Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ingeniería en Ciencias y Sistemas

DESARROLLO DE APLICACIONES EN PHP UTILIZANDO DESARROLLO GUIADO POR PRUEBAS

Carlos Fernando Narez Garrido
Asesorado por el Ing. Daniel Caciá Rivas

Guatemala, enero de 2016

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

DESARROLLO DE APLICACIONES EN PHP UTILIZANDO DESARROLLO GUIADO POR PRUEBAS

TRABAJO DE GRADUACIÓN

PRESENTADO A LA JUNTA DIRECTIVA DE LA
FACULTAD DE INGENIERÍA
POR

CARLOS FERNANDO NAREZ GARRIDO
ASESORADO POR EL ING. DANIEL CACIÁ RIVAS

AL CONFERÍRSELE EL TÍTULO DE

INGENIERO EN CIENCIAS Y SISTEMAS

GUATEMALA, ENERO DE 2016

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

NÓMINA DE JUNTA DIRECTIVA

DECANO	Ing. Pedro Antonio Aguilar Polanco
VOCAL I	Ing. Angel Roberto Sic García
VOCAL II	Ing. Pablo Christian de León Rodríguez
VOCAL III	Inga. Elvia Miriam Ruballos Samayoa
VOCAL IV	Br. Raúl Eduardo Ticún Córdova
VOCAL V	Br. Henry Fernando Duarte García
SECRETARIA	Inga. Lesbia Magalí Herrera López

TRIBUNAL QUE PRACTICÓ EL EXAMEN GENERAL PRIVADO

DECANO	Ing. Murphy Olympo Paiz Recinos
EXAMINADORA	Inga. Virginia Victoria Tala Ayerdi
EXAMINADOR	Ing. Freiry Javier Gramajo López
EXAMINADOR	Ing. César Augusto Fernández Cáceres
SECRETARIA	Inga. Marcia Ivónne Véliz Vargas

HONORABLE TRIBUNAL EXAMINADOR

En cumplimiento con los preceptos que establece la ley de la Universidad de San Carlos de Guatemala, presento a su consideración mi trabajo de graduación titulado:

DESARROLLO DE APLICACIONES EN PHP UTILIZANDO DESARROLLO GUIADO POR PRUEBAS

Tema que me fuera asignado por la Dirección de la Escuela de Ingeniería en Ciencias y Sistemas, con fecha 16 de agosto de 2015.

Carlos Fernando Narez Garrido

Guatemala, 26 de Octubre de 2015

Señores
Comisión de Revisión de Trabajos de Tesis
Escuela de Ciencias y Sistemas
Facultad de Ingeniería
Universidad de San Carlos de Guatemala

Respetables Señores:

De manera atenta me dirijo a ustedes para hacer de su conocimiento que el estudiante Carlos Fernando Narez Garrido con número de carné 1998-11092 ha finalizado su Trabajo de Graduación, titulado "Desarrollo de aplicaciones en PHP utilizando Desarrollo guiado por pruebas".

A dicho trabajo se le efectuaron las revisiones necesarias y se realizaron los ajustes pertinentes para que pueda continuar con el proceso de revisión. Me es grato suscribirme de ustedes.

Atentamente,

Daniel Caciá Rivas
Ingeniero en Ciencias y Sistemas
Colegiado No. 8882

Ing. Daniel Caciá Rivas
Colegiado 8,882

Universidad San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ingeniería en Ciencias y Sistemas

Guatemala, 4 de Noviembre de 2015

Ingeniero
Marlon Antonio Pérez Türk
Director de la Escuela de Ingeniería
En Ciencias y Sistemas

Respetable Ingeniero Pérez:

Por este medio hago de su conocimiento que he revisado el trabajo de graduación del estudiante **CARLOS FERNANDO NAREZ GARRIDO** con carné **1998-11092,** titulado: **"DESARROLLO DE APLICACIONES EN PHP UTILIZANDO DESARROLLO GUIADO POR PRUEBAS",** y a mi criterio el mismo cumple con los objetivos propuestos para su desarrollo, según el protocolo.

Al agradecer su atención a la presente, aprovecho la oportunidad para suscribirme,

Atentamente,

Ing. Carlos Alfredo Azurdia
Coordinador de Privados
y Revisión de Trabajos de Graduación

ESCUELA DE CIENCIAS Y SISTEMAS

UNIVERSIDAD DE SAN CARLOS
DE GUATEMALA

FACULTAD DE INGENIERÌA
ESCUELA DE CIENCIAS Y SISTEMAS
TEL: 24767644

El Director de la Escuela de Ingeniería en Ciencias y Sistemas de la Facultad de Ingeniería de la Universidad de San Carlos de Guatemala, luego de conocer el dictamen del asesor con el visto bueno del revisor y del Licenciado en Letras, del trabajo de graduación ***"DESARROLLO DE APLICACIONES EN PHP UTILIZANDO DESARROLLO GUIADO POR PRUEBAS"****, realizado por el estudiante CARLOS FERNANDO NAREZ GARRIDO, aprueba el presente trabajo y solicita la autorización del mismo.*

"ID Y ENSEÑAD A TODOS"

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA
DIRECCION DE INGENIERIA EN CIENCIAS Y SISTEMAS

Ing. Marlon Antonio Pérez Türk
Director, Escuela de Ingeniería en Ciencias y Sistemas

Guatemala, 21 de Enero de 2015

Universidad de San Carlos
de Guatemala

Facultad de Ingeniería
Decanato

DTG. 029.2016

El Decano de la Facultad de Ingeniería de la Universidad de San Carlos de Guatemala, luego de conocer la aprobación por parte del Director de la Escuela de Ingeniería en Ciencias y Sistemas, al Trabajo de Graduación titulado: **DESARROLLO DE APLICACIONES EN PHP UTILIZANDO DESARROLLO GUIADO POR PRUEBAS,** presentado por el estudiante universitario: **Carlos Fernando Narez Garrido,** y después de haber culminado las revisiones previas bajo la responsabilidad de las instancias correspondientes, autoriza la impresión del mismo.

IMPRÍMASE:

Ing. Pedro Antonio Aguilar Polanco
Decano

Guatemala, enero de 2016

/gdech

ACTO QUE DEDICO A:

Mis padres	Carlos Narez y Teresa Garrido, por su incansable esfuerzo para brindarme el mayor de los regalos, mi educación.
Mis hermanas	Ana Patricia y Cristy Narez Garrido, por apoyarme y animarme en cada etapa de mi carrera.
Mi esposa e hijos	Ligia de Narez, Carlos Ernesto y María Fernanda Narez, por ser el pilar de apoyo en esta etapa de mi carrera y sobre todo el incentivo para completarla.

AGRADECIMIENTOS A:

Universidad de San Carlos de Guatemala	Por darme la oportunidad de estudiar en ella y facilitarme los conocimientos que serán mis principales herramientas de trabajo.
Mi asesor	Daniel Caciá, por su importante labor en este trabajo de graduación.

ÍNDICE GENERAL

ÍNDICE DE ILUSTRACIONES

FIGURAS

GLOSARIO

API	Interfaz de programación de aplicaciones por sus siglas en inglés (Application Program Interface).
DIP	Principio de diseño Inversión de Dependencias por sus siglas en inglés (Dependency Inversion Principle).
Framework	Conjunto de conceptos, prácticas o herramientas para desarrollar soluciones de problemas de la misma índole.
ISP	Principio de diseño Segregación de Interfaces por sus siglas en inglés (Interface Segregation Principle).
LSP	Principio de diseño Sustitución Liskov por sus siglas en inglés (Liskov Substitution Principle).
OCP	Principio de diseño abierto/cerrado, por sus siglas en inglés (Open/Close Principle).
PHP	PHP Hypertext Proccesor es un lenguaje de programación del lado del servidor utilizado mayormente en el desarrollo web.

PHPSpec	*Framework* para pruebas unitarias en PHP basado en comportamientos.
PHPUnit	*Framework* para pruebas unitarias en PHP.
S+OLID	Principios de diseño en programación orientada a objetos.
SRP	Principio de diseño de responsabilidad única por sus siglas en inglés (Single Responsability Principle).

RESUMEN

A lo largo de la evolución del desarrollo de software se han conocido varias técnicas y procesos para mejorar la calidad del mismo. En el presente trabajo se profundiza sobre el desarrollo guiado por pruebas, específicamente en el lenguaje de programación PHP, el cual es utilizado mayormente para la implementación de sistemas web.

El desarrollo guiado por pruebas es un proceso en el cual se crean pruebas antes de empezar a programar el requerimiento específico. Con esto se logra el enfoque de lo que debería de hacer este código y al final de la implementación se posee un amplio repositorio de pruebas para verificar que el sistema funciona como estaba diseñado desde el principio.

PHP es uno de los lenguajes más populares hoy en día para el desarrollo en internet. Sin embargo, el desarrollo guiado por pruebas en dicho lenguaje no es tan utilizado hasta el momento. El enfoque de este documento es presentar las buenas prácticas y el procedimiento para poder implementarlo, obteniendo así sistemas confiables, legibles y fáciles de mantener.

OBJETIVOS

General

Establecer un proceso utilizando el desarrollo guiado por pruebas para el desarrollo de aplicaciones en PHP.

Específicos

1. Realizar una investigación documental para conocer el ciclo de desarrollo guiado por pruebas.

2. Definir los principios de diseño orientado a objetos SOLID, para la implementación de buenas prácticas en el desarrollo guiado por pruebas.

3. Realizar una investigación documental para conocer al menos 2 herramientas para la realización de pruebas unitarias en PHP.

4. Realizar una encuesta a los estudiantes de la Escuela de Ingeniería en Ciencias y Sistemas de la Universidad de San Carlos de Guatemala sobre el conocimiento y práctica del desarrollo guiado por pruebas.

INTRODUCCIÓN

Internet es una de las plataformas preferidas para desarrollar software, debido a su dinamismo y capacidad de llegar a más usuarios alrededor del mundo. La cantidad de desarrolladores ha aumentado grandemente y uno de los lenguajes de programación favoritos para implementar en dicha plataforma ha sido PHP, ya que su API es fácil de entender y es bastante flexible en su forma de uso.

Todo este dinamismo alrededor del desarrollo web, ha hecho que en ciertas ocasiones se sacrifique la calidad del software, en favor de la velocidad para crear sistemas que lleguen cada vez a más personas. El efecto de esto es encontrar muchos sistemas poco mantenibles y difícil de construir alrededor de ellos.

Debido a esta problemática hoy en día se está prestando más atención al diseño, procesos de implementación y verificación alrededor de PHP. Esto ha permitido que procesos de desarrollo totalmente maduros en otros lenguajes de programación más rígidos como .NET y Java sean totalmente viables ahora en PHP.

Uno de dichos procesos es el desarrollo guiado por pruebas, que mediante un ciclo de fases claramente definidas, logra enfocarse en el diseño, legibilidad y mantenibilidad del software, brindando además, un método de verificación para que cada parte del sistema siga funcionando como fue diseñada desde el inicio.

En el presente trabajo se desarrolla cada aspecto alrededor del desarrollo guiado por pruebas, desde los principios que se deben cumplir al diseñar aplicaciones orientadas a objetos, hasta cómo implementar cada fase que compone dicho proceso, en búsqueda de implementar sistemas altamente mantenibles y escalables.

1. DISEÑO ORIENTADO A OBJETOS

Según James Bender en su libro *Professional Test Driven Development with* dice: la programación orientada a objetos es un paradigma en el cual elementos del mundo real son representado en código fuente y este puede ser utilizado para modelar comportamientos reales de la lógica del negocio.

Esta abstracción permite obtener pequeños fragmentos de código que se comunican entre sí y pueden dar solución a problemas reales del negocio y a su vez ser reutilizados para que junto a otros fragmentos den solución a problemas aún más complejos. De allí radica la importancia que tiene el poder crear un buen diseño orientado a objetos.

Es por esto que el diseño orientado a objetos toma una gran importancia, ya que entre mejor esté construido dicho modelo, más fácil será implementar nuevas funcionalidades y mejor aún, se tendrá una base sólida ante las solicitudes de cambio.

Pero antes de profundizar en las buenas prácticas para obtener un buen diseño, se deben conocer los principios que rigen el paradigma de la programación orientada a objetos.

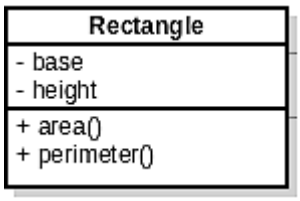
1.1. Principios de programación orientada a objetos

Como se mencionó anteriormente, en la programación orientada a objetos existen fragmentos de código que modelan elementos de la lógica del negocio, estos fragmentos son conocidos como clases y se conforman por atributos, que

son las características del objeto modelado, y métodos, que son los encargados de transformar dichas características o devolver un resultado con base en ellas.

Un ejemplo de clase podría ser un rectángulo, donde sus atributos serían su base y altura, y sus métodos, su área y perímetro.

Figura 1. **Clase rectángulo**



Fuente: elaboración propia.

La comunicación e interacción entre clases no sería posible si no existieran los 3 conceptos principales en la programación orientada a objetos, los cuales se explican a continuación.

1.1.1. Encapsulamiento

El principio del encapsulamiento se basa en convertir a una clase en una caja negra, donde solo se conoce su interfaz pública y no el detalle de su funcionamiento interno, todo esto desde la perspectiva de las clases que se comunican con ella.

Esto se consigue manejando el alcance de los atributos y métodos de una clase. En resumen solo los atributos y métodos que se van a comunicar con otras clases, deberían de ser públicos. Los atributos protegidos serían todos

aquellos que se pueden comunicar con clases descendientes de la misma y atributos privados todos los demás.

En la figura 1 se puede observar, que en la clase rectángulo, los atributos base y altura son privados, representados con el símbolo “-” y las clases que se quieran comunicar con rectángulo, solo pueden hacerlo mediante sus métodos área y perímetro, que son públicos representados con el símbolo “+”. Los atributos y métodos protegidos son representados con el símbolo “#”.

Según Luis Joyanes, en su libro *Programación orientada a objetos* dice: "Con el encapsulamiento se logra obtener independencia entre clases y se logra uno de los objetivos de la programación orientada a objetos, que es construir clases débilmente acopladas y con fuerte cohesión entre ellas".

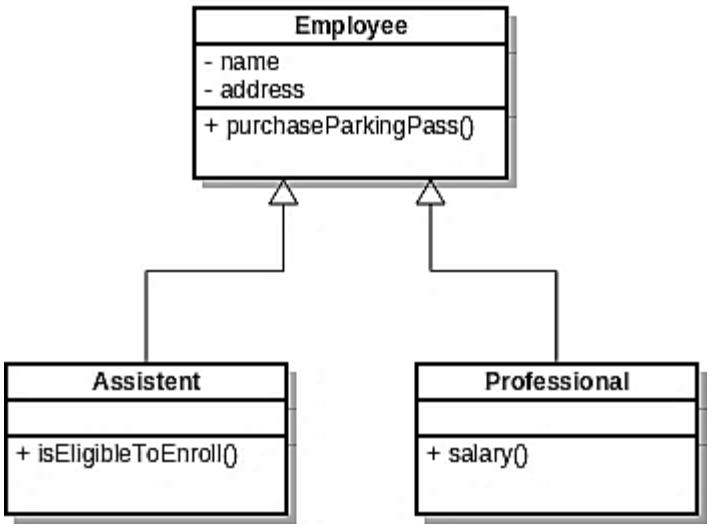
1.1.2. Herencia

Uno de los objetivos de la programación orientada a objetos es la reutilización de código, es decir que rutinas que dan solución a un problema, puedan ser reutilizadas en algún otro escenario donde aplique. La herencia permite mucho de esta reutilización, esto debido a que en un sistema se encuentran componentes que pueden clasificarse en orden jerárquico, de tal manera que se pueden definir en superclases y subclases.

Según Luis Joyanes, en su libro *Programación orientada a objetos* dice: "En sí, la herencia permite que clases hereden ciertos atributos y métodos a sus subclases y estas últimas solo implementan las funcionalidades que las hacen distintivas".

Como se puede ver en la figura 2, la clase Empleado, hereda sus atributos a las clases Asistente y Profesional, las cuales implementan los métodos característicos que las difieren una de la otra y a su vez comparte el método de pago de parqueo, el cual fue heredado de la clase Empleado.

Figura 2. **Diagrama de clases representando herencia**



Fuente: elaboración propia.

1.1.3. Polimorfismo

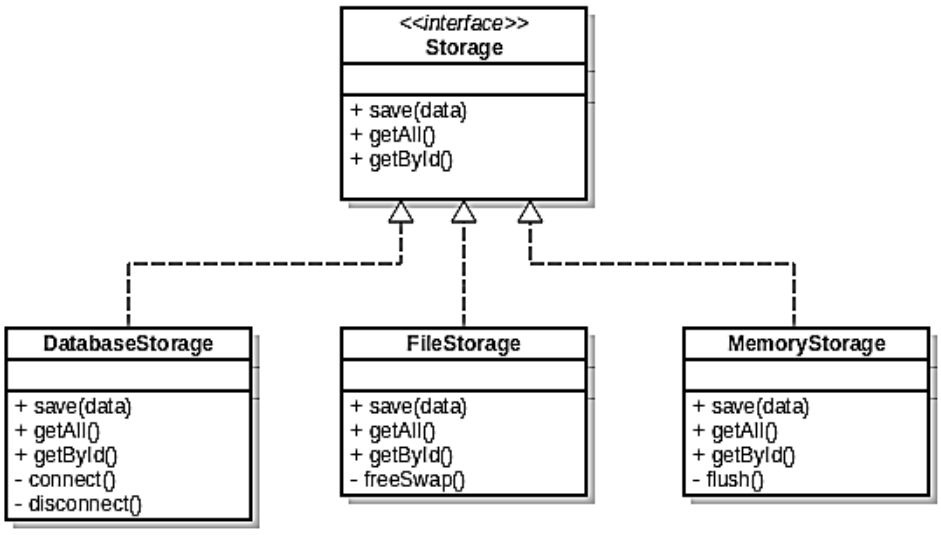
El último principio de la programación orientada a objetos es el polimorfismo y en síntesis es la habilidad de que 2 clases puedan compartir el mismo comportamiento, pero que su implementación sea diferente en cada una.

Esto permite una ventaja en el diseño de software orientado a objetos, debido a que se pueden definir contratos los cuales futuras clases deban cumplir y así puedan ser intercambiables entre sí. Según James Bender en su libro *Professional Test driven development with c#*: *developing real world*

applications with TDD dice: "Cuando se habla de contratos se refiere a especificar el conjunto de métodos que la clase debe implementar, para cumplir con el requerimiento y así asegurarnos que la clase esté obligada a implementar cada uno de ellos".

En la figura 3 se puede observar como una interfaz de almacenamiento especifica un contrato con los métodos guardar, obtener todos y obtener por id y así conseguir que las clases sean intercambiables en cualquier momento del proyecto y poder ofrecer el funcionamiento de almacenamiento por base de datos, sistema de archivos o en memoria, sin que la clases consumidoras sufran algún cambio.

Figura 3. **Diagrama de clases ejemplificado el polimorfismo**



Fuente: elaboración propia.

1.2. Principios de diseño orientado a objetos SOLID

Cuando se desarrollan sistemas relativamente grandes, una de las características que deben tener, es que su código sea fácilmente entendible y mantenible. Por lo general en sistemas grandes participan varios desarrolladores y estos deben de comprender fácilmente el código, para implementar nuevas funcionalidades, reutilizando los módulos ya desarrollados y sobre todo estar en la capacidad de hacer fácilmente cambios sobre estos.

Con base en esto, Robert “Uncle Bob” Martin introduce el término SOLID como principios de diseño, los cuales están enfocados en conseguir fácilmente los aspecto antes mencionados. Cada una de las letras corresponde a uno de los principios.

- (S)ingle Responsability Principle (Principio de Responsabilidad Única)
- (O)pen/Close Principle (Principio Abierto/Cerrado)
- (L)iskov Substitution Principle (Principio de Sustitución Liskov)
- (I)nterface Segregation Principle (Principio de Segregación de Interfaces)
- (D)ependency Invertion Principle (Principio de Inversión de Dependencia)

1.2.1. Principio de Responsabilidad Única (SRP)

Conocido también como SRP por sus siglas en inglés (Single Responsability Principle), este principio se basa en la premisa según Roberto Martin, en su libro *Agile principles, patterns, and practices in C* dice: “Cada clase debe tener solo una razón por cual cambiar”.

Este es uno de los principios más fáciles de asimilar, sin embargo, es uno de los más difíciles de alcanzar, ya que por lo general se adjudican más

responsabilidades de las que debiera tener cada clase, debido a que se tienden a agrupar responsabilidades, creando diseños acoplados y con alta probabilidad de cometer errores al exponerlo a distintas razones de cambio.

Pero ¿Cómo determinar las responsabilidades de cada clase? Esto depende de la lógica del negocio y de cómo se identifiquen las razones de cambio que esta dicta sobre el sistema.

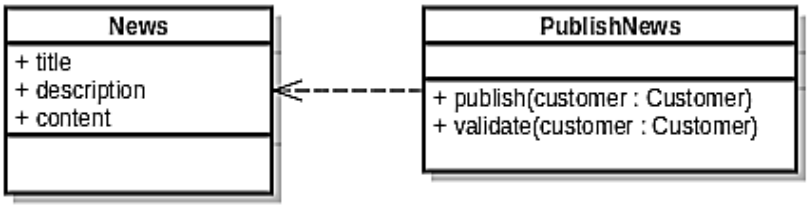
En el ejemplo anterior, sobre el almacenamiento en base de datos (figura 3) la clase DatabaseStorage, tiene más responsabilidades que la que debería tener. Además de manejar las funcionalidades de acceso a la información, se encarga de la conección a la base de datos. Por lo que un diseño más flexible sería el mostrado en la figura 4.

Figura 4. **Diagrama de clases representando el principio de responsabilidad única**

Fuente: elaboración propia.

Si se analiza otro ejemplo, esta vez el de un sistema de Publicación de Noticias, cometen el error de agrupar las responsabilidades y diseñar las clases de la figura 5.

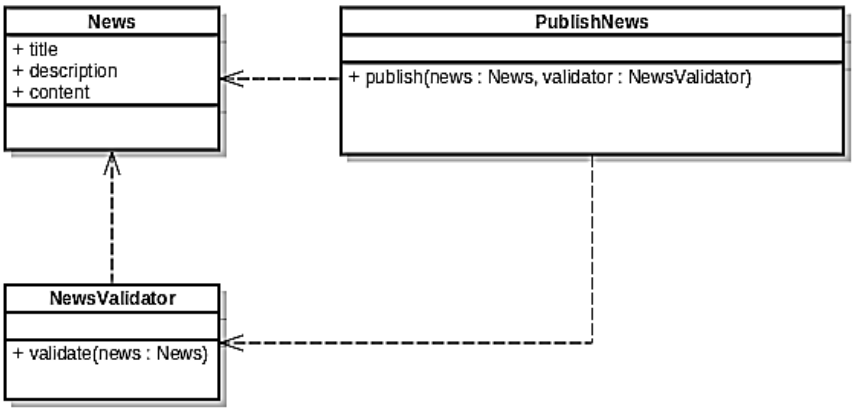
Figura 5. **Diseño de clases violando SRP**



Fuente: elaboración propia.

Con esto se ve fácilmente que la clase de publicación de noticias no tiene una única responsabilidad. Si las condiciones de validación variaran, se tendría que modificar la clase de publicación, por lo que sería mejor abstraer la validación a su propia clase y así poder reutilizarla en otros módulos, como se muestra en la figura 6.

Figura 6. **Diseño de clases cumpliendo con el principio de responsabilidad única**



Fuente: elaboración propia.

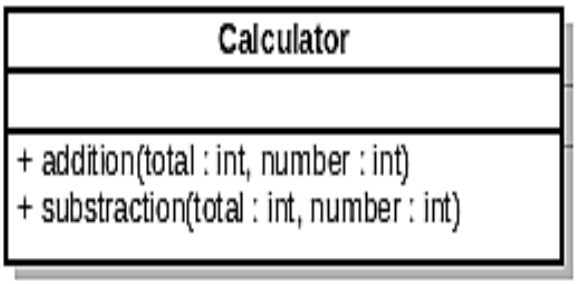
1.2.2. Principio abierto/cerrado (OCP)

Este principio que también es conocido como OCP por sus siglas en inglés (Open/Close Principle), indica que los módulos, clases o métodos de un sistema deben estar abiertos para su extensión pero cerrados para su modificación.

OCP es ligado fuertemente a la herencia y composición, ya que de esta manera se garantiza que el diseño sea incremental en caso de una nueva característica y que esta no implique una modificación del código existente, sobre todo en las clases consumidoras de las características actuales.

Si se analiza el diseño de una calculadora mostrado en la figura 7, se puede observar, que en caso se tenga que implementar una nueva operación, se tendria que ir a modificar esta clase en cada ocasión, incrementando la probabilidad de incorporar errores en las operaciones ya desarrolladas, además incrementaría su tamaño directamente proporcional a la cantidad de operaciones que se implementen, dificultando así el mantenimiento de la misma.

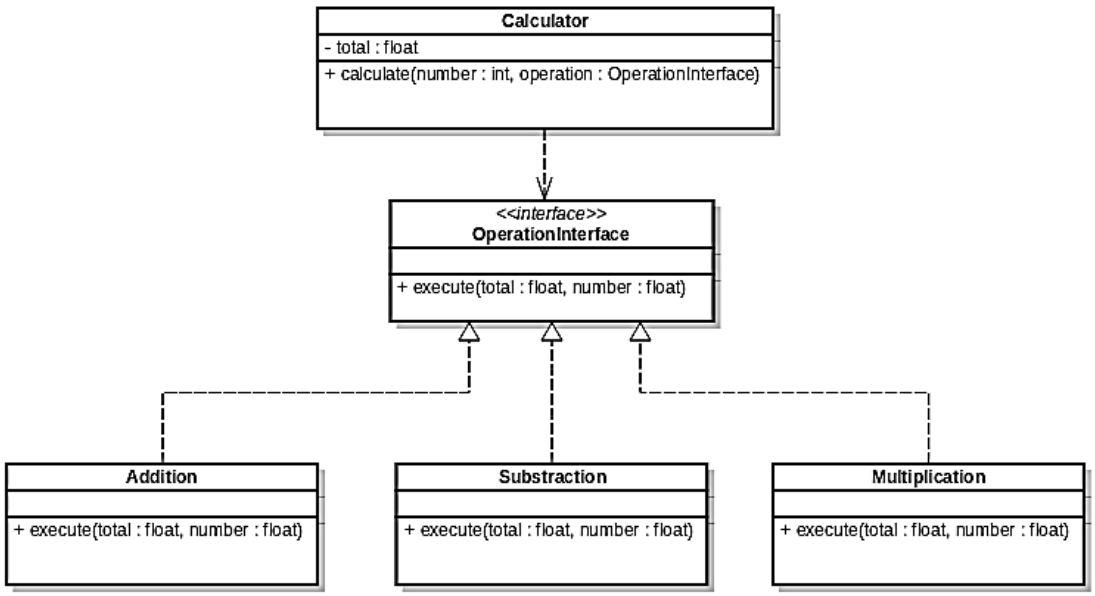
Figura 7. **Diseño de clases de una calculadora, violando OCP**



Fuente: elaboración propia.

En la búsqueda de crear un diseño débilmente acoplado, lo que se debería hacer es extraer detrás de una interfaz el comportamiento que varía según la lógica del negocio, en este caso las operaciones, e invertir las dependencias. Por lo que un diseño que cumpla con OCP sería el siguiente.

Figura 8. **Diseño de calculadora que cumple con OCP**



Fuente: elaboración propia.

De esta manera cualquier implementación de una nueva operación, no debería de implicar modificar la clase calculadora y por consiguiente, no deberían de sufrir cambios los consumidores de esta. Por otra parte, el diseño está abierto para la inclusión de cualquier número de operaciones que la lógica del negocio indique implementar y mejor aún, el sistema debería de ser fácilmente mantenible ya que cada operación está aislada en su propia clase.

1.2.3. Principio de sustitución Liskov (LSP)

Es conocido como LSP por sus siglas en inglés (Liskov Substitution Principle) y este principio fue creado con base en la definición de subtipos de Barbara Liskov en 1987, que dice lo siguiente:

Según Roberto Martin, en su libro *Agile principles, patterns, and practices in C* dice: “Si por cada objeto o1 de tipo S, existe un objeto o2 de tipo T y tenemos un programa P en términos de T. Si el comportamiento de P no cambia al sustituir o2 por o1, se dice que S es un subtipo de T”.

En un principio esta definición parece algo confusa, pero se resume en que un subtipo debe ser intercambiable por el tipo base y el comportamiento del sistema debería de continuar funcionando correctamente.

Para explicarlo se analizará las clases rectángulo y cuadrado de la figura 9. Geométricamente un cuadrado es un rectángulo con el ancho y la altura del mismo tamaño.

Figura 9. **Diagrama de clases de figuras geométricas**

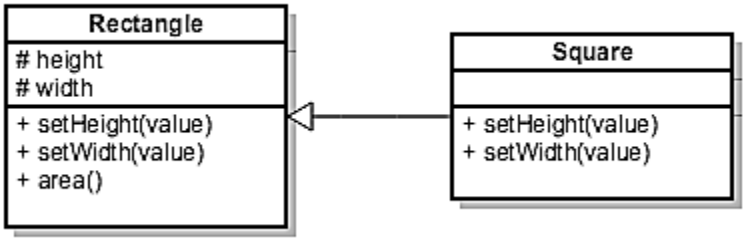


Figura: elaboración propia.

Para satisfacer esta condición se deben sobreescribir los métodos asignadores de valores en la clase cuadrado, para asignar el valor dado tanto al alto como al ancho de la misma.

```
// Square.php

    public function setHeight($value)
        $this->height = $value;
        $this->width = $value;
        public function setWidth($value)
        $this->height = $value;
        $this->width = $value;
...
```

Con esta implementación en la clase Cuadrado se soluciona el problema a través de herencia. Pero qué pasa si con anterioridad se tiene un verificador de Rectángulos con el siguiente código.

```
class AreaValidator
    public function validate($shape)
        $shape->setHeight(5);
        $shape->setWidth(4);

       if($shape->area() != 20) {
              throw new Exception();
        return true;
```

Y se corre el verificador de la siguiente manera.

```
$shape = new Rectangle;
$validator = new AreaValidator;
var_dump($validator->validate($shape));         // return true
```

Se ve que con la clase Rectángulo no existiría ningún problema ya que el área de un rectángulo de 5x4 es 20. Pero si se intercambian los tipos como lo dicta el principio de Sustitución Liskov, se verá que el validador lanza una excepción porque espera un área de 16 con un cuadrado de 4x4.

```
$shape = new Square;
$validator = new AreaValidator;
var_dump($validator->validate($shape));         // return Exception
```

Para poder satisfacer el validador, se debe de hacer un cambio en la clase, realizando una verificación de tipos en el método validar, con lo que el validador quedaría de la siguiente manera.

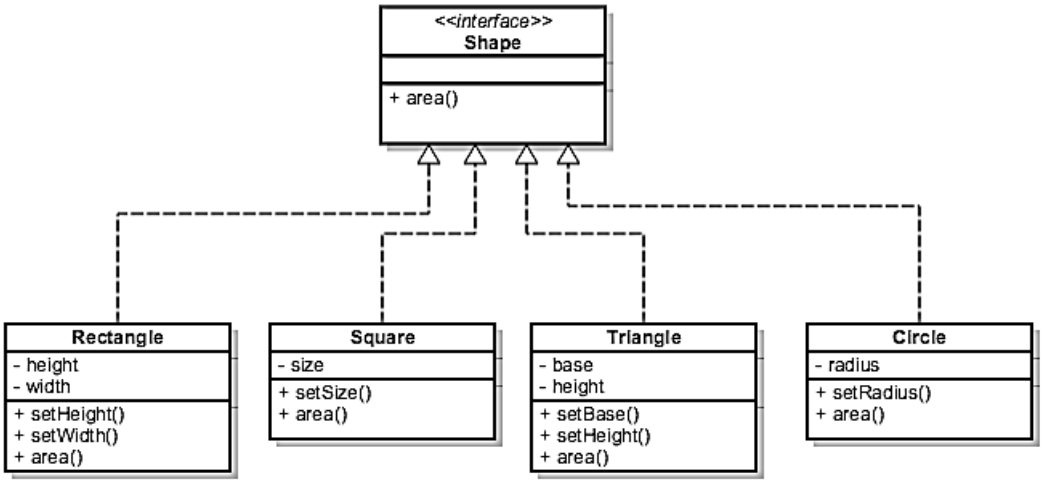
```
class AreaValidator
   public function validate($shape)
      $shape->setHeight(5);
      $shape->setWidth(4);

      if (is_a($shape, 'Square')) {
         if($shape->area() != 16) {
            throw new Exception();
      } elseif (is_a($shape, 'Rectangle')) {
         if($shape->area() != 20) {
            throw new Exception();
      return true;
```

Con este cambio se soluciona el problema, pero se incurre en otra falta, se está violando OCP, al tener que modificar una clase cuando se quiere implementar un nuevo comportamiento.

Se puede concluir con esto, que un rectángulo y un cuadrado son geométricamente equivalentes, pero no desde el punto de vista de la implementación en el sistema, por lo que esta solución no es la indicada si se quiere satisfacer LSP.

Una solución más adecuada se obtendría al favorecer la composición sobre la herencia y diseñar una solución con base en contratos, creando una interfaz de figuras geométricas donde se obligue a cada una a implementar su propio método área.

Figura 10. **Diagrama de clases de figuras geométricas que satisfacen LSP**



Fuente: elaboración propia.

A esto hay que agregar que la interfaz tiene que determinar los tipos de los parámetros de los métodos a implementar, así como el tipo del valor

retornado, por lo que la interfaz de figura geométrica, quedaría de la siguiente manera.

```
// Shape.php
interface Shape
    * Calculate shape area
    * @return float
   public function area();
```

1.2.4. Principio de Segregación de Interfaces (ISP)

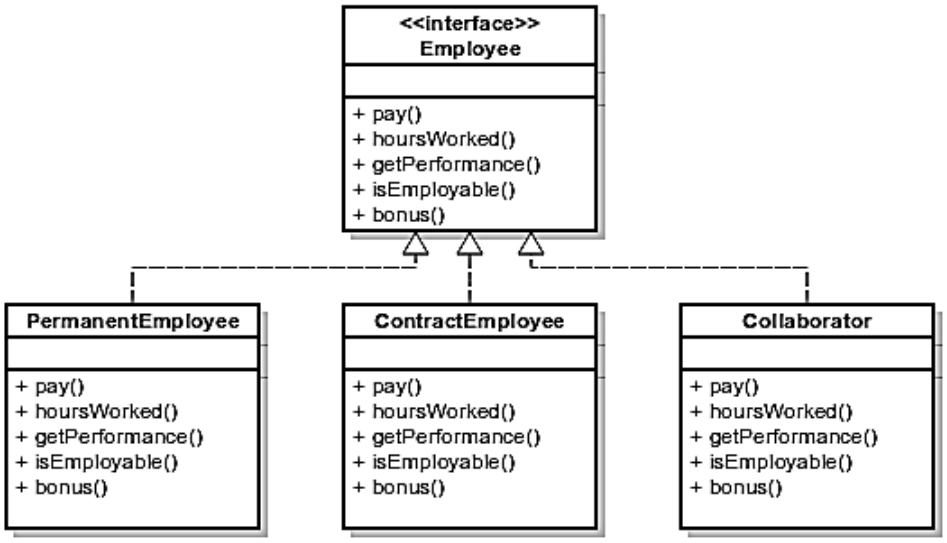
ISP por su siglas en inglés (Interface Segregation Principle) se basa en la premisa de que ningún cliente debe implementar interfaces que lo obliguen a depender de métodos que no utiliza.

Este caso se encuentra a menudo en diseños basados en herencia donde el síntoma es implementar métodos vacíos, sin embargo, también se puede encontrar este tipo de problemas cuando no se hace un buen diseño con base en contratos.

Para cumplir con el principio de segregación de interfaces, es mejor construir interfaces pequeñas y no interfaces grandes que pocas clases implementan en su totalidad.

Si se analiza el diseño de la figura 11, se ve que algunas clases tendrán que implementar métodos vacíos, debido a que lo dicta la interfaz Empleado, pero hay tipos de empleados que no los utilizan.

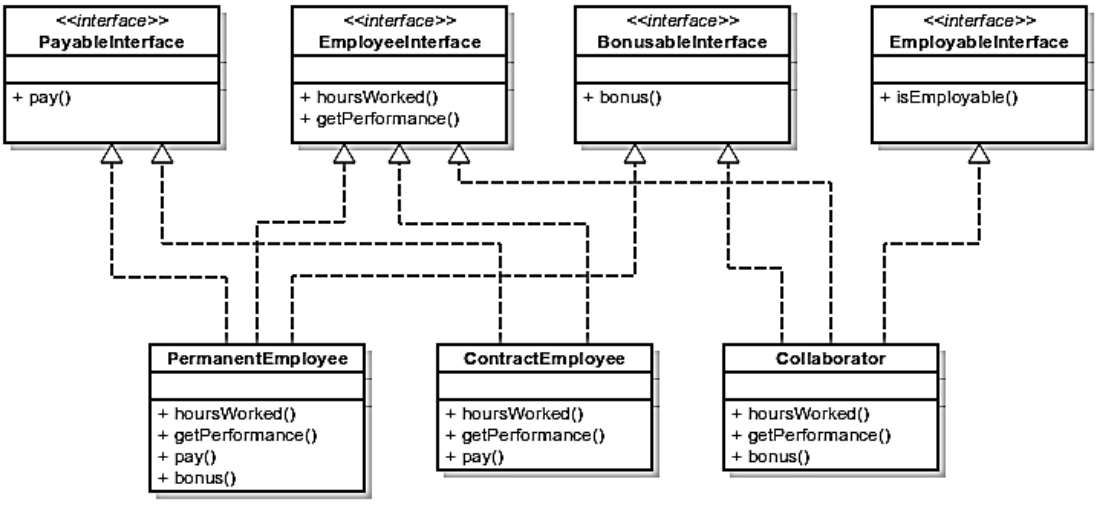
Figura 11. **Violación de ISP al tener que implementar una interfaz muy grande**



Fuente: elaboración propia.

La solución a este problema es segmentar o segregar la interfaz en interfaces más pequeñas, haciendo que las clases de empleados solo implementen los métodos que necesitan para por lograr su comportamiento, por lo que el diseño que cumple con
ISP sería el siguiente.

Figura 12. **Diseño de clases cumpliendo con ISP**



Fuente: elaboración propia.

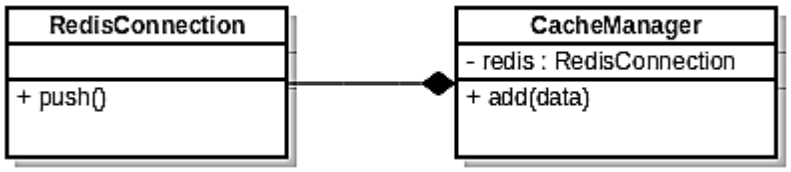
1.2.5. Principio de Inversión de Dependencia (DIP)

También conocida como DIP por sus siglas en inglés (Dependency Inversion Principle) es el último principio de diseño SOLID que se basa en la condición que código de alto nivel no debe depender de código de bajo nivel, en su lugar debe depender de abstracciones y dichas abstracciones no deben depender de detalles.

En concreto dicta que el código de alto nivel debe depender de abstracciones y no de implementaciones. Esto da mucha ventaja en el mantenimiento del código, ya que resulta un diseño débilmente acoplado y fácilmente de reutilizar.

Primero se ve un ejemplo donde se viole el DIP, como lo es el de la figura 13. En este se diseñó un manejador de Caché, el cual es fuertemente acoplado y además obliga a violar también el principio OCP.

Figura 13. **Diseño de sistema de cache que viola DIP**



Fuente: elaboración propia.

Se ve que el código de alto nivel CacheManager, depende de código de bajo nivel, como lo es una implementación de RedisConnection, lo que contradice totalmente el principio DIP, pero peor aún, un cambio en el comportamiento de Caché, implica modificar la clase CacheManager violando así OCP.

Para mejorar este diseño se deben invertir las dependencias y hacer que CacheManager dependa de una Abstracción y que esta no dependa de detalles. El código que satisface estas condiciones es el siguiente.

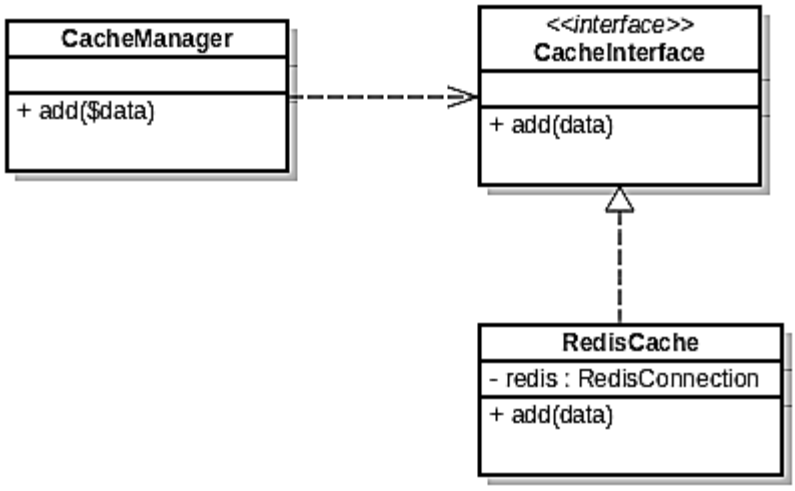
```
// CacheInterface.php
interface CacheInterface
    public function add($data);

// RedisCache.php
class RedisCache implements CacheInterface
    protected $redis;
```

```
    public function __construct(RedisConnection $redis)
        $this->redis = $redis;
    public function add($data)
        return $this->reddis->push($data);

// CacheManager.php
class CacheManager
    protected $cache;
    public function __construct(CacheInterface $cache)
        $this->cache = $cache;
    public function add($data)
        $this->cache->add($data);
```

En el código anterior se observa que CacheManager ahora depende de una abstracción (CacheInterface) y que un cambio de comportamiento por ejemplo, manejar el Caché a través de Memcached en el futuro, no implicaría un cambio en dicha clase, sino que solo se haría la clase correspondiente para el manejo de Memcached, que debe implementar la interfaz CacheInterface y que sea inyectada al Manager para su utilización.

El diagrama de clases de la solución quedaría de la siguiente manera.

Figura 14. **Diseño de caché que cumple con DIP**



Fuente: elaboración propia.

2. DESARROLLO GUIADO POR PRUEBAS

Conocido también como TDD por su siglas en inglés (Test-Driven Development) es un proceso de desarrollo que se basa en un ciclo, el cual consiste en desarrollar pruebas unitarias, codificar y refactorizar pequeñas partes de código.

Producto de este proceso es el código que funciona como sus pruebas lo indican y que al basarse en cumplir con los requerimientos indicados en estas, minimiza la probabilidad de encontrar errores inesperados en los consumidores de este código. En resumen, se podría decir que el código resultante hace lo que debería de hacer con base en la especificación que se le estableció en las pruebas.

Siguiendo el orden de este proceso, diseñando pruebas antes de empezar la codificación, se consigue algo más importante, una herramienta de diseño y documentación, antes que un sistema de verificación.

Pero antes de profundizar en el desarrollo guiado por pruebas, se define que son estas pruebas, así como sus características y ventajas.

2.1. Pruebas unitarias

Las pruebas unitarias son pruebas que toman un pequeño segmento de código y aislándolo del resto, define el comportamiento que este debe de tener, de tal manera que se prueba individualmente cada segmento, antes de integrarlo al resto de la aplicación, asegurando su calidad individual.

Las pruebas unitarias son relativamente fácil de implementar, al enfocarse en una sola unidad de trabajo. No se necesitsa saber detalles de otros componentes de la aplicación y se dedica todos los esfuerzos a un simple requerimiento.

2.1.1. Características

Debido a su estructura y comportamiento, las pruebas unitarias, comparten ciertas características.

- Todas las pruebas unitarias aíslan el código evaluado del resto de la aplicación.
- Son independientes entre otros desarrolladores del equipo, es decir que al ser autónomas, no dependen de código realizado por otros.
- Son enfocadas a un solo requerimiento.
- Su funcionamiento se puede predecir.
- Son independientes, por lo que podrían ejecutarse en paralelo y no necesitan de información manejada en otras pruebas unitarias.

2.1.2. Beneficios

A su vez, el uso de utilizar pruebas unitarias en el desarrollo de pequeños segmentos de código, tiene beneficios bien definidos.

- Reduce la probabilidad de tener errores de código en producción.

- Debido a que se enfoca en una sola unidad de trabajo, puede ahorrar tiempo de desarrollo.

- Tener un conjunto de pruebas unitarias da la confianza de modificar código en búsqueda de la reutilización y mantenimiento, garantizando que no se arruine ninguna funcionalidad definida por dichas pruebas.

- Al definir el comportamiento de las distintas secciones de la aplicación, crean un repositorio de documentación, que describe lo que hace en su totalidad.

Si se considera la división como una unidad de un sistema mayor como lo es una calculadora, se podría desarrollar la siguiente prueba unitaria para definir parte de su comportamiento. Para su ejemplificación se utilizará un *framework* de pruebas unitarias llamado PHPUnit que se explicará a detalle más adelante.

```
// DivisionTest.php
<?php
class DivisionTest extends PHPUnit_Framework_TestCase {
    public function test_returns_3_when_divides_6_by_2()
    {
        $number1 = 6;
        $number2 = 2;

        $division = new Division();
        $result = $division->execute($number1, $number2);

        $this->assertEquals(3, $result);

    /** more tests */
```

La prueba unitaria define claramente que retornará 3 cuando se divida 6 entre 2. Si se interpreta el código de esta prueba, se puede leer con claridad lo siguiente:

Dados los números 6 y 2, cuando se crea una instancia de la clase División y se le mandan como parámetros a su método ejecutar, su resultado se espera que sea igual a 3

Esto define claramente el funcionamiento del método ejecutar de la clase división y el código que implementará este método debería de cumplir este requerimiento.

Otra prueba unitaria que podría agregarse a esta clase podría ser la siguiente, en la que se ve claramente que debería retornar 2 cuando se divide 4 entre 2.

```
// DivisionTest.php
<?php
// …
    public function test_returns_2_when_divides_4_by_2()
        $number1 = 4;
        $number2 = 2;

        $division = new Division();
        $result = $division->execute($number1, $number2);

        $this->assertEquals(2, $result);
```

Una vez más, establece un requerimiento puntual que se interpretaría como Dado los números 4 y 2, cuando a una instancia de la clase División se

les manda como parámetro a su método ejecutar, su resultado se esperaría que fuera 2.

Por último para definir el caso especial de la División, donde no está definida si se divide cualquier número entre cero, podría crearse la siguiente prueba unitaria.

```
// DivisionTest.php
<?php
     * @expectedException     Exception
    public                                                          function
test_returns_an_exception_when_divides_a_number_by_zero()
        $number1 = 5;
        $number2 = 0;

        $division = new Division();

        $result = $division->execute($number1, $number2);
```

El nombre de la prueba define claramente que retorna una excepción cuando se divide un número entre cero y más importante aún si se interpreta el código de ésta, se puede leer que Se espera una excepción tipo Exception, dado que los números 5 y 0 (en ese orden) se le mandan como parámetros al método ejecutar de la instancia de la clase División.

Las pruebas unitarias más que una garantía de funcionamiento, son una guía de los requerimientos del sistema y al final de su implementación queda un gran sistema de documentación, el cual de manera comprobable, específica lo que realiza la aplicación en su totalidad.

2.1.3. Objetos simulados

En el desarrollo de pruebas unitarias se necesita en ciertas ocasiones imitar objetos reales, a estos objetos se les conoce como objetos simulados pero más popularmente como *mocks*.

Los *mocks* ayudan a simplificar escenarios en una prueba unitaria, debido a que la prueba está enfocada a un método específico y este, bien puede utilizar otro objeto para conseguir cierta información. Este segundo objeto es un candidato a ser un *mock* en la prueba unitaria, ya que no nos interesa como funciona internamente, solo se necesita que sea llamado en el método que realmente se está probando.

Si se imagina la prueba unitaria de un método que convierte dólares a quetzales, consiguiendo el tipo de cambio del dólar frente al quetzal a través de un API externa y luego retorna lo equivalente en quetzales de la cantidad en dólares que pasaran como argumento al método, se puede observar que un objeto externo interactuará dentro del método que se necesita probar, de tal manera que es candidato a convertirse en objeto simulado o *mock* dentro de la prueba, porque este objeto externo ya tiene un conjunto de pruebas que se realizaron cuando fue desarrollado.

La prueba unitaria en PHPUnit del ejemplo anterior quedaría de la siguiente utilizando *mocks*.

```
// ExchangeServiceTest.php
<?php
use ExchangeService;

class ExchangeServiceTest extends PHPUnit_Framework_TestCase {
```

```
public function test_gets_exchange_rate_from_dollar_to_quetzal()
{
    $mockAPI = $this->getMockBuilder('GuatemalaBankAPI')
                    -> getMock();
    $mockAPI->expects($this->once())
            ->method('dollarRate')
            ->willReturn('7.70');

    $service = new ExchangeService($mockAPI);
    $result = $service->dollarToQuetzal(10);

    $this->assertEquals(77, $result);
```

En la prueba se puede observar que se hace un *mock* del API del Banco de Guatemala, donde se le indica que debe llamar una vez al método ‘dollarRate’ y este debe devolver ‘7,70’ simulando su comportamiento. De tal manera que el método ‘dollarToQuetzal’ que es el que realmente se está probando, debe retornar un resultado de 77 al pasarle por parámetro la cantidad de 10 dólares.

La implementación final de la clase que dará solución a esta prueba se puede ver a continuación, donde se observa claramente que el método ‘dollarRate’ de la clase del API del Banco de Guatemala es llamado para satisfacer la prueba realizada, junto con la realización del producto entre el valor devuelto por el API y el valor enviado como parámetro correspondiente a la cantidad de dólares por convertir.

```
// ExchangeService.php
<?php

use GuatemalaBankAPI;
```

```
class ExchangeService {
    protected $api;

    public function __construct(GuatemalaBankAPI $api) {
        $this->api = $api;

    public function dollarToQuetzal($amount)
        return $this->api->dollarRate() * $amount;
```

2.1.4. Otro tipo de pruebas

Este trabajo está limitado al desarrollo guiado por pruebas unitarias, pero existen otros tipos de pruebas que pueden y deben desarrollarse, para asegurar aún más la calidad de la aplicación.

Uno de esos tipos de pruebas son las pruebas de integración, las cuales se realizan entre módulos. Estas definen cómo módulos completos deberían de comportarse de manera global y no poniendo atención a los detalles internos, más bien solo se enfoca en los parámetros de entrada y el resultado del proceso.

Otro tipo de pruebas son las pruebas de aceptación. Estas se enfocan en cómo los requerimientos deben ser vistos en producción y se basan por lo general en interfaces de usuario. Muchas veces estas son validadas por el dueño del proyecto, para ver si el resultado cumple con los requerimientos que dieron origen al sistema.

En sí existen muchos otros tipos de pruebas, que evalúan el sistema desde distintos puntos de vista, pero en conjunto logran que se desarrolle un producto de mucha calidad y sobre todo apegado a sus requerimientos.

2.2. Ciclo del desarrollo guiado por pruebas

Como su nombre lo indica, el desarrollo guiado por pruebas significa que las pruebas guíen el desarrollo, que indiquen el camino a seguir en la elaboración del sistema. Esta es la mayor dificultad que tiene este proceso, romper con el paradigma de iniciar el desarrollo cuanto antes y dejar las pruebas como sistema de verificación, al final del mismo.

Ya se explicó anteriormente lo que implica en diseño y documentación el desarrollar las pruebas previo al desarrollo de software, pero ¿cómo asegurar que se está llevando el desarrollo en el sentido correcto? Antes de contestar esta pregunta, se deben ver las reglas que Robert C. Martin sugiere para enfocar en pequeñas unidades del sistema.

- No escribir código hasta que se haya escrito una prueba unitaria que falle.

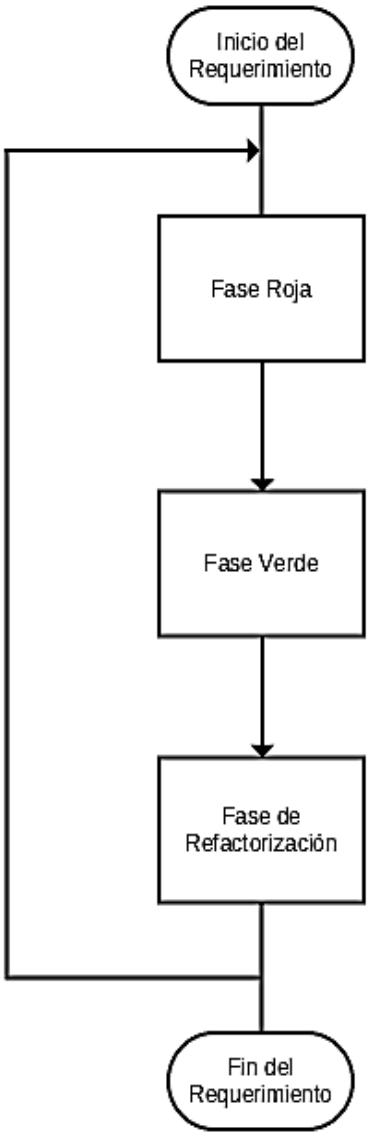
- No escribir más de una prueba unitaria que es suficiente para fallar.

- No escribir más código que el suficiente para que pase, la prueba que falla.

Estas reglas ya dan una idea del camino a tomar, y si se pone a pensar en las implicaciones de su uso, se ve como el enfoque es total a un pequeño requerimiento. La solución es más obvia debido a que se aisla del resto de la aplicación y se codifica únicamente lo que se necesita.

Pero aún más claro queda el camino a tomar, siguiendo el ciclo del desarrollo guiado por pruebas, el cual se puede observar en la siguiente figura.

Figura 15. **Ciclo del desarrollo guiado por pruebas**



Fuente elaboración propia.

Un requerimiento pequeño debe de pasar varias iteraciones por las fases roja - verde - refactorización, hasta que se implemente en su totalidad.

2.2.1. Fase roja

Esta fase es muy importante ya que es la que da inicio a la iteración, la que con previo análisis del requerimiento va a dar lugar al diseño que lo solucionará, va a ser la guía de trabajo a corto plazo y será el material de verificación, el cual indicará si se cumple o no con el objetivo.

Cuando se crea una prueba unitaria, se debe pensar en el funcionamiento que se quiere implementar y diseñar el mecanismo que indicará que dicho funcionamiento es satisfecho en su totalidad. Seguramente después de escribir la primer prueba unitaria, se harán más diagramas, más anotaciones, que ayudarán a diseñar las próximas pruebas con base en las variantes del funcionamiento que se quieren implementar.

Estar en la fase roja no es sinónimo de estar equivocado, de estar haciendo mal las cosas, es la fase que indica un punto de partida, un punto en el cual no queda otra opción que movernos hacia la fase verde.

2.2.2. Fase verde

Teniendo la prueba unitaria en rojo se debe empezar a codificar. Si, es hasta ahora, después de analizar el requerimiento y diseñar una solución, que es el momento de crear el código que dará solución a esa pequeña unidad del sistema. Es importante que el código que se implemente satisfaga la prueba unitaria en cuestión y las pruebas escritas anteriormente.

Es importante mantenerlo simple y solo escribir el código suficiente para que la prueba pase. No se debe pensar en futuros escenarios, se debe limitar solo a la prueba unitaria que hizo que el estado esté en rojo y cambiarlo a

verde. Futuros escenarios se derivan en más iteraciones sobre dicho requerimiento y por lo consiguiente deberán de pasar por todas las fases del ciclo.

2.2.3. Fase de refactorización

Después de cambiar a verde el estado de la prueba, se podria decir que la iteración está finalizada, pero eso es solo desde la perspectiva de que el funcionamiento del sistema sea implementado. Pero, ¿qué hay sobre el mantenimiento de dicho código?

Si, se debe considerar la posibilidad de que el sistema crezca, que el código sea reutilizado por otros desarrolladores o simplemente cambie el requerimiento. Es por eso que se debe darle importancia a la legibilidad y mantenibilidad del código.

Esta fase es la indicada para hacer todos esos cambios, que harán que elcódigo sea fácil de entender, modificar y reutilizar. Es aquí donde toman importancia los conceptos sobre diseño, vistos en el capítulo anterior y permite que con poco código implementado, se pueda buscar satisfacer todas las reglas de diseño que indican dichos principios.

Después de estas modificaciones, se está listo para finalizar el requerimiento o simplemente empezar el ciclo de nuevo desde la fase roja y escribir otra prueba unitaria que describe un escenario alterno del requerimiento en cuestión. A estos escenarios alternos se les conoce como triangulación y hace que con base en varias pruebas unitarias se llegue a una solución globalizada que cumpla con el requerimiento en su totalidad.

2.3. Ejemplo del ciclo del desarrollo guiado por pruebas

Para ejemplificar cada una de estas fases, se desarrollará el requerimiento anterior de implementar la operación división de una calculadora, esta vez pasando fase por fase para darle solución.

- Iteración 1

En este momento no se ha escrito nada y solo se tiene el requerimiento en sí, que es implementar la operación división de una calculadora. Empezando en la fase roja y lo primero que se tiene que hacer el primer *test* para que falle.

```
// DivisionOperationTest.php
<?php
class DivisionOperationTest extends PHPUnit_Framework_TestCase {
    public function test_returns_3_when_divides_6_by_2()
        $number1 = 6;
        $number2 = 2;

        $division = new DivisionOperation();
        $result = $division->execute($number1, $number2);

        $this->assertEquals(3, $result);
```

En la prueba unitaria anterior, se define que dado que se tienen los números 6 y 2, si se envian como parámetros al método ejecutar de la instancia de la clase operación división, se espera que dicho resultado sea igual a 3.

Si se corre la prueba unitaria se verá que da un error, ya que la clase Operación División, aún no existe.

```
PHP Fatal error: Class 'DivisionOperation' not found in
/tests/DivisionOperationTest.php on line 9
```

Esto indica que primero se debe crear dicha clase. Es aquí donde las pruebas empiezan a guiar para la implementación del requerimiento. Se crea la clase y se implementa lo mínimo para pasar la prueba unitaria a verde.

```
// DivisionOperation.php
<?php
class DivisionOperation {
    public function execute($number1, $number2)
        return 3;
```

Con esto se implementa lo mínimo para pasar la prueba unitaria y se esta listos para pasar de la fase verde a la fase de refactorización. En esa iteración no hay nada que refactorizar.

- Iteración 2

Ya en la fase de refactorización, se debe pasar de nuevo a la fase roja agregando una nueva prueba unitaria, esta vez la prueba unitaria intenta dividir 4 entre 2 y espera un resultado de 2.

```
// DivisionOperationTest.php
<?php
class DivisionOperationTest extends PHPUnit_Framework_TestCase {
    public function test_returns_2_when_divides_4_by_2()
        $number1 = 4;
        $number2 = 2;

        $division = new DivisionOperation();
```

```
        $result = $division->execute($number1, $number2);

        $this->assertEquals(2, $result);
```

Debido a cómo está implementado el método hasta el momento, la prueba falla indicándo que espera un valor de 2 pero recibe un valor de 3.

```
There was 1 failure:

1) DivisionOperationTest::test_returns_2_when_divides_4_by_2
Failed asserting that 3 matches expected 2.
```

Se necesita ahora cambiar el contenido del método ‘ejecutar’, con solo el código necesario, para que pase, tanto la última como la primera prueba unitaria que se escribe. Por lo que la clase quedaría de la siguiente manera.

```
// DivisionOperation.php
<?php
class DivisionOperation {
    public function execute($number1, $number2)
    {
        if ($number1 == 4) {
            if ($number2 == 2) {
                return 2;
            } else {
                return 3;
        } else {
            return 3;
```

Con este código se puede pasar de la fase roja a la verde pero se ve que es demasiado complejo y puede ser refactorizado, por lo que se hacen algunos cambios y se debe asegurar que estos, sigan dejando en la fase verde.

```
// DivisionOperation.php
<?php
class DivisionOperation {
    public function execute($number1, $number2)
        if ($number1 == 4 && $number2 == 2) {
            return 2;
        return 3;
```

- Iteración 3

Con la refactorización anterior, el método ya es más simple y entendible. Por lo que se agrega otra prueba unitaria con otro posible escenario de la división. Esta vez se espera un resultado de 2 cuando se divide 10 entre 5.

```
// DivisionOperationTest.php
<?php
class DivisionOperationTest extends PHPUnit_Framework_TestCase {
// ...
    public function test_returns_2_when_divides_10_by_5()
        $number1 = 10;
        $number2 = 5;

        $division = new DivisionOperation();
        $result = $division->execute($number1, $number2);

        $this->assertEquals(2, $result);
```

Al correr las pruebas, la última falla debido a que espera un resultado de 2 pero en su lugar recibe un 3.

```
There was 1 failure:
```

```
1) DivisionOperationTest::test_returns_2_when_divides_10_by_5
Failed asserting that 3 matches expected 2.
```

Estando en la fase roja de nuevo, ahora se escribe una nueva implementación del método ‘ejecutar’, pero que satisfaga las 3 pruebas escritas hasta el momento. Por lo que quedaría de la siguiente manera.

```
// DivisionOperation.php
<?php
class DivisionOperation {
    public function execute($number1, $number2)
        return $number1 / $number2;
```

Esta solución ya se acerca a la versión final del método. Se trata de refactorizar y se ve que está en óptimas condiciones, así que se mueve a la fase roja una vez más, escribiendo una prueba más para cumplir con el requerimiento.

- Iteración 4

Se agrega la prueba en el caso el dividendo sea cero y se espera que el método lance una excepción.

```
// DivisionOperationTest.php
<?php
class DivisionOperationTest extends PHPUnit_Framework_TestCase {
     * @expectedException Exception
    public                                                    function
test_throws_an_exception_when_second_number_is_zero()
        $number1 = 10;
        $number2 = 0;
        $division = new DivisionOperation();
```

```
        $result = $division->execute($number1, $number2);
```

Al ejecutar las pruebas de nuevo lanza un error, debido a que no se ha lanzado una excepción y en su lugar notifica un error al tratar de dividir una cantidad entre cero.

```
There was 1 error:

1) DivisionOperationTest::test_throws_an_exception_when_second_number_is
   _zero
Division by zero
```

Por lo que una vez más se debe modificar el método ‘ejecutar’ para satisfacer las 4 pruebas unitarias y pasar a la fase verde una vez más.

```
// DivisionOperation.php
<?php
class DivisionOperation {
    public function execute($number1, $number2)
        if($number2 == 0) {
            throw new \Exception("Division by zero is not defined");

        return $number1 / $number2;
```

Esta sería la versión final del método ‘ejecutar’ de la clase Operación División. Se ve que no hay nada que refactorizar y que cumple con cada una de las pruebas unitarias escritas.

Como se ve el ciclo del desarrollo guiado por pruebas hace pasar por cada una de sus fases una y otra vez, indicando cual es el siguiente paso hasta llegar

a una versión óptima de la implementación, donde se cumple con cada una de las pruebas diseñadas.

Como resultado se obtiene un código muy bien probado, mucho antes que los consumidores hagan uso de él y un repositorio de documentación que describe claramente su funcionamiento.

3. DESARROLLO GUIADO POR PRUEBAS EN PHP

Para implementar el desarrollo guiado por pruebas en PHP se deben conocer los *frameworks* de pruebas unitarias más utilizados que existen, estos son PHPUnit y PHPSpec. Se analizaran individualmente cada uno de estos *frameworks*, evidenciando sus fortalezas y debilidades a la hora de implementar dichas pruebas.

3.1. *Frameworks* de pruebas unitarias

Es una forma de comprobar el correcto funcionamiento de un módulo de código. Esto sirve para asegurar que cada uno de los módulos funcione correctamente por separado.

3.1.1. PHPUnit

Como lo define su creador, PHPUnit es un *framework* para programación orientada a pruebas. Es una instancia de xUnit que fue originada por SUnit, utilizado para el mismo propósito en Smalltalk y popularizado por JUnit que se utiliza en Java.

En este capítulo no se intenta hacer un manual de PHPUnit por lo que se enfocará en ciertos aspectos importantes para su utilización.

- Instalación y utilización

Hoy en día Composer es quizás el administrador de paquetes y dependencias, más utilizados en PHP. PHPUnit permite manejar su instalación a través de Composer a través del archivo composer.json, como se ve a continuación

```
// composer.json
"require-dev": {
        "phpunit/phpunit": "~4.0",
```

Luego se ejecuta Composer update en el directorio donde se encuentra el archivo composer.json y se descargará la versión 4.0 PHPUnit, junto con sus dependencias y estará listo para utilizarse.

Para utilizarse basta con realizar la primer prueba unitaria como por ejemplo

```
// ExampleTest.php
<?php
class ExampleTest extends PHPUnit_Framework_TestCase {
    public function test_example()
    {
        $this->assertTrue(true);
```

Por último para ejecutar la prueba unitaria solo se corre en consola, el ejecutable de PHPUnit, que se encuentra en la carpeta vendor/bin/phpunit (ubicación destinada por composer) e indicar el archivo que contiene la prueba unitaria.

```
vendor/bin/composer ExampleTest.php
```

Lo que produce la siguiente salida indicando que todos las pruebas pasaron satisfactoriamente.

```
PHPUnit 4.8.9 by Sebastian Bergmann and contributors.
.
Time: 1.31 seconds, Memory: 4.50Mb
OK (1 tests, 1 assertions)
```

- API

El API (Interfaz de programación de aplicaciones por sus siglas en inglés) de los *frameworks* de pruebas unitarias está definida en su mayoría por las afirmaciones e imitaciones, más conocidas como *asserts* y *mocks* respectivamente.

PHPUnit contiene un gran repositorio de *asserts*, permitiendo comparar si 2 cadenas son iguales, si un valor es verdadero o falso, si un valor se encuentra en un arreglo, entre otros. Con estos *asserts* se puede llegar a hacer comparaciones de casi cualquier tipo de dato y sobre todo comparar el valor esperado con el valor retornado por el método evaluado, para así determinar si es el valor correcto.

Ejemplos de *asserts* son los siguientes.

```
// ExampleTest.php
<?php
class ExampleTest extends PHPUnit_Framework_TestCase {
    public function test_example()
        $this->assertTrue(true);
        $this->assertFalse(false);
        $this->assertInstanceOf(‘Calculator’, new Calculator);
```

```
        $this->assertEquals(1, 1);
        $this->assertContains(4, array(1,3,4));
        $this->assertEmpty(array());
        $this->assertCount(1, array('apple'));
        $this->assertNull(NULL);
```

Todos estos *asserts* reciben el valor esperado por lo que pasarán como satisfactorios, mostrando como resultado una salida como la siguiente:

```
PHPUnit 4.8.9 by Sebastian Bergmann and contributors.
Time: 1.31 seconds, Memory: 4.50Mb
OK (1 tests, 8 assertions)
```

Cada prueba unitaria está representada por un punto "." y al final se muestra un resumen de la cantidad de pruebas y *asserts* que pasaron o fallaron, en esta ocasión 1 prueba y 8 afirmaciones. En la documentación del API de PHPUnit se pueden ver a detalle todos los tipos de *asserts* disponibles.

Mocks

Como se vio en el capítulo de pruebas unitarias, los *mocks* son un componente importante para desarrollarlas. Esto permite realizar pruebas en aislamiento y no depender de objetos externos al método al que se le están realizando pruebas. PHPUnit permite hacer *mocks* fácilmente con la siguiente sintaxis.

```
// SomeTest.php
    public function test_using_mocks()
        $mock = $this->getMockBuilder('SomeClassOrInterface')
                     ->getMock();
        $mock->expects($this->once())
```

```
        ->method('someMethod')
        ->willReturn('someValue');
```

PHPUnit cuenta con un conjunto de métodos que hace aún más fácil el uso de *mocks* en pruebas unitarias en casi todos los escenarios, que incluye hacer pruebas de Traits, Clases Abstractas, Servicios Web y Sistemas de Archivos.

3.1.2. PHPSpec

Otro *framework* para pruebas unitarias para PHP es PHPSpec. Ellos se definen como SpecBDD, lo que vendría siendo como un desarrollo guiado por comportamientos con base en especificaciones, por sus siglas de su nombre en inglés, Spec Behaviour Driven Development. Este enfoque nace de las bases del desarrollo guiado por pruebas TDD, en el cual se escribe siempre la prueba al inicio, en sí a la prueba se le da el nombre de especificación. Es decir, se escribe una especificación de la clase a desarrollar.

El enfoque es por más interesante debido a que se le da el toque semántico al desarrollo, la sintaxis de PHPSpec hace que la prueba unitaria sea aún más entendible por un ser humano, al utilizar una sintaxis muy parecida a como se habla comúnmente el idioma inglés.

Instalación y utilización

El proceso de instalación, al igual que PHPUnit, se puede realizar a través de composer. Simplemente se crea el archivo como se muestra a continuación, se ejecuta composer update y ya se está listo para trabajar con PHPSpec.

```
// composer.json
```

```
    "require-dev": {
        "phpspec/phpspec": "~2.0"
```

La forma de utilización, junto con la sintaxis, son las mayores diferencias frente a PHPUnit y es que realmente PHPSpec guía a través de las pruebas unitarias, a partir de ahora llamadas especificaciones. Para crear la primera especificación no se tiene que crear ningún archivo sino desde la terminal ejecutar el siguiente comando y él creará la clase de la especificación.

```
$ phpspec describe Example

Specification for Example created in  ~/spec/ExampleSpec.php.
```

Siguiendo sin crear o modificar aún nada, se corre el siguiente comando y él se ofrecerá en crear la clase que implementará la solución.

```
$ phpspec run
Example
  10  - it is initializable
      class Example does not exist.

                                    100%                                    1
1 specs
1 example (1 broken)
46ms


  Do you want me to create `Example` for you?
                                                                  [Y/n]
```

Se le indica que sí y él crea la clase. Por último hay que indicarle cómo serán incluidos los archivos, para ello se hace desde el composer.json, mediante psr-4 como se muestra a continuación:

```
//composer.json
    "require-dev": {
        "phpspec/phpspec": "~2.0"
    "autoload": {
        "psr-4": {
            "": "src"
```

Listo, con esto ya se puede empezar a diseñar la especificación. Las especificaciones irán en la carpeta spec y el código implementado, en la carpeta src por defecto en el proyecto. Después de los comandos anteriores el sistema de archivos quedará de la siguiente manera.

```
    Project
├──spec
│     ├──ExampleSpec.php
├──src
│     ├──Example.php
```

- API

Como ya se mencionaba anteriormente, el API de PHPSpec da ese toque semántico en la especificación haciéndola más legible. Los *asserts* en PHPSpec son conocidos como *matchers* o coincidencias. A continuación se ven algunos *matchers*.

```
// ExampleSpec.php
<?php
```

```
namespace spec;

use PhpSpec\ObjectBehavior;
use Prophecy\Argument;

class ExampleSpec extends ObjectBehavior
    function it_is_initializable()
        $this->shouldHaveType('Example');

    function it_shows_some_matchers()
        $this->add(2,3)->shouldBe(5);
        $this->inLetters(5)->shouldBeEqualsTo('five');
        $this->subtract(3,2)->shouldReturn(1);
        $this->subtract(3,2)->shouldBeLike('1');
        $this->totalOperands([3,4,1,2])->shouldHaveCount(4);
        $this->shouldThrow('\Exception')->during('divide', array(5, 0));
```

En el ejemplo se ve como los *matchers* hacen que sea una lectura casi gramatical de la especificación que se está haciendo. Por último si, se ejecuta la especificación con la solución completada, nos muestra el resumen de las pruebas, como se muestra a continuación.

```
                                    100%                                    6

1 specs
2 examples (6 passed)
355ms
```

Objetos simulados

PHPSpec implementa las herramientas necesarias para utilizar objetos simulados en su API. Lo puede realizar de dos maneras, a través de dobles,

objetos que son llamados y devuelven un único valor, o a través de *mocks*, los que se puede modificar su comportamiento a través de la prueba.

Ejemplos de objetos simulados son los siguientes, asumiendo que son pruebas en diferentes clases.

```
 SomeSpec.php
 function              it_sends_the_message(Message              $message)
$message->getContents()->willReturn('abc');
$this->send($message);
OtherSpec.php
 // …
 function       it_gets_three_random_numbers(RandomGenerator       $rand)
$rand->generate()->willReturn(123,               432,               874);
$this->getNumbers($rand,      3)->shouldReturn([123,      432,      874]);

 //OneMoreSpec.php
 function              it_sends_the_message(Message              $message)
$message->getContents()->shouldBeCalled()->willReturn('abc');
$this->send($message);
```

4. PROCESO DE IMPLEMENTACIÓN DE UN NUEVO REQUERIMIENTO UTILIZANDO DESARROLLO GUIADO POR PRUEBAS EN PHP

En capítulos anteriores se ha hablado de los principios de diseño a considerar cuando se formula una solución, de las fases que conlleva el desarrollo guiado por pruebas, así como las herramientas que se tienen disponibles para realizar pruebas unitarias en PHP. En el presente capítulo se unificarán todos estos conceptos, para implementar la solución a un requerimiento completo dentro de una aplicación.

4.1. Definición del requerimiento

Se utilizan como datos de entrada en la etapa de diseño del producto. Establecen qué debe hacer el sistema, pero no cómo hacerlo.

4.1.1. Asunción

Para enfocarse en el nuevo requerimiento, se debe asumir que ya se desarrolló parte de una aplicación que lleva el control de trabajadores de una empresa. Específicamente se asume que se tiene implementada la clase que lleva la información del Empleado y los servicios de ISR y Préstamos.

- Employee
- IsrService
- LoanEmployeeService

4.1.2. Requerimiento

En la empresa se tienen dos tipos de empleados, por contrato y por planilla. Cada uno de estos tipos tienen distintas variables para calcular su pago mensual.

Se necesita un módulo para calcular el salario que devenga mensualmente tanto un empleado en planilla como uno por contrato. Siendo la distribución de la siguiente manera:

- Empleado por contrato

Este tipo de empleado recibe mensualmente su sueldo base menos el 5 % del ISR y si este solicitó algún préstamo a la empresa, se le descontará la cuota mensual de dicho préstamo.

- Empleado en planilla

Un empleado en planilla mensualmente recibe su sueldo base, menos el 3 % del Igss, además se le resta la cantidad proyectada del ISR y si tiene algún préstamo con la empresa, se le debita la cuota respectiva.

4.2. Análisis y diseño

Partiendo que actualmente están desarrollados ya los prerrequisitos para poder implementar el nuevo requerimiento, se puede asumir que ya se posee la clase empleado, que cuenta con los datos individuales del empleado y además implementa la siguiente interfaz.

```
// EmployeeInterface.php
<?php
namespace Payroll\Employee;

interface EmployeeInterface
     * Get employee type. Could be Regular or Contract
     *
     * @return string
    public function type();

     * Get base salary.
    public function getBaseSalary();
```

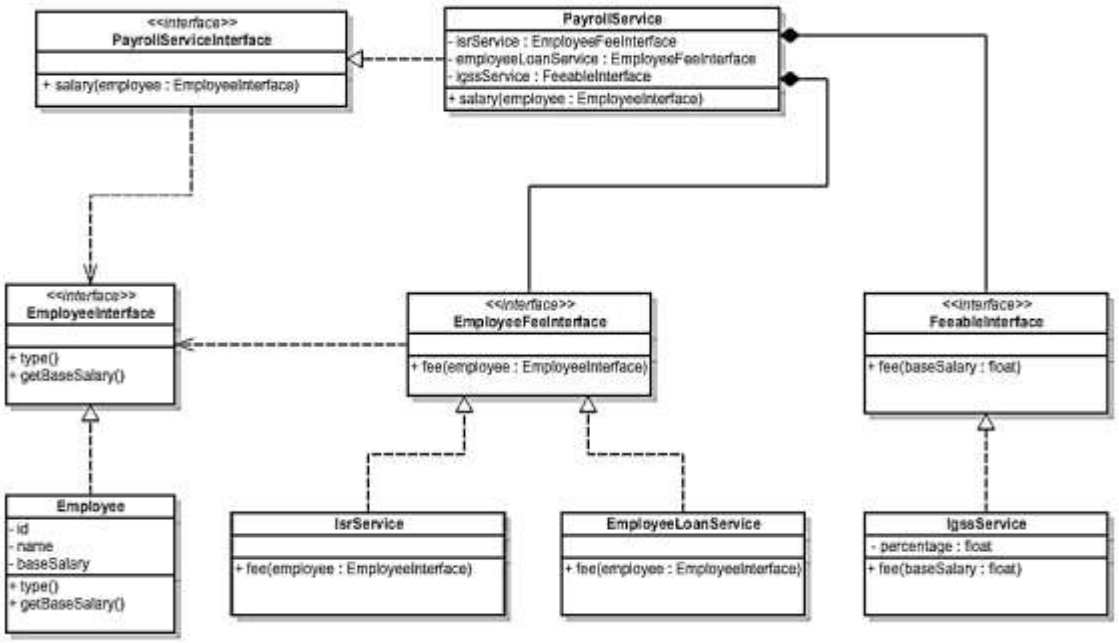
También se cuenta con la clase de servicios del ISR, la cual está encargada de proporcionar la cuota mensual de ISR que debe pagar el empleado en cuestión. Implementa la siguiente interfaz.

```
// EmployeeFeeInterface.php
<?php
namespace Payroll\FeeService;

use Payroll\Employee\EmployeeInterface;

interface EmployeeFeeInterface
     * Get fee for specific employee.
     *
     * @param  \Payroll\Employee\EmployeeInterface $employee
     * @return float
    public function fee(EmployeeInterface $employee);
```

Y por último se tiene la clase de Servicio de préstamos, que maneja las cuotas de los préstamos que tienen los empleados en la empresa, implementando la misma interfaz.

Con base en lo anterior, para poder cumplir con el requerimiento se debe crear una clase que será encargada de llevar el control de las cuotas del servicio del IGSS y una más que a través del patrón de diseño inyección de dependencia, utilice las otras clases existentes para calcular el salario de un empleado. El diseño resultante de la aplicación sería el siguiente.

Figura 16. **Diagrama de clases para la implementación del requerimiento de cálculo de salario**



Fuente: elaboración propia.

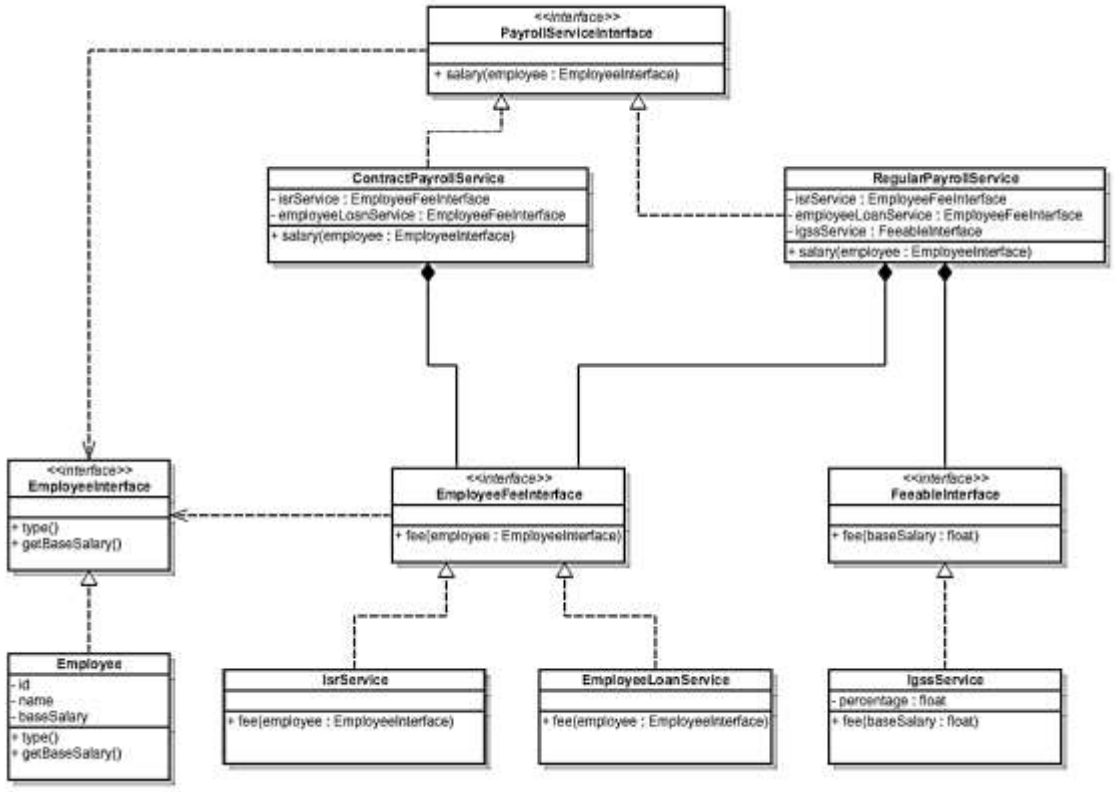
Como se ve en el diagrama la solución incluye la clase de servicio de IGSS y la del servicio de nómina.

Cuestionando un poco esta solución frente a los principios de diseño SOLID vistos en el capítulo 1, se ve que la clase de servicio de nómina (PayrollService) tiene más de una responsabilidad, violando SRP. Es encargada de calcular el salario tanto para un empleado en planilla como un empleado por contrato. Esto lleva al siguiente cuestionamiento, qué pasa si cambia las reglas del negocio y cambia el modo en que se calcula el salario de cada tipo de empleado, o si aparece en el futuro un nuevo tipo de empleado.

Por las razones anteriores, se ve que la clase de servicio de nómina también viola OCP, por lo que es más conveniente dividirla y crear un servicio de nómina por cada tipo de empleado que se tiene, en este caso son 2 pero la solución queda abierta para que se integren más tipos de empleado en el futuro.

El diseño final, el cual toma en cuenta estas consideraciones y todo los principios SOLID quedaría de la siguiente manera.

Figura 17. **Diagrama de clases de la solución cumpliendo con los principios de diseño SOLID**



Fuente: elaboración propia.

4.3. Desarrollo guiado por pruebas

.

Como se ve en la sección anterior, antes de empezar a programar la solución se debe de hacer un fuerte análisis del problema, de lo que se tiene y proponer un diseño con base en eso y luego cuestionarlo frente a los principios de diseño aprendidos.

Si se analiza el diseño propuesto de la figura 17 y se le compara con las asunciones que s en la sección 4.1.1, se puede ver que se deben implementar 3 clases. IgssService, RegularPayrollService y ContractPayrollService.

Para la implementación de estas clases se usará el desarrollo guiado por pruebas unitarias en PHP. Como regla general se debe de empezar por las clases independientes, aquellas que son más fáciles de implementar y sobre todo que no se necesitan *mocks* para realizar sus pruebas unitarias. Esto permitirá enfocarse en las secciones más sencillas y luego se irá incrementando la dificultad con la implementación de las clases dependientes de otras.

4.3.1. Implementación de la clase del servicio del Igss

De las 3 clases que se deben implementar la única que es independiente es la clase del servicio del Igss, la cual debe implementar una interfaz, por lo que se empezará por allí, cumpliendo así con el diseño con base en contratos o interfaces.

```
// FeeableInterface.php
<?php
namespace Payroll\FeeService;

interface FeeableInterface
    * Calculate fee for a specific amount.
    *
    * @param  float $amount
    * @return float
   public function fee($amount);
```

Con esta interfaz se sabe que la clase de servicio del IGSS debe implementar el método cuota (fee) por lo que se empezarán las pruebas unitarias con ese método.

- Iteración 1

Como se inicia el ciclo de TDD se debe entrar a la fase roja, entonces se hará la prueba unitaria lo suficientemente básica para que PHPUnit dé rojo. Considerando esto la primer prueba queda de la siguiente manera.

```
// IgssServiceTest.php
<?php
use Payroll\FeeService\IgssService;

class IgssServiceTest extends PHPUnit_Framework_TestCase
     * @test
    public function it_calculates_igss_fee_when_amount_is_100()
        $igssService = new IgssService();
        $amount = 100;

        $result = $igssService->fee($amount);
        $expected = 3.00;

        $this->assertEquals($expected, $result);
```

Para que las pruebas unitarias sean realmente una fuente de documentación de lo que hace el sistema, deben de titularse de la manera correcta, siendo lo más legibles y descriptivas posibles. En esta prueba no queda ninguna duda de lo que realiza, ya que se titula “Esto calcula la cuota del IGSS cuando la cantidad es de 100”.

Si se continua leyendo ahora dentro de la prueba, se ve que se puede interpretar de la siguiente manera “Dado que se tiene una instancia de la clase “Servicio del IGSS” y la cantidad de 100, cuando se invoca el método “cuota” de dicha instancia y se le manda la cantidad de 100 como parámetro, se espera que devuelva un resultado igual a 3”.

Al correr la prueba da error y con ello se puede avanzar a la fase verde. La implementación mínima de la clase del servicio del IGSS para volver la prueba verde, es la siguiente.

```
// IgssService.php
<?php
namespace Payroll\FeeService;
class IgssService implements FeeableInterface
    public function fee($amount)
        return 3.00;
```

Se ve que aunque no es una solución definitiva, es lo suficiente para pasar la prueba a verde. Como el contenido del método “cuota” es demasiado básico, no es necesaria la fase de refactorización, así que se volverá a la fase roja agregando una prueba más con la mínima complejidad posible.

- Iteración 2

La segunda prueba para que con la mínima complejidad pase a la fase roja, puede ser calcular la cuota correspondiente a otra cantidad, como se ve a continuación.

```
// IgssServiceTest.php
     * @test
    public function it_calculates_igss_fee_when_amount_is_200()
        $igssService = new IgssService();
        $amount = 200;
        $result = $igssService->fee($amount);
        $expected = 6.00;
        $this->assertEquals($expected, $result);
```

Como se ve en esta prueba se calcula ahora la cuota del IGSS para una cantidad de 200 y se espera que el resultado sea 6. Con esto ya se está en la fase roja y se necesita moverse a la fase verde implementando la mínima solución para que satisfaga ambas pruebas. La implementación del método “cuota” podría ser la siguiente.

```
// IgssService.php
<?php
namespace Payroll\FeeService;

class IgssService implements FeeableInterface
    public function fee($amount)
        if ($amount == 100) {
            return 3.00;
        return 6.00;
```

Con esta implementación se pasa de nuevo a fase verde, al igual que la anterior es demasiado básica como para pasar a la fase de factorización, por lo que se está listos para una iteración más.

Es importante realizar estas implementaciones básicas, para enfocarse en la solución mínima y no pensar desde el principio en la solución definitiva, porque si no se agregaría complejidad al desarrollo y no se tendría pruebas para validar dicha implementación.

- Iteración 3

Para entrar de nuevo a la fase roja, se agrega una prueba más, ahora calculando la cuota del Igss para la cantidad de 500, como se ve a continuación.

```
// IgssServiceTest.php
     * @test
    public function it_calculates_igss_fee_when_amount_is_500()
        $igssService = new IgssService();
        $amount = 500;

        $result = $igssService->fee($amount);
        $expected = 15.00;

        $this->assertEquals($expected, $result);
```

Esta vez se entra a la fase roja con 3 pruebas unitarias que nos describen 3 diferentes escenarios del método “cuota”, por lo que ya se puede generalizar la solución a través de la triangulación y corroborar con 3 pruebas unitarias que satisface el requerimiento. La implementación generalizada sería la siguiente.

```
// IgssService.php
<?php
namespace Payroll\FeeService;
```

```
class IgssService implements FeeableInterface
    public function fee($amount)
    {
        return $amount * 0.03;
```

Esta solución se acerca a la solución definitiva, pero en esta ocasión sí se puede entrar a la fase de refactorización. Según Robert C. Martin, para que un código sea mantenible y legible, no debería de incluir “números mágicos”, es decir, es recomendable sustituir esos números por una constante que describa lo que son.

Después de refactorizar la implementación quedaría de la siguiente manera.

```
// IgssService.php
<?php
namespace Payroll\FeeService;

class IgssService implements FeeableInterface
    const IGSS_PERCENTAGE = 0.03;

    public function fee($amount)
        return $amount * self::IGSS_PERCENTAGE;
```

En esta implementación se ve claramente que el 0,03 es el porcentaje correspondiente a la cuota del Igss por lo que queda claramente legible la implementación.

Se podría concluir acá la implementación del método cuota, pero se debe probar aún escenarios menos comunes. ¿Qué pasa si se manda cero como parámetro, se comportará de la misma manera con una cantidad 1 000 veces

más grande? Se deben hacer estas pruebas una por una, para garantizar que la solución es la definitiva y considera esos escenarios.

- Iteración 4

En la siguiente prueba se espera que la cuota del IGSS sea cero si la cantidad enviada como parámetro es cero.

```
// IgssServiceTest.php
     * @test
    public function it_calculates_igss_fee_when_amount_is_zero()
        $igssService = new IgssService();
        $amount = 0;

        $result = $igssService->fee($amount);
        $expected = 0.00;

        $this->assertEquals($expected, $result);
```

- Iteración 5

Se ve que al correr las pruebas siguen dando verde, por lo que solo queda una prueba más por hacer y es probar con un número mucho más grande.

```
// IgssServiceTest.php
     * @test
    public function it_calculates_igss_fee_when_amount_is_35000()
        $igssService = new IgssService();
        $amount = 35000;

        $result = $igssService->fee($amount);
        $expected = 1050.00;
```

```
        $this->assertEquals($expected, $result);
```

Se corren las pruebas de nuevo y todas pasan. Pero qué pasa si se prueva un número que tengan un resultado con decimales.

- Iteración 6

Se hace una prueba en la que tiene que devolver 2,55 de cuota del IGSS para la cantidad de 85.

```
// IgssServiceTest.php
     * @test
    public function it_calculates_igss_fee_when_amount_is_85()
        $igssService = new IgssService();
        $amount = 85;

        $result = $igssService->fee($amount);
        $expected = 2.55;

        $this->assertEquals($expected, $result);
```

Se corre todas la pruebas y pasan sin problemas, por lo que se da por finalizada la implementación de la clase.

Con las pruebas anteriores, se logran 3 cosas, diseñar la solución con base en escenarios reales, dejar una documentación de lo que realmente hace el código, al construir pruebas descriptivas y por último, se crea un método de verificación, para que en futuras modificaciones, se garantice que siga aplicando la solución para el requerimiento inicial.

Se debe agregar las anotaciones respectivas, para que cualquiera que las vea no le quede duda alguna de lo que hace viendo el código directamente, quedando de la siguiente manera.

```
// IgssService.php
<?php
namespace Payroll\FeeService;

class IgssService implements FeeableInterface
     * Constant for the IGSS percentage.
    const IGSS_PERCENTAGE = 0.03;

     * Get IGSS fee for a specific amount.
     * @param  float $amount
     * @return float
     */    public function fee($amount)
        return $amount * self::IGSS_PERCENTAGE;
```

4.3.2. Implementación de la clase del servicio de nómina por contrato

En este punto quedan 2 clases por implementar, la clase de servicio de nóminas por contrato y servicio de nóminas regulares, pero según el diseño de la figura 17 la que menos dependencias tiene es la de servicio de nóminas por contrato.

Esta debe implementar la interfaz de servicio de nómina (PayrollServiceInterface) por lo que se empezará definiendo este contrato.

```
// PayrollServiceInterface.php
<?php
namespace Payroll\PayrollService;
```

```
use Payroll\Employee\EmployeeInterface;

interface PayrollServiceInterface
     * @param  \Payroll\Employee\EmployeeInterface
     * @return float
    public function salary(EmployeeInterface $employee);
```

Al conocer que la clase de servicio de nómina por contrato debe implementar la interfaz de servicio de nómina, es claro que el método a implementar es Salario. Además por el diseño de la figura 17, se ve que depende de 2 instancias de clases que implementan la interfaz cuota de empleado, específicamente para el cálculo del ISR y la cuota del préstamo, si el empleado lo posee. Como se menciona anteriormente, estas dependencias se resolverán por el patrón de diseño de inyección de dependencias, el cual dicta enviarselas al constructor de la clase que las necesita.

De acuerdo a la lógica del negocio, el salario de un empleado por contrato es calculado con base en su salario base, menos las deducciones del ISR y la cuota del préstamo si lo tiene.

- Iteración 1

Recordar que en una prueba unitaria solo interesa lo que ocurra dentro del método al cual se le hará la prueba. El detalle de lo que pase con sus dependencias es de importancia de las pruebas unitarias para dichas clases. Con base en esto, es claro que se debe usar *mocks* para imitar el comportamiento de sus dependencias y en la prueba solo indicar cómo interactuarán entre sí. La primer prueba quedaría de la siguiente manera.

```
// ContractPayrollServiceTest.php<?php
use Payroll\PayrollService\ContractPayrollService;

class ContractPayrollServiceTest extends PHPUnit_Framework_TestCase
     * @test
    public function
it_calculates_salary_for_contract_employee_who_earn_a_base_salary_of_1000()
        $employeeMock = $this->getMock('Payroll\Employee\EmployeeInterface');
        $employeeMock->expects($this->once())
                        ->method('getBaseSalary')
                        ->will($this->returnValue(1000));
        $isrServiceMock = $this-
>getMock('Payroll\FeeService\EmployeeFeeInterface');
        $isrServiceMock->expects($this->once())
                        ->method('fee')
                        ->will($this->returnValue(50));
        $loanServiceMock = $this-
>getMock('Payroll\FeeService\EmployeeFeeInterface');
        $loanServiceMock->expects($this->once())
                    ->method('fee')
                    ->will($this->returnValue(50));

        $payrollService = new ContractPayrollService($isrServiceMock,
            $loanServiceMock);
        $result = $payrollService->salary($employeeMock);

        $this->assertEquals(900, $result);
```

Esta prueba aunque es un poco más grande en código es muy fácil de leer. Empesando por el título, el cual claramente describe el funcionamiento a probar y dice “Esto calcula el salario para un empleado por contrato quien gana un sueldo base de 1 000”.

Analizar el código por partes. Al inicio se encuentra el primer *mock*, el cual pertenece al empleado. Si se lee el código se interpretará lo siguiente “Obtenga un *mock* que implemente la interfaz empleado, el cual espera llamar 1 vez al método ‘obtener salario base’ y que este devuelva la cantidad de 1 000”. Con esto ya se imita el comportamiento de esta dependencia.

```
        $employeeMock = $this->getMock('Payroll\Employee\EmployeeInterface');
        $employeeMock->expects($this->once())
                        ->method('getBaseSalary')
                        ->will($this->returnValue(1000));
```

Si se continúa se encontrará ahora con el *mock* que corresponde al Servicio del ISR. Si se lee el código dice “Obtenga un *mock* que implemente la interfaz ‘Cuota de empleado’, el cual necesita llamar una vez a su método ‘cuota’ y que este devuelva un valor de 50”. Con esto es suficiente para imitar la dependencia del Servicio del ISR.

```
        $isrServiceMock = $this-
>getMock('Payroll\FeeService\EmployeeFeeInterface');
        $isrServiceMock->expects($this->once())
                        ->method('fee')
                        ->will($this->returnValue(50));
```

Ahora se debe encontrar el siguiente *mock* correspondiente al Servicio de préstamos, el cual dice “Obtenga un *mock* que implemente la interfaz de ‘Cuota de empleado’, el cual espera llamar 1 vez al método ‘cuota’ y que este devuelva la cantidad de 50”. Con esto se imita la dependencia de la clase de Servicio de préstamos que se necesita en el método que se esta probando.

```
        $loanMock = $this-
>getMock('Payroll\FeeService\EmployeeFeeInterface');
```

```
$loanMock->expects($this->once())
            ->method('fee')
            ->will($this->returnValue(50));
```

Después de crear los *mocks*, se crea la instancia de la clase que se está probando (servicio de nómina por contrato) y como esta depende de dos instancia de clase que implementen la interfaz ‘Cuota de empleado’, se le manda como parámetro tanto el *mock* de Servicio del ISR como el *mock* del servicio de préstamos que se hizo anteriormente.

```
$payrollService = new ContractPayrollService($isrServiceMock,
    $loanServiceMock);
```

Ya teniendo la instancia del servicio de nómina por contrato, se accede a su método ‘salario’ y se le envía por parámetro el *mock* de la clase empleado. Por último se necesita que el valor devuelto de este método sea igual al 900.

```
$result = $payrollService->salary($employeeMock);
$this->assertEquals(900, $result);
```

Con esta prueba ya se entra en la fase roja y se necesita la implementación mínima para pasar a la fase verde. Debido a que la prueba que se hizo es bastante completa y sobre todo define bien como las dependencias deben interactuar, la solución que pasa a verde esta prueba, es casi la solución final. Por lo que la clase de servicio de nómina por contrato quedaría de la siguiente forma.

```
// ContractPayrollService.php
<?php
namespace Payroll\PayrollService;
```

```
use Payroll\Employee\EmployeeInterface;
use Payroll\FeeService\EmployeeFeeInterface;

class ContractPayrollService implements PayrollServiceInterface
    private $loanService;

    public function __construct(EmployeeFeeInterface $loanService)
        $this->loanService = $loanService;
    public function salary(EmployeeInterface $employee)
        return $employee->getBaseSalary() -
            $this->isrService->fee($employee) -
            $this->loanService->fee($employee);
```

La solución es sencilla, al sueldo base del empleado, que le se pasa por parámetro al método salario, se le resta la cuota del ISR correspondiente y la cuota de préstamo para dicho empleado. Si el empleado no tiene ningún préstamo dicha cuota devolverá cero, pero esa ya es lógica de la dependencia, que no importa en esta prueba unitaria. Debido a que la solución se reduce a una simple resta, no es necesario pasar por la fase de refactorización.

- Iteración 2

Para tratar de entrar de nuevo a la fase roja, se escribirá la siguiente prueba, la cual describe un escenario diferente para el mismo método. En esta ocasión el sueldo base del empleado es de 5 000 y su cuota de préstamo correspondiente al presente mes es de 750, por lo que su salario debería de ser de 4 250.

```
// ContractPayrollServiceTest.php
     * @test
```

```
    public function
it_calculates_salary_for_contract_employee_who_earn_a_base_salary_of_5000()
        $employeeMock = $this->getMock('Payroll\Employee\EmployeeInterface');
        $employeeMock->expects($this->once())
                        ->method('getBaseSalary')
                        ->will($this->returnValue(5000));
        $isrServiceMock = $this-
>getMock('Payroll\FeeService\EmployeeFeeInterface');
        $isrServiceMock->expects($this->once())
                        ->method('fee')
                        ->will($this->returnValue(150));
        $loanServiceMock = $this-
>getMock('Payroll\FeeService\EmployeeFeeInterface');
        $loanServiceMock->expects($this->once())
                    ->method('fee')
                    ->will($this->returnValue(750));

        $payrollService = new ContractPayrollService($isrServiceMock,
$loanServiceMock);
        $result = $payrollService->salary($employeeMock);

        $this->assertEquals(4100, $result);
```

Se correrán una vez más las pruebas y se verá que siguen en verde, lo que indica que la solución también satisface este escenario.

- Iteración 3

¿Qué pasaría si un empleado no tiene un préstamo? Cada vez que se tenga un cuestionamiento, quiere decir que representa un posible escenario y lo que se debería hacer es escribir una prueba para verificarlo. La siguiente prueba debería de ser la siguiente.

Si un empleado cuyo sueldo base es de 3 500, no tiene préstamo alguno y su cuota de ISR es de 100, el salario que debería de recibir es de 3 400.

```
// ContractPayrollServiceTest.php
     * @test
    public function
it_calculates_salary_for_contract_employee_who_earn_a_base_salary_of_3500()
        $employeeMock = $this->getMock('Payroll\Employee\EmployeeInterface');
        $employeeMock->expects($this->once())
                        ->method('getBaseSalary')
                        ->will($this->returnValue(3500));
        $isrServiceMock = $this-
>getMock('Payroll\FeeService\EmployeeFeeInterface');
        $isrServiceMock->expects($this->once())
                        ->method('fee')
                        ->will($this->returnValue(100));
        $loanServiceMock = $this-
>getMock('Payroll\FeeService\EmployeeFeeInterface');
        $loanServiceMock->expects($this->once())
                    ->method('fee')
                    ->will($this->returnValue(0));

        $payrollService = new ContractPayrollService($isrServiceMock,
$loanServiceMock);
        $result = $payrollService->salary($employeeMock);

        $this->assertEquals(3400, $result);
```

Con esta prueba se da por terminada la implementación, por lo que se le debe agregar anotaciones al código y la clase final quedaría de la siguiente manera.

```
// ContractPayrollService.php
<?php
namespace Payroll\PayrollService;

use Payroll\Employee\EmployeeInterface;
use Payroll\FeeService\EmployeeFeeInterface;

class ContractPayrollService implements PayrollServiceInterface
     * Instance of Fee Service for ISR
     * @var \Payroll\EmployeeFeeInterface
    private $isrService;

     * Instance of Fee Service for Loan
     * @var \Payroll\EmployeeFeeInterface
    private $loanService;

     * Create a instance.
     * @param \Payroll\FeeService\EmployeeFeeInterface
     * @param \Payroll\FeeService\EmployeeFeeInterface
    public function __construct(EmployeeFeeInterface $isrService,
EmployeeFeeInterface $loanService)
        $this->isrService = $isrService;
        $this->loanService = $loanService;
     * Calculate salary for a Contract Employee.
     * @param  \Payroll\Employee\EmployeeInterface
     * @return float
    public function salary(EmployeeInterface $employee)
        return $employee->getBaseSalary() -
            $this->isrService->fee($employee) -
            $this->loanService->fee($employee);
```

4.3.3. Implementación de la clase del servicio de nómina para un empleado regular

Por último se debe implementar la clase de servicio de nómina para empleados de planilla, empleados regulares a partir de ahora. Esta clase es la que cuenta con mayor número de dependencias, pero su implementación no difiere mucho de la anterior.

Según la lógica del negocio, el salario de un empleado regular, se calcula, restándole a su sueldo base, el 3 % del Igss, la cuota proyectada del ISR y la cuota mensual del préstamo que tenga. Además según el diseño de la figura 17 se ve que debe implementar la interfaz de servicio de nómina, por lo que se debe desarrollar el método ‘salario’. Con estas indicaciones se puede proceder a la primer iteración.

- Iteración 1

Cuando se necesita probar un método con varias dependencias, es difícil crear un mínimo escenario para pasar a la fase roja, ya que se debe de establecer desde el inicio como estas dependencias interactúan, por lo que la primera prueba quedaría de la siguiente manera.

```
// RegularPayrollServiceTest.php
<?php
use Payroll\PayrollService\RegularPayrollService;

class RegularPayrollServiceTest extends PHPUnit_Framework_TestCase
     * @test
    public function
it_calculates_salary_for_a_regular_employee_whose_base_salary_is_1000()
        $employeeMock = $this->getMock('Payroll\Employee\EmployeeInterface');
```

```
        $employeeMock->expects($this->exactly(2))
                        ->method('getBaseSalary')
                        ->will($this->returnValue(1000));

        $isrServiceMock = $this-
>getMock('Payroll\FeeService\EmployeeFeeInterface');
        $isrServiceMock->expects($this->once())
                        ->method('fee')
                        ->will($this->returnValue(50));

        $loanServiceMock = $this-
>getMock('Payroll\FeeService\EmployeeFeeInterface');
        $loanServiceMock->expects($this->once())
                        ->method('fee')
                        ->will($this->returnValue(100));

        $igssServiceMock = $this-
>getMock('Payroll\FeeService\FeeableInterface');
        $igssServiceMock->expects($this->once())
                        ->method('fee')
                        ->will($this->returnValue(30));

        $payrollService = new RegularPayrollService(
                                    $isrServiceMock,
                                    $igssServiceMock,
                                    $loanServiceMock);
        $result = $payrollService->salary($employeeMock);

        $this->assertEquals(820, $result);
```

Esta prueba aparenta ser muy grande, pero es fácil de entender. Se ve que su título es totalmente descriptivo, leyéndose “Esto calcula el salario para un empleado regular cuyo salario base es de 1 000”.

En su interior lo primero que se ve es un *mock* de la clase que implementa la interfaz empleado y que necesita que su método ‘obtener salario base’ sea llamado 2 veces exactamente y que este devuelva una cantidad de 1 000.

Después se ve que se hace un *mock* de la clase que implementa la interfaz ‘cuota de empleado’ correspondiente al Servicio ISR, el cual debe llamar a su método ‘cuota’ una vez y este devolverá la cantidad de 50.

Siguiendo con los *mocks*, después se encuentra el correspondiente al servicio de préstamo, que también implementa la interfaz ‘Cuota de empleado’. Este *mock* necesita que llamen una vez a su método ‘cuota’ y este devolverá la cantidad de 100.

El último *mock* corresponde al Servicio del IGSS que implementa la interfaz ‘habilitado para cuotas’ y este necesita que llamen una vez su método ‘cuota’ y devolverá la cantidad de 30.

Hasta este momento se tiene 1 *mock* de la clase empleado y 3 *mocks* correspondientes a los servicios los cuales depende nuestra clase a la que le estamos realizando la prueba.

El siguiente paso fue crear la instancia de la clase que se está probando ‘servicio de nómina regular’. Para construirse necesita como dependencia instancias de los servicios ISR, Préstamo e IGSS respectivamente, por lo que se le envía como parámetros los *mocks* de servicios que se crean.

Por último en la instancia del servicio de nómina regular se invoca al método ‘salario’ y su resultado se necesita que sea igual a 820. Si se corren las pruebas esta última falla por lo que ya se puede avanzar a la fase verde.

Al prestar atención a las dependencias y a como obtener el resultado, se ve que la implementación de la solución es sencilla.

En otras palabras, el sueldo base (1 000) - cuota del ISR (50) - cuota del préstamo (100) - cuota del Igss (30) debe ser igual a 820.

```
// RegularPayrollService.php
<?php
namespace Payroll\PayrollService;

use Payroll\FeeService\EmployeeFeeInterface;
use Payroll\Employee\EmployeeInterface;

class RegularPayrollService implements PayrollServiceInterface
    private $isrService;
    private $igssService;
    private $loanService;

    public function __construct(EmployeeFeeInterface $isrService,
                                EmployeeFeeInterface $igssService,
                                EmployeeFeeInterface $loanService)
        $this->isrService = $isrService;
        $this->igssService = $igssService;
        $this->loanService = $loanService;

    public function salary(EmployeeInterface $employee)
        return $employee->getBaseSalary() -
                $this->isrService->fee($employee) -
                $this->igssService->fee($employee->getBaseSalary()) -
                $this->loanService->fee($employee);
```

Como se adelantó anteriormente la solución es sencilla, al delegar responsabilidades, la implementación se reduce a una resta. Estas

responsabilidades están delegadas a sus dependencias, por eso es que se inyectan en su constructor y luego se hace uso de ellas en el método ‘salario’ a cual se le estan haciendo las pruebas unitarias. En cuanto a la fase refactorización no es necesaria debido a la simplicidad de la solución.

- Iteración 2

Para asegurar que la solución es la correcta se debe realizar otra iteración ahora con un escenario diferente. Que tal si ahora el empleado gana 5 000 de sueldo base, su ISR mensual es de 250, la cuota del IGSS es de 150 y no tiene préstamos es decir, la cuota de préstamo es cero, entonces su Salario sería de 4 600. Con estas condiciones se construye la siguiente prueba para tratar de entrar en la fase roja.

```
// RegularPayrollServiceTest.php
     * @test
    public function
it_calculates_salary_for_a_regular_employee_whose_base_salary_is_5000()
        $employeeMock = $this->getMock('Payroll\Employee\EmployeeInterface');
        $employeeMock->expects($this->exactly(2))
                        ->method('getBaseSalary')
                        ->will($this->returnValue(5000));

        $isrServiceMock = $this-
>getMock('Payroll\FeeService\EmployeeFeeInterface');
        $isrServiceMock->expects($this->once())
                        ->method('fee')
                        ->will($this->returnValue(250));

        $loanServiceMock = $this-
>getMock('Payroll\FeeService\EmployeeFeeInterface');
        $loanServiceMock->expects($this->once())
```

```
                        ->method('fee')
                        ->will($this->returnValue(0));

        $igssServiceMock = $this-
>getMock('Payroll\FeeService\FeeableInterface');
        $igssServiceMock->expects($this->once())
                        ->method('fee')
                        ->will($this->returnValue(150));

        $payrollService = new RegularPayrollService(
                                    $isrServiceMock,
                                    $igssServiceMock,
                                    $loanServiceMock);
        $result = $payrollService->salary($employeeMock);

        $this->assertEquals(4600, $result);
    }
```

Se corren las pruebas unitarias y una vez más dan verde, por lo que se podría dar por terminada la implementación de la clase de Servicio de Nómina regular, solo faltaría agregar las anotaciones respectivas, quedando la clase de la siguiente manera.

```
// RegularPayrollService.php
<?php
namespace Payroll\PayrollService;

use Payroll\FeeService\EmployeeFeeInterface;
use Payroll\Employee\EmployeeInterface;

class RegularPayrollService implements PayrollServiceInterface
     * ISR Service instance.
     * @var \Payroll\FeeService\EmployeeFeeInterface
```

```
private $isrService;

 * IGSS Service instance.
 * @var \Payroll\FeeService\EmployeeFeeInterface
private $igssService;

 * Loan Service instance.
 * @var \Payroll\FeeService\EmployeeFeeInterface
private $loanService;

 * Make instance of Regular Payroll Service.
 * @param \Payroll\FeeService\EmployeeFeeInterface $isrService
 * @param \Payroll\FeeService\EmployeeFeeInterface $igssService
 * @param \Payroll\FeeService\EmployeeFeeInterface $loanService
public function __construct(EmployeeFeeInterface $isrService,
                            FeeableInterface $igssService,
                            EmployeeFeeInterface $loanService)
    $this->isrService = $isrService;
    $this->igssService = $igssService;
    $this->loanService = $loanService;

 * Calculate employee salary.
 * @param  \Payroll\Employee\EmployeeInterface $employee
 * @return float
public function salary(EmployeeInterface $employee)
    return $employee->getBaseSalary() -
            $this->isrService->fee($employee) -
            $this->igssService->fee($employee->getBaseSalary()) -
            $this->loanService->fee($employee);
```

4.4. Uso del API

En este punto se ha dado fin al requerimiento y se ha desarrollado la interfaz de programación de aplicaciones o mejor conocida como API por sus siglas en inglés (Application programing interface). Pero ¿cómo usarla? Ya las pruebas unitarias dan varios ejemplos de cómo hacerlo. A continuación se ve un ejemplo básico de cómo calcular el salario de la nómina de empleados.

```
// index.php
<?php
require 'vendor/autoload.php';

use Payroll\Employee\Employee;
use Payroll\FeeService\EmployeeLoanService;
use Payroll\FeeService\IgssService;
use Payroll\FeeService\IsrService;
use Payroll\PayrollService\ContractPayrollService;
use Payroll\PayrollService\RegularPayrollService;

$employees[] = new Employee(1, 'Jon Snow', 6000, 'contract');
$employees[] = new Employee(2, 'Tyrion Lannister', 12000, 'regular');
$employees[] = new Employee(3, 'Arya Stark', 8000, 'contract');
$employees[] = new Employee(4, 'Daenerys Targaryen', 20000, 'regular');

$payroll = [];
foreach ($employees as $employee) {
    switch ($employee->type()) {
        case 'contract':
            $payrollService = new ContractPayrollService(
                new IsrService,
                new EmployeeLoanService);
            break;
```

```
        default:
            $payrollService = new RegularPayrollService(
                new IsrService,
                new IgssService,
                new EmployeeLoanService);
            break;

    $payroll[$employee->name] = $payrollService->salary($employee);

print_r(var_dump($payroll));
```

Para que se pueda evaluar el API más detenidamente, se deja el código en la siguiente direccion en internet. https://github.com/carnar/Payroll

5. ENCUESTA EN ESTUDIANTES Y EGRESADOS DE LA ESCUELA DE CIENCIAS Y SISTEMAS DE LA USAC

Para conocer el nivel de entendimiento y utilización del desarrollo guiado por pruebas en los estudiantes y egresados de la Escuela de Ciencias y Sistemas de la Universidad de San Carlos de Guatemala, se realizó una encuesta, la cual tuvo una participación de 110 personas. Esta encuesta fue circulada en redes sociales, específicamente en grupos de estudiantes y egresados de sistemas de la Usac.

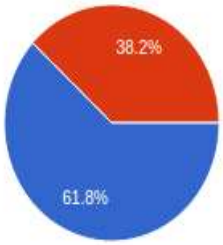
5.1. Encuesta y resultados

A continuación se describen las preguntas así como los resultados de la misma.

- Pregunta 1

Figura 18. **Encuestas y resultados 1**

¿Cuál es su relación actual con la Escuela de Ciencias y Sistemas?



Estudiante	**68**	61,8 %
Egresado	**42**	38,2 %

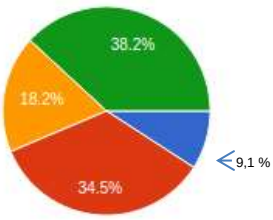
Fuente: elaboración propia.

En esta pregunta se evalúa cuántos de los encuestado fueron estudiantes y cuántos egresados de la Escuela de Ciencias y Sistemas de la Usac. El 61,8 % fueron egresados y el resto estudiantes.

- Pregunta 2

Figura 19. **Encuestas y resultados 2**

¿Cuál es su conocimiento acerca de TDD?



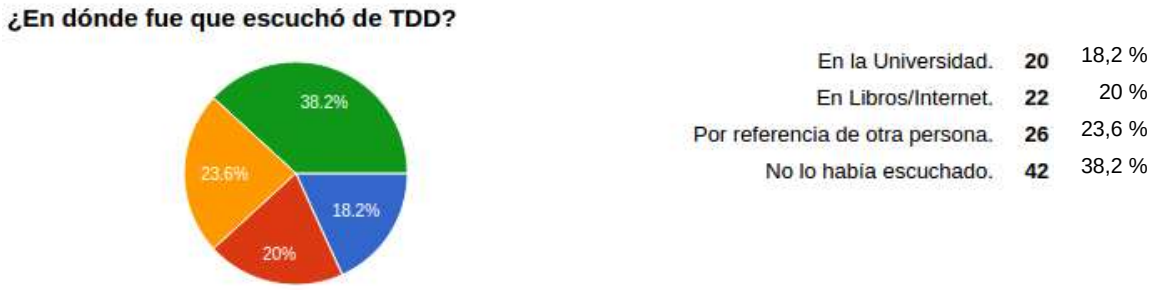
Lo practico cuando desarrollo software.	**10**	9,1 %
Lo he puesto en práctica alguna vez.	**38**	34,5 %
He oído/leido acerca del tema.	**20**	18,2 %
Desconozco el tema.	**42**	38,2 %

Fuente: elaboración propia.

Al evaluar el conocimiento acerca del desarrollo guiado por pruebas, se puede observar que predominan las personas que desconocen el tema con el 38,2 %, seguido por los que lo han puesto en práctica alguna vez con el 34,5 %.

- Pregunta 3

Figura 20. **Encuestas y resultados 3**

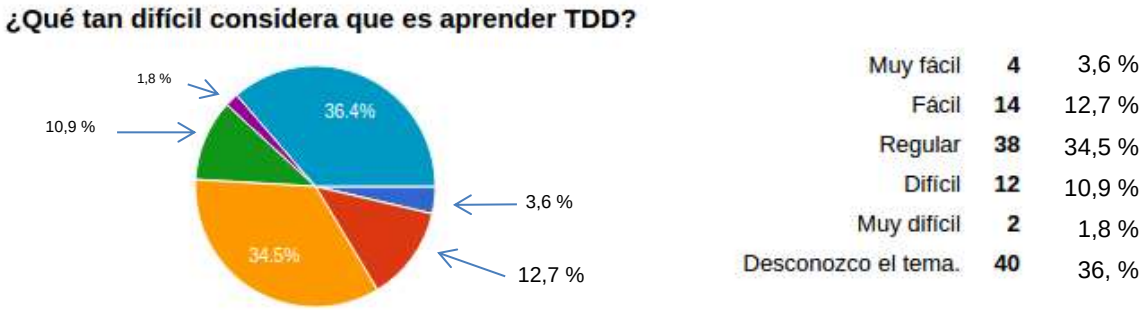


Fuente: elaboración propia.

En esta pregunta se evaluó cómo conoció el tema el encuestado y se ve que la mayoría no lo habían escuchado con el 38,2 %, seguido por la referencia de otra persona con el 23,6 %.

- Pregunta 4

Figura 21. **Encuestas y resultados 4**

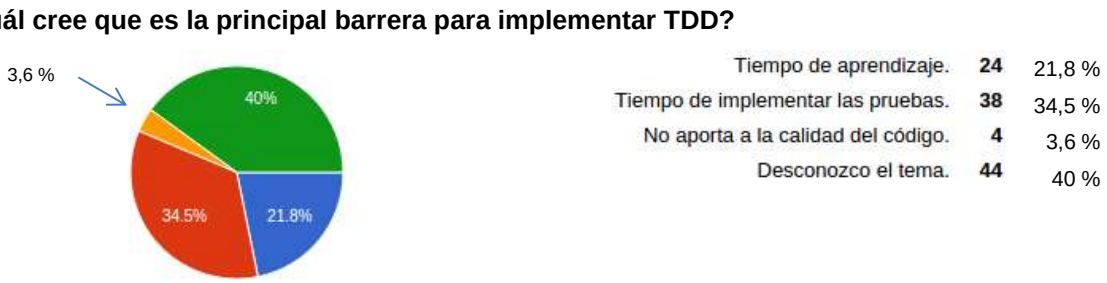


Fuente: elaboración propia.

Evaluando el nivel de complejidad al aprender TDD, se encuentran que el 36,4 % desconoce el tema siendo esta las respuesta mayormente escogida. Le sigue con un 34,5 % los que consideran que el aprendizaje de TDD tiene una dificultad media.

- Pregunta 5

Figura 22. **Encuestas y resultados 5**

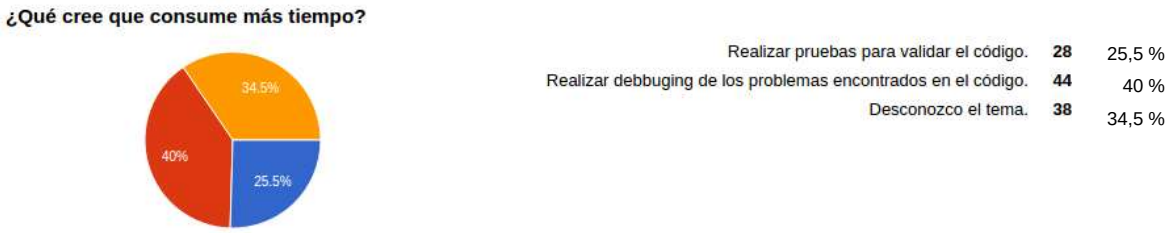


Fuente: elaboración propia.

Esta pregunta evalúa las dificultades de implementar TDD. La primera opción es del 40 % desconociendo el tema y la segunda considera que la principal barrera para implementar TDD es el tiempo que toma implementar las pruebas con 34,5 %.

- Pregunta 6

Figura 23. **Encuestas y resultados 6**

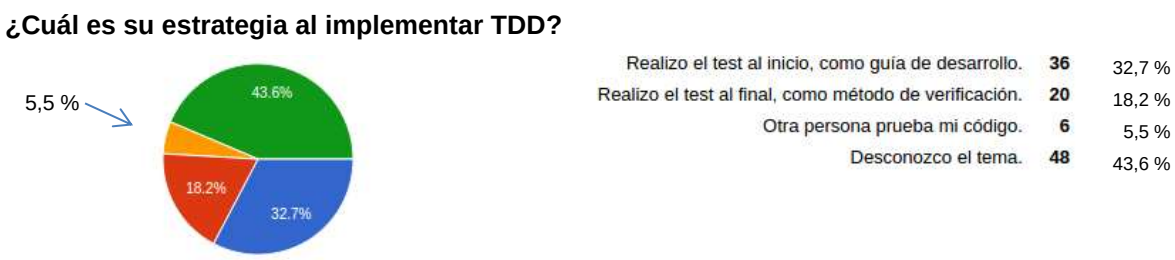


Fuente: elaboración propia.

También se evaluó lo que consideran que consume más tiempo en el desarrollo de software. La mayoría considera que es la depuración de errores cuando estos aparecen, con el 40 %. La segunda posición la ocupa el 34,5 % desconociendo el tema.

- Pregunta 7

Figura 24. **Encuestas y resultados 7**

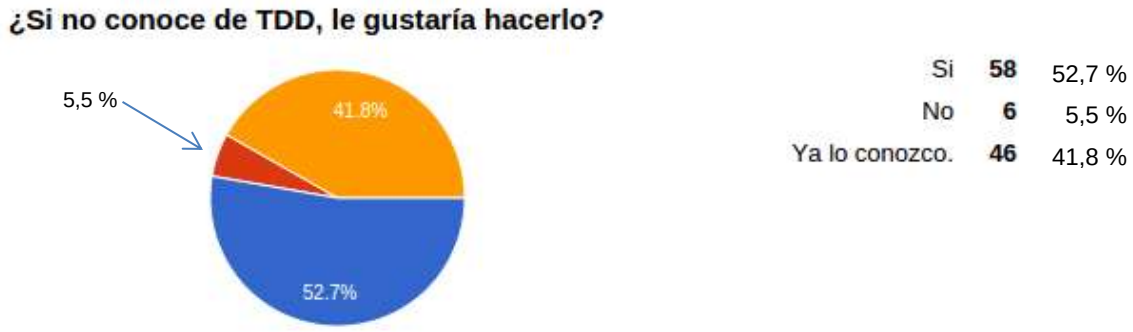


Fuente: elaboración propia.

Con esta pregunta se intentó evaluar cómo las personas utilizan TDD. La mayoría desconoce del tema con el 43,6 %, siguiéndole la estrategia de realizar la prueba al inicio con el 32,7 %.

- Pregunta 8

Figura 25. **Encuestas y resultados 8**



Fuente: elaboración propia.

Por último, se evaluó el interés de las personas, de conocer TDD. El 52,7 % de las personas si quieren hacerlo, mientras que el 5,5 % dijeron que no. Los demás ya lo conocían.

5.2. Análisis de resultados

El principal punto a destacar en la presente encuesta es que en la mayoría de preguntas, los encuestados desconocen del tema con un porcentaje cercano al 40 %. Por otro lado solo el 9,1 % practica TDD cuando desarrolla software.

TDD es un tema el cual la mayoría escucha por referencia de otras personas, por iniciativa propia consultando fuentes bibliográficas o en internet.

Las personas que conocen de TDD consideran que su aprendizaje tiene complejidad media con un 34,5 % y considerando que el ciclo de TDD dicta que se deben hacer las pruebas antes de empezar a programar, la mayoría de persona que ha utilizado TDD siguen esta estrategia con el 32,7 %, frente al 18,2 % que escriben las pruebas al final de la implementación del código como método de verificación.

Por último se puede destacar que las personas que no conocen el tema, muestran un interés por ampliar sus conocimientos con un 52,7 % frente al 5,5 % que prefiere no hacerlo. Los demás ya dominan o conocen de él.

CONCLUSIONES

1. A pesar que el desarrollo guiado por pruebas tiene un ciclo claramente definido, su utilización no puede ser fácilmente implementada si no se consideran aspectos externos. Estos aspectos se enfocan en las buenas prácticas para el desarrollo de software, que permiten crear código que se puede probar fácilmente.

2. En la actualidad el desarrollo guiado por pruebas en PHP es totalmente viable, debido a que herramientas como PHPUnit o PHPSpec proporcionan todos los componentes necesarios para su implementación. Dichos componentes permiten hacer todo tipo de verificaciones, así como la posibilidad de realizar pruebas en aislamiento con el uso de *mocks* para imitar el comportamiento de sus dependencias.

3. Aunque el desarrollo guiado por pruebas es un proceso de desarrollo de software que posee más de 15 años de existencia, no es tan conocido hoy en día en el ámbito del desarrollo web, pero su penetración ha ido aumentando a medida de su viabilidad en lenguajes tan conocidos como PHP.

RECOMENDACIONES

1. Para minimizar la curva de aprendizaje del desarrollo guiado por pruebas se debe comprender y dominar cada uno de los principios de diseño SOLID.

2. Para maximizar el beneficio del desarrollo guiado por pruebas, se debe de utilizar la estrategia de creación de pruebas antes de cualquier implementación de código. Esto permitirá que realmente guíe al desarrollador y proporcione una herramienta de diseño además de una herramienta de verificación.

3. Para facilitar la implementación del desarrollo guiado por pruebas se debe trabajar ampliamente en el análisis de requerimientos, así como el diseño de la solución. Esto permitirá poder segmentar cada componente de la aplicación y así poder enfocarse en cada funcionalidad por separado.

BIBLIOGRAFÍA

1. BENDER, James; MCWHERTER Jeff. *Professional test driven development with c#*: *developing real world applications with TDD*. USA: Wiley Publishing, 2011. 135 p.

2. *Dependency injection pattern.* [en línea]. <https://msdn.microsoft.com/en-us/library/ vstudio/hh323705(v=vs.100).aspx>. [Consulta: octubre de 2015].

3. FREEMAN, Eric; FREEMAN, Elisabeth; SIERRA, Kathy; BATES, Bert. *Design patterns head first*. USA: O'Really, 2004. 185 p.

4. JOYANES, Luis. *Programación orientada a objetos: conceptos, modelado, diseño y codificación en C++*. Mexico: McGraw-Hill, 1996. 197 p.

5. *Laracast*. [en línea]. <https://laracasts.com/>. [Consulta: septiembre de 2015].

6. MARTIN, Robert C. *Code clean*. USA: Pearson Education, 2009. 210 p.

7. _____. MARTIN, Micah. *Agile principles, patterns, and practices in C.* USA*:* Pearson Education, 2007. 260 p.

8. Microsoft. *Unit testing.* [en línea]. <https://msdn.microsoft.com/en-us/library/aa292197(v=vs.71).aspx>. [Consulta: octubre de 2015].

9. OTWELL, Taylor. *Laravel: from apprentice to Artisan - Advanced Architecture With Laravel 4*. Leanpub 2013.

10. *PHPUnit documentation.* [en línea]. <https://phpunit.de/manual/current/en/ index.html>. [Consulta: octubre de 2015].

11. *PHPUnit en GitHub.* [en línea]. <https://github.com/ sebastian bergmann/ phpunit>. [Consulta: octubre de 2015].

12. *Unified modeling language* [en línea]. (UML). <http://www.uml.org/>. [Consulta: septiembre de 2015].