UNIVERSIDAD PRIVADA DE TACNA


INGENIERIA DE SISTEMAS

TEMA:

# Test-Driven Development

## CURSO:

BASE DE DATOS II

## DOCENTE(ING):

Patrick Jose Cuadros Quiroga

Integrantes:

| | |
|---|---|
| Marko Antonio RIVAS RIOS | (2016055461) |
| Jorge Luis MAMANI MAQUERA | (2016055236) |
| Andree Ludwed VELASCO SUCAPUCA | (2016055286) |
| Yofer Nain CATARI CABRERA | (2017059289) |
| Adnner Sleyder ESPERILLA RUIZ | (2015050543) |
| Jesus ESCALANTE ALANOCA | (2015050641) |

# Índice

# 1. Objetivos del Desarrollo Orientado A Pruebas (TDD)

El propósito de esta técnica tiene los siguientes 3 objetivos básicos:

**1. Minimizar el número de bugs:** Mientras más bugs salgan menos rentable es el proyecto, porque corregir los bugs es tiempo que se puede invertir mejor en otras tareas. Si no hay bugs se puede conseguir una mayor rentabilidad de la aplicación.

**2. Implementar las funcionalidades justas que el cliente necesita:**

Es muy común que cuando se explican los requisitos de una aplicación o las especificaciones en la fase de análisis y diseño se esboza una aplicación, y el diseñador por su experiencia con funcionalidades pensando que van a ser útiles para la aplicación, para el cliente o para otros componentes. Sin embargo casi el 95 % de esas funcionalidades extras no se usan en ninguna parte de la aplicación, eso implica tiempo invertido desarrollando algo que no ha llegado a nada. El objetivo de TDD es eliminar ese código innecesario y esas funcionalidades que no ha pedido el cliente, con lo cual redunda en eficiencia, tanto en el trabajo y como en la rentabilidad de la aplicación.

**3. Producir software modular, altamente reutilizable y preparado para el cambio:** Esto es realmente más técnica, porque con buenos hábitos de programación siempre se logra que el proyecto sea modular y reutilizable. Prepararlo para el cambio es una característica que no se consigue siempre y que con TDD si, ya que muchas veces cuando se tiene que cambiar la funcionalidad de la aplicación se tiene que refactorizar código ajeno, trabajar con código complicado, entre otras cosas; en cambio con TDD se tiene la confianza de que cuando se haga cambios no se van a estropear las funcionalidades que ya se tienen. Esto se consigue ya que la forma funcional de TDD es que primero se construye la prueba y luego el código hace que todo lo que surja en él ya este testeado, así que cualquier cambio que se vaya a introducir estará cubierto por los tests y si llegas a dañar algo alguno de ellos reaccionara cuando se ejecuten.

Todo esto cambia un poco la mentalidad tradicional que es: primero analizar los requisitos, luego hacer un diseño completo y profesional, después empezar a codificar y por último testear. Lo que hay que hacer es que, en vez de planear tareas pensar en ejemplos y datos concretos, ya que en eso se basa los test, tener parámetros de entrada y de salida y luego ver si la respuesta es lo que se esperaba. Si con TDD consigues tener en las especificaciones o requisitos una lista de ejemplos muy completa, que sean concretos, que elimine cualquier tipo de ambigüedad y que se puedan transformar en pruebas, al final tendrás una batería de tests que te cubrirán todas las funcionalidades y el código resultante estará 100 % cubierto por dichos test.

# 2. Desarrollo Orientado A Pruebas (TDD)

### 1. ¿Qué es Desarrollo Orientado A Pruebas (TDD)?

Esta técnica llamada TDD (Test Driven Development), se puede definir como un proceso de desarrollo de software que se basa en la idea de desarrollar unas pequeñas pruebas, codificarlas y luego refactorizar el código que hemos implementado anteriormente. Podemos decir que esta técnica e implementación de software está dentro de la metodología XP donde deberíamos de echarle un ojo a todas sus técnicas, tras leer varios artículos en un coincido con Peter Provost con un diseño dirigido o implementado a base de ejemplos hubiese sido mejor pero TDD se centra en 3 objetivos claros:

- Una implementación de las funciones justas que el cliente necesita y no más, solamente las funciones que necesitamos, estoy cansado de duplicar dichas funciones para que hagan lo mismo

  Mínimos defectos en fase de producción

- Producción de software modular y sobre todo reutilizable y preparado para el cambio

  Esta técnica se basa en la idea de realizar unas pruebas unitarias para un código que nosotros debemos construir, Nuestro TDD lo que nos dice es que primero los programadores debemos realizar una prueba y a continuación empezar a desarrollar el código que la resuelve. El método que debemos seguir a para empezar a utilizar TDD es sencillo, Nos sirve para elegir uno de los requisitos a implementar, buscar un primer ejemplo sencillo, crear una prueba, ejecutarla e implementar el código mínimo para superar dicha prueba. Obviamente la gracia de ejecutar la prueba después de crearla es ver que esta falla y que será necesario hacer algo en el código para que esta pase. El ciclo de desarrollo de TDD es empezar la prueba, en test realizar un test, revisar el código y pasar el refactor.

  **Crear la prueba o test**

- Ejecutar los tests: falla (ROJO)
- Crear código específico para resolver el test
- Ejecutar de nuevo los tests: pasa (VERDE)
- Refactorizar el código
- Ejecutar los tests: pasa (VERDE)

  **Personalmente, añadiría lo siguiente:**

- Incrementa la productividad.
- Nos hace descubrir y afrontar más casos de uso en tiempo de diseño.
- La jornada se hace mucho más amena.
- Uno se marcha a casa con la reconfortante sensación de que el trabajo está bien hecho.

  Ahora bien, como cualquier técnica, no es una varita mágica y no dará el mismo resultado a un experto arquitecto de software que a un programador junior que está empezando. Sin

embargo, es útil para ambos y para todo el rango de integrantes del equipo que hay entre uno y otro. Es una técnica a tener en cuenta en el desarrollo web y sobre todo en el desarrollo de ingeniería software donde debemos tener en cuenta muchos fallos antes de pasar a producción.

# 3. Ventajas del TDD

- Puedes mejorar el código de tu aplicación en cualquier momento sin miedo a que dañes algo, ya que las pruebas ya las tienes listas y deberán pasar siempre.
- Los test que realizamos sobre las interfaces de nuestra app no siempre son completos, generalmente es lo que nos acordamos probar.
- Los equipos de testing, development y analyst serán más felices.
- La lectura del código será mucho mejor al tener ejemplos de uso (las pruebas).

# 4. Herramientas

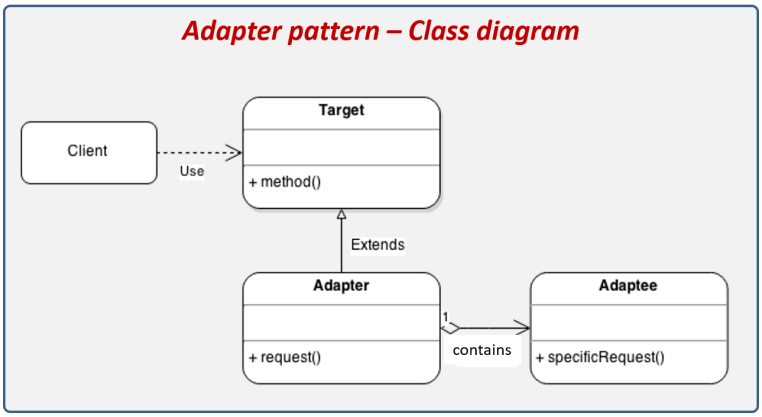
Dependiendo del lenguaje de programación, puedes escoger entre decenas de frameworks que te permitirán hacer pruebas, algunos conocidos son:

Java - JUnit, REST assured, Selenium, Mockito, Spock, etc.

- JavaScript - Jasmine, AVA, Tape, Mocha, Jest, etc.
- PHP - PHPUnit, Codeception, Behat, PHPSpec, SimpleTest, Storyplayer, etc.
- Python - [https://wiki.python.org/moin/PythonTestingToolsTaxonomy](clic para ver algunas herramientas)
- Go - El paquete testing nativo de Go.

# 5. Patrones de Diseño Adapter

El patrón de diseño Adapter es utilizado cuando tenemos interfaces de software incompatibles, las cuales a pesar de su incompatibilidad tiene una funcionalidad similar. Este patrón es implementado cuando se desea homogeneizar la forma de trabajar con estas interfaces incompatibles, para lo cual se crea una clase intermedia que funciona como un adaptador. Esta clase adaptador proporcionará los métodos para interactuar con la interface incompatible.



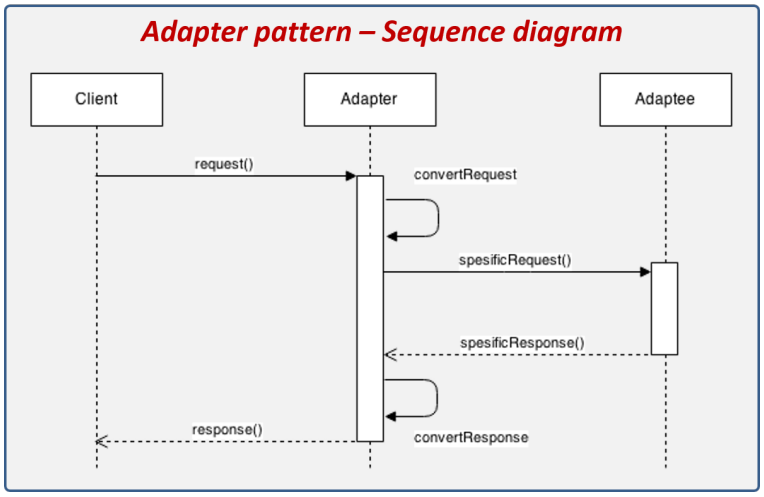

**Los componentes que conforman el patrón son los siguientes:**

a) Client: Actor que interactua con el Adapter.

b) Target: Interface que nos permitirá homogenizar la forma de trabajar con las interfaces incompatibles, esta interface es utilizada para crear los Adapter.

c) Adapter: Representa la implementación del Target, el cual tiene la responsabilidad de mediar entre el Client y el Adaptee. Oculta la forma de comunicarse con el Adaptee.

d) Adaptee: Representa la clase con interface incompatible.




a) El Client invoca al Adapter con parámetros genéricos.

b) El Adapter convierte los parámetros genéricos en parámetros específicos del Adaptee.

c) El Adapter invoca al Adaptee.

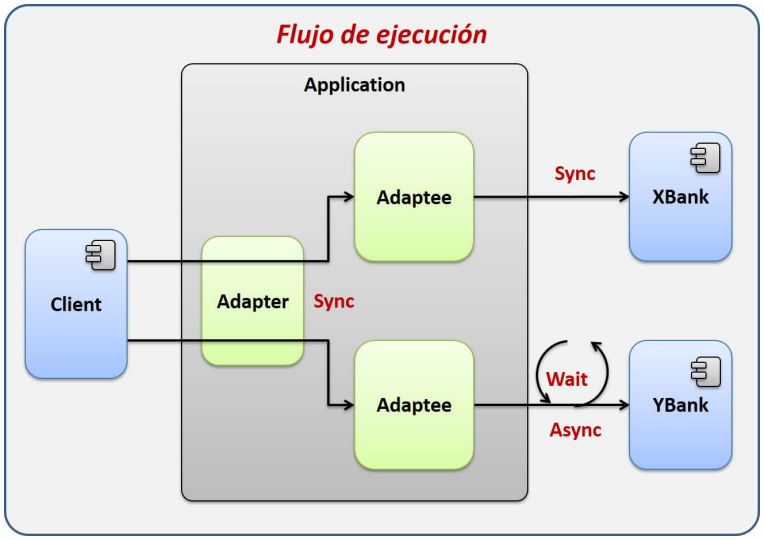
d) El Adaptee responde.

e) El Adapter convierte la respuesta del Adaptee a una respuesta genérica para el Client.

f) El Adapter responde al Client con una respuesta genérica.

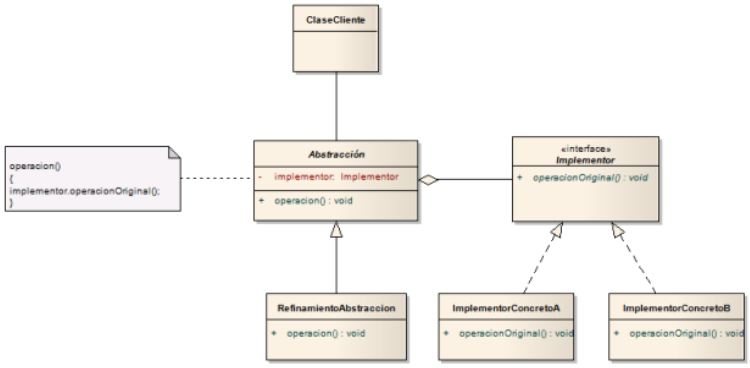
**EJEMPLO DEL MUNDO REAL**

Mediante la implementación del patrón de diseño Adapter crearemos un adaptador que nos permite interactuar de forma homogénea entre dos API bancarías, las cuales nos permite aprobar créditos personales, sin embargo, las dos API proporcionadas por los bancos cuenta con interfaces diferentes y aunque su funcionamiento es prácticamente igual, las interfaces expuestas son diferentes, lo que implica tener dos implementaciones diferentes para procesar los préstamos con cada banco. Mediante este patrón crearemos un adaptador que permitirá ocultar la complejidad de cada implementación del API, exponiendo una única interface compatible con las dos API proporcionadas, además que dejáramos el camino preparado por si el día de mañana llegara una nueva API bancaría.



# 6. Patrón de Diseño Bridge

Es normalmente uno de los patrones que más cuesta entender, especialmente si nos ceñimos únicamente a su descripción. La idea tras este patrón, sin embargo, es sencilla: dado que cualquier cambio que se realice sobre una abstracción afectará a todas las clases que la implementan, Bridge propone añadir un nuevo nivel de abstracción entre ambos elementos que permitan que puedan desarrollarse cada uno por su lado.

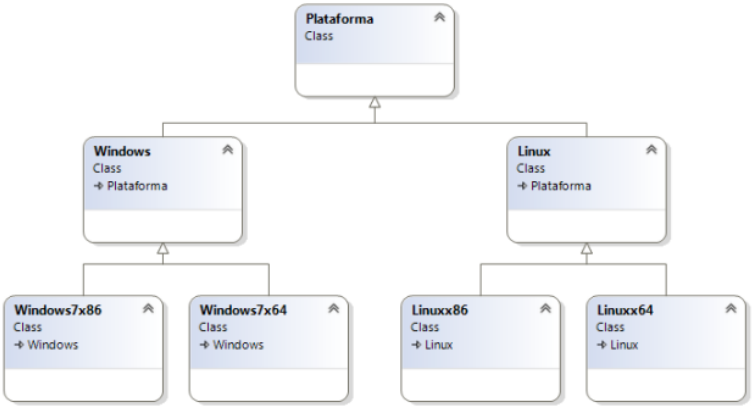



**1. ¿Por qué "Bridge"?**

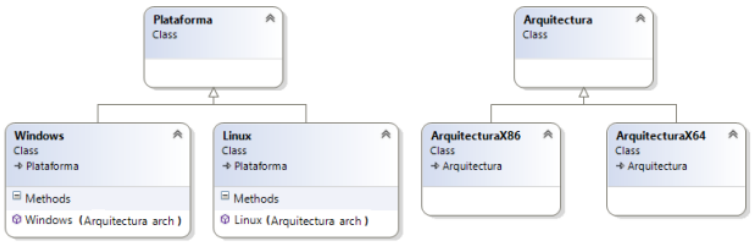
Patrón tras patrón, vemos que los mismos conceptos se repiten una y otra vez. Minimizar el acoplamiento, hacer que las clases dependan de abstracciones en lugar de depender de implementaciones, preferir el uso de composición antes que el uso de herencia... El patrón Bridge no es una excepción. Sin embargo, hasta el momento todos los patrones que hemos visto tenían una relación entre su nombre y su funcionalidad. Un Adapter adapta. Una factoría fabrica objetos. Un Builder construye. Pero... ¿Bridge? ¿Qué tiene que ver un puente con todo esto?

Una de las razones por las que opino que este patrón es complicado de entender a la primera es, precisamente, que no existe una relación clara entre su nombre y su descripción. ¿Llamar puente a desligar una interfaz de la implementación? ¿Por qué? La razón no está tanto en este proceso sino en el camino que existe entre la clase que refina la abstracción y las implementaciones de la interfaz.

Hablando en plata, y a modo de resumen, realizaremos la transformación del siguiente árbol de herencia:



En una composición como la siguiente:




**2. Similitudes**

La estructura de este patrón se parece mucho a la del patrón Adapter, ya que nuestra clase Abstracción hace las veces de "adaptador" entre nuestra clase cliente y la interfaz Implementor. Sin embargo, nos movemos por la sinuosa senda de la ingeniería, por lo que afirmar que Adapter y Bridge realizan lo mismo simplemente porque su estructura sea muy parecida es quedarnos en la superficie del problema que tratamos de resolver. La estructura de este patrón se parece mucho a la del patrón Adapter, ya que nuestra clase Abstracción hace las veces de "adaptador" entre nuestra clase cliente y la interfaz Implementor. Sin embargo, nos movemos por la sinuosa senda de la ingeniería, por lo que afirmar que Adapter y Bridge realizan lo mismo simplemente porque su estructura sea muy parecida es quedarnos en la superficie del problema que tratamos de resolver.

– **Un ejemplo de patrón Bridge**

Veamos la aplicación de nuestro patrón Bridge con un ejemplo en código C sharp. El ejemplo, como seguro que habréis adivinado, estará basado en vehículos. Nuestra abstracción simbolizará el vehículo en sí, mientras que la parte que se tenderá "al otro lado del puente" será el motor del mismo. Los tipos de vehículo podrán así evolucionar con independencia de los motores que éstos posean. Comenzaremos codificando la interfaz Implementor, que en nuestro ejemplo estará representada por el motor. O más específicamente, por la interfaz IMotor.

**MOTOR**

```
1  // Implementor
2  public interface IMotor
3  {
4      void InyectarCombustible(double cantidad);
5      void ConsumirCombustible();
6  }
```

Como vemos, nada complicado: nuestro Implementor expone dos métodos, InyectarCombustibley ConsumirCombustible, que deberán ser codificados en las clases que implementen la interfaz. Y dicho y hecho, añadiremos un par de clases cuyo papel en el patrón se corresponderá con ImplementorConcretoA e ImplementorConcretoB, y modelarán dos tipos de motores: diesel y gasolina.

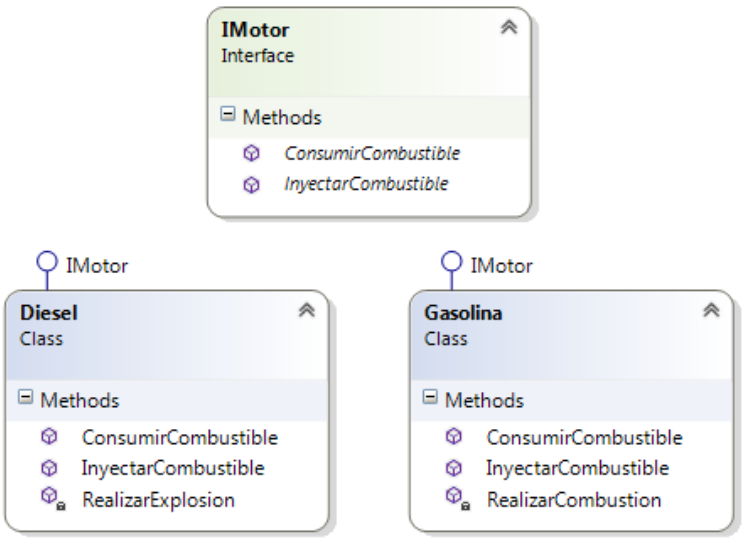
**DIESEL**

```
// ImplementorConcretoA
public class Diesel : IMotor
{
    #region IMotor Members

    public void InyectarCombustible(double cantidad)
    {
        Console.WriteLine("Inyectando " + cantidad + " ml. de Gasoil");
    }

    public void ConsumirCombustible()
    {
        RealizarExplosion();
    }

    #endregion

    private void RealizarExplosion()
    {
        Console.WriteLine("Realizada la explosión del Gasoil");
    }
}
```

## GASOLINA

```
// ImplementorConcretoB
public class Gasolina : IMotor
{
    #region IMotor Members

    public void InyectarCombustible(double cantidad)
    {
        Console.WriteLine("Inyectando " + cantidad + " ml. de Gasolina");
    }

    public void ConsumirCombustible()
    {
        RealizarCombustion();
    }

    #endregion

    private void RealizarCombustion()
    {
        Console.WriteLine("Realizada la combustión de la Gasolina");
    }
}
```

Con estas tres clases ya habríamos desarrollado el subárbol izquierdo del diagrama UML que mostramos al comienzo del artículo: la interfaz Implementor junto a sus implementaciones:



La siguiente parte será encapsular la interfaz dentro de nuestra abstracción Vehiculo, que dispondrá de una referencia a IMotor y de un método que hará uso de los métodos de nuestra interfaz, encapsulando su funcionalidad tal y como hacíamos en el patrón Adapter:

## VEHICULO

```
// Abstracción
public abstract class Vehiculo
{
    private IMotor motor;

    public Vehiculo(IMotor motor)
    {
        this.motor = motor;
    }

    // Encapsulamos la funcionalidad de la interfaz IMotor
    public void Acelerar(double combustible)
    {
        motor.InyectarCombustible(combustible);
        motor.ConsumirCombustible();
    }

    public void Frenar()
    {
        Console.WriteLine("El vehículo está frenando.");
    }

    // Método abstracto
    public abstract void MostrarCaracteristicas();
}
```

Como venimos observando a lo largo de los últimos patrones, el objeto que implementará el motor es inyectado en el constructor, siguiendo el quinto de los principios SOLID.
Por último, codificaremos la evolución de nuestra abstracción, que se corresponderá con RefinamientoAbstracciónA y RefinamientoAbstracciónB, representadas por dos tipos de vehículos: Berlina y Monovolumen.

## VERLINA

```
// RefinamientoAbstraccionA
public class Berlina : Vehiculo
{
    // Atributo propio
    private int capacidadMaletero;

    // La implementacion de los vehículos se desarrolla de forma independiente
    public Berlina(IMotor motor, int capacidadMaletero) : base(motor)
    {
        this.capacidadMaletero = capacidadMaletero;
    }

    // Implementación del método abstracto
    public override void MostrarCaracteristicas()
    {
        Console.WriteLine("Vehiculo de tipo Berlina con un maletero con una capacidad
            capacidadMaletero + " litros.");
    }
}
```

**3. ¿Cuándo utilizar este patrón?**

Un ejemplo típico de un patrón Bridge lo puede conformar cualquier familia de drivers de un dispositivo, tal y como vimos en el primer ejemplo.

Otro ejemplo típico suele ser el de las APIs de dibujo. Los elementos genéricos, tales como formas y figuras serían las abstracciones (por ejemplo, Forma sería el elemento Abstraction del que derivarían abstracciones refinadas como Circulo o Cuadrado), mientras que la parte "dependiente" del sistema sería la API concreta que se encargaría de dibujar en pantalla las formas genéricas definidas en la abstracción. Este funcionamiento puede observarse en los paquetes de java java.awt y java.awt.peer. (en Button y List, por ejemplo).

Las situaciones óptimas en los que se debe utilizar este patrón serán, por tanto:

- Cuando se desea evitar un enlace permanente entre la abstracción y (toda o parte de) su implementación.
- Cuando los cambios en la implementación de una abstracción no debe afectar a las clases que hace uso de ella.
- Cuando se desea compartir una implementación entre múltiples objetos.

# 7. Webgrafía


- https://www.oscarblancarteblog.com/2014/10/07/patron-de-diseno-composite/
- http://arantxa.ii.uam.es/ eguerra/docencia/0708/05 %20Composite.pdf
- https://programacion.net/articulo/patrones_de_diseno_ix_patrones_estructurales_composite_1011
- http://design-patterns-with-uml.blogspot.com/2013/02/facade-pattern.html
- https://www.genbeta.com/desarrollo/diseno-con-patrones-y-fachadas
- http://blog.koalite.com/2016/12/los-patrones-de-diseno-hoy-patrones-estructurales/
- https://slideplayer.es/slide/3097642/
- https://slideplayer.es/slide/5568855/