## **Что такое чистый код и почему это важно**

К условно «чистому» можно отнести код, обладающий такими свойствами:

* легко читается и помогает быстро вникнуть новым разработчикам;
* оформлен в соответствии со стандартами, принятыми в сообществе;
* не создает проблем при расширении или изменении;
* имеет предсказуемое поведение.

Для чего вообще писать код по каким-то правилам, ведь «работает и так»? Представьте команду разработчиков, которые начинают с нуля довольно простой проект. В первые месяцы все прекрасно, но бизнес-задачи периодически меняются, и команде все сложнее вносить изменения. В какой-то момент задача по созданию простой кнопки может затянуться на часы или даже дни. Причиной этого может быть низкое качество кода, который просто не готов к изменениям.

Вообще, понятие «чистый код» очень субъективно и сложно измеримо, но все же есть некоторые простые правила, которые помогут сделать код более читаемым, гибким и поддерживаемым.

## **Основные требования к чистому коду**

Все описанные ниже примеры максимально упрощены и многие детали опущены для лучшего понимания. Можно считать это некоторым подобием псевдокода.

### **Содержательные имена**

Выбирайте имена переменным, функциям, классам так, чтобы это имя достаточно точно объясняло, что делает этот код и для чего он создан.

*// Bad*

const DMYDATE = “d.m.Y”;

*// Good*

const PUBLIC\_DATE\_FORMAT = “d.m.Y”;

### **Функции/методы**

Функции и методы должны выполнять только одну операцию и быть предельно короткими. Функции не должны содержать вложенных структур, так как это приводит к их увеличению.

*// Bad*

public function notify(array $usersId): void

{

$users = DB::table(‘users’)->whereIn(‘id’, $usersId)->get();

foreach ($users as $user) {

notify($user);

}

}

*// Good*

public function getUsers(array $usersId): Collection

{

return DB::table(‘users’)->whereIn(‘id’, $usersId)->get();

}

public function notify(Collection $users): void

{

foreach ($users as $user) {

notify($user);

}

}

### **Блоки и отступы**

Блоки в командах if, else, while должны состоять из одной строки, в которой обычно содержится вызов функции. Максимальный уровень отступов в функции не должен превышать один-два. Это упрощает ее чтение и понимание.

По возможности избавьтесь от блока else, если используете if. Иногда полезно следовать от отрицания if (! $var) — таким образом может сократиться количество вложенных if блоков.

*// Bad*

$user = DB::table(‘users’)->find($id);

if ($user) {

$post = $user->post()->first();

if ($post) {

return $post->created\_at;

} else {

throw new ModelNotFoundException();

}

} else {

throw new ModelNotFoundException();

}

*// Good*

$user = DB::table(‘users’)->find($id);

if (! $user) {

throw new ModelNotFoundException();

}

$post = $user->post()->first();

if (! $post) {

throw new ModelNotFoundException();

}

return $post->created\_at;

*// Best*

$user = DB::table(‘users’)->findOrFail($id);

$post = $user->post()->first();

if (! $post) {

throw new ModelNotFoundException();

}

return $post->created\_at;

### **Один уровень абстракции на функцию**

Постарайтесь скрывать второстепенные подробности в ваших функциях/методах. Не стоит смешивать уровни абстракции в функциях — это всегда делает код запутанней.

*// Bad*

function saveFile(Request $request)

{

file\_put\_content(‘someFileName’, $request->file(‘file’)->body);

$file = new File;

$file->body = $request->file(‘’)->body;

$file->save();

}

*// Good*

class Storage {

public function store(string $name, string $body) {

file\_put\_content($name, $body);

}

}

function saveFile(Request $request)

{

(new Storage)->store(‘someFileName’, $request->file(‘file’)->body);

$file = new File;

$file->body = $request->file(‘file’)->body;

$file->save();

}

### **Чтение кода сверху вниз**

За каждой следующей функцией должны следовать функции, вызванные выше. Таким образом мы можем читать наш код последовательно, как рассказ. Дядюшка Боб (Роберт Мартин — инженер, автор книги «Чистый код») называет такой подход «правилом понижения».

*// Bad*

function isAvailablePost(int $id): bool

{

return DB::table(‘posts’)

->where(‘id’, $id)

->where(‘status’, ‘active’)

->exists();

}

function getPost(int $id)

{

return DB::table(‘posts’)->find($id);

}

function update(Request $request)

{

if (isAvailablePost($request->get(‘post\_id’))) {

$post = getPost($request);

$post->update([‘title’ => ‘Some new title’]);

}

}

*// Good*

function update(Request $request)

{

if (isAvailablePost($request->get(‘post\_id’))) {

$post = getPost($request);

$post->update([‘title’ => ‘Some new title’]);

}

}

function isAvailablePost(int $id): bool

{

return DB::table(‘posts’)

->where(‘id’, $id)

->where(‘status’, ‘active’)

->exists();

}

function getPost(int $id)

{

return DB::table(‘posts’)->find($id);

}

### **Команды switch**

С оператором switch связано много спорных решений. Функция с ним по определению не может выполнять одну операцию, даже если switch содержит всего пару условий. Если обойтись без switch не получается, то можно опустить его в низкоуровневую логику приложения.

### **Аргументы функций**

Функции и методы должны содержать минимальное количество аргументов, которых в идеале не должно быть вообще. Для использования функции с большим количеством аргументов должна быть очень веская причина.

### **Объекты как аргументы**

Если количество аргументов функции/метода превышает 2-3, то стоит задуматься об объединении некоторых аргументов в отдельную абстракцию или класс.

*// Bad*

function sendNotification(string $userName, string $email, string $message);

*// Good*

function sendNotification(User $user, string $message);

### **Использование аргументов-флагов**

Аргументы-флаги могут приводить к некоторой путанице в коде, поэтому их лучше вовсе не использовать. Такие аргументы усложняют сигнатуру метода и говорят о том, что функция выполняет более одной операции. Помните, при истинном значении флага выполняется одна операция, а при ложном — другая.

*// Bad*

public function context(Request $request): void

{

$someFlag = $request->get(‘someParam’);

$this->someProcess($request, $someFlag);

}

private function someProcess(string $someString, bool $flag)

{

if (! $flag) {

doSomeStuff();

}

doSomeAnotherStuff();

}

*// Good*

public function context(Request $request): void

{

$someFlag = $request->get(‘someParam’);

if (! $someFlag) {

$this->someProcess($request);

}

$this->someAnotherProcess($request);

}

### **Избавьтесь от побочных эффектов**

Не стоит обманывать себя и других разработчиков, работающих с вашим кодом. Функции/методы не должны делать того, для чего не предназначены, исходя из их названия.

*// Bad*

function getPost(int $id)

{

$post = DB::table(‘posts’)->find($id);

$post->views += 1;

$post->save();

return $post;

}

*// Good*

function getPost(int $id)

{

return DB::table(‘posts’)->find($id);

}

### **Изолируйте блоки try/catch**

По возможности старайтесь изолировать try/catch в отдельной функции/методе. Иначе вы создаете запутанность в вашем коде, смешивая нормальную обработку с обработкой ошибок.

### **Плохие комментарии**

Хорошо написанный код не нуждается ни в каких дополнительных комментариях. Если у вас есть комментарии, которые описывают, как работает ваш метод/функция, или оправдывают путаницу, то стоит задуматься о переработке кода.

### **Обязательные комментарии**

Несмотря на существующие правила, не стоит писать для каждой функции или переменной комментарий PHPDoc. Иначе вы просто дублируете описание. В PHP 7+ есть все необходимые конструкции языка для того, чтоб избавиться от таких «обязательных» комментариев. Их точно стоит писать, только когда вы разрабатываете API.

### 

### **Закон Деметры**

Если модуль «А» знает о модуле «B», а модуль «B» знает о модуле «С», то модуль «А» не должен знать про модуль «С». Также объекты/модули никак не должны раскрывать свое внутреннее устройство.

*// Bad*

class Author {

private Post $post;

...

public function (Image $image)

{

$this->post->image->setUrl($image->getUrl());

}

}

*// Good*

class Post {

private Image $image;

...

public function setImage(Image $image)

{

$this->image->setUrl($image->getUrl());

}

}

class Author {

private Post $post;

...

public function (Image $image)

{

$this->post->setImage($image);

}

}

### **Не связывайтесь с null**

По возможности избегайте использование null в вашей бизнес-логике. Это создает лишнюю работу и проблемы на вызывающей стороне. Вместо кучи проверок на null лучше бросить исключение. В крайнем случае опускайте эту переменную на низкий уровень абстракции.

Если у вас возникает соблазн вернуть null из метода, рассмотрите возможность создания исключения или возврата объекта специального случая. Special case — подкласс, который обеспечивает особое поведение для конкретных случаев, известных как Null Object. Помните, возврат null из метода — это плохо, но передача null в метод еще хуже.

*// Bad*

function findUser(int $id): ?User

{

return User::find($id);

}

*// Good*

*/\*\**

*\* @param int $id*

*\* @throws ModelNotFoundException*

*\*/*

function findUser(int $id): User

{

$user = User::find($id);

if (! $user) {

throw new ModelNotFoundException;

}

return $user;

}

*// Best*

class UnknownUser extends User {

public function getName()

{

return ‘Some default name’;

}

}

function findUser(int $id): User

{

$user = User::find($id);

if (! $user) {

return new UnknownUser;

}

return $user;

}

Описанные принципы — это малая часть тех подходов, которые стоит освоить каждому разработчику. Но даже они помогут писать более стабильный, расширяемый и понятный код. Очень рекомендую ознакомиться как минимум с первыми двумя источниками из этой статьи.

Уверен, многие разработчики сочтут эти правила неким ограничением. Следовать им очень желательно, но это скорее рекомендации, чем ультиматум. Желаю всем чистого кода.

## **Полезная литература**

1. «Чистый код. Создание, анализ и рефакторинг». Роберт Мартин
2. «Чистая архитектура. Искусство разработки программного обеспечения» Роберт Мартин

## **Практикум**

Требуется привести ваш код в проекте “Объявления” (*результирующий вариант в лабе №5 MySQL*) в “чистый” вид