Modernizing Legacy Applications in PHP

Paul M. Jones

Modernizzare Applicazioni Legacy in PHP

Paul M. Jones and Damiano Venturin

This book is for sale at http://leanpub.com/modernizinglegacyapplicationsinphpitalian

This version was published on 2015-06-04

Indice

1. Applicazioni legacy

Nella sua definizione più semplice, una "applicazione legacy" è una qualunque applicazione che è stata ereditata da un altro sviluppatore. È stata scritta prima del tuo arrivo ed avevi poco o nessun potere decisionale sul come costruirla.

Tuttavia, la parola "legacy" ha un significato più ampio tra gli sviluppatori. Ha connotazioni negative come "male organizzato", "difficile da mantenere e migliorare", "difficile da capire", "non testata" ed altro. L'applicazione in sè funziona, nel senso che produce qualcosa, ma è un programma fragile e molto sensibile ai cambiamenti.

Poichè questo libro è specifico per applicazioni legacy scritte in PHP, voglio descrivere alcune caratteristiche specifiche e comuni che ho visto lavorando in questo campo. Direi che un'applicazione PHP è legacy quando due o più delle seguenti condizioni sono verificate:

- Gli script sono posizionati direttamente nella document root del web server
- Ha speciali index file collocati in certe directory per impedirne l'accesso
- All'inizio di alcuni file sono presenti porzioni logiche come `die ()` o `exit ()` che vengono eseguite quando non è impostato un certo valore
- La sua architettura è include-oriented invece di class-oriented (o objetc-oriented)
- Ha relativamente poche classi
- La struttura delle classi è disorganizzata, sconnessa o incoerente
- Si basa più sulle funzioni che sui metodi di classe.
- Gli script, le classi e le funzioni mescolano [model, view e controller] (https://en.wikipedia.org/wiki/Model-view-controller) nello stesso ambito.
- Mostra segni di uno o più tentativi incompleti di riscrittura, a volte nel fallace tentativo di integrare un framework
- Non implementa alcuna suite per i test automatici

Queste situazioni sono probabilmente familiari a chiunque abbia avuto a che fare con una vecchia applicazione PHP. Rappresentano quello che io chiamo una "tipica applicazione PHP".

La tipica applicazione PHP

La maggior parte degli sviluppatori PHP non ha avuto una formazione accademica appropriata e alcuni sono quasi completamente autodidatta. Normalmente approdano al linguaggio arrivando da altre professioni, spesso non tecniche. In un modo o nell'altro, sono incaricati di creare pagine web perché sono visti come la persona più tecnicamente preparata del gruppo. Dal momento che PHP è un linguaggio che perdona molto e che concede tanta potenza senza tanta disciplina, è molto facile produrre pagine web funzionanti - e perfino applicazioni - senza avere necessariamente molta preparazione.

Questi ed altri fattori influenzano fortemente le fondamenta di un'applicazione PHP. Molte applicazioni non sono scritte usando popolari micro-framework o framework full-stack. Spesso, invece, sono una serie di script collocati direttamente nella document root del web server che i client possono addirittura navigare direttamente. Le funzionalità destinate al riutilizzo vengono raccolte in una serie di `include` file. Ci sono

`include` file per la configurazione e le impostazioni, l' intestazione ed il piè di pagina, form e contenuti condivisi, definizioni di funzione, navigazione e così via.

L'utilizzo di `include` file è quello che io chiamo architettura "include-oriented". L'applicazione legacy utilizza chiamate `include` ovunque per assemblare i pezzi del programma in un tutt'uno. Questo approccio è in contrasto con l'architettura "class-oriented" dove, anche se l'applicazione non aderisce ai buoni principi di programmazione object-oriented, almeno le parti sono raggruppate in classi.

La struttura dei file

Una tipica applicazione PHP include-oriented ha un aspetto simile a questa:

/path/to/docroot/

```
bin/                    # command-line tools
cache/                  # cache files
common/                 # commonly-used include files
    classes/            # custom classes
        Image.php       #
        Template.php    #
    functions/          # custom functions
        db.php          #
        log.php         #
        cache.php       #
    setup.php           # configuration and setup
css/                    # stylesheets
img/                    # images
index.php               # home page script
js/                     # JavaScript
lib/                    # third-party libraries
log/                    # log files
page1.php               # other page scripts
page2.php               #
page3.php               #
sql/                    # schema migrations
sub/                    # sub-page scripts
    index.php           #
    subpage1.php        #
    subpage2.php        #
theme/                  # site theme files
    header.php          # a header template
    footer.php          # a footer template
    nav.php             # a navigation template
```

Questa struttura è un esempio semplificato ed esistono molte possibili varianti. In alcune applicazioni legacy ho visto letteralmente centinaia di script di primo livello e decine di sottodirectory ciascuna con la propria specifica gerarchia di pagine. Il fattore chiave è che l'applicazione legacy sta di solito nella radice del documento, ha script che gli utenti navigano direttamente ed utilizza gli `include` file per gestire il comportamento del programma piuttosto che utilizzare classi ed oggetti.

Page scripts

Le applicazioni legacy usano "page scripts" *(chiarimento: l'autore intende per page script una pagina php indipendente che svolge un set di funzioni dell'applicazione)* individuali come punto di accesso pubblico. Ogni page script è responsabile della creazione del contesto globale (global environment), dell'esecuzione della logica ed, infine, della consegna dell'output al client.

L'Allegato A contiene una versione sterilizzata ed anonima di un tipico page script legacy preso da una reale applicazione. Mi sono preso la libertà di sistemare l'indentazione (che in origine era un po' casuale) e di impostare il ritorno a capo (wrapping) a 60 caratteri in modo che si adatti meglio agli schermi degli e-reader. Prova a dargli un'occhiata, ma attenzione … non mi assumo alcuna responsabilità se diventi cieco o se vai in stress post-traumatico!

Se lo esaminiamo, vi troviamo ogni sorta di errore che rende difficile la manutenzione e l'aggiornamento:

- `include` statements per eseguire il setup e la logica di presentazione
- definizioni di funzione inline
- variabili globali
- la logica "model, view, controller" è tutta insieme in un singolo script
- massima fiducia nell'input utente
- possibile vulnerabilità a SQL injection
- possibili vulnerabilità al cross-site scripting
- chiavi degli array non quotate che generano notifiche
- blocchi `if` non racchiusi tra parentesi (l'aggiunta di una riga successiva al blocco non è effettivamente parte del blocco)
- ripetizioni da copia-e-incolla

L'esempio dell'_ Allegato A _ è relativamente decente. Ho visto altri script dove il codice JavaScript e CSS sono mescolati, dove vengono inclusi file remoti e dove coesistono tutte le possibili problematiche di sicurezza. Questo è solo (!) 400 righe. Altri script sono lunghi migliaia di righe che generano diverse variazioni di pagina tutte contenute in una singola istruzione `switch` con più di una dozzina di condizioni `case`.

Riscrittura o refactoring?

Molti sviluppatori, quando vengono in contatto con una tipica applicazione PHP, resistono poco prima che si sentano spinti a cestinarla e a riscriverla da zero. "Cestinarla è l'unico modo per stare sicuri!" è il grido di battaglia di questi programmatori entusiasti ed energici. Altri sviluppatori, pacati dalle loro precendenti dolorose esperienze, sono più cauti. Sono pienamente consapevoli della scarsa qualità del codice ma sanno anche che il male conosciuto è meglio di quello ignoto.

I pro ed i contro della riscrittura

La riscrittura completa è certamente un'opzione molto allettante ma i fautori della riscrittura, a volte, peccano d'orgoglio ritenendo di essere in grado di scrivere tutte le cose giuste al primo colpo. Pensano di sapere scrivere gli unit test, di saper applicare le migliori pratiche, di saper separare i contenuti secondo moderni pattern e di utilizzare l'ultimo framework o addirittura di scriverne uno ad hoc (loro sanno bene quali sono le loro esigenze). Siccome possono utilizzare l'applicazione esistente come riferimento/esempio,

credono che sarà necessario poco lavoro di trial-and-error (letteralmente "prova e vedi gli errori") per riscriverla. Immaginano che basti copiare le funzionalità di base già esistenti nel nuovo sistema e che le specifiche non implementate o mal implementate nel sistema esistente saranno aggiunte, o corrette, sin dall'inizio della riscrittura.

La riscrittura, per quanto è allettante, è anche fitta di pericoli. Joel Spolsky ha detto questo a proposito della riscrittura del vecchio browser Netscape Navigator avvenuta nel 2000:

> Netscape ha fatto il "peggior singolo errore strategico che un'azienda di software possa fare" decidendo di riscrivere il codice da zero. Lou Montulli, una delle 5 superstar della programmazione che ha scritto la versione originale del browser, mi ha inviato un'email per dire: "Sono completamente d'accordo, ed è una delle principali ragioni per cui ho dato le dimissioni da Netscape." Questa decisione è costata a Netscape 3 anni di lavoro. Sono stati tre anni in cui la società non ha potuto aggiungere nuove funzionalità, non poteva rispondere agli slanci competitivi di Internet Explorer, ed ha dovuto sedersi sulle proprie mani, mentre Microsoft mangiava il loro pranzo.
>
> – Joel Spolsky, Netscape Goes Bonkers[1]

Per questa ragione Netscape ha dovuto cessare l' attività.

Josh Kerr racconta una storia simile che riguarda TextMate:

> Macromates, una società indipendente che possedeva un editor di testo molto buono chiamato TextMate, ha deciso, nella versione 2, di riscrivere tutto il codice. Ci sono voluti 6 anni per ottenere una versione beta funzionante, che al giorno d'oggi è un eternità, ed hanno perso una grande quota di mercato. Quando hanno deciso di rilasciare la versione beta era troppo tardi e 6 mesi più tardi hanno abbandonato il progetto ed hanno deciso di caricarlo su Github come progetto open source.
>
> - Josh Kerr, TextMate 2 e perché non si dovrebbe riscrivere il codice[2]

Fred Brooks definisce l'impulso a riscrivere come "l'effetto secondo-sistema." Ha scritto in merito nel 1975:

> Il secondo è il sistema più pericoloso che un uomo possa mai progettare. … La tendenza generale è quella di eccedere con le specifiche di progetto includendo tutte le idee e gli abbellimenti che erano stati cautamente evitati nel primo. … L'effetto secondo-sistema è … la tendenza a perfezionare tecniche la cui esistenza è stata resa obsoleta da cambiamenti nelle ipotesi di base del sistema. … Come fa un responsabile di progetto a difendersi dall'effetto secondo-sistema? Lavorando con un architetto senior che abbia alle spalle almeno due sistemi.
>
> - Fred Brooks, The Mythical Man-Month[3], pp. 53-58.

Gli sviluppatori di oggi sono come gli sviluppatori di 40 anni fa. Mi aspetto che restino tali nel corso dei prossimi 40 anni; gli esseri umani rimangono esseri umani. L' eccessiva sicurezza di sè, l' insufficiente pessimismo e l' ignoranza della storia portano gli sviluppatori che tentano una riscrittura a credere che "questa volta sarà diverso".

[1] http://www.joelonsoftware.com/articles/fog0000000027.html
[2] https://joshkerr.com/textmate-2-and-why-you-shouldnt-rewrite-your-code/
[3] http://www.amazon.com/Mythical-Man-Month-Software-Engineering-Anniversary/dp/0201835959/

Perché la riscrittura non funziona?

Ci sono molte ragioni per cui la riscrittura funziona raramente, ma mi concentrerò su un solo motivo generale: l'intersezione di risorse, conoscenze, comunicazione e produttività. (Vedi The Mythical Man-Month[4] (pg. 13-26) per un' eccellente descrizione dei problemi correlati all' idea che le risorse e la pianificazione possano essere considerati elementi intercambiabili.)

Come in tutte le cose, abbiamo un limitato ammontare di risorse da mettere in campo per la riscrittura. Ci sono solo un certo numero di sviluppatori nell'organizzazione che dovranno fare *sia* la manutenzione del programma esistente *che* la scrittura della versione completamente nuova e gli sviluppatori che lavorano alla riscrittura non potranno lavorare sull'altra.

Il problema del cambio di contesto

Un'idea alternativa è quella di avere sviluppatori che trascorrono parte del loro tempo sulla vecchia applicazione e parte su quella nuova. Tuttavia, spostare uno sviluppatore tra i due progetti non sarà vantaggioso dal punto di vista della produttività. A causa del carico cognitivo legato al cambio di contesto, lo sviluppatore sarà meno della metà produttivo in ciascun contesto.

Il problema della conoscenza

Nel tentativo di contrastare le perdite di produttività dovute alla commutazione di contesto, l'organizzazione può decidere di assumere più sviluppatori. In questo modo alcuni sviluppatori possono quindi essere dedicati al vecchio progetto ed altri al nuovo. Sfortunatamente, questo approccio rivela ciò che F. A. Hayek chiama il problema della conoscenza[5]. Originariamente applicato al campo dell'economia, il problema della conoscenza vale anche anche per la programmazione.

Se allochiamo i nuovi sviluppatori alla riscrittura, non sapranno abbastanza sul sistema esistente, sui problemi esistenti, sugli obiettivi di business e, forse, nemmeno le migliori pratiche per rendere la riscrittura efficace. Sarà necessario formarli su questi argomenti, molto probabilmente usando gli sviluppatori esistenti. Questo significa che gli sviluppatori esistenti, che vengono relegati a mantenere il programma esistente, dovranno spendere molto tempo a trasferire conoscenza ai nuovi assunti. La quantità di tempo in questione non è banale ed il trasferimento di conoscenza dovrà continuare fino a quando i nuovi sviluppatori non saranno scafati quanto gli sviluppatori esistenti. Ciò significa che l'aumento lineare di risorse determina un aumento tutt'altro che lineare della produttività: un aumento del 100% del numero di programmatori determinerà un aumento di produttività inferiore al 50%, a volte molto meno (cfr The Miserable Mathematics of the Man-Month[6]).

In alternativa, si potrebbe allocare gli sviluppatori esistenti alla riscrittura ed i nuovi assunti al mantenimento del programma esistente. Anche in questo caso si incappa nel *problema della conoscenza* perché i nuovi sviluppatori sono hanno familiarità con il sistema. Da dove trarranno la conoscenza necessaria per fare il loro lavoro? Dagli sviluppatori esistenti, naturalmente, che dovranno ancora spendere tempo prezioso per trasferirer le loro competenze. Ancora una volta, si vede che l'aumento lineare del numero di sviluppatori porta ad un aumento tutt'altro che lineare della produttività.

La pianificazione della tabella di marcia

Per affrontare il *problema della conoscenza* ed i relativi costi di comunicazione, alcuni possono pensare che il modo migliore per gestire il progetto sarebbe quello di dedicare tutti gli sviluppatori esistenti alla

[4] http://www.amazon.com/Mythical-Man-Month-Software-Engineering-Anniversary/dp/0201835959/
[5] http://www.econlib.org/library/Essays/hykKnw1.html
[6] http://paul-m-jones.com/archives/1591

riscrittura ritardando la manutenzione e gli aggiornamenti del sistema esistente fino a quando la riscrittura non sarà pronta. Questa scelta fa gola perché gli sviluppatori sono fin troppo ansiosi di placare i loro dolori e di diventare "clienti di loro stessi" - ovvero entusiasti di implementare e correggere quello che a loro piace. Questa bramosia li porterà a sovrastimare la propria capacità di eseguire una riscrittura completa e a sottovalutare la quantità di tempo necessaria per completarla. I manager, dal canto loro, accetteranno di buon grado l'ottimismo degli sviluppatori, magari aggiungendo qualche prudenziale misura compensativa alla tabella di marcia.

L'eccessiva confidenza e l'ottimismo degli sviluppatori si trasformerà velocemente in frustrazione e pena quando si renderanno conto che l'attività è, in realtà, molto più complessa e schiacciante di quanto pensassero. La riscrittura durerà molto più del previsto e non di poco, ma di qualche ordine di grandezza o più. Per tutta la durata della riscrittura, la versione esistente languirà - zeppa di bug e di feature mancanti - deludendo i clienti esistenti ed incapace di attrarne di nuovi. La riscrittura, alla fine, diventerà una spaventosa marcia della morte verso la conclusione a tutti i costi della nuova versione ed il risultato sarà codice tanto scadente quanto il primo. Sarà semplicemente una copia del primo sistema perchè, a causa delle pressioni della tabella di marcia, si sarà deciso che le nuove caratteristiche (feature) verranno ritardate finchè non si avrà un primo rilascio.

Refactoring iterativo

Tenuto conto dei rischi connessi alla completa riscrittura, raccomando invece il refactoring[7]. "Refactoring" *(letteralmente "rifattorizzare")* significa che la *qualità* del programma viene migliorata a piccoli passi, senza modificare la *funzionalità* del programma. Un unico, relativamente piccolo, cambiamento viene introdotto in tutto il sistema. Il sistema è poi testato per assicurarsi che funzioni ancora correttamente ed, infine, viene messo in produzione. Si introduce poi un secondo piccolo cambiamento basato sul precedente e così via. Dopo un periodo di tempo il sistema diventa notevolmente più facile da mantenere e migliorare.

Il refactoring è decisamente meno attraente di una completa riscrittura. Sfida la sensibilità della maggior parte degli sviluppatori. Gli sviluppatori devono continuare a lavorare con il sistema così com'è, con tutti i suoi problemi, per lunghi periodi di tempo e non hanno la possibilità di passare all'ultimissimo framework. Non possono diventare "clienti di loro stessi" e soddisfare il desiderio di "fare la cosa giusta al primo colpo". Il refactoring, essendo una strategia a lungo termine, non è attraente per una cultura che valorizza lo sviluppo rapido di nuove applicazioni piuttosto che aggiustare quelle esistenti. Gli sviluppatori di solito preferiscono avviare i propri nuovi progetti piuttosto che mantenere progetti precedenti sviluppati da altri.

Tuttavia, come strategia di riduzione del rischio, l'approccio del refactoring iterativo è innegabilmente migliore della riscrittura. I singoli passi di refactoring sono piccoli rispetto ad una qualsiasi parte del processo di riscrittura. Possono essere introdotti in periodi di tempo molto più brevi di quanto non accadrebbe in una riscrittura e mantengono l'applicazione funzionante alla fine di ogni iterazione. Non esiste un momento del refactoring iterativo in cui l'applicazione smette di funzionare o di progredire. Il refactoring iterativo può essere integrato in un processo più ampio di pianificazione che permette cicli di correzione bug, di aggiungere funzionalità o di fare refactoring per migliorare il ciclo successivo.

L'obiettivo di ogni singolo passo di refactoring non è la "perfezione". L'obiettivo di ogni passo è semplicemente il "miglioramento". Non si cerca di realizzare un obiettivo impossibile per un lungo periodo di tempo. Si fanno piccoli passi verso obiettivi facilmente visualizzabili e realizzabili in tempi brevi. Ogni piccolo successo nel refactoring alza il morale ed esorta alla successiva iterazione. Successo dopo successo si arriva ad un' unica grande vittoria: un codice completamente modernizzato che non ha mai smesso di generare entrate.

[7] http://refactoring.com/

Framework legacy

Fino ad ora, si è discusso di applicazioni legacy come di agglomerati di script include-oriented. Tuttavia esiste un gran numero di applicazioni legacy basate su framework non proprietari.

Applicazioni legacy basate su framework

Ogni framework PHP ha le sue problematiche. Le applicazioni scritte in CakePHP[8] soffrono di problematiche legacy diverse da quelle scritte in CodeIgniter[9], Solar[10], Symfony 1[11], Zend Framework 1[12] e così via. Ciascuno di questi diversi framework, e loro derivati, incoraggiano le applicazioni a diversi tipi di tight-coupling *(letteralmente: accoppiamento-stretto) _(chiarimento: è un termine comune che indica che le classi sono fortemente dipendenti da altre classi e ciascuna ha troppe responsabilità).* Perciò i passaggi specifici necessari per il refactoring di applicazioni costruite utilizzando un framework sono molto diversi dai passi necessari per gli altri framework.

Per questo motivo le varie parti di questo libro possono essere utili come guida per il refactoring di diverse parti di un'applicazione legacy basata su framework ma, nel complesso, il libro non si focalizza sul refactoring di applicazioni scritte con uno specifico framework.

I framework proprietary sviluppati sotto il controllo diretto degli architetti dell'azienda molto probabilmente **troveranno beneficio** dalle tecniche di refactoring spiegate in questo libro.

Refactoring in un framework

A volte mi capita di sentire alcuni sviluppatori che saggiamente vogliono evitare una completa riscrittura e che invece vogliono fare "refactoring" o "migrare" verso un framework pubblico. Questo sembrerebbe il miglior compromesso tra i due mondi (riscrittura e refactoring), combinando l'approccio iterativo con il desiderio degli sviluppatori di utilizzare nuove tecnologie.

Secondo la mia esperienza, però, le applicazioni PHP legacy sono tanto resistenti all'integrazione di framework quanto lo sono allo unit testing. Se l'applicazione fosse già in uno stato in cui la logica può essere portata in un altro framework, allora non ci sarebbe bisogno di migrarla.

Tuttavia, dopo aver completato i passi di refactoring illustrati in questo libro, è molto probabile che l'applicazione sia in uno stato tale da consentire una facile migrazione ad un framework pubblico. Se gli sviluppatori avranno ancora voglia di farlo è un'altro discorso.

Riassunto e fasi successive

A questo punto, ci siamo resi conto che una riscrittura, per quanto attraente, è un approccio pericoloso. Un approccio di refactoring iterativo suona molto più noioso, ma ha il vantaggio di essere realizzabile e realistico.

[8] http://cakephp.org
[9] http://codeigniter.com
[10] http://solarphp.com
[11] http://symfony.com/legacy
[12] http://framework.zend.com

Il passo successivo è quello di prepararci per il refactoring togliendo di mezzo alcuni prerequisiti. Dopo di che, si procederà verso la modernizzazione della nostra applicazione legacy con una serie di relativamente piccoli passi – un passo per capitolo – ciascuno dei quali suddiviso in processi facili da seguire e con risposte alle domande più comuni.

Iniziamo!