

Apunte I - ISFT Sistemas - POO = Programación Orientada a Objetos

Índice Temático

Conceptos a definir	2
Computadora:	2
Paradigma:	3
Abstracción	3
Ocultamiento de Información:	4
Encapsulación o Encapsulamiento:	4
Objeto	5
Clase	6
Ejemplo del Martillo	7
Diseñando un ejemplo más realista al mundo de la programación	8
TDD (Desarrollo dirigido por pruebas)	10
El algoritmo TDD	10
Tipos de Pruebas	11
Test de Aceptación:	11
Test de Funcionales:	12
Test de Sistema:	12
Test Unitarios	12
Las tres partes del test: AAA	13
Ejemplo TDD:	13
Herencia	20
Ejemplos de uso de la herencia en C++	21
Herencia de clase final:	21
Herencia de clase abstracta:	24
Herencia de interfaz:	28



Conceptos a definir

Computadora:

- Como “[metamedia](#)” ([Marshall McLuhan](#)) de representación y expresión: Es capaz de modificar la comunicación, es un medio que podemos modificar con el lenguaje, la representación concreta la construimos nosotros con observación. Un metamedio es un medio que se puede transformar en otro medio. ([McLuhan, 1994](#))
- Es el proteo de las máquinas ([Seymour Papert](#)): Proteo es el dios de las 1000 caras. Entonces la computadora es un simulador al cual, por ejemplo, le puedo decir que sea un reloj, escribiendo un mensaje (programa) de lo que yo imagino que es un reloj. Es por eso que la computadora es el Proteo de las máquinas, puede tomar muchas formas. ([Papert, 1993](#))

Paradigma:

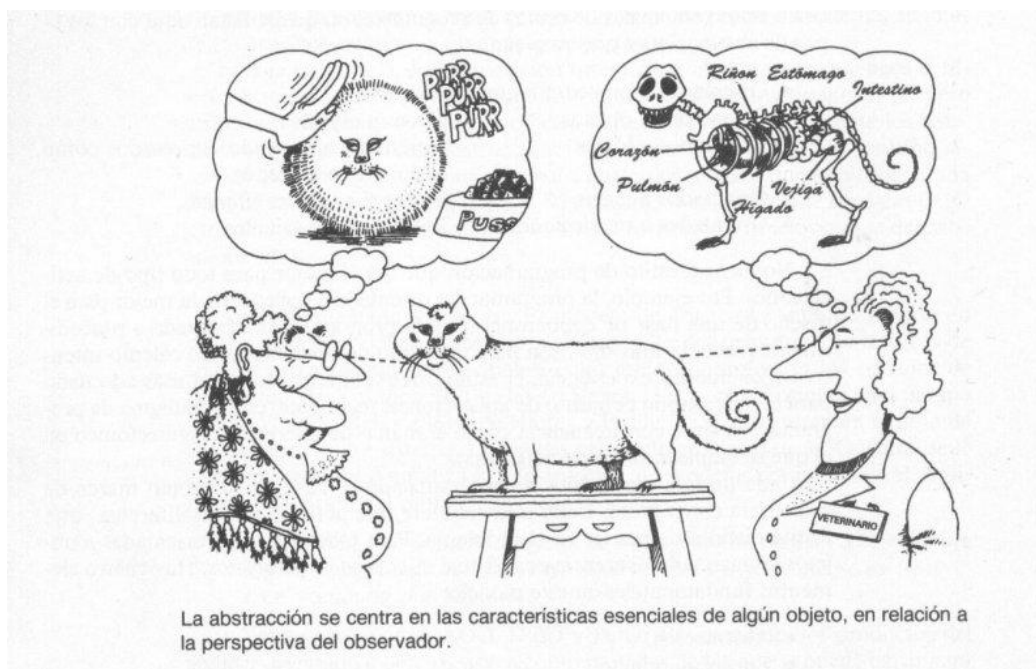
“La forma de pensar acerca de algo”. Es un conjunto de convenciones que utilizamos para comprender el mundo que nos rodea, una visión del mundo, una perspectiva general, una forma de desmenuzar la complejidad. Dos paradigmas de programación de los que escucharon hablar son:

- **Paradigma Funcional:** Si las abstracciones son el resultado de pensar acerca de la función. *(Recuerden que siempre hago referencia a los verbos, es difícil describir la realidad solo con verbos).*
- **Paradigma Orientado a Objetos:** Las abstracciones son el resultado de pensar acerca de la forma, al paradigma se lo llama paradigma basado en clases debido a que las clases definen características comunes a un conjunto de objetos: *POO. (Cuando me refiero a objetos pienso en un sustantivo con sus características). Aclaración: existe paradigma orientado a objetos sin clases (Por ejemplo el lenguaje: [Self](#) en el cual se basó [Javascript](#))*

Abstracción

Cita a [Wulf \(1980\)](#): "Nosotros hemos desarrollado una técnica excepcionalmente potente para tratar con la complejidad. Nos abstraemos de ella, la ignoramos. Incapaces de manejar la totalidad de un objeto complejo, elegimos ignorar aquellos detalles no esenciales, tratando en lugar con un modelo generalizado e idealizado de él..."

Abstracción: La abstracción es la técnica que nos ayuda a identificar que es significativo y que no lo es de un objeto.



Es necesario ahora definir el concepto de objeto y clase entonces:

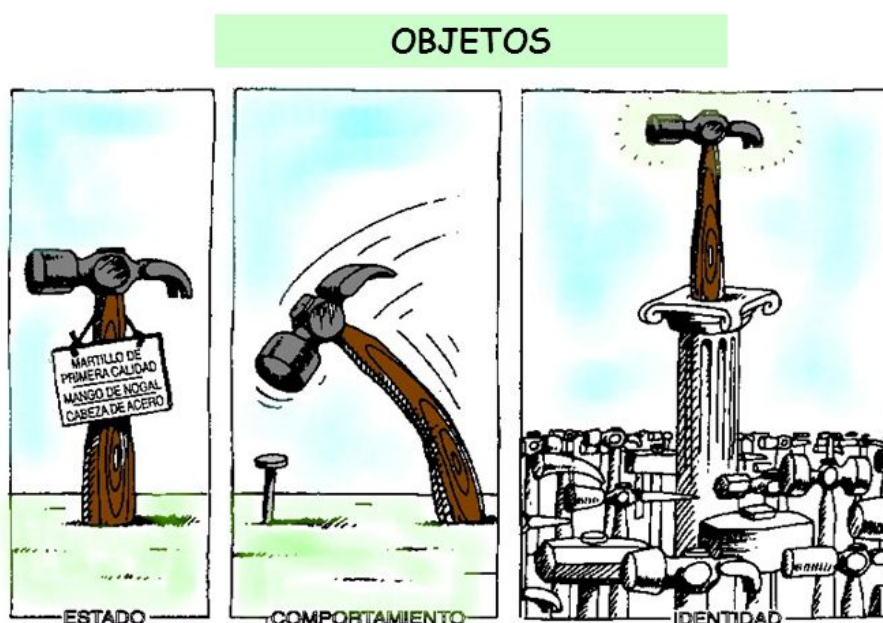
Objeto

Según [Booch \(1996\)](#) un objeto es una entidad tangible que exhibe algún comportamiento bien definido, desde la perspectiva de la cognición humana, un objeto es cualquiera de las siguientes cosas:

- Una cosa tangible y/o visible.
- Algo que puede comprenderse intelectualmente.
- Algo hacia lo que se dirige un pensamiento o acción.
- Modela la realidad y existe en el tiempo y espacio.

Es muy importante considerar que los objetos del mundo real no son el único tipo de objeto de interés en el desarrollo de software. Otros tipos importantes de objetos son invenciones resultantes del proceso de diseño.

Un objeto tiene estado, comportamiento e identidad; la estructura y comportamiento de objetos similares están definidos en su clase común; los términos instancia y objeto son intercambiables:



Un objeto tiene estado, exhibe algún comportamiento bien definido, tiene una identidad única

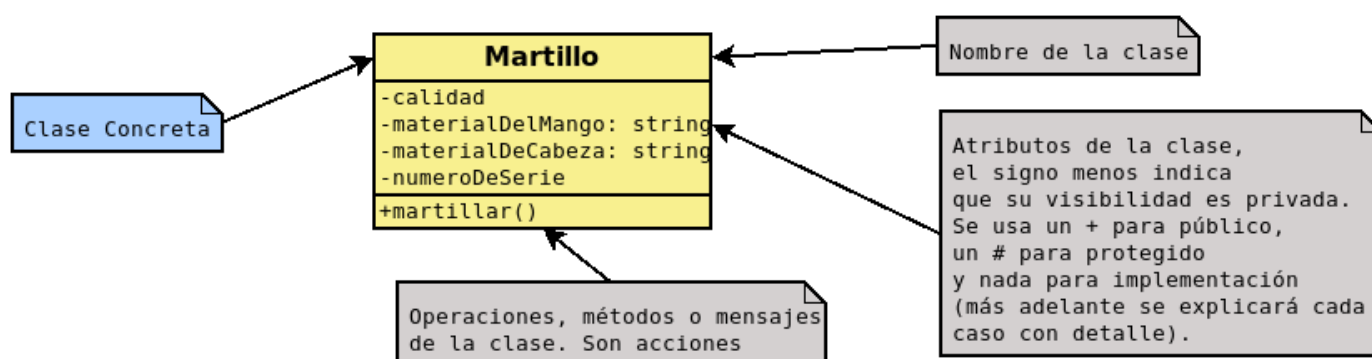
- ★ *El estado de un objeto abarca todas las propiedades (normalmente estáticas) del mismo, más los valores actuales que estas tienen (normalmente dinámicos) de cada una de esas propiedades.*
 - ★ *El comportamiento es cómo actúa y reacciona un objeto, en términos de sus cambios de estado y paso de mensajes.*
 - ★ *El estado de un objeto representa los resultados acumulados de su comportamiento.*
 - ★ *La identidad es aquella propiedad de un objeto que lo distingue de los demás objetos.*
- Khosahfian y Copeland (1986)*

Clase

[Booch \(1996\)](#) define que una clase es un **conjunto de objetos que comparten una estructura común y un comportamiento común**.

- ★ *Un solo objeto no es más que una instancia de una clase.*
- ★ *Un objeto no es una clase.*

El primer ejemplo con el cual ustedes van a trabajar es la clase y en lenguaje de modelado: UML ([wikipedia](#)). Este lenguaje unificado de modelado es un lenguaje gráfico para visualizar, especificar, construir y documentar un sistema. Vamos a empezar a ver el primer diagrama que nos va a permitir ver el concepto de clase, para esto vamos a usar como ejemplo el martillo de Grady Booch con su estado, comportamiento e identidad:



Ejemplo del Martillo

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

class Martillo
{
private:
    string calidad;
    string materialDelMango;
    string materialDeCabeza;
    int numeroDeSerie;

public:
    //método constructor: se invoca al crear una instancia u objeto
    //de esta clase
    Martillo(string c, string mDM, string mDC, int nDS)
    {
        cout << "Me construyo..." << endl;
        calidad = c;
        materialDelMango = mDM;
        materialDeCabeza = mDC;
        numeroDeSerie = nDS;
    }
    //método destructor: se invoca al destruir una instancia u objeto
    //de esta clase
    ~Martillo()
    {
        cout << "Me destruyo..." << endl;
    }
    void martillar()
    {
        cout << "Mi calidad es de: " << calidad << endl;
        cout << "Mi mango es de: " << materialDelMango << endl;
        cout << "Mi cabeza es de: " << materialDeCabeza << endl;
        cout << "Mi numero de serie es: " << numeroDeSerie << endl;
        cout << "Estoy martillando" << endl;
    }
};

int main()
{
    //Creación del objeto martillo,
    //se invoca al constructor con parámetros de la clase Martillo:
    Martillo objetoMartillo = Martillo("primera", "nogal", "acero", 12345);

    //Llamar a la operación (método o mensaje) del objeto martillo
    objetoMartillo.martillar();

    return 0;
}
```

Complementando los conceptos de clase y objeto ([wikipedia](#)):

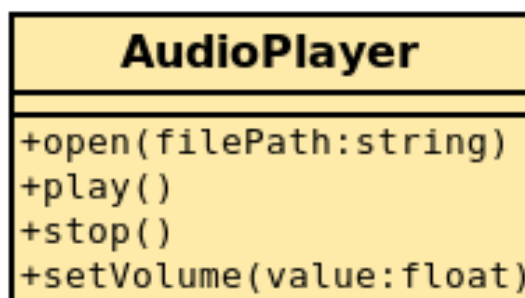
- ★ *Una clase es una especie de "plantilla" en la que se definen los atributos y métodos predeterminados de un tipo de objeto. Esta plantilla se crea para poder crear objetos fácilmente. Al método de crear nuevos objetos mediante la lectura y recuperación de los atributos y métodos de una clase se le conoce como instanciación.*
- ★ *Un objeto es una instancia de una clase. Entidad provista de un conjunto de propiedades o atributos (datos) y de comportamiento o funcionalidad (métodos), los mismos que consecuentemente reaccionan a eventos. Se corresponden con los objetos reales del mundo que nos rodea, o con objetos internos del sistema (del programa).*
- ★ **Método:** *Algoritmo asociado a un objeto (o a una clase de objetos), cuya ejecución se desencadena tras la recepción de un "mensaje". Desde el punto de vista del comportamiento, es lo que el objeto puede hacer. Un método puede producir un cambio en las propiedades del objeto, o la generación de un "evento" con un nuevo mensaje para otro objeto del sistema.*
- ★ **Atributo:** *Características que tiene la clase y por consiguiente la instancia de la clase u objeto.*
- ★ **Miembros del Objeto:** *Atributos, identidad, relaciones y métodos.*

Diseñando un ejemplo más realista al mundo de la programación

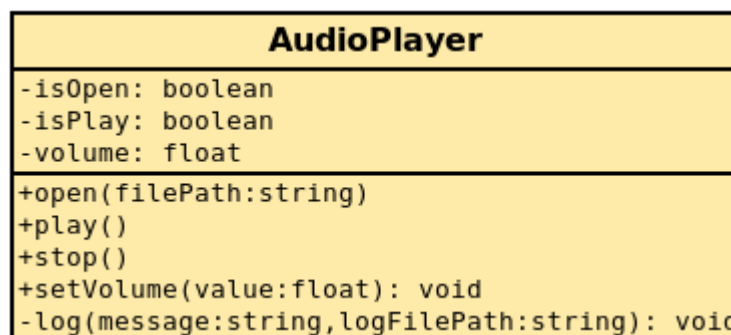
Vamos a usar el software de diseño de diagramas UML, DIA, pueden descargarlo de:

<http://dia-installer.de/download/index.html.en>

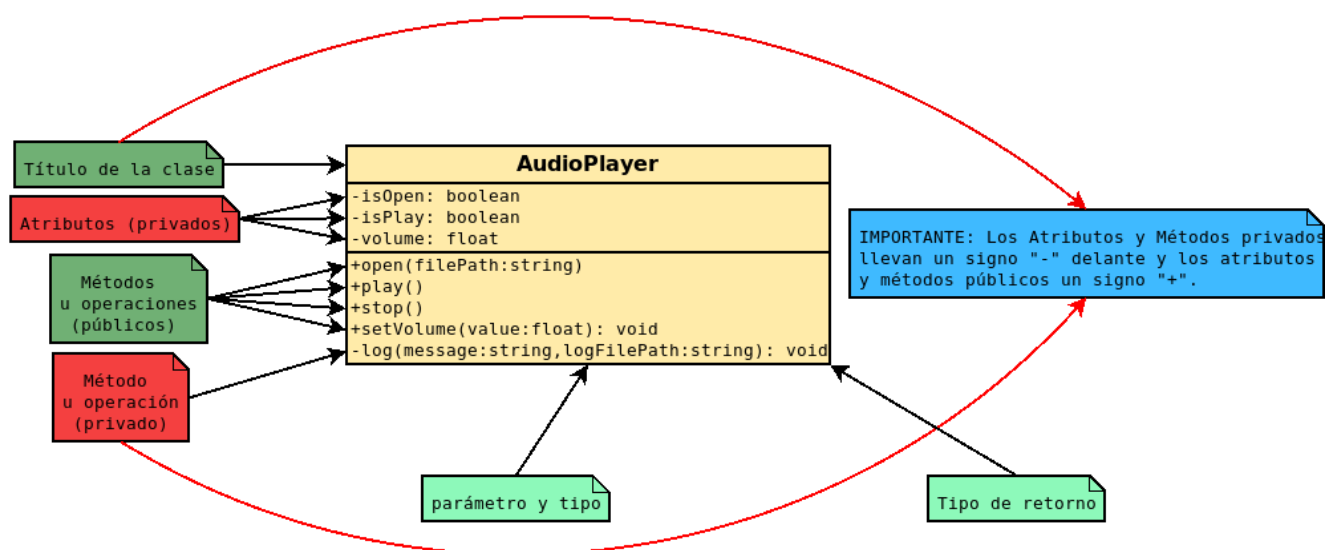
Un ejemplo de aplicación “relativamente simple” y real en el mundo del software puede ser un reproductor de archivos de audio vamos a comenzar a esbozar una posible y mínima clase en UML de ese reproductor:



En el anterior diagrama solo se puede ver el título de la clase y el comportamiento (métodos) que se definirán en esa clase. Normalmente con este tipo de diseño (solo mostrando comportamiento) se puede trabajar, dejando la responsabilidad de la implementación del ámbito privado al desarrollador. Pero también puede darse el caso de un diseño completo con todo lo que sucede en el ámbito privado, por lo tanto con fines didáctico vamos a escribir ese modelo de clase un UML:



Vamos a explicar nuevamente cada parte de este diagrama UML de clase:



Se agregaron tres atributos privados:

- -isOpen: boolean
- -isPlay: boolean
- -volume: float

Y un método privado:

- log(message:string, logFilePath: string): void

Estos no tienen acceso desde la instancia/objeto de esta clase, pero serán usados de forma interna. Todo esto es una estimación y podemos equivocarnos, existe una técnica muy poderosa para mejorar el porcentaje de estimación de la funcionalidad de una entidad u objeto y no depender tanto de la estimación y mejorar la precisión de los requerimientos de un software. La técnica se llama:

TDD (Desarrollo dirigido por pruebas)

TDD según [Blé Jurado \(2010\)](#) es una técnica para diseñar software que se centra en tres pilares fundamentales:

- La implementación de las funciones justas que el cliente necesita y no más.
- La minimización del número de defectos que llegan al software en fase de producción.
- La producción de software modular, altamente reutilizable y preparado para el cambio.

No se trata de escribir pruebas a granel como locos, sino de diseñar adecuadamente según los requisitos.

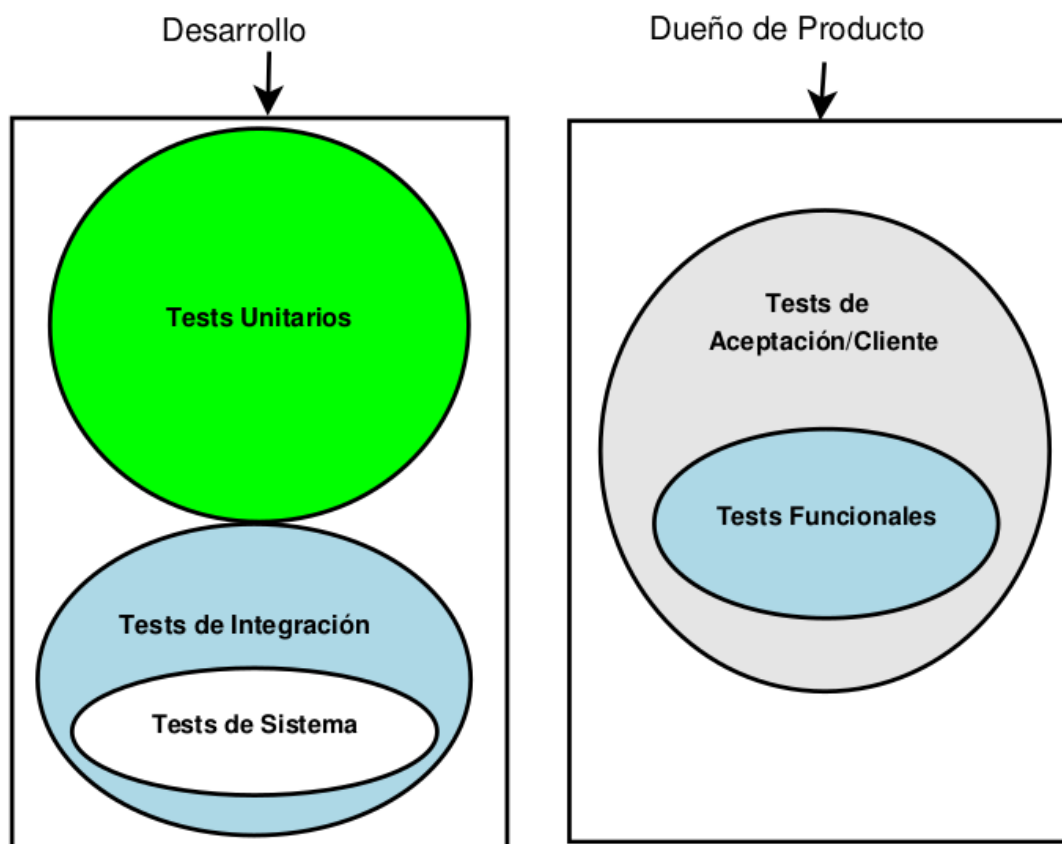
El algoritmo TDD

La esencia de TDD es sencilla pero ponerla en práctica correctamente es cuestión de entrenamiento, como tantas otras cosas. El algoritmo TDD sólo tiene tres pasos:

- **Escribir la especificación del requisito (el ejemplo, el test):** *Una vez que tenemos claro cuál es el requisito, lo expresamos en forma de código.*
- **Implementar el código según dicho ejemplo:** *Teniendo el ejemplo escrito, codificamos lo mínimo necesario para que se cumpla, para que el test pase. Típicamente, el mínimo código es el de menor número de caracteres porque mínimo quiere decir el que menos tiempo nos llevó escribirlo. No importa que el código parezca feo o chapucero, eso lo vamos a enmendar en el siguiente paso y en las siguientes iteraciones.*
- **Refactorizar para eliminar duplicidad y hacer mejoras:** *Refactorizar no significa reescribir el código; reescribir es más general que refactorizar. Según [Martín Fowler\(2018\)](#), refactorizar 10 es modificar el diseño sin alterar su comportamiento.*

Tipos de Pruebas

Hay varios tipos de tests, vamos a definir la terminología usada para los más utilizados y elegiremos para este ejemplo un solo tipo. “Desde el aspecto potestad, es decir, mirando los tests según a quién le pertenecen, distinguimos entre tests escritos por desarrolladores y tests escritos por el dueño del producto. Recordemos que el dueño del producto es el analista de negocio o bien el propio cliente. Lo ideal es que el analista de negocio ayude al cliente a escribir los tests para asegurarse de que las afirmaciones están totalmente libres de ambigüedad”. ([Blé Jurado, 2010](#))



Test de Aceptación:

Es un test que permite comprobar que el software cumple con un requisito de negocio. Un test de aceptación es un ejemplo escrito con el lenguaje del cliente pero que puede ser ejecutado por la máquina. Por ejemplo:

- El producto X con precio \$5000 tiene un precio final de \$5500 después de aplicar el impuesto Z.
- Si el paciente nació el 1 de junio de 1981, su edad es de 28 años en agosto de 2009.

Test de Funcionales:

Todos los tests son en realidad funcionales, puesto que todos ejercitan, en el nivel más elemental, un método u operación de una clase. No obstante, cuando se habla del aspecto funcional, se distingue entre test funcional y test no funcional. Un test funcional es un subconjunto de los tests de aceptación. Es decir, comprueban alguna funcionalidad con valor de negocio. Los tests de aceptación tienen un ámbito mayor porque hay requerimientos de negocio que hablan de tiempos de respuesta, capacidad de carga de la aplicación, etc; cuestiones que van más allá de la funcionalidad. Un test funcional es un test de aceptación pero, uno de aceptación, no tiene por qué ser funcional. ([Blé Jurado, 2010](#))

Test de Sistema:

Es el mayor de los tests de integración, ya que integra varias partes del sistema. Se trata de un test que puede ir, incluso, de extremo a extremo de la aplicación o del sistema. Se habla de sistema porque es un término más general que aplicación, pero no se refiere a administración de sistemas, no es que estemos probando el servidor web o el servidor de correo aunque, tales servicios, podrían ser una parte de nuestro sistema. Así pues, un test del sistema se ejercita tal cual lo haría el usuario humano, usando los mismos puntos de entrada (aquí sí es la interfaz gráfica) y llegando a modificar la base de datos o lo que hay en el otro extremo. ¿Cómo se puede automatizar el uso de la interfaz de usuario y validar que funciona? Hay software que permite hacerlo. Por ejemplo, si la interfaz de usuario es web, el plugin Selenium 6 para los navegadores web nos permite registrar nuestra actividad en una página web como si estuviéramos grabando un vídeo para luego reproducir la secuencia automáticamente y detectar cambios en la respuesta del sitio web. ([Blé Jurado, 2010](#))

Test Unitarios

Estos son los tests que más nos importan ahora. Son los tests más importantes para el practicante TDD, los ineludibles. Cada test unitario o test unidad (unit test en inglés) es un paso que andamos en el camino de la implementación del software. Todo test unitario debe ser:

- **Atómico:** significa que el test prueba la mínima cantidad de funcionalidad posible. Esto es, probará un solo comportamiento de un método de una clase. El mismo método puede presentar distintas respuestas ante distintas entradas o distinto contexto. El test unitario se ocupará exclusivamente de uno de esos comportamientos, es decir, de un único camino de ejecución.

- **Independiente:** significa que un test no puede depender de otros para producir un resultado satisfactorio. No puede ser parte de una secuencia de tests que se deba ejecutar en un determinado orden. Debe funcionar siempre igual independientemente de que se ejecuten otros tests o no.
- **Inocuo:** significa que no altera el estado del sistema. Al ejecutarlo una vez, produce exactamente el mismo resultado que al ejecutarlo veinte veces. No altera la base de datos, ni envía emails ni crea ficheros, ni los borra. Es como si no se hubiera ejecutado.
- **Rápido:** porque ejecutamos un gran número de tests cada pocos minutos y se ha demostrado que tener que esperar unos cuantos segundos cada rato, resulta muy improductivo. Un sólo test tendría que ejecutarse en una pequeña fracción de segundo. ([Blé Jurado, 2010](#))

A partir de aquí vamos a ver la última característica muy importante que debemos conocer para escribir test unitarios de forma correcta:

Las tres partes del test: AAA

Un test tiene tres partes, que se identifican con las siglas AAA en inglés: Arrange (Preparar), Act (Actuar), Assert (Afirmar). ([Blé Jurado, 2010](#))

- **Arrange (Preparar):** Una parte de la preparación puede estar contenida en un método o función setUp, por ejemplo, si es común a todos los tests de la clase. Si la etapa de preparación es común a varios tests de la clase pero no a todos, entonces podemos definir otro método o función en la misma clase, que aúne tal código. No le pondremos la etiqueta de [TEST] sino que lo invocaremos desde cada punto en que lo necesitemos.
- **Act (Actuar):** El acto consiste en hacer la llamada al código que queremos probar.
- **Assert (Afirmar):** La afirmación o afirmaciones se hacen sobre el resultado de la ejecución, bien mediante **validación del estado** o bien mediante validación de la interacción. Se afirma que nuestras expectativas sobre el resultado se cumplen. Si no se cumplen el framework marcará en rojo cada falsa expectativa.

Ejemplo TDD:

Vamos a aplicar TDD (Desarrollo Dirigido por Pruebas) de un modo simple para ver cada paso de esta técnica e implementar una simulación de la funcionalidad del reproductor de audio. De este modo puede que aparezcan otras operaciones orientadas por los tests.

Aplicando las tres A de los test unitarios. Para esto vamos a usar el marco (framework) de pruebas unitarias para C++, Catch2 (<https://github.com/catchorg/Catch2>) y el compilador online de <https://godbolt.org/> , esto nos permitirá correr las pruebas de un modo simple y generar un primer prototipo del reproductor de audio que cumpla con las pruebas mínimas, empecemos:

Lo primero que vamos a hacer es un test para el método open para abrir archivos de audio, recordemos nuestra clase mínima de audio player:

AudioPlayer
-isOpen: boolean -isPlay: boolean -volume: float
+open(filePath:string) +play() +stop() +setVolume(value:float): void -log(message:string,logFilePath:string): void

Ahora escribamos un test usando las características del framework [Catch2](#) para el método open:

```
#include <catch2/catch_test_macros.hpp>
#include <iostream>
using namespace std;

TEST_CASE( "Correct opening of audio files", "[open method]" )
{
    // Arrange
    string filePath = "./resources/orchestral.ogg";
    AudioPlayer player;

    // Act
    player.open(filePath);

    // Assert
    REQUIRE( player.getOpenStatus() == true );
}
```

Es importante identificar las tres partes del test unitario: Arrange (Preparar), Act (Actuar), Assert (Afirmar).

Si lo hacemos en el compilador online <https://godbolt.org/> rápidamente nos va a indicar que no declaramos una clase **AudioPlayer** y por consiguiente tampoco podemos implementar una instancia u objeto **player** y menos aún llamar a una operación o método **open**:


```
1 #include <catch2/catch_test_macros.hpp>
2 #include <iostream>
3 using namespace std;
4
5 TEST_CASE( "Correct opening of audio files", "[open method]" )
6 {
7     // Arrange
8     string filePath = "./resources/orchestral.ogg";
9     AudioPlayer player;
10
11     // Act
12     player.open(filePath);
13
14     // Assert
15     REQUIRE( player.getOpenStatus() == true );
16 }
```

```
Could not execute the program
Compiler returned: 1
Compiler stderr
<source>: In function 'void CATCH2_INTERNAL_TEST_0()':
<source>:9:5: error: 'AudioPlayer' was not declared in this scope
9 | AudioPlayer player;
  | ~~~~~
<source>:12:5: error: 'player' was not declared in this scope
12 | player.open(filePath);
   | ~~~~~
```

¡Ya empezamos a hacer TDD entonces, la prueba dirigiendo el desarrollo, vamos a hacer esa clase y método para conformar al test unitario!

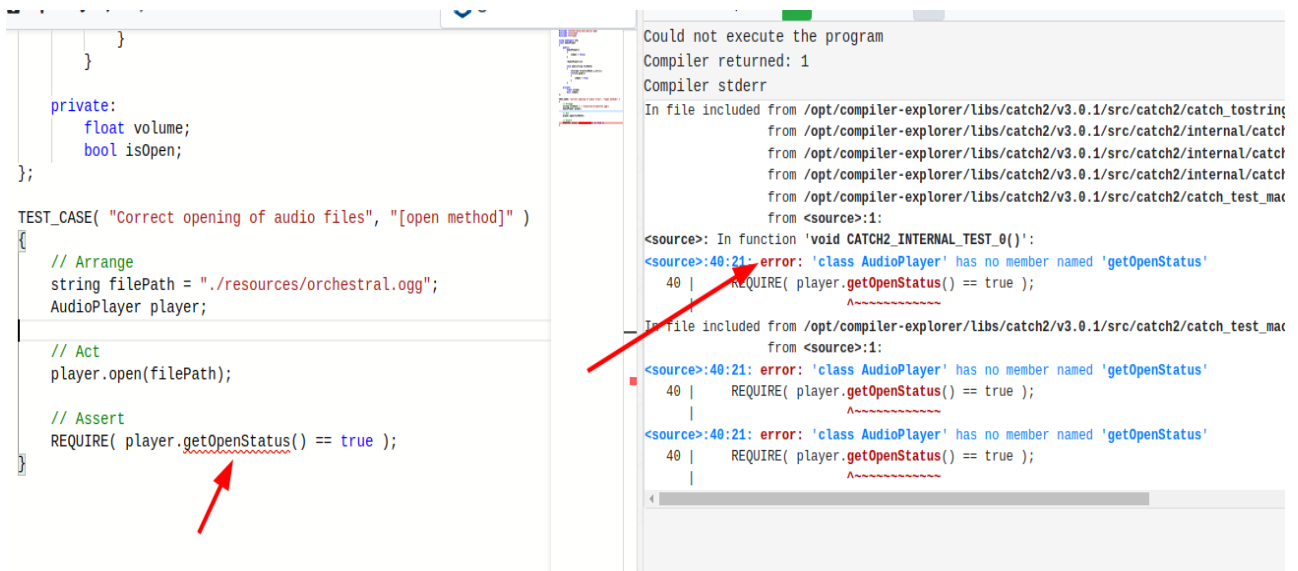
```
#include <catch2/catch_test_macros.hpp>
#include <iostream>
#include <fstream>
using namespace std;
class AudioPlayer
{
public:
    AudioPlayer()
    {
        isOpen = false;
    }
    ~AudioPlayer(){}
    void open(string filePath)
    {
        //@ToDo: por ahora simulamos la apertura correcta
        isOpen = true;
    }
private:
    float volume;
    bool isOpen;
};
```

```
TEST_CASE( "Correct opening of audio files", "[AudioPlayer]" )
{
    // Arrange
    string filePath = "../resources/orchestral.ogg";
    AudioPlayer player;

    // Act
    player.open(filePath);

    // Assert
    REQUIRE( player.getOpenStatus() == true );
}
```

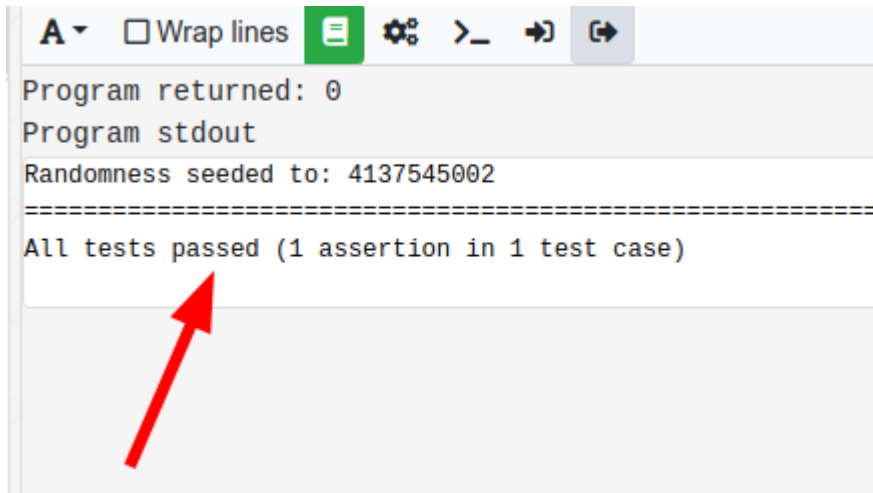
Y aún falla el test, porque en el mismo dimos por entendido que existía una operación para obtener el estado de apertura del archivo de audio: `getOpenStatus()`



Vamos implementar el método `getOpenStatus` entonces:

```
class AudioPlayer
{
public:
    ...
    bool getOpenStatus()
    {
        return isOpen;
    }
private:
    float volume;
    bool isOpen;
};
```

¡Ahora sí!



```

A ▾ □ Wrap lines [icon] [icon] >_ ➡ [icon]
Program returned: 0
Program stdout
Randomness seeded to: 4137545002
=====
All tests passed (1 assertion in 1 test case)

```

*Es importante aclarar que la conformación del test es mínima y no cumple con la funcionalidad de apertura de un archivo de audio, más adelante vamos a resolver eso. Seguiremos con los otros métodos. El método open no está abriendo ningún archivo de audio solo asigna true al estado interno representado por el atributo booleano **isOpen**.*

```

void open(string filePath)
{
    isOpen = true;
}

```

Link para probar el primer test: <https://godbolt.org/z/6sa7xnqf8>

Ahora hagamos otro test para probar el método play:

```

TEST_CASE( "Correct playing of audio files", "[AudioPlayer]" )
{
    // Arrange
    string filePath = "./resources/orchestral.ogg";
    AudioPlayer player;
    player.open(filePath);

    // Act
    player.play();

    // Assert
    REQUIRE( player.getPlaybackStatus() == true );
}

```

Otra vez el mismo error y procedimiento:

```
<source>: In function 'void CATCH2_INTERNAL_TEST_2()':  
<source>:46:12: error: 'class AudioPlayer' has no member named 'play'  
46 |     player.play();  
    |             ^~~~~  
In file included from /opt/compiler-explorer/libs/catch2/v3.0.1/src/catch2/catch_tostring:  
from /opt/compiler-explorer/libs/catch2/v3.0.1/src/catch2/internal/catch_  
from /opt/compiler-explorer/libs/catch2/v3.0.1/src/catch2/internal/catch_  
from /opt/compiler-explorer/libs/catch2/v3.0.1/src/catch2/internal/catch_  
from /opt/compiler-explorer/libs/catch2/v3.0.1/src/catch2/catch_test_ma  
from <source>:1:  
<source>:49:21: error: 'class AudioPlayer' has no member named 'getPlayStatus'; did you i  
49 |     REQUIRE( player.getPlayStatus() == true );  
    |                   ^~~~~~  
In file included from /opt/compiler-explorer/libs/catch2/v3.0.1/src/catch2/catch_test_ma  
from <source>:1:  
<source>:49:21: error: 'class AudioPlayer' has no member named 'getPlayStatus'; did you i  
49 |     REQUIRE( player.getPlayStatus() == true );
```

Debemos crear los métodos que solicita el error, en la clase:

```
class AudioPlayer  
{  
public:  
    AudioPlayer()  
    {  
        isOpen = false;  
        isPlaying = false;  
    }  
    ~AudioPlayer(){}  
    void open(string filePath)  
    {  
        //@@ToDo: por ahora simulamos la apertura correcta  
        isOpen = true;  
    }  
    void play()  
    {  
        //@@ToDo: por ahora simulamos la reproducción correcta  
        if(isOpen) isPlaying = true;  
    }  
  
    bool getPlaybackStatus()  
    {  
        return isPlaying;  
    }  
private:  
    float volume;  
    bool isOpen;  
    bool isPlaying;  
};
```

```
Program returned: 0
Program stdout
Randomness seeded to: 4068727886
=====
All tests passed (2 assertions in 2 test cases)
```

Enlace al código del test:

<https://godbolt.org/z/5Evr883jn>

Así deberíamos continuar con todos los tests hasta llegar conformar todas las funcionalidades que se requieran de un reproductor de audio y volver a correrlos y refactorizarlos en cada implementación de una nueva funcionalidad. Lo mejor sería hacerlo en nuestro entorno de desarrollo en nuestro sistema operativo para no tener las limitaciones del entorno online.

Hasta ahora nuestra clase de `AudioPlayer` quedó así:

AudioPlayer
-isOpen: boolean -isPlay: boolean -volume: float
+open(filePath:string) +play() +stop() +getOpenStatus(): boolean +getPlaybackStatus(): boolean +setVolume(value:float): void +getVolume(): float -log(message:string,logFilePath:string): void

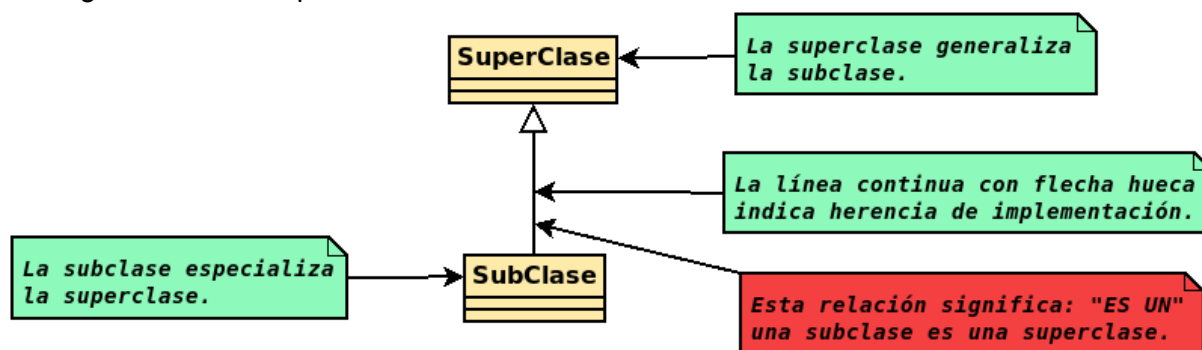
Vamos a llevar a este simple diseño a un estadio de mayor reuso, para eso debemos empezar por entender el concepto de herencia. Y también entender que refactorizar es un proceso circular donde: puedo diseñar, escribir pruebas, escribir el código que conforme esas pruebas y luego volver a tener que diseñar, escribir más pruebas y refactorizar pruebas, y volver a escribir código que conforme esas pruebas, todo esto por n veces.

Herencia

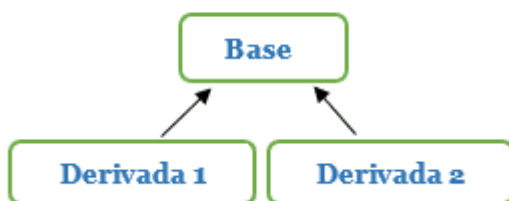
Herencia es cuando una clase se basa en otra clase, usando la misma implementación o comportamiento. Es el mecanismo por el cual una clase permite heredar las características (atributos y métodos) de otra clase.

Según [Booch \(1996\)](#) la herencia es una relación entre clases en la que una clase comparte la estructura y/o el comportamiento definidos en una (herencia simple) o más clases (herencia múltiple). La clase de la que otras heredan se denomina superclase.

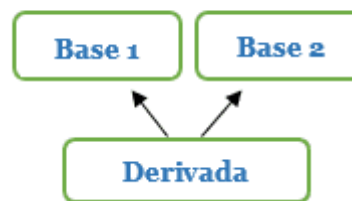
Análogamente, clase que hereda de otra o más clases se denomina subclase.



Herencia simple



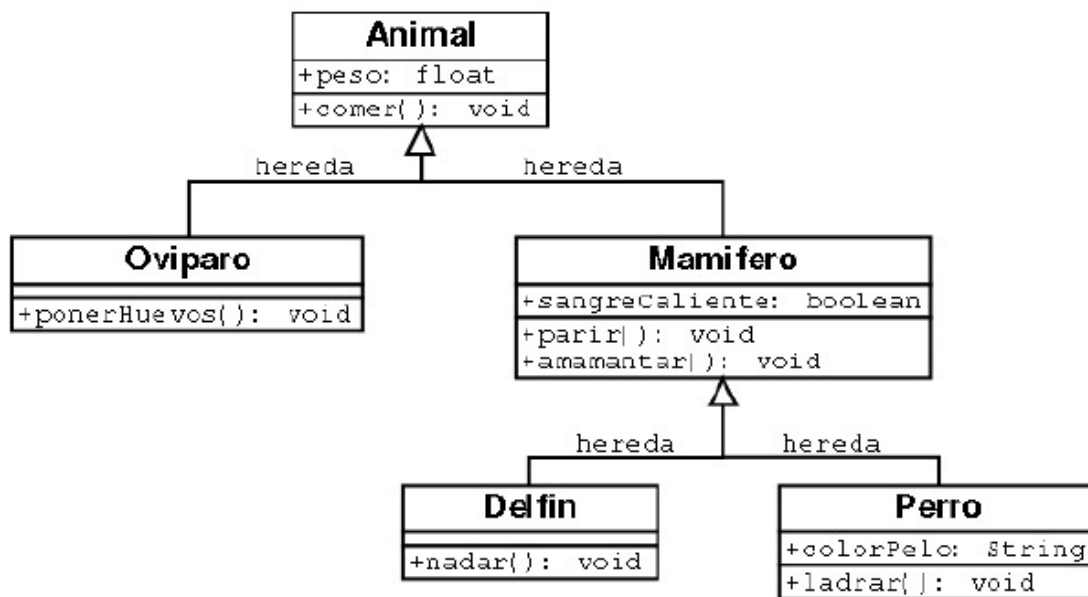
Herencia múltiple



*Se debe tener mucho cuidado con el uso de herencia múltiple, sólo es absolutamente seguro usarla con el uso de interfaces como veremos más adelante, pueden encontrar muchos artículos mostrando sus problemas cuando se usa incorrectamente, algunos lenguajes no dan soporte a la herencia múltiple sino es con interfaces y para ese tipo de herencia usan la palabra reservada **implements***



Es muy importante entender el significado de la relación “es un” en la herencia:



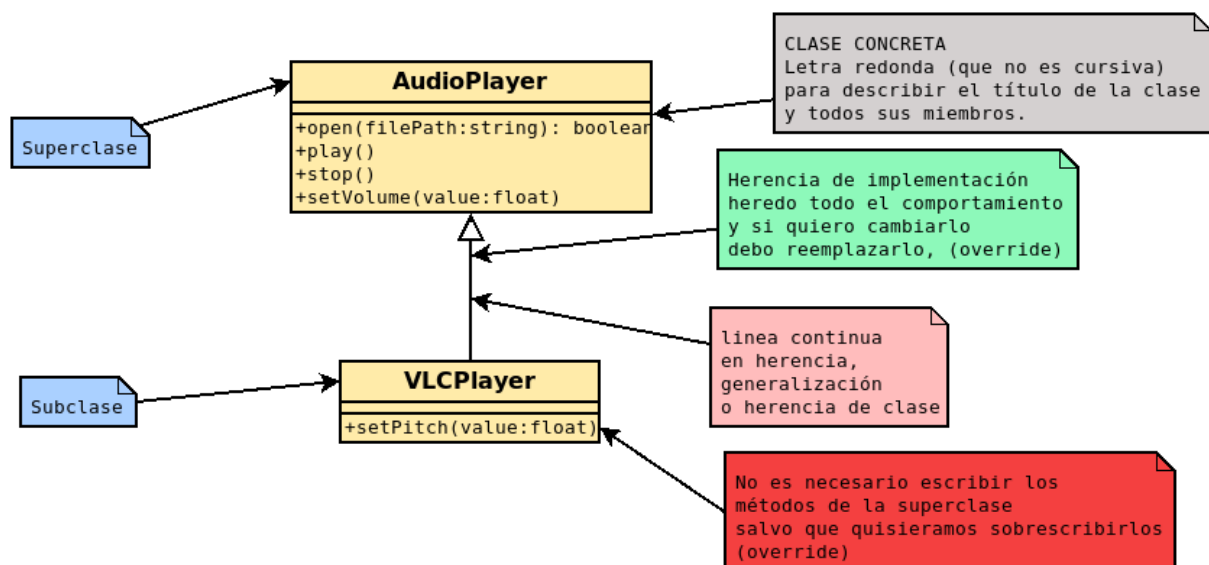
Pero cómo veremos más adelante esa relación “es un” tiene un problema y si bien siempre la vamos a respetar podemos usarla de otro a nuestro favor con otro método.

Ejemplos de uso de la herencia en C++

Volvamos a nuestro ejemplo de reproductor de audio (simplificado), para ver tres casos de herencia que C++ soporta, pero no por eso siempre es correcto utilizarlas:

Herencia de clase final:

Hereda toda la implementación de la superclase



```
#include <iostream>
#include <fstream>
using namespace std;
class AudioPlayer
{
public:
    AudioPlayer()
    {
        cout << "The AudioPlayer constructor was invoked." << endl;
        isOpen = false;
        isPlaying = false;
        setVolume(10);
    }
    ~AudioPlayer()
    {
        cout << "The AudioPlayer destructor was invoked." << endl;
    }
    void open(string filePath)
    {
        //@ToDo: por ahora simulamos la apertura correcta
        isOpen = true;
        cout << "The audiofile: " << filePath << " is open." << endl;
    }
    void play()
    {
        //@ToDo: por ahora simulamos la reproducción correcta
        if(isOpen) isPlaying = true;
        cout << "The audiofile is playing." << endl;
    }
    void stop()
    {
        //@ToDo: por ahora simulamos la detención correcta
        if(isPlaying) isPlaying = false;
        cout << "The audiofile is stopped." << endl;
    }
    void setVolume(float value)
    {
        volume = value;
        cout << "The volume value is: " << volume << endl;
    }
private:
    bool isOpen;
    bool isPlaying;
    float volume;
};
```

```
class VLC : public AudioPlayer
{
    public:
        VLC ()
        {
            cout << "The VLC constructor was invoked." << endl;
            setVolume(10);
            setPitch(0);
        }
        ~VLC ()
        {
            cout << "The VLC destructor was invoked." << endl;
        }
        void setPitch(float value)
        {
            pitch = value;
            cout << "The pitch value is: " << pitch << endl;
        }

        float pitch;
};

int main()
{
    cout << endl;
    AudioPlayer player;
    player.open("./resources/orchestral.ogg");
    player.play();
    player.setVolume(4);

    cout << endl;

    VLC vlcPlayer;
    vlcPlayer.open("./resources/orchestral.ogg");
    vlcPlayer.play();
    vlcPlayer.setVolume(13);

    cout << endl;
    return EXIT_SUCCESS;
}
```

Ejecución:

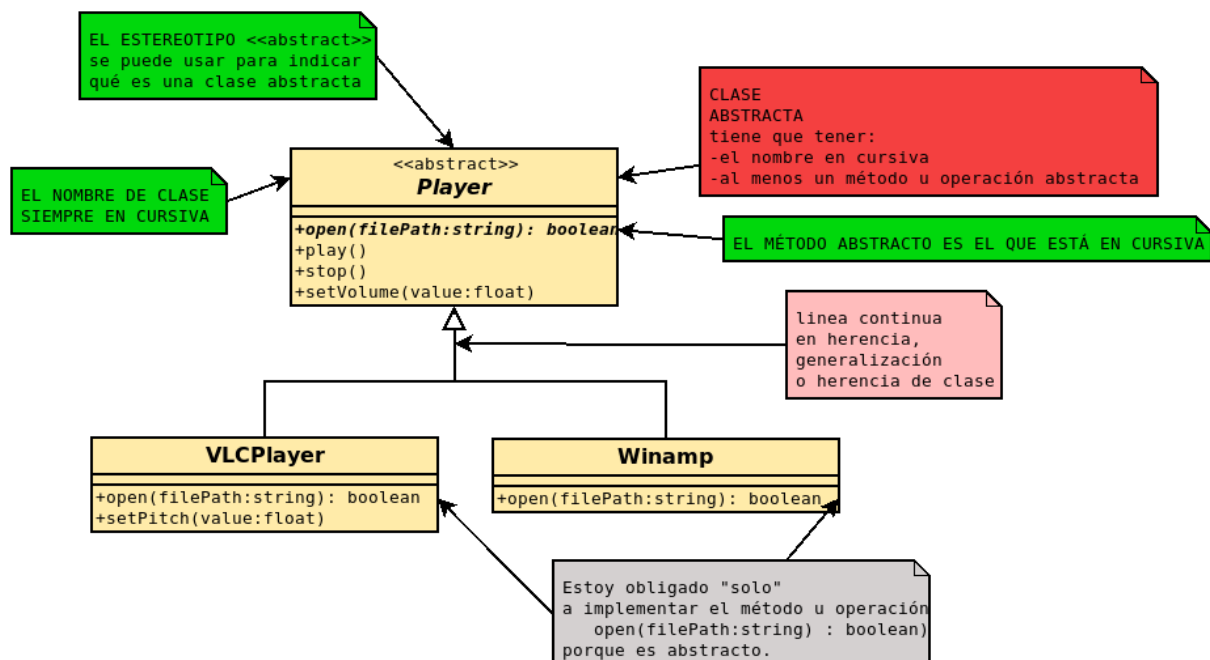
```
The AudioPlayer constructor was invoked.  
The volume value is: 10  
The audiofile: ./resources/orchestral.ogg is open.  
The audiofile is playing.  
The volume value is: 4
```

```
The AudioPlayer constructor was invoked.  
The volume value is: 10  
The VLC constructor was invoked.  
The volume value is: 10  
The pitch value is: 0  
The audiofile: ./resources/orchestral.ogg is open.  
The audiofile is playing.  
The volume value is: 13
```

The VLC destructor was invoked.

Herencia de clase abstracta:

Hereda parte de la implementación de la superclase



```
#include <iostream>
#include <fstream>
using namespace std;
//Player abstract class
class Player
{
    public:
        Player()
        {
            cout << "The Player constructor was invoked." << endl;
            isOpen = false;
            isPlaying = false;
            setVolume(10);
        }
        ~Player()
        {
            cout << "The Player destructor was invoked." << endl;
        }
        //método abstracto o virtual puro:
        virtual void open(string filePath) = 0;
        void play()
        {
            //@ToDo: por ahora simulamos la reproducción correcta
            if(isOpen) isPlaying = true;
            cout << "The audiofile is playing." << endl;
        }
        void stop()
        {
            //@ToDo: por ahora simulamos la detención correcta
            if(isPlaying) isPlaying = false;
            cout << "The audiofile is stopped." << endl;
        }
        void setVolume(float value)
        {
            volume = value;
            cout << "The volume value is: " << volume << endl;
        }
    protected:
        bool isOpen; //@Protegido para poder ser modificado por la subclase
    private:
        bool isPlaying;
        float volume;
};

//VLC concrete class
```

```
class VLC : public Player
{
    public:
        VLC()
        {
            cout << "The VLC constructor was invoked." << endl;
        }
        ~VLC()
        {
            cout << "The VLC destructor was invoked." << endl;
        }
        void open(string filePath)
        {
            //@ToDo: por ahora simulamos la apertura correcta
            isOpen = true;
            cout << "The audiofile: " << filePath << " is open." << endl;
        }
        void setPitch(float value)
        {
            pitch = value;
            cout << "The pitch value is: " << pitch << endl;
        }
    private:
        float pitch;
};

//Winamp concrete class
class Winamp : public Player
{
    public:
        Winamp()
        {
            cout << "The Winamp constructor was invoked." << endl;
        }
        ~Winamp()
        {
            cout << "The Winamp destructor was invoked." << endl;
        }
        void open(string filePath)
        {
            //@ToDo: por ahora simulamos la apertura correcta
            isOpen = true;
            cout << "The audiofile: " << filePath << " is open." << endl;
        }
};
```



```
int main()
{
    //NO SE PUEDE CREAR UN OBJETO O INSTANCIA DE UNA CLASE ABSTRACTA:
    //Player player;//NO!
    VLC vlcPlayer;
    cout << endl;
    vlcPlayer.open("./resources/orchestral.ogg");
    vlcPlayer.play();
    vlcPlayer.setVolume(13);
    cout << endl;
    cout << endl;
    Winamp winampPlayer;
    cout << endl;
    winampPlayer.open("./resources/orchestral.ogg");
    winampPlayer.play();
    winampPlayer.setVolume(13);
    cout << endl;

    return EXIT_SUCCESS;
}
```

The Player constructor was invoked.

The volume value is: 10

The VLC constructor was invoked.

The audiofile: ./resources/orchestral.ogg is open.

The audiofile is playing.

The volume value is: 13

The Player constructor was invoked.

The volume value is: 10

The Winamp constructor was invoked.

The audiofile: ./resources/orchestral.ogg is open.

The audiofile is playing.

The volume value is: 13

The Winamp destructor was invoked.

The Player destructor was invoked.

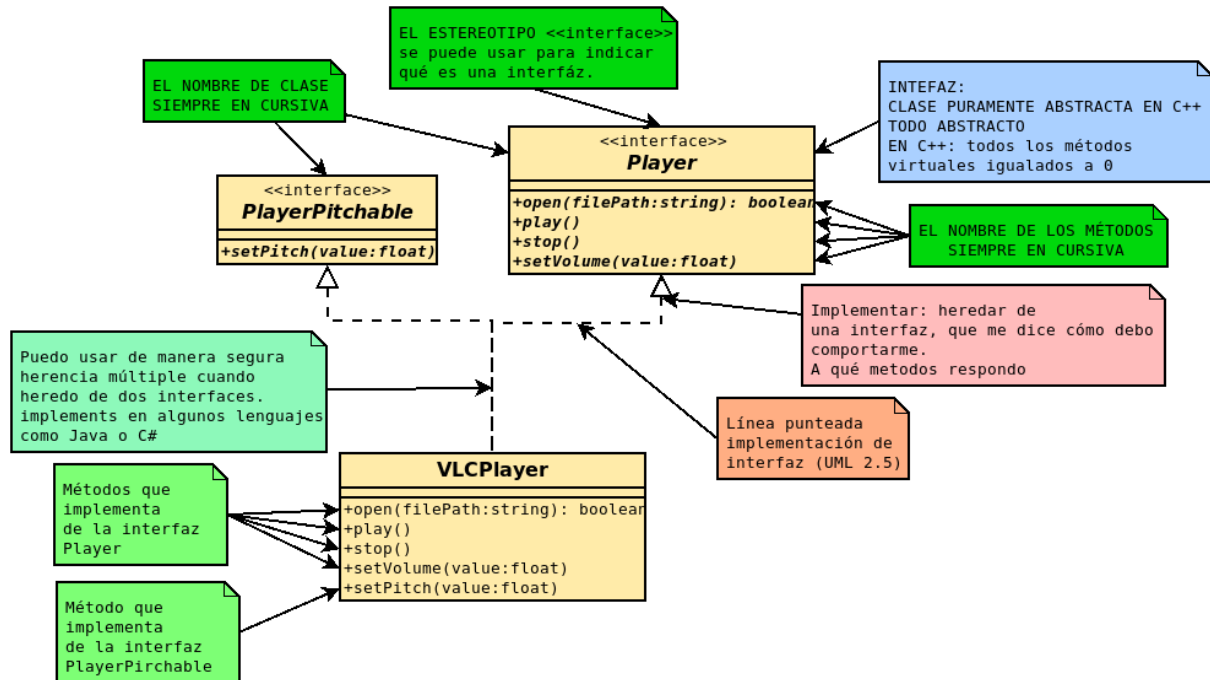
The VLC destructor was invoked.

The Player destructor was invoked.

—

Herencia de interfaz:

Hereda solo comportamiento de la superclase, no hereda implementación



```

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

//Player Interface:
class Player
{
public:
    virtual void open(string filePath) = 0;
    virtual void play() = 0;
    virtual void stop() = 0;
    virtual void setVolume(float value) = 0;
};

//PlayerPitchable Interface:
class PlayerPitchable
{
public:
    virtual void setPitch(float pitch) = 0;
};
    
```

```
//VLC concrete class
class VLC : public Player, public PlayerPitchable
{
public:
    VLC()
    {
        cout << "The VLC constructor was invoked." << endl;
        isOpen = false;
        isPlaying = false;
        setVolume(10);
    }
    ~VLC()
    {
        cout << "The VLC destructor was invoked." << endl;
    }
    void open(string filePath)
    {
        //@ToDo: por ahora simulamos la apertura correcta
        isOpen = true;
        cout << "The audiofile: " << filePath << " is open." << endl;
    }
    void play()
    {
        //@ToDo: por ahora simulamos la reproducción correcta
        if(isOpen) isPlaying = true;
        cout << "The audiofile is playing." << endl;
    }
    void stop()
    {
        //@ToDo: por ahora simulamos la detención correcta
        if(isPlaying) isPlaying = false;
        cout << "The audiofile is stopped." << endl;
    }
    void setVolume(float value)
    {
        volume = value;
        cout << "The volume value is: " << volume << endl;
    }
    void setPitch(float value)
    {
        pitch = value;
        cout << "The pitch value is: " << pitch << endl;
    }
}
```

```
private:
    bool isOpen;
    bool isPlaying;
    float volume;
    float pitch;
};

int main()
{
    //NO SE PUEDE CREAR UN OBJETO O INSTANCIA DE UNA INTERFAZ:
    //Player player;//NO!
    VLC vlcPlayer;
    cout << endl;
    vlcPlayer.open("./resources/orchestral.ogg");
    vlcPlayer.play();
    vlcPlayer.setVolume(13);
    cout << endl;

    return EXIT_SUCCESS;
}
```

```
The VLC constructor was invoked.
The volume value is: 10
```

```
The audiofile: ./resources/orchestral.ogg is open.
The audiofile is playing.
The volume value is: 13
```

```
The VLC destructor was invoked.
```

Referencias

- Blé Jurado, Carlos y colaboradores (2010) Diseño Agil con TDD. Primera Edición, Enero de 2010.
- Booch, G. (1996). Análisis y diseño orientados a objetos con aplicaciones, 2ª edición. Addison-Wesley/Díaz de Santos. ISBN: 0-201-60122-2
- Brooks Jr, F. P. (1995). The mythical man-month: essays on software engineering. Pearson Education.
- Fowler, Martin (2018). Refactoring: improving the design of existing code. Addison-Wesley Professional.
- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (2003). Patrones de diseño: *Elementos de software orientado a objetos reutilizable*. Addison-Wesley/Pearson Education.
- Khoshafian, S. and Copeland, G. (1986). Object Identity. *SIGPLAN Notices vol. 21*.
- McLuhan, Marshall (1994). Understanding Media: The Extensions of Man. Cambridge: The MIT Press. Print.
- Papert, Seymour (1993) Mindstorms: Children, Computers and Powerful Ideas, 2nd edition. Harvester Wheatsheaf, UK.
- Wulf, W. A. (1980). Trends in the design and implementation of programming languages. *Computer*, 13(01), 14-25.