ESCUELA DE PREPARACIÓN DE OPOSITORES
E. P. O.
C/. La Merced, 8 – Bajo A Telf.: 968 24 85 54
30001 MURCIA

INF27

Programación orientada a objetos. Clases. Herencia. Polimorfismo. Lenguajes.

SAI29

Programación orientada a objetos. Clases. Herencia. Polimorfismos.

Esquema.

1 Introducción.

En la construcción de software existen determinados factores que contribuyen a su calidad. Estos factores podemos clasificarlos en externos e internos. Los factores externos son los que pueden ser detectados por los usuarios y son los que realmente les preocupan (calidad externa). Los factores internos sólo los perciben los diseñadores e implementadotes, y son el mecanismo para conseguir la calidad externa (calidad interna). Entre los factores externos tenemos: Eficiencia, Portabilidad, Facilidad de uso, Funcionalidad, Economía, Integridad y los que se pueden considerar más importantes, Corrección, Robustez, Extensibilidad y Reutilización. Como factores internos tenemos: Modularidad y Legibilidad.

En función de la calidad del software podemos definir la Programación Orientada a Objetos (POO) como un conjunto de técnicas para obtener calidad interna como medio para obtener calidad externa.

Actualmente una de las áreas más candentes en la industria y en el ámbito académico es la orientación a objetos. La orientación a objetos promete mejorar de amplio alcance la forma de diseño, desarrollo y mantenimiento del software ofreciendo una solución a largo plazo a los problemas y preocupaciones que han existido desde el comienzo en el desarrollo de software:

- Falta de portabilidad del código y reusabilidad.
- Falta de extensibilidad.
- Falta de modularidad y legibilidad.

El concepto de POO no es nuevo, lenguajes clásicos como SmallTalk se basan en ella. La POO se basa en la idea natural de la existencia de un mundo lleno de objetos y en que la resolución del problema se realiza en términos de estos objetos.

Las principales dificultades surgidas con la utilización de técnicas estructuradas parten de su enfoque. En las metodologías estructuradas a la hora de realizar un programa se hace una división entre procesos y datos, los procesos son los que guían la lógica del programa. Las dependencias existentes entre datos y procesos quedan reflejadas en el programa; lo que implica que cualquier cambio en el proceso o en los datos podrá suponer cambios importantes en el programa.

La POO es una nueva manera de "atacar" los problemas de programación. Divide un problema en pequeñas unidades lógicas de código (que incluyen datos y funciones), independientes del resto del programa y que interactúan entre sí. A estas pequeñas unidades lógicas de código se les ha denominado objetos para establecer una analogía entre las mismas y los objetos materiales del mundo real.

El desarrollo de software orientado a objetos podemos definirlo como un método de desarrollo de software que basa la arquitectura del sistema en módulos deducidos de los tipos de objetos que se manipulan (en lugar de basarse en la función o funciones a las que el sistema está destinado a asegurar).

Como frase que resume la filosofía de la POO, podríamos decir: "No preguntes primero que hace el sistema, pregunta ¡¡A quien lo hace!!

2 Objetos y clases.

En el proceso de desarrollo de software orientado a objetos, primero es necesario encontrar tipos de objetos relevantes y las operaciones que podemos realizar con ellos, en segundo lugar describir los tipos de objetos encontrados y hallar las relaciones existentes entre ellos. Finalmente usarlos para estructurar el software.

En la terminología de POO los tipos de objetos son lo que conocemos como clases de objetos, o simplemente clases.

Para que la descripción de los tipos de objetos sea lo más correcta, debería ser completa, precisa y no ambigua, e independiente de la representación. Esto nos lleva a utilizar el concepto de Tipo Abstracto de Dato, el cual nos permitirá realizar una descripción formal.

Los TAD proporcionan un mecanismo de descripción de alto nivel, libre de cuestiones de implementación. Una clase es una implementación de un TAD de objetos en un determinado LPOO.

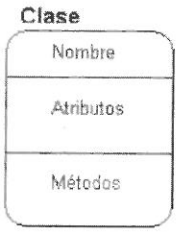
2.1 Clases.

Una clase según hemos indicado es una implementación total o parcial de un TAD. Es una entidad sintáctica que describe objetos que van a tener la misma estructura y el mismo comportamiento.

Una clase tiene en POO doble naturaleza. Es al mismo tiempo un módulo de software y un Tipo de Datos. Como módulo (concepto sintáctico) permite organizar el software, encapsular componentes software y ocultar la implementación de éstos. Como Tipo (concepto semántico), las clases proveen el mecanismo de definición de nuevos tipos, describiendo una estructura de datos (objetos) para representar valores de un dominio y las operaciones aplicables.

Las clases están constituidas por atributos y métodos (rutinas). Los atributos determinan la estructura de almacenamiento para cada uno de los objetos de la clase. Los métodos son las operaciones aplicables a los objetos y debería ser el único modo de acceder a los atributos. Por ejemplo al modelar un banco, encontramos objetos "cuenta". Todos los objetos *"cuenta"* tienen propiedades comunes:

- Atributos: *saldo, titular, ...*
- Operaciones: *reintegro, ingreso, ...*



En POO la existencia de un objeto implica que éste se ha creado a partir de una clase como ejemplificación o instanciación. Es decir, decimos que cualquier objeto es una instancia (ejemplo) de alguna clase.

2.2 Objetos.

Un objeto es una instancia de una clase creada en tiempo de ejecución. Durante la ejecución de un programa OO se crearán un conjunto de objetos. Un objeto es una estructura de datos formada por tantos campos como atributos tiene la clase. El estado de un objeto viene dado por el valor de los campos. Los métodos permiten consultar y modificar el estado del objeto.

Podemos ver un objeto como una "caja negra" que recibe y envía mensajes. Una caja negra que contiene código y datos. Una regla principal de la programación Orientada a Objeto es: al usar un objeto nunca necesitamos ver lo que hay dentro de la caja. Cualquier interacción con los objetos se realiza a través de mensajes, es decir invocando a métodos los cuales reciben los datos, los procesan y regresan un resultado.

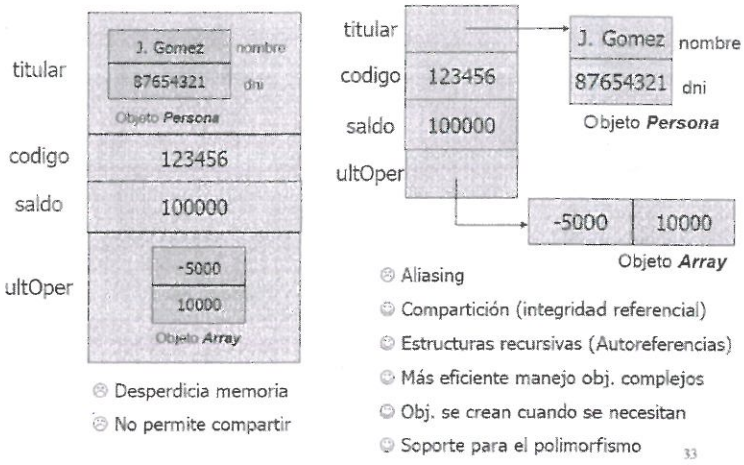
Todos los objetos poseen su propia identidad y se pueden distinguir entre sí. El término *identidad* significa que los objetos se distinguen por su existencia inherente y no por las propiedades descriptivas que puedan tener. Uno de los puntos importantes en los lenguajes OO es el tratamiento que se hace de los conceptos de *identidad* e *igualdad* (íntimamente relacionada con la gestión del almacenamiento de los objetos: semántica de referencia o de almacenamiento, y su principal problema en su tratamiento, el aliasing).

2.3 Atributos y campos.

Un atributo es una característica que describe los objetos de una clase. *Nombre, edad y peso* son atributos de los objetos del tipo (clase) *Persona. Color, peso y año* del modelo son atributos del objeto del tipo (clase) *coche*. Cada objeto instanciado de una clase posee un campo por cada atributo de la clase.

Los tipos de atributos de una clase pueden ser básicos (en C++ int, char, float, double), o bien compuestos (son a su vez clases).

Un objeto instanciado de una clase con atributos compuestos tiene campos que son a su vez objetos. El almacenamiento de estos objetos se puede realizar como subobjetos (semántica almacenamiento) o como referencias (semántica referencia). La utilización de un método y/u otro es particular de cada lenguaje y acarrea ventajas e inconvenientes. El gráfico siguiente trata de aclararlo:



2.4 Métodos y mensajes.

Los métodos implementan las operaciones que se pueden realizar con los objetos de la clase. Están compuestos por una cabecera (Identificador y Parámetros) y un cuerpo (Secuencia de instrucciones). Por ejemplo en Eiffel lo siguiente sería un método:

```
reintegro (suma: REAL) is do
  if puedo_sacar(suma) then saldo:= saldo - suma
end
```

¿En que se diferencia la llamada a un método en POO de un procedimiento en programación estructurada?. La respuesta está en el concepto de mensaje.

¿Qué es un mensaje?. Es un mecanismo básico de la POO mediante el cual se comunican los objetos (son los encargados de activar los objetos). Es la invocación de la aplicación de un método sobre un objeto.

La modificación o consulta del estado de un objeto se realiza mediante mensajes. Un mensaje está formado por tres partes: Objeto receptor, selector o identificador del método a aplicar y argumentos.

Una misma operación puede aplicarse a clases distintas. La implementación de dicha operación en cada clase será el método. El comportamiento de cada operación será distinto dependiendo de la clase del objeto receptor. Todo objeto "conoce" su clase y, por tanto, la implementación correcta de la operación. Tal operación será *polimórfica*; esto es, una misma operación adopta distintas formas en distintas clases.

2.5 Relaciones entre clases.

Entre las clases se establecen una serie de relaciones. Cada Lenguaje de POO admite algunas de estas relaciones. Las más importantes que se dan son:

- **Herencia**. Por esta relación una clase (subclase) comparte la estructura y/o comportamiento definidos en una (herencia simple) o más (herencia múltiple) clases, llamadas superclases.

  Representa una relación del tipo "es un" entre clases.

  Una subclase aumenta o restringe el comportamiento o estructura de la superclase (o ambas cosas).

- **Agregación/Clientela**. Representa una relación del tipo "tener un" entre clases. Tenemos dos posibilidades:
  - Por valor: Es un tipo de relación estática, en donde el tiempo de vida del objeto incluido esta condicionado por el tiempo de vida del que lo incluye. Este tipo de relación es comúnmente llamada **Composición.**
  - Por referencia: Es un tipo de relación dinámica, en donde el tiempo de vida del objeto incluido es independiente del que lo incluye. Este tipo de relación es comúnmente llamada **Agregación**.

2.6 Relaciones entre clases y objetos.

Todo objeto es el ejemplo de una clase, y toda clase tiene 0 ó más objetos. Las clases representan la parte estática, el modelo a partir del cual crear objetos que se van a interrelacionar. Estos objetos y las interrelaciones constituyen la parte dinámica que es en si la ejecución de la aplicación. Es decir, mientras las clases son estáticas, con semántica, relaciones y existencia fijas previamente a la ejecución de un programa, los objetos se crean y destruyen rápidamente durante la actividad de una aplicación.

3 Encapsulamiento y ocultación.

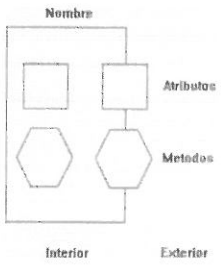
Como hemos visto, cada objeto es una estructura compleja en cuyo interior hay datos y programas, todos ellos relacionados entre sí, como si estuvieran encerrados conjuntamente en una cápsula (módulo). Esta propiedad (**encapsulamiento**), es una de las características fundamentales en la POO.

Los objetos son inaccesibles, e impiden que otros objetos, los usuarios, o incluso los programadores conozcan cómo está distribuida la información o qué información hay disponible. Esta propiedad de los objetos se denomina **ocultación** de la información.

Esto no quiere decir, sin embargo, que sea imposible conocer lo necesario respecto a un objeto y a lo que contiene. Si así fuera no se podría hacer gran cosa con él. Lo que sucede es que las peticiones de información a un objeto deben realizarse a través de **mensajes** dirigidos a él, con la orden de realizar la operación pertinente. La respuesta a estas órdenes será la información requerida, siempre que el objeto considere que quien envía el mensaje está autorizado para obtenerla.

El hecho de que cada objeto sea una cápsula facilita enormemente que un objeto determinado pueda ser transportado a otro punto de la organización, o incluso a otra organización totalmente diferente que precise de él. Si el objeto ha sido bien construido, sus métodos seguirán funcionando en el nuevo entorno sin problemas. Esta cualidad hace que la POO sea muy apta para la reutilización de programas.

La encapsulación y ocultación permite distinguir dos caras de las clases, la interna y la externa. La cara externa es el modo en que las clases (y objetos de éstas) se comunican con el exterior, esta parte se denomina interfaz y puede incluir atributos y métodos. Por el contrario la cara interna constituye el carácter privado de la clase.



Introducir un ejemplo en C++ y/o Java donde se defina una clase con varios atributos y métodos con distintos tipos de modificadores de visibilidad.

4 Herencia.

De acuerdo con B. Meyer, la principal aportación de esta nueva tecnología a la Ingeniería del Software es la posibilidad de reutilización de componentes software. Tal posibilidad se sustenta en una serie de mecanismos y de conceptos, entre los cuales merecen especial atención, por su carácter innovador, la **herencia** y el **polimorfismo**.

Herencia. Es un tipo de relación entre clases, en la cual una clase (subclase) comparte la estructura y/o comportamiento definidos en una (herencia simple) o más (herencia múltiple) clases, llamadas superclases.

Representa una relación del tipo "es un" entre clases.

Una subclase aumenta o restringe el comportamiento o estructura de la superclase (o ambas cosas).

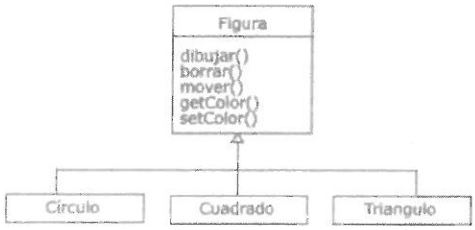
4.1 Superclase y subclase.

Al crear una clase nueva, en lugar de escribir variables atributos y métodos totalmente nuevos, el programador puede indicar que la nueva clase incluya (herede) los atributos y los métodos de una *superclase* previamente definida. Decimos que la nueva clase es una *subclase*. Cada subclase es una candidata para convertirse en superclase de alguna subclase futura.

La *superclase directa* de una subclase es la superclase de la que la subclase hereda explícitamente. Una superclase indirecta hereda desde dos o más niveles hacia arriba en la jerarquía de clases.

Una subclase normalmente agrega sus propios atributos y métodos, de modo que una subclase generalmente es mayor que su superclase. Una subclase es más específica que su superclase y representa un grupo más pequeño de objetos. Sólo con heredar, la subclase inicialmente es prácticamente igual a su superclase. El verdadero valor de la herencia radica en la capacidad de definir en la subclase adiciones a las características heredadas de la superclase o sustituciones de éstas.

Un ejemplo es el clásico de la "figura", quizá usado en diseño de sistemas asistido por computador o simulación de juegos. El tipo base es "figura", los atributos de figura podrían ser: un tamaño, un color, una posición, etc. Las operaciones a realizar: dibujar, borrar, mover, colorear, etc. Un tipo específico de figura puede ser derivado (heredado): circulo, cuadrado, triangulo, ..., cada uno de los cuales puede tener características y comportamientos adicionales. Ciertos tipos de figuras pueden ser rotados. Algunos comportamientos pueden ser diferentes, tal como calcular el área de la figura. El tipo de jerarquía contiene las similitudes y diferencias entre las figuras.

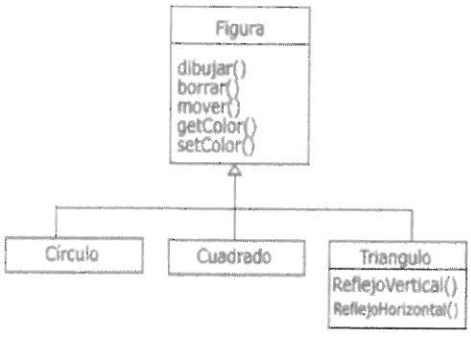


Cuando se hereda se crea un nuevo tipo. Este nuevo tipo contiene no sólo todos los miembros del tipo base, sino algo que es igualmente importante, duplica la interfaz de la clase base. Esto es, todos los mensajes que pueden enviarse al objeto de la clase base pueden también enviarse al objeto de la clase derivada. Sabemos el tipo de una clase por los mensajes que podemos enviarle, esto significa que una clase derivada *es del mismo tipo que la clase base*. En el ejemplo anterior, "un círculo es una figura". Este tipo de equivalencia vía herencia es una de las entradas fundamentales para entender el significado de la POO.

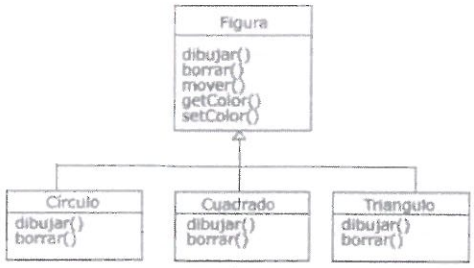
Si se hereda de una clase base y no se hace nada más, los métodos desde la interfaz de la clase base están exactamente en la clase derivada. Esto significa que los objetos derivados no sólo tienen el mismo tipo, sino que también tienen los mismos comportamientos, lo cual no es particularmente interesante.

Cuando se tiene una clase *B* que hereda de *A* entonces *B* incorpora la **estructura** (atributos) y **comportamiento** (métodos) de la clase A, pero puede incluir **adaptaciones:**

- Añadir nuevos atributos y métodos en la clase derivada. Esto significa que la clase base no hace tanto como se necesita, así que se agregan nuevas propiedades. Sin embargo, se debería mirar más de cerca la posibilidad de que la clase base necesite también esas funciones. Este proceso de descubrimiento e iteración del diseño sucede regularmente en la POO.



- La segunda y más importante forma de diferenciar las nuevas clases es cambiando los comportamientos y estructura de la clase base que existe. Los cambios a realizar pueden ser varios y son dependientes del lenguaje (es decir, que no todos están permitidos en todos los lenguajes), los más comunes son:
  - Sobreescribir métodos (no deberían cambiar la semántica del método original).
  - Renombrar métodos y atributos (EIFFEL).
  - Resolver ambigüedades en los nombres de las características.



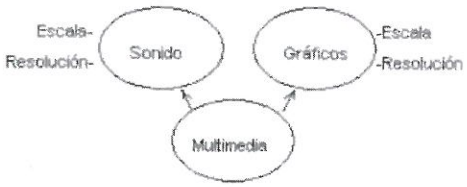
  - Hacer efectiva una rutina.
  - Cambiar el tipo de un atributo o un parámetro de una rutina (EIFFEL).
  - Cambiar el estado de visibilidad de una característica.
  - Convertir en efectiva una rutina diferida o al contrario.
  - Eliminar conflictos de ligadura dinámica cuando existe herencia repetida.

4.2 Herencia múltiple.

Se da cuando una clase derivada tienen más de un ascendiente inmediato. La herencia múltiple puede plantear 2 tipos de problemas:

- Los derivados de la herencia repetida de una misma clase: por ejemplo "profesor universitario hereda 2 veces los atributos de persona". Las soluciones son dependientes de las adaptaciones que permita el lenguaje, así en C++ tendremos en cuenta la replicación (por defecto), la compartición (herencia virtual), la calificación (operador de ámbito) y la regla de dominancia (cuando están implicada la redefinición).

- Colisión de nombres. En las clases base puede haber atributos que se llamen igual. Las soluciones aquí, igualmente son dependientes del lenguaje. En C++ utilizaremos la calificación de nombre (operador de ámbito).



Sólo 2 lenguajes incorporan herencia múltiple: Eiffel y C++.

4.3 Ventajas e inconvenientes de la herencia.

El uso de la herencia añade a los lenguajes POO una serie de ventajas e inconvenientes. Entre las **ventajas** destacamos:

- Reutilización y compartición del código: La herencia nos permite reutilizar comportamientos de una clase, en todo o en parte, lo cual implica no tener que rescribirlo y como éste ya está probado nos presenta mayor confiabilidad.
- Consistencia de la interfaz: Cuando las clases heredan de otras antecesoras, estas se aseguran que el comportamiento exhibido será consistente. Las interfaces de objetos similares (en los que hayamos usado herencia para crearlos) serán equivalentes, lo cual significa que a los desarrolladores y usuarios no se le presentarán colecciones de objetos parecidos, pero con comportamientos diferentes.

Entre las **desventajas** destacamos:

- La velocidad de ejecución.
- La dificultad de aprendizaje. Manejar con soltura una jerarquía de clases puede suponer mucho esfuerzo y tiempo de formación. Esto supone que el aprendizaje puede ser más lento al principio, pero se compensa con la creación más rápida de aplicaciones por reutilización de bibliotecas de clases existentes en los LPOO.

Poner un ejemplo simple de cómo se hereda en C++ y Java.

5 Polimorfismo.

Genéricamente, el polimorfismo es la capacidad de tomar varias formas. Aplicado a los lenguajes de programación, es la capacidad de una entidad de referenciar distintos elementos en distintos instantes. Consideraremos los siguientes tipos:

- Sobrecarga (Overloading, Polimorfismo ad-hoc): un solo nombre de método y muchas implementaciones distintas.
- Sobreescritura (Overriding, Polimorfismo de inclusión): Tipo especial de sobrecarga que ocurre dentro de relaciones de herencia.
- Variables polimórficas (Polimorfismo de asignación): variable que se declara como de un tipo pero que referencia en realidad un valor de un tipo distinto.

5.1 Sobrecarga, overloading, polimorfismo ad-hoc.

La sobrecarga, **overloading**, o polimorfismo ad-hoc, es la posibilidad de tener asociadas a un solo nombre de método muchas implementaciones distintas. La asociación de nombre de método e implementación se realiza en tiempo de compilación (enlace estático) en función de la signatura completa del mensaje.

Podemos distinguir dos tipos de sobrecarga:

Basada en ámbito

```
Factura::imprimir()
ListaCompra::imprimir()
```

Basada en signatura

```
Factura::imprimir()
Factura::imprimir(int numCopias)
```

5.2 Sobreescritura, Overriding o Polimorfismo de Inclusión.

Un método en una clase derivada sobrescribe un método en la clase base si los dos métodos tienen el mismo nombre, la misma signatura de tipos y enlace dinámico (Ligadura dinámica).

Es decir, hay un mismo nombre de mensaje asociado a varias implementaciones (como la sobrecarga) pero:

- en clases relacionadas mediante jerarquías de herencia.
- La signatura de tipos debe ser exactamente la misma
- Los métodos sobreescritos pueden suponer un reemplazo del comportamiento (método independiente del padre, igual que la sobrecarga) o un refinamiento (método no independiente)

La resolución del método a invocar se produce en tiempo de ejecución (enlace dinámico o Ligadura dinámica) en función del tipo dinámico del receptor del mensaje.

El tipo de cualquier otro argumento pasado junto con el mensaje sobrescrito generalmente no juega ningún papel en el mecanismo de selección del método sobreescrito.

En algunos lenguajes (Java, Smalltalk) la simple existencia de un método con el mismo nombre y signatura de tipos en clase base y derivada indica sobreescritura.

En otros lenguajes (Object Pascal), la clase derivada debe indicar que sobreescribe un método.

Otros lenguajes (C#, Delphi Pascal) exigen que tanto la clase base como la derivada lo indiquen.

En C++ es la clase base la que debe indicar explícitamente que un método puede ser sobreescrito (aunque dicha marca no obliga a que lo sea)

```
class Padre{
public: virtual int ejemplo(int a){cout<<"padre";};
};
class Hija : public Padre{
public: int ejemplo (int a){cout<<"hija"};
};
```

5.3 Variables polimórficas.

Una variable polimórfica es aquélla que puede referenciar más de un tipo de objeto. Puede mantener valores de distintos tipos en distintos momentos de ejecución del programa.

- En un lenguaje débilmente tipado todas las variables son potencialmente polimórficas.
- En un lenguaje fuertemente tipado la variable polimórfica está restringida por la herencia.

Podemos usar las variables polimórficas como:

- Variables simples

```
En C++.
Barco *b; //Puntero a Barco que en realidad contiene punteros a las
distintas clases derivadas
```

- Variable receptor de mensaje, especialmente potente cuando se combina con sobreescritura.
- Downcasting (polimorfismo inverso): proceso de asignar un objeto de tipo base sobre uno de tipo derivado. Este uso de las variables polimórficas puede dar problemas. Por ejemplo en C++:

```
Child *c=dynamic_cast<Child *>(aParentptr);
if (c!=0) {…
//nulo si no es legal, no nulo si OK
}
```

- Polimorfismo puro o método polimórfico. Variable polimórfica utilizada como argumento de un método. Un solo método puede ser utilizado con un número potencialmente ilimitado de tipos distintos de argumentos.

En las variables polimórficas es necesario distinguir entre:

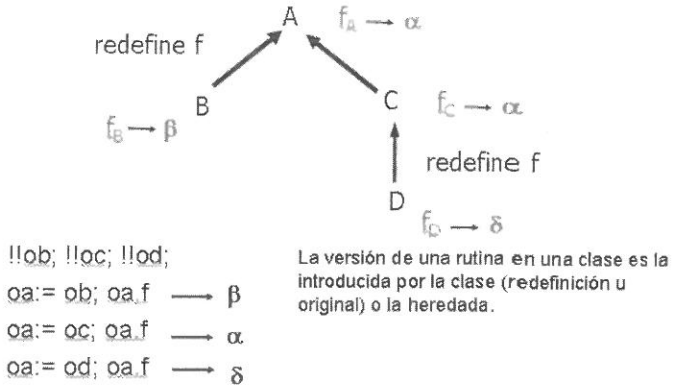
- **Tipo estático:** tipo con el que se declara la variable.
- **Tipo dinámico:** tipo de objeto asociado a una variable en un momento dado.

5.4 Ligadura estática y Ligadura dinámica.

En los casos de Overriding (sobreescritura) la resolución del método a invocar se produce en tiempo de ejecución en función del tipo dinámico del receptor del mensaje. Este proceso se conoce como ligadura dinámica. En contraposición tenemos la ligadura estática. Por ligadura estática entendemos que en tiempo de compilación se establece la asociación del método con la implementación a ejecutar y está no variará en ningún momento.

Así por ejemplo, volviendo al ejemplo de las figuras y siendo f de tipo Figura, ante la llamada *f.dibujar()*, en tiempo de compilación se realiza una comprobación de tipos estática para comprobar que la clase Figura dispone de un método llamado dibujar. Pero no se le asocia esa implementación, sino que será en tiempo de ejecución cuando se compruebe el tipo real del objeto que referencia *f* el cual determinará la implementación a ejecutar.

Veamos un ejemplo en EIFFEL. (!! Crea un objeto y lo asocia a la referencia correspondiente):



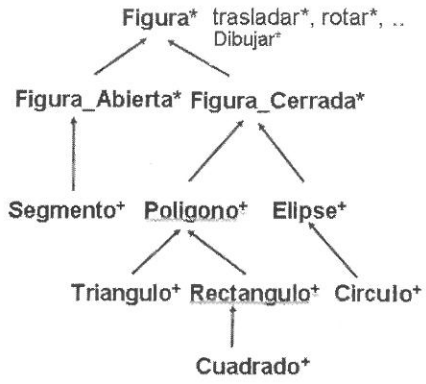
Java utiliza semántica de referencia (todo son referencias) y la ligadura que establece para los distintos métodos es siempre dinámica.

En C++ la ligadura por defecto en las funciones miembro (métodos) es estática. Para conseguir la ligadura dinámica necesitamos utilizar la semántica referencia (punteros) y especificarlo con la palabra clave *virtual.*

	Ligadura Estática	**Ligadura Dinámica**
Eiffel	**frozen** f() is do .. end	Por defecto
Java	public **final** void f() {..}	Por defecto
C++	Por defecto	**virtual** void f() {..}

5.5 Clases abstractas.

Al aplicar la herencia como mecanismo de clasificación, aparecen clases que representan categorías generales o conceptos abstractos, como pueden ser "figura", "colección", "publicación", "dispositivo", ...



Sea la declaración

```
of: FIGURA;  op: POLIGONO
```

y el código

```
!!op;
of:=op;
of.dibujar
```

¿Sería legal?

La rutina dibujar no puede ser implementada en Figura pero *of.dibujar* es dinámicamente correcto. La solución a este problema la encontramos en las clases abstractas.

Una clase abstracta es aquella que contiene métodos (abstractos) que deben ser implementados en sus subclases. Puede ser total (todas los métodos son abstractos) o parcialmente abstracta (sólo algunos métodos son abstractos).

Los métodos efectivos de una clase abstracta especifican una funcionalidad que es común a un conjunto de subclases. Pueden invocar a los métodos abstractos lo cual proporciona un importante medio para escribir código genérico. Incluyen comportamiento abstracto común a todos los descendientes.

No es posible crear instancias (objetos) de una clase abstracta (total o parcial), pero si declarar entidades de estas clases. (En C++ sólo punteros, en Java y EIFFEL referencias).

La declaración de clases abstractas se realiza según la tabla:

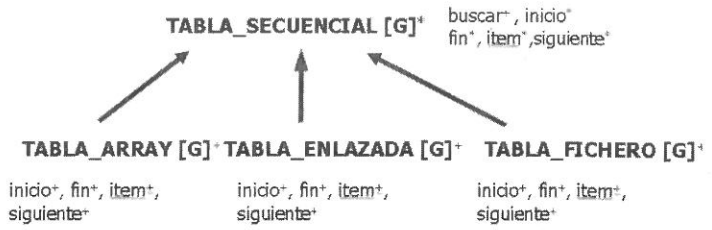
Eiffel	Deferred class Figura feature dibujar is deferred end ...

Java	abstract class Figura {
	abstract void dibujar();
	...}

C++	class Figura {
	public: virtual void dibujar() = 0;
	...

Una subclase de una clase abstracta seguirá siendo abstracta, si no implementa todos los métodos abstractos heredados.

Ejemplo de clase parcialmente abstracta:




```
buscar(v:T): Boolean is do
      from  inicio  until  fin or else v=item  loop
            siguiente
      end
      Result:= not fin
end
```

Clases como TABLA_SECUENCIAL [G] que incluyen un comportamiento común a varias subclases se denominan: "CLASES COMPORTAMIENTO"

6 Genericidad.

La genericidad es la posibilidad de parametrizar las clases de forma que en la definición se trabaje con parámetros formales no específicos, pero que en la ejecución los parámetros reales sean tipos de datos concretos.

Por ejemplo, se crea la clase Lista[P] donde no se hace referencia a ningún tipo de datos concreto, sino a un tipo genérico P.

La genericidad es una facilidad útil para describir estructuras contenedoras generales que se implementan de la misma manera, independientemente de los datos

que finalmente contengan. Se trata de conseguir que el tipo base de la clase sea un parámetro.

```
Class ARRAY[P], class PILA[P], class LISTA[P], ...
```

Veamos un ejemplo en EIFFEL de declaración de una clase genérica:

```
Class PILA[P]
Feature{all}
Count:INTEGER;
x:P;
Put (X:P) is do .. end;
End.
```

Las operaciones aplicables sobre el atributo x de tipo genérico P, es cualquiera aplicable a cualquier tipo.

Cinco posibles operaciones: (en EIFFEL)

```
1)  x:= y          (y es una expresión de tipo P)
2)  y:= x          (y es una entidad de tipo P)
3) x=y  ó  x/=y    ( y es de tipo P)
4) a.f(...,x,...) (x actúa como argumento en un mensaje,
                   el correspondiente parámetro es de tipo P o ANY.
5) Receptor de un mensaje que invoca a una rutina de ANY
```

En C++, la genericidad está permitida y se utiliza la palabra reservada **template** para su definición. Veamos un ejemplo:

```
template <class T> class vector {
public:
vector(const unsigned int & numberElements, const T & initialValue);
~vector();
T &operator [](const unsigned int & index) const;
protected:
      T *data;
};
template <class T>
vector<T>::vector (const unsigned int & numberElements):
size(numberElements)
{ … }
```

En C++ se puede aplicar cualquier operación sobre el tipo genérico. Cuando la clase genérica sea parametrizada con un tipo concreto, se comprobará si existen las operaciones aplicadas genéricamente en el tipo concreto, y será en este momento cuando se genere el error.

7 Corrección y robustez. Programación por contrato.

La reutilización y la extensibilidad propias del paradigma OO no se deben lograr a expensas de la fiabilidad. Fiabilidad implica corrección y robustez.

- Corrección: Capacidad de los sistemas software de ajustarse a la especificación. Asegura que el programa hace lo correcto durante la ejecución normal. Existe una necesidad de incorporar a las clases la semántica de aquello que representan. Se utilizarán los asertos para establecer las condiciones que se deben cumplir.
- Robustez: Capacidad de los sistemas software de reaccionar ante circunstancias inesperadas. Necesidad de manejar errores en tiempo

ejecución. El mecanismo de excepciones proporciona un mecanismo para manejar estas situaciones excepcionales durante la ejecución de un programa.

La programación por contrato trata de mejorar la fiabilidad en el diseño de software. Ideada por B. Meyer está actualmente muy reconocida por la comunidad software.

Eiffel es el lenguaje de POO ideado por Meyer que incorpora POO por contrato. Para un buen número de lenguajes OO (Java, Smalltalk, C++, Python, ...) existen extensiones de distinto tipo destinadas a dar soporte a la programación por contrato.

El diseño por contratos puede ser visto como la aplicación a la construcción de software de los contratos que rigen los asuntos de las personas. Cuando dos personas establecen un contrato se desprenden, de éste, obligaciones y beneficios. Este tipo de contratos en software especifican, en forma no ambigua, las relaciones entre las rutinas y los llamadores de las mismas.

El Diseño por Contrato da una visión de la construcción de sistemas como un conjunto de elementos de software cooperando entre sí. Los elementos juegan en determinados momentos alguno de los dos roles principales *proveedores* o *clientes*. La cooperación establece claramente *obligaciones* y *beneficios*, siendo la especificación de estas obligaciones y beneficios los contratos.

Los contratos de software se especifican mediante la utilización de expresiones lógicas (más una forma de especificar el valor anterior a una computación: *old*) denominadas ***aserciones***.

7.1 Aserciones. Precondición, Postcondición e Invariante.

El lenguaje de aserciones (Eiffel) es muy simple. Las aserciones son expresiones booleanas con unas pocas extensiones (old, ';' en el caso de EIFFEL). Una aserción puede incluir funciones. Ejemplos de aserciones:

```
Positivo: n>0;
autor= not void
not vacia; conta:=old conta + 1
saldo := old saldo –cantidad
```

En la programación por contrato se utilizarán las aserciones en dos casos:

- Especificación semántica de rutinas: PRE y POST-CONDICIONES
- Especificación de propiedades globales de clase: INVARIANTE

Antes de explicar las aserciones y su rol en el Diseño por Contratos, se detallará el concepto de *tripleta de Hoare* la cual es una notación matemática que viene de la validación formal de programas. Sea **A** alguna computación y **P**, **Q** aserciones, entonces la siguiente expresión: { *P* } **A** { *Q* } representa lo que se llama fórmula de corrección. La semántica de dicha fórmula es la siguiente: cualquier ejecución de **A** que comience en un estado en el cual se cumple **P** dará como resultado un estado en el cual se cumple *Q*. Por ejemplo:

{ x > 10 } x := x /2 { x >= 5}

- PRECONDICIONES: Condiciones para que una rutina funcione adecuadamente.

- POSTCONDICIONES: Describen el efecto de una rutina, definiendo el estado final.
- INVARIANTE: Aserción que expresa restricciones de integridad que deben ser satisfechas por cada instancia de la clase si se encuentra en una situación estable. Entendiendo por situación estable, los "momentos" en los que una instancia está en un estado observable: Después de la creación, y antes y después de la invocación remota de una rutina de la clase a la que pertenece la instancia.

La precondición compromete al cliente, ya que define las condiciones por las cuales una llamada a la rutina es válida. Las poscondiciones comprometen a la clase (donde se implementa la rutina) ya que establecen las obligaciones de la rutina. Es muy importante notar que la precondición y la poscondición que definen el contrato forman parte del elemento de software en sí.

```
put (elemento: T; key: STRING) is
-- Insertar en la tabla elemento con clave key
require -- precondición
not_full: count< capacity
do
... "algoritmo de inserción"
ensure -- postcondiciones
count <= capacity;
item (key) = elemento;
count:= old count + 1;
end - put
```

put	Obligaciones	Beneficios
Cliente	Satisfacer precondición	De la postcondición
Servidor	Satisfacer postcondición	De la precondición

Los invariantes de clase sirven para expresar propiedades globales de las instancias de una clase, mientras que las pre y postcondiciones describen las propiedades de rutinas particulares.

Por ejemplo si se tiene una clase PERSONA se puede establecer como invariante que la edad debe ser mayor que cero y además que si es casada entonces el cónyuge tiene como cónyuge a sí mismo. En Eiffel se vería de la siguiente forma:

```
invariant
  edad_no_negativa: edad >= 0
  matrimonio _ correcto: soltero or else conyuge.conyuge = Current
```

En el invariante con etiqueta *matrimonio_correcto* se establece que, o bien se es soltero o sino el cónyuge tiene como cónyuge a Current (o sea a sí mismo).

```
class STACK [G] feature
capacity: Integer;
count: Integer;
feature {None}
representation: Array[G]
....
invariant -- invariante
0<=count; count <=capacity;
```

```
capacity = representation.capacity;
(count>0) implies representation.item(count)=item
end
```

7.1.1 Condiciones de Corrección de una clase.

Una clase es correcta con respecto a sus aserciones sí y sólo si se cumplen las siguientes condiciones (Meyer,1997):

- Para toda rutina de creación rc: {prerc } dorc { postrc and INV}
- Para toda rutina exportada r: {INV and prer } dor { postr and INV}

Donde prea y posta representan la pre y poscondición respectivamente de una rutina a, INV es el invariante de la clase donde se implementa a y doa representa la ejecución de a.

De aquí se deduce que el rol principal del procedimiento de creación (algunas veces llamado constructor) es establecer el invariante de clase.

Utilidad de las aserciones:

- Escribir software correcto. Nos permiten describir los requisitos exactos para cada rutina y las propiedades globales de cada clase, lo cual implica la construcción de software correcto desde el principio.
- Ayuda a la documentación. Las pre, postcondiciones e invariante proporcionan información precisa a los clientes de los módulos.
- Apoyo para la prueba y depuración: El programador establece como opción del compilador el efecto de las aserciones en tiempo de ejecución.

7.2 Programación por contrato frente a Programación defensiva.

La programación defensiva propugna no escribir rutinas parciales (sólo tienen sentido para argumentos en un subconjunto de su dominio), ya que no hay garantía que el cliente realice la invocación correcta y se pueden obtener resultados erróneos.

La programación defensiva propugna la escritura de rutinas totales que son robustas (comportamiento definido para todas las entradas en el dominio). Bajo este tipo de programación ¿qué hacer si surge un problema?: informar al cliente a través bien de un *Valor de retorno*, bien mediante *Excepciones* para que éste pueda hacer algo o al menos evitar las consecuencias perjudiciales del error.

El Diseño por Contrato establece que "bajo ninguna circunstancia debe el cuerpo de la rutina comprobar el cumplimiento de la precondición".

```
sqrt(x: REAL): REAL
      requiere: x>=0
      efecto: retorna una aproximación a la raíz cuadrada de x
```

La programación por contrato rechaza la "programación defensiva":

```
sqrt(x: REAL): REAL is do
if x<0 then "Manejar error"
else "Calcular raíz"
end;
```

En la programación por contrato se establecen responsabilidades. La rutina establece la precondición y la postcondición

```
sqrt(x: REAL): REAL is
      require x>=0;
      do …
      ensure
      end
```

El código cliente debe asegurar la precondición

```
if (valor >= 0) then sqrt(valor)
```

Las ventajas de la utilización de programación por contrato son muchas:

- Se simplifica la programación.
- La redundancia es perjudicial: software más complejo, más posibilidades de cometer errores.
- Mejora la eficiencia: menos controles.
- Paradoja: "la fiabilidad se mejora comprobando menos".

7.3 Excepciones.

Existen casos en que el uso de aserciones (por avanzado que este sea) no puede dar soporte a todas las situaciones, como por ejemplo:

- Errores de hardware o del SO.
- Detección de errores tan pronto como sea posible aunque no se pueda detectar con una precondición.
- Tolerancia frente a fallos software.

En estos casos parece necesario el uso de técnicas basadas en excepciones.

El mecanismo de excepciones es una solución elegante en el tratamiento de los errores que permite el mantenimiento de un equilibrio entre la corrección (comprobar todos los errores) y la claridad (no desordenar el código del flujo normal con excesivas comprobaciones). Permite separar el funcionamiento correcto de las situaciones de error.

Una vez producida una excepción E en una rutina A, se puede prevenir el fracaso de esta rutina A capturando la excepción y tratando de restaurar un estado a partir del cual el cómputo pueda proseguir.

Si ocurre una excepción E durante la ejecución de una rutina A y esta rutina A no se puede recuperar a dicha excepción E, o no trata dicha excepción E producida, la rutina A fracasará.

Existen varias fuentes de excepciones:

- Violación del contrato: Llamada a una rutina sin cumplir la precondición (excepción recogida por la rutina que llama). Terminación de la rutina sin cumplir la postcondición o el invariante (recogida en la rutina que no cumple la postcondición y/o invariante).
- Condiciones anormales notificadas por el Hardware o el SO.
- Una rutina fracasa, lo cual provocará una excepción en quien la llama.
- Se lanzan explícitamente

7.4 Lenguajes y Programación por contrato.

Eiffel es el único lenguaje de Programación OO que incluye Programación por contrato. Java incluye como mecanismos para la programación por contrato los Asertos y las Excepciones pero B. Meyer no lo considera programación por contrato por los siguientes motivos:

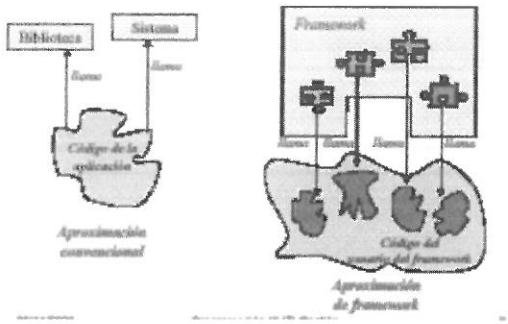
- En Programación por contrato existen distintos tipos de asertos. En java sólo existe assert.
- En la programación por contrato, en tiempo de ejecución se puede seleccionar los tipos de asertos que queremos activar. Por defecto Eiffel activa sólo precondiciones.
- Los asertos no forman parte de la interfaz de los métodos
- javadoc no genera la información relativa a los asertos

C++ incluye mecanismos para el tratamiento de excepciones, y la instrucción **assert** para la depuración de programas.

8 FrameWorks.

Actualmente un alto porcentaje del software que se crea es utilizando Frameworks. Estos FrameWorks son marcos de trabajo dirigidos por eventos, en los que el cliente proporciona una serie de rutinas en respuesta a determinados eventos y personaliza determinados comportamientos.

¿Qué significa esto? Construir software a partir de clases, a partir de bloques, equivalente a un "lego".Pero puede ser más práctico pensar en un "puzzle" donde encajamos las piezas correctas



En la programación OO convencional la aplicación se construye de la siguiente manera:	En la programación OO a través de Frameworks, el FrameWork:
- Conjunto de clases instanciadas por el cliente. - El cliente llama a las funciones. - Flujo de control no predefinido. Es establecido por quien crea la aplicación. - Interacción no predefinida. - No hay comportamiento por defecto.	- Llama a las funciones cliente - Controla el flujo de la ejecución - Define la interacción de objetos - Ofrece un comportamiento por defecto

Ejemplos de Framework:

- OLE (Object Linking and Embedding)
- AWT (Abstract Window Toolkit)

- MFC (Microsoft Foundation Classes)
- DCOM (Distributed Common Object Model)
- CORBA (Common Object Request Broker Arquitecture)
- .NET

9 Lenguajes.

9.1 Características de un LPOO

Las distintas características que podemos identificar en un LPOO, determinan sus posibilidades a la hora de desarrollar aplicaciones, estas son: Tipificación fuerte (comprobación de tipos en tiempo de compilación. Hay una excepción: **Smalltalk**), Ocultación, Compilación Incremental, Genericidad, Paso de mensajes, Herencia, Polimorfismo (basado en la ligadura dinámica), Excepciones y aserciones, Concurrencia, Semántica referencia o almacenamiento y Recolector de basura.

9.2 LPOO Puros vs. Híbridos.

Los lenguajes de POO pueden ser puros e híbridos. Los puros son lenguajes que se han construido con características exclusivamente orientadas a objeto. Los híbridos son aquellos que parten de un lenguaje de programación convencional (estructurada) y funcional a los que se le añaden las características orientadas a objeto.

Las ventajas e inconvenientes de ambos las podemos ver en el siguiente cuadro:

	Puros	*Híbridos*
Ventajas	Más potencia. Flexibilidad para modificar el lenguaje. Más fácil de asimilar (enseñar).	Más rápido. Más fácil el paso de programación estructurada a POO. Implementación de operaciones-algoritmos fundamentales más sencilla.
Inconvenientes	Más lento. Implementación de operaciones-algoritmos fundamentales muy complicada.	Trabaja con 2 paradigmas a la vez. Modificar el lenguaje es difícil.

9.3 Manejo de la memoria dinámica

El manejo de la memoria dinámica en los LPOO es un aspecto muy importante. Este manejo lo pueden hacer de dos maneras distintas:

- Recolección automática. Consiste en que el programador no se ocupa de liberar la memoria cogida de la memoria montón (Heap). El lenguaje de programación se ocupa de detectar cuando no se está utilizando una determinada zona de memoria para marcarla como libre. Este proceso consume recursos del procesador pero libera al programador de esta labor. Ejemplos de estos LPOO son: EIFFEL, JAVA.
- Recolección manual. El programador es el responsable de liberar la memoria montón. Para lo cual los LPOO proporcionan las herramientas adecuadas. Por ejemplo en C++ la instrucción *delete*.

9.4 Lenguajes.

Intentaremos mostrar los distintos LPOO que han aparecido, y describiremos las características más importantes de los más utilizados.

Los primeros lenguajes fueron:

- SIMULA: Se considera que **SIMULA** es el primer Lenguaje de Programación Orientado a Objetos (en adelante LPOO), fue diseñado por **Kristen Nygaard** y **Ole Johan Dhal**, del **Norwegan Computer Center** (NCC). Se concibió para el desarrollo de simulaciones de procesos industriales y científicos, pero llegó a considerarse apto para desarrollar grandes aplicaciones.
- Smalltalk: Más tarde surge el primer LPOO en sentido estricto: **Smalltalk_80**, diseñado por **Alan Kay** y **Adele Goldberg**, de la **XEROX PARC**. Además de ser considerado como primer LPOO es el más puro (sólo tiene objetos, NO ES programación estructurada)

Otros son extensiones de lenguajes convencionales:

- C++ [1986]: Diseñado por **Stroustrup**, es un LPOO híbrido basado en **C** y **Smalltalk**. Ha tenido mucho éxito.
- Objetive C [1986]: Es una extensión de **C**. Hoy está en desuso. Surge para las plataformas **NeXT** y fracasó junto a las máquinas **NeXT**.
- Object Pascal [1987]: Extensión de **Pascal**, lo popularizó **Borland** con **Turbo Pascal 5.5**
- Object COBOL [1992-3]: Extensión de **COBOL**, nuevo LPOO que parece haber tenido aceptación.
- Delphi [1995]: Proviene de **Object Pascal**, creado por **Borland**.

Otros extensiones de Lenguajes Funcionales:

- PROLOG++ ,CLOS (**Common Lisp** orientado a objetos)

Lenguajes fuertemente tipificados:

- Ada_95: **Ada_83** ya era casi un LPOO. **Ada_95** sólo añade herencia y ligadura dinámica.
- Eiffel [1988]: Diseñado por **Beltran Meyer**, está aún en desarrollo.

9.5 Smalltalk_80.

Características:

- Todas las entidades que maneja **Smalltalk** son objetos.
- Todas las clases derivan de una clase base llamada **Object**.
- Herencia simple.
- Usa métodos y paso de mensajes.
- Tiene una tipificación débil: la comprobación de los tipos se hace en tiempo de ejecución.
- Soporta concurrencia, pero pobre.
- Se comercializa con un conjunto de bibliotecas de clases predefinidas, agrupadas en categorías o jerarquías (E/S, etc...)
- Existe una versión, **Smalltalk V** para computadoras **IBM**.

C++ y **CLOS** se han apoyado en características de **Smalltalk**.

9.6 Eiffel.

Es un lenguaje inspirado en **SIMULA** con características añadidas de **Smalltalk** y **Ada**. Se destina a aplicaciones de bases de datos e inteligencia artificial. Por su diseño, es bueno para la ingeniería de software. Sus características son:

- Fuertemente tipificado.
- Genericidad.
- Herencia múltiple.
- Ligadura dinámica y polimorfismo.
- Se ha empezado a implementar concurrencia y persistencia de objetos.
- Proporciona una biblioteca de clases predefinidas: Estructuras de datos, E/S, E/S XWindow.
- La memoria dinámica es gestionada por el entorno y no por el sistema operativo: incorpora recogida automática de basura.
- El entorno tiene editor, y Browser, que muestra las clases y jerarquías.
- El entorno proporciona recompilación automática.
- Programación por contrato.

9.7 C++.

C++ [1986]: Diseñado por **Stroustrup**, es un LPOO híbrido basado en **C** y **Smalltalk**. Este lenguaje tiene un gran éxito comercial, sobre todo tras la incorporación en las ediciones visuales de los compiladores (Visual C++, Builder C++). C++ ha sido uno de los responsables del gran impulso que ha sufrido la tecnología orientada a objetos. Algunas de sus características son:

- Soporte para herencia, incluyendo herencia múltiple.
- Ligadura estática por defecto. Para establecer ligadura dinámica hay declarar las funciones como virtuales.
- Las funciones virtuales puras permiten implementar las características diferidas.
- Comprobación de tipos más estricta que en C, pero aun cuenta con la posibilidad del Casting.
- No cuenta con un mecanismo de gestión automática de memoria (recolección de basura). Incorpora el mecanismo de destructores para la liberación de espacio de objetos.
- Genericidad
- Manejo de excepciones.
- Sobrecarga de operadores
- Incorpora la instrucción assert para la depuración, pero no permite realizar una programación por contrato.

9.8 Java.

Java es un LPOO desarrollado con el fin de conseguir que los programas desarrollados con él pudieran ejecutarse sobre cualquier arquitectura (Sun es la empresa que más esfuerzo dedica a su desarrollo). Es el lenguaje POO que más difusión está teniendo en la actualidad debido a su utilización en Internet, aunque no sea precisamente el LPOO que mejor represente las características orientadas a objeto.

La gran difusión de Java se debe a su utilización en la programación en Internet. Así se consigue su incorporación en los principales navegadores como "maquina virtual" capaz de interpretar pequeños programas hechos en Java y precompilados llamados "Applets Java" (constituyen una de la herramientas más potentes para el desarrollo de páginas web).

Las principales características de Java son:

- Es interpretado.
- No incorpora punteros.
- Tienen recolección automática de memoria.
- Está fuertemente tipificado.
- Todos las clases derivan de una clase base única llamada Object.
- Admite cierto grado de herencia múltiple a través de las interfaces (No todos los autores opinan lo mismo, por ejemplo, Meller)
- Java como lenguaje OO posee muchas críticas de los estudiosos del paradigma orientado a objeto
- Proporciona mecanismo para excepciones y asertos, pero B. Meyer no la considera suficiente para la programación por contrato.

10 Conclusiones.

El auge sufrido por las técnicas basadas en la orientación a objeto durante los años 80 ha traído consigo el desarrollo de multitud de LPOO y técnicas de Análisis y Diseño Orientadas a Objeto.

Este auge sufrido ha sido provocado por que las metodologías estructuradas no han sido capaces de dar soluciones satisfactorias para ofrecer un software de calidad, calidad ésta medida en función de la reusabilidad, extensibilidad y abstracción.

El nuevo enfoque que utiliza el paradigma orientado a objeto (que incluye en los objetos tanto los datos como los procedimientos que los manejan) facilita la construcción de software modular, permite una abstracción problema mucho más objetiva (es reconocible la solución del problema).

Algunos de estos lenguajes han surgido como evolución de lenguajes convencionales (programación estructurada y funcional), otros se han diseñado partiendo prácticamente de cero. Tanto unos como otros incorporan en mayor o menor medida las características propias del paradigma orientado a objeto.

Las características más importantes y que deben poseer todos los LPOO son: Clasificación (Clases de objetos), Ocultación y encapsulación, Herencia, y Polimorfismo. Estas características nos permiten conseguir que los desarrollos

orientados a objetos consigan las máxima de calidad del software: Reusabilidad y extensibilidad.

Pero como siempre ocurre no es el LPOO que mejor cumple con la filosofía de orientación a objeto el que más éxito tiene. Véase el caso de Java, que según autores prestigiosos como Meller no es el mejor ejemplo de paradigma OO, que por su uso en Internet ha alcanzado un gran éxito.