

# Anti-patterns

---

⋮

From time to time it's necessary to review your TDD techniques and remind yourself of behaviours to avoid.

The TDD process is conceptually simple to follow, but as you do it you'll find it challenging your design skills. **Don't mistake this for TDD being hard, it's design that's hard!**

This chapter lists a number of TDD and testing anti-patterns, and how to remedy them.

---

## Not doing TDD at all

Of course, it is possible to write great software without TDD but, a lot of problems I've seen with the design of code and the quality of tests would be very difficult to arrive at if a disciplined approach to TDD had been used.

One of the strengths of TDD is that it gives you a formal process to break down problems, understand what you're trying to achieve (red), get it done (green), then have a good think about how to make it right (blue/refactor).

Without this, the process is often ad-hoc and loose, **which *can* make engineering more difficult than it *could* be.**

---

## Misunderstanding the constraints of the refactoring step

I have been in a number of workshops, mobbing or pairing sessions where someone has made a test pass and is in the refactoring stage. After some thought, they think it would be good to abstract away some code into a new struct; a budding pedant yells:

You're not allowed to do this! You should write a test for this first, we're doing TDD!

This seems to be a common misunderstanding. **You can do whatever you like to the code when the tests are green, the only thing you're not allowed to do is add or change behaviour.**

The point of these tests are to give you the *freedom to refactor*, find the right abstractions and make the code easier to change and understand

---

## Having tests that won't fail (or, evergreen tests)

It's astonishing how often this comes up. You start debugging or changing some tests and realise: there are no scenarios where this test can fail. Or at least, it won't fail in the way the test is *supposed* to be protecting against.

This is *next to impossible* with TDD if you're following **the first step**,

Write a test, see it fail

This is almost always done when developers write tests *after* code is written, and/or chasing test coverage rather than creating a useful test suite.

---

## Useless assertions

Ever worked on a system, and you've broken a test, then you see this?

```
false was not equal to true
```

I know that false is not equal to true. This is not a helpful message; it doesn't tell me what I've broken. This is a symptom of not following the TDD process and not reading the failure error message.

Going back to the drawing board,

Write a test, see it fail (and don't be ashamed of the error message)

---

## Asserting on irrelevant detail

An example of this is making an assertion on a complex object, when in practice all you care about in the test is the value of one of the fields.

```
1 // not this, now your test is tightly coupled to the whole object
2 if !cmp.Equal(complexObject, want) {
3     t.Error("got %+v, want %+v", complexObject, want)
4 }
5
6 // be specific, and loosen the coupling
7 got := complexObject.fieldYouCareAboutForThisTest
8 if got != want {
9     t.Error("got %q, want %q", got, want)
10 }
```

Additional assertions not only make your test more difficult to read by creating 'noise' in your documentation, but also needlessly couples the test with data it doesn't care about. This means if you happen to change the fields for your object, or the way they behave you may get unexpected compilation problems or failures with your tests.

This is an example of not following the red stage strictly enough.

- Letting an existing design influence how you write your test **rather than thinking of the desired behaviour**
- Not giving enough consideration to the failing test's error message

---

## Lots of assertions within a single scenario for unit tests

Many assertions can make tests difficult to read and challenging to debug when they fail.

They often creep in gradually, especially if test setup is complicated because you're reluctant to replicate the same horrible setup to assert on something else. Instead of this you should fix the problems in your design which are making it difficult to assert on new things.

A helpful rule of thumb is to aim to make one assertion per test. In Go, take advantage of subtests to clearly delineate between assertions on the occasions where you need to. This is also a handy technique to separate assertions on behaviour vs implementation detail.

For other tests where setup or execution time may be a constraint (e.g an acceptance test driving a web browser), you need to weigh up the pros and cons of slightly trickier to debug tests against test execution time.

---

# Not listening to your tests

Dave Farley in his video "When TDD goes wrong" points out,

TDD gives you the fastest feedback possible on your design

From my own experience, a lot of developers are trying to practice TDD but frequently ignore the signals coming back to them from the TDD process. So they're still stuck with fragile, annoying systems, with a poor test suite.

Simply put, if testing your code is difficult, then *using* your code is difficult too. **Treat your tests as the first user of your code and then you'll see if your code is pleasant to work with or not.**

I've emphasised this a lot in the book, and I'll say it again **listen to your tests**.

**Excessive setup, too many test doubles, etc.** 🤔

Ever looked at a test with 20, 50, 100, 200 lines of setup code before anything interesting in the test happens? Do you then have to change the code and revisit the mess and wish you had a different career?

What are the signals here? *Listen*, complicated tests == complicated code. Why is your code complicated? Does it have to be?

- When you have lots of test doubles in your tests, that means the code you're testing has lots of dependencies - which means your design needs work.
- If your test is reliant on setting up various interactions with mocks, that means your code is making lots of interactions with its dependencies. Ask yourself whether these interactions could be simpler.

Leaky interfaces

If you have declared an `interface` that has many methods, that points to a leaky abstraction. Think about how you could define that collaboration with a more consolidated set of methods, ideally one.

Think about the types of test doubles you use

- Mocks are sometimes helpful, but they're extremely powerful and therefore easy to misuse. Try giving yourself the constraint of using stubs instead.
- Verifying implementation detail with spies is sometimes helpful, but try to avoid it. Remember your implementation detail is usually not important, and you don't want your tests coupled to them if possible. Look to couple your tests to **useful behaviour rather than incidental details**.
- [Read my posts on naming test doubles](#) if the taxonomy of test doubles is a little unclear

## Consolidate dependencies

Here is some code for a `http.HandlerFunc` to handle new user registrations for a website.

```
1 type User struct {
2     // Some user fields
3 }
4
5 type UserStore interface {
6     CheckEmailExists(email string) (bool, error)
7     StoreUser(newUser User) error
8 }
9
10 type Emitter interface {
11     SendEmail(to User, body string, subject string) error
12 }
13
14 func NewRegistrationHandler(userStore UserStore, emitter Emitter) http.HandlerFunc {
15     return func(writer http.ResponseWriter, request *http.Request) {
16         // extract out the user from the request body (handle error)
17         // check user exists (handle duplicates, errors)
18         // store user (handle errors)
19         // compose and send confirmation email (handle error)
20         // if we got this far, return 2xx response
21     }
22 }
```

At first pass it's reasonable to say the design isn't so bad. It only has 2 dependencies!

Re-evaluate the design by considering the handler's responsibilities:

- Parse the request body into a `User`
- Use `UserStore` to check if the user exists

- Use `UserStore` to store the user
- Compose an email
- Use `Emailer` to send the email
- Return an appropriate http response, depending on success, errors, etc

To exercise this code, you're going to have to write many tests with varying degrees of test double setups, spies, etc

- What if the requirements expand? Translations for the emails? Sending an SMS confirmation too? Does it make sense to you that you have to change a HTTP handler to accommodate this change?
- Does it feel right that the important rule of "we should send an email" resides within a HTTP handler?
  - Why do you have to go through the ceremony of creating HTTP requests and reading responses to verify that rule?



**Listen to your tests.** Writing tests for this code in a TDD fashion should quickly make you feel uncomfortable (or at least, make the lazy developer in you be annoyed). If it feels painful, stop and think.

What if the design was like this instead?

```

1 type UserService interface {
2     Register(newUser User) error
3 }
4
5 func NewRegistrationHandler(userService UserService) http.HandlerFunc {
6     return func(writer http.ResponseWriter, request *http.Request) {
7         // parse user
8         // register user
9         // check error, send response
10    }
11 }

```

- Simple to test the handler 
- Changes to the rules around registration are isolated away from HTTP, so they are also simpler to test 

---

## Violating encapsulation

Encapsulation is very important. There's a reason we don't make everything in a package exported (or public). **We want coherent APIs with a small surface area to avoid tight coupling.**

People will sometimes be tempted to make a function or method public in order to test something. By doing this you make your design worse and send confusing messages to maintainers and users of your code.

A result of this can be developers trying to debug a test and then eventually realising the function being tested is *only called from tests*. Which is obviously a **terrible outcome, and a waste of time.**

In Go, consider your default position for writing tests as *from the perspective of a consumer of your package*. You can make this a compile-time constraint by having your tests live in a test package e.g `package gocoin_test`. If you do this, you'll only have access to the exported members of the package so it won't be possible to couple yourself to implementation detail.

---

## Complicated table tests

Table tests are a great way of exercising a number of different scenarios when the test setup is the same, and you only wish to vary the inputs.

*But* they can be messy to read and understand when you try to shoehorn other kinds of tests under the name of having one, glorious table.

```
1 cases := []struct {  
2     X          int  
3     Y          int  
4     Z          int  
5     err        error  
6     IsFullMoon bool  
7     IsLeapYear  bool  
8     AtWarWithEurasia bool  
9 }{}
```

**Don't be afraid to break out of your table and write new tests** rather than adding new fields and booleans to the table `struct`.

A thing to bear in mind when writing software is,

Simple is not easy

"Just" adding a field to a table might be easy, but it can make things far from simple.

---

## Summary

Most problems with unit tests can normally be traced to:

- Developers not following the TDD process
- Poor design

So, learn about good software design!

The good news is TDD can help you *improve your design skills* because as stated in the beginning:

**TDD's main purpose is to provide feedback on your design.** For the millionth time, listen to your tests, they are reflecting your design back at you.

Be honest about the quality of your tests by listening to the feedback they give you, and you'll become a better developer for it