomina *chequeo de tipos estatico*. Por el contario, un sistema puede realizar la validacion de tipos durante la ejecucion de los programas, a lo que se denomina *chequeo de tipos dinamico*.

Entre los tipos se puede establecer una *jerarquia de tipos* que establezca que un tipo es *subtipo* de otro. Algunos sistemas declaran en forma explicita cuando un tipo es subtipo de otro, mientras que en otros casos, se dispone de un conjunto de reglas que establecen cuando un tipo es subtipo de otro, por lo que el sistema puede inferirlo.

La jerarquia de tipos es un ejemplo de *poliformismo por inclusion*: siendo T y T' tipos, si T' es subtipo de T entonces en cualquier expresion en la que se requiera un valor de tipo T' puede utilizarse un valor de tipo T. La relacion subtipo es una relacion de orden (cumple las propiedades reflexiva, transitiva y antisimetrica) y no necesariamente es una relacion completa, por lo que es un orden parcial.

La estructura jerarquica puede ser un arbol dirigido a la que se denomina *herencia simple*: cada tipo es subtipo de un unico tipo. Por otra parte, se puede definir una jerarquia multiple o *herencia multiple*: un subtipo puede tener varios supertipos [ADG*89]. En este caso, la estructura es un grafo dirigido aciclico.

En los sistemas tipeados se dispone de operadores *constructores de tipos* que permiten definir tipos estructurados a partir de tipos primitivos, de otros tipos estructurados creados por el usuario, o de combinacion de ambos casos. En muchos sistemas se dispone de constructoras para definir estructuras de tipo registro, tuplas, uniones discriminadas, secuencias, funciones, valores modificables, etc.

2.3 Clases e Instancias

Las *clases* agrupan a los objetos de acuerdo a caracteristicas comunes que tengan. El *nombre de la clase* denota a los elementos (objetos) existentes en ella. En muchos de los lenguajes a cada clase se le asocia un *tipo*, que especifica la estructura de los objetos existentes en ella. Se dispone normalmente de operaciones para crear objetos nuevos de una clase, para borrarlos de la clase y para iterar por cada uno de los elementos. Todos los objetos creados en una misma clase tienen un comportamiento uniforme.

Cuando un objeto es creado, debe especificarse a que clase pertenece. Cada objeto creado en una clase se dice que es una *instancia de la clase*. Si C es una clase, la notacion *O :=: C* expresa que O es una instancia de la clase C.

En un sistema que dispone de clases se puede establecer una jerarquia entre ellas. La relacion << es un orden parcial entre clases que define *superclases* y *subclases*. Si C y C' son clases, la relacion *C' << C* expresa que todos los objetos (instancias) de C' son objetos de C, o sea C' $\subseteq$ C *(restriccion extensional)*, y que los objetos de C' heredan todas las propiedades que cumplen los objetos de C *(restriccion intensional)*. Se dice que *C' es subclase de C* o bien que *C es superclase de C'*. A esta jerarquizacion se la denomina *generalizacion* o *especializacion*. Suele decirse que los objetos de una subclase son una especializacion de los objetos de la superclase.

Basicamente, una clase dispone de una parte responsable de crear los objetos de la clase y de otra parte responsable de contener a los objetos que se han creado en ella. Las operaciones de creacion de objetos de una clase transmiten al objeto las propiedades que establece la clase.

Una clase puede denotarse como C(V,M), donde V es un conjunto de variables que representan la estructura del estado interno o *variables de instancia*, y M es un conjunto de mensajes. Cada elemento de M es una pareja (P,E), donde P es una signatura y E es el conjunto de expresiones asociadas (metodo). Cuando se crea una instancia O de una clase C, es decir, O :=: C(V,M), O hereda a los conjuntos V y M. El conjunto de variables de instancia de O se denomina O.V y su conjunto de mensajes se los denomina O.M.

Considerense ahora dos clases C(V,M) y C'(V',M') tales que C'(V',M') << C(V,M). Sean O y O' dos instancias O :=: C(V,M) y O' :=: C'(V',M'). Si solamente permitimos que la subclase C' agregue nuevas variables y metodos a los definidos en su superclase C, o eventualemente que haga redefinicion de ellos, tendremos que O.V $\subseteq$ O'.V' y O.M $\subseteq$ O'.M'.

Cuando se permite que una subclase herede en forma selectiva de su superclase, es decir, herede algunas variables de instancia y algunos metodos, se dice que se dispone de mecanismos de tipo *drop*.

La nocion de *clase* tiene algunas similitudes con la nocion de tipo ya presentada, aunque es un concepto aplicable a la ejecucion de un programa, que expresa caracteristicas extensionales de los objetos. El concepto de tipo se refiere a los objetos posibles que se pueden tener, mientras que el concepto de clase se refiere a los objetos que realmente existen en un momento dado en el sistema.

Como observacion podemos decir que las clases se encargan de crear y manipular objetos, mientras que los tipos apuntan mas bien a validar la correctitud de las expresiones. Sin embargo, en muchos sistemas, la diferencia entre tipos y clases es muy sutil. En los sistemas de bases de datos orientadas a objetos, se establece una clara distincion entre ambos conceptos.

2.4 Herencia

El concepto de *herencia* es la caracteristica que distingue a la programacion orientada a objetos, permitiendo la reusabilidad de las partes de un sistema a los efectos de crear y extender los objetos ya definidos. Esto permite la evolucion en el tiempo de un sistema, ademas de ser una herramienta de modelado de datos sumamente poderosa.

La *herencia* es un mecanismo que sirve a los efectos de clasificar a las clases en base a comportamientos compartidos. De esa forma, decimos que una clase *hereda* caracteristicas del comportamiento de otra clase. Cuando se crea una clase a partir de otra, denominada *superclase*, las operaciones que se aplican a los objetos de la superclase son heredadas por los objetos de la nueva clase. Cuando se hereda de una unica superclase, se denomina a la herencia, *herencia simple*; si se hereda a partir de varias superclases, se habla de *herencia multiple*.

Cuando se utiliza el mecanismo de herencia surgen algunas cuestiones en lo que refiere a la herencia de operaciones. Por ejemplo, en muchas situaciones, es deseable utilizar el mismo nombre para operaciones diferentes (por ejemplo, "+" para operar entre reales o complejos). En muchos sistemas es necesario diferenciar los nombres de tales operaciones, aunque a veces es deseable por el programador usar el mismo identificador. Esto causa que el mensaje tenga el mismo identificador "+", mientras que su signaturas (y seguramente sus metodos asociados) sean distintos. En un sistema orientado a objetos podemos definir un tipo Numero con la operacion de suma, y luego definir subtipos, Real y Complejo, donde la suma es redefinida acorde a la implementacion de cada uno de los nuevos tipos. Esta redefinicion para cada uno de los tipos (reales y complejos) se denomina *pasar por alto (overriding)*. Por supuesto que ahora dispondremos de un mismo mensaje que se asocia a dos operaciones diferentes: a esta situacion se la denomina *overloading*. A los efectos de que estas caracteristicas esten disponibles, el sistema no es capaz, durante la compilacion, de asociar los nombres de los mensajes con sus metodos. Esta tarea solamente puede hacerse durante la ejecucion, y se la denomina *asociacion tardia (late binding)*. En general, cuando una operacion definida con argumentos de tipo T, es invocada con argumentos de un subtipo T' de T, se adopta la politica de seleccionar el metodo mas especializado (o sea, el definido en el subtipo).

2.5 Abstraccion de datos

La *abstraccion de datos* es el mecanismo por el cual se puede ocultar la implementacion del comportamiento de los objetos, es decir, la representacion (estructura) de los datos y la representacion (algoritmos) de las operaciones. Decimos que un objeto es una abstraccion de datos cuando su estado interno solamente es accesible via los mensajes disponibles en su protocolo.

2.6 Persistencia de datos

La propiedad de *persistencia* de un dato determina la vida util del mismo. En los lenguajes convencionales, los datos persisten solamente durante la ejecucion del programa que los define. Solamente algunos datos privilegiados sobreviven a la ejecucion de un programa, por ejemplo, aquellos almacenados en una base de datos. Cuando a un lenguaje orientado a objetos se lo dota de la propiedad de persistencia mas control de concurrencia, el lenguaje queda en condiciones de ser una herramienta para la implementacion de bases de datos [MS87], en el sentido de que una base de datos puede ser vista como un objeto con propiedades especiales, y que debe sobrevivir a la ejecucion de los programas que acceden concurrentemente a los datos. Sin embargo, la mayoria de los lenguajes orientados a objetos no ofrecen persistencia, excepto alguna forma de mantener los datos creados en una sesion, grabados en algun archivo del sistema [Ban88]. Tal es el caso de los archivos *imagen* de Smalltalk/V [Dig86].

En la mayoria de los lenguajes procedurales tenemos que los datos almacenados en una estructura de datos persisten en tanto la estructura sea accesible por el usuario. Algunos lenguajes, como Pascal y Modula permiten controlar la persistencia con operaciones que asignan y liberan memoria para las estructuras de datos. Es frecuente encontrar en tales lenguajes algun tipo de datos que admite persistencia aun cuando la ejecucion del programa haya terminado. Por ejemplo, el tipo *file* de Pascal.

En general, solamente un conjunto muy pequeño de tipos ofrece persistencia, forzando al operador a utilizar estructuras que a veces no lo son las mas apropiadas para lograr la ansiada persistencia de los datos.

Segun [AB87], principios que deben cumplir los lenguajes son: *a)* La persistencia no deberia ser propia de algunos tipos, *b)* Todos los valores tienen derecho a ser persistentes, y *c)* Cuando un valor persiste, tambien lo debe hacer su descripcion (tipo).

3 Clasificacion de Lenguajes

De acuerdo a los conceptos mencionados en los parrafos anteriores, estableceremos la siguiente clasificacion de los lenguajes, segun soporten algunas o todas las caracteristicas descriptas:

- basados en objetos = objetos
- basados en clases = objetos + clases
- orientados a objetos = objetos + clases + herencia

Podemos hacer una distincion mas fina en esta clasificacion, si consideramos el concepto de tipo, con lo cual tendriamos lenguajes tipeados o no tipeados.

3.1 Lenguajes, objetos, clases y herencia

Decimos que un *lenguaje basado en objetos* es un lenguaje que soporta el concepto de objeto entre sus caracteristicas. Un lenguaje basado en objetos es un *lenguaje basado en clases* cuando cada objeto pertenece a una clase. Un lenguaje basado en clases se convierte en un *lenguaje orientado a objetos* cuando se ha definido una relacion de herencia entre sus clases.

Segun nuestra definicion y a modo de ejemplo, tenemos que Ada y Modula son lenguajes basados en objetos, mientras que Pascal y Algol no lo son. En Ada, los objetos se denominan *packages*, mientras que en Modula se denominan *modulos*. Los objetos en Ada y Modula no tienen clase asociada, por lo cual dichos lenguajes no se ajustan a nuestra definicion de lenguaje basado en clases. El lenguaje CLU maneja el concepto de *cluster* que puede asimilarse al concepto de clase que hemos manejado, aunque no dispone de mecanismos de herencia. Por lo tanto, CLU es un ejemplo de lenguaje basado en clases, pero no es un lenguaje orientado a objetos. Lenguajes que cumplen con las caracteristicas de los lenguajes orientados a objetos son Simula y Smalltalk.

3.2 Lenguajes y tipos

Un *lenguaje orientado a objetos fuertemente tipeado* es aquel cuyo sistema de tipos dispone de un chequeo estatico de tipos y todos los objetos son abstracciones de datos.

Bajo esta optica, tenemos que los lenguajes orientados a objetos fuertemente tipeados forman un subconjunto dentro de los lenguajes orientados a objetos. Tenemos ejemplos de lenguajes orientados a objetos fuertemente tipeados, como Galileo [ACO85,AGOP88], mientras que otros lenguajes orientados a objetos no lo son. Entre los lenguajes orientados a objetos que no son tipeados, tenemos a Smalltalk [Dig86,GR89,DT88].

Es interesante notar que el tipeado fuerte y la abstraccion de datos son conceptos *independientes* u *ortogonales*: por un lado, el tipeado fuerte es posible aun cuando no se disponga de abstraccion de datos (tal como ocurre en Simula), mientras que por otra parte, podemos disponer de lenguajes con abstraccion de datos sin tipeado fuerte, tal como ocurre en Smalltalk, y aun sin tipos, tal como ocurre en variantes de LISP con orientacion a objetos (ML, Flavors). Se puede argumentar a favor y en contra sobre el requerir que los lenguajes orientados a objetos dispongan de abstraccion de datos y un sistema de tipos fuerte [Weg87,Ban88]. Digamos que parece ser un consenso general que el uso de lenguajes orientados a objetos fuertemente tipeados son especialmente aptos para la programacion a gran escala, mientras que los lenguajes orientados a objetos escasamente tipeados (o aun, sin tipos) resultan adecuados para el desarrollo de prototipos.

4 Conclusiones

Se han presentado los conceptos de uso diario en la programacion orientada a objetos, junto con las principales caracteristicas de los lenguajes que se ajustan a dicha linea. El desarrollo y aceptacion de la programacion orientada a objetos puede ser una evolucion importante dentro del marco de los lenguajes de programacion, aunque es importante hacer notar que aun existen varias cuestiones que deben ser resueltas, como ser: definir un marco formal para construir modelos de datos y resolver problemas de performance que permitan a estos lenguajes competir en igualdad de condiciones frente a otros lenguajes convencionales

[Ban88].

A modo de conclusion, digamos que la PO2 provee un marco apto para el manejo de datos y operaciones, y que ha sido usada exitosamente en aplicaciones de simulacion y prototipeado, desarrollo de interfaces graficas, modelado conceptual de datos, inteligencia artificial y desarrollo de proyectos a gran escala. Se pueden encontrar las raices de la PO2 en el lenguaje Simula 67, mientras que su popularizacion se halla en el lenguaje Smalltalk-80. Ideas propias de la PO2 se han aplicado en ambientes de los lenguajes imperativos (Eiffel, Object Pascal, Objective-C, C++, Smalltalk), lenguajes funcionales (Loops, Flavor, ML, Scheme), lenguajes de programacion en logica (Prolog/V de Smalltalk/V) y en lenguajes de bases de datos (Taxis, Galileo, GemStone, O2, EXODUS, Trellis/Owl, Vbase, Statice). La proliferacion de estos lenguajes, tanto en versiones experimentales, prototipos, asi como en versiones comerciales, muestra el creciente interes en el desarrollo e implementacion de sistemas orientados a objetos.

Referencias

[AB87] MP Atkinson y PO Buneman. Types and persistence in database programming languages. *ACM Computing Surveys*, 19(2):105–190, Junio 1987.

[ACO85] A Albano, L Cardelli y R Orsini. Galileo: a strongly-typed, interactive, conceptual language. *ACM Transactions on Database Systems*, 10(2):230–260, Junio 1985.

[ADG*89] A Albano, A Dearle, G Ghelli, C Marlin, R Morrison, R Orsini y D Stemple. A framework for comparing type systems for database programming languages. In *2nd. Intl. Workshop on Database Programming Languages*, 1989. Oregon, 4-7 Junio.

[AGOP88] A Albano, F Giannotti, R Orsini y D Pedreschi. The type system of Galileo. En M Atkinson, P Bunemann y D Morrison, editores, *Persistence and Data Types*, pags 101–119, Springer-Verlag, 1988.

[Ban88] F Bancilhon. *Object-Oriented Database Systems.* Marzo 1988. Invited lecture 7th ACM SIGART-SIGMOD-SIGACT Symposium on Principles of Database Systems, Austin, Texas.

[Cox86] BJ Cox. *Object Oriented Programming.* Addison-Wesley, 1986.

[Dig86] Digital Inc. *Smalltalk/V Tutorial and Programming Handbook.* Digital Inc., 1986.

[DT88] S Danforth y C Tomlinson. Type theories and object-oriented programming. *ACM Computing Surveys*, 20(1):29–71, Marzo 1988.

[GR89] A Goldberg y D Robson. *Smalltalk-80, The Language*. Addison-Wesley, 1989.

[MS87] D Maier y J Stein. Development and implementation of an object-oriented DBMS. En P Shriver y P Wegner, editores, *Research Directions in Object-Oriented Programming*, pags 355–392, MIT Press, 1987.

[Weg87] P Wegner. Dimensions of object-based language design. En *Proc. OOPSLA '87, ACM SIGPLAN Notices*, pags 168–182, Diciembre 1987.