Learn how to write good code

Dion Moult

December 15, 2014

Original article at:
http://sevenstrokes.net/learn-how-write-good-code/.

I am writing this as a short starter guide to those interested in writing good code. Although I hope for it to stand as an article in its own right, it is by no means comprehensive. At the end, there is a further reading list for those interested.

Contents

1 Why does good code matter? 2

2 What is good code? 3

3 Understandable syntax 4

4 Understandable architecture 12

5 Good workflow 19

6 Code examples 24

7 Handling legacy projects 25

8 Further reading 26

1 Why does good code matter?

As a programmer, you have probably experienced bad code. Bad code tends to grow over time as your software becomes more complex, and causes three big problems:

1. It can be difficult to integrate new features as you are worried about compatibility with the existing system.
2. Time is spent fixing old code, which have developed mysterious bugs that are hard to reproduce and to fix.
3. You may have forgotten what a lot of the code does, and so you may be afraid that touching it will break the system.

These three problems will give any developer a headache, but it can cost the business much more! Slower development costs time and money, unpredictable integration periods make it harder to make sales promises, and quirky bugs can make your customers lose faith.

Bad code is also a very difficult problem to solve. This is because existing developers are afraid that changes will worsen the system, new developers take too long to train, and every day the system accumulates more bad code. The accumulation of bad code accelerates the three big problems stated above, until there is no option but to heavily refactor or start from scratch. The business is held hostage by the state of the code, and often we just give up and decide to start from scratch.

Although starting from scratch is an attractive option, it is expensive, difficult to manage alongside your legacy version for customers, and is merely a short-term solution. Give it a few more years, and you'll notice bad code creeping in yet again, and the cycle will repeat itself.

Thankfully, there is a way to prevent bad code. This article is going to show you the principles of writing code in a clean, maintainable, and understandable manner that'll make you happier as a developer, and save your company time and money.

2 What is good code?

Good code is understandable code. End of story. Understandable code is easier to maintain, easier for new developers to learn, easier to debug, faster to find what you're looking for, and most importantly: helps that the programmers writing the code actually understand business objectives.

Although simple to describe, understandable code is hard to write. This is because there are three diverse aspects of how to write understandable code. Only by implementing all three aspects simultaneously will you be able to write understandable code. These three aspects are:

Syntax These are the concrete details of how to write each line of code in an understandable manner. This helps solve day-to-day frustrations when reading code. Syntax covers topics such as variable naming, code conventions, and refactoring strategies.

Architecture These are the abstract concepts of how to organise large portions of code. This helps improve system flexibility. Architecture includes topics such as design patterns, file structures, and dependency flow.

Workflow These are the managerial techniques of how to coordinate within a team, deliver consistently, and ensure that code is aligned with business interests. Workflow includes topics such as collaboration tools, TDD/BDD, and automation.

These three aspects are perhaps best approached in that order, and get progressively more complex.

Let's get started.

3 Understandable syntax

Good syntax is understandable syntax. Understandable syntax normally doesn't look like code. Code is cryptic and unnatural to follow. Instead, understandable syntax should look like English. English is easier to read, is easier to talk and reason verbally, and most importantly, it can encourage the developer to understand business requirements.

If your code is starting to look less cryptic and more like English, that's a good sign. I will now run through a list of bite-sized tips which will make it easier to write code like English. Each tip will be accompanied by a short code example showing the tip in action, like below:

```
if ($user->has_registered())
    return $user->registration_code;
else
    throw new Exception\Authorisation;
```

Stick to a formatting convention

Conventions are good. It doesn't matter what you pick, as long as you stick to them. Standardised code is easier to read and skim over. If you can't tell who wrote the code, you know you're doing it right.

```
class Formatter
{
    public function format_code($code)
    {
        $this->remove_camel_case($code);
        $this->replace_tabs_with_spaces($code);
        // ...
        return $code;
    }
}
```

Avoid writing comments

Comments in general are bad. If your code needs a comment to explain it, you've probably written confusing code. Comments get outdated easily, and their truthfulness cannot be validated. Even worse are comments for the sake of comments which just add clutter to the code. A much better solution is to refactor the code until it is more understandable.

If you absolutely have to write a comment, write about "why" rather than "what".

```
$emailer->send($html_message);
```

... is better than ...

```
/**
 * Send an email
 *
 * A long time ago in a galaxy far far away...
 */
$em->process_msg($str);
```

Prefix boolean variables

If you have a boolean, prefix it with `has_`, `is_`, or `should_` so that it is obvious that it is a boolean.

```
$has_registered = TRUE;
$is_valid = FALSE;
$should_display_profile = TRUE;
```

Name numeric variables in a way that suggests they are numeric

It is also possible to connote numeric variables through good naming:

```
$total_comments = 50;
$elapsed_seconds = 42;
$unix_timestamp = time();
```

Use plural names to name lists

Taking care to use singular and plural words helps connote multiple items:

```
$new_posts = array(...);
$post->message = '...';
$post->comments = array(...);
```

Use consistent terminology

Just like a legal document, things are clearer if certain words are chosen specifically and given clear definitions. If you have a `$photo`, don't refer to it as an `$image` later on. Use words that are unambiguous.

```
$photo->file_path;
$directory->total_files;
$avatar_url;
```

Avoid meaningless words as names

Certain phrases are a bit useless, like `process_data`, or `x`. Try to describe what they actually are. Iterator variables are an exception.

```
$article->calculate_human_time_duration();
$article->format_comments();
$article->remove_swear_words();
```

Make sure names are pronounceable

Pronounceable names can be talked about easily. Don't abbreviate unnecessarily.

```
$message;
$handler;
$text;
```

Name according to type of object

Names can differentiate between names of system entities and names of libraries. Here are some system entities:

```
$user;
$photo;
$message;
```

Here are some libraries:

```
$validator->validate();
$emailer->send();
$photoshopper->crop();
```

Function names should always begin with a verb

Functions should always be verbs, or verb-noun combinations.

```
get_message();
download_image();
$user->authorise();
```

No getters and setters for the sake of it

Just writing getters and setters out of habit? See if you really need it.

```
$api = new Api($oauth_token);
$api->oauth_token;
```

Refactor into functions as soon as you can

The smallest unit of code organisation is a function. Once you've written a few lines of code that do a task, group that into a single function. Don't delay. Functions are the first step to writing code that looks like English. If a function is doing more than one task, your function should be split up into more functions. Keep on creating functions until you can't create any more.

```
interface Validator
{
    public function setup(array $data);
    public function set_not_empty_rule($key);
    public function set_email_rule($key);
    public function is_valid();
    public function get_error_keys();
}
```

Make function names connote return type

Functions generally either process or retrieve things. If they are doing both, they are doing too many things. What functions retrieve can be described by the function name.

```
$user->create(); // NULL
$user->get_username(); // String
$user->get_total_likes(); // Integer
```

```php
$user->get_posts(); // Array
$user->has_validated(); // Boolean
$article->get_user(); // User
```

Make function parameter orders obvious

Try to make it obvious what the parameters and their order is from the function name. Also keep order consistent, so if you usually ask for `$user_id` as the first parameter, keep it as the first parameter in future function definitions.

```php
$database->update_user_email($user_id, $email);
$database->update_user_first_name($user_id, $first_name);
$user->get_name_and_address() // return array($name, $address);
```

Minimise function parameters

More than four function parameters may be a sign of an overly complex function. Instead consider having an array or object as a parameter. Especially avoid using multiple booleans in function parameters, as `$library->get_books('id', TRUE, FALSE, FALSE, TRUE)` is incredibly confusing.

```php
$library->get_books($filter_options);
$library->get_books(array($this, 'is_new_book'));
```

Avoid negative boolean functions

Don't name a function `is_not_foo`, because using a "not" operator results in a double negative. Double negatives get very hard to understand especially in complex if statements.

```php
if ( ! ($this->is_registered() AND $this->is_verified())
    OR ! $this->is_guest())
    return;
```

Make sure function code does what the function says it does

If you're reading a function named `get_user`, and you see it also logging a user in and registering that user, that function is clearly doing much more than getting a user. Split that function into smaller more specific functions.

```
if ( ! $user->exists())
    return $user->generate();
```

Try to name classes after meaningful business objects

We all know that classes are best named as nouns. However also keep in the back of your mind that the OOP paradigm was aimed at better representing the real-world. So name your classes after actual real things in your business, rather than abstract programming concepts.

```
$invoice = new Invoice;
$recipient = new Recipient;
$registrant = new Registrant;
```

Refactoring into classes

A class is technically a group of functions. But which functions do you put in which classes? A class contains functions that share state. If you have two groups of states, then you probably have two classes.

```
class Formatter { ... }
class Emailer { ... }
class Validator { ... }
```

Order functions according to how they are used

If a series of functions are used in a particular order, define them in that order so that those reading the definitions find it easier to understand. For example, you wouldn't define the constructor at the end, right? Also, keep public API functions at the top of the code, and leave private details at the bottom.

```
interface Emailer
{
    public function setup(Transport $transport);
    public function set_to_emails(array $to_emails);
    public function set_from_emails(array $from_emails);
    public function queue();
    public function send();
}
```

Put the most important stuff at the top of the file

If your class provides an API, or a public execute function, or important state definitions, keep these at the top of the file. Reduce unnecessary hunting for important information.

```
app.widget.tabs = function() {
    "use strict";

    var api = {
        name: null,
        init: init,
        changeTab: changeTab
        refreshTab: refreshTab
    };

    function init() { ... }
    function changeTab() { ... }
    function refreshTab() { ... }

    return api;
}();
```

Keep code dry

DRY stands for Don't Repeat Yourself. This is the number one rule when refactoring. If you find copy pasted or even dead code lying around, be sure to isolate it in a reusable function.

```
def print_invoice(self):
    self.net_total = self.convert_cents_to_dollars(self.net_total)
    self.tax = self.convert_cents_to_dollars(self.tax)
```

Limit language tricks

Avoid using obscure language tricks that only those familiar with the language would understand. Idiomatic code is fine, but quirks that may cause people to have to refer to the documentation is bad.

```
b, a = a, b
```

Using OOP strategically

OOP is great for controlling what dependencies are used by what classes, accurately representing business objects, and making code read naturally like English. When writing OOP, read it out loud to see if it makes sense, and if it's something you could show to a non-programmer to explain what's going on.

```
$registrant = new Registrant(new $user);
$registrant->validate_registration_form();
$registrant->send_verification_message();
$registrant->register();
```

Code summaries

If your code contains complex logic, see if you can summarise it in a single function. This'll ideally contain a short poem of code that acts to summarise the complexities of what is happening. Those interested in the details can code dive as necessary.

```
def run(self, request, response, api):
    if (self.has_access_token())
        secret = api.get_secret(request.access_token)
        api.setup(secret)
        return response.redirect(api.get_customer_url())
    else
        return response.redirect(api.get_authorisation_url())
```

4 Understandable architecture

Just as good syntax is understandable syntax, good architecture makes it easy to understand what the system is used for. For example, an online booking system is used for things like adding bookings, removing bookings, and checking for available seats. These individual scenarios that describe what a system is used for are known as usecases.

Achieving usecases is vital to the success of the application. Equally vital is the ability to **keep the system flexible to adapt to changing usecases**. We will now briefly walk through a series of abstract concepts on code organisation to help achieve these two needs.

What is a usecase?

A usecase is a single scenario of how your system may be used. For example, "Register user", "Add to cart", or "Purchase order". A usecase consists of:

1. A descriptive name
2. Input data that the scenario requires
3. A step-by-step description of what happens
4. Output data that the scenario will provide
5. Any possible exceptional circumstances that may occur

Let's take a simple "Register user" usecase as an example.

```
Name: Register user
Input: $user->username, $user->password

Interactions:

 1. The user username and password is validated
 2. The user is saved

Output: $user->id
Exceptions: Validation error
```

This usecase, once written in code, becomes the single abstract go-to point of what happens in a user registration scenario. Usecases should form the core of your application, and contain only business related logic, rather than technical logic.

Common design patterns useful for usecase modules include the interactor pattern, facade pattern, and Data-Context-Interactions.

What is not a usecase?

Anything which is *not* an abstract description of a usage scenario is not a usecase.

Notice that the usecase above only uses abstract business terminology. The usecase makes no mention of using a database, or whether data is input via a web browser or desktop application. If it were sending an email, it wouldn't mention the mail transport, and if it were processing an image, it wouldn't mention the image processing libraries used. These are all technical implementation details, and should be separate from the usecase.

This separation of the abstract, object-oriented code that describes usecases and the concrete technical details allows you to ship the usecase code in one module, and have each implementation detail as separate modules which plug into the usecase module, usually using interface contracts.

Separating what usecases are and how they are implemented offers you a great amount of flexibility. Imagine if you now had to change your current project's database from MySQL to another, or had to suddenly develop an API for a web app, or switch web servers. Because your database, interface and other details are merely plugins to your usecase module, they are easy to swap out.

These non-usecase plugins can be split into four categories:

Data plain old simple definitions of different types of data used by the system, such as a user, a category, or an account.

Interfaces how your application is consumed, such as through a web interface, a desktop GUI, or a CLI app.

Repositories things which allow you to store and retrieve data, such as databases, flat files, and memory.

Tools libraries that process technical data, such as email, encryption, and image manipulation.

Each category as a minimum should be split up into a separate module. We will talk briefly about each category, then move on to more general organisation concepts.

What are data?

Data are simple objects or data types that describe the various types of data used by the system. For example, a user, which might have attributes for username, password, and email. These data types are usually heavily used by usecases, and form a comprehensive dictionary of all the system data.

These may also include any other data structures you might like to use for passing between usecase and non-usecase modules. These include request and response data structures, which have varying input and output attributes.

These are extremely simple objects which minimal to no logic. They are purely defining objects with no restrictions or knowledge about their use.

```php
class User
{
    public $id;
    public $username;
    public $password;
    public $email;
}
```

What are interfaces?

Your application interface, not to be confused with code interface contracts, is how your application is delivered and consumed by the end user. This could be a web page served by a web server, a custom XML RPC protocol, desktop GUI, or CLI.

This code generally deals with listening for input requests, parsing request data, routing to the correct application module to execute, rendering a response, and deciding the application flow. These different responsibilities suggest further sub modules of code.

The interface itself doesn't know how the usecase works, but merely decides which to execute, supplies input to each usecase, and reacts to the output.

```
Parse request data -> Route to usecase -> Execute usecase -> Generate response
```

Common design patterns for interfaces include Model-View-Controller, Model-View-Presenter, and Controller-View-Template.

What are repositories?

Repositories contain code which store and retrieve data. This is usually thought to be database queries, but is not always the case. Data can be stored in files for file uploads or flat file systems, in memory for session data, or even to a third party API, perhaps for CDNs or banking details.

Repositories are very simple code. They usually contain minimal to no logic, don't worry about formatting for output, or do any fancy processing of any kind.

```php
namespace Repository\User;

interface Register
{
    public function does_username_exist($username);
    public function save_user($username, $password, $email);
    public function get_saved_user_id();
}
```

Common design patterns for repositories include the gateway pattern, and adaptor pattern.

What are tools?

Tools are other miscellaneous libraries that your application may need. Whereas most applications have an interface and need a repository, what tools are needed are different for each application. For example, one application might need image processing, and another might not.

However, tools still contain no business logic, and so this means they are normally reusable across applications. Often tools can be taken directly from third party libraries. In fact, many frameworks already include many tools for common tasks.

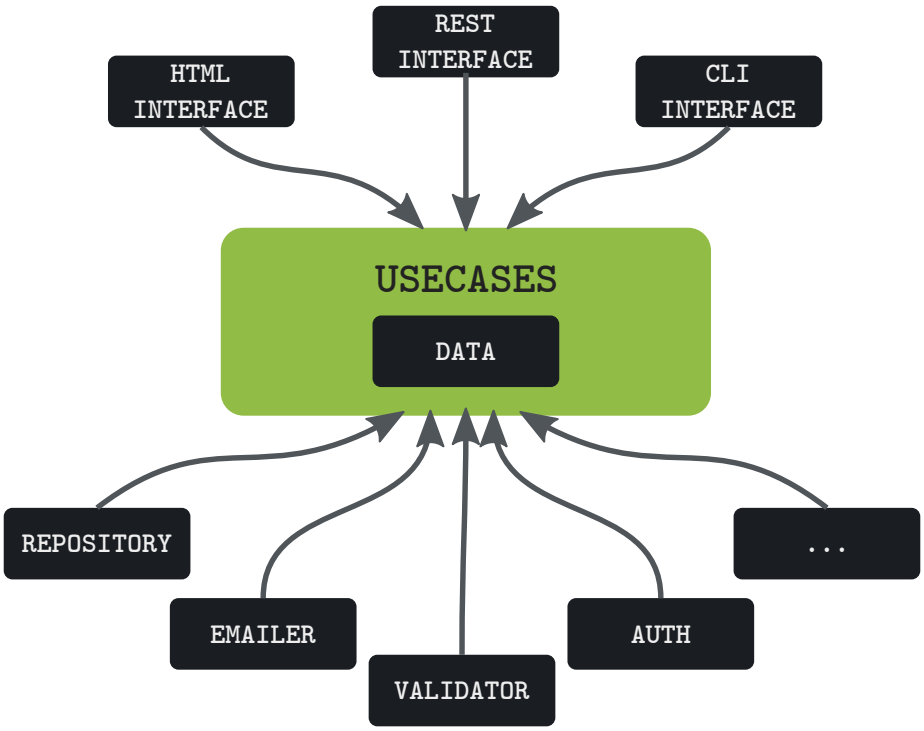
```
namespace Tools;

interface Authenticator
{
    public function authenticate($id);
    public function deauthenticate($id);
    public function get_authenticated_id();
}
```

Common design patterns for tools include the adaptor pattern, factories, and singletons.

How does this look altogether?

As a result, your application should center around *usecases*, which use a well defined dictionary of *data*. Every single other technical implementation detail turns into *plugins* to your usecases. Here's a diagram showing that:



Now, let's talk about some generic concepts of code organisation.

What are design patterns?

Design patterns are solutions of code organisation to common organisation problems. Design patterns also give developers a common vocabulary, so that developers can quickly understand the structures of foreign systems. For example, MVC tackles interface organisation, and suggests that splitting interface markup from application logic is beneficial to maintenance.

Design patterns target general design problems, and also offer general design solutions. Because they are general, you need to understand them fully before using it in your scenario. Sometimes, it is better not to use any common pattern at all, if your code has evolved naturally otherwise. There are three things you need to know:

1. **Do not use patterns blindly.** You need to know the history of a design pattern before deciding whether it is appropriate to use.
2. **There is no one-size-fits all design pattern.** Different parts of your code need different design solutions.
3. **Don't be afraid to deviate from a design pattern.** As long as you understand the fundamentals of understandable architectures, letting a system evolve into its own pattern is not only normal, it is recommended.

The best test to see if a design pattern that has emerged is effective is to try and maintain your code. If maintaining and adapting features is a breeze, that's a good sign.

What is KISS?

KISS stands for Keep It Simple, Stupid! Over-engineering code is just as bad as under-engineering it. Another acronym is YAGNI - You Ain't Gonna Need it! Predicting common engineering separations such as interface and logic is OK, but don't over-think it. The solution is to start with the simplest solution possible, and then let the system naturally evolve of its own accord, of course following these architectural concepts along the way.

Avoiding jargon

Don't be too attached to jargon. Design terminologies are useful, but can be unintuitive. Use natural terms that are appropriate to your system.

Stability and instability

Code should be separated in terms of volatility. Separate code that changes frequently from code that changes rarely.

Single Responsibility Principle

The SRP states that each class should have one, and only one responsibility. This means that code that changes for different reasons, or may be changed by different people, should belong in different classes. For example, UI code is changed for aesthetic reasons by frontend developers, and so should be separate from application logic, which changes for business reasons by backend developers.

Open Closed Principle

The OCP states that code modules should be open to extension, but closed to modification. This means that to add a new feature, you should be able to do it simply by writing new code that conforms to abstract interface contracts instead of modifying existing code. This makes adding new features much more straightforward, and reduces the potentially negative impact of them. To do this, abstractions that allow for a plugin interface should be made each time we think a customer will need a lot of changes done on a module.

Liskov Substitution Principle

The LSP states that objects may be replaced by their subtypes without breaking any intended behaviour. This means that when you extend classes, make sure that the extension is a valid one and doesn't have any quirks.

Interface Segregation Principle

The ISP states that no class should depend on methods that it does not use. This means that that a class depending on another interface should need all the functions available in that interface and not less. This helps prevent against god models and encourages small, client-specific interfaces.

Dependency Inversion Principle

The DIP states that a class should only depend on abstractions, and not on concrete details. For example, a class in an invoicing system that describes its abstract behavior should not depend on a specific PDF rendering library. This allows high level concepts to stay clean of technical details prone to change.

What is dependency injection?

Dependency injection allows a target class to have other classes plugged into it. The target class does not hardcode the other classes they depend on. These other classes can then be easily swapped out as a plugin. These classes can be injected via constructors, setter methods, or even an entire other class.

Structure files to expose system intent

One of the first steps to searching for the code we want to change is to look at the directory and file names. Naming them logically to expose the most important concepts of the system makes the source much easier to navigate. Similarly, try to structure files such that more important ones are higher up the file hierarchy, and thus easier to discover.

For example, if you see file paths like "Usecase/User/Suspend", it's pretty clear that users can be suspended. A file path like "Modules/Controllers/User" is much less useful.

Many small architectures

A large system is a difficult system to manage. There is no easy solution to how to architect a large system. Instead, the objective should be to split the system up into as many smaller manageable, isolated modules as possible. Don't be afraid of using many different tailored architectures instead of a single overarching one.

5 Good workflow

Good workflow is when developers are using a development environment and development processes that help them understand business requirements, and then ensure the code they produce help achieve a business requirement. First, we will talk about the environment, and then talk about processes.

Good development environments

A development environment consists of the software tools that developers use to manage their workflow. The following software is a minimum, not a comprehensive list.

Version control

Version Control Systems, such as Git, Mercurcial, SVN, or CVS are software that keeps track of a log of who changed which lines of code, when the change was made, and why the change was made. Benefits include:

- Easier to collaborate with multiple developers as the system can help merge the work of two separate developers working simultaneously.
- Acts of a backup of the code you delete
- Helps you switch between working on multiple features, as the VCS can keep track of where you were
- Similar to working on multiple features, allows you to work on different versions of the same software. So you can do experimental work in an experimental version while slowly moving code into a more stable version for launch.
- Allows you to find out why a piece of code was written, if you ever get confused.
- Allows you to find out who wrote a piece of code, if you want to ask them why it's written that way
- Allows you to review your changes periodically to make sure that you're not accidentally leaving in unnecessary code
- Makes it easier to "atomically" group code per feature
- Helps when doing code reviews as it offers a structured view of the changed code
- Makes it easier to make snapshots of the code at specific points of time
- Makes it easier to collaborate with new developers joining the project

- May provide access controls to control who can edit which parts of the codebase
- Because deleted code is recoverable, you don't need to leave stray commented code lying around
- Because author information is recorded, you don't need excessive boilerplate credits
- Can generate useful statistics such as developer activity, and which parts of the code are more volatile
- Can be integrated into other code software

Issue tracking

Issue tracking ensures:

- Developers know what tasks need to be done
- You know who is working on what
- You are aware of progress towards milestones
- It highlights things that are taking longer than they should
- It focuses conversations on specific issues

Continuous integration

Anything that can be automated, should. This includes building the software, deploying the software, and testing the software. If you are unable to do any of the above in a single step, there is a problem.

The primary benefit is continuous integration, which allows for a much more rapid build and deploy cycle, meaning more time discussing changing usecases with the customer.

Other software

Although not necessary for all projects, you may consider:

- Isolated development environments for development, staging and production
- Statistic generators
- Code quality analysers
- Package and/or dependency managers

Good coding processes

Coding processes are disciplines that developers practice as they write code.

Test-Driven-Development and Behaviour-Driven-Development

A test is a short snippet of code that tests the behavior of another short snippet of code. For example, if a function is called `is_negative_number`, the test will check that `is_negative_number(-1)` returns `TRUE` and `is_negative_number (1)` returns `FALSE`.

Tests are a form of quality control, and are traditionally written after the production code - the code you want to test - has been written.

Test-Driven Development is a (initially unintuitive) practice of writing tests before writing any functioning code. Specifically, there are three rules to follow:

1. You are not allowed to write any production code unless it is to make a failing unit test pass.
2. You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.
3. You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

These three rules produce short iterative cycles of development that:

- Ensure the developer is always asking themselves, what is this code meant to do?
- Ensure that every line of code a developer writes is covered by a verifiable unit test
- Reduce debugging time, as errors are limited to short periods of time
- Reduce the fear of large system changes, as breakages are immediately detectable and quantifiable
- Create reliable low level documentation of how the code is used and should behave
- Naturally evolve decent architectures, as the testing process ensures code modularity

Behavior-Driven Development is a derivative of Test-Driven Development, which accomplishes the same thing but uses English-like syntax. It may help encourage developers to keep their mindset aligned with business requirements.

SpecBDD and StoryBDD

BDD further specialises into two forms of syntax: SpecBDD and StoryBDD. Here's an example of SpecBDD syntax:

```
$this->get_user_id()->shouldReturn('id');
$this->shouldThrow('Exception\Authorisation')->duringAuthorise();
```

Here is an example testing the same code using StoryBDD syntax:

```
When I ask for a user ID
Then I should receive 'id'
When I try to authorise
Then there should be an authorisation exception
```

Whereas SpecBDD focuses on specific function calls of the class you are testing and offers assertions such as `shouldBe` or `shouldNotEqual`, StoryBDD allows you to invent your own sentences to describe a scenario following a simple structure of `Given [this assumption], When [I do this], Then [this should happen]`.

SpecBDD and StoryBDD are also technically interchangeable, but it is usually more natural to use SpecBDD to test small units of code as unit tests, and use StoryBDD to test the full stack in a short scenario as acceptance tests.

Be agile! Release early, release often

Delivering working software, adapting it often, and keeping close to the customer is vital. Being agile is easily summarised by these 12 principles:

1. Customer satisfaction by rapid delivery of useful software
2. Welcome changing requirements, even late in development
3. Working software is delivered frequently (weeks rather than months)
4. Close, daily cooperation between business people and developers
5. Projects are built around motivated individuals, who should be trusted
6. Face-to-face conversation is the best form of communication (co-location)
7. Working software is the principal measure of progress
8. Sustainable development, able to maintain a constant pace
9. Continuous attention to technical excellence and good design

10. Simplicity — the art of maximizing the amount of work not done — is essential
11. Self-organizing teams
12. Regular adaptation to changing circumstances

Other processes

Although not vital, you may consider:

- Semantic versioning
- Code reviews
- Pair programming

6 Code examples

Show me the code!

Unfortunately, I have not yet made most of my current code repositories public.

TODO.

7 Handling legacy projects

Great, so you're convinced by the benefits of understandable code and want to start writing some right away. Unfortunately, you're stuck with an old project without tests, and cryptic code. How do you deal with it?

There is a simple solution: *every time you make a commit, make sure you are leaving the code cleaner than when you found it.* If every single commit results in a minute improvement, such as a refactoring or even simple syntax cleanup, over time even the most messy of software would fix itself. There is no need for an expensive rewrite, just incremental improvement.

If you feel that your code is too fragile even for incremental improvement, at least enforce a policy that every new bug fix or feature implementation is accompanied by a test. Over time, you will develop a comprehensive test suite. This test suite allows you to reliably detect and quantify breakages. This will give you the confidence to refactor mercilessly when the time is right, as you can be assured of the impacts of the refactor.

8 Further reading

None of the concepts I have talked about are original. They are practiced and proven concepts by much smarter people. It is worth noting that I have changed many of the terminology of the original concepts for the benefit of simplicity and explanation.

For example, Data is usually called Entities, Tools known as Helpers or Collaborators, Interfaces are known as delivery mechanisms, functions doing what they say they do is usually named misplaced responsibility, repositories usually come hand in hand with gateways, and so on.

Many of these concepts have originated from and heavily inspired by Charles Simonyi, Robert Martin, and Trygve Reenskaug.

Many thanks to Paul Schwarz for proof-reading.

Please visit the original article online for the up to date list of further reading.
`http://sevenstrokes.net/learn-how-write-good-code/`.