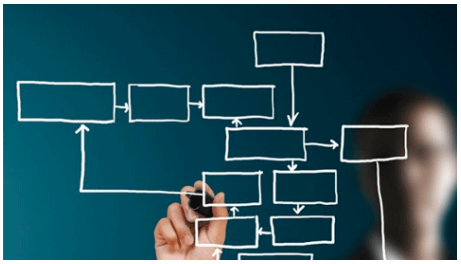
PABLO R. RAMIS

# DISEÑO ORIENTADO A OBJETOS

[ May 22, 2019 at 17:17 – classicthesis version 1.0 ]

# DISEÑO ORIENTADO A OBJETOS

PABLO R. RAMIS



Una introducción

Departamento de Informática
Instituto Politécnico Superior
Universidad Nacional de Rosario

Mayo 2019 – version 1.0

Hablar es barato.
Enséñame el código.

— Linus Torvalds

La educación en computación no puede hacer a nadie un experto programador más que el estudio de pinceles y pigmentos puede hacer a alguien un pintor experto.

— Eric S. Raymond

## RESUMEN

Este trabajo se recopila el material generado para la cátedra de Diseño Orientado a Objetos de la carrera Analista Universitario en Sistemas del Instituto Politécnico Superior.

La cátedra la cual lleva casi 15 años de dictado ha ido cambiando, evolucionando hasta quedar como se encuentra plasmado aquí.

Se verá un balance entre practica y teoría, entre el diseño y la programación. Se busco incorporar ejemplos en C++ y no solo en Java como fue originalmente.

## LICENCIA

# PUBLICACIÓN

Este trabajo fue realizado en LaTeX. Fue editado en texmaker http://www.xm1math.net/texmaker/download.html

*Atención*: se utilizó la plantilla `ClassicThesis`. Puede ser descargada en https://www.ctan.org/tex-archive/macros/latex/contrib/classicthesis/.

# CONTENTS

# LIST OF FIGURES



# LIST OF TABLES

# LISTINGS



# ACRONYMS

**UML** Unified Modeling Language

**OMG** Object Management Group

# Part I

# CONCEPTOS FUNDAMENTALES

# 1 INTRODUCCIÓN

Ei choro aeterno antiopam mea, labitur bonorum pri no. His no decore nemore graecis. In eos meis nominavi, liber soluta vim cu. Sea commune suavitate interpretaris eu, vix eu libris efficiantur.

## 1.1 EL MODELO DE OBJETOS

Los principios básicos del modelo de objeto son: Abstracción, Encapsulamiento, Modularidad, Jerarquía, tipos, concurrencia y persistencia.

De la filosofía planteada por el modelo de objetos se desprenden el diseño y la programación orientada a objetos.

El diseño orientado a objetos representa un desarrollo evolutivo, no una revolución. No rompe con el pasado, sino que se basa en avances ya probados.

### 1.1.1 *Programación Orientada a Objetos*

La programación orientada a objetos es un método de implementación en el que los programas se organizan como colecciones cooperativas de objetivos, cada uno de los cuales representa una instancias de alguna clase, y cuyas clases son, todos ellas, miembros de una jerarquía de clases unidas mediante relaciones de herencia.

En resumen:

- Utiliza objetos en vez de algoritmos para sus bloques lógicos de construcción fundamentales
- Cada objeto es una instancia de alguna clase
- Las clases están relacionadas con otras clases por medio de relaciones de herencia.

La programación puede parecer orientada a objetos, pero si falta cualquiera de estos elementos ya no lo es. Sería simplemente un programa que utiliza tipos de datos abstractos.

Por lo tanto un lenguaje orientado a objetos es aquel que soporta los siguientes requisitos:

- Soporta objetos que son abstracciones de datos con una interfaz de operaciones con nombre y un estado local oculto.
- Los objetos tienen un tipo asociado (clase)

- Los tipos pueden heredar atributos de los supertipos

Durante el curso, trabajaremos con C++, aunque no será el único lenguaje con el cuál veremos ejemplos ya que en algunas situaciones podremos ver que hay ejemplos en ***java*** por adecuarse mejor al tema que se trate.

### 1.1.2 *Diseño Orientado a Objetos*

Es un método de diseño que abarca el proceso de descomposición orientada a objetos y una notación para describir los modelos lógicos y físicos, así como los modelos estático y dinámico del sistema que se diseña.

Cabe resaltar dos cosas: El poder descomponer al sistema y la utilización de notaciones.

Como notación usaremos Unified Modeling Language (UML) - Lenguaje de Modelado Unificado.

### 1.1.3 *Análisis Orientado a Objetos*

El análisis orientado a objetos es un método de análisis que examina los requisitos desde la perspectiva de las clases y objetos que se encuentran en el vocabulario del dominio del problema.

Generalmente, los productos del Análisis Orientado a Objetos sirven como punto de partida para realizar el Diseño Orientado a Objetos, los productos de dicho diseño pueden utilizarse como anteproyecto para la implementación completa de un sistema utilizando métodos de programación orientada a objetos.

## 1.2 ELEMENTOS DEL MODELO OBJETO

Elementos fundamentales:

- Abstracción
- Encapsulamiento
- Modularidad
- Jerarquía

Estos son los rasgos distintivos, hay otros tres que podrían considerarse secundarios:

- Tipos (tipificación)
- Concurrencia
- Persistencia

### 1.2.1 *Abstracción*

Una abstracción denota las características esenciales de un objeto que lo distinguen de todos los demás tipos de objeto y proporciona así fronteras conceptuales nítidamente definidas a la perspectiva del observador.

Se centra en la visión externa de un objeto, y por tanto sirve para separar el comportamiento esencial de un objeto de su implantación.

En resumidas palabras, conozco al objeto, se que hace o para que sirve pero no profundizo en como lo hace o funciona.

Se persigue construir abstracciones de entidades, porque imitan directamente el vocabulario de un determinado dominio del problema.

### 1.2.2 *Encapsulamiento*

Es el proceso de almacenar en un mismo compartimiento los elementos de una abstracción que constituyen su estructura y su comportamiento; sirve para separar el interfaz contractual de una abstracción y su implementación.

El interfaz de una clase captura sólo su vista externa, abarcando la abstracción que se ha hecho del comportamiento común de todas las instancias de la clase. La implementación de una clase comprende la representación de la abstracción así como los mecanismos que consiguen el comportamiento deseado. La interfaz de una clase es el único lugar en el que se declaran todas las suposiciones que un cliente puede hacer acerca de todas las instancias de la clase; la implantación encapsula detalles acerca de los cuales ningún cliente puede realizar suposiciones.

Hay autores que han llamado a esos elementos encapsulados los *secretos de la abstracción*.

En conclusión, la abstracción y el encapsulamiento son conceptos complementarios: la abstracción se centra en el comportamiento observable de un objeto, mientras el encapsulamiento se centra en su implementación que da lugar a este comportamiento. El encapsulamiento se consigue a menudo mediante la ocultación de información, que es el proceso de ocultar todos los secretos de un objeto que no contribuyen a sus características esenciales, típicamente la estructura de un objeto está oculta, así como la implantación de sus métodos.

### 1.2.3 *Modularidad*

La modularidad es la propiedad que tiene un sistema que ha sido descompuesto en un conjunto de módulos cohesivos y débilmente acoplados.

Es el acto de fragmentar un programa en componentes para reducir su complejidad en algún grado. De esta forma se generan fronteras bien definidas y documentadas del sistema.

Encontrar las clases y objetos correctos y organizarlos después en módulos separados con decisiones de diseño.

Más adelante veremos en detalle características de esta particularidad.

### 1.2.4 *Jerarquía*

Es una clasificación u ordenación de las abstracciones.

Las dos jerarquías mas importantes son su estructura de clases (jerarquía de clases) y su estructura de objetos (jerarquía "de partes").

Estos conceptos son los que define cuando un objeto ***es un*** en caso del primero o ***es parte de*** en el segundo.

### 1.2.5 *Tipos (tipificación)*

Los tipos son la puesta en vigor de la clase de objetos, de modo que los objetos de tipos distintos no pueden intercambiarse o, como mucho, pueden intercambiarse sólo de formas muy restrictivas.

El significado de tipos deriva de la teoría sobre tipos de datos abstractos. Un tipo es una caracterización precisa de propiedades estructurales o de comportamiento que comparten una serie de entidades. Para nuestro uso general, los términos tipo y clase son iguales.

Aquí además del diseño, influye mucho el lenguaje con el que se trabajará y la comprobación de tipos que este realice, puede ser estricta o débil o incluso no tener tipos. Esta última, es la más flexible pero a su vez la de mayor riesgo, ya que no puede conocerse si hay incongruencias de tipo hasta el momento de su ejecución. La comprobación estricta de tipos permite utilizar el lenguaje de programación para imponer ciertas decisiones a nivel de diseño, esto comienza a tener mucha importancia en la medida que el sistema es complejo y de gran tamaño. La desventaja que se puede sufrir es que produce una dependencia semántica muy importantes al punto que el cambio de una interfaz en una clase base de la que heredan otras, provocaría la recompilación de todas las subclases.

Ventajas del uso de lenguajes fuertemente tipados:

- Sin la comprobación de tipos, un programa puede "estallar" en plena ejecución
- En la mayoría de los sistemas, el ciclo editar-compilar-depurar es tedioso y conviene la detección de errores en forma temprana.

- La declaración de tipos ayuda a documentar mejor los programas.
- En general, se genera un código mas eficiente si se declaran los tipos.

De la teoría de tipos se desprende un concepto muy importante, el polimorfismo, en el que un solo nombre puede denotar objetos de muchas clases diferentes que se relacionan por alguna superclase común. Ej: Se posee un objeto TransporteDeCarga, este será clase base de los subtipos de transportes del sistema: Camión, Tren y Avión. TransporteDeCarga posee una función que es llenarBodega(), por lo tanto, los tres subtipos también disponen de esa funcionalidad por heredarla pero como cada uno con características diferentes, o sea, dicha función se comportará de manera distinta en cada uno, y en caso que alguno de esos vehículos tenga la necesidad de poseer distintos modos de cargas, por ejemplo dependiendo el material a guardar el modo de llenado es otro, pueden especificarse estas formas siempre invocando a llenarBodega(materialLiquido) o llenarBodega(materialSólido).

### 1.2.6 *Concurrencia*

La concurrencia se centra en la abstracción de los procesos y la sincronización de los mismos. Permite a diferentes objetos actuar al mismo tiempo.

Podemos ver tres enfoques de concurrencia dentro del modelo de objetos:

- La concurrencia es una característica de ciertos lenguajes de programación, por ejemplo: en Ada para expresar un proceso concurrente se utiliza task, en Smalltalk se posee la clase Process.
- Se puede usar una biblioteca de clases que soport alguna forma de procesos ligeros, es el enfoque usado por C++ con los threads.
- Pueden usarse interrupciones para dar la ilusión de concurrencia, esto exige mucho conocimiento de ciertos detalles a nivel de hardware.

### 1.2.7 *Persistencia*

Es la propiedad de un objeto por la que su existencia trasciende el tiempo (es decir, el objeto existiendo después de que su creador deja de existir) y/o el espacio (es decir, la posición del objeto varía con respecto al espacio de direcciones en el que fue creado).

Generalmente la persistencia de los datos se logra con el uso de Bases de Datos, pero también se posee y se necesita que variables y

datos de las mismas se conserven para la comunicación de los distintos procesos. Cada lenguaje tendrá una forma de implementar esto: variables propias, globales, elementos del montículo (heaps), etc.

Un enfoque moderno y razonable para la persistencia es proporcionar una capa (en la arquitectura de la implementación del proyecto) orientada a objetos bajo la cual se oculta una base de datos relacional, ejemplo: el uso del frameworks Hibernate.

# 2

# ELEMENTOS BÁSICOS Y CONCEPTOS

ntoSi bien, dependiendo el autor y el contexto (en ocaciones temporal) las definiciones y conceptos pueden ser muchos y variados. Sin embargo, todos estarán dentro del mismo sentido general.

Veremos y analizaremos a varios de ellos, buscaremos ser objetivos en los significados para que la idea del mundo de objetos quede definida.

Antes de comenzar con el significado de un ***objeto*** propiamente dicho, introduciremos otro el cual nos resulta fundamental:

CONCEPTO O TIPO DE OBJETO: Es una idea o noción que nosotros aplicamos a las cosas u objetos en nuestro conocimiento. Es por lo cual reconocemos a los objetos.

La existencia de conceptos lleva a la existencia de un conjunto de objetos que lo satisfacen o lo componen:

| ESCRITORES | POLÍTICOS | FAMOSOS | MÚSICOS | P.PERFECTA |
|---|---|---|---|---|
| Borges | Menem | Borges | McCartney | |
| Cortazar | Alfonsín | Cortazar | | |
| Gorodicher | Macri | McCartney | | |
| | Kirschner | Alfonsin | | |
| | | Macri | | |
| | | Kirschner | | |

Table 1: Conceptos y objetos que responden a ellos.

De un universo de objetos, tenemos tipos de objetos que los agrupan, esto no implica que un objeto no pueda responder a varios tipos, o que existan tipos que no tienen un objeto que lo satisface (podría existir en un futuro Table 1).

En el momento de analizar un sistema, habrá que plantearse, definir, o encontrar los conceptos que existan para dicho sistema y a partir de esto, agrupar a los objetos que existen en ellos. Esto nos ayudará a comprender el modelo que intentamos construir.

Como vemos en Figure 1, además de tener el concepto en si mismo, tenemos una definición, una idea o comprensión del mismo, y además, a los objetos que lo componen.

Para representar a los conceptos, utilizaremos como símbolo a rectángulos, y dentro de ellos estará el nombre del concepto Figure 2.

De las tres partes mencionadas arriba, podría ocurrir que:

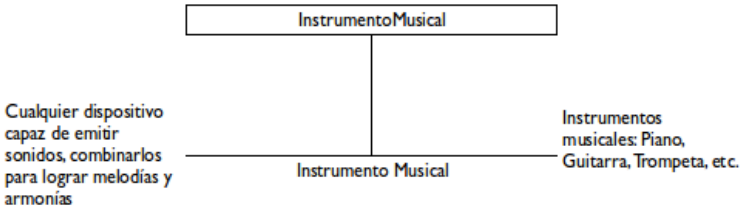



Figure 1: Representaciòn del Concepto. Definido por extensión y comprensión


Figure 2: Concepto

1. Nos falte el nombre. O sea, tengamos la definición, y los objetos que la integran pero no un nombre significativo que lo represente.
2. Nos falte la definición. Tenemos el nombre del concepto, los objetos que responden a el, pero no una definición o idea de su significado.
3. Nos falten los objetos. Tenemos el nombre y la definición del concepto pero ningún objeto existente lo satisface.

De estas tres opciones mencionadas, solo la última es válida. Las dos anteriores son errores o falta de análisis que puede resolverse con mejores consultas a las personas que posean el conocimiento del sistema que se está analizando.

SINÓNIMOS O ALIAS: Es cuando un tipo de objeto puede tener dos nombres diferentes, o sea responden a la misma definición y por lo tanto comparten el conjunto de objetos. Figure 3

HOMÓNIMOS DE TIPOS DE OBJETOS: Cuando un concepto puede estar definido de dos formas diferentes. Esto implica que posee dos conjuntos posibles de objetos que responden a el, uno por cada definición. Figure 4

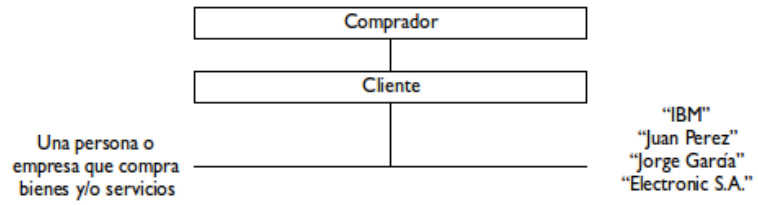



Figure 3: Sinònimo o Alias

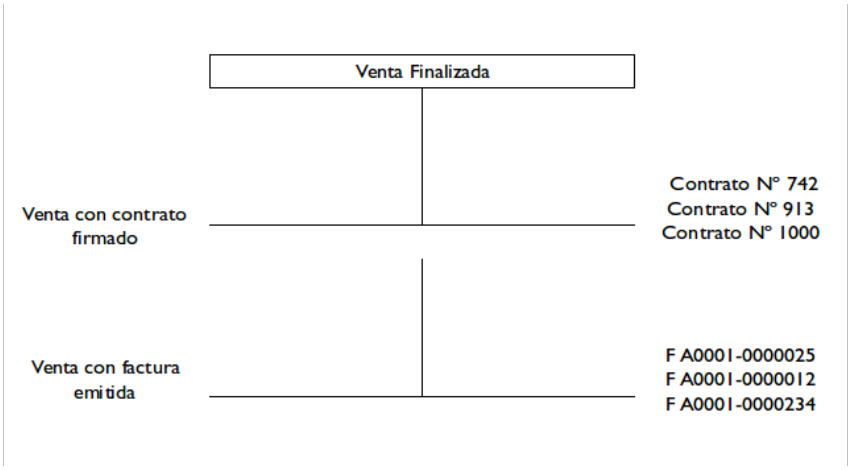



Figure 4: Homonimos

## 2.1 OBJETOS

Son muchas las definiciones que se pueden dar de la orientación a objetos o de los objetos en si mismos. Por ejemplo:

> Un sistema construido con métodos orientados a objetos es uno cuyos componentes son bloques encapsulados de datos y funciones, que pueden heredar atributos y comportamiento de otros componentes de este tipo, y cuyos componentes se comunican a través de mensajes entre sí.

En la definición encontramos conceptos que iremos viendo con mucho mas detalle, como encapsulamiento, herencia, etc. Antes, repasaremos varios de diferentes autores:

OBJECT Una abstracción de algo en un dominio de problema, que refleja las capacidades del sistema para mantener información sobre él, interactuar con él o ambos; Una encapsulación de los valores de los atributos y sus servicios exclusivos (sinónimo de una instancia)[Coad and Yourdon:1990] [2]

CLASE Una colección de uno o mas Objetos con un conjunto uniforme de atributos y servicios incluyendo la descripción de como crear nuevos Objetos de esa Clase. [Coad and Yourdon:1990] [2]

El consorcio de vendedores de software Object Management Group (OMG) nos da la siguiente definición:

UN OBJETO ES UNA COSA. Este es creado como la instancia de un tipo de objeto. Cada objeto tiene una identidad única que lo distingue independientemente de sus características. Cada objeto ofrece una o mas operaciones. [OMG:1992] [7]

Otra que podemos encontrar es:

OBJETO se define como una abstracción del software que modela todo aspecto relevante de una simple entidad tangible o conceptual o cosa de un dominio de aplicación o espacio de solución. Un objeto es una de las entidades de software primarias en una aplicación orientada a objetos, típicamente corresponde a un módulo del software, y consiste en un conjunto de atributos relacionados, atributos, mensajes, operacione y objetos componentes opcionales. [Firesmith:1996] [3]

Un referente de la orientación a objetos nos da otra definición muy particular:

OBJETO SEGÚN PERSPECTIVA DEL CONOCIMIENTO HUMANO es cualquier cosa que cumple con:

1. Una cosa tangible y/o visible.
2. Algo que puede ser aprehendido intelectualmente.
3. Algo hacia el cual se dirige el pensamiento o la acción.

Un objeto tiene un estado, comportamiento e identidad; la estructura y el comportamiento de los objetos similares están definidos en una clase común; el término instancia y objeto son intercambiables. [Booch:1991] [1]

Shlaer y Mellor ofrecen una explicación similar:

OBJETOS es una abstracción de un conjunto de cosas tal que:

1. Todos los objetos del conjunto (Instancias) tienen las mismas características y,
2. Todas las instancias están sujetas y se ajustan al mismo conjunto de reglas y políticas.

Ver: [Shlaer and Mellor:1992] [8]

Ivar Jacobson, otro referente de la talla de Booch, junto a otros colaboradores nos dan una concisa definición:

OBJETO esta caracterizado por un número de operaciones y un estado que recuerda los efectos de dichas operaciones. [Jacobson et al.:1992] [4]

Finalmente, James Martin y James Odell ofrecen estas dos:

OBJETO esta caracterizado por un número de operaciones y un estado que recuerda los efectos de dichas operaciones. [Martin and Odell:1992] [5]

OBJETO es toda cosa que responde a un ***concepto o tipo de objeto*** [Martin and Odell:1994] [6]

En definitiva es *cualquier cosa* que "pueda ser pensada en", "referida a", "descripta por".

Hay más autores y todos tienen una variante o posible definición.

Sigue siendo importante el notar que todos están observando similitudes, en mayor o menor detalles. Hay dos puntos en este momento que si vale la pena resaltar:

1. Un ***objeto*** es una instancia de un ***concepto***
2. Todo ***concepto*** puede ser ***objeto***

Veremos que ***Instancia*** será utilizada en varias ocaciones y dependerá del contexto no siempre significará lo mismo. Ocurre lo mismo con ***Objeto*** y ***Clase***.

El concepto de ***Objeto*** y el de ***Clase*** están ligados y bajo un concepto amplio, pueden ser considerados sinónimos.

La representación de la ***clase*** es la siguiente:

*Nota: Si bien, la representación mas clásica y común del un objeto dentro de los lenguajes de programación es la clase, en algunos lenguajes otras estructuras también son objetos. Por ejemplo, en C++ los **namespace** y las **struct** son también objetos.*

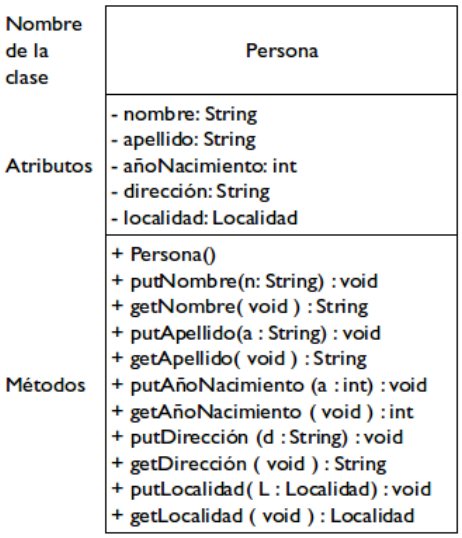



Figure 5: clase

La representación en código la vemos en Listing 1

Listing 1: Declaración de una clase

```
#include<iostream>
#include<string>
using namespace std;

class Persona{
    private:

        string nombre, apellido, direccion;
        int aNacimiento;
        Localidad localidad;

    public:

        Persona();
        void putNombre(string);
        string getNombre();
        void putApellido(string);
        string getNombre();
        void putDireccion(string);
        string getDireccion();
        void setANacimiento(int);
        int getANacimiento();
        void setLocalidad(Localidad);
        Localidad getLocalidad();
}
```

Suele ocurrir que se encuentren diferencias en algunos términos según las bibliografías usadas o las costumbres. Por ejemplo, en una visión exclusivamente de código, del lenguaje, es común ver a la clase como "la declaración del objeto" en una visión estática del mismo, y a su instancia, o sea al objeto creado en memoria como al objeto en forma dinámica.

## 2.2 RELACIONES

Son "vínculos" entre objetos, El vinculo esta representado por una línea que une a dos objetos, dependiendo el tipo de asociación, los extremos de dicha línea finalizarán de distintos modos, flechas u otro gráfico.

Veremos que existen distinto tipos de relaciones entre objetes, La asociación, la generalización y la dependencia.

### 2.2.1 *Asociación*

Se utiliza para especificar que un conjunto de objetos esta conectado con otros objetos. Dada la asociación entre dos clases, se puede establecer la relación entre dos dos objetos (instancias) de ambas.

Una relación binaria se compone de dos extremos, los cuales pueden ser los mismos, o sea, coincidir en la misma clase.


Figure 6: Asociación

Aquí vemos que el alumno está asociado de algún modo con la materia, cuando existan dos objetos que correspondan a cada uno de esos conceptos, ese vinculo puede formarse y generar a una tupla, ej: (Martín, Análisis Matemático) que representa a esa asociación, la cual puede ser, "cursa"

En algunas ocasiones, la relación posee un concepto lo suficientemente importante como para que deba ser considerada un objeto también, por ejemplo, en una relación laboral, entre un empleado y su empresa, existe un contrato de trabajo, el cual posee atributos propios que deben ser representados, esto hace que la relación este representada por el objeto "Contrato".

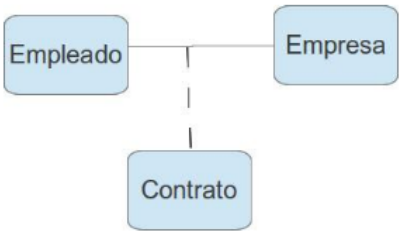



Figure 7: Objeto relación

A este objeto, a esta nueva clase la llamaremos ***Clase Relación***

### 2.2.2 *Adornos*

#### 2.2.2.1 *Nombre y Dirección*

Para una mejor lectura y modelización, las asociaciones pueden tener un nombre para describir la naturaleza de la relación y una dirección para eliminar la interpretación ambigua en caso que sea necesario.

#### 2.2.2.2 *Rol*

Es un adorno que indica el papel que juega la clase en la asociación. La clase puede jugar el mismo o diferentes roles en otras asociaciones. Se vuelve importante el explicitar un rol cuando existen varias asociaciones entre los mismos objetos Figure 8.


Figure 8: Rol

El uso del rol y la dirección no es meramente decorativo, puede llegar a tener mucha importancia cuando los objetos tienen múltiples relaciones con los mismos tipos Figure 9.

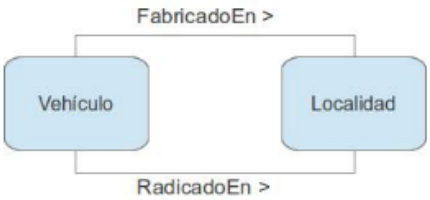



Figure 9: Roles y dirección

Vemos como Vehículo tiene dos referencias al mismo tipo de objeto, Localidad. Gracias a la dirección queda mucho más claro el motivo de esta doble relación.

El código desarrollado para esto podría verse como en Listing 2.

Otra posible situación podría ser Figure 10

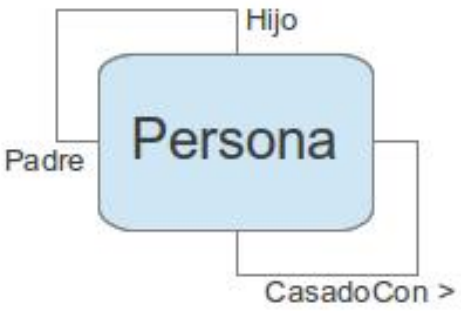



Figure 10: Relaciones Humanas

Aquí vemos otra forma de plantear la relación entre objetos, en este caso, del mismo tipo. Es algo muy común sobre todo cuando se modelizan relaciones humanas

Listing 2: Relación entre clases

```
#include<iostream>
#include<string>
using namespace std;

class Localidad{

    private:
        string codigoPostal;
        string nombre;

     // Aqui sigue el resto de las declaraciones
}

class Vehiculo {
    private:
        Localidad fabricadoEn;
        Localidad radicadoEn;

     // Aqui sigue el resto de las declaraciones
}
```

#### 2.2.2.3 *Multiplicidad o Cardinalidad*

Con este adorno se indica cuantos objetos pueden conectarse a través de una instancia de asociación. Se expresa con un número entero o un rango de enteros. Cuando no se explicita, se asume que es de cero a muchos Table 2.

| POSIBLES EXTREMOS | POSIBLES EXTREMOS |
|---|---|
| 1 (uno) | 2...5 (de dos a cinco) |
| 1...* (uno o más) | * (muchos) |
| 0...1 (cero o uno) | 3 (hasta 3) |
| 0...3 (cero a tres) | |

Table 2: Relaciones con canitdad de objetos.

Como vemos en Figure 11, el gráfico indica que la persona trabajará para 1 y como máximo 5 Empresas que serán sus Empleadores, y la Empresa podrá tener 1 o muchos Empleados

Este adorno es fundamental en el momento del diseño, la omisión o error en el mismo provocará que el desarrollo del código o del modelo de datos (la estructura de la base de datos) se realice con error y por lo tanto que el sistema no cumpla con los requisitos impuestos.

Los ejemplos **??** muestran como se plantean las relaciones tanto en el código como en el modelo de datos. Analizando el diseño el mapeo

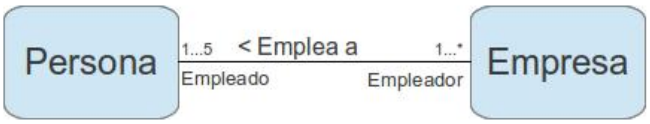



Figure 11: Cardinalidad

que se realiza esta centrado sobre la empresa que emplea a personas (según la dirección que se indica).

En estos ejemplos, aunque incompletos vemos como se establece la relación entre los objetos. La clase Empresa tiene un contenedor para guardar a los objetos Empleados que vaya contratando.

Debe tenerse en cuenta que este no es el único modo de modelizar, todo depende de la interpretación del diseño y en que se acentúa, si en la empresa o en el empleado. Si fuera el segundo caso, veriamos a la Empresa como información del Empleado, en vez de declarar el *vector* donde se encuentra, en los atributos del Empleado podríamos encontrar dentro del bloque *private:* el siguiente código Listing 4

De este modo se define un arreglo para ubicar las 5 posibles empresas en las que puede trabajar indicadas en la cardinalidad del Empleado, si así fuera indicada en el diseño.

Si vemos un posible diseño de la base de datos, la estructura será diferente. Como se ve en Figure 12, la relación se materializa en una tercera tabla a través de las claves primarias de Persona y Empresa. Cabe la mención que las bases de datos relacionales no son orientadas a objetos pero será la situación más común el que tengan que convivir ambos diseños.

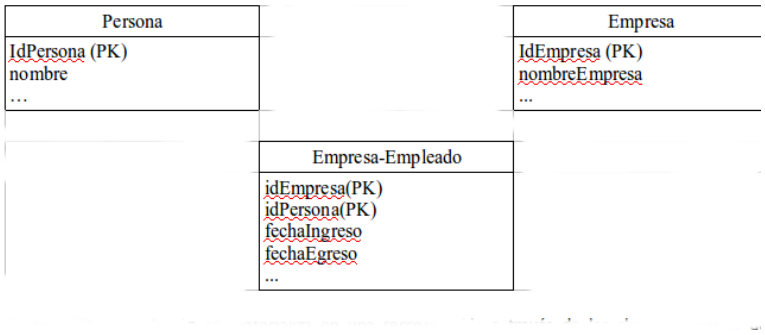



Figure 12: Normalizaciòn de tablas

### 2.2.3 *Agregación*

Es un caso especial de asociación, las asociaciones normales indican la relación entre dos clases de la misma jerarquía, En la agregación

Listing 3: Relación entre clases

```
#include<iostream>
#include<string>
#include<vector>
using namespace std;

class Persona {

    private:
    //atributos
    string idPersona;
    string nombre;
    //aqui continuan las declaraciones

    public:
    //metodos
        Persona (string, string);
    //continuan los metodos

}

class Empresa {

   private:
   //atributos
       vector <Empleado> empleados;
   //continua la definicion de atributos

   public:
   //metodos
   //Lista de metodos y constructores necesarios para manipular
       al objeto

}
```

una de las clases tendrá el papel del "todo" y la otra el papel de "parte".

Resumiendo, esta asociación se utiliza para modelizar objetos complejos que están formados por "partes" las cuales son otros objetos como se ve en Figure 13.

Vale mencionar que los adornos indicados antes, principalmente la cardinalidad son validos también en este tipo de asociación.

Un ejemplo más completo es el que vemos en Figure 14

El modelo que se realiza dependerá del grado de detalle que se pretende alcanzar. Por ese motivo podrán existir mas o menos cantidades de clases componentes.

Listing 4: Relación entre clases

```
class Empleado{
    private:
        Empresa empresas[5];
    //resto del codigo.
}
```

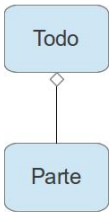



Figure 13: Agregación

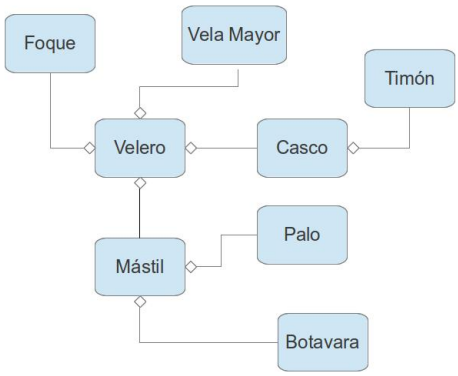



Figure 14: Partes componentes de un velero

### 2.2.4 *Composición*

*El hecho de usar estas relaciones, tanto la agregación como la composición, no implica el no usar el resto de los adornos vistos. Es más, es recomendable que además de indicar estas relaciones cuando corresponden, se las complete con los roles y la cardinalidad principalmente.*

Es una variante de la agregación, con una fuerte relación de pertenencia y vida coincidentes de las partes con el todo. Esto significa que un objeto puede formar parte de sólo un "Todo" compuesto a la vez y que ese todo debe su existencia a que posee esa parte como se ve en el ejemplo de la Figure 15.

## 2.3 CLASIFICACIÓN

Es el acto que resulta de aplicar conceptos (tipos de objetos) a un objeto coo se puede ver en la Figure 16

Podemos decir que es un mecanismo que permite manejar, ordenar a la complejidad del mundo.

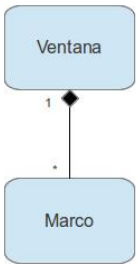



Figure 15: Partes que componen la ventana

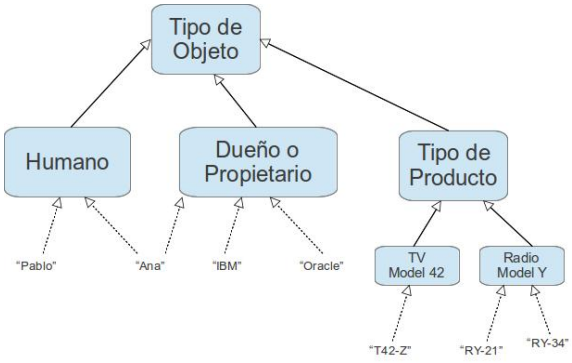



Figure 16: Jerarquia de conceptos clasificados

### 2.3.1 *Herencia - Generalización*

Es el acto o resultado de distinguir un concepto que se encuentra incluido otro como se ejemplifica en Figure 17, es una forma particular de ver a la *clasifiación* enunciada antes.

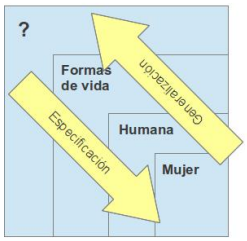



Figure 17: Generalización y Especifiación

Opuesto a la ***generalización*** es la ***especificación*** de la cual se parte de un concepto más general hasta llegar a otros que forman parte de ese tipo de objeto.

De este concepto se desprende dos términos importantes:

Subtipo : refiere a un tipo de objeto que:

- El conjunto de miembros están todos incluidos dentro de otro conjunto.
- La definición es más específica que otra.

Supertipo : se refiere a un tipo de objeto que:

- El conjunto incluye a todos los miembros de uno o mas conjuntos.
- La definición es más general que otras.

Cuando hablamos que un objeto hereda de otro, que es subtipo de otro, o hijo de otro, estamos diciendo que ese objeto "es un" objeto del tipo del padre, o del supertipo. Esto implica que posee todas las características del objeto padre más las suyas propias particulares.

Como vemos tenemos nombres que son sinónimos para los objetos de nivel superior en la generalización o la herencia podemos usar *padre*, *supertipo*; para los objetos de niveles inferiores o que heredan de uno superior: *hijo* o *subtipo*.

En el gráfico del ejemplo de clasificación, Figure 16, podemos notar que el objeto instanciado ***Ana*** hereda de ***Humano*** y de ***Dueño o Propietario***. Esto significa que ***Ana*** **ES** un objeto de tipo ***Humano*** y de tipo ***Dueño o Propietario***.

#### 2.3.1.1 *Particiones*

Una partición es una división de un tipo de objetos en subtipos. Ejemplo: Figure 18

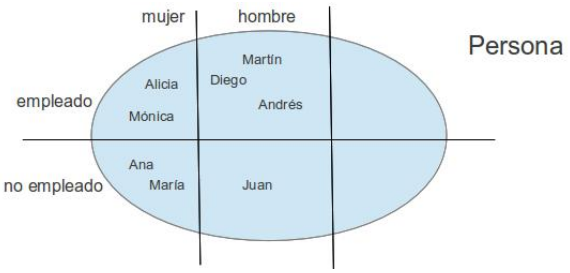



Figure 18: Partición

Entre particiones podrían existir superposiciones pero dentro de una partición no pueden solaparse los objetos, deben ser exclusivos. Ejemplo: Figure 19

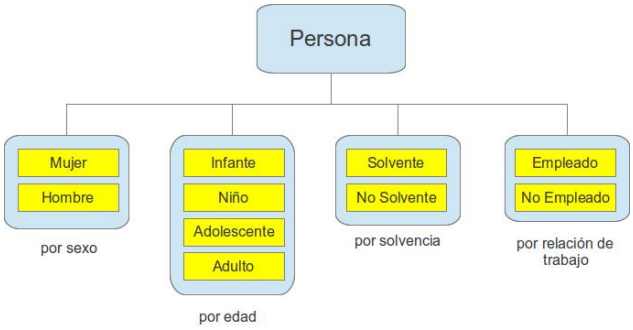



Figure 19: Posibles particiones de un objeto

Con esto, lo que se pretende decir es que la *persona* que este en la partición *Por Solvencia* o es *Solvente* o es *No Solvente* pero no puede ser ambos.

#### 2.3.1.2 *Particiones completas*

Una partición es completa si tiene todos los subtipos especificados como se ve en la Figure 20.

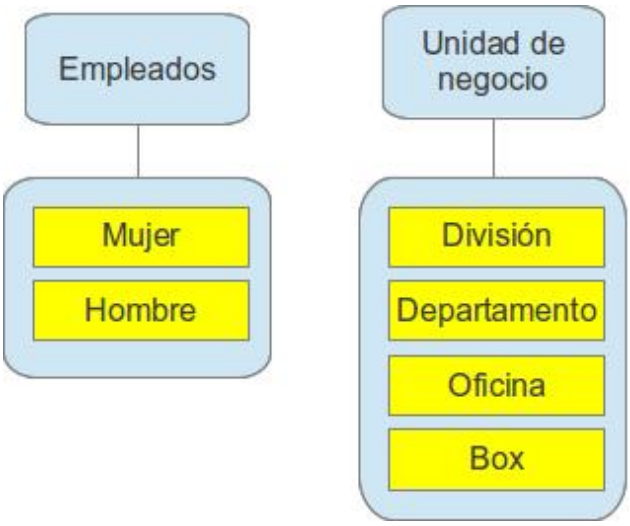



Figure 20: Particiones completas de objetos

Las particiones completas especifican una clasificación obligatoria: El empleado es hombre o mujer; no ambas.

#### 2.3.1.3 *Particiones incompletas*

Una participación incompleta es aquella que tiene una lista parcial de los subtipos que abarca.

No es tan extraño el que se utilicen igual, teniendo en cuenta que uno modeliza solo lo fundamental o importante para el sistema.

## 2.4 ATRIBUTOS, ESTADO, EVENTOS Y MÉTODOS

### 2.4.1 *Atributos*

Un atributo es un valor de datos lógico de un objeto. Se incluirán solo aquellos que que los requisitos sugieran o implican una necesidad de registrar la información.

En la Figure 21 vemos un ejemplo de notación en UML.

Los atributos poseen las siguientes características:

*visibilidad* / nombre: *tipo* [multiplicidad] = valor por defecto propiedadesyconstraints

La *visibilidad* puede ser +, -, #, y significan *public, private, protected o package* respectivamente.

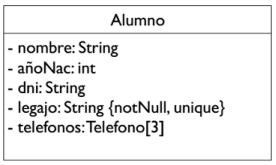



Figure 21: Atributos de la clase Alumno

/ indica que el atributo es derivado. Esto significa que es un atributo que puede ser calculado de otros atributos de la clase.

*Nombre:* es el nombre o frase corta con la que se nombra al atributo. Comienza en minúscula y en caso de ser varias palabras se separan con la mayúscula correspondientes sin espacios ni guiones.

*Tipo:* es el tipo de dato al que pertenece el atributo, puede ser una clase, una interface, o un tipo de dato primitivo.

*Multiplicidad:* especifica cuantas instancias de ese tipo de atributo referenciado.

*Valor por defecto:* es el valor que tomará el atributo en caso que no se complete en forma manual.

*propiedades y constraints:* son aquellas que el atributo debe cumplir una vez instanciado el objeto, ejemplo: no puede ser nulo, o debe ser mayor a 0, debe ser constante.

En el modelo conceptual es conveniente mantener los atributos de forma simple y utilizar la asociación para indicar a los atributos complejos y no es imprescindibles que se expliciten como se muestra en la . Figure 22.


Figure 22: Atributos complejos

#### 2.4.1.1 *La diferencia con la implementación*

El último ejemplo se centra en la descripción conceptual del dominio, del modelo de negocio. Esto no implica que en la visión de la implementación haya diferencias o sea conveniente la descripción completa y detallada. Hay que ser conciente que esto tambien podrá variar de acuerdo al lenguaje que se utilice y las herramientas que este brinde.

#### 2.4.1.2 *Tipos de datos*

Podemos diferenciar dos tipos de datos esencialmente, los primitivos y los no primitivos (objetos). Los primeros son aquellos que no son objetos, no poseen métodos propios para trabajarlos no es necesario instanciarlos con metodos constructores, más adelante veremos este destalle para los objetos.. En C++ dichos tipos primitivos son:

bool : tiene un tamaño de 8 bits y un rango entre 0 y 1, en pocas palabras es cero o es uno (falso o verdadero). Este tipo de dato, es comúnmente usado en condicionales o variables que solo pueden tomar el valor de falso o verdadero. Las variables de tipo bool no suelen llevar modificadores, pues son innecesarios, ya que su rango es solo 0 y 1.

int : El tipo de dato int, tiene un tamaño de 32 bits y un rango entre -2.147.483.648 y 2.147.483.647. Este tipo de dato, es usado para números enteros (sin cifras decimales). A continuación alguna combinaciones con los modificadores:

short int : Tiene un tamaño de 16 bits y un rango entre -32.768 y 32.767.

unsigned short int : Tiene un tamaño de 16 bits y un rango entre 0 y 65535.

unsigned int : iene un tamaño de 32 bits y un rango entre 0 y 4.294.967.295.

long long int : Tiene un tamaño de 64 bits y un rango entre -9.223.372.775.808 y 9.223.375.775.807.

unsigned long long int : Tiene un tamaño de 64 bits y un rango entre 0 y 2exp64.

float : tiene un tamaño de 32 bits, es usado comúnmente en números con 6 o menos cifras decimales. Tiene un rango entre $1,17549 * (e^{-}38)$ hasta $3,40282 * (e^{+}38)$.

double : tiene un tamaño de 64 bits, es usado para números de menos de 15 cifras decimales. Tiene un rango entre $2,22507 * (e^{-}308)$ hasta $1,79769 * (e^{3}08)$.

long double : Tiene un tamaño de 96 bits y una precisión de 18 cifras decimales. Tiene un rango entre $3,3621 * (e^{-}4932)$ hasta $1,18973 * (e^{4}932)$.

char : es simplemente un caracter. la forma mas simple de declaración es char nombre = 'valor' notar que va entre comillas simples.

Todos los tipos de datos pueden ser declarados como vectores o arrays, esto se logra dando un tamaño máximo entre corchetes luego del nombre de la variable de forma idéntica como se realiza en C. Esto no se justifica en caso de los caracteres ya que C++ en su librería *string.h* posee el tipo *string* con una serie de métodos que facilitan mucho su uso.

Otro modificador importante es el que define a la variable como *constante*

Declarado de esta forma el valor no puede ser cambiado, mientras el programa esté en ejecución el valor de dicha variable no puede alterarse por lo tanto no podría hacerse lo siguiente:

En otros lenguajes estos tipos de datos y modificadores pueden llegar a variar.

Listing 5: tipo constante

```
#include<iostream>

using namespace std;

int main()
{
    const float PI  = 3.1416; //Definimos una constante llamada
        PI
    cout << "Mostrando el valor de PI: " << PI << endl;

    return 0;
}
```

Listing 6: tipo constante - uso erroneo

```
#include<iostream>
using namespace std;

int main()
{
    const float PI  = 3.1416; //Definimos una constante llamada
        PI
    cout << "Mostrando el valor de PI: " << PI << endl;
    PI = 2; //Esto generara un error pues PI es de solo lectura (
        constante)
    return 0;
}
```

#### 2.4.1.3 *Visibilidad de los atributos*

En el diseño de la clase, al declarar a los atributos que la conforman debemos indicar que visibilidad poseen: públicos, privados, protegidos, de acuerdo a esto, el objeto será manipulado de diferentes maneras:

public : Se permite que la variable sea accedida de modo directo, o sea, puede dársele valor, cambiarlo o borrarlo con una simple asignación.

private : Solo se permite la manipulación de la variable a través de métodos ( y estos deben ser públicos). De este modo uno puede garantizarse que el seteo, el borrado o los cambios de los valores guardados en la misma se realicen de forma correcta.

protected : es un estado intermedio a los anteriores, Las clases derivadas de la que se esta definiendo, osea que heredan esos atributos, pueden manipularlos directamente como si fueran propios y no a través de los métodos de la clase padre.

#### 2.4.1.4 *Resumiendo*

Los atributos de un objeto son sus características, es lo que lo describe, conforma. Dichos atributos se definen a traves de tipos, estos pueden ser primitivos como los descriptos anteriormente o complejos, o sea objetos con sus propios atributos.

### 2.4.2 *Estado de un objeto*

Es la situación en un momento determinado (cuando se esta observando al objeto) del objeto. Sus atributos, sus relaciones, sus asociaciones existentes, etc.

Dicho de otra forma: Un conjunto de circunstancias o atributos que caracterizan a una persona o cosa en un momento dado; manera o forma de ser: condición [V.A.:1980] [9].

Mas avanzados ahondaremos un poco más sobre este tema ya que estará muy relacionado con los dos que siguen y con la importancia e ingeniería que se necesita para describirlos y representarlos.

#### 2.4.2.1 *Cambio de estado*

Un cambio de estado es la transición de un objeto desde un estado a otro.

Según el punto de vista:

- El cambio de estado es el cambio de la asociación del objeto
- o es el cambio en uno o mas atributos o relación del objeto.

### 2.4.3 *Eventos*

Un evento en la vida de un objeto es un cambio de estado significativo.

si bien estos estarán definidos según el modelo de negocio estudiado, algunos son generales y estarán presente siempre como por ejemplo:

Evento de creación En el evento de creación aparece enteramente un nuevo objeto. Se recibe la indicación que un objeto es creado como la "instancia" de un tipo de objeto. Sin su tipo apropiado de objeto, ningún objeto puede ser reconocido. Una vez creado este objeto pasa a formar parte del conjunto de tipos de objeto.

Evento terminación Es cuando un objeto es borrado de nuestro "conocimiento", deja de existir la instancia del tipo de objeto.

#### 2.4.3.1 *Pre Estado y Post Estado de un Evento*

En muchas ocaciones, al analizar un objeto en un determinado ***estado***, se puede relevar que para llegar a dicha situación se tendrían que haber cumplido ciertos requisitos o que ese objeto cumpla con determinadas condiciones, a esto lo llamaremos ***pre-estado***.

Un pre estado es un estado que debe considerarse en un objeto antes que ocurra un evento.

Un post estado es un estado que debe poseerse en un objeto despues que ocurra un evento.

#### 2.4.3.2 *Eventos externos, internos y temporales*

Evento externo: es cuando el resultado de la operación realizada es externa al dominio analizado.

Evento interno: es aquel en que el resultado de la ocurrencia del mismo queda dentro del dominio analizado.

Evento temporal: es el resultado de operaciones fijadas por tiempos de reloj.

### 2.4.4 *Métodos*

Los distintos lenguajes orientados a objetos han influenciado mucho en los nombres de los conceptos generales del paradigma, según los autores estos nombres pueden ir cambiando a pesar que referencien a las mismas ideas.

Una operación o método es un proceso que puede ser requerido como una unidad, invocado por o al objeto en que pertenece.

Es la implementación del *evento*. Ejemplo: el evento de *Persona Empleada* es el resultado de ejecutar el método *emplearPersona*

Hemos estado hablando que el encapsulamiento es una de las características principales de la orientación a objetos. Esta provee protección alrededor de los datos de un objeto almacenados en sus variables/atributos. Podemos definir el nivel de acceso a esos datos y por lo tanto estos se podrán manipular solo en determinados métodos definidos con el objeto.

#### 2.4.4.1 *Variables de entrada de una operación*

Cada operación puede requerir objetos con quien operar. Estos pueden ser recibidos en los métodos al inicio del mismo, en el momento en que se invocan.

#### 2.4.4.2 *Variables de salida de una operación*

Esto también puede ser conocido como *Valores de retorno*. Una operación o método es una función que dada una entrada retorna una salida.

#### 2.4.4.3 *Operaciones con múltiples eventos*

En muchas situaciones las operaciones son modeladas con un evento simple. Esto suele ser lo ideal, lo recomendable: un método sirve para hacer una cosa y solo esa cosa. Pero a veces multiples eventos pueden ocurrir cuando una operación se completa, dependiendo la complejidad de lo requerido.

#### 2.4.4.4 *Pre y Post condición*

Una pre condición especifica una restricción bajo la cual una operación se desempeña correctamente y una post condición especifica aquellas condiciones que deben resultar cuando se completa el método.

#### 2.4.4.5 *Sintaxis*

La forma de definir en UML a los métodos es la siguiente:

visibilidad nombre ( parámetros ) : tipo de retorno ( propiedades )

Donde: parámetros es:

nombre : tipo = valor por defecto

La visibilidad es idéntica a los conceptos de los atributos. Y los parámetros se encadenan con , cuando se requieren mas de uno.

Cuando el método no retorna ningún valor se declara la palabra *void*.

# 3

# LA CLASE

Ahondaremos un poco mas en el concepto de ***Clase*** ya que para nosotros será la estructura fundamental en el momento de programar el diseño.

## 3.1 LA CLASE COMO ESTRUCTURA

Un concepto fundamental es el de *Tipo de Dato Abstracto*, es lo que permite tener flexibilidad en el diseño, modularizar correctamente. Uno de los principales tipos es la Clase. Ya la hemos mencionado y ahora veremos sus características principales.

¿Cuál es el concepto central de la tecnología de objetos? Hay que pensar dos veces antes de contestar "objeto". Los objetos son útiles, pero no son nuevos en sí mismo: desde que Cobol ha tenido estructuras, desde que Pascal ha tenido registros, C tiene estructuras. Obviamente los objetos siguen siendo importantes para describir la ejecución de un sistema orientado a objetos. Pero la idea básica, de la que todo deriva de la tecnología de objetos, es la de *clase*.

Una clase es un tipo abstracto de datos equipado posiblemente con una implementación parcial de la aplicación.

Como todo tipo de dato abstracto (TDA), una clase es un tipo: describe un conjunto de posibles estructuras de datos, llamadas instancias de una clase. Hablar de tipos de datos abstractos es hablar en forma muy genérica, pero hablar de una clase es hablar de una estructura de datos que se puede representar en el memoria de un ordenador y manipulada por un sistema de software.

Por ejemplo, si se ha definido una clase **STACK**, añadiéndole la información adecuada para su representación, las instancias de esa clase serán estructuras de datos que representan las pilas individuales.

Otro ejemplo, desarrollado es una clase **POINT** modelizando la noción de punto en un espacio bidimensional, bajo alguna representación adecuada, una instancia de esa clase es un estructura de datos que representa un punto. En una representación cartesiana, cada instancia de **POINT** es un registro con dos campos representando las coordenadas horizontal y vertical, X e Y, de un punto.

Una definición que ya hemos dicho es que el concepto de *clase* esta enlazado con el de *objeto*. A modo de diseño, análisis, las clases son los objetos del sistema. Y siendo mas precisos, mas técnicos, un objeto es simplemente la instancia de una clase. Es la materialización de esa clase en la memoria de la computadora.

Una clase es un modelo y un objeto es una instancia de ese modelo. Esta propiedad es tan obvia que normalmente no merecen comentarios más allá de las definiciones anteriores para los que están acostumbrado a la tecnología de objetos, pero en general esto es un error, es facil caer en confusiones.

Veamos el siguiente ejemplo de un libro:

"Se puede identificar un objeto "Usuario" en el espacio de un problema donde el sistema no necesita mantener ningún tipo de información acerca del usuario. Este caso, el sistema no necesita el número habitual de identificación, nombre, privilegios de acceso, y similares. Sin embargo, el sistema no necesita supervisar el usuario, en respuesta a peticiones y proporcionar información de manera oportuna. Y así, debido al servicio requerido en beneficio de la cosa del mundo real (en este caso, el Usuario), tenemos que añadir un objeto que corresponde al modelo del espacio del problema."

En el ejemplo vemos que el objeto representado por la palabra "usuario" se usa en dos sentidos diferentes: El usuario típico del sistema interactivo en discusión y el concepto de usuario en general.

### 3.1.1 *El molde y la Instancia*

Pensemos en un libro, considérelo como un objeto en el sentido mas común del término. Tiene sus propias características individuales: puede ser nuevo o ya manoseado por los lectores anteriores, tal vez escrito por dichos lectores (si fue usado para estudiar o le pusieron el nombre) si es de una biblioteca cuenta con un código de identificación. Sin embargo, las propiedades básicas como título, editor, autor y contenidos, están determinados por una descripción general que se aplica a cada copia individual. Este conjunto de propiedades no define a un objeto, sino a una clase de objetos (un tipo de objeto, en este sentido, las nociones de tipo y clase son idénticas).

Llame a la clase Libro A. Este se define como un cierto molde. Los objetos construidos a partir de este molde, serían las copias del libro, se llaman instancias de la clase. Otro ejemplo de molde sería el molde de yeso que un escultor hace para obtener una versión invertida del diseño de un conjunto de estatuas idénticas; cualquier estatua derivada del molde es una instancia del molde.

Lo que va a escribir en su sistemas de software es la descripción de las clases, tal como una clase **LINKED_STACK** describiendo propiedades de las pilas en una cierta representación. Cualquier ejecución particular de su sistema puede utilizar las clases para crear objetos; cada uno de estos objetos se deriva de una clase, y se llama una instancia de esa clase. Por ejemplo, la ejecución puede crear un objeto de la pila vinculado, derivado de la descripción dada en la clase **LINKED_STACK**; tal objeto es una instancia de la clase **LINKED_STACK**.

La clase es un texto de software. Es estática, es decir, que existe independientemente de cualquier ejecución. Por el contrario, un objeto derivado de la clase es un conjunto de datos creados de forma de estructura dinámica, que sólo existe en la memoria de la computadora durante la ejecución de un sistema.

Esto, por supuesto, está en línea con la discusión anterior sobre los tipos abstractos de datos: cuando especificando a un **STACK** como un **TDA**, no se describe ninguna pila en particular, pero la noción general de pila, un molde del que se puede derivar casos individuales.

Las declaraciones de ***X es una instancia de T*** y ***X es un objeto de tipo T*** serán consideradas como sinónimos para esta discusión. Igualmente hay que tener en cuenta que esto no siempre será así, ya que cuando incorporemos el concepto de *herencia* se tendrá que distinguir el concepto directo de instancia de una clase (construido a partir del patrón exacto definido por la clase) y su instancia en el sentido más general (casos directos de la clase o de cualquiera de sus particularizaciones o casos heredados).

Como vemos, las ideas si bien no son complejas a veces se prestan a una confusión, generalmente debido al uso semántico de las palabras, se puede tener la tentación de usar la palabra objeto por simple en más oportunidades de las convenientes. A veces, cuando nos referimos a la manipulación informática de cosas materiales y claramente visibles, como cuentas bancarias, aviones, documentos y en otras ocasiones de otras tantas que no tienen fuera del sistema representación alguna como pueden ser listas o estados. Lo importante es contextualizar a las palabras y definir claramente su significado para no confundir los términos reales y no confundir al objeto con la clase o la instancia de la misma según su uso.

Algunos lenguajes orientados a objetos, especialmente Smalltalk, han introducido una noción de **MetaClass** para manejar este tipo de situaciones. Un meta-clase es una clase cuyas instancias son clases mismas. Un poco más adelante veremos mejor este tipo especial de clases, al que llamaremos Clase Abstracta.

#### 3.1.1.1 *El rol de la clase*

Para entender el enfoque orientado a objetos, es esencial darse cuenta de que las clases juegan dos funciones que los enfoques pre-OO siempre habían tratado por separado: el módulo y el tipo.

Los lenguajes de programación y otras anotaciones utilizadas en el desarrollo de software (diseño lenguas, lenguajes de especificación, notaciones gráficas para análisis) siempre incluyen tanto alguna facilidad para indicar módulos y algún sistema de tipos.

Un módulo es una unidad de descomposición de software. Hay varias posibles formas de módulo, tales como rutinas y paquetes. Independientemente de la elección exacta de la estructura del módulo,

ya que la descomposición en módulos sólo afecta a la forma del software, y no lo que el software puede hacer, de hecho es posible, en principio, escribir cualquier programa Ada como un solo paquete o cualquier programa de Pascal como un único programa principal. No se recomienda tal enfoque, cualquier desarrollador de software competente utilizarán los módulos del lenguaje que este usando para descomponer su software en partes manejables. Pero si tomamos un programa existente, por ejemplo en Pascal o C, que siempre puede combinar todos los módulos en uno solo y aún así obtener un sistema de trabajo con la semántica equivalente y buena performance cumpliendo con lo que debe hacer. Así que la práctica de la descomposición en módulos es dictado por la ingeniería de software y los principios de gestión de proyectos y no por necesidad intrínseca.

El concepto de Tipo, a primera vista, es muy diferente. Un tipo es la descripción estática de ciertos objetos dinámicos: los diversos elementos de datos que se procesarán durante la ejecución de un sistema de software. El conjunto de tipos por lo general incluye los tipos predefinidos, tales como CHARACTER o INTEGER y así como aquellos que pueden ser definidos por el desarrollador o por los lenguajes: estructuras, punteros, matrices, listas, etc.

El concepto de tipo es un concepto semántico, ya que cada tipo influye directamente en la ejecución de un sistema de software mediante la definición de la forma de los objetos que en el sistema se crean y manipulan en tiempo de ejecución.

### 3.1.2 *La clase como módulo y tipo*

En los enfoques no-OO, el módulo y el concepto de tipo son distintos como digimos antes. La más notable propiedad de la noción de clase es que subsume estos dos conceptos, la fusión en una sola construcción lingüística. Una clase es un módulo, o una unidad de descomposición de software, pero también es un tipo (o, en los casos de generalidad, un patrón de tipo).

Gran parte de la potencia del método orientado a objetos se deriva de esta identificación. La Herencia, en particular, sólo puede entenderse plenamente si la miramos para proporcionar tanto a Módulos de ampliación y el tipo de especialización.

### 3.1.3 *Sistema de tipos uniforme*

Un aspecto importante de la orientación a objetos es por la simplicidad y uniformidad del sistema de tipos, eso se puede resumir en la siguiente propiedad:

REGLA DEL OBJETO Cada objeto debe provenir de una clase.

Esta regla no solo se aplicará a los objetos definidos por el desarrollador (por ejemplo, estructuras de datos con varios campos), sino también a los objetos básicos tales como números enteros, números reales, valores booleanos y caracteres, que todos serán considerados como casos de predefinido clases de biblioteca (INTEGER, REAL, DOUBLE, BOOLEAN, CHARACTER).

Insistir en esto ultimo tiene ventajas:

- Siempre es conveniente contar con un marco simple y uniforme en lugar de muchos casos especiales. Aquí el sistema de tipos se basa completamente en la noción de clase.
- Describir los tipos básicos como TDA y por lo tanto las clases es simple y natural. No es difícil, por ejemplo, ver cómo definir la clase INTEGER, la cual debería tener funciones que abarcan operaciones aritméticas tales como: operaciones booleanas de comparación, suma, resta, y las propiedades asociadas, derivadas de los axiomas matemáticos correspondientes.
- Mediante la definición de los tipos básicos como clases, les permitimos participar en todo el juego de características de la Orientación a Objetos, especialmente de herencia y generalidad. Si no tratamos a los tipos básicos como clases, tendríamos que introducir limitaciones severas y muchos casos especiales.

## 3.2 LA ESTRUCTURA CLASS

Veremos algunas características que son generalmente compartidas por todas las clases.

Veamos el siguiente ejemplo, simple y común: un Punto.

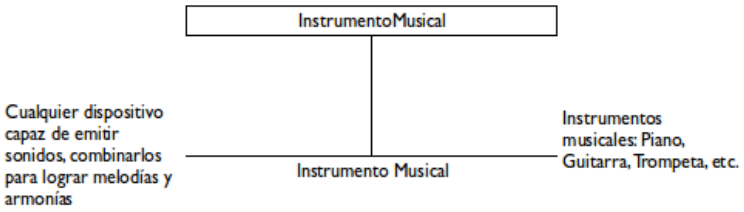



Figure 23: Representaciòn del Concepto. Definido por extensión y comprensión

Para la caracterización de un tipo **POINT** como un tipo abstracto de datos, necesitaríamos las cuatro funciones de consultas para $x, y, \rho, \Theta$. (Los nombres de los dos últimos se especificarán como ro y theta en nuestro sistema). La función x nos da la abscisa de un punto (coordenada horizontal), Y es la ordenada (coordenada vertical), $\rho$ su distancia al origen, $\Theta$ el ángulo con el eje horizontal.

Los valores de x y y para un punto se llaman sus coordenadas cartesianas, los de Θ y ρ son las coordenadas polares. Otra función útil es la de distancia, lo que dará lugar a la distancia entre dos puntos.

La especificación podría contar con comandos como traducir (para mover un punto por un desplazamiento horizontal y vertical dada), rotar (girar el punto por un cierto ángulo, alrededor del origen) y la escala (para llevar el punto más cerca o más lejos de la origen por un factor determinado).

Cualquier tipo de datos abstracto como **POINTS** caracterizados por un conjunto de funciones, que describe el operaciones aplicables a los casos de la ADT. En las clases de implementaciones (ADT), funciones producirán funciones -las operaciones correspondientes a las instancias de la clase.

Además de las funciones u operaciones (que a nivel de especificación irían acompañadas por una explicación de como cada función se implementa: por el espacio o el tiempo, se requiere atributos que serán los datos con los que trabajarán las funciones.

Podríamos definir una clasificación, esto es una definición algo arbitraria en el sentido de los nombres, ya que varían según las bibliografías, pero las características de la clase podría ser así:

1. los atributos, los cuales instanciados estarán en la memoria de la computadora

2. las rutinas: las cuales realizan una acción de cómputo. De las cuales podríamos diferenciar los procedimientos, que no tienen retorno alguno, o sea ejecutan acciones internas; y las funciones, las cuales retornan un resultado.

En general, resumiremos a las operaciones, funciones o rutinas, con el nombre de método, ya que no nos resultará necesario el distinguirlos.

Un tema asociado a esto, es el acceso que se debe tener a los atributos de una clase, ya se hizo una aproximación a esto. El acceso a los atributos debe ser uniforme en todo el diseño e implementación, debe estar definido. Puede ser tanto accediendo directamente, o mediante funciones. Esta última opción es la aconsejable, la que se debe usar siempre. De este modo se utiliza la potencialidad de la clase, y principalmente le da mayor consistencia y seguridad en la manipulación del objeto, ya que solo se hará a través de la interfaz que se publique.

### 3.2.1 *Las clases según los distintos lenguajes*

El primer lenguaje en usar la clase fue Simula. Dependiendo el lenguaje, obviamente la sintaxis cambiará, pero es importante siempre tener en claro el concepto de lo que implica la clase, la rep-

resentación del objeto en si y no el modo en que se escribe en el lenguaje.

Hay muchos lenguajees orientados a objetos, solo veremos algunos ejemplos de como se implementa:

#### 3.2.1.1 *Simula*

Fue el primer lenguaje orientado a objetos, creado por Ole Johan Dahl y Kristen Nygaard en mayo de 1967, fue el que introdujo los conceptos de clase, objeto, herencia, polimorfismo, etc.

Listing 7: Declaración de una clase en Simula

```
Class nombre_clase (lista_ parametros);
        tipo_param_1 param1,...,  tipo_param_n param_n;
Begin
        declaracion_y_especificacion_de_atributos_y_metodos
        cuerpo_de_la_clase
End of nombre_clase;
```

Este sería un ejemplo completo y real para que puedan apreciar:

Listing 8: Clase Rectangulo en Simula

```
Class Rectangulo (Ancho, Alto);
        Real Ancho, Alto;
Begin ! Comienzo de la definicion de la clase;
        Real Area, Perimetro; ! Declaracion de atributos;

        Procedure Actualizar; ! Declaracion de metodos;
        Begin
                Area := Ancho * Alto;
                Perimetro := 2*(Ancho + Alto)
        End of Actualizar;

        Boolean Procedure EsCuadrado;
                EsCuadrado := Ancho=Alto;

        Update; ! Comienzo del cuerpo de la clase;
        OutText("Rectangulo creado: "); OutFix(Ancho,2,6);
        OutFix(Alto,2,6); OutImage
 End of Rectangulo; ! Fin de la clase;
```

#### 3.2.1.2 *C++*

Este lenguaje es una variedad enriquecida y bajo el paradigma de objetos del lenguaje C. Si bien tiene una complejidad elevada, el código y la potencia del mismo es enorme. Fue creado a principio de 1980 por Bjarne Stroustrup. C++ admite tanto la programación orientada a objetos como la procedural, bajo la visión estricta es un híbrido por admitir y permitir mezclar ambos paradigmas.

En Listing 9 vemos el esquema de la clase en C++.

Listing 9: Declaracion de una clase en C++

```
class <identificador de clase> [<:lista de clases base>] {
public:
        //declaracion de atributos publicos
        tipo_1 nombre_1;
        tipo_2 nombre_2

        //declaracion de metodos publicos
        void f_1 (tipo_1 parametro, tipo_2 parametro_2);

protected:
        // declaracion de atributos y metodos protegidos

private:
        // declaracion de atributos y metodos privados
}
```

Un ejemplo completo sería Listing 10:

#### 3.2.1.3 *Smalltalk*

Los orígenes de Smalltalk se encuentran en las investigaciones realizadas por Alan Kay, Dan Ingalls, Ted Kaehler, Adele Goldberg y otros durante los años setenta en el Palo Alto Research Institute de Xerox (conocido como Xerox PARC), para la creación de un sistema informático orientado a la educación. El objetivo era crear un sistema que permitiese expandir la creatividad de sus usuarios, proporcionando un entorno para la experimentación, creación e investigación.

***Python***

Fue diseñado por Guido Van Rossum en 1991. Es multiparadigma, por lo tanto también se lo considera híbrido. Es un lenguaje interpretado.

Listing 10: clase Tupla en C++

```
class Tupla {
private:
        // Datos miembro de la clase "tupla"
        int a, b;
public:
        // Funciones miembro de la clase "tupla"
        void Lee(int &a2, int &b2);
        void Guarda(int a2, int b2) {
                a = a2;
                b = b2;
        }
};

void Tupla::Lee(int &a2, int &b2) {
        a2 = a;
        b2 = b;
}
```

Listing 11: clase Empleado en Smalltalk

```
"Una subclase de Empleado que agrega un protocolo que se
    necesita para los empleados con salario"

 Empleado subclass: #EmpleadoConSalario
        instanceVariableNames: 'posicion salario'
        classVariableNames: ' '
        poolDictionaries: ' ' !
 ! EmpleadoConSalario publicMethods !
        posicion: unaPosicion
                posicion := unaPosicion !
        posicion
                ^posicion !
        salario: unSalario
                salario := unSalario !
        salario
                ^salario ! !
```

#### 3.2.1.4 *Java*

El lenguaje de programación Java fue originalmente desarrollado por James Gosling de Sun Microsystems (la cual fue adquirida por la compañía Oracle) y publicado en el 1995. Su sintaxis deriva mucho de C y C++, pero tiene menos facilidades de bajo nivel que cualquiera de ellos. Las aplicaciones de Java son compiladas a bytecode (.class) que puede ejecutarse en cualquier máquina virtual Java (JVM) sin

Listing 12: Clase Persona en Python

```
class Persona(object):
     def __init__(self, nombre, edad):
         self.nombre = nombre # Una Propiedad cualquiera
         self.edad = edad # Otra propiedad cualquiera
     def mostrar_edad(self): # Es necesario que, al menos, tenga
         un parametro, generalmente: "self"
         print self.edad # mostrando una propiedad
     def modificar_edad(self, edad): # Modificando Edad
         if edad < 0 or edad > 150: # Se comprueba que la edad no
              sea menor de 0 (algo imposible), ni mayor de 150 (
             algo realmente dificil)
             return False
         else: # Si esta en el rango 0-150, entonces se modifica
             la variable
             self.edad = edad # Se modifica la edad
```

importar la arquitectura de la computadora subyacente. Java es un lenguaje de programación de propósito general, concurrente, orientado a objetos y basado en clases.

Listing 13: Definicion de clase en Java

```
[propiedad] class [nombre] extends [clase base]{
        //declaracion de atributos
        [propiedad] [tipo] nombreVariable;
        // declaracion de metodos
        [propiedad] [tipo de retorno] nombreMetodo ( variable1,
            variable2);
}
```

### 3.2.2 *Clientes y Proveedores*

Veremos el modo de uso de una clase. Siguiendo con nuestro ejemplo anterior, sólo hay dos maneras de utilizar una clase como POINT. Una forma es heredar de ella. La otra es convertirse en un punto en cliente de.

CLIENTE - PROVEEDOR Sea S una clase. Una clase C, que contiene una declaración de la forma a: S se dice que es un cliente de S. Entonces S es un proveedor de C.

## 3.3 EL OBJETO COMO ESTRUCTURA DINÁMICA

Hemos visto que hay diferentes significados para algunas de las palabras según el contexto o la generalización que se haga, principalmente esto se da con las palabras como *objeto* o con *instancia*. No son las únicas pero si tal vez las más significativas.

Conceptualmente, y por eso el paradigma se denomina así, el objeto de diseño en nuestro sistema es algo que debe ser representado en el mismo, dentro de un diseño técnico, ya no simplemente funcional, hablamos *clases*, como vimos antes, la cual es algo mas inmaterial, es el código de nuestro sistema. En este enfoque, el objeto está representado por la instancia de la clase, o sea, la materialización de la misma. A esto nos referimos como una estructura dinámica en tiempo de ejecución.

Daremos nuevos conceptos bajo este enfoque.

### 3.3.1 *Objetos*

En cualquier momento durante su ejecución, un sistema orientado a objetos habrá creado un cierto número de objetos. La estructura de tiempo de ejecución es la organización de estos objetos y de sus relaciones.

Veremos algunas de sus características.

#### 3.3.1.1 *¿Qué es un objeto?*

En primer lugar, debemos recordar lo que significa la palabra *objeto* para esta discusión y en este contexto. El concepto es simple y nada ambiguo:

OBJETO Es la instancia en tiempo de ejecución de una clase.

Un sistema de software que incluye una clase C puede, en diversos puntos de su ejecución, crear (a través de las operaciones de creación y la clonación, por ejemplo) instancias de C; tal caso es una estructura de datos construida de acuerdo con el modelo definido de C.

Igualmente, tenemos que tener cuidado, el término *objeto* está tan sobrecargado de significados cotidianos que a pesar de los beneficios que esto puede tener, en un sentido técnico de software ha dado su cuota de confusión. En particular:

- No todas las clases corresponden a tipos de objetos del dominio del problema. Las clases introducidas para el diseño y aplicación no tienen homólogos inmediatos en el sistema modelado. Ellos son a menudo entre los más importantes en la práctica, y el más difícil de encontrar (Ejemplo: Clases Abstractas).

- Algunos conceptos del dominio del problema pueden ser clases en el software (y objetos en la ejecución del programa) a pesar de que no serían necesariamente clasificados como objetos en el sentido habitual del término. Una clase como ESTADO o COMANDO entran en esta categoría por ejemplo.

Consideremos el ejemplo dado con anterioridad. El del Punto, poseía dos atributos y sus métodos. El código sería así :

Listing 14: Declaracion de una clase en C++

```
class Punto {
private:
        float x, y;

public:
        // declaracion de atributos y metodos privados
}
```

La instancia del mismo o sea, el objeto generado en tiempo de ejecución lo podríamos representar de la siguiente forma Figure 24

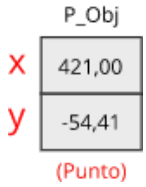



Figure 24: Representaciòn de la instancia de la clase punto.

Debe quedar claro que esto es una simple representación del espacio en memoria de la computadora que estaría siendo usado por la instancia de la clase Punto.

Aquí vemos que el objeto se conforma con el espacio de memoria para los dos atributos que lo conforman, el que posean valores implica que en algún momento de la ejecución del programa ocurrió lo siguiente Listing 15:

Listing 15: Seteo de atributos

```
Point.setX(421.00);
Point.setY(-54.41);
```

#### 3.3.1.2 *Referencias*

Hasta ahora, hemos visto atributos que son tipos de datos primitivos, pero estos limitan mucho la potencia de la clase al momento de definirla como una estructura, como un tipo de dato abstracto y complejo.

Este sería el caso por ejemplo de una clase *Libro*, el cual uno de los campos es el atributo *Autor* de tipo *Escritor* como se ve en Listing 16.

Listing 16: Tipo Abrstracto y Complejo

```
class Escritor {
private:
     string nombre, nombreReal;
     int anioNacimiento, anioMuerte;
  // continua el codigo
}

class Libro {
     string titulo;
     int fecha;
     int cantPaginas;
     Escritor autor;
  // continua el codigo
}
```

Ejecutando al sistema, cada instancia de estas clases serán objetos diferentes, los objetos *libros* que se creen harán *referencia* a un objeto *Escritor*.

Las instancias podrian ser de la siguiente forma Figure 25

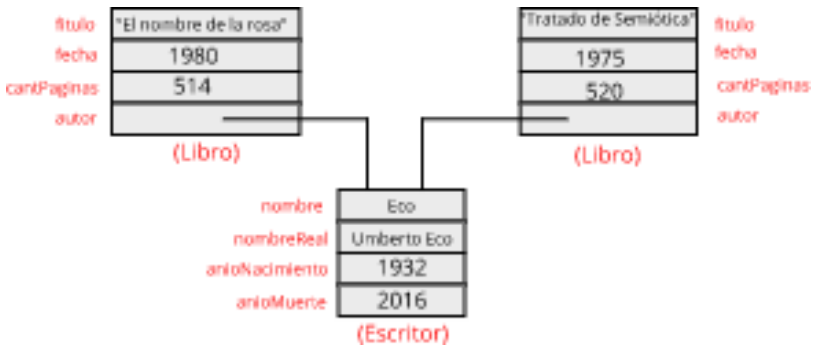



Figure 25: Instancias de estructuras complejas.

Como vemos, ambos objetos libros hacen referencia al mismo autor y por lo tanto al mismo objeto instanciado. En los libros, el valor guardado en la variable author es la dirección de memoria del objeto Escritor que se creó.

Una referencia es un valor en tiempo de ejecución. Puede ser nulo o no. En este último caso, la referencia señala, identifica a un objeto que queda *adjuntado*. Teniendo en cuenta esta definición y siguiendo el ejemplo, si se instanciara un libro de autor anónimo la referencia al objeto Escritor seria nula como vemos en Figure 26.

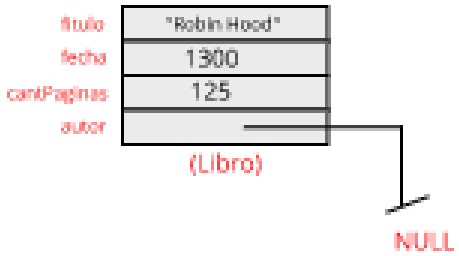



Figure 26: Referencia nula.

### 3.3.1.3 *La identidad de los objetos*

La noción de referencia trae el concepto de la identidad del objeto. Cada objeto creado durante la ejecución de un sistema orientado a objetos tiene una identidad única, independiente de el valor del objeto definido por sus campos. En particular:

1. Dos objetos con diferentes identidades pueden tener campos idénticos.
2. Por el contrario, los campos de un determinado objeto puede cambiar durante la ejecución de un sistema, pero esto no afecta a la identidad del objeto.

### 3.3.1.4 *Sub Objetos*

Por ejemplo podríamos pensar en un objeto libro, en una nueva versión LIBRO2 de la clase de libro, como tener un campo de autor que es en sí un objeto, como informalmente sugiere el siguiente cuadro Figure 27


Figure 27: Sub-Objetos.

Esta situación, esta opción para este ejemplo en particular tiene serias desventajas:

- Es una pérdida de espacio en la memoria.

- Aún más importante, esta técnica no tiene en cuenta la necesidad de expresar el compartir la información. Independientemente de las opciones de representación, los campo author de los dos objetos se refieren a la misma instancia del escritor, si se actualiza el objeto ESCRITOR (por ejemplo, para registrar la muerte del autor), es deseable que el cambio afecta a todos los objetos de libros asociado con el autor dado.

#### 3.3.1.5 *Auto-Referencias*

Algunos de estos conceptos los hemos visto en un sentido de diseño, como un modo de relación.

Un objeto puede contener un campo referencia que se une a sí mismo. Este tipo de auto-referencia también puede ser indirecta. En Figure 28 que se muestra a continuación, el objeto con *Almaviva* en su campo de nombre es su propio dueño (ciclo de referencia), el objeto *Figaro* ama *Susana*, que le encanta *Figaro* (ciclo de referencia indirecta).

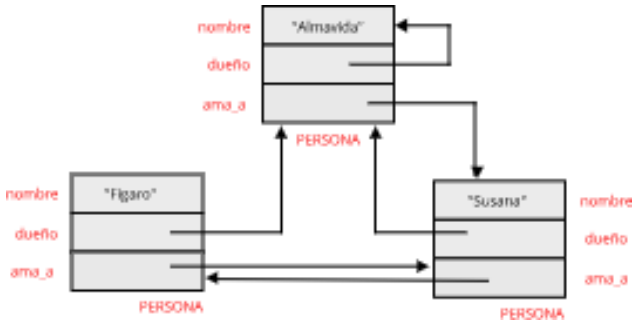



Figure 28: Auto-Referencia directa e indirecta.

Si lo pensamos en el modo de diseño veriamos lo siguiente Figure 29:

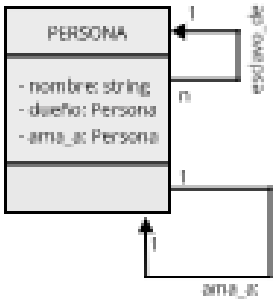



Figure 29: Auto-Referencia vista de diseño.

### 3.3.2 *Manipulando Objetos y Referencias*

#### 3.3.2.1 *La creación*

Hemos visto anteriormente, en forma rápida, como son los eventos de creación, Ahondaremos un poco mas en detalle y veremos como funciona el tema de las referencias en sí misma.

Haremos un paralelo con respecto a Java y C++. En general esto nos mostrará los mismos conceptos pero con las diferencias de implementación de los lenguajes.

El evento de creación, del que hemos hablado anteriormente en otros apartados, podría ser invocado implicita o explicitamente según el lenguaje. En general, el modo de hacerlo es a través de ***new*** para un objeto X, donde X es un tipo de referencia basado en una clase C.

En java es necesaria que la creación sea explicita como vemos Listing 17.

Listing 17: Instanciacion en Java

```
public class Ejemplo{

        private String ejemplo;

        public Ejemplo(){ ejemplo = null;}
        public Ejemplo(String e){ ejemplo = e;}

        public setEjemplo(String e){ ejemplo = e;}

    public static void main (Strings []args){
        Ejemplo Ej; /* Se declara el objeto */
        Ej = new Ejemplo(); /* Se instancia */

        Ej.setEjemplo("Hola"); /* Se lo utiliza */
    }

}
```

Como vemos, debemos invocar la creación ya que la sola declaración no genera una instancia del objeto.

En C++ se mantiene la filosofía de C. Puedo declarar variables o referencias a variables y las clases son una variable en el lenguaje, un tipo de dato.

En Listing 18 vemos las variables, declaradas localmente en el main. C++ invoca implicitamente al constructor por defecto para crear al objeto.

En caso que se omita la declaración de un constructor, C++ impone uno por defecto pero no inicializa las variables, o sea, estas tendrán la basura que este en momoria en el momento de la instancia.

Listing 18: Declaracion de objetos en C++

```
class claseC {
    /* atributos privados */

    claseC() { };

    /* Metodos publicos */
}

int main (){
    claseC C; /* se invoco al constructor */

    C.setVar("Hola"); /* se utiliza al objeto */
}
```

El código cambia si en vez de usar variables por valor, utilizamos referencias a los objetos, o sea, punteros. En Listing 19 vemos como cambia el ejemplo anterior:

Listing 19: Declaracion de objetos en C++

```
class claseC {
    /* atributos privados */

    claseC() { };

    /* Metodos publicos */
}

int main (){
    claseC *C; /* se declara el puntero */

    C = new claseC(); /* se instancia el objeto */

    C->setVar("Hola"); /* se utiliza al objeto */
}
```

En este caso es necesario hacer de forma explicita el llamado al constructor.

Hagamos una pausa y veamos algunas observaciones. El código debe estar sujeto a lo que es nuestro diseño, por lo tanto, no sería correcto que no se expliciten constructores y se utilicen los que por defecto haga el sistema o que por ejemplo, todas las variables sean inicializadas en null cuando esto puede no ser correcto para nuestro modelo de negocio.

Es responsabilidad del diseñador y del programador proveer la interfaz correcta para manipular a los objetos aprovechando la potencia del leguaje que se use.

#### 3.3.2.2 *Constructores en C++*

Veremos algunos ejemplos más sobre la sintaxis de los constructores y la forma de invocarlos.

Listing 20: Uso de constructores por defecto en C++

```
class claseC {
    /* atributos privados */

    /* Metodos publicos */
}

int main (){
    claseC c1; /* se instancia con el constructor por default que
         asigno implicitamente */
    claseC c2(); /* Esto es un error*/
    /* no deben usarse los parentesis cuando el constructor no
        recibe parametros*/
}
```

En Listing 21 se define un constructor que recibe parametros.

Del ejemplo planteado por Listing 21 se desprende una elegante implementación como vemos en Listing 22. C++ hace un tratamiento de los datos primitivos como objetos

#### 3.3.2.3 *Comparación, copia y clonación de objetos*

Intuitivamente diremos que dos objetos son iguales si sus atributos son iguales. Con respecto a la comparación de referencias, debemos tener en claro lo que se pretende: La referencia tecnicamente es la dirección de memoria en que se encuentra un objeto determinado, por lo tanto, al comparar dos referencias, si son iguales, implica que señalan al mismo objeto, pero esto no siempre es lo recomendable o lo requerido, muchas veces se debe tener que observar al objeto por sus valores contenidos para compararlos.

El desasignar la referencia de un objeto es asignar *null* a la misma. Esto no hace que el objeto se borre de la memoria, simplemente se quita la dirección al mismo, dependiendo del lenguaje que se use, se deberá tener que liberar la memoria manualmente o no, en caso que tenga un *garbage collector*.

La clonación es la generación de un objeto idéntico a otro. Hay que tener en claro que a partir de ese acto, cada uno tiene una vida independiente.

Listing 21: Uso de constructores con parametros en C++

```
class claseC {
    /* atributos privados */
private:
    int a, b;

    claseC (int, int);

    /* Metodos publicos */
}
claseC::claseC (int x, int z){
        a = x;
        b = z;
}
int main (){
    claseC c1; /* Esto es un error, no existe constructor sin
        parametros */
    claseC c2(3, 6); /* instancia el objeto de tipo claseC */

}
```

Listing 22: Uso de constructores con parametros en C++

```
class claseC {
    /* atributos privados */
private:
    int a, b;

    claseC (int, int);

    /* Metodos publicos */
}
claseC::claseC (int x, int z): a(x), b(z) {

}
int main (){
    claseC c1; /* Esto es un error, no existe constructor sin
        parametros */
    claseC c2(3, 6); /* instancia el objeto de tipo claseC */

}
```

Para poder ver si son iguales, no podremos hacer una comparación de las referencias, cada uno tendría una dirección de memoria diferente, el método equals() deberá comparar atributo por atributo o sobrecargar al operador = en caso de C++.

Vale aclarar una diferencia entre dos conceptos: *clonación* y *copia* de dos objetos. En el primero se crea un objeto idéntico al que se poseía. En el segundo, los dos objetos existen y

#### 3.3.2.4 *Persistencia de los objetos y la relacion con sus referencias*

Cada vez que un mecanismo de almacenamiento almacena un objeto, debe guardar con el los objetos dependientes del mismo. Cada vez que un mecanismo de recuperación recupera un objeto almacenado previamente, también deberá recuperar a todas las referencias que este posea guardadas.

Part II

# APPENDIX

# BIBLIOGRAPHY


[1] Grady Booch. *Object–Oriented. Design, with Applications*. Redwood City, CA, USA: Benjamin/Cummings, 1991.

[2] Peter Coad and Edward Yourdon. *Object–Oriented Analysis*. 2nd. Englewood Cliffs, NJ, USA: Yourdon Press; Prentice Hall, 1990.

[3] Donald G. Firesmith. *Oriented–Object Requirements Analysis and Logical Design*. 1st. Wiley Professional Computing. Wiley, 1996.

[4] Ivar Jacobson, Magnus Christerson, Patrick Jonsson, and Gummar Övergaard. *Object Oriented Software Engineering: A Use Case Driven Approach*. Reading, MA, USA: Addison–Wesley, 1992.

[5] James Martin and James Odell. *Object Oriented Analysis and Design*. Englewood Cliffs, NJ, USA: Prentice Hall, 1992.

[6] James Martin and James Odell. *Object Oriented Design. A Foundation*. Englewood Cliffs, NJ, USA: Prentice Hall, 1994.

[7] OMG. *Object Analysis and Design, Volume 1, Reference Model*. Draft 7.0. Framinghan, MA, USA: Hartley & Marks Publishers, 1992.

[8] Sally Shlaer and Stephen Mellor. *Object Lifcycles. Modeling the World in States*. Englewood Cliffs, NJ, USA: Yourdon Press; Prentice Hall, 1992.

[9] V.A. *Webster's New World Dictionary*. New York: Simon and Shuster, 1980.