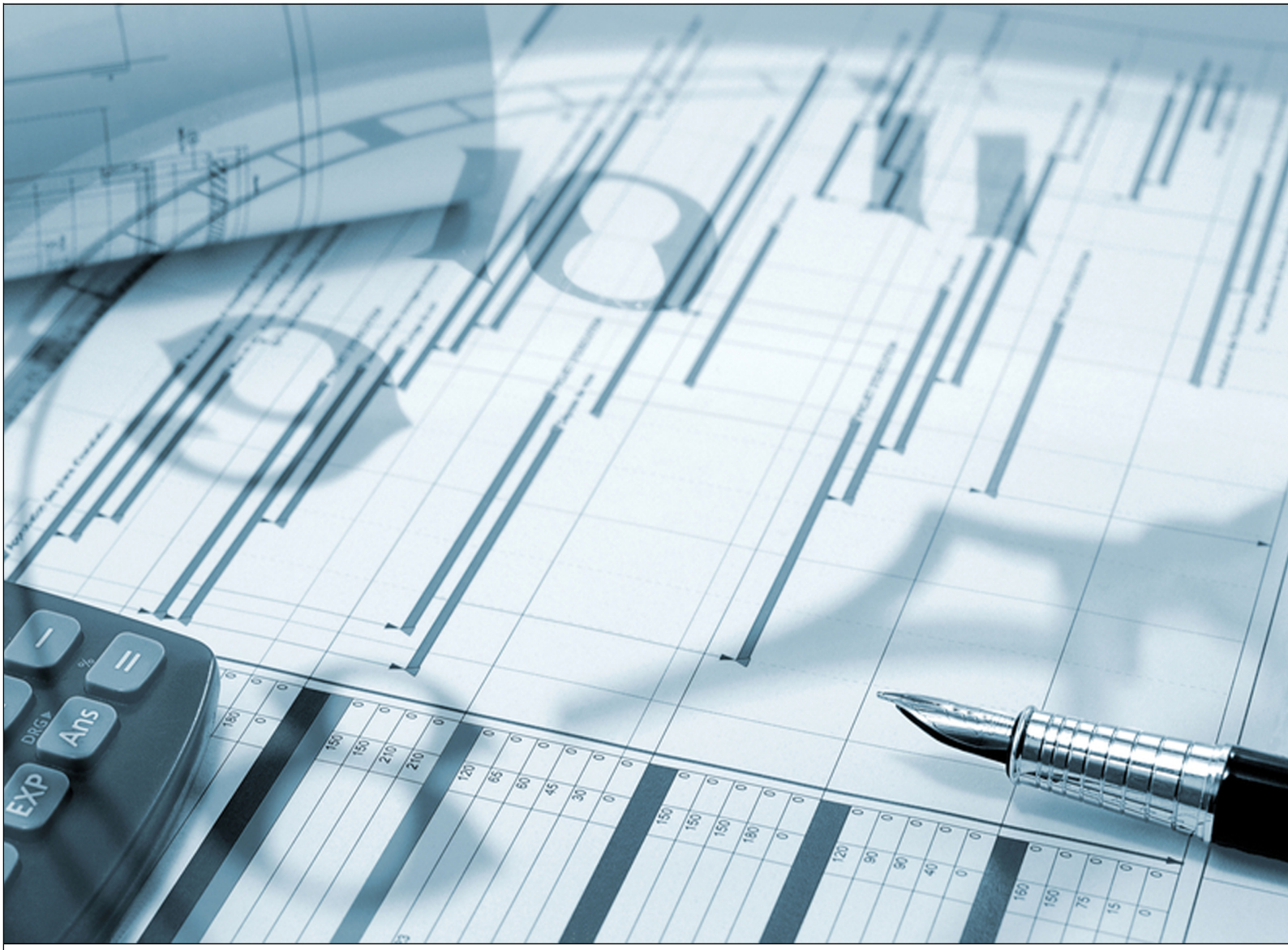

Laravel4でこなすプログラム術

Getting Stuff Done with Laravel 4

ホットで最新のフレームワークを使い、
アプリケーション設計と開発の旅に出かけよう

著者：Chuck Heintzelman 翻訳：Hirohisa Kawase

Laravel4 でこなすプログラム術　Getting Stuff Done

ホットで最新のフレームワークを使い、アプリケーション設計と開発の旅に出かけよう

Chuck Heintzelman and Hirohisa Kawase

This book is for sale at http://leanpub.com/gsd-laravel-jp

This version was published on 2014-02-26

Tweet This Book!

Please help Chuck Heintzelman and Hirohisa Kawase by spreading the word about this book on Twitter!

The suggested hashtag for this book is #LaravelGSD.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#LaravelGSD

Contents

謝辞

この書籍を購入いただいた皆様へ、心から感謝いたします。この本が皆様にとって興味深く、楽しめる書籍でありますよう、とりわけ一番に、お役に立てるように願っています。

Laravel の関連書

- 公式サイト[1]　なにか疑問があれば最初にチェックすべき場所です。フォーラムをチェックしてみてください。情報の宝庫です。
- NetTuts[2]　良い Laravel のチュートリアルが、いくつか存在しています。
- Laravel Testing Decoded[3] 著者：Jeffery Way（日本語版[4]）　この書籍は、あなたの Laravel コードをどうやってテストするのか教えてくれる、素晴らしい情報源です。
- Code Bright[5] 著者：Dayle Rees（日本語版[6]）　この書籍は面白く、情報にあふれています。
- From Apprentice to Artisan[7] 著者：Taylor Otwell（日本語版[8]）　著者は Laravel の開発者です…これ以上、話す必要はないでしょう。
- Implementing Laravel[9] 著者：Chris Fidao（日本語版[10]）　この書籍は Laravel を使用したプロジェクトの実証に的を絞っています。構造と一般的に利用されるパターンをカバーしています。素晴らしい本です。
- Laravel 4 Cookbook[11] 著者：Christopher Pitt（日本語版[12]）　この書籍は Laravel4 で構築された、色々なプロジェクトを紹介しています。
- Laravel in Action[13] 著者：Maks Surguy　この本は現在、Manning Publications 社の Early Access program から入手できます。

---

[1] http://laravel.com
[2] http://net.tutsplus.com/
[3] https://leanpub.com/laravel-testing-decoded
[4] https://leanpub.com/laravel-testing-decoded-japanese
[5] https://leanpub.com/codebright
[6] https://leanpub.com/codebright-jp
[7] https://leanpub.com/laravel
[8] https://leanpub.com/laravel-jp
[9] https://leanpub.com/implementinglaravel
[10] https://leanpub.com/laravel4cookbook-jp
[11] https://leanpub.com/laravel4cookbook
[12] https://leanpub.com/laravel4cookbook-jp
[13] http://www.manning.com/surguy/

改訂履歴

現在のバージョンは 1.2 です。

バージョン	日付	変更点
1.2	2014 年 2 月 2 日	誤記修正と Laravel4.1 対応
1.1	2013 年 11 月 28 日	誤記とその他全般的に修正
1.0	2013 年 11 月 2 日	誤記と Leanpub 出版物の修正
0.9	2013 年 10 月 27 日	原稿完成。Leanpub で出版。
0.8	2013 年 10 月 20 日	4 章追加。Leanpub で出版。
0.7	2013 年 10 月 23 日	3 章追加。Leanpub で出版。
0.6	2013 年 10 月 6 日	8 章追加。パート 3 完成。Leanpub で出版。
0.5	2013 年 9 月 29 日	パート 3 に 7 章追加。Leanpub で出版。
0.4	2013 年 9 月 22 日	パート 3 に 7 章追加。Leanpub で出版。
0.3	2013 年 9 月 15 日	パート2全体を推敲。初版を Leanpub で出版。
0.2	2013 年 9 月 8 日	パート2の原稿完成。
0.1	2013 年 8 月 31 日	冒頭とパート 1 の原稿完成。
0.0	2013 年 8 月 3 日	原稿執筆開始。

感謝の辞

このリストは誤りを見つけたり、その他の間違いを指摘していただき、この本の品質向上に助力して下さった方々です。

- Peter Steenbergen
- Jeremy Vaught
- George Gombay
- Mike Bullock
- Kristian Edlund

ご協力に感謝いたします。

表紙

[14] http://www.dreamstime.com/kemaltaner_info
[15] http://www.dreamstime.com/
[16] http://www.dreamstime.com/kemaltaner_info
[17] http://www.dreamstime.com/

サンプル版について

これはダウンロードサンプル版です。書籍全体の一部です。

「ようこそ」の章から 8 章、それと「堅牢なオブジェクトデザイン」からの 1 章をお読みください。これで私のライティングスタイルと、どうしてこの書籍が役に立つのか、ご理解いただけるでしょう。

有料版の内容

Leanpub の紹介ページ[18]で、目次をご覧いただけます。

この書籍の完全版が皆さんのお役に立つかを判断いただき、購入の検討をお願いします。

[18] https://leanpub.com/laravel4cookbook-jp

ようこそ

「*Laravel4* でこなすプログラム術 *Getting Stuff Done*」へようこそ。この**ようこそ**パートでは、この本から得られる内容を説明します。

全体の概要をご覧ください。

ようこそ
: この書籍から得られる内容を説明します。

パート1： 設計哲学と原則
: このパートはアプリケーション作成時に従うべき、一般的な設計原則についてお話します。

パート2 ： アプリケーションの設計
: 実際のアプリケーションを設計するパートです。

パート3 ： コンソールアプリケーション
: 次に、コンソールから使用できるアプリケーションを作成します。

パート4 ： Web アプリケーション
: 今度は、Web アプリケーションを作成し、設置します。

追補
: 補助的な情報です。例えば、Composer のインストール方法などです。

第１章　この本の目的

この本は Laravel に乗り込み、あなたを旅へと誘います。できれば、皆さんをまだ行ったことのない場所へ案内し、まだ見ぬ景色をお見せできますように。これは旅行談です。私達は目的地（作成するアプリケーション）を決め、そこへ向かう道すがら、景色の素晴らしい場所を案内していきましょう。目的地に到着したら、私に手紙でも投函してください。(chuckh@gmail.com[19]) この旅の感想に、興味津々なのです。

これは経験するための本です。使用するための本です。内容に従い、各章のアプリケーションを作成してください。各章の内容は連続しています。各章の中のセクションも、前の内容の続きです。この本の全箇所は、それ以前の内容を基にし、構成してあります。

各章の中のセクションは、「都市」だと考えてください。章は国です。パートは大陸で…**はい、はい。分かりました。無理やり旅行に例えるのは、もう十分ですね。**

この本全体を通し、Laravel 4を利用したアプリケーションの段階的な作成に主眼を置いています。

典型的なマニュアル本ではありません

できるだけ実際の設計と開発のプロセスにそっくりになるように書きました。出だしでしくじり、設計を変更し、リファクタリングを行なっています。

ニヤつかないように、忠告しておきますよ。

この本で行わないこと

- Laravel の全部を取り扱いません。この本はフレームワーク全体の参考書ではありません。
- キャッシュ、イベント、ログは取り扱いません。重要な話題ですが、これから作成するアプリケーションには必要ありません。
- キュー、認証、クッキー、セッション。これらも重要です。しかし、必要ありません。
- データベース。これを決めるのは心苦しいことでした。*Fluent* **クエリービルダー**と *Eloqunet ORM* は Laravel のとても重要な機能ですからね。とても有名です。評判通り実装は完璧です。でも、残念ながら、使用しません。なぜなら…お考えの通り…作成するアプリケーションには必要ないからです。

では、何を教えるのでしょう。

主に、アプリケーションの作成に置いて、私が何を行なっているか、どうして行なっているかの秘密をばらしています。ある部分には、みなさん同意してくれるでしょう。ある場合では、きっと

[19] mailto:chuckh@gmail.com

皆さんは私を完全に抜けていると考えるでしょう。そういう時は、できれば　**「ああ、そうか。面白い冗談だな。」**　と考えておいてください。しかし、最終的には、使える本当のシステム作成の根本を身に付けてもらえるでしょう。

第２章　読んでいるのは、誰ですか？

ほとんどの書籍では著者についての情報から始まりますが、本当に重要なのは「読者の皆さんが誰なのか？」です。

私は皆さんを以下のような方々であると推測しています。

- 大抵の人々より、コンピューターについて詳しい。
- プログラマーである。
- PHP でどのようにプログラムを組むか理解している。たぶん多少なりとも知っている。たぶん良く知っている人もいるだろう。
- Laravel[20]について聞いたことがある。（**これは、さほど重大な問題ではありません。これから *Laravel* について説明するのですからね。**）
- プログラミングが好きだ。もしくは、最初にコンピューターへ命令を従わせた頃のような、情熱を取り戻したいと願っている。
- あなたの名前が Taylor Otwell[21]でないこと。なぜなら、もしもそうであるならば、この本は価値がないだろう。

初心者にも学んでもらえるように、また中間レベルのプログラマーの方でも十分に興味深く読んでもらえるよう、ベストを尽くしました。

最低ライン

Laravel についてもっと学びたい意志を持っていることです。

---

[20] http://laravel.com

[21] Taylor Otwell 氏は、Laravel の開発者です。実際、コンセプトからコーディング、テストまで、ほとんど一人で開発しています。

第３章　私は誰でしょう？

私についての章ですから、熱心に読むようものではありません。なにせ、Laravel については、一言も触れませんからね。賢い皆さんなら、今すぐ次の章へ飛ばすでしょう。

こんにちは。私の名前は、Chuck Heintzelman で、コンピュータープログラムを書いています。

（何だかサポートグループのフロントマンだった頃のようです。お願いですから、”Hi Chuck” と返事しないでください。）

真面目な話、9学年の時からプログラムを書いています。学校から借りていた *BASIC* **言語リファレンス** マニュアルにくびっきりになり、アステロイド[22]に似た薄っぺらいゲームを書いていました。違いは小惑星があなたに向かって飛んでくる代わりに、他の宇宙船から死の細長い白いブロックが発射されることでした。

長時間に渡るデバッグの後、TRS-80 へプログラムを「大容量ストレージ」（カセットテープのことです）に保存／読み込みをじっと待ったあと、そのゲームはついに動作しました。実に33年前のことです。コンピューターの恐竜時代に戻ってみれば、大きな獰猛な猛獣が温度調節された部屋に詰め込まれていました。いいえ、**本当に** パンチカードを使ったことはありませんよ。けど、使われているのを見たことならあります。

それから、Fortran、COBOL（ええ、知っていますとも）、アセンブラ言語、Basic、C、C++、C#、Java、Pascal、Perl、Javas、PHP でプログラムを書いてきました。他にも多くの、とても多くの言語を触って来ました。でも、実際に他の人に使ってもらうプログラムは書いていません。

Fortune500 に入っている会社のために、システムを作成したこともあります。同時に、家族だけで営業している商店のためにも作成したことがあります。Xenix 上で動作しているメールオーダーシステムから、PHP で動作している Web アプリケーションまで全てです。それから、今日のインターネットの繁栄前（**本当の** インターネットの始まりではありません。90年台中頃に始まった刺激的なネットブームの前のことです）に、いくつかの会社と.com を立ち上げました。こうした経験を通じ行なってきたこと、つまりコンピュータープログラムを書くことを私は愛しています。

やれやれ？はい、わかりました。私の自慢話はもううんざりですね。

私の言いたいことは

私のキャリアーの中で、今までプログラムの本を書こうと思ったことは、一回もありませんでした。これを書いている唯一の理由は、Laravel だからです。

[22] http://ja.wikipedia.org/wiki/%E3%82%A2%E3%82%B9%E3%83%86%E3%83%AD%E3%82%A4%E3%83%89

第４章　Laravelって、何？

よく耳にする人は、手を上げてください。

> 今まで皆さん、自分の会社で既存のシステムへ、機能追加をした経験をお持ちでしょう。不幸なことに、そのシステムが PHP4 で書かれていて、誰がもともとのプログラマーであったにせよ、彼らは「Wordpress に夢中」動画を何度も見たのであろうと推測するでしょう。
>
> クラス未使用、たっぷりのグローバル変数、５万ピースのジグソーパズルそっくりな構造を「ため息」と共に、引き付いだのでしょう。
>
> あなたは自分の仕事、狭い視野しか持たない管理チーム、最初にプログラムでお金を稼ぐことを思い立たせた何かを恨みます。
>
> **何と言っても、プログラムは楽しくなくてはなりません。そうでしょう？**

ここに全てがあります。

Laravel へ飛び込んでください。

（はい、ここでドラムロール！*ダダダダ、ダダダダ、ダダダダ、ダダダダ*。）

Laravel はプログラミングを再び楽しくしてくれる、PHP フレームワークです。

冗談だろう…ただのフレームワークかよ！

Laravel は新しい言語ではありません。フレームワークに過ぎません。全ての誇張を取り去り、ただ観察してみれば、Laravel はただの PHP フレームワークです。

たとえそうだとしても、私は Laravel の Web サイトのモットーに深く共感します。

「Web 職人のための PHP フレームワーク」

Ruby on Rails は、**ただのフレームワークです。**それでも、その背景には自由を見て取ることができます。

Laravel は決して PHP スパゲティーコードを魔法のように修正してくれません。しかし、仕事をこなすための新しくて早く、エレガントな方法を提供してくれます。**（注目：仕事をこなす *(Getting Stuff Done)* のコンセプトはこの本の中に何度か現れてきます。）**

簡単に言えば、Laravel は PHP プログラミングを楽しくしてくれる構造を提供しているのです。あなたは読み書きしやすい、スタリッシュで保守がしやすく、将来の拡張にそなえた方法へ、既に存在するコードをリファクタリングすることができるようになります。

Laravel は万能薬ではありません。既にあるコードベースがひどければ、**「いま、どこにいるんだ」**を**「どこですべきか」**に改善するには、苦痛が伴います。これは私達、ソフトウェア産業の本質なのです。

しかし、あなたがシンプルで、読み書きしやすく、**（ここにもっと、どんな単語を追加したら良いでしょうか？）**してくれるフレームワークに変更しようと考えているのなら、Laravel が答えです。

第５章　Laravel 使用の正当性

こんな問題があります…

> あなたは自分が所属している会社の束縛の下で働いています。すなわち、既存のソフトウエアをサポートし、既存のシステム上で滞りなく動作する新しいコードを開発しなくてはなりません。.Net と Java が混じっているかも知れませんが、存在するほとんどのコードは PHP です。

最近、あなたは Laravel を見つけ、惚れてしまいました。新しい開発に使用したいと思っています。

どうやって、Laravel へ移行する正当化を主張したらよいのでしょうか？

しばらく、探偵の役を演じてみましょう。

ふーむ。私の知っている探偵は（テレビのシリーズだけですが）、証拠と動機を探す場合、お金の流れを追いかけるようです。ですから、お金を追求しましょう…

顧客は商品とサービスの交換として、お金を提供します。良い商品であれば、より多くの顧客が欲しがり、より多くのお金をビジネスに支払ってくれます。

管理者達はビジネスを繁栄させたいことでしょう。彼らはできるだけ多くの顧客と、できるだけ頻繁に、できるだけ高額の取引を望みます。

管理者の視点で考えてみましょう…

- 顧客をより幸せにしたい。
- 新しい顧客を獲得したい。
- 顧客の幸せとは、期待に答えることだ。
- プログラーマーたちには、時間通りに要求を提供できるようにしたい。
- プログラムチームはアジャイル（機敏）でいて欲しい。（**これが意味しているのが何であれ…後述をご覧ください。**）
- 顧客が望む機能をタイミングよく提供したい。
- 素晴らしい製品を提供する、素晴らしい開発者が欲しい。

アジャイルとは何を意味してきたのか？

皆さん、余りにも頻繁にこの言葉を言ったり書いたりして、意味なんて無くなったんじゃないですか？ほとんどスマーフと同じでしょう・・・全てスマーフしている、スマーフ可能、スマーフ的。[a] アジャイルもこうした言葉の一つです。昔のもったいぶった言葉です。これが全部アジャイルです、あれがアジャイルです。人々は繰りかえすソフトウェアの手順や何か他のことについて話しているのでしょうか？それとも魔法のようなことについてでしょうか？私は本当

に分かりません。

[a]スマーフは身元を隠してゲームに参加するプレーヤーを指す言葉です。意味のない言葉の代表として使用されています。

上のリスト項目が管理者の視点であれば、Laravel の使用は簡単に正当化できるでしょう。

- 顧客はニーズを提示され、それを満たされた時、幸福である。
- 顧客は期待が実行されると、更に幸福になる。
- Laravel フレームワークが提供するのは…
  - 機能の拡張を用意にする。
  - 設計のベストプラクティスに従わせる。
  - 多くのプログラマーに効果的に協力させられる。
  - プログラマーを幸福にする。（管理者が覚えておくこと：幸福なプログラマーは生産的なプログラマーである。）
  - 仕事をこなすのが早くなる。
  - ユニットテストと、アプリケーションのコア構成物を熟考しテストすることを推奨できる。

プログラマーたちがより多く、より早く仕事をこなし、Web アプリケーション開発で代々伝承されている多くの障害物を取り除く能力を Laravel は管理者へ授けてくれます。

正当化するのは簡単です。ですよね？

第 6 章　なぜプログラマーは Laravel を好むのか

早速、本題に入りましょう…なぜ、プログラマーはフレームワークに Laravel を使いたがるのでしょうか？

フレームワークへの憧れについて、少し話をさせてください。

> （今、セラピストと話している映像を思い浮かべています。彼は賢人ぶって頷いています。パイプで一服し、一言話しかけます。「Zee フレームワークへの憧れついて、話してください。」）[23]

私は PHP で書かれたプロジェクトに携わらなくてはなりませんでした。学校における飛び級のように、**"class"** の概念をすっ飛ばした開発者により書かれた、PHP4 のプロジェクト達は膨大に膨れ上がっていました。通りの向こうには Ruby 開発者が見え、地震、竜巻、もしくは雷でもかまわないので、彼らのいるビルの、その階が自然の脅威に見舞われることを静かに祈っていました。

（これが私を悪い人間にしたのでしょうか？）

これは Ruby がまだ新しく、輝いていた時のことです。（Ruby は言語として素晴らしく素敵な一面を持っているにせよ）Ruby を素敵に思わせたのは、言語自身ではありません。

全ての開発者は Ruby on Rails に群がりました。

では、なぜ彼らは群がったのでしょう？

なぜなら、面白い開発方法を約束していたからです。面白いとは、パワフルで、読み書きしやすく、素早い開発のことです。私は RoR がプログラミングを再び、喜びに満ちたものにする雰囲気を醸しだしたのだと確信しています。RoR によりもたらされるコーディングの喜びは、私達全員にプログラムへの欲求を起こさせる、プログラミングを始めた頃の刺激と同じものです。

PHP 界を鑑みると、とても悲しい状態だったでしょう？太郎に次郎に花子さん[24]は「PHP プログラマー」です。だって、Wordpress のインストールができるんですから。

（Wordpress については次の章、「WrodPress: 良い点、悪い点、ひどい点」をご覧ください。）

しかし、だめでした。プロジェクトは PHP のままでという要求に縛られ続けました。私達は Ruby 開発者のように、格好良くはいきませんでした。彼らは最先端でした。彼らは言語の垣根を超え、名声を手にした人々です。

そこにやってきたのが Laravel です。Laravel は Ruby on Rails の一番良い部分を PHP 界に取り入れてくれました。突然、PHP 開発者は別々のスクリプトの代わりに、コントローラーへのルートを使い始めました。DRY(Don't Repeat Yourself: 自分で繰り返すな）のコンセプトが意

[23] Zee に特別深い意味はありません。Zee フレームワークが実在するわけでありません。

[24] 原文は"Tom, Dick and Harry" で、「普通の人、平均的な人」という意味です。

味を持ち始めました。Smarty テンプレートだけが提供していた方法で、PHP の真髄と融合した”Blade” テンプレートエンジンを突然手にしたのです。素早く開発でき、読み書きしやすい、PHP のポテンシャルにおける解脱の境地を手にしたのです。

私が Laravel が凄いと言っているように聞こえますか？正にその通りです。

第 7 章　Wordpress：良い点、悪い点、ひどい点

Wordpress はブログに革命を起こしました。ブログを大衆のものにしました。もちろん、Blogger や Livejournal のような他のプラットフォームもありますが、Wrodpress は PHP により書かれたシステムとして、パブリックドメインの中で大きく、人気を得たのです。

Wordpress の登場により、誰もがブログプラットフォームを好きなように動かすため、PHP スクリプトをハックできるようになりました。

「大きな力には、大きな責任が伴う。」Ben おじさん（スパイダーマンより）

残念ながら、力強い Wordpress の利用には、責任が伴わなかったようです。スクリプトは全体の設計やユーザビリティを全く考慮されずハックされました。問題に輪をかけたのは、Wordpress がリリースされたのは PHP4 の時代であり、言語には真のプログラマーが保守可能なシステムを構築できるだけの能力がありませんでした。

Wordpress は PHP に起きた最高の出来事でしたが、言語としての面では最悪の出来事だったのです。

少なすぎる職人の管理下で、多すぎる成功が生まれたケースです。

これにより、PHP に汚名が被せられました。

Softwarati[25]

> 定義：自分の考えに夢中になり、言語に対しコメントを言う、プログラムの知識人。

たぶんあなたが考えているのは…Softwarati によって、良く引用されるセリフのことでしょう。

「ああ、PHP はかわいそうな人々の言語だろう。ひどいし、保守できないしね。でも動くんだよね…ほとんどの場合。」

ありがたいことに、Softwarati のお高くとまった鼻面に蹴りを入れるため、Laravel がやってきました。

[25]もちろん、この言葉は私が勝手に作った単語です。

第８章　本書の規約

この書籍では、規約をいくつか使用しています。

コードは２つの空白でインデント

通常、私は4つの空白をインデントで使用していますが、この書籍は様々な電子書籍フォーマットとして利用されるため、小さいスクリーン向きに、横幅を狭くしておくのは重要です。

```
for ($i = 0; $i < 10; $i++)
{
  echo "I can count to ", $i, "\n";
}
```

これはヒントです

これは特に便利な情報にハイライトを当てる場合に使用します。

これは警告です

これは何か、注意喚起を促す場合に使用しています。

これは情報ブロックです

これは重要な情報を繰り返す場合に使用します。

何かを行う場合です

この記号があったら、コーディング、もしくは他の行動を実際に行なってください。指示が書かれています。

?> はタグが開かれている場合に使用しています。

実際のコーディングでは、最後の?> をいつも省略しています。この書籍を書いているエディターでそうしてしまうと、全てがおかしくなってしまうのです。そのため、この本の中では PHP ブロックを PHP タグで開いた場合、毎回確実に閉じています。例えば:

```php
<?php
class SomethingOrOther {
    private $dummy;
}
?>
```

PHP の開始タグと終了タグ

コードのサンプルの中で、時々必要がない PHP の開始タグ (<?php) が使用されています。（例えばファイルの一部分を示す場合です。）時々、終了タグ ('?>') も不必要ですが、ついています。

```php
<?php
  function somethingOrOther()
  {
    $this->callSetup();
  }
?>
```

本当の PHP コードでは、私は **いつでも** ファイルの最後の終了タグを省略しています。タグが必要かどうかは皆さんの判断に委ねています。注意していただきたいのは、サンプルコード中の開始タグと終了タグを書かれたまま受け取らないでください。

どんな OS を使用しているか？

私はこのマニュアル、コード、その他を Debian、Ubuntu ベースのLinux Mint 16[26]上で作成しています。これは基本的にはUbuntu 13.10[27]と同じものです。

[26] http://www.linuxmint.com/
[27] http://www.ubuntu.com/

パート１　設計哲学と原則

この本の中でも、このパートはコードが最もありません。すいませんが、全て設計について書かれています。最初の時点で、書籍中でも使用されている、一般的な設計原則について説明しておこうと思います。

たぶんあなたは、「コードを見せてくれよ」と思っているでしょう。私もほとんどのページでは、あなたに同意します。多くの場合、一番早くて簡単に学ぶ方法は、ただコードに飛び込むことですしね。もし、堅牢 (SOLID) オブジェクト設計、契約としてのインターフェイス、依存注入、疎結合化、制御の逆転を理解しているのでしたら、アプリケーションの設計を行うパート 2 へスキップしてください。

第１６章　堅牢 (SOLID) なオブジェクト設計

SOLID（堅牢）と呼ばれるオブジェクト指向の原則があります。それぞれの頭文字を取ったものです。

- 単一責任原則 (**S**ingle Responsibility)
- 開放／閉鎖原則 (**O**pen/Close Principle)
- リスコフ置換原則 (**L**iskov Substitution Principle)
- インターフェイス分離原則 (**I**nterface Segregation Principle)
- 依存関係逆転原則 (**D**ependency Inversion Principle)

これを合わせると、ベストプラクティスの代表です。従うなら、開発するソフトウェアは保守が簡単になり、いつでも拡張が容易になります。

この章では、各原則の詳細を説明していきます。

私を SOLID にして？

もしプログラマーがあなたのところにやってきて、「私を SOLID にしてくれますか？」と尋ねたら、何を尋ねているのか明確にしましょう。[28]

単一責任原則

SOLID（堅牢）の最初の文字、'S' は"Single Responsibility" の頭文字です。この原則の内容は：

> 「すべてのクラスは、単一の責任を持たなくてはならない。その責任はクラスにより完全にカプセル化される必要がある。そのサービスは、その責任に厳格に沿わなくてはならない。」

別の考え方をすれば、クラスは一つだけ、唯一の、変更理由を持たなくてはなりません。

市場リポートを編集し、ユーザーに送信する役割を持ってるクラスがあったとしましょう。配布方法の変更があればどうなるでしょう？ユーザーがテキストでレポートを欲しがった場合は？もしくは、Web で見たがった場合は？レポートのフォーマットを変更したい場合は？レポートの情報源を変更する場合は？この例のクラスには、こうした多くの理由で変更が起きる可能性があります。

[28] solid はソフトウェア的な文脈では「堅牢」、「しっかりした」という意味ですが、一般的には「固体の」、「濃い」、「堅実な」という意味です。原文は"Can you do me a SOLID?"です。今回は名詞として使用されており、その場合は「固体」、「塊」という意味になり、どの様な意味かはっきりしません。いささかセクシャルな意味合いにも取れるので、ジョークでしょう。

AND（〜と）をクラスの目的に使わない

単一責任原則を実践する簡単な方法は、クラスが何を行うのかを定義するために、「〜と」や、「〜して」「および」という類の言葉 (AND) を使わないことです。

これをコードで示してみましょう。

```
1  // クラスの目的：市場レポートを組み立て「て」、ユーザーにメールする
2  class MarketingReport {
3    public function execute($reportName, $userEmail)
4    {
5      $report = $this->compileReport($reportName);
6      $this->emailUser($report, $userEmail);
7    }
8    private function compileReport($reportName) { ... }
9    private function emailUser($report, $email) { ... }
10 }
```

このケースの場合、クラスを2つに分け、どんなレポートを送るのか、誰にどうやって送るのかをより上位の責任者に決定してもらいましょう。

```
1  // クラスの目的：市場レポートを組み立てる。
2  class MarketingReporter {
3    public function compileReport($reportName) { ... }
4  }
5
6  interface ReportNotifierInterface {
7    public function sendReport($content, $destination);
8  }
9
10 // クラスの目的：ユーザーにレポートをメールする。
11 class ReportEmailer implements ReportNotifierInterface {
12   public function sendReport($content, $email) { ... }
13 }
```

こっそりとインターフェイスを紛れ込ませたのに気がついましたか？その通り。私はインターフェイスを密かに滑りこませる奴なんですよ。

より頑丈なクラス

単一責任原則に従うことにより、クラスを頑丈にすることができます。唯一の関心ごとに焦点をあてることができるため、クラスの外の機能を変更により、壊してしまうようなことが少なくなります。

開放／閉鎖原則

SOLID の2番めの文字が表すのは"Open/Closed principle" です。この原則の内容は:

> 「ソフトウェアのエンティティー（クラス、モジュール、ファンクションなど）は、拡張に対して門戸を開き、変更に対して門戸を閉じるべきである。」

言い換えれば、一度コードを書いたら、バグ修正以外の理由で、決して変更してはいけないのです。もし機能追加が必要であれば、そのクラスを拡張する必要があります。

この原則はあなたに「どうやったら変更できるか？」を考えることを強要します。ここでも、経験が最高の教師です。以前の状況で使用してきた、共通のパターンを基礎として、ソフトウェアの設計方法を学んでいるのです。

あまりにも厳格になるな

この原則を厳格に固執すると、その結果、技術の使いすぎに陥ってしまうことに私は気が付きました。これが起きるのはプログラマーがクラスが利用される可能性全部を考える時に起きます。これを少し考えてみるのは、良いことです。しかし、あまりに考えすぎると、複雑にする必要がないクラスまで、余計に複雑にしてしまう結果になります。

もし、この原則の頑固な支持者の気分を害したら、ごめんなさい。けど、ねえ、私はただ、見たままを叫んでいるだけです。[29] これは原則で、法律ではありません。

そうだとしても、良くある例を学びましょう。あなたはアカウントの請求に取り組んでいて、特に払い戻し (refund) の処理の部分を担当しています。

```
1  class AccountRefundProcessor {
2    protected $repository;
3
4    public function __construct(AccountRepositoryInterface $repo)
5    {
6      $this->repository = $repo;
7    }
8    public function process()
9    {
10     foreach ($this->repository->getAllAccounts() as $account)
11     {
12       if ($account->isRefundDue())
13       {
14         $this->processSingleRefund($account);
15       }
16     }
17   }
18 }
```

[29] 原文は"I just calls 'em like I sees 'em" です。ラップのような表現です。（実際、ラップでも使われるフレーズです。）

では、上のコードを確認して行きましょう。アカウントのストレージはリポジトリークラスとして分離され、依存注入されています。いいですね。

不幸なことに、ある日仕事に来てみると、上司が怒っています。管理部門が大きな払い戻しは、担当者による見直しが必要だという、新しい規則を決めたのです。うわーー。

では、上のコードのどこが悪いのでしょう？これを変更しなくてはなりません。つまり、これは変更に対して門戸を閉じていません。これをサブクラスに拡張することはできますが、`process()`のコードのほとんどは重複してしまうでしょう。

ですから、あなたは払い戻し手順をリファクターするのです。

```
1  interface AccountRefundValidatorInterface {
2    public function isValid(Account $account);
3  }
4  class AccountRefundDueValidator implements AccountRefundValidatorInterface {
5    public function isValid(Account $account)
6    {
7      return ($account->balance > 0) ? true : false;
8    }
9  }
10 class AccountRefundReviewedValidator implements
11   AccountRefundValidatorInterface {
12   public function isValid(Account $account)
13   {
14     if ($account->balance > 1000)
15     {
16       return $account->hasBeenReviewed;
17     }
18     return true;
19   }
20 }
21 class AccountRefundProcessor {
22   protected $repository;
23   protected $validators;
24
25   public function __construct(AccountRepositoryInterface $repo,
26     array $validators)
27   {
28     $this->repository = $repo;
29     $this->validators = $validators;
30   }
31   public function process()
32   {
33     foreach ($this->repository->getAllAccounts() as $account)
34     {
35       $refundIsValid = true;
36       foreach ($this->validators as $validator)
```

```
37        {
38          $refundIsValid = ($refundIsValid and $validator->isValid($account));
39        }
40        if ($refundIsValid)
41        {
42          $this->processSingleRefund($account);
43        }
44      }
45    }
46  }
```

これで、AccountRefundProcess はコンストラクターでバリデーターの配列を受け取ります。次のビジネスルールの変更時には、新しいバリデーターをサッと取り出し、バリデーターの配列に追加、それであなたは平穏安泰に過ごせます。

リスコフ置換原則

SOLID の”L” は”Liskov substitution principle” を表します。この原理の内容は：

> 「コンピューターのプログラミングにおいて、S が T のサブタイプの場合、タイプ T のオブジェクトはそのプログラムの望ましい特性（正当性、動作など）を一切変更すること無くタイプ S のオブジェクトと置き換えることができる。

何だって？他の原則よりも更に混乱を引き起こす、この原則を堅牢な設計の一部だと考える冒険に、私は出たのです。

この原則は本当にすべて「代用可能性 (Substitutability)」に関することです。ですが、”Substitutability” の”S” を省略形として採用すると、SOLID の代わりに SOSID になります。こんな会話が想像できますか？

プログラマー1：「クラスを設計するなら、S.O.S.I.D. 原則に従わなくちゃ。」

プログラマー2：「ソー…セージ？」

プログラマー1：「いいや、SOSID。」

プログラマー2：「シーフード？」

プログラマー1：（Michael Feathers に電話をかける。）「ねえ、もっと良い省略形が必要なようだよ。」

単純に言えばリスコフ置換原則は、クラスを使用するところならどこでも、クラスのサブクラスを使用できるようにプログラムすることを意味しています。

言い換えれば、もし長方形クラスがあるなら、その長方形クラスと、派生した正方形クラスをプログラムで使用するのです。それから、長方形クラスを使用する場所であればどこでも、正方形クラスを使えるようにします。

まだ混乱していますか？私も最初に学んだ時とても混乱したことを覚えています。だって「はあー？当たり前だろう。」ですからね。今でも、まだ理解していないことがあるんじゃないか心配しているんです。こんなに当たり前のことを堅牢な設計の教義に、なぜ入れているんでしょう？

この原則が主張しているサブタイプの前提（タイプの継承）には微妙な側面があるのですが、強調すべきではないでしょう。そして事後条件（良い部分の変更なし）は弱めるべきでありません。他に何かあるのでしょうか？それとスーパータイプが投げるのと同じ（もしくは継承した）例外を投げるサブタイプは持てません。

おお！他に何かあるのでしょうか？リスコフの原則について、他の詳細をここではもう紹介しません。本当に時間を使いすぎたのではないかと心配しているんです。

リスコフの原則は厄介ではありません。

何ですって？

そうです！言った通りです。（技術的に原則は厄介ですから、私は嘘を付いたことになります。しかし、その言い訳をさせてください。）

PHP では、以下の3ガイドラインに従えば、リスコフの99%は守ったことになります。

１．インターフェイスを使用する

インターフェイスを使ってください。理にかなっているところであれば、どこにでも使ってください。インターフェイスを追加しても、オーバーヘッドはそんなに大きくありませんし、リスコフ置換原則に従っていることを保証する抽象化を自然なレベルで達成できる結果を手に入れることができます。

２．インターフェイスには実装の詳細を含めない

インターフェイスを使用するなら、実装の詳細を見せてはいけません。`UserAccountInterface` にストレージがどの程度残っているかという詳細を含める必要がありますか？

３．タイプをチェックするコードに気をつける

特定のタイプにだけ、別の操作を行うコードを書けば、リスコフ置換原則（たぶん、それと他の堅牢原則）を破ることになるでしょう。

以下のコードについて、考えてください。

```php
class PluginManager {
  protected $plugins; // プラグインの配列

  public function add(PluginInterface $plugin)
  {
    if ($plugin instanceof SessionHandler)
    {
      // ユーザーがログインしている場合のみ追加
      if ( ! Auth::check())
      {
        return;
      }
    }
    $this->plugins[] = $plugin;
  }
}
```

この小さいコードのどこが悪いのでしょうか？このタスクはオブジェクトタイプにより、異なった処理を行なっているため、リスコフを破っています。これは仕事をこなす精神に則り、小さなアプリケーションで、よく見られるコードです。これを紹介できて、とっても嬉しいのです。

しかしですね…ある `add()` メソッドが何を追加させるのかをなぜ選り好みさせる必要があるのでしょう。皆さんも考えてもらえば、'add()' は多少コントロールマニアであると思いませんか。一度深呼吸させ、コントロールをあきらめさせる必要があります。**(ああ。コントロールの逆転のことだな。)**

インターフェイス分離原則

SOLID の'I' が意味しているのは"Interface Segregation Principle" です。インターフェイス分離原則の内容は：

> 「クライアントは使用しないメソッドへ依存を強要されてはならない。」

普通の英語で表現するなら、太りすぎたインターフェイスを使用するなです。「単一責任原則」をインターフェイスに適用すれば、通常問題はありません。

私達がこの書籍で設計するアプリケーションは小さく、この原則は、多分さほど適用できません。インターフェイスの分離はアプリケーションがもっと大きくなった場合に重要になります。

この原則はドライバーで頻繁に破られます。キャッシュドライバーについて考えましょう。データーの値を一時的に保管することだけしない、本当に単純なキャッシュです。ですから、`save()` か `put()` メソッドが必要で、それに対応して `load()` か `get()` メソッドも必要です。しかし、キャッシュの設計者が頻繁にやってしまうのは、余計なおまけを付けてしまうことです。次のようにインターフェイスを設計します。

```
interface CacheInterface {
  public function put($key, $value, $expires);
  public function get($key);
  public function clear($key);
  public function clearAll();
  public function getLastAccess($key);
  public function getNumHits($key);
  public function callBobForSomeCash();
}
```

キャッシュを実装しようとしてみましょう。すると `getLastAccess()` は実装不可能だと気づきます。ストレージがサポートしていないからです。同様に `getNumHits()` も問題です。それから、`callBobForSomeCash`（お金についてボブに電話を掛ける）にはお手上げです。「Bobって誰だい？電話代は誰が払うんだい？」インターフェイスを実装するなら、ただ例外を投げることになるでしょう。

```
class StupidCache implements CacheInterface {
  public function put($key, $value, $expires) { ... }
  public function get($key) { ... }
  public function clear($key) { ... }
  public function clearAll() { ... }
  public function getLastAccess($key)
  {
    throw new BadMethodCallException('not implemented');
  }
  public function getNumHits($key)
  {
    throw new BadMethodCallException('not implemented');
  }
  public function callBobForSomeCache()
  {
    throw new BadMethodCallException('not implemented');
  }
}
```

美しくない。美しくないですよね？

これがインターフェイス分離原則のポイントとなります。代わりに以下のように小さなインターフェイスを作成しましょう。

```
interface CacheInterface {
  public function put($key, $value, $expires);
  public function get($key);
  public function clear($key);
  public function clearAll();
}
interface CacheTrackableInterface {
  public function getLastAccess($key);
  public function getNumHits($key);
}
interface CacheFromBobInterface {
  public function callBobForSomeCash();
}
```

論理的でしょう?

依存逆転原則

SOLID の最後の'D'は、"Dependency Inversion Principle" を意味します。これは2つの内容を含んでいます。

> A. ハイレベルのモジュールは、ローレベルのモジュールに依存してはならない。両モジュールとも抽象に依存しなくてはならない。
>
> B. 抽象は詳細に依存してはならない。詳細は抽象に依存しなくてはならない。

何かすごいことを意味しているのでしょうか?

ハイレベル
: ハイレベルコードはローレベルコードより複雑で、ローレベルコードの機能を利用します。

ローレベル
: ローレベルコードは基本的な部分を実行し、ファイルシステムへのアクセスや、データベース管理のような操作に集中します。

ローレベルとハイレベルの間には幅があります。例えば、セッション管理はローレベルと考えましょう。でもセッション管理は、さらに別のローレベルコードであるセッションストレージを利用します。

それぞれの関係に置いて、ハイレベルとローレベルがあると考えるほうが便利です。ユーザーのログイン処理のような、アプリケーションの特定の機能に比べれば、明らかにセッション管理はローレベルです。しかし、セッション管理はデータベースのアクセス層に比べればハイレベルです。

「そうか、依存逆転とは、制御の逆転を実装した場合なんだ。」とか、「いいや。依存注入が、依存逆転だよ。」と人々が話しているのを聞いたことがあります。両方の答えとも部分的には正解です。両方の答えとも、依存逆転を実装できるからです。

依存逆転原則はクラス依存の分離テクニックを良く表しています。分離することにより、ハイクラスロジックも、ローレベルオブジェクトも、お互いに直接関連しません。代わりに抽象と関係します。

PHP の世界であれば…依存逆転は何により上手く構築できるでしょうか?お考えの通り、インターフェイスです。

全設計原則が、どのようにいっしょに働くのか、興味がありませんか?それと、この原則が、最も使用されない PHP の構造であるインターフェイスをいかに頻繁に活用しているのか、知りたくありませんか?

> **依存逆転のシンプルなルール**
>
> ハイレベルコードで使用しているオブジェクトには **いつでも**インターフェイスを使用させる。そしてローレベルコードはそれらのインターフェイスを実装すれば、あなたはこの原則を捉えています。

一例

ユーザー認証は良い例になります。一番上位レベルのコードはユーザーを認証することです。

```
1  <?php
2  interface UserInterface {
3    public function getPassword();
4  }
5
6  interface UserAuthRepositoryInterface {
7    /**
8     * $username の UserInterface を返す
9     */
10   public function fetchByUsername($username);
11 }
12
13 class UserAuth {
14   protected $repository;
15
16   /**
17    * リポジトリーを依存注入する
18    */
19   public function __construct(UserAuthRepositoryInterface $repo)
20   {
21     $this->repository = $repo;
22   }
23
```

```
24    /**
25     * $username と $password が有効であれば true を返す
26     */
27    public function isValid($username, $password)
28    {
29      $user = $this->repository->fetchByUsername($username);
30      if ($user and $user->getPassword() == $password)
31      {
32        return true;
33      }
34      return false;
35    }
36  }
37  ?>
```

これで、ハイレベルクラスの `UserAuth` ができ、抽象である `UserAuthRepositoryInterface` と `UserInterface` に関連しています。では、とてもありふれた残り2つの抽象を実装しましょう。

```
1   <?php
2   class User extends Eloquent implements UserInterface {
3     public function getPassword()
4     {
5       return $this->password;
6     }
7   }
8   class EloquentUserRepository implements UserAuthRepositoryInterface {
9     public function fetchByUsername($username)
10    {
11      return User::where('username', '=', $username)->first();
12    }
13  }
14  ?>
```

簡単、簡単、超簡単。レモンを絞るぐらい簡単です。

では、`UserAuth` が使えるように、`EloquentUserRepository` で組み立てるか、自動で注入されるように `EloquentUserRepository` をインターフェイスに結合しましょう。

Laravel の `Auth` ファサードを使ったこの章の認証サンプルコードで、混乱しないでください。原則を示すためのサンプルに過ぎません。Laravel の実装も、これとよく似ていますが、もっと上手くやっています。

サンプルをお読み頂きありがとうございました。

お楽しみいただけたでしょうか。

Laravel4 でこなすプログラム術 Getting Stuff Donw[30]を購入いただき、残りの章からも何かを学んでいただけることを願っています。

[30] https://leanpub.com/gsd-laravel-jp