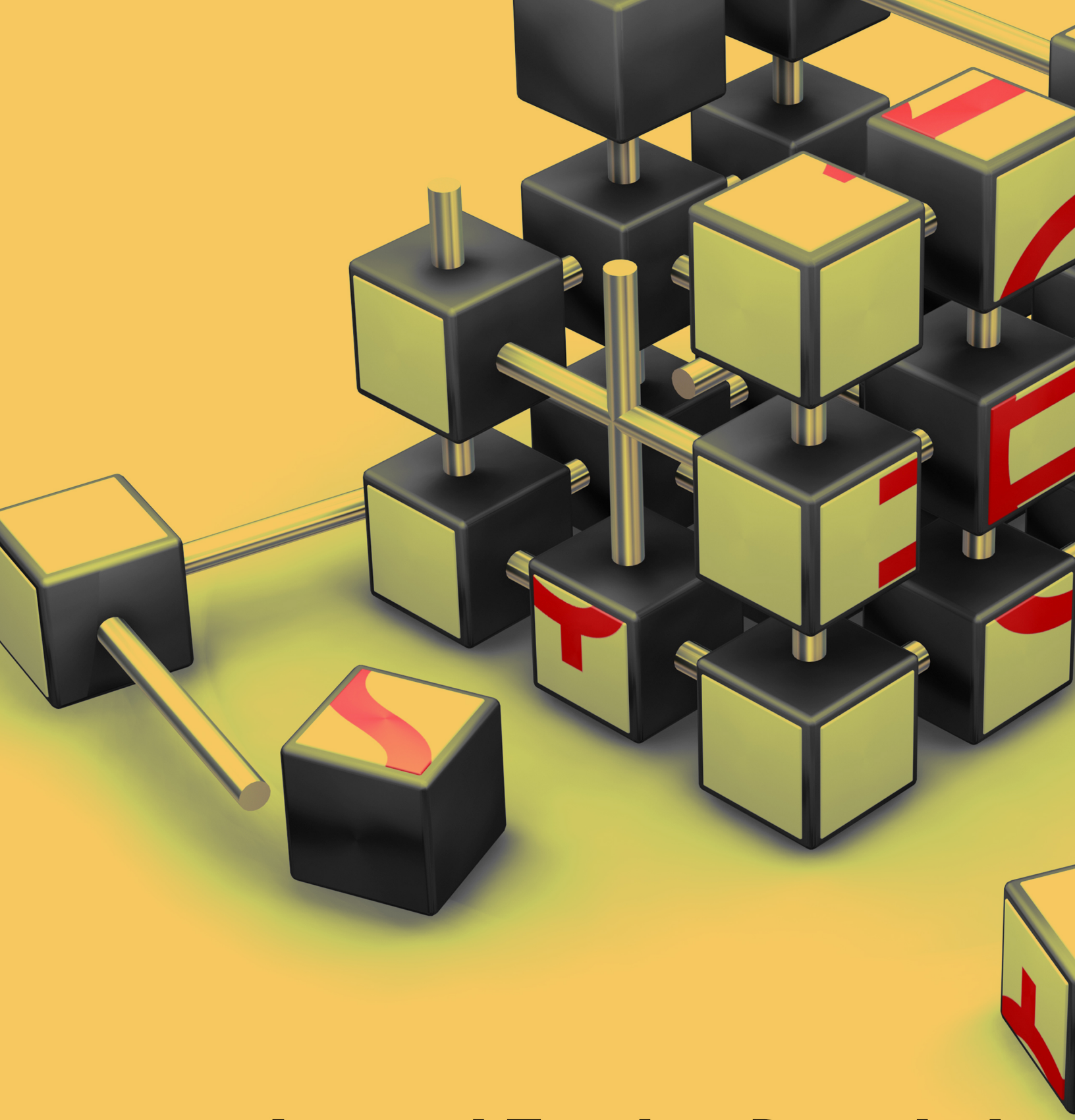


Laravel Testing Decoded

...with Jeffrey Way

Laravel Testing Decoded (Italian)

Il libro sul testing che stavi aspettando.

JeffreyWay and Francesco Malatesta

This book is for sale at http://leanpub.com/laravel-testing-decoded-italian

This version was published on 2014-06-04

Indice

Benvenuto

L'hai visto tante, troppe volte. La tua applicazione cresce e la stessa cosa succede per la tua immensa codebase. Rigorosamente non testata. Molto prima di quanto tu riesca ad immaginare, ti ritrovi li, impantanato e recuperare il terreno perso diventa qualcosa di irrealistico (o addirittura impossibile, in certi casi). In quel momento (e forse ti sarà già capitato) ti accorgi del tuo errore: è arrivato il momento di approfondire quell'argomento che prende il nome di "testing".

Certo, molto probabilmente hai letto un sacco di libri sul tema in passato. Non lo metto in dubbio. Tuttavia, come molte cose nella vita (e in questo lavoro) serve pratica. Pratica nella vita reale, per essere più precisi. Serve quell'istante in cui viviamo il momento che io chiamo "aha": il punto esatto in cui capisci veramente quella cosa e la fai tua.

C'è un problema però: a volte il testing può essere davvero insidioso. In alcune situazioni, infatti, la stessa codebase può essere strutturata in modo tale da essere totalmente impossibile da testare! Qui arriviamo ad un secondo punto fondamentale: il testing non migliora solo il risultato del tuo codice. Non si limita ad un semplice "funziona/non funziona". Il Test Driven Development (TDD) ti migliora come programmatore, ti offre una tecnica non indifferente che cambia il tuo modo di scrivere il codice. Niente più spaghetti code, niente più caos.

Quando cominci sempre più spesso a chiederti "come posso testare questa cosa?" sei sulla strada giusta. Te lo assicuro. Torni a guardare indietro, a quello che eri prima e ti senti soddisfatto: sei migliorato davvero.

Benvenuto nel mondo dello sviluppo di software moderno.

Se i principi del TDD sono indipendenti dal linguaggio che usi, quando si tratta di esecuzione il discorso cambia, così come cambiano i tool e le tecniche a tua disposizione. Questo libro è quindi un'introduzione al TDD, vero, ma analizza anche in maniera profonda e completa la strada seguita da Laravel in questo senso.

È Cominciata!

Quando si parla di linguaggi di programmazione ognuno deve dire la sua. Per non parlare di quando il discorso verte su PHP. Preparati alla guerra! Nonostante negli ultimi anni PHP sia maturato considerevolmente (soprattutto negli ultimi cinque), ci sono ancora molti sviluppatori che non riescono a guardare PHP in un modo diverso da quello di un padre che non riesce a vedere un figlio che ormai è cresciuto.

Tutti questi "padri" non vedono la nuova versione, 5.5, non vedono l'OOP, non vedono i framework moderni come Laravel, non vedono Composer. Per non parlare della crescente enfasi intorno allo sviluppo test-first. Tutto quello che riescono a vedere è il codice classico di PHP4, magari fatto male o, ancora peggio, l'unica cosa che vedono è un template Wordpress fatto nel 2008.

Qualcuno avrà da ridire. Certo, PHP non è un linguaggio bello come Ruby. Le API sono abbastanza inconsistenti, a volte. Senza dubbio. A quel punto c'è da chiedersi "perchè?". Insomma, perché PHP domina il mercato (con l'80% di share) con tutti gli altri linguaggi concorrenti, ritenuti "migliori"? Forse c'è altro, sotto, che vale la pena di esaminare. La verità è che la flessibilità di PHP non è una pecca, ma una virtù.

Insomma: PHP non è molto sexy, non è in beta, ma la verità è che ci facciamo un sacco di cose e il suo dovere lo fa bene. Puoi dire quello che vuoi su PHP 4, ma c'è un sacco di gente che sta facendo un bel po' di cose con PHP 5 e con tutto quello che l'ecosistema, ad oggi, ha da offrire.

Basta con questo odio nei confronti di PHP! La rinascita di questo linguaggio di programmazione è, finalmente, cominciata!

Usiamo delle tecniche moderne di programmazione ad oggetti, condividiamo i nostri package con Composer, usiamo il version control e facciamo di tutto per promuovere gli ultimi ritrovati in termini di framework. Crediamo nella potenza del testing (e presto sono sicuro che convincerò anche te).

Tra l'altro, vuoi sapere quale sarà la parte migliore di tutto questo? Semplice: come utilizzatore di Laravel sarai praticamente alla frontiera di questo nuovo cambiamento. Ricordo quando mi avvicinai, per la prima volta, al canale IRC di Laravel. Nel giro di pochi minuti c'era già qualcuno che mi diceva "Benvenuto in famiglia!". Ecco, credo non ci sia modo migliore di descrivere la situazione. Siamo tutti insieme ed è questo il tipo di community che amo.

Dato che hai comprato questo libro mi sembra di capire che potrebbe esserci la necessità di un aiuto nel settore testing.

Dunque, seguendo lo spirito di Laravel, te lo dico: benvenuto in famiglia. Vediamo come risolvere insieme i tuoi problemi!

Questo Libro è Davvero Adatto a Me?

Scrivere un testo tecnico ti mette davanti ad alcune difficoltà. A volte è necessario capire il punto esatto in cui tracciare la linea, in termini di pre-requisiti del lettore necessari ad una corretta comprensione. Chiariamo subito questo punto.

Hai delle conoscenze (anche basilari) di questi tre argomenti?

- PHP 5.3
- Laravel 3 (preferably version 4)
- Composer

Si? Bene! Allora direi che puoi continuare.

Perché Proprio Laravel?

Non spaventarti: le tecniche che apprenderai qui potrai adattarle ovunque, a qualsiasi linguaggio o framework. Tuttavia, per quella che è la mia esperienza in questi anni, quando c'è da muovere dei primi passi è sempre meglio farlo con scarpe comode. Vuoi sapere se puoi imparare le stesse cose per Java? Certo che puoi!

D'altro canto, comunque, lasciare il libro troppo generico non mi avrebbe dato la possibilità di farti conoscere tutti gli strumenti e features specifiche di PHP che uso nel mio lavoro di tutti i giorni. A partire dagli helper package di PHPUnit fino ad arrivare ad interi framework dedicati come Codeception. Senza considerare che, a volte, vedremo questo o quel trucchetto (o aspetto) di Laravel.

Esercizi

Di tanto in tanto, in questo libro scriverò dei capitoli "Esercizi". Saranno un buon lavoro di integrazione per le tue conoscenze appena acquisite. In questo modo non basterà semplicemente leggere il capitolo: la teoria è importante, ma solo se la integriamo con una buona dose di pratica.

Quindi, nel momento in cui leggi "Esercizi" prendi il PC e seguimi passo dopo passo!

Errata

Come puoi ben immaginare ho corretto più volte il testo e ho ripetutamente controllato ogni cosa. Non posso assicurarti, però, che non ci sia qualche errore li fuori, da qualche parte. Dopotutto sono umano anche io. Se dovessi trovarne uno fammelo sapere, aprendo un issue su GitHub[1] e correggerò il libro non appena mi sarà possibile. Come ricompensa, un bell'abbraccio per ognuno dei segnalatori!

Come Usare Questo Libro

Nessuno ti vieta di leggere il libro da una parte all'altra, così come nessuno ti vieta di leggerti i capitoli di cui hai bisogno. Ogni capitolo infatti è il più possibile a se stante. Per farti un esempio, se conosci le basi dell'unit testing allora non avrai nessun bisogno di leggerti il capitolo "Unit Testing 101". Saltalo a piè pari e passa avanti, verso quello che ti interessa di più.

Contatti

Senza dubbio da questo momento passeremo un po' di tempo insieme: per questo motivo potresti avere la necessità di contattarmi, in qualche modo. In tal caso riporto qui il mio account Twitter e il canale IRC di Laravel (con il mio nick).

[1] https://github.com/JeffreyWay/Laravel-Testing-Decoded

- **IRC (#laravel channel)**: JeffreyWay
- **Twitter**: @jeffrey_way[2]

[2] http://twitter.com/jeffrey_way

Nel Grande Ignoto

La notte era buia e tempestosa.

No, aspetta, è il libro sbagliato.

Ah si. La notte era silenziosa, salvo solo per lo strano, confortante girare della ventola del mio ventilatore da soffitto. Mia moglie e gli animali mi avevano già abbandonato da un po' e dormivano beatamente. Il cane, come sempre, aveva resistito più di tutti. Non potevo di certo biasimarlo: non è poi così comune rimanere svegli fino alle tre di notte e oltre. Ad ogni modo, l'ambiente era perfetto. Potevo sentirlo.

Gli sviluppatori conoscono bene questa sensazione: il momento in cui, inspiegabilmente, quasi magicamente, tutto prende forma nella mente in un modo perfettamente armonioso. I cosiddetti momenti "aha!". Il mio primo momento "aha" è stato quando ho capito bene a cosa serviva il tag " ". Lo so, adesso sempre stupido, ma quando sei all'inizio è tutto diverso. Perchè mai avrei dovuto mettere tutto quel codice all'interno del div? Che senso aveva? L'output era lo stesso in ogni caso, anche senza! Poi mi è stato detto di vederli come dei "blocchi" dove mettere il codice. Sarebbe bastato spostare quei blocchi per spostare tutti i loro contenuti di conseguenza.

Ne potrei trovare una marea di questi momenti… quando ho scoperto la programmazione orientata agli oggetti, il lavoro con le interfacce o magari il test-driven development. Devo ammettere, però, che l'amore per lo sviluppo guidato dai test non è stato così immediato. Come molti altri sviluppatori leggevo qualche articolo, o magari un libro sull'argomento, e concludevo tutto con un "wow, interessante", per poi rimanere con le mie idee e le mie pratiche.

A prescindere dal fatto che lo sapessi o meno, avevo piantato dei semi. Il tempo ha fatto il suo corso, gradualemnte, e piano piano una nuova idea è cresciuta nella mia testa, sempre più forte e dirompente.

> "*Dovresti fare dei test, Jeffrey.*" "*Se avessi scritto dei test per questa classe, non staresti li a provare tutto manualmente di volta in volta.*" "*Ti rideranno in faccia se per quella pull request non presenterai dei test a corredo delle tue modifiche.*"

Come molte cose nella vita, per vivere davvero un cambiamento radicale c'è bisogno di piantare i piedi per terra ed urlare "*Non stavolta! Ho finito di fare come sempre!*". Così l'ho fatto. Migliaia di sviluppatori l'hanno già fatto.

Adesso è il tuo turno.

Come direbbe Leeroy Jenkins[3], facciamolo! Questo è il libro di cui avevi bisogno.

[3] http://www.youtube.com/watch?v=LkCNJRfSZBU

Capitolo 1: Testa Tutto

Ogni libro sul testing ha bisogno di un capitolo "*Perché i Test*". Se ci pensi un po', il semplice fatto che tu abbia comprato questo libro implica che il concetto, almeno un po', ti è già entrato in testa. Ad essere precisi, il concetto di necessità dei test. Ad ogni modo, è ottimo capire bene perché c'è bisogno del testing.

Imparare a testare adeguatamente le proprie applicazioni presenta, sfortunatamente, una curva di apprendimento decisamente più ostica rispetto ad altre pratiche della programmazione. Perlomeno, per me è stato così.

Ad ogni modo il principio di fondo, di base, è davvero semplice: scrivi dei test per provare che il tuo codice funziona esattamente come ci si aspetta.

> Ti ritrovi ad usare il tuo Chrome continuamente, refreshando di continuo la pagina per quella specifica funzionalità? Dimentica una cosa del genere. Scrivi un test e via.

Insomma, tutto molto semplice, vero? Bene. Allora cos'è che rende il tutto così difficile?

Semplice: la parte complicata comincia quando devi capire "cosa" testare.

> Cosa devo testare? I controller? I model? Le view hanno bisogno di test? Riguardo al codice del framework o del database, invece, come mi comporto? Ho letto una marea di opinioni diverse su StackOverflow ma non so come regolarmi! Tra l'altro, quale framework devo usare? PHPUnit? Rspec? Capybara? Codeception? Mink? Da dove comincio?

Alt, alt, alt! Non sono un tipo da scommesse (*beh, ad eccezione di quando gioco a Scrabble con mia moglie, dove mi diverto ad umiliarla anche in pubblico*[4]). Sono sicuro, però, che prima o poi nella tua carriera ti porrai una o più domande del genere. Per cui procediamo con ordine.

Riassumiamo i primi concetti fondamentali.

Testare è semplice.

Capire cosa e come testare è ben altro discorso.

Tranquillo però: questo libro è stato scritto di proposito.

[4] http://notes.envato.com/team/jeffrey-wins-a-bet/

Si, tu fai già dei test.

La verità è che tu stai già testando. Sei già un maestro del testing. Magari hai usato, qualche volta, un file "console.log" oppure hai inviato un form in una specifica web app. In quei casi stavi testando un determinato componente o una certa funzionalità. Fin da piccoli siamo abituati a testare, se ci pensi: "se giro la maniglia, la porta si apre… WOAH!"

L'unico problema è che questi test, se ci pensi, vengono fatti manualmente. A quel punto una domanda sorge spontanea: "non è che posso automatizzare tutto questo lavoro?". Il nostro obiettivo, in questo libro, è proprio questo: automatizzare l'intero processo. Il che porta a vantaggi immensi: man mano che vai avanti, che scrivi il tuo codice, hai i tuoi test pronti per evitare che qualcosa vada storto. Una volta presa la giusta confidenza sarai stupito dal livello di sicurezza che il testing porta al tuo codice.

> "Quando uno sviluppatore scopre le meraviglie del test-driven development, è come entrare in nuovo mondo migliore, dove c'è meno insicurezza e meno stress."

6 Pro del TDD (Test Driven Development)

Il testing è qualcosa di ingannevole. Inizialmente, infatti, potresti ritrovarti a pensare che l'unico beneficio è quello che di assicurarti di avere un codice che funziona esattamente come ti aspetti. In realtà, però, non è così: ci sono svariati vantaggi aggiuntivi. Vediamoli insieme.

1. Sicurezza

Se hai, accidentalmente, fatto un errore o creato un blocco in una qualche funzionalità già esistente, un test ti notificherà che c'è qualcosa che non va. Immagina di modificare qualcosa, salvare e ricevere praticamente al volo una notifica che ti dice "ehi, stai attento perché hai appena fatto un casino!". Dormiresti decisamente meglio la notte, ammettilo. Già vedo tornarti in mente quel codice che non volevi sistemare e che durante la presentazione… no ok non tiriamo fuori quell'episodio! In ogni caso, troppa paura di fare un refactor!

Con i test queste paure non esistono più.

2. Contributi

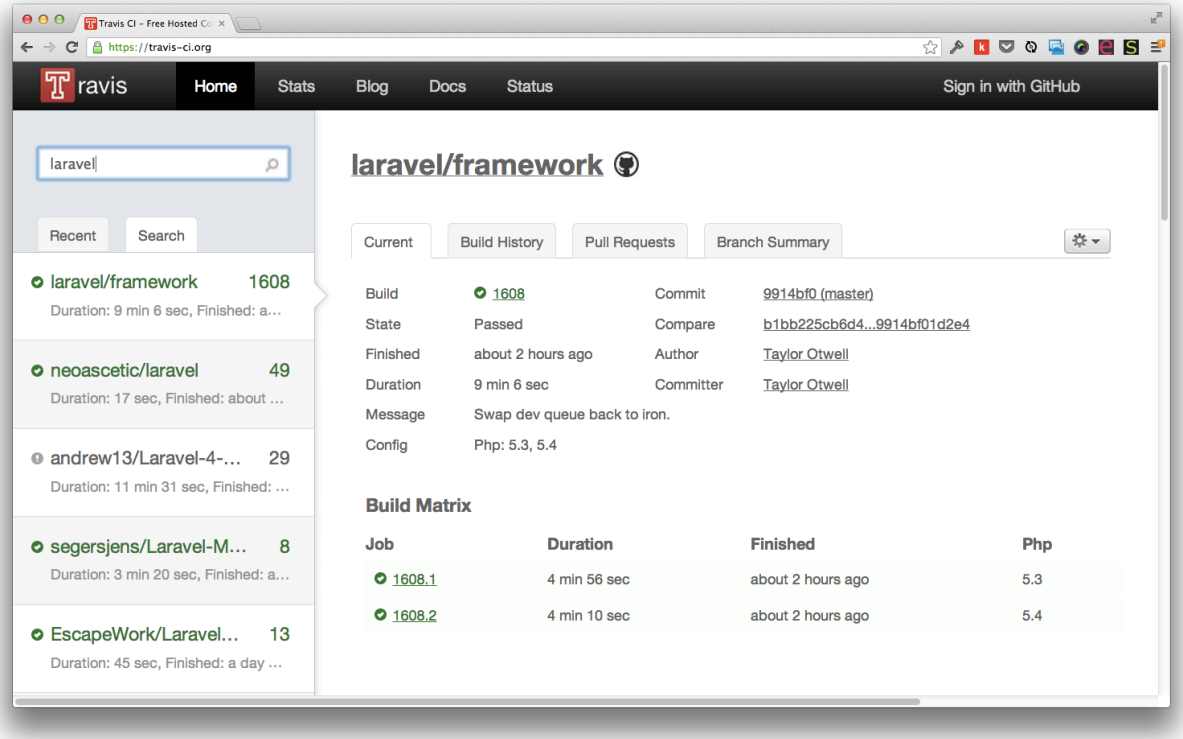
Se cominci a sviluppare software open-source, probabilmente imparerai ad apprezzare il social coding tramite GitHub. Probabilmente altri membri del tuo team contribuiranno al tuo software, scovando i bug oppure implementando nuove funzionalità. Tuttavia, se il tuo progetto non contiene una test suite, la distanza tra il tuo progetto e una bolgia infernale si accorcia in un modo che neanche puoi concepire.

Prova ad immaginare: come potrebbero i tuoi collaboratori (e tu) capire dove hanno sbagliato per ogni pull? Non potrebbero, questa è la verità. Non senza testare manualmente di nuovo tutto il codice interessato. Diciamocelo francamente: chi ha davvero il tempo di fare una cosa del genere? No, non tu. Neanche io.

Ecco invece cosa succede se una test-suite c'è:

1. Clono il repository
2. Scrivo un test che descrive il bug
3. Faccio tutte le modifiche necessarie affinché funzioni
4. Avvio il test, per assicurarmi che le modifiche funzionino al 100%
5. Faccio la commit e relativa submit della mia pull request

Ci sono un sacco di servizi, in giro, per l'integrazione continua. C'è, ad esempio, Travis[5], che lancia i vari test per un determinato progetto ad ogni submit. Se i test falliscono viene notificato immediatamente cosa non funziona e cosa non dovrebbe essere "incluso" nel progetto principale. Ne parleremo, in ogni caso, più avanti.



travis-ci.org

[5] https://travis-ci.org/

3. Mettersi nei Panni del "Ragazzo Più Grande"

Dovendo fare delle osservazioni trascendenti, anche solo per un momento, c'è da dire che quando si parla di community PHP e best practice Wordpress è un'arma a doppio taglio. Ha tanti meriti, indubbiamente: ha portato il blogging alle masse e ha creato dei sistemi di templating davvero facili da usare. Basta creare un file `index.php`, metterci un loop e poi dargli uno stile. Poche cose sono più semplici.

Nulla da eccepire. Tuttavia, c'è da dire che questa pratica ha inavvertitamente allevato una "sotto-community" di sviluppatori che hanno cominciato ad evitare altre soluzioni esterne a Wordpress per i loro progetti. Di conseguenza, molti pattern moderni come il TDD, l'MVC e il version control hanno cominciato a latitare sempre di più.

Una cosa da non sottovalutare, che ha portato a due effetti collaterali non indifferenti:

1. Una buona parte della community PHP, fino a pochissimi anni fa, era identificabile nel binomio "PHP 4 e Wordpress".
2. Fare il "salto" da Wordpress fino ad un framework full-stack, come Laravel, si è rivelato incredibilmente difficile e complesso. I tool da imparare ad usare sono infatti tanti e la curva di apprendimento è diventata più difficile da sopportare, almeno all'inizio.

Insomma, Wordpress ha fatto davvero tutti questi casini? Beh, si e no. Una cosa è certa, però: certamente non ha portato lo sviluppo PHP al suo apice. Di fatto, un buon 90-95% dei plugin di Wordpress non ha test a corredo.

Così arriviamo al punto: arriva il momento di diventare grandi e capire che ci sono cose importanti da conoscere. Lo "scrivere codice senza pensarci", o il "la butto li e poi vediamo" non deve più appartenerti.

> **Siamo meglio di questo. Siamo sviluppatori. Non siamo cowboy.**

Pensaci bene: quando ti imponi di pensare prima di scrivere il tuo codice si verifica un impressionante miglioramento nella qualità dello stesso codice. Chi ci avrebbe mai scommesso? Scrivere dei test allora non è solo verificare che il codice funziona. Quando facciamo dei test interagiamo con il nostro codice prima ancora di scriverlo. Prevediamo ogni sua mossa, ogni suo imprevisto. Questo ci rende più forti e rimuove tutti i nostri limiti, permettendoci di concentrarci di più sul resto.

Un aspetto su tutti? La leggibilità.

Stai scrivendo un servizio web e ti preoccupi, in anticipo, di come verrà letto da fuori. Complimenti: sei già un passo avanti rispetto alla normalità.

Una cosa meravigliosa, ma non basta: passiamo quindi al punto successivo come logica conseguenza.

4. La testabilità migliora l'architettura.

Una cosa che imparerai attraverso questo libro sarà il saperti porre la fatidica domanda: "come potrei testare questa cosa?".

Succederà ogni volta scriverai qualcosa di nuovo. Questa semplice, importantissima domanda sarà la tua sicurezza, e ti terrà al riparo dall'imperfezione del tuo passato da cowboy. Non vedrai mai più quei lunghi metodi noiosi delegati a fare un milione di cose.

L'isolamento delle responsabilità equivale ad un codice migliore. Design prima del codice.

5. Documentazione

Un altro punto molto, molto importante nella scrittura dei test è che permettono di fornire una documentazione a costo zero per l'intero sistema. Vuoi sapere quali funzionalità offre una determinata classe? Bene, gioca un po' con i test e, se ben denominati, avrai le idee molto più chiare nel giro di pochi secondi!

6. È Divertente

Guardiamo in faccia la realtà: siamo tutti geek ed un geek adora giocare ad un buon gioco. In un certo senso, il TDD trasforma quello che è il tuo lavoro in un gioco. Quel benedetto test deve uscire fuori verde. Deve andare tutto bene. Devi vincere. Si, può sembrare idiota all'inizio, ma poi ci prendi la mano. Diventa divertente. Lo vedrai da solo a breve.

Cosa Dovrei Testare?

La regola base - quella che ti ripeterò in continuazione per tutto il libro - sarà quella di testare ogni singola cosa può causare, potenzialmente, un break all'interno del tuo sistema. Una determinata funzionalità può reindirizzarti verso una pagina 404? Bene, va scritto un test a riguardo. Cosa fare, invece, con quella classe che ti crea un report su file dopo una lettura su database? Dici che è il caso di testarla? Certo, anche li vai giù di test. Cosa dire invece riguardo a quell'utility class che esegue un fetch degli ultimi tweet che hai scritto? Se l'unica alternativa è aprire Chrome e controllare da te la sidebar… sai già cosa sto per dirti.

Test, test, test.

Lo so cosa stai pensando: all'inizio il mantra *testa tutto* può essere noioso e eccessivo. Con una curva di apprendimento del genere ti capisco: va benissimo se decidi di fare un passo alla volta. Comincia con il testare i tuoi models. Poi, una volta presa la giusta dimestichezza, procedi con qualcosa di più. Piccoli passi, ma ben fatti: questo è l'importante. Deve essere un metodo che, piano piano, diventa il tuo metodo.

Una Nota Importante

Ricordati bene queste parole. Testare qualasiasi cosa può creare un break è un obiettivo davvero nobile, ma c'è una cosetta chiamata over-testing (sul suo significato si dibatte ampiamente, quindi non starò qui a dilungarmi troppo). Una parte della community, infatti, sostiene che è meglio limitare i test per quelle parti "critiche" dove il beneficio è maggiore. In effetti, se è raro che tu faccia errori in metodi semplici come un accessor o un mutator, allora non disturbarti nel creare test su altri test.

I test servono a te, innanzitutto. Il tuo compito come sviluppatore sarà quello di capire dove tracciare la linea, sul cosidetto punto di diminishing returns[6] o di diminuzione di guadagno, a dir si voglia.

Sei Segni di un Codice Non Testabile

Imparare come testare il tuo codice è un po' come muoversi in un paese dove nessuno parla la tua lingua. Il principio è lo stesso: più insisti nel cercare di capire come regolarti, e più cominciano ad entrarti in testa i pattern giusti. Fino al momento in cui cominci a parlare fluentemente. Non stiamo parlando di scienza per pochi: chiunque può imparare. Tutto quello di cui hai bisogno è volontà… e tempo. (*Leggi quest'ultima frase con il tono di Morgan Freeman*)

Più migliori nel testing e più velocemente cominci a riconoscere cosa non va nel tuo codice. Qualcosa di istintivo: ti ritrovi a scandire ogni singola parte del tuo codice, facendo caso anche al minimo anti-pattern.

Ecco un po' di cose da tenere a mente.

1. Operatore "new"

I principi dello unit testing esigono che il test avvenga in condizioni di "isolamento". Copriremo meglio questo concetto nei capitoli successivi ma, per il momento, il tuo compito dovrebbe essere quello di testare la classe "attuale" e null'altro. Niente accesso al database e niente che vada a coinvolgere quella classe "Filesystem".

Ecco il primo consiglio: quando cominci ad usare l'operatore "new" non rispetti la regola dell'isolamento. Testare in isolamento richiede che la classe non istanzi nessun oggetto.

Anti-pattern:

[6] http://en.wikipedia.org/wiki/Diminishing_returns

```
{: lang="php" }
public function fetch($url)
{
    // Non puoi testare una cosa del genere!
    $file = new Filesystem;

    return $this->data = $file->get($url);
}
```

Questa è una di quelle situazioni in cui PHP non è così flessibile come speriamo. Se alcuni linguaggi, infatti, ci permettono di ri-aprire una classe (qualcuno ha parlato del monkey-patching di Ruby?) e sovrascrivere dei metodi (*prassi particolarmente utile per il testing*), PHP sfortunatamente non lo permette. Se non attraverso la modifica di PHP stesso, tramite alcune estensioni.

Per questo motivo dobbiamo fare un largo uso dell'iniezione di dipendenze (Dependency Injection).

Così:

```
{: lang="php" }
protected $file;

public function __construct(Filesystem $file)
{
    $this->file = $file;
}

public function fetch($url)
{
    return $this->data = $this->file->get($url);
}
```

Con questa modifica, infatti, una versione "mock" della classe Filesystem potrà essere usata senza problemi, permettendo una testabilità di gran lunga superiore rispetto a prima. Guarda l'esempio di seguito e non ti preoccupare se non capisci alcune cose: le vedremo nel dettaglio dopo.

```
{: lang="php" }
public function testFetchesData()
{
    $file = Mockery::mock('Filesystem');
    $file->shouldReceive('get')->once()->andReturn('foo');

    $someClass = new SomeClass($file);
    $data = $someClass->fetch('http://example.com');

    $this->assertEquals('foo', $data);
}
```

Questo è l'unico esempio in cui va bene istanziare una classe all'interno di un'altra classe: stavolta infatti abbiamo a che fare con un cosiddetto value-object, ovvero un oggetto delegato solo al mantenimento di alcuni valori che non effettua un vero e proprio lavoro.

> **Suggerimento**: vai sempre a caccia di "new" nel tuo codice. Non perdertene manco uno.

2. Costruttori "Sbagliati"

L'unica responsabilità di un costruttore dovrebbe essere quella di assegnare le eventuali dipendenze. Esatto, un po' come se la tua classe "chiedesse" ciò di cui ha bisogno. Nel momento in cui la tua classe fa qualsiasi altra cosa che non sia questa appena detta, considera un refactoring.

Anti-pattern:

```
{: lang="php" }
public function __construct(Filesystem $file, Cache $cache)
{
    $this->file = $file;
    $this->cache = $cache;

    $data = $this->file->get('http://example.com');
    $this->write($data);
}
```

Così va bene:

```
{: lang="php" }
public function __construct(Filesystem $file, Cache $cache)
{
    $this->file = $file;
    $this->cache = $cache;
}
```

La ragione è semplice: quando effettui dei test segui sempre la stessa procedura (considerata poi un pattern vero e proprio, in un certo senso):

1. Arrange (sistema ciò di cui hai bisogno)
2. Act (agisci, facendo le operazioni che ti servono)
3. Assert (verifica il risultato)

Ogni test dovrebbe lavorare su questi tre punti fondamentali.

> **Suggerimento**: mantieniti sempre sul semplice, limitando i tuoi costruttori ad un lavoro di assegnazione delle dipendenze.

3. E, e, e...

Capire bene quali sono le responsabilità di una singola classe può essere qualcosa di davvero difficile, all'inizio. Certamente conosciamo a memoria il *Principio di Isolamento delle Responsabilità*, ma tra la teoria e la pratica, a volte, c'è un abisso...

Quattro Modi di Capire Quando una Classe Ha Troppe Responsabilità

1. Il modo più semplice di capire se una classe sta "facendo troppe cose" e raccontarti cosa fa la classe in questione. Se ti senti ripetere troppo spesso la congiunzione "e" allora c'è un'ottima possibilità che, effettivamente, la tua classe stia facendo troppe cose.
2. Allenati ad analizzare immediatamente il numero delle linee in ogni singolo metodo. Idealmente, infatti, un metodo dovrebbe avere poche linee di codice per fare quel che deve (a volte ne basta solo una). Se, d'altro canto, ogni metodo ha una marea di linee di codice... beh, stai sbagliando qualcosa.
3. Se hai qualche difficoltà a trovare un nome per la tua classe dovresti riflettere sul fatto che forse stai andando fuori tema. Anche qui, probabilmente, la tua classe fa troppe cose.
4. Se i tre primi passi non portano a risultati, prova a mostrare la tua classe ad un altro sviluppatore. Se il tuo amico non capisce subito quale dovrebbe essere lo scopo della classe allora hai bisogno di rivedere un po' le cose.

Qualche esempio di classe che fa le cose per bene:

- Una classe `FileLogger` è responsabile di sistemare in un file i dati di log.
- Una classe `TwitterStream` si occupa di ritornare gli ultimi tweet tramite le API di Twitter, partendo da uno Username.
- Una classe `Validator` si occupa di validare dei dati a fronte di alcune regole.
- Una classe `SQLBuilder` costruisce una query SQL.
- Una classe `UserAuthenticator` determina se le credenziali di un utente sono corrette oppure no.

Fai caso al fatto che MAI, in queste spiegazioni, appare la congiunzione "e".

> **Suggerimento**: riduci la responsabilità di ogni classe in modo tale che si occupi di fare una sola cosa. Si tratta del cosiddetto *Single Responsibility Principle*.

4. Troppi Percorsi? È Tempo di Polimorfismo!

Sicuramente, capire bene il polimorfismo richiede un cosiddetto momento "aha"[7]. Tuttavia, come molte cose nella vita, una volta capito non lo scordi più.

> **Definizione**: Per Polimorfismo si intende la divisione di una classe complessa in più sottoclassi, che condividono entrambe un'interfaccia di base ma che possiedono funzionalità uniche.

Il modo più semplice per capire se una classe ha bisogno di polimorfismo è andare a caccia di blocchi "switch" (o troppi condizionali ripetuti, in generale). Immagina una classe che abbia il compito di gestire un conto corrente. Supponiamo che questa classe si occupi, tra le varie cose, di calcolare l'interesse annuale. In maniera differente, però, dato che i tipi di conto possono essere diversi. Ecco un esempio molto semplificato:

```
{: lang="php" }
function addYearlyInterest($balance)
{
    switch ($this->accountType) {
        case 'checking':
            $rate = $this->getCheckingInterestRate();
            break;

        case 'savings':
            $rate = $this->getSavingsInterestRate();
            break;
```

[7] https://tutsplus.com/2012/04/the-aha-moment/

```
12
13          // other types of accounts here
14      }
15
16      return $balance + ($balance * $rate);
17  }
```

In situazioni come queste, la cosa più intelligente da fare è estrarre tutto ciò che è simile, quindi lavorare di conseguenza con svariate sottoclassi.

Definire un'interfaccia permette di essere sicuri che verrà sempre implementato un metodo `getRate`. Spesso avrai sentito parlare delle interfacce come "contratti". Sicuramente è un bel modo di intenderle: le interfacce definiscono, infatti, i termini di un contratto che devono essere rispettati. Per poter "lavorare", una classe dovrà rispondere a determinati requisiti.

Esattamente così:

```
1  {: lang="php" }
2  interface BankInterestInterface {
3      public function getRate();
4  }
```

Una volta creata l'interfaccia si procede con l'implementare le varie sottoclassi di conseguenza.

```
1  {: lang="php" }
2  class CheckingInterest implements BankInterestInterface {
3      public function getRate()
4      {
5          return .01;
6      }
7  }
```

Ed ecco l'altra.

```
1  {: lang="php" }
2  class SavingsInterest implements BankInterestInterface {
3      public function getRate()
4      {
5          return .03;
6      }
7  }
```

Adesso, il metodo "originale" può essere scritto in modo molto, molto più pulito. Nota come ho specificato il tipo per la variabile `$interest` nell'esempio successivo.

```
{: lang="php" }
function addYearlyInterest($balance, BankInterestInterface $interest)
{
    $rate = $interest->getRate();

    return $balance + ($balance * $rate);
}

$bank = new BankAccount;
$bank->addYearlyInterest(100, new CheckingInterest); // 101
$bank->addYearlyInterest(100, new SavingsInterest); // 103
```

In questo modo il codice sarà più sicuro: non verrà mai richiamato il metodo `getRate` se questo non è mai stato specificato. Il punto è esattamente questo: implementando l'interfaccia siamo forzati ad usare i metodi richiesti dall'interfaccia stessa.

Niente più percorsi e diramazioni pericolose nel nostro codice. Il testing sarà molto più facile, come logica conseguenza.

```
{: lang="php" }
public function testAddYearlyInterest()
{
    $interest = Mockery::mock('BankInterestInterface');
    $interest->shouldReceive('getRate')->once()->andReturn(.03);

    $bank = new BankAccount;
    $newBalance = $bank->addYearlyInterest(100, $interest);

    $this->assertEquals(103, $newBalance);
}
```

> **Nota**: il polimorfismo ti permette di suddividere le classi più complesse in "pezzi" più piccoli, spesso denominati "sotto-classi". Ricorda: più piccola è la classe, più facile risulterà il test.

5. Troppe Dipendenze

Come detto prima nel punto due, le dipendenze dovrebbero essere "iniettate" nel costruttore della classe. Detto questo, se una determinata classe ha bisogno di tante dipendenze c'è sicuramente qualcosa che non va. Come direbbe qualcuno che conosciamo bene:

> Generalmente non accetto una classe che ha più di quattro [dipendenze]." - *Taylor Otwell*[8]

[8] https://twitter.com/taylorotwell/status/334789979920285697

Un principio basilare della programmazione orientata agli oggetti specifica che c'è una certa correlazione tra il numero di parametri che un metodo costruttore di una classe può accettare ed il suo "grado di flessibilità" (e, di conseguenza, testabilità). Ogni volta che rimuovi una dipendenza da una classe, il tuo codice migliora.

Agisci di conseguenza.

6. Troppi Bug

Una volta ho sentito Ben Orenstein ricordare che gli "insetti (bugs) amano la compagnia". Si tratta di una delle frasi più vere che io abbia mai sentito. Se ci fai caso, nel momento in cui cominciano a presentarsi in una certa classe troppo frequentemente c'è qualcosa che non va. La necessità di fare refactoring e dividere in sottoclassi diventa più forte. Pensaci: un bug esiste perché il codice non è stato capito e scritto sufficientemente bene la prima volta. Forse era troppo complicato… e un codice complesso equivale ad un codice non testabile.

Tra l'altro, ricorda anche che quando il "coupling" è troppo forte viene spianata la strada per i bug.

> **Definizione**: Il "Coupling" è una "misura" del grado in cui due componenti in un sistema dipendono l'uno dall'altro. Se rimuovere o sostituire uno pregiudica l'effetto dell'altro, c'è qualcosa che non va nel codice che hai scritto.

> **Suggerimento**: quando si presentano bug del genere, chiediti se non è il caso di dividere la classe che stai scrivendo in più sotto-classi. Oltre alla maggiore testabilità avrai un codice decisamente più leggibile.

Il "Jargon" del Testing

Non ho la presunzione di credere che questo libro sarà la tua unica fonte di conoscenza riguardo il testing (anzi, spero proprio di no).

Tuttavia, durante il percorso che faremo insieme incontrerai dei termini più frequenti di altri, importanti per il contesto in cui opereremo. A volte questa terminologia può risultare confusionaria, o comunque può indurti in errore. Chiariamo insieme, ora, qualcuno di questi possibili dubbi dando delle definizioni per questi termini.

A volte penso che l'informatica dovrebbe prendere spunto dall'astrofisica, per quanto riguarda la nomenclatura. Come vai a chiamare quelle regioni di spazio dalle quali non esci? Buchi neri. Come chiami una stella grande e rossa? Gigante rossa. Semplice, no?

Comunque, procediamo.

Unit Testing

Pensa allo Unit Testing come il prendere un pettine e assicurarti che ogni pezzo del tuo codice funzioni esattamente come ti aspetti. Gli unit test dovrebbero essere eseguiti in "isolamento", per rendere il processo stesso di testing più semplice possibile. Normalmente, l'80% dei tuoi test sarà di questo tipo.

Quando pensi allo Unit Testing devi pensare ad "un oggetto, un solo oggetto". Perché se il test fallisce sai cosa cercare.

Model Testing

Alcuni membri della community di Ruby on Rails associano il Model Testing (anche quando questi test vanno a toccare il database) con lo Unit Testing. Sfortunatamente, questa associazione può indurre in errore. Gli Unit test, infatti, dovrebbero essere isolati rispetto alle dipendenze esterne. Una volta che ignori questa regola non stai facendo più Unit Testing. Stai scrivendo gli Integration Test (di cui parleremo a breve).

In questo libro, quando testeremo i nostri models faremo sempre del sano Unit testing (a meno che non sia specificato altrimenti). Ricordalo.

Integration Testing

Se lo Unit testing deve verificare determinate condizioni in isolamento, un integration test dovrà occuparsi dell'altro lato della medaglia. Normalmente arrivati a questo punto non avremo più a che fare con mock e stub. Per sicurezza, molto probabilmente, ti creerai un database di test dedicato.

Per capirci meglio, immagina una macchina. Sicuramente, il motore e il sistema di iniezione di carburante lavoreranno come previsto (dopo gli opportuni unit test), ma come puoi sapere se lavoreranno bene insieme? Ecco a cosa serve un Integration test.

Functional Testing

Come puoi definire il processo di testing di uno specifico controller? Molti framework si riferiscono a questa pratica come il "Functional Testing". Per quanto riguarda il nostro percorso, ne parleremo semplicemente come *controller testing*.

Sempre seguendo la linea, pensa al functional testing come un modo per te e il tuo team di essere sicuri che il codice funziona esattamente come previsto. Se gli unit test hanno il compito di verificare tutto quello che riguarda una specifica classe, il functional testing si occupa di testare, insieme, svariate parti della tua applicazione.

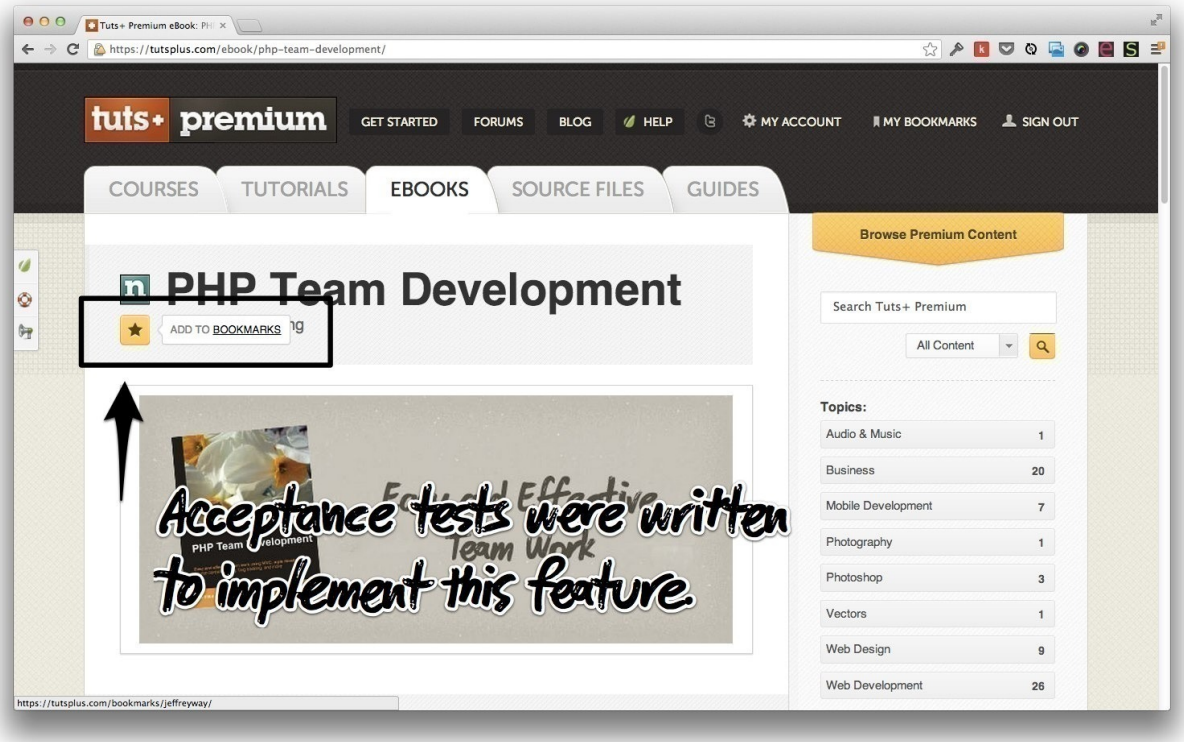
Acceptance Testing

Hai appena letto di cosa si occupa il functional testing. Tuttavia, ci sono casi in cui anche se i test vanno bene, la feature implementata non rispecchia la richiesta del cliente. Questo è quello che viene denominato "acceptance testing": cosa può essere "accettato", appunto, e cosa no. Un software può passare tutti i test, dagli unit fino ai functional... ma se non passa gli acceptance test c'è poco da fare.

> **Suggerimento**: così come il functional testing verifica quello che lo sviluppatore si aspetta, l'acceptance testing verifica quello che il cliente si aspetta.

Pensa a Tuts+ Premium[9], il servizio di training basato su iscrizione a cui lavoro. Recentemente abbiamo aggiunto una nuova feature: la possibilità di salvare determinati corsi ed eBook da leggere in un secondo momento.

Questa feature ha richiesto un certo acceptance test, prima dell'implementazione.



La feature di Bookmarking su Tutplus ha richiesto degli acceptance test.

Nella parte finale di questo libro vedremo insieme come scrivere degli acceptance test tramite il framework Codeception.

[9] http://tutsplus.com

> Ok, va bene… ma non sono un po' troppo "dispendiosi" questi test?

Purtroppo questo è un errore che si fa spesso. Si pensa che, nonostante gli evidenti benefici del testing, quando si arriva a lavorare con un cliente dotato di budget tutto cambia.

Come se non bastasse, tra l'altro, è un preconcetto del tutto sbagliato. Svariate prove che abbiamo svolto nell'ultimo periodo hanno registrato un ciclo di sviluppo notevolmente ridotto per un progetto che beneficia dello unit testing.

Relax

Sono il primo a dirti che queste definizioni necessitano di un bel po' di tempo per essere metabolizzate ed apprese nel modo giusto. Non mi aspetto di certo che, finita questa parte, il discorso si chiuda qui. Tieni in mente quello che riesci, oppure torna quando avrai bisogno di questa o quella definizione.

Detto questo, secondo me è una vera sfortuna che la comunità degli sviluppatori, nella sua interezza, non sia ancora riuscita a definire una terminologia univoca e standard per facilitarsi la vita. Spesso incapperai in svariati termini come system testing, specifiche della richiesta, medium test e così via. In molti casi c'è davvero poca differenza con quanto visto nei paragrafi precedenti. La cosa più importante, comunque, è cominciare: svilupperai un tuo stile nel tempo.

Insomma, per dirla in breve:

> Non importa come fai i tuoi test… dipende per quanto lo fai.

La dissonanza continua, comunque, ben oltre la terminologia. Nonostante molti sviluppatori dicono quanto è importante scrivere dei test, non è chiaro il come quanti di loro scrivano questi test. Alcuni evangelist, come Bob Martin, raccomandano la cosiddetta prassi "strict": non si scrive una sola linea di codice senza prima aver scritto il test corrispondente.

Molti altri sviluppatori, invece, come DHH (il creatore di Ruby on Rails), ammettono senza problemi di scrivere il codice dei test dopo il codice di produzione. Normalmente per l'80% dei casi.

Il tuo lavoro sarà anche questo: trovare la soluzione ideale per le tue abitudini. Anche per questo motivo rifletti su una cosa importante: quando leggi questo libro non prenderlo come una Bibbia, ma come un'interpretazione di qualcosa.

Un'interpretazione mia, personale, che può cambiare.