BY ITALO MORALES F.

TDD

LO QUE DEBES SABER

CREANDO GRANDES PROGRAMADORES

TDD - Lo que debes saber

Rimorsoft Online y el Profesor Italo Morales F

Este libro está a la venta en http://leanpub.com/tdd-lo-que-debes-saber

Esta versión se publicó en 2019-09-19

Dedicado a mi madre.

Índice general

Sobre Nosotros

Somos una comunidad de programadores web de habla hispana.

Enseñamos a trabajar, nos gusta ayudar y consideramos como nuestro mayor placer ver a las personas crecer, formar sus empresas o escalar rápidamente como empleados. Tenemos como meta usar nuestra tecnología de estudio para transformar la manera en que se enseña y así vencer las barreras de estudio.

Somos reconocidos en la comunidad por nuestra entrega de calidad, deseamos desarrollar de forma confiable, segura y rentable conocimientos con calidad humana y capacidad de **Rimorsoft Online**.

Desarrollamos líderes que se anticipan, adaptan y crean cambios, que aprenden de la experiencia e innovan permanentemente.

Nuestra web https://rimorsoft.com

Al Estudiante

Este es un curso completamente práctico, la idea real es enseñarte a trabajar y prepararte con textos sencillos pero de gran competencia para enfrentar el mundo laboral como un profesional verdadero altamante calificado.

Este material es parte de una obra más extensa que te formará en varias áreas técnicas, donde al juntar cada granito de arena buscamos desarrollarte como un profesional web competente.

Yo me formé como programador y luego de unos años como profesor... La metodología aquí empleada está probada y funciona.

No te ofrezco mas que ayudarte en el bello mundo del saber, específicamente en estos temas informáticos. Espero que este libro sea un bonito viaje al descubrimiento y futuros logros empresariales.

Te deseo el mayor de los éxitos.

– Profesor, Italo Morales F.

Nota Importante

Al estudiar este libro me voy a asegurar de que realmente aprendas… En ese sentido necesito tu ayuda, no pases ninguna página si no está completamente claro el tema o si alguna palabra no está totalmente comprendida.

Una persona abandona un estudio, un tema o un curso cuando no comprende, y esto no se debe exactamente por la falta de capacidad, en realidad se debe a que no se atrevió a aclarar eso que quedó confuso o muy poco claro.

Por favor, no sigas adelante si algo parece confuso, la forma correcta de atacar es ir ANTES de que tuvieras problemas, encontrar eso que no entiendes y definirlo hasta que tengas una imagen clara del tema o palabra.

Si lo hacemos así serás un gran profesional, serás una persona de excelente competencia en el mercado laboral.

Recuerda que puedo ayudarte en cualquier momento.

– Profesor, Italo Morales F.

Agradecimiento

Te agradezco a ti querido mi amigo y suscriptor, me siento escritor cuando sé que me estás leyendo y me estás ayudando a mejorar cada día.

Mil gracias por:

1. Adquirir el curso premium de **TDD en Laravel**.
2. Ver mis videos en nuestro canal de https://youtube.com/rimorsoft.
3. Escribir y manifestar tu confianza a través de Slack y mi email.
4. Apoyarme y seguirme en Instagram, Facebook y Twitter.

Y en general gracias y mil gracias por todo lo que haces. *Cada like, cada comentario, cada vez que compartes y toda acción me ayuda mucho a mejorar.*

– Profesor, Italo Morales F.

Posibles Errores

Toda obra tiene varios filtros de calidad, pero en quien más confío es en ti, eres el mejor control y supervisor de calidad, si te consigues con algún error te pido paciencia y tu ayuda.

Puedes mejorar esta obra enviando un email con cualquier error que hayas encontrado a i@rimorsoft.com con los detalles necesarios para editarlo y publicar la nueva versión.

Los errores serán corregidos conforme se vayan descubriendo. Estás ayudando a cientos de personas que como tú han adquirido este libro. Recuerda que es un producto online y ayudaremos a muchas personas a mejorar sus habilidades profesionales.

> Los reconocimientos los haré publico en esta misma hoja.

Introducción

Bienvenida

Hice un gran libro llamado TDD en Laravel y esto sería una introducción al tema, en este libro encontrarás un gran por qué. En otras palabras:

1. ¿Por qué es difícil?.
2. ¿Por qué te podrías oponer? o ¿Por qué otros a mi alrededor se podrían oponer?.
3. ¿Por qué TDD?.
4. Y algunos más.

Quiero a través de este texto ayudarte a comprender cómo desarrollar el hábito de hacer pruebas, que como nuestro libro TDD en Laravel dice "podrás dormir tranquilo". Sabemos que nos hará mejorar la calidad de nuestro trabajo y a largo plazo nuestro nivel profesional aumentará de forma natural.

Si sabemos que esto tiene muchos beneficios por qué entonces nos cuesta aprenderlo y hacerlo un hábito.

- Si hablamos de Python deberíamos usar PyUnit.
- Si trabajamos con Java estaría recomendando JUnit.
- Pero estamos en PHP, aquí nuestro amigo es PHPUnit.

Cada quien hace lo propio en cada lenguaje. Quiero que seas un programador con buenos hábitos.

Yo, Italo Morales ¿qué he ganado?. Enumeraré algunas cosas:

Trabajo con el estrés adecuado: Sabes, el estrés me llegó a tal punto que me empezó a dar vértigo "es una sensación de movimiento o giros, todo te da vueltas".

¿Por qué estrés?... Sientes mucha presión cuando un sistema está sin pruebas y ya lo están usando mas de 60 personas (vendedores, técnicos, gerencia y contabilidad) porque al hacerlo así se desarrollan nuevas funciones y estas a veces dañan los módulos antiguos. Cuando algo fallaba sonaba mi teléfono y mi celular, muchos mensajes, emails y créeme, esto es para volverse loco.

> Al trabajar con pruebas el trabajo proporciona el estrés propio del trabajo y no por errores cometidos al implementar nuevas funciones.

Gané certeza: Esto es vital, tener el conocimiento, estar seguro y claro te hará sentir profesional y capaz.

¡Ahora me siento genial!, desde que aplico estos datos puedo decirte que tengo el estrés adecuado o propio del trabajo y tengo certeza.

¿Te familiarizaste con lo que me pasó?, todo depende de la cantidad de años que tengas como programador.

Este corto libro va mucho más allá de lo que puedes esperar. Quiero ayudarte con lo siguiente:

1. ¿Por qué deberías escribir pruebas?.
2. ¿Cómo realmente te ayudan?.
3. ¿Por qué existe resistencia con el tema?.

Tanto si eres aprendiz o experimentado este libro de igual forma te ayudará a adoptar o mejorar tu nuevo hábito.

Hay otro factor importante que nos ayudará a mejorar nuestra calidad de vida y es el ingreso, puedes saber mucho y puedes aportar grandes soluciones, pero si no trabajas con pruebas seguirás siendo programador junior y esto afectará directamente tu ingreso.

Te cuento rápidamente una experiencia con números reales. En una entrevista laboral me hicieron ver los diferentes niveles de salarios.

- Programador Junior: 600 USD
- Programador que sabe TDD: 1400 USD
- Programador que sabe TDD e Inglés: 2200 USD

Esto lo viví en el 2010 y por no saber pruebas ni la metodología TDD ni mucho menos inglés trabajé por un sueldo de 600 USD menos los descuentos de ley correspondientes.

No sé que edad tengas, pero si esto te sucede con esposa he hijos creo que vas a optar por buscar un empleo con mejor nivel económico pero alejado de este mundo que te apasiona. Sé que esto es así, créeme, me escriben desde muchos países contándome sus experiencias al respecto. Quieren programar y dedicarse a tiempo completo pero por diferentes factores prefieren vender autos, trabajar en ferreterías o en otros sitios que paguen mejor.

Lo que yo veo mas importante es que nuestro nivel profesional aumenta de forma exponencial. Yo llevo muchos años programando, pero nunca había aprendido tanto como cuando comencé con las pruebas de mi código, para ser más claro contigo, nunca aprendí tanto como cuando incluí en mi código a PHPUnit.

¿Has tenido deudas y te has atrasado dos o tres meses?, SI… ¿Cómo te sientes cuando te llama un número de teléfono desconocido?, quizás te da ansiedad, miedo, rabia o angustia.

Bueno esto mismo pasará cuando programes sin pruebas.

¿Eres muy chico y aún no tienes deudas?, OK… ¿Cómo te sientes cuando tu novia te dice que tienen que hablar pero no sabes aún de qué?.

¿Eres aún más chico y aún no tienes pareja?, OK… ¿Cómo te sientes cuando tus padres dicen que en la casa hablamos y no sabes de qué exactamente?.

Ese miedo, esa incertidumbre, ese "no sé si funcione" se esfuma cuando programamos como profesional usando pruebas o en nuestro caso cuando usamos PHPUnit. Ese pensamiento fijo de miedo o angustia al presentar un proyecto desaparece.

Si crees que exagero probablemente que te falten líneas de código en tu carrera para que se te hagan reales estos ejemplos. No quiero asustarte, quiero enseñarte y ayudarte.

Esto es realmente valioso y no solo eso, tendrás lo que todo buen jefe quiere en su empresa, hablo de: Resultados reales y tangibles.

Quiero que veas algo, nuestro libro de TDD en Laravel te dará muchos conceptos y temas prácticos y aquí en este libro tienes la experiencia ganada.

Quiero que tengas éxito, te ayudaré siempre que pueda a que seas un mejor profesional en todo este mundo llamado programación. Todo lo escrito se basa en mi experiencia, lo que he usado para ganarme la vida entregando sistemas. Tu tiempo es valioso y si te consigues con un código de ejemplo será un ejemplo que hice con mis manos, lo probé y sé que te funcionará.

No creo ejemplos con foo, bar, baz, etc, que son por si no lo sabías variables de ejemplo para no señalar algo específico, es como cuando en la vida real decimos fulano, mengano y zutano o si nos vamos a las matemáticas verás que allá usan a la famosa X, Y y Z.

Siempre y de por vida te encontrarás con ejercicios prácticos, precisos y muy cercanos al mundo laboral.

¿Este libro es para ti?

¡Seguramente si!, pero te comentaré para quien está escrito realmente para que desde ya estés seguro de ello y puedas continuar con confianza.

- Has programado por un tiempo importante en PHP.
- Entiendes la importancia de PHPUnit y las pruebas.
- Te gusta aprender.
- Quieres hacer proyectos de gran alcance.
- Te has frustrado luchando contra muchos errores.
- No quieres irte a la cama con el problema en la mente.

¿Encajas en alguna?… Ya sabes si este libro es para ti, espero que si y nos sigamos viendo en el resto de páginas.

He hecho proyectos de más de un año sin usar PHPUnit ni mucho menos la metodología TDD, empecé bien con el desarrollo y luego de ocho meses solo pasaba mi tiempo reparando errores. Hoy día es muy fácil usar esta tecnología, vamos poco a poco aprendiendo y conociendo los términos que tienen que ver. Te enseñaré mucho porque esto se basa en mi experiencia.

¿Este libro te ayudará?

Yo enseño PHP porque es la tecnología que uso para ganar dinero, porque los proyectos que he vendido han sido escrito en este lenguaje. En su momento usé PHP puro y duro, luego CakePHP, CodeIgniter, Zend, Symfony y ahora Laravel.

He experimentado con Java, Python y otras tecnologías pero con quien he hecho cosas importantes es con PHP.

Tuve empresas, dirigí proyectos y muchas de esas entregas fueron un completo desastre, al momento de entregar el proyecto me daba cuenta en la sala de reuniones que habían errores nuevos y eso por supuesto postergaba el ingreso y además a un cliente viendo en vivo cómo resolvía el problema.

El resultado es algo común, había estrés y trabajaba muchas horas extras solo o con el resto de mis compañeros. El proyecto terminaba siendo visualmente hermoso pero nuestro código era un desastre lleno de errores porque nos pasábamos mucho tiempo parcheando.

Entonces, este libro realmente te ayudará porque se basa en la experiencia, he trabajado muchos años sin pruebas y otros muchos con ellas.

Hice un seguimiento importante y aquí conseguirás un nivel básico de conocimiento importante que necesitas para entrar en este mundo de las pruebas con el pie derecho. Recomiendo que sigas paso a paso cada palabra, concepto y página, el orden está planificado y tiene una secuencia que será muy provechoso a nivel educativo para ti.

> Si sigues este consejo las probabilidades de dominar esta tecnología crecen.

Parte 1

En esta parte del libro quiero mostrarte algo más que código.

Resolver los problemas en el hermoso mundo laboral necesitará algo mas que código y es criterio. En esta sección aprenderás los conceptos, algunos objetivos y te presentaré algunos casos importantes.

Capítulo 1: Conceptos

Diversión

Esto no parece ser un concepto, pero hablemos al respecto:

¿Te gusta el ajedrez? a mí sí y mucho, he descubierto que me encanta porque es un juego en el que constantemente mejoras ganes o pierdas. Para las personas aficionadas como yo nunca hay un techo, siempre subes y si sigues jugando no dejarás de subir, solo aprendes, aprendes y no paras de aprendes.

Para ser mejor en el ajedrez necesitamos aprender lo más básico, luego tácticas y estrategias. Pero más allá de ello necesitamos practicar constantemente, es decir, necesitamos "jugar".

Con el tiempo cuando avanzas mucho en el ajedrez necesitamos luego aprender y mejorar las aperturas, defensas y finales. Pero ¿por qué este orden?... Porque si somos buenos en apertura quizás no ganamos, nos vencerían en el medio juego y para qué aprender sobre finales si en el medio juego somos vencidos. ¿Ves la importancia del orden?.

Aprendiendo tácticas y estrategias nos motivaremos porque ganaremos algunas partidas y cuando pierdas sabrás en que te has equivocado. Recuerda que soy un aficionado del ajedrez no un profesional pero seguramente de seguir así conseguiré un buen nivel en el futuro. Sin embargo ¿entiendes la idea?.

Si llevo esto al mundo de **PHPUnit** es lo mismo y yo lo veo así porque para aprender tuve que encontrarle el lado divertido. En este mundo hay programadores excelentes y otros que están comenzando y al igual que en el ajedrez la constante práctica te ayudará a crecer cada día. Y esto significa convertirte más temprano que tarde en un programador Senior.

> Las pruebas existen para respaldar un resultado esperado, si obtenemos un resultado inesperado mejoraremos nuestro código hasta que la prueba pase.

PHPUnit es la herramienta y la metodología es TDD, recuerda esto para entender futuros conceptos. PHPUnit es el ajedrez y TDD es la técnica usada en el juego.

Existen otras herramientas pero como primer paso enfoquémonos en PHPUnit, verás que la continuación de este proyecto en Rimorsoft te ayudará con la meta de ser un experto en el tema. Ya estás afiliado, seguiré creando material al respecto para llevarte de la mano en este maravilloso mundo.

Seguro te has preguntado ¿cómo es que hay sistemas de más de diez años sin construirlo desde cero cada vez?. Es lo divertido, aquí lo descubriremos.

Sigamos con los términos: Este es un texto del libro de TDD en Laravel, donde hablo de los términos necesarios para entender el resto de conceptos y ejercicios. Principalmente quiero que conozcas y aprendas sobre el ciclo TDD. Veamos el famoso Rojo, Verde y Refactor.

El ciclo de TDD

Son tres niveles que la verdad me gustan mucho, el ciclo es **Rojo**, **Verde** y **Refactor** que se repite constantemente. Esto se trata de programar una mínima funcionalidad cada vez, comenzamos en Rojo luego pasamos a Verde y por último Refactor.

Rojo

Este color por si mismo indica que algo no esta bien, todo lo peligroso tiene el color rojo. En programación no es diferente, todo lo que vayas a programar empieza fallando, empieza en rojo. Si tienes por ejemplo un modulo login normal (email y password) y luego quieres hacer un login social, es normal que a la primera esto no vaya bien. Así que es normal, esto va a fallar y debes esperar que esto suceda.

Verde

Bueno ya sabemos que nuestro código ha fallado ahora debemos hacer todo lo necesario para que funcione, tenemos que llevarlo a verde. Aquí hacemos copia y pega, código estructurado y lo que sea para que nuestro código funcione. En otras palabras, debemos crear el código necesario para que la prueba pase.

Rojo: No pasó la prueba, el código falla. **Verde**: La prueba pasó, he creado el código para que funciones nuestro modulo.

Sabemos entonces que el código actual no es necesariamente óptimo, puede incluso ser difícil de leer y mantener, así que viene la fase de Refactor.

Refactor

Es aquí donde nos volvemos artistas, aquí mejoramos la estructura del código, agregamos comentarios, borramos lo innecesario y creamos código reutilizable… Aquí hacemos todo lo necesario para facilitar el mantenimiento futuro de nuestro sistema.

1. ¿Qué código se repite?.
2. ¿Es legible?.

Este es el paso más poderoso porque es ordenar, modificar y la prueba debe seguir pasando (debe seguir en verde).

¿Cómo sabemos si nuestro código es inestable?

Aprender TDD y tener como principio probar nuestro código es como mudarse de país (te lo digo como extranjero), ningún programador amigo lo hace y nadie hablará tu idioma. Pero de pronto comienzas a reconocer ciertas costumbres y jergas. En poco tiempo te sentirás como en casa y te vas a dirigir con fluidez. Cualquiera puede aprender esto. Todo lo que se necesita es ganas y tiempo. Tus habilidades mejorarán con el tiempo y vas a reconocer al instante las trampas que antes te volvían loco. Te vas a manejar casi que por instinto y llegará el día (más temprano que tarde) en que serás un gran programador.

Jerga de TDD

No soy fanático de memorizar datos, tampoco creo que tengo la última palabra a nivel de educación… Sé que este curso no será tu única fuente (la verdad espero que no lo sea). Si eres como yo, serás un curioso y pasarás horas buscando nuevas cosas en la Web hasta la madrugada para completar tus conocimientos.

En el proceso, sé que te encontrarás con palabras o una jerga realmente confusa. Voy en este momento a decirte lo que conozco de la forma más simple y cómoda para ti. Pero no sientas que debes memorizarlas, poco a poco esto irá siento parte de ti.

Estoy muy en contra de lo difícil así que confía en mí.

Functional Tests (Controlador)

Es una prueba creada con la intención de probar un controlador y por supuesto cada uno de sus métodos.

Unit Testing (Prueba unitaria)

Una prueba unitaria revisa una clase y uno o varios métodos, de esta manera te vas a asegurar de que el sistema funcione como esperabas. Es una prueba de comportamiento exacto y la vamos a probar de forma aislada.

Unit Test es igual a un objeto (no varios). Si esto falla sabrás dónde buscar y reparar.

> **Resumen**: Son pruebas automáticas para verificar el comportamiento de clases y métodos de forma aislada.

Integration Testing (Pruebas de integración)

Una prueba de integración revisa varias partes del sistema y por lo general usamos una base de datos aparte y probamos un conjunto de cosas, básicamente probamos cómo funciona cada parte del sistema en conjunto.

Model Testing (Prueba de modelo o entidad)

Esto podría ser una prueba unitaria, la intención aquí es probar un método de un modelo o entidad.

Acceptance Testing (Test de aceptación)

Tenemos pruebas funcionales, pero habrá casos en los que, aunque las pruebas estén en verde no funcionan como el cliente quiere. Esto es a lo que nos referimos con él termino "pruebas de aceptación".

Veamos desde otro ángulo, el código probablemente no cumple con los requisitos del cliente, sin embargo el código pasa todas las pruebas unitarias, las pruebas funcionales y las pruebas de integración, pero todavía no pasa las pruebas de aceptación. En otras palabras, prueba tu sistema como programador, pero también asegúrate de que cumpla con las expectativas del cliente.

Esto podría ser otro Functional Tests (Pruebas funcionales) y si, lo es, en este caso nuestro motor de pruebas se comportará como un usuario y probará las páginas, los botones, formularios, etc... Más como usuario y menos como un programador. Así nos aseguramos de que nuestro sistema se comporte de la forma en que esperamos. Otro enfoque, es pensar en esta prueba como una revisión exterior, otra forma de llamarla es "pruebas del sistema".

¿Muchos conceptos? - No te preocupes

Vamos a resumirlo, cuando vamos a Laravel veremos por defecto dos carpetas, una llamada `Feature` y otra llamada `Unit`, expliquemos entonces que son las pruebas unitarias y las pruebas funcionales.

1. **Pruebas funcionales (Feature)**: Son escritas desde la óptica o vista del usuario. Aquí probamos una funcionalidad que integra muchas cosas, si queremos probar que una página abre correctamente se simulará que recorremos el sistema, se ejecutarán rutas, controladores, vistas, etc.
2. **Pruebas unitarias (Unit)**: Son escritas desde la óptica del programador. Aquí se quiere probar y dar garantía de una unidad o tarea especifica, por ejemplo un método de una clase.

Debes estar tranquilo

Esto muy poco se comprende a la primera si no ves ningún código y si no tienes algunas horas de práctica, por eso te digo que estas definiciones tomarán algún tiempo para aprenderlas bien, quiero que estes tranquilo si esto no lo comprendes del todo. De momento quiero que memorices los conceptos de Pruebas funcionales (Feature) y Pruebas unitarias (Unit) y sigue adelante.

Lo más importante es que desde ya empieces a probar, poco a poco con el tiempo comenzarás a fluir. No importa cómo, solo empieza a probar tu código.

¿Por qué TDD funciona?

Hoy día esto es vital, de hecho uno de los aspectos más importantes es asegurarnos de que el código haga lo que esperamos que haga. Piensa que cada día será más complejo y al mismo tiempo difícil de depurar y probar.

TDD realmente ayuda y si algo sale mal, sólo hay unas pocas líneas de código que modificar y volver a comprobar. La verdad es que así los errores son fáciles de encontrar y corregir. La prueba se centra en el resultado, en que funcione funcionando, no en la forma en que lo escribiste.

TDD funciona porque vemos que el código escrito realmente funciona, porque vemos a través del terminal que nuestro código arroja el resultado esperado.

PHPUnit

PHPUnit es un Framework, es un entorno para realizar pruebas cuando programamos en PHP. El principio es "cuanto antes se detecten los errores en el código antes podrán ser corregidos".

PHPUnit & Laravel

PHPUnit es el estándar y Laravel el gran Framework que lo implementa. Laravel proporciona las herramientas que vamos a necesitar para escribir y ejecutar pruebas y por supuesto analizar los resultados. Con PHPUnit escribimos nuestras pruebas y en Laravel escribimos nuestro sistema.

Veamos un poco de lo que necesitaremos:

- **AssertTrue**: Comprueba o verifica que el parámetro es igual a verdadero.
- **AssertFalse**: Comprueba o verifica que el parámetro es igual a valor falso.
- **AssertEquals**: Compara resultados y verifica coincidencias.
- **AssertNull**: Comprueba que una variable sea null.

Esto es lo que conocemos como afirmaciones. Nos ayudará precisamente a afirmar que el resultado de nuestro código es el esperado. Estos conceptos los vamos a ampliar al momento de trabajar con **PHPUnit**.

Ejemplo Grafico

Primero creo la prueba, la ejecuto y esta fallará, es entonces cuando obtengo **ROJO**.

```
$ vendor/bin/phpunit
PHPUnit 7.5.9 by Sebastian Bergmann and contributors.

F                                                                   1 / 1 (100%)

Time: 69 ms, Memory: 4.00 MB

There was 1 failure:

1) ExampleTest::test_basic_example_true
Failed asserting that false is true.

/tests/ExampleTest.php:10

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

TDD dice que primero debo crear la prueba, luego debo crear el código para que la prueba pase y obtengamos el color verde.

Si creo el código y ejecuto la prueba obtengo lo siguiente:

```
$ vendor/bin/phpunit
PHPUnit 7.5.9 by Sebastian Bergmann and contributors.

.                                                                   1 / 1 (100%)

Time: 32 ms, Memory: 4.00 MB

OK (1 test, 1 assertion)
```

Ahora, si deseo mejorar “refactor” podré hacerlo, solo debo estar consciente de seguir recibiendo el mismo resultado... Podríamos decir que este paso solo trata de mejorar la técnica de desarrollo.

En próximos capítulos veremos un ejemplo sencillo que nos permitirá practicar, de momento quédate con el flujo de desarrollo o ciclo; rojo, verde y refactor.

Capítulo 2: Objetivos

Necesitamos mucho esfuerzo y paciencia para que en nuestro trabajo podamos incluir a las pruebas, de un día a otro inyectar PHPUnit no es fácil. Recuerdo que esto lo quise hacer en mi equipo y el primer año solo gané que instalarán PHPUnit en sus proyectos. ¿Por qué pasó esto?, porque yo seguía programando como antes, obviamente el resto no hizo caso debido a que no les di el buen ejemplo.

Comencé los viernes a incluir capacitación y a través de esta educación comencé a descubrir porqué no las están implementando y la razón principal es que no comprenden su importancia. El siguiente paso fue adoptar esto como buena práctica y por último en estas clases de los viernes hacíamos un debate sobre los diferentes problemas encontrados y cómo uno a uno se fue resolviendo.

Nuestro enfoque cambió, ahora sí nos sentíamos programadores.

Motivación principal

Necesitamos un primer paso, si hacemos el cambio de forma abrupta tendremos problemas, comenzamos con lo siguiente: (Regla firme) Para que tu código sea usado y puesto en producción debe estar respaldado por pruebas.

Esto me motivo a mí y motivó a los chicos porque ver como las pruebas pasan es cool y si luego de esto tu módulo era aceptado podíamos ir tranquilamente por un café y conversar por un momento de lo que sea en la sala de reuniones. La disciplina siempre trae excelentes resultados, esto con un poco de capacitación nos trajo gran satisfacción y sin darnos cuenta desarrollamos una bonita cultura donde todos aprendíamos sin egoísmo y nos apoyábamos en gran medida.

¿Alguien sabe más que tú en el equipo? ¡EXCELENTE!… No hagas nada más que aprender de él con respeto, admiración y aprecio.

El primer obstáculo

Este primer obstáculo por lo general no se logra pasar y por ello no aplicamos pruebas en nuestro proyecto. Este obstáculo es: ¿cómo hago mi primera prueba?.

La respuesta: Empieza con lo básico, lo importante es ir incluyendo las pruebas, por ejemplo puedes inicialmente probar si tu ruta raíz responde, es decir, que tu ruta raíz te retorna un estado HTTP 200. Es como ir directamente a nuestro navegador y escribir el nombre de nuestra página si abre entonces todo OK, si no abre entonces resolvemos el error y lo volvemos a intentar.

Si es tu primera vez y necesitas hacer una prueba entonces busca hacer la más sencilla. Ya te has comprometido a hacerlo, ahora haz poco a poco algo así sea algo sencillo, terminará con una sonrisa cuando veas a tu prueba funcionar.

No te quejes, he oído "no he podido avanzar por estas pruebas…", ahí no hablas tu ahí habla tu rabia por ti. Cuidado con eso, solo busca aprender. Los buenos programadores sienten orgullo por sí mismo cuando crean un código que se puede probar. Esto es una nueva habilidad, pasa a través de ella con buena actitud.

Probar un nuevo módulo

Esto debe ser más fácil que probar el código que ya existe, partir de aquí ayuda mucho porque la prueba nos va guiando y ayuda en gran medida a tener un código limpio, así que si tienes que desarrollar un módulo nuevo enfócate en las pruebas, piensa en el módulo, planifica y no vayas programando de forma improvisada.

Piensan en:

Quiero probar que sé pueda listar, que cada elemento pueda ser accedido desde allí, que esta vista me permita ver el botón de editar y que además pueda borrar. Entonces ¿qué hago?.

Ya sé que quiero, ahora vamos dividiendo.

Cada prueba debe ser pequeña, inicialmente probemos que podemos entrar en esta ruta y que luego podemos ver el listado de elementos. Ahora con esta prueba, vamos a programar… ¿Lo lograste?, entonces continúa con las siguientes funciones.

Creo con una prueba nuevamente pequeña para probar que puedo hacer clic y ver el detalle de un elemento… Y así.

El ciclo es: Creo una prueba pequeña, luego desarrollo el código para que la prueba funcione y así continuo hasta la próxima prueba.

Esto ayuda mucho porque para un nuevo código debe existir inicialmente una prueba, funciona perfecto para todo código nuevo. Y no sé si lo notaste, pero aquí lo que estamos ganando es la habilidad de programar lo realmente necesario y específico. La estrategia real es escribir el código necesario y no el famoso "código futuro".

¿Qué pasa con el código viejo?, esto si es realmente difícil, a veces los códigos sin pruebas son difíciles de probar porque el código en si no lo permite. En ese caso sería bueno escribir la prueba y buscar hacer ese bloque de código de nuevo.

Nada será automático en este contexto explicado, es mejor seguir adelante aplicando pruebas que culpar al programador anterior o criticar al código ya escrito.

¿Cómo sé que estoy programando bien? o ¿cómo reconocería a un sistema que está bien programado con pruebas? ambas son válidas, veamos algunos buenos indicadores:

1. Una clase tiene pocos parámetros.

2. Mi nuevo código no cambia el que ya está desarrollado.
3. Una clase retorna un valor y no muchos.

Piensa en ello y crearás una prueba y un código fácil de probar.

¿Has visto código estructurado dentro de un método?, de ello te hablo, un código así es difícil de probar y si te lo consigues lo recomendable es crear a la prueba y al nuevo código en la medida de lo posible.

¿Cuándo dejar de probar un código?

Existe TDD, se trata de:

1. Crear la prueba y esperar ROJO.
2. Luego crear el código hasta obtener VERDE.
3. Terminamos con REFACTORIZAR.

¿En realidad qué es refactorizar?

Es cuando modificamos un código que funciona (si el código aún no funciona no es refactorizar, es escribir y reescribir código)… La única intención de modificar un código que funciona es la de aplicar nuevas y mejores técnicas de programación, luego de refactorizar debemos seguir obteniendo el mismo resultado, el resultado anterior.

Recuerda, la salida de este código no debe cambiar, estas básicamente mejorando el proceso o la forma de llegar a ese resultado.

> **Nota**: Si el resultado cambia entonces no es refactorización, es escribir o reescribir código.

La pregunta es ¿cuántas veces se debe refactorizar? o ¿cuándo dejar de hacerlo?… La respuesta no es nada científica ni avanzada, solo repite los pasos propios de TDD hasta que estés satisfecho con tu código y resultado.

Si en el futuro aprendes algo nuevo y te apetece refactorizar puedes hacerlo, solo te debes asegurar de no alterar el resultado y de seguir obteniendo VERDE.

¿Qué hacer cuando nos salta un error en producción?

Estamos trabajando con pruebas, sin embargo, nos saltó un error que no probamos ni esperábamos, no nos dimos cuenta, algo sucedió… ¿Qué hacer?.

Esto puede suceder y lo que debes hacer antes de repararlo es crear la prueba para verificar mediante una prueba este nuevo error, debemos escribir la prueba y luego el código correspondiente hasta obtener VERDE.

En resumen:

1. ¿Nos salió un error?.
2. OK. Escribimos la prueba para que sea la prueba y no el navegador quién nos dé el error.
3. Escribimos el código para que la prueba pase, es decir, hasta obtener VERDE.

Es normal que algo se nos pase, con el tiempo esto debe ir mermando, sin embargo sí sucede ya sabes el consejo. Esto te dará una mejor óptica porque pruebas el sistema en condiciones normales y además probamos con datos y parámetros falsos para ver cómo manejamos esos errores. Imagina que hacemos una clase que espera una variable, pero luego alguien sin saber envía un array obviamente el sistema fallará. Debemos prepararnos para enfrentar correctamente estos escenarios.

Verificar errores es una muy buena estrategia. Nos permitirá revisar a fondo nuestro código, la experiencia del día a día te llevará cada vez a nuevos niveles de pruebas. Es tu oportunidad para aprender cada vez más.

Cómo evitar problemas entre el equipo

Es muy importante programar con `Composer` como gestor de paquetes, de esa manera nos aseguramos que todos los programadores del equipo tengamos la misma versión de todo, sin embargo, si por casualidad estás usando PHPUnit 7 y tu compañero PHPUnit 8 busca la manera de emparejar, o tu subes tu versión o tu compañero la baja, pero deben llegar a un acuerdo para el mayor bien.

Si hablamos de PHP, toma en cuenta este mismo consejo, todos con la misma versión para que a partir de allí manejen los mismos componentes y versiones.

Esto puede escalar a otros sistemas, deben incluso asegurarse de usar entornos gemelos que por supuesto incluye bases de datos y herramienta de desarrollo, por ejemplo todos deben estar usando Sublime Text v3, etc.

Así vas a eliminar errores extraños por diferentes comportamientos y entornos, esto es crear un estándar. Yo lo solucioné llegando al acuerdo de siempre trabajar con la última versión estable de todo y dando a conocer cualquier cambio en las reuniones.

Estándares

Esto es muy importante, todo el equipo debe llegar a un acuerdo, es complicado encontrarse con lo siguiente:

1. Usas cuatro espacios de sangría y algún compañero dos espacios.
2. Usas como editor de código Sublime Text y algún compañero Visual Studio Code.
3. Comentas tu código y tu compañero no.
4. Comentas en Inglés y tu compañero en Español.

¿Me explico?, es importante acordar un estándar y un estilo para hablar el mismo idioma siempre y así evitar problemas o roces personales. ¿Esto afecta a las pruebas? SI y al tiempo al proyecto entero, así que atento.

Elige una forma de hacerlo, documenta el estándar y hazle saber al resto del equipo cómo se debe trabajar.

¿Quieres un código que pueda mantenerse?, entonces llega a acuerdos con el resto del grupo sino dile adiós a las buenas prácticas y código limpio a mediano plazo.

> La idea es que todos se lleven bien y puedan crear y probar código propio y ajeno, se trabaja en paz verdadera siguiendo estos consejo.

Si creas una mejora para Laravel siguiendo tu estándar por ejemplo: Comentando en español, programando estructuralmente, etc tu mejora será rechazada. Cuando un código sigue buenas prácticas y una estructura debemos tener como principio seguir el estándar correcto o establecido por el grupo.

Esto te ayudará incluso a tener muchos amigos.

Capa de revisión

¿Sabes quién se molesta con la revisión? ¡los sabelotodo!, no creo que seas uno de ellos porque estás aquí estudiando… Si aún te queda un poco de sabelotodo no te sientas mal, porque estás mejorando y ya dejaste de serlo.

> Un sabelotodo no aprende nada más porque cree que ya sabe todo de todo.

Hubo una época donde me revisaban el código y se me corregía muy seguido; a mí me molestaba mucho y discutía con las personas defendiendo mi código y mi técnica, cuando descubrí que no sabía tanto como ellos se me quitó la mala actitud porque cambié mi pensar y decidí aprender. **Mi punto de vista cambió cuando decidí aprender**.

Recuerdas la tabla de salarios en la empresa que te mencioné anteriormente, bueno, allí entendí porqué ganaba tan poco. La promesa de ascenso te ayuda a crecer.

Revisar el código ayuda tanto a quién crea el código como a quién lo revisa, esto significa que estamos muy seguros de lo que estamos subiendo al servidor.

Si hablamos de proyectos personales este paso no aplica, digamos, nadie me revisa el código de `rimorsoft.com`, pero si hablamos de una empresa o de un grupo de programadores este paso es muy importante porque agrega una capa de calidad. Tu código y forma de pensar mejorará en gran medida.

La idea no es hacerte ver en qué has fallado, la idea principal es beneficiar al proyecto con un gran resultado final.

Siguiendo la metodología TDD:

1. Creas la prueba.
2. Desarrollamos el código.
3. El encargado lo revisa y valida que tu código está respaldado por pruebas.
4. Subimos a GitHub por ejemplo a la rama de desarrollo.
5. El encargado cuando lo crea conveniente lo sube al master.
6. ...Y así hasta obtener un gran resultado final.

No es muy bonito que te digan que el código que llevas haciendo dos días no está correcto en algunos puntos o que podría ser mejor. Entender que todo esto al final va a beneficiar al proyecto nos hará recibir de mejor manera estas críticas. Lo importante es que no justifiques cada cosa que te digan.

Otra cosa importante: Quien revise o verifique el código debe ser muy educado y debe decir las cosas con tino, tacto y en general de una manera muy amable porque él debe hacer recomendaciones de mejora, críticas o comentarios de la idea y código no de la persona. Es muy importante esto para mantener buenas relaciones interpersonales.

El propósito real de las pruebas

Vamos con un ejemplo: Tenemos el módulo de usuarios y me interesa que alguien revise las funciones de listar, registrar, hacer login, luego salir del sistema, etc.

Luego tenemos el módulo de artículos y quisiera que esta misma persona haga las pruebas, de nuevo quiero que pruebe si puede listar, editar, ver, agregar, etc. Al mismo tiempo quisiera que revise de nuevo el módulo usuarios, quiero confirmar que nada se ha dañado... Cuando el sistema crece y se vuelve en un gigante sabemos que nuestro ayudante no revisará nuevamente todo. Él piensa "esto ya sirve" y avanza, es el problema en sí, en cambio creando pruebas logramos que se revise siempre absolutamente todo cada vez que queramos.

Otra cosa, por alguna extraña razón cuando hacemos pruebas también nos volvemos buenos comentando nuestro código y esto ayudará mucho a la hora de crear manuales de nuestro sistema, de nuevo, ya no le pedimos a alguien que cree esta documentación, esta se crea de forma automática a partir de los comentarios. Aquí comprendemos que mientras más pruebas hagamos más sentido cobra nuestro propósito.

Para que no te quede la duda de cómo crear la documentación a partir de los comentarios, para ello usaría **PhpDocumentor** o **Sami**. Ambos crean y generan la documentación de código, automáticamente analiza el código y produce la API de lectura, así de fácil tendríamos nuestra documentación generada y lista para revisar.

Certeza

Ya comenté algo al respecto, pero aquí quisiera extenderme un poco mas. Certeza es conocimiento pleno, es saber realmente porqué nuestro código funciona como se espera. Eso es certeza y cuando tienes ello no hay nada mas que haga falta.

1. Escribimos las pruebas y hacemos que pasen.
2. Seguimos un estándar.
3. Aumentamos una capa de revisión.
4. Entendemos el propósito real de la pruebas y capacitamos a nuestro equipo.

Y así cada punto y cada cosa.

¿Siguiendo estos textos al pie de la letra crees que tendrás éxito? pues no tengas la menor duda. Así se crea con el tiempo un verdadero profesional. Un código guiado por pruebas garantiza el éxito y te hará mejor programador.

Te hablé incluso de algunos temas y problemas personales por sabelotodo, y de eso se trata, de enseñar a partir de mi experiencia siempre dando a conocer nuevas técnicas y tips que te pueden servir en el mundo laboral.

He demorado meses desarrollando sistemas sin pruebas solo para saber que luego de un año solo vivía corrigiendo y creando parches.

En programación llamamos parche a esos cambios que solo se hacen para corregir errores. Te digo con sinceridad, yo di un salto profesional cuando incluí a PHPUnit en mi rutina de trabajo, ya te detalle cómo lo logré… Funcionó muy bien en mi vida y desde entonces enseño programación. Se debe trabajar duro para lograrlo, a partir de aquí vas a confiar en tus compañeros y en tu código.

¿Crees que vale la pena?

Capítulo 3: Algunos casos

Veamos algunos textos de interés entre ellos programar sin pruebas y mas para tener mejor información de valor al respecto.

Cuando empezamos en este mundo lo único que queremos es crear algo, nunca se nos dijo nada respecto de las pruebas y en PHP menos que tiene fama de ser exageradamente flexible. ¿Puedo programar sin pruebas? pues la respuesta es SI.

Esto en cierta medida está bien porque PHP es un lenguaje de programación y PHPUnit es un Framework de pruebas que uno instala de forma opcional, si estamos empezando en este mundo no estamos obligado a saberlo. Yo encontré que esto existía por casualidad, en el 2010 mi equipo de programación habló de esto y yo no lo entendía pero supe que eso llamado PHPUnit era importante.

Entonces, ¿se puede programar sin pruebas? claro que si, tú y yo lo hemos hecho pero aprendamos cómo mejorar nuestra técnica.

Llevar un proyecto a producción sin pruebas

¿Por qué esto es posible?

1. Estás empezando y no sabe de la existencia de las pruebas.
2. Es difícil de aprender.
3. Solo sabemos resolver.
4. En algunas empresas piensan que esto es una pérdida de tiempo. Recuerdo que en el 2015 un empleador me dijo “entiendo que pruebes tu código pero también enfócate en programar”.

Si estás empezando es probable que avances sin pensar en las pruebas, me atrevería a decir que algunos avanzados tampoco prueban su código, teniendo como resultado un buen sistema que funciona pero que a otro programador le costaría mucho entender para extender y mantener.

Una prueba es un código, un robot que uno crea para que pruebe código y que este se comporte como esperamos. Es algo que si se explica de otra manera será muy difícil de entender.

Fíjate lo que me dijeron; “entiendo que pruebes tu código pero también enfócate en programar”, eso quiere decir que esta persona entiende que es una pérdida de tiempo. Otros pueden pensar en que programan bien, que lo hacen solos y que nunca han visto la necesidad de incluir esta técnica a su trabajo. La verdad del asunto es que es difícil avanzar sin pruebas, tu código se llena de errores y es difícil dar mantenimiento.

Yo lo pondría así porque he vivido las dos experiencias (programar con y sin pruebas) el tiempo que "pierdes" haciendo una prueba lo ganas al no corregir errores futuros. No es bueno enterarse de qué tienes errores en producción ni mucho menos frente al cliente en el momento que haces la entrega.

Hay mucho valor en la escritura de pruebas, sin embargo, si deseas avanzar sin ellas sabes que te encontrarás con algunos de los problemas aquí contados. Si has decidido hacerlo entonces ¡bienvenido!. Estos textos te ayudarán a convencer a quienes se opongan a tu nueva forma de programar.

¿Qué hacer para solucionar errores?

Encontrarse con un error es observar que la prueba no pasa, que a pesar de que hemos desarrollado ha saltado tiempo después un error o resultado inesperado. Básicamente esperamos un resultado y obtenemos otro. ¿Qué hacer?.

Creamos una prueba que enfrente el error para que sea corregido a través de nuestra prueba, es decir, creamos la prueba y sabemos que dará el error conocido, ahora desarrollamos el código para corregirlo realmente.

Debemos probar de forma individual y luego todo nuestro conjunto de pruebas. Lo más importante de esto es: Hemos corregido el error conocido y además hemos verificado que no hemos dañado nuestro código antiguo.

Enfrentar nuevos módulos

Puedes escribir muchas pruebas, la experiencia será vital para crear las necesarias y las realmente útil. Cuando desarrollamos y hacemos cambios lo más importante es hacer código que podamos entender luego para modificarlo si es necesario. Por ejemplo, podemos querer modificar un código sin cambiar su resultado ni su comportamiento. En otras palabras, estaríamos refactorizando o reestructurando de manera segura porque estamos haciéndolo mediante pruebas.

Aquí es donde TDD muestra su poder, porque a medida que desarrollamos podemos notar con mucha claridad si nuestra mejora está rompiendo otro módulo o código antiguo. Es hermoso pensar que nuestro nuevo código no está ingresando errores.

¿Por qué sabemos que esto es así?

1. Nuestras pruebas pasadas pasan.
2. Hemos escrito pocas clases y pocos métodos.

Al hacerlo así te darás cuenta de que sin TDD es difícil vivir.

1. ¿Necesito una nueva clase?.

2. ¿Crearé código nuevo?.
3. ¿Modificaré el código existente?.
4. ¿Es un módulo completamente nuevo?.

Sea cual sea tu pregunta hacer que algo funcione es la mejor manera de ver brillar a TDD.

¿Si escribo pruebas mi código es perfecto?

No necesariamente, lo importante es hacerlo porque reducimos el factor error, es importante hacer énfasis allí en que reducimos no eliminamos por completo la probabilidad de fallar.

Te engañaría si dijera que SI, digamos que las pruebas solo cubren eso que tu has dicho que cubra. Por eso en el libro de TDD vemos pruebas unitarias y pruebas de integración o funcionales para reducir al máximo cualquier posible función que se nos esté escapando.

> Un mal código con pruebas es mejor que no tener ninguna prueba en tu código.

Parte 2

Llénate de conceptos realmente utiles, quiero enseñarte los fundamentos, porqué esto es importante y cómo mantener la constancia.

Capítulo 4: Fundamentos de PHPUnit

Instalación de PHPUnit

Usamos para ello el gran poder de **composer**, piensa en PHPUnit como un componente más de PHP.

Debes tenerlo disponible, así que vamos directo a la carpeta y creamos el proyecto. En el libro de TDD en Laravel describimos con mucho detalle cada paso, vamos aquí a hacerlo directo.

Para instalar PHPUnit solo usamos el comando `composer require --dev phpunit/phpunit 7.*`, debemos estar dentro de la carpeta del proyecto que en este caso está vacía.

Ejecutando el comando `composer require phpunit/phpunit` sería suficiente pero es interesante saber que PHPUnit es un componente solo usado en el entorno de desarrollo por ello agregamos `--dev` y especificamos la versión `7.*` donde básicamente decimos que se instale la última versión de la etapa `7`.

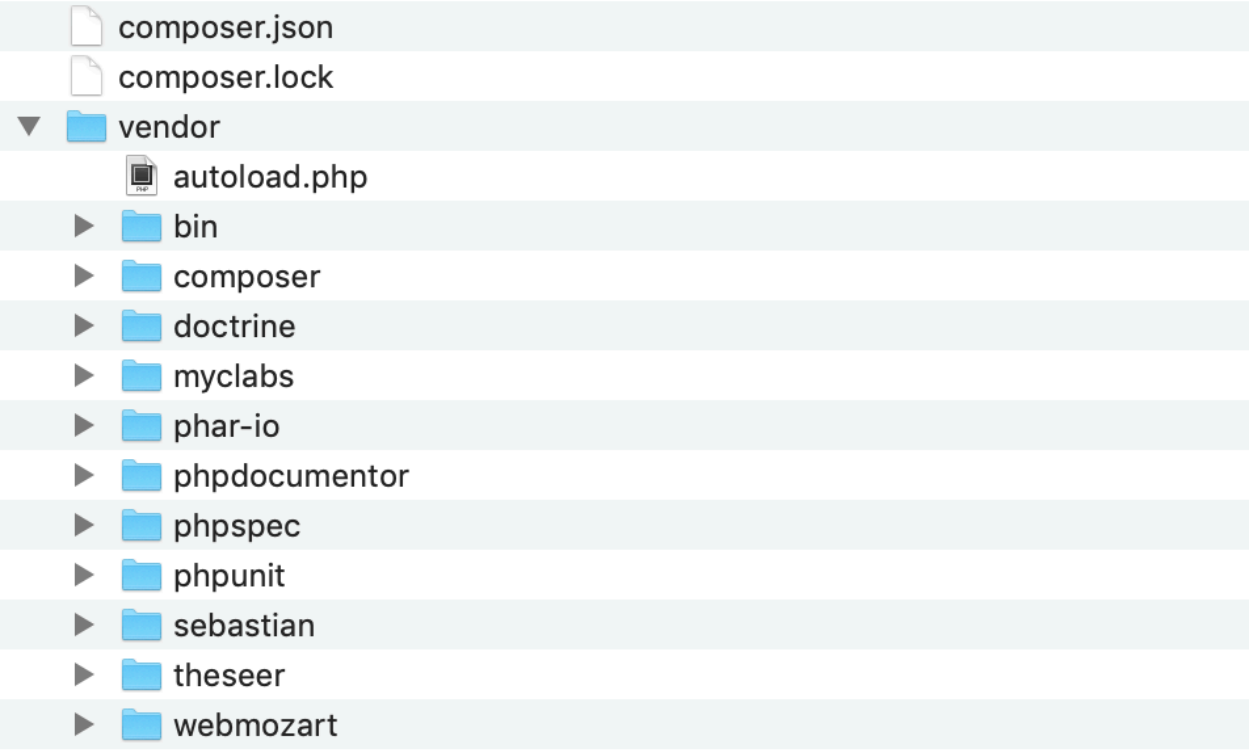
```
$ composer require --dev phpunit/phpunit 7.*
./composer.json has been created
Loading composer repositories with package information
Updating dependencies (including require-dev)
Package operations: 28 installs, 0 updates, 0 removals
  - Installing sebastian/version (2.0.1): Loading from cache
  - Installing sebastian/resource-operations (2.0.1): Loading from cache
  - Installing sebastian/recursion-context (3.0.0): Loading from cache
  - Installing sebastian/object-reflector (1.1.1): Loading from cache
  - Installing sebastian/object-enumerator (3.0.3): Loading from cache
  - Installing sebastian/global-state (2.0.0): Loading from cache
  - Installing sebastian/exporter (3.1.0): Loading from cache
  - Installing sebastian/environment (4.2.1): Loading from cache
  - Installing sebastian/diff (3.0.2): Loading from cache
  - Installing sebastian/comparator (3.0.2): Loading from cache
  - Installing phpunit/php-timer (2.1.1): Loading from cache
  - Installing phpunit/php-text-template (1.2.1): Loading from cache
  - Installing phpunit/php-file-iterator (2.0.2): Loading from cache
  - Installing theseer/tokenizer (1.1.2): Loading from cache
  - Installing sebastian/code-unit-reverse-lookup (1.0.1): Loading from cache
  - Installing phpunit/php-token-stream (3.0.1): Loading from cache
  - Installing phpunit/php-code-coverage (6.1.4): Loading from cache
  - Installing doctrine/instantiator (1.2.0): Loading from cache
```

```
  - Installing symfony/polyfill-ctype (v1.11.0): Loading from cache
  - Installing webmozart/assert (1.4.0): Loading from cache
  - Installing phpdocumentor/reflection-common (1.0.1): Loading from cache
  - Installing phpdocumentor/type-resolver (0.4.0): Loading from cache
  - Installing phpdocumentor/reflection-docblock (4.3.1): Loading from cache
  - Installing phpspec/prophecy (1.8.0): Loading from cache
  - Installing phar-io/version (2.0.1): Loading from cache
  - Installing phar-io/manifest (1.0.3): Loading from cache
  - Installing myclabs/deep-copy (1.9.1): Loading from cache
  - Installing phpunit/phpunit (7.5.9): Loading from cache
sebastian/global-state suggests installing ext-uopz (*)
phpunit/php-code-coverage suggests installing ext-xdebug (^2.6.0)
phpunit/phpunit suggests installing phpunit/php-invoker (^2.0)
phpunit/phpunit suggests installing ext-xdebug (*)
Writing lock file
Generating autoload files
```

Ya estamos por `8.1.*` al momento de escribir este libro, pero está bien trabajar con la versión `7.5.*`. En este caso se ha instalado exactamente la versión `7.5.9`... Es probable que pronto actualice los ejercicios a la versión 8.

Pero no hay apuro, la etapa 7 de PHPUnit tendrá soporte hasta el 7 de febrero del 2020.

Sigamos: Al culminar con la instalación logramos ver que ahora nuestro proyecto tiene algunos archivos y la carpeta `vendor`.



Estructura del Proyecto Inicial

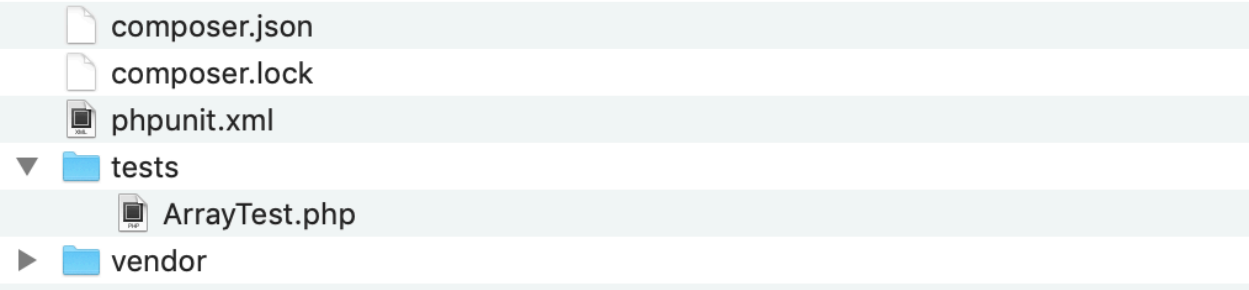
`vendor` por supuesto tendrá dentro los componentes necesarios con las versiones adecuadas.

Puedes revisar un proyecto Laravel y verás la misma estructura, hazlo y observa cómo esto que acabamos de instalar forma parte de todo un proyecto web como es el caso de Laravel.

De esto trata este curso, de una sección de todo proyecto, luego crearemos la carpeta `tests` y el archivo `phpunit.xml` necesarios también para lograr hacer algo con esta tecnología.

> PHPUnit nos trae poder con mucha simplicidad y enfoque.

Estructura del Proyecto



Estructura del Proyecto

La estructura ha crecido, ahora tenemos en nuestro proyecto la configuración necesaria. Allí puedes ver una nueva carpeta y un nuevo archivo.

Tenemos a `vendor`, `composer.json` y `composer.lock`, estos se han generado solos al momento de instalación de PHPUnit. Luego `tests` es la carpeta donde colocaremos nuestras pruebas, en este caso vamos a experimentar usando una clase llamada `ExampleTest.php`.

¿Qué hay dentro de este archivo?

```
1  <?php
2
3  use PHPUnit\Framework\TestCase;
4
5  class ExampleTest extends TestCase
6  {
7
8      public function test_basic_example()
9      {
10         $this->assertTrue(true);
11     }
12
13 }
```

De momento una clase con un único método. Mi clase se llama `ExampleTest`, si fuésemos a probar una clase llamada `User` nuestra prueba se llamaría `UserTest`, es lo correcto.

Nuestros métodos deben comenzar con la palabra `test`, en nuestro ejemplo tenemos `test_basic_example`.

Observa que tenemos el código `$this->assertTrue(true);` y eso no prueba nada en realidad, estoy diciendo que espero `true` pero le estoy enviando manualmente `true`. ¿por qué lo hago así? porque me interesa que en este momento te familiarices con la estructura de carpetas, de la clase y de la forma en que se escriben los métodos.

Otra manera de escribir el método sería `testBasicExample` y funcionaría... ¿y si uso español? también funcionaria solo debes asegurarte de que todo comience con el texto `test`, por ejemplo `test_basico`.

> No te recomiendo usar español por los acentos y caracteres especiales, quizás para tus proyectos personales puedes hacerlo pero cuando trabajas con un equipo el estándar es el inglés y lo correcto es mantener el estándar.

Para que esto funcione debemos configurar nuestro archivo `phpunit.xml`, allí definimos todo lo relacionado con este componente. Por ejemplo allí colocamos cuál es la carpeta de pruebas, invocamos el archivo `autoload.php` que está dentro de `vendor` y además activamos los colores, esto último es importante porque nuestras pruebas se ejecutan en una consola (pantalla negra con texto blando) y es interesante ver claramente el ROJO y luego el VERDE.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<phpunit bootstrap="vendor/autoload.php"
    colors="true">
    <testsuites>
        <testsuite name="Application Test Suite">
            <directory>./tests</directory>
        </testsuite>
    </testsuites>
</phpunit>
```

Mi Primera Prueba

Ya tenemos todo listo, tenemos una prueba en nuestra carpeta `tests`, a PHPUnit instalado y también completamente configurado a nuestro archivo `phpunit.xml`. Para ejecutar nuestra prueba debemos ir a la consola y ejecutar el comando `vendor/bin/phpunit`. El resultado sería el siguiente:

```
$ vendor/bin/phpunit
PHPUnit 7.5.9 by Sebastian Bergmann and contributors.

.                                                                   1 / 1 (100%)

Time: 26 ms, Memory: 4.00 MB

OK (1 test, 1 assertion)
```

Aquí básicamente estamos comprobando que estamos esperando `true` y realmente estamos obteniendo `true`, y por ser un ejemplo donde lo importante es la estructura de carpetas vemos directamente VERDE.

Esto es sencillo, pero es un ejemplo con fines educativos, en el mundo laboral nosotros usamos una clase real como `User` y tal vez nos interese comprobar si algún usuario está logueado.

Los ejercicios reales serían:

1. Probar que un usuario logueado pueda entrar al sistema.
2. Probar que un invitado (visitante, persona no logueada) no pueda entrar al sistema.
3. Probar que estoy guardando en una tabla de mi base de datos un registro.
4. Probar que un dato se está eliminando de alguna tabla correctamente.
5. Etc, etc.

Pero primero aprende la estructura y filosofía, en el libro de TDD en Laravel conocerás cómo hacer todo lo que te he mencionado.

Vamos a modificar nuestra clase `ExampleTest`, veamos la ampliación de este ejemplo:

```php
<?php

use PHPUnit\Framework\TestCase;

class ExampleTest extends TestCase
{

    public function test_basic_example_true()
    {
        $this->assertTrue(true);
    }

    public function test_basic_example_false()
    {
        $this->assertFalse(false);
    }

}
```

En un método probaría sí estoy obteniendo `true` y en el otro método probaría sí estoy obteniendo `false`.

¿Suena lógico? solo si imaginamos los ejemplo anteriores.

Recuerda el objetivo, nuestro objetivo al probar código es asegurarnos de que estamos obteniendo el resultado esperado.

1. Está logueado el usuario SI o NO.
2. Puede el usuario revisar el sistema SI o NO.
3. Etc, etc.

Y es allí donde colocamos o usamos `$this->assertTrue()` o `$this->assertFalse()` de acuerdo al caso.

Si ejecutamos el código tendremos como respuesta que hemos ejecutado dos pruebas y hemos recibido dos afirmaciones, en otras palabras hemos probado dos métodos y dos `assert` han pasado correctamente.

```
$ vendor/bin/phpunit
PHPUnit 7.5.9 by Sebastian Bergmann and contributors.

..                                                                  2 / 2 (100%)

Time: 51 ms, Memory: 4.00 MB

OK (2 tests, 2 assertions)
```

Lo correcto es tener por ejemplo una clase llamada `User` que debe tener su clase de pruebas llamada `UserTest`. En nuestro caso trabajamos con `ExampleTest` y no probamos una clase realmente, solo nos familiarizamos con esta tecnología.

En el libro de TDD en Laravel tendremos muchos ejemplos donde aprenderás a crear las clases y sus respectivas pruebas. Incluso verás ejercicios muy ilustrativos y correctamente explicados en el canal de Youtube, hay una serie gratis de PHPUnit que te acerca bastante bien al tema.

Capítulo 5: A quién le importan las pruebas

Podemos preguntarnos:

1. ¿Si soy independiente qué debo hacer?.
2. ¿Si a mi jefe no le importa este tema?.
3. ¿Si a mi cliente este tema le da igual?.
4. ¿Si mi salario es realmente bajo?.
5. ¿Debo cobrar mas sí programo con pruebas?.
6. Etc, etc.

Cada pregunta es una forma de excusarse y la verdad lo entiendo porque a veces vemos este tema muy difícil al momento de empezar.

Veamos qué hice al respecto.

Vamos a inspeccionar tu alcance

Las preguntas podrían ser:

1. ¿Si creas un código es tu responsabilidad probarlo?.
2. ¿Esta prueba debe ser desde el navegador o con PHPUnit?.

Siempre pruebas antes de presentar al cliente, no lo veas como algo adicional u opcional, siempre lo hacemos porque esto forma parte de tu trabajo, te paguen o no por ello siempre estamos probando… Esto te hace profesional, las pruebas te dan beneficios a ti directamente.

Si ganas muy poco para hacerlo entonces creo que puedes hacer dos cosas al respecto: 1) Consigue un mejor empleo o 2) Aprovecha ese empleo para aprender bien sobre este tema y ve tu sueldo como un incentivo por aprender.

> PHPUnit debe formar parte de ti.

Errores que cuestan dinero

Si estás trabajando en un nuevo módulo recuerda que lo ideal es tener pruebas que te digan que todo está OK, si sale algún error es fácil de resolver. Pero sí estás trabajando sin pruebas y subes nuevos cambios un jueves y detectan el viernes que ese nuevo código tiene errores habrá problemas.

¿Por qué pasó esto?

Ya vimos todos los puntos en objetivos que pudimos habernos saltados. Imagina que esta página es un comercio electrónico así que mientras siga el error no hay ventas. Tienes que arreglarlo urgente y esto en algunos casos cuesta tiempo y mucho dinero.

¿Arreglaste el problema anterior? quizás si, pero al programar sin pruebas no sabemos si hay otro error por allí. Eso es lo que sucede con grandes sistemas que ya están funcionando.

Ese arreglo fue un mientras tanto y así se va poco a poco construyendo un mal código.

Si un error así solo ocasiona que se deje de ganar dinero también se estaría perdiendo. Esta razón es importante, si hablamos de un proyecto grande podría haber incluso demandas laborales.

Un error realmente costoso es que el cliente empiece a perder dinero por fallas en el sistema o que por un mal código pierda la datas.

Las pruebas deben formar parte de tu flujo de trabajo porque corregir esos errores te van a costar a ti y en el peor de los casos al cliente que al final irá por ti para que le des explicaciones y te hagas responsable.

> Hacer pruebas es “encontrar errores antes de que el sistema se suba al servidor”. Ese es el objetivo puro y duro, es el objetivo y es realmente valioso.

Capítulo 6: Mantener la Constancia

Pueden haber dos casos y esto es muy interesante, podemos empezar sin pruebas para "hacerlas luego" o dejamos de hacerlas cuando el proyecto ya ha sido comenzado. La razón parece ser tiempo. Veamos:

"Tenemos que entregar pronto como para hacer pruebas", esto siempre es así ¿recuerdas lo que me dijeron en el empleo del 2015?.

Imagina si un albañil terminará un piso sin nivel solo porque está apurado como para ser cuidadoso y profesional, esta es una razón muy común para dejar de escribir pruebas pero no es lo que debes hacer, un albañil sin nivel podrá hacer un piso pero será el peor trabajo de su vida.

Te diré algo, cuando las pruebas se ven interrumpidas es porque no sabemos probar eso que queremos construir, por ejemplo: Queremos hacer un API pero no sabemos cómo probar un API. Ahí paramos, justo ahí detenemos las pruebas y lo mejor que podemos hacer es aprender mediante la investigación. No hay mejor manera de invertir el tiempo.

En el futuro cuando te encuentres con el desafío de probar un API ya tendrás el conocimiento y siempre podrás respaldar tu sistema con pruebas.

Piensa en que las pruebas deben ser parte de tu forma de trabajar, debe ser esto tú técnica de trabajo.

Aprende todo esto, la verdad es que aquí están los textos para que puedas convencer a quien se opone con tu nueva forma de trabajar.

Te dije hace un segundo que el factor "culpable" es el tiempo, partiendo de otra idea cuando tenemos un proyecto muy grande ejecutar las pruebas demora un poco y a veces mucho. ¿Por qué algunas pruebas son demasiado lentas?, básicamente por falta de técnicas respecto a las pruebas, esto se controla siguiendo esta ruta de aprendizaje. Lo importante en realidad es seguir aplicando esta tecnología, si te rindes ahora cuando empiezan a ser lentas entonces no llegaremos a un nivel avanzado en este mundo llamado PHPUnit.

La velocidad la conseguimos con pruebas dobles como Dummy, Fake, Stubs y Mocks. Mi recomendación, por ahora como estamos aprendiendo es que no nos metamos con estos conceptos, lo aprenderás más temprano que tarde porque el mismo sistema te llevará a ellos.

Cuando nuestro proyecto es pequeño las pruebas son rápidas, la lentitud empieza cuando se va haciendo más y más grande.

Las barreras

Podemos resumir este contenido con lo siguiente:

1. **El primer acercamiento**: Esto es lo que más cuesta en este mundo solo esfuérzate en hacerlo, luego de hacer esa primera prueba tendrás un poco de ritmo y podrás continuar hasta formar el hábito.
2. **Mantener la decisión**: Mantén una política firme en que las pruebas forman parte de tu trabajo y no dejarás de hacerlo, usa estos conocimientos para ayudar a tu equipo.
3. **Son muchos términos**: Es verdad, pero observa que te estoy enseñando por parte y con una metodología adecuada. Abrazarás esta metodología, solo confía en mi.
4. **La documentación de PHPUnit es difícil**: Esto también es verdad. Yo creo que el creador de esa documentación asume que sabes mucho al respecto y habla a quienes ya tienen experiencia en el tema. Además de que los ejemplos no son muy parecidos a lo que necesitamos realmente. Con la documentación obtienes el concepto pero no una aplicación real y clara, por eso este libro y el de TDD en Laravel son muy útiles.
5. **El tiempo**: Cómo ya conversamos, muchas personas lo consideran perdida de tiempo y un alargue innecesario en el desarrollo, pero ya sabemos que al no tener el panorama completamente claro al respecto esto se vuelve creíble. Es tu labor mostrar con paciencia y determinación los grandes beneficios. No se pierde tiempo, se gana y evitamos perder mucho dinero.

¿Debo irme de esta mala empresa? o ¿debo ayudarlos?. Cuando hablamos de ayuda nuestro punto de vista cambia, yo incluso lo vería como una forma de aprender sin tanta presión encima porque seria yo quién esté innovando.

Hazlo y realmente crecerás como programador.

¿Quieres ser un programador senior? ¡HAZLO, trabaja con TDD!, ese gran título te espera.

¿Qué debo hacer?

Primero que nada me gustaría saber tu opinión, este texto estará en constante actualización. Cada vez añadiré muchos conceptos y experiencias para dar al mundo un gran material, en esta etapa debemos mantener comunicación porque entre tú y yo podemos mejorar mucho este producto. Daré crédito a todo gran aporte y esto te ayudará mucho en tu historial profesional.

Dime a través de mis redes sociales o las de **Rimorsoft** qué te pareció, qué crees que deba tener y todo lo que creas conveniente.

Cuando haga la actualización esta te llegará automáticamente a tu email y siempre tendrás un libro actualizado. Te recomiendo dar el siguiente paso y obtener el gran proyecto de TDD en Laravel: Ve a https://rimorsoft.com/curso-de-tdd-en-laravel[1]. Desde allí te ayudaré de forma práctica cómo enfrentar este tema y tendrá la misma filosofía de entrega, será un solo pago y muchas actualizaciones.

[1] https://rimorsoft.com/curso-de-tdd-en-laravel

Para comunicarte conmigo tienes muchos medios, puedes usar el email i@rimorsoft.com[2] y lo ya mencionado como redes sociales y nuestro canal en Youtube.

Toma en cuenta a los podcast que están alojados en: https://rimorsoft.com/podcast[3] la idea es irte contando con ejemplos y vivencias propias la mejor manera de entender este tema y ponerlo en práctica.

Ya sabes que debes hacer…

Gracias por tu confianza :)

[2] mailto:i@rimorsoft.com
[3] https://rimorsoft.com/podcast

Capítulo 7: TDD en la vida real

Reflexión y resumen

Código Profesional

Esto es lo increíble, cuando comenzamos con TDD y resistimos en nuestro aprendizaje descubrimos algo mágico… "un código mal escrito no se puede probar así funcione a ojos del cliente".

Hacer un proyecto pequeño se siente fluido al principio, pero a medida que avanzamos en él y se hace más grande cada día podemos sentir el desastre o que algo malo se acerca. Esto tiene un costo y todos pierden; pierde el cliente, pierdes tu y pierde la empresa…

¿Te ha pasado que escribes lo que sea solo para entregar rápido?

Siempre he dicho que TDD te acerca al mundo Senior. Esta es la principal razón, TDD te obliga a desarrollar enfocado y te ayuda a avanzar garantizando que cada cosa esté respaldada por una prueba.

Si estoy apurado y no uso TDD "porque me quita tiempo" escribo código como loco hasta que funcione. Esto lo hice por años pero ya comprendí que los millones de problemas vendrán a largo plazo.

Un código profesional es aquel que se puede extender, modificar, cambiar, eliminar y finalmente testear.

Entrega a tiempo

Cuando programamos sin TDD el comienzo es super rápido, sin embargo, a medida que avanzamos (como escribes mal código) esta velocidad disminuye cada vez mas. Imagina caminar, luego caminar en agua y por último caminar en lodo. *Esos ejemplos de caminar son las etapas que atravesamos cuando trabajamos sin TDD.*

Ahora, cuando trabajamos con TDD el comienzo lleva un ritmos no tan rápido pero como estamos escribiendo buen código limpio y profesional, y además está respaldado por pruebas cada día avanzamos mas y mas rápido, es decir, en el proceso va aumentando esta velocidad.

Comenzar con TDD es duro, muchos lo abandonan precisamente porque no comprenden las ventajas debido a que nunca han tenido problemas como los que he mencionado en este libro.

Toma en cuenta y memoriza la siguiente frase "el enfoque de TDD es ayudarte en los proyectos grandes". Esta frase seguro te hará estallar la cabeza. Si es una Landing Page, una web con botones

comunes, un sistema de recepción o cualquier cosa de máximo un par de meses entonces no uses TDD. Pero si es el proyecto de tu vida, si este requiere meses de dedicación entonces hazlo sin pensar, ahí usa TDD.

Proyecto "sin errores"

Decir esta frase es mentira, para hacerlo mas real podemos decir "pocos errores" y no "sin errores". Lo que sucede es que son tan pocos errores que es como si no tuviéramos ninguno. Yo vivía corrigiendo errores, decía de forma chistosa que antes era programador y ahora soy un corrector.

Sentí mucha tristeza y frustración cuando un cliente me gritó por teléfono diciendo que su sistema no servía (la falla fue en el formulario de contactos, el sistema falló por no tener un email configurado, tenía `null` y no un email real).

Esto fue un despiste, yo tenía en un archivo llamado `config.php` y ahí dentro una llave donde registraba el email del formulario de contactos, por defecto estaba en `null` y al subir el proyecto a producción debía registrar ese email real. Cómo no lo hice el sistema mostraba un error cuando intentaban contactar a la empresa. Parece tonto, pero cuando es una empresa grande y viven de sus ventas online, este formulario es muy usado y todo el departamento de ventas está pendiente de su funcionamiento.

Uno de los argumentos del cliente eran los contactos perdidos y el dinero que dejó de entrar a la empresa por el error técnico, al mismo tiempo me preguntaba ¿quién me repone ese dinero?. Te podrás imaginar el pánico que sentía en el momento.

Para mantenerme en esta profesión debí aprender **PHPUnit**, sabía que esto me ayudaría a entregar proyectos profesionales.

Resultado: Reduje en gran medida este tipo de errores "tontos" y no viví nunca más una mala experiencia como la que te acabo de contar. Esto me hizo profesional, lo digo porque a si me sentía y me siento, con el tiempo he venido cotizando todos los proyecto a un precio mayor. De hecho uno de mis argumentos de ventas es la garantía y el rápido soporte.

> Por eso mi libro **TDD en Laravel** tiene impresa en la portada la frase "CRÉEME, PODRÁS DORMIR TRANQUILO".

Siempre existirán errores

Extendiendo la idea de **Proyecto "sin errores"** que acabas de leer quiero comentarte que siempre los errores existirán, quizás tu código es perfecto, pero puede que en algún momento falle tu servicio de base de datos, el servidor entero o alguna actualización del sistema operativo del servidor ha traído problemas. Esto siempre sucederá.

Solo comprende que un código profesional hará que ganes mas ingresos debido a que todos a tu alrededor estarán satisfechos con el resultado final y confiarán en ti.

Ganarás más porque evitas en mayor medida los errores, porque eres profesional, porque eres responsable, porque tu código se puede extender, etc. ¿Quién no quiere a un programador así en su equipo cueste lo que cueste?.

Solo pon de tu parte y domina a TDD.

Piensa en los programadores que vendrán

TDD te ayudará a tener mas amigos, cuando usas TDD tu proyecto será muy fácil de mantener y de extender. No importa si tu o el nuevo programador quieren agregar nuevas funciones o cambiar lo que ya existe. Este solo deberá crear la prueba y el código, luego ejecutará todas las pruebas y confirmar que nada se ha roto.

¿Alguna vez te ha llamado un programador de tu antiguo trabajo para preguntarte que rayos hiciste en equis módulo?, a mi si y yo también he hecho esa famosa llamada.

Si eso no te ha pasado entonces eres muy joven, el punto es que con TDD evitaremos pasar a través de esa situación.

El paso #3 de TDD

Hablo de la **refactorización** y es vital cuando desarrollamos un sistema. Te parezca o no, esto es tan importante que tengo pensado escribir un libro solo dedicado a este tema.

Con TDD estamos constantemente mejorando nuestro sistema, el reto de esto es no afectar a las pruebas y a los resultados existentes.

La documentación

No todo ha sido caos o tragedias en mi vida laboral, también me han llamado para que les explique algo concreto sobre mi código y les digo “por favor todo está en la carpeta tests, avísenme si hay algo ahí que no se comprende del todo bien”.

La pruebas son también una forma de documentar. Ahí se puede ver si alguna vista necesita login, qué se está validando, que datos requiere equis una tabla, etc. Las pruebas son como una receta de cocina y describen exactamente que hace el código. Esto permite que cualquier programador pueda navegar allí para entender rápidamente nuestro código.

Cuando NO usar TDD

Debería hablarte solo sobre usar a TDD, pero me gusta decir lo que es verdad para mi, la verdad es que no siempre debes usar TDD, veamos porqué:

El Proyecto ya existente

La verdad del asunto es que es muy difícil implementar TDD en un código ya escrito. Si lo intentas, en muchos casos te darás cuenta de que debes hacer el código desde cero. Es aún mas difícil cuando eres nuevo usando esta tecnología.

Con esto fui muy radical, ya no abordo proyectos existente a menos que el pago sea realmente bueno, si la oferta no es buena rechazo el proyecto de inmediato. En este caso en particular es mejor no usar TDD y continuar el proyecto con lo que ya sabemos. Pero si quieres fabricar un nuevo módulo en este proyecto existente, entonces es valido hacerlo con TDD.

Incluir TDD es reescribir muchas cosas.

1. Si el proyecto ya existe entonces uso TDD en las nuevas funciones.
2. ¿Uso TDD en lo que ya existe?... Depende, yo recomiendo hacerlo cuando queremos mejorarlo y tengamos tiempo para hacerlo debido a que casi siempre te tocará reescribir el código.

El problema principal es que al no comprender de inmediato lo que ya existe vamos a perder tiempo y dinero. En mi caso si llego a aceptar un proyecto existente aplico lo explicado en el punto (1) y (2).

> **La máxima es**: "Si el proyecto es grande y no tiene pruebas entonces ya tiene errores".

Es tu primera vez

No uses TDD si no tienes la experiencia necesaria, es probable que le digas a tu jefe o cliente que demorarás 15 días pero como a la vez estás empezando con TDD probablemente pasará mas tiempo y quedarás mal. Me he enterado de casos donde el alumno sabe algunas cosas y luego pierde mucho tiempo porque no sabe qué hacer cuando le toca testear una descarga, email o si se generó correctamente un archivo.

Si eres nuevo o si es tu primera vez con TDD te recomiendo esperar un tiempo para aprenderlo bien, estudia mis libros y videos y gana mucha seguridad y técnica. Da el paso solo si la empresa esta dispuesta a pagarte por ese tiempo.

Tus compañeros no saben TDD

Si estás avanzando en esta ruta de aprendizaje y tus compañeros no saben o no les interesa aprender entonces decide no usar esta tecnología. Te meterás en problemas si los obligas a hacerlo.

¿Cuál es el problema real?... El problema es que no avanzarán y la culpa será de las pruebas, en ese caso los clientes exigirán respuesta y no se les puede decir "es que las pruebas bla bla bla", si estás rodeado de Juniors entonces no uses TDD.

Si estas aprendiendo y tus compañeros no cuentan con la experiencia necesaria entonces no hagas lento el proceso queriendo implementar allí este nuevo método de desarrollo. Terminarán odiando a TDD y a ti también.

El cliente no está claro

Todos vemos clases de procesos, diseño y metodología pero en la vida real el cliente te llama y ocurre algo mas o menos así.

- **Día 1**: Quiero un sistema para registrar a mis clientes.
- **Luego de un par de días**: Y si le agregamos la función de enviar emails.
- **Tiempo después**: Vamos a exportar en PDF con estos filtros.
- **En la entrega**: Mejor hagamos el login con Facebook.
- **Otro par de reuniones después**: Cambié de opinión. Se me ocurre bla bla bla…
- …

No todos los clientes son así pero cuando te toque debes decidir no usar TDD… Es una locura tener pruebas y que luego de tres días tengas que borrarlas porque la decisión tomada por el cliente es otra. ¿Imaginas la frustración?, créeme que esto pasa sobre todo cuando es un buen cliente para la empresa y tú ganas solo por 8 horas de trabajo diario. Incluso dan ganas de llorar terminar tu hermoso código para luego tener que borrarlo todo.

Esto me pasó, este famoso cliente era muy bueno para la empresa porque pagaba mucho de forma mensual y cuando el quería llamaba a la empresa y hablaba conmigo sobre los nuevos cambios, a veces estas conversaciones eran tan largas que debíamos mudarnos a Skype para continuar la discusión.

Si el cliente es tuyo directamente quizás puedes hacer algo, solo debes aclararle que ha perdido el dinero porque te hace perder tiempo con las indecisiones y constantes cambios. Haz que tus clientes entiendan parte de tu mundo y grábate este frase “un cliente educado siempre te contratará”.

Tu cliente debe ser tu socio, hablen pensando en un mejor futuro. Si a tu cliente le va bien a ti te irá mejor.

Notas finales

Te agradezco y te admiro por este tiempo invertido. Dios premia la constancia, no abandones este tema ni tu camino profesional por nada en el mundo.

Tu aprendizaje... Incluye Regalo

Gracias por acompañarme en este proceso aquí te compartiré todo lo que aprenderás con el curso **TDD en Laravel** la verdad me siento escritor porque tu eres mi lector y eso te lo agradezco de corazón.

TDD en Laravel es un material educativo que incluye un libro de más de 200 páginas, ejercicios funcionales y videos prácticos, cortos, precisos y muy cercanos al mundo laboral (como nuestro canal en Youtube).

Quiero hacerte saber que esto es un proceso que incluye varios temas que ya has estudiado de forma gratuita.

1. **Libro**: TDD lo que debes saber (este libro).
2. **12 Videos**: PHPUnit.
3. **9 Videos**: Mockery.

Eso para que tengas una buena introducción sobre el tema, y además veas de primera mano mi estilo de enseñanza, ejercicios y material muy útil y gratis. Todo esto te ayudará por supuesto a tomar una mejor decisión.

¿ESTÁS LISTO PARA COMENZAR?...

Veamos el contenido del curso de forma general: He creado para ello un primer paquete.

- Lo esencial - Fundamentos
  - Libro con un poco mas de 200 páginas, en formato PDF.
  - Conjunto completo de ejercicios que acompañan al libro.
  - Mas de 30 videos donde hacemos un ejemplo práctico de TDD en Laravel.
  - **PLUS**: Todas las preguntas que me hagan las responderé en videos y las iré subiendo progresivamente para que todos puedan descargarse el material.

El precio de este primer paquete es $89, pero como motivo de celebración y al mismo tiempo agradeciendo tu apoyo y acompañamiento tienes en este libro un 25% de descuento:

Da clic en este enlace y disfruta tu curso: https://gum.co/curso-tdd-en-laravel

Detalle del precio:

- **Lo esencial - Fundamentos** ~~$89~~ $67

De nuevo gracias y recuerda siempre: **¡Un solo pago y muchas actualizaciones!**, todo feedback será tomado en cuenta, con ello se mejorará siempre este producto y lograremos al final un producto de máxima calidad.

Si tienes algún comentario recuerda que puedes escribirme y desde allí podemos conversar.

Muchas gracias por apoyarme en este gran viaje, ¡no puedo esperar verte estudiando este gran curso!

– Profesor, Italo Morales F.

Feedback

Puedes enviarme cualquier feedback que tengas sobre el contenido de esta obra o lo que quieras, mi email sigue siendo i@rimorsoft.com

Me voy a esforzar para responder a todos y a tiempo.

Haz tu importante aporte.

Actualizaciones del Libro

En este momento hicimos ejemplos en `PHPUnit 7.5.*`, la idea es continuar mejorando a la par de las versiones y por supuesto cada vez ir mejorando y agregando conceptos para ayudarte mejor en este tema.

Las próximas actualizaciones las tendrás en tu email.

Todos nuestros productos tendrán la filosofía de un solo pago y muchas actualizaciones. Vamos a mejorar siempre y a ayudar a muchas personas a través de estos textos, ejercicios y videos.

Contactos

Puedes ponerte en contacto para ayudarte en lo que respecta al mundo del desarrollo web. Los canales para entrar en contacto:

Rimorsoft Online

1. **Slack**: https://rimorsoft.slack.com[4]
2. **Web**: https://rimorsoft.com[5]
3. **Instagram**: https://instagram.com/rimorsoft[6]
4. **Twitter**: https://twitter.com/rimorsoft[7]
5. **Facebook**: https://facebook.com/rimorsoft[8]
6. **Github**: https://github.com/rimorsoft[9]
7. **Nuestro canal**: https://youtube.com/rimorsoft[10]

Profesor: Italo Morales F.

Te espero en:

1. **Instagram**: http://instagram.com/italomoralesf[11]
2. **Twitter**: https://twitter.com/italomoralesf[12]
3. **Github**: https://github.com/italomoralesf[13]
4. **Facebook**: https://facebook.com/ProfesorItaloMoralesF[14]

Hay dentro de **Slack** un canal dedicado al mundo de las pruebas, entra, ayúdate y ayuda a otros en #PHPUnit.

[4] https://rimorsoft.slack.com
[5] https://rimorsoft.com
[6] https://instagram.com/rimorsoft
[7] https://twitter.com/rimorsoft
[8] https://facebook.com/rimorsoft
[9] https://github.com/rimorsoft
[10] https://youtube.com/rimorsoft
[11] http://instagram.com/italomoralesf
[12] https://twitter.com/italomoralesf
[13] https://github.com/italomoralesf
[14] https://facebook.com/ProfesorItaloMoralesF

Guía de Futuros Pasos

> Una pregunta ¿cuál debería ser tu siguiente paso?.

Esta es parte de una obra mucho más extensa. No tienes que preocuparte, tengo grandes planes para ti. Seguiré escribiendo y te quiero recomendar el proyecto **TDD en Laravel**.

Esta y todas las obras relacionadas con TDD y PHPUnit tendrán sus respectivas actualizaciones para ayudarte siempre a mejorar tu técnica de trabajo.

Siempre recibirás un email al momento de realizar alguna actualización. ¡Tan sencillo como eso!. Puedes hacerme sugerencias vía email o redes sociales, hazme saber qué deseas aprender en un futuro cercano y cómo puedo mejorar estos temas.

Gracias una vez más.

Sinceramente,

– Profesor, Italo Morales F.

Profesor - Italo Morales F

Profesor Italo Morales F, CEO & Founder de **Rimorsoft Online**, la voz de los videos que estudias en el canal de Youtube.

Mediante este proyecto educativo estamos ayudando a muchas personas, sigo adelante porque en mi email hay muchos mensajes de agradecimientos y muchos mensajes de aliento para seguir adelante.

> “Me gusta ayudar y considero como mi mayor placer ver a las personas crecer, formar sus empresas o escalar velozmente como empleados”.
>
> – Profesor, Italo Morales F.

Mensaje Final

Con esta página finalizamos…

Sé que sabrás hacerlo bien, creo en ti y no dudo de que conseguirás dominar está tecnología, da el paso con **TDD en Laravel**, sigue estudiando, lo vas a lograr.

¡Que sigas estando bien!.

– Profesor, Italo Morales F.