Extracted from:

Modern C++ Programming with Test-Driven Development

Code Better, Sleep Better

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Modern C++ Programming with Test-Driven Development

Code Better, Sleep Better

Jeff Langr

Edited by Michael Swaine

Modern C++ Programming with Test-Driven Development

Code Better, Sleep Better

Jeff Langr

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The team that produced this book includes:

Michael Swaine (editor)
Potomac Indexing, LLC (indexer)
Kim Wimpsett (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-937785-48-2
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—October 2013

2.1 Setup

Write a test, get it to pass, clean up the design. That's all there is to TDD. Yet these three simple steps embody a significant amount of sophistication. Understand how to take advantage of TDD, and you will be rewarded with many benefits. Fail to heed what others have learned, and you will likely give up on TDD.

Rather than let you flounder, I'd like to guide you through test-driving some code in order to help you understand what happens at each step. You'll learn best by coding along with the example in this chapter. Make sure you've set up your environment properly (see Chapter 1, *Global Setup*, on page ?).

While not large, the example we'll work through isn't useless or trivial (but it's also not rocket science). It provides many teaching points and demonstrates how TDD can help you incrementally design a reasonably involved algorithm.

I hope you're ready to code!

2.2 The Soundex Class

Searching is a common need in many applications. An effective search should find matches even if the user misspells words. Folks misspell my name in endless ways: Langer, Lang, Langur, Lange, and Lutefisk, to name a few. I'd prefer they find me regardless.

In this chapter, we will test-drive a Soundex class that can improve the search capability in an application. The long-standing Soundex algorithm encodes words into a letter plus three digits, mapping similarly sounding words to the same encoding. Here are the rules for Soundex, per Wikipedia:[1]

1. Retain the first letter. Drop all other occurrences of *a*, *e*, *i*, *o*, *u*, *y*, *h*, *w*.
2. Replace consonants with digits (after the first letter):
   - *b*, *f*, *p*, *v*: 1
   - *c*, *g*, *j*, *k*, *q*, *s*, *x*, *z*: 2
   - *d*, *t* : 3
   - *l*: 4
   - *m*, *n*: 5
   - *r*: 6
3. If two adjacent letters encode to the same number, encode them instead as a single number. Also, do so if two letters with the same number are

1. http://en.wikipedia.org/wiki/Soundex

separated by *h* or *w* (but code them twice if separated by a vowel). This rule also applies to the first letter.

4. Stop when you have a letter and three digits. Zero-pad if needed.

2.3 Getting Started

A common misconception of TDD is that you first define *all* the tests before building an implementation. Instead, you focus on one test at a time and incrementally consider the next behavior to drive into the system from there.

As a general approach to TDD, you seek to implement the next simplest rule in turn. (For a more specific, formalized approach to TDD, refer to the TPP [Section 10.4, *The Transformation Priority Premise*, on page ?].) What useful behavior will require the most straightforward, smallest increment of code to implement?

With that in mind, where do we start with test-driving the Soundex solution? Let's quickly speculate as to what implementing each rule might entail.

Soundex rule #3 appears most involved. Rule #4, indicating when to stop encoding, would probably make more sense once the implementation of other rules actually resulted in something getting encoded. The second rule hints that the first letter should already be in place, so we'll start with rule #1. It seems straightforward.

The first rule tells us to retain the first letter of the name and...*stop*! Let's keep things as small as possible. What if we have *only* a single letter in the word? Let's test-drive that scenario.

c2/1/SoundexTest.cpp

```
Line 1 #include "gmock/gmock.h"
     2 TEST(SoundexEncoding, RetainsSoleLetterOfOneLetterWord) {
     3    Soundex soundex;
     4 }
```

- On line 1, we include gmock, which gives us all the functionality we'll need to write tests.
- A simple test declaration requires use of the TEST macro (line 2). The TEST macro takes two parameters: the name of the test case and a descriptive name for the test. A *test case*, per Google's documentation, is a related collection of tests that can "share data and subroutines."[2] (The term is overloaded; to some, a test case represents a single scenario.)

2. http://code.google.com/p/googletest/wiki/V1_6_Primer#Introduction:_Why_Google_C++_Testing_Framework?

Test Lists

Each test you write in TDD and get to pass represents a new, working piece of behavior that you add to the system. Aside from getting an entire feature shipped, your passing tests represent your best measure of progress. You name each test to describe the small piece of behavior.

While you don't determine all the tests up front, you'll probably have an initial set of thoughts about what you need to tackle. Many test-drivers capture their thoughts about upcoming tests in a *test list* (described first in *Test Driven Development: By Example [Bec02]*). The list can contain names of tests or reminders of code cleanup that you need to do.

You can keep the test list on a scratch pad on the side of your workstation. (You could also type it at the bottom of your test file as comments—just make sure you delete them before checking in!) The list is yours alone, so you can make it as brief or cryptic as you like.

As you test-drive and think of new test cases, add them to the list. As you add code you know will need to be cleaned up, add a reminder to the list. As you complete a test or other task, cross it off the list. It's that simple. If you end up with outstanding list items at the end of your programming session, set the list aside for a future session.

You might think of the test list as a piece of initial design. It can help you clarify what you think you need to build. It can also help trigger you to think about other things you need to do.

Don't let the test list constrain what you do or the order in which you do it, however. TDD is a fluid process, and you should usually go where the tests suggest you go next.

Managing test lists can be particularly useful when you're learning TDD. Try it!

Reading the test case name and test name together, left to right, reveals a sentence that describes what we want to verify: "Soundex encoding retains [the] sole letter of [a] one-letter word." As we write additional tests for Soundex encoding behavior, we'll use SoundexEncoding for the test case name to help group these related tests.

Don't discount the importance of crafting good test names—see the following sidebar.

The Importance of Test Names

Take great care with naming. The small investment of deriving highly descriptive test names pays well over time, as tests are read and reread by others who must maintain the code. Crafting a good test name will also help you, the test writer, better understand the intent of what you're about to build.

You'll be writing a number of tests for each new behavior in the system. Think about the set of test names as a concordance that quickly provides a developer with a concise summary of that behavior. The easier the test names are to digest, the more quickly you and other developers will find what you seek.

- On line 3, we create a Soundex object, then…stop! Before we proceed with more testing, we know we've just introduced code that won't compile—we haven't yet defined a Soundex class! We'll stop coding our test and fix the problem before moving on. This approach is in keeping with Uncle Bob's Three Rules of TDD:

  - Write production code only to make a failing test pass.
  - Write no more of a unit test than sufficient to fail. Compilation failures are failures.
  - Write only the production code needed to pass the one failing test.

  (*Uncle Bob* is Robert C. Martin. See Section 3.4, *The Three Rules of TDD,* on page ? for more discussion of the rules.

  Seeking incremental feedback can be a great approach in C++, where a few lines of test can generate a mountain of compiler errors. Seeing an error as soon as you write the code that generates it can make it easier to resolve.

  The three rules of TDD aside, you'll find that sometimes it makes more sense to code the entire test before running it, perhaps to get a better feel for how you should design the interface you're testing. You might also find that waiting on additional slow compiles isn't worth the trade-off in more immediate feedback.

  For now, particularly as you are learning TDD, seek feedback as soon as it can be useful. Ultimately, it's up to you to decide how incrementally you approach designing each test.

The compiler shows that we indeed need a Soundex class. We could add a compilation unit (`.h`/`.cpp` combination) for Soundex, but let's make life easier for ourselves. Instead of mucking with separate files, we'll simply declare everything in the same file as the test.

Once we're ready to push our code up or once we experience pain from having everything in one file, we'll do things the proper way and split the tests from the production code.

c2/2/SoundexTest.cpp

```
➤ class Soundex {
➤ };

#include "gmock/gmock.h"

TEST(SoundexEncoding, RetainsSoleLetterOfOneLetterWord) {
   Soundex soundex;
}
```

Q.: *Isn't putting everything into a single file a dangerous shortcut?*

A.: *It's a calculated effort to save time in a manner that incurs no short-term complexity costs. The hypothesis is that the cost of splitting files later is less than the overhead of flipping between files the whole time. As you shape the design of a new behavior using TDD, you'll likely be changing the interface often. Splitting out a header file too early would only slow you down.*

As far as "dangerous" is concerned: are you ever going to forget to split the files before checking in?

Q.: *But aren't you supposed to be cleaning up code as you go as part of following the TDD cycle? Don't you want to always make sure that your code retains the highest possible quality?*

A.: *In general, yes to both questions. But our code is fine; we're simply choosing a more effective organization until we know we need something better. We're deferring complexity, which tends to slow us down, until we truly need it. (Some Agile proponents use the acronym YAGNI—"You ain't gonna need it.")*

If the notion bothers you deeply, go ahead and separate the files right off the bat. You'll still be able to follow through with the rest of the exercise. But I'd prefer you first try it this way. TDD provides you with safe opportunities to challenge yourself, so don't be afraid to experiment with what you might find to be more effective ways to work.

We're following the third rule for TDD: write only enough production code to pass a test. Obviously, we don't have a complete test yet. The RetainsSoleLetterOfOneLetterWord test doesn't really execute any behavior, and it doesn't verify anything. Still, we can react to each incremental bit of negative feedback (in this case, failing compilation) and respond with just enough code to get past the negative feedback. To compile, we added an empty declaration for the Soundex class.

Building and running the tests at this point gives us positive feedback.

```
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from SoundexEncoding
[ RUN      ] SoundexEncoding.RetainsSoleLetterOfOneLetterWord
[       OK ] SoundexEncoding.RetainsSoleLetterOfOneLetterWord (0 ms)
[----------] 1 test from SoundexEncoding (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (0 ms total)
[  PASSED  ] 1 test.
```

Party time!

Well, not quite. After all, the test does nothing other than construct an instance of the empty Soundex class. Yet we have put some important elements in place. More importantly, we've proven that we've done so correctly.

Did you get this far? Did you mistype the include filename or forget to end the class declaration with a semicolon? If so, you've created your mistake within the fewest lines of code possible. In TDD, you practice safe coding by testing early and often, so you usually have but one reason to fail.

Since our test passes, it might be a good time to make a local commit. Do you have the right tool? A good version control system allows you to commit easily every time the code is *green* (in other words, when all tests are passing). If you get into trouble later, you can easily revert to a known good state and try again.

Part of the TDD mentality is that every passing test represents a proven piece of behavior that you've added to the system. It might not always be something you could ship, of course. But the more you think in such incremental terms and the more frequently you seek to integrate your locally proven behavior, the more successful you'll be.

Moving along, we add a line of code to the test that shows how we expect client code to interact with Soundex objects.

c2/3/SoundexTest.cpp

```
 TEST(SoundexEncoding, RetainsSoleLetterOfOneLetterWord) {
    Soundex soundex;

➤   auto encoded = soundex.encode("A");
 }
```

We are making decisions as we add tests. Here we decided that the Soundex class exposes a public member function called encode() that takes a string argument. Attempting to compile with this change fails, since encode() doesn't exist. The negative feedback triggers us to write just enough code to get everything to compile and run.

c2/4/SoundexTest.cpp

```
  class Soundex
  {
➤ public:
➤    std::string encode(const std::string& word) const {
➤       return "";
➤    }
  };
```

The code compiles, and all the tests pass, which is still not a very interesting event. It's finally time to verify something useful: given the single letter *A*, can encode() return an appropriate Soundex code? We express this interest using an *assertion*.

c2/5/SoundexTest.cpp

```
  TEST(SoundexEncoding, RetainsSoleLetterOfOneLetterWord) {
     Soundex soundex;

     auto encoded = soundex.encode("A");

➤    ASSERT_THAT(encoded, testing::Eq("A"));
  }
```

An assertion verifies whether things are as we expect. The assertion here declares that the string returned by encode() is the same as the string we passed it. Compilation succeeds, but we now see that our first assertion has failed.

```
[==========]  Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from SoundexEncoding
[ RUN      ] SoundexEncoding.RetainsSoleLetterOfOneLetterWord
SoundexTest.cpp:21: Failure
Value of: encoded
Expected: is equal to 0x806defb pointing to "A"
  Actual: "" (of type std::string)
[  FAILED  ] SoundexEncoding.RetainsSoleLetterOfOneLetterWord (0 ms)
[----------] 1 test from SoundexEncoding (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (0 ms total)
[  PASSED  ] 0 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] SoundexEncoding.RetainsSoleLetterOfOneLetterWord

 1 FAILED TEST
```

At first glance, it's probably hard to spot the relevant output from Google Mock. We first read the very last line. If it says "PASSED," we stop looking at the test output—all our tests are working! If it says "FAILED" (it does in our example), we note how many test cases failed. If it says something other than "PASSED" or "FAILED," the test application itself crashed in the middle of a test.

With one or more failed tests, we scan upward to find the individual test that failed. Google Mock prints a [RUN] record with each test name when it starts and prints a [FAILED] or [OK] bookend when the test fails. On failure, the lines between [RUN] and [OK] might help us understand our failure. In the output for our first failed test shown earlier, we see the following:

```
[ RUN      ] SoundexEncoding.RetainsSoleLetterOfOneLetterWord
SoundexTest.cpp:21: Failure
Value of: encoded
Expected: is equal to 0x806defb pointing to "A"
  Actual: "" (of type std::string)
[  FAILED  ] SoundexEncoding.RetainsSoleLetterOfOneLetterWord (0 ms)
```

Paraphrasing this assertion failure, *Google Mock expected the local variable* `encoded` *to contain the string* `"A"`*, but the actual string it contained was equal to the empty string.*

We expected a failing assertion, since we deliberately hard-coded the empty string to pass compilation. That negative feedback is a good thing and part of the TDD cycle. We want to first ensure that a newly coded assertion—representing functionality we haven't built yet—doesn't pass. (Sometimes it does, which is usually not a good thing; see Section 3.5, *Getting Green on Red*, on page ?.) We also want to make sure we've coded a legitimate test; seeing it first fail and then pass when we write the appropriate code helps ensure our test is honest.

The failing test prods us to write code, no more than necessary to pass the assertion.

c2/6/SoundexTest.cpp

```
std::string encode(const std::string& word) const {
➤   return "A";
}
```

We compile and rerun the tests. The final two lines of its output indicate that all is well.

```
[==========] 1 test from 1 test case ran. (0 ms total)
[  PASSED  ] 1 test.
```

Ship it!

I'm kidding, right? Well, no. We want to work incrementally. Let's put it this way: if someone told us to build a Soundex class that supported encoding only the letter *A*, we'd be done. We'd want to clean things up a little bit, but otherwise we'd need no additional logic.

Another way of looking at it is that the tests specify all of the behavior that we have in the system to date. Right now we have one test. Why would we have any more code than what that test states we need?

We're not done, of course. We have plenty of additional needs and requirements that we'll incrementally test-drive into the system. We're not even done with the current test. We *must*, *must*, *must* clean up the small messes we just made.

2.4 Fixing Unclean Code

What? We wrote one line of production code and three lines of test code and we have a problem? Indeed. It's extremely easy to introduce deficient code even in a small number of lines. TDD provides the wonderful opportunity to fix such small problems as they arise, before they add up to countless small problems (or even a few big problems).

We read both the test and production code we've written, looking for deficiencies. We decide that the assertion in our test isn't reader-friendly.

```
ASSERT_THAT(encoded, testing::Eq("A"));
```

Much as the test declaration (the combination of test case and test name) should read like a sentence, we want our asserts to do the same. We introduce a `using` directive to help.

c2/7/SoundexTest.cpp
```
 #include "gmock/gmock.h"
➤ using ::testing::Eq;

 TEST(SoundexEncoding, RetainsSoleLetterOfOneLetterWord) {
    Soundex soundex;
    auto encoded = soundex.encode("A");
➤    ASSERT_THAT(encoded, Eq("A"));
 }
```

Now we can paraphrase the assertion with no hiccups: *assert that the* encoded *value is equal to the string* "A".

Our small change is a *refactoring*, a code transformation in which we retain existing behavior (as demonstrated by the test) but improve the design. In this case, we improved the test's design by enhancing its expressiveness. The namespace of Eq() is an implementation detail not relevant to the test's meaning. Hiding that detail improves the level of abstraction in the test.

Code duplication is another common challenge we face. The costs and risks of maintenance increase with the amount of code duplication.

Our Soundex class contains no obvious duplication. But looking at both the test and production code in conjunction reveals a common magic literal, the string "A". We want to eliminate this duplication. Another problem is that the test name (RetainsSoleLetterOfOneLetterWord) declares a general behavior, but the implementation supports only a specific, single letter. We want to eliminate the hard-coded "A" in a way that solves both problems.

How about simply returning the word passed in?

c2/8/SoundexTest.cpp
```
 class Soundex
 {
 public:
    std::string encode(const std::string& word) const {
➤       return word;
    }
 };
```

At any given point, your complete set of tests declares the behaviors you intend your system to have. That implies the converse: if no test describes a behavior, it either doesn't exist or isn't intended (or the tests do a poor job of describing behavior).

Where am I going with this? We have one test. It says we support one-letter words. Therefore, we can assume that the Soundex code needs to support only one-letter words—for now. And if all words are one letter, the simplest generalized solution for our test is to simply return the whole word passed to `encode()`.

(There are other TDD schools of thought about what we might have coded at this point. One alternate technique is *triangulation*[3]—see Triangulation, on page ?—where you write a second, similar assertion but with a different data expectation in order to drive in the generalized solution. You'll discover more alternate approaches throughout the book, but we'll keep things simple for now.)

Our changes here are small, bordering on trivial, but now is the time to make them. TDD's refactoring step gives us an opportunity to focus on all issues, significant or minor, that arise from a small, isolated code change. As we drive through TDD cycles, we'll use the refactoring step as our opportunity to review the design impacts we just made to the system, fixing any problems we just created.

Our primary refactoring focus will be on increasing expressiveness and eliminating duplication, two concerns that will give us the most benefit when it comes to creating maintainable code. But we'll use other nuggets of design wisdom as we proceed, such as SOLID class design principles and code smells.

2.5 Incrementalism

It's question-and-answer (Q&A) time!

Q.: *Do you really code like this, hard-coding things that you know you'll replace?*

A.: *I always get this question. Yes.*

Q.: *It seems stupid!*

A.: *That's not a question, but yes, it's an OK first reaction to think this is stupid. It felt stupid to me at first, too. I got over it.*

Q.: *Are we going to keep working like this? How will we get anything done if we hard-code everything?*

A.: *That's two questions, but I'm happy to answer them both! Yes, we will keep working incrementally. This technique allows us to get a first passing test in place quickly. No worries, the hard-coded value will last only minutes at most. We know we're not done with what we need to build, so we'll have to write more tests to describe additional behavior. In this example, we know we must support the rest of the rules. As we write additional tests, we'll have to replace the hard-coding with interesting logic in order to get the additional tests to pass.*

3. *Test Driven Development: By Example [Bec02]*

Incrementalism is at the heart of what makes TDD successful. An incremental approach will seem quite unnatural and slow at first. However, taking small steps will increase your speed over time, partly because you will avoid errors that arise from taking large, complicated steps. Hang in there!

Astute readers will note that we've already coded something that does not completely meet the specification (*spec*) for Soundex. The last part of rule #4 says that we must "fill in zeros until there are three numbers." Oh, the joy of specs! We must read them comprehensively and carefully to fully understand how all their parts interact. (Better that we had a customer to interact with, someone who could clarify what was intended.) Right now it seems like rule #4 contradicts what we've already coded.

Imagine that the rules are being fed to us one by one. "Get the first part of rule #1 working, and then I'll give you a new rule." TDD aligns with this latter approach—each portion of a spec is an incremental addition to the system. An incremental approach allows us to build the system piecemeal, in any order, with continually verified, forward progress. There is a trade-off: we might spend additional time incorporating a new increment than if we had done a bit more planning. We'll return to this concern throughout the book. For now, let's see what happens when we avoid worrying about it.

We have two jobs: write a new test that describes the behavior, and change our existing test to ensure it meets the spec. Here's our new test:

c2/9/SoundexTest.cpp
```
TEST(SoundexEncoding, PadsWithZerosToEnsureThreeDigits) {
   Soundex soundex;

   auto encoded = soundex.encode("I");

   ASSERT_THAT(encoded, Eq("I000"));
}
```

(One reviewer asks, "Why didn't we read the Soundex rules more carefully and write this first?" Good question. Indeed, we weren't careful. A strength of TDD is its ability to let you move forward in the face of incomplete information and in its ability to let you correct earlier choices as new information arises.)

Each test we add is independent. We don't use the outcome of one test as a precondition for running another. Each test must set up its own context. Our new test creates its own Soundex instance.

A failing test run shows that `encode()` returns the string "I" instead of "I000". Getting it to pass is straightforward.

c2/9/SoundexTest.cpp
```
   std::string encode(const std::string& word) const {
➤     return word + "000";
   }
```

Hard-coding an answer may again ruffle feathers, but it will help us keep on track. Per our tests so far, the Soundex class requires no additional behavior. Also, by building the smallest possible increment, we're forced to write additional tests in order to add more behavior to the system.

Our new test passes, but the first test we wrote now fails. The behavior it describes, by example, does not match the specification we derived from Wikipedia.

When done test-driving, you'll know that the tests correctly describe how your system works, as long as they pass. They provide examples that can read easier than specs, if crafted well. We'll continue to focus on making the tests readable in our exercises (and I might even casually refer to them as specs).

c2/9/SoundexTest.cpp
```
   TEST(SoundexEncoding, RetainsSoleLetterOfOneLetterWord) {
      Soundex soundex;

      auto encoded = soundex.encode("A");

➤     ASSERT_THAT(encoded, Eq("A000"));
   }
```

That wasn't too tough!

We now have two tests that perform the same steps, though the data differs slightly. That's OK; each test now discretely documents one piece of behavior. We not only want to make sure the system works as expected, we want everyone to understand its complete set of intended behaviors.

It's time for refactoring. The statement in encode() isn't as clear about what's going on as it could be. We decide to extract it to its own method with an intention-revealing name.

c2/10/SoundexTest.cpp
```
   public:
      std::string encode(const std::string& word) const {
➤        return zeroPad(word);
      }

   private:
➤     std::string zeroPad(const std::string& word) const {
➤        return word + "000";
➤     }
```