Test Cases for Tip Calculator:

**As a customer**

I want to calculate the total bill (total plus the tip)

Because I want to compliment the restaurant for the service

Scenario #1: Customer calculates the total for terrible service

Given that the restaurant total is $100.00

And the service was terrible

When the tip calculator calculates the total charge

Then tip calculator shows $100.00 total charge

Scenario #2: Customer calculates the total for poor service

Given that the restaurant total is $100.00

And the service was poor

When the tip calculator calculates the total charge

Then tip calculator shows $105.00 total charge

Scenario #3: Customer calculates the total for good service

Given that the restaurant total is $100.00

And the service was good

When the tip calculator calculates the total charge

Then tip calculator shows $110.00 total charge

Scenario #4: Customer calculates the total for great service

Given that the restaurant total is $100.00

And the service was great

When the tip calculator calculates the total charge

Then tip calculator shows $115.00 total charge

Scenario #5: Customer calculates the total for excellent service

Given that the restaurant total is $100.00

And the service was excellent

When the tip calculator calculates the total charge

Then tip calculator shows $120.00 total charge

**First design decision**

At this point, we are forced to make a decision – how will our nascent Tip Calculator know if the service rating provided by the user is valid or not?

The design decision that comes to mind is that Tip Calculator must be able to store and retrieve some data. In this case, the data we’re interested in is the service rating.

If we go back to the user story and review the five acceptance criteria, we will see that the expectations are that Tip Calculator must be able to recognize five different service ratings:

Terrible

Poor

Good

Great

Excellent

**Replace hardcoded value with actual processing logic**

Let’s now replace the hardcoded value with actual running code. We first teach our Tip Calculator that there is a service rating called “Terrible” and that tip percentage associated with this rating is 0.

**Flip the coin**

Any time we satisfy a positive expectation, it is a prudent practice to turn things on their head and describe the negative expectation.

At this point, since we’ve satisfied that a legitimate service rating value is found in the Tip Calculator, we want to ensure that non-legitimate values are not found in the Tip Calculator.

**In praise of steady state**

Software engineering is a balancing act between steady state and periods of unstable state. What do we mean by steady state?

If we have a system (a running application) that behaves the way we expect it to behave (i.e. it produces values we have specified as expected values), we declare that the system is in a steady state. It is running, and delivering some value.

That value delivery is still partial. In our case, the only value to the users this Tip Calculator delivers is its ability to recognize service rating “Terrible” as a legitimate rating. In addition, it is capable of informing us that service rating “Whatever” is not a legitimate rating.

**Check if correct tip percentage is associated with service rating**

Let’s now introduce another change into our application – a test to verify if correct tip percentage is associated with service rating “Terrible”.

**What tip percentage do we expect for non-legitimate service ratings?**

Many years of experience in the field taught me to be a bit pessimistic. Now that we have our application back in the steady state and delivering value (answering questions about legitimate service ratings and also giving us correct tip percentage for the “Terrible” rating), we need to see what happens when we run our application by giving it non-legitimate service rating value (for example, by giving it service rating “Whatever”).

**Populate other service rating tip percentages**

Now is a good time to take a breather and make less risky changes, following the already established pattern. Leave the safety of the steady state and make short trips into the volatile territory by adding a new test to verify if service rating “Poor” is a valid, legitimate rating

**Calculate grand total given the total and the service rating**

We are now ready for the final step – given the total bill and the service rating, we expect Tip Calculator to calculate tip percentage and add it to the total, producing the grand total to be paid to the restaurant.

**Red-Green-Refactor-Reflect**

In this case, the simple goal is to calculate the tip amount from the supplied service rating and the restaurant bill total. The calculated tip amount is then automatically added to the total.

From there we proceeded to build our shipping application by following the TDD methodology. As we’ve demonstrated, the methodology consists of writing a failing test, observing it fail (the Red phase of TDD), then immediately making code changes that ensure the test passes (the Green phase of TDD). Once the test passes, we move into the Refactor phase (we restructure the code without affecting its behaviour). That way, we make sure our code is not expensive to change.

A proper TDD practice also mandates frequent retrospective – we call it reflection. We stop and think about the things we’ve accomplished thus far, to see if we could learn from our recent experiences. This reflection fortifies the process, as it relies on frequent and tight feedback provided by the failing, then succeeding tests.

I have already compared Test Driven Development to the experience of riding a galloping horse. While riding a horse, we’re alternating between flying through the air (i.e. speed achieved when the horse leaps from the ground) and steering the horse. It is impossible to steer the horse while we’re off the ground, up in the air. At that point, we gain speed, but we cannot make any changes of the direction. It is only once the horse touches the ground that we can make a change in direction.

In TDD, we strive to touch the ground as frequently as possible. The longer the leaps we make without touching the ground, the less chance we have for correcting the course.

I also compared software development practices that don’t follow TDD principles to the experience of flying a kite. When flying a kite, we never touch the ground. It is an exhilarating feeling of letting the wind pick the kite up and bounce it up in the air. We can achieve considerable speed that way. But we struggle in such situations to maintain desired course. And after we eventually land the kite, it usually does not land in the spot we originally wanted it to land.

Once they start learning about agile and TDD, they may reconsider their practices and decide that writing tests before writing implementation code may make more sense. Still, because of the ingrained waterfall mentality, some of those engineers make the mistake of writing all tests first, and only then move into writing the code.

That approach is completely wrong. It is equivalent to the traditional waterfall approach where we go through the development process by respecting gated phases.

**Conclusion**

Building a simple Tip Calculator is a toy sized problem, and using that exercise to illustrate TDD methodology is not necessarily providing a convincing argument in favour of TDD. Still, within the constraints of a technical article, going over this hands-on exercise may provide valuable insights into the benefits of adopting TDD.

We would still argue that the real benefits of TDD only become apparent when dealing with much larger, more complex software engineering efforts. The ability to remain grounded while making potentially risky changes to a large, complex system is often a life saver.

In addition to that, building software using TDD methodology results in much less rework. TDD drives high degree of modularization, which results in high cohesiveness of the modules and low coupling between the modules.

All these characteristics produce a shipping application whose codebase is easy and inexpensive to change. And lowering the cost of change has proven to be the best way on the path to embracing changes and abandoning the concept known as ‘scope creep’.

Bottom line, TDD enables software engineering teams to deliver high degree of flexibility to the business.