TDD in Ruby

A Gentle Introduction for Beginners

Bala Paranj
www.rubyplus.com

Published By Zepho Inc

ISBN-13: 978-1508957850

Let's Start with the Basics

Terminology

Kata

Kata is a Japanese word meaning form. In the martial arts, it describes a choreographed pattern of movements used to train yourself to the level of muscle memory.

What is a Coding Kata?

A Coding Kata is a short exercise, usually 30 mins to an hour long. It can be coded in many different ways. It is likely that the Katas have many solutions. Your focus is on learning when you work through them. The goal is to practice in order to improve your skills, not perfection. Instead of having theory in your head, you want the skills to be at your finger tips.

Why Coding Kata?

Test Driven Development is a difficult skill to master. Coding Kata is the best way to learn TDD.

What is a Domain?

A domain is defined as a specified sphere of activity or knowledge. For instance, visual communication is the domain of the Graphic Designer.

What is a Problem Domain?

Problem Domain refers to real-world things and concepts related to the problem.

What is a Solution Domain?

Solution Domain refers to real-world things and concepts related to the solution.

Example

Let's say that you have a leaking sink problem. You search on the Internet for the term leaking sink. This is the problem domain.

Once you read about the solution to this problem, you learn about terms like Clevis Screw, Stopper Rod, Horizontal Pivot Rod, Retaining Nut and so on. These terms belong to the solution domain.

You find out that you need to buy a Retaining Nut to fix the leak. You now search using the term found in the solution domain 'Retaining Nut' to find the nearest store carrying this item.

What vs How

The music sheet is not music. It is the description of music. This is the 'What' or Logical Design. Music is played using musical instruments. This is the 'How' or the Physical Design.

John Lennon wrote the song Come Together. This lyrics is the 'What'. The examples of 'How' in this case are the performances of Beatles, Aerosmith and Michael Jackson to the same song, Come Together.

The blueprint for a house is the 'What'. You can build many houses using the same blueprint, this is the 'How'.

Why distinguish the What and How?

We need to separate the 'What' from 'How' because it allows us to change implementation without breaking the tests. The focus of the test is 'What'. The focus of the production code is 'How'. As long as the behavior is the same, the implementation changes should not break the tests.

How to Separate the What from How?

Chris Stevenson's TestDox style expresses the subject in the code as part of a sentence.

- A sheep eats grass.
- A sheep bleats when frightened.
- A sheep produces delicious milk.
- A sheep moves away from sheep dogs.

In these sentences, the sheep is the subject. It's behavior is expressed in a sentence. This can be automatically converted to specifications in code.

```
describe Sheep do
  it 'eats grass'
  it 'bleats when frightened'
  it 'produces delicious milk'
  it 'moves away from sheep dogs'
end
```

When you think about the system from the outside, you focus on the intent. You focus on 'what' rather than the implementation which is the 'how'. In this example: What does the sheep do?

Focus of What

When you work on a programming problem, you start from the problem statement and you will ignore the implementation details. Your aim is to understand the problem. Abstraction allows us to think about what is being done, not how it is implemented.

Focus of How

You will focus on the internal details of the system such as data structures and algorithms to use in the code.

A Brief Introduction to TDD

What is TDD?

Test Driven Development is a software development practice where the test is written before the production code. The goal is clean code that works.

The old way of doing software development made it difficult to get feedback between decisions, since the decisions were separated in time. Development in the context of TDD means a complex dance of analysis, logical design, physical design, implementation, testing, review, integration and deployment.

Why TDD?

It leads to better quality and fewer defects in code. It eliminates the need to spend days in a debugger to hunt down hard to find bugs. It significantly reduces debugging effort. It shortens the feedback loop on design decisions. The feedback comes in seconds or minutes as you translate your ideas into code.

We end up with a simple design as a result of coding only what we need for the tests and removing all duplication. This design can adapt to the current requirements and all future requirements. The mind set of enough design to have the perfect architecture for the current system also makes writing the tests easier.

The goal of TDD is clean code that works. Kent Beck says:

TDD gives you a chance to learn all of the lessons that the code has to teach you. If you only slap together the first thing you think of, then you never have time to think of a second, better thing.

TDD is a way of constraining the problem, encapsulating the process of design by using the abstraction of a test and providing rapid feedback as to how well our design is working. The process gives us a way to think about and do iterative design.

What are the steps in TDD?

There are five steps, Kent Beck sums it up as follows:

1. Quickly add a test.
2. Run all tests and see the new one fail.
3. Make a little change to pass the test as quickly as possible. Commit whatever sins necessary in the process.
4. Run all tests and see them all succeed.
5. Refactor to remove duplication.

We start by writing a failing test, which leads us to the next question.

Why write a failing test first?

It is a way to test the test. If all tests are passing, it gives us feedback that there are no known problems with our code. By writing a test to expose a deficiency, we are clarifying the problem. We are also sketching out a design. It is a way to think about design. When you are writing the test, you have to think about:

1. Where does the functionality belong?
2. What should be name of the class or method?
3. How are we going to check for the right answer?
4. What is the right answer?
5. What other tests does this test suggest?

How to Write a Failing Test?

Let's look at the questions we need to ask ourselves to write a failing test:

1. What is our system's responsibility?
2. What should it do?
3. What is the API for making our system do this?
4. What does the system need to fulfill this responsibility? It may need data or collaborators.
5. What is the right answer?
6. How are we going to check for the right answer?

How to Make the Test Pass?

Once we write a failing test, we make our test pass with the simplest code. Is it too hard to get it working? Then we drop back to change our test. We get the feedback from TDD, the test is forcing too big a leap. Is the implementation so trivial it has obvious flaws? We use the feedback from TDD to tell us what the next test should expose.

Refactor

Finally, we have the refactoring step. Our code works, but we have been focused on passing a very specific test which only shows a very small picture of the application. Now is our chance to zoom out and take in the entire application. If we have used a naive implementation to get the test pass, we can clean up the duplication or extract methods to make our code self-describing.

How to Get All Benefits of TDD?

In order to get all benefits of TDD, programmers should follow each step. For instance, the second step states that

programmers should watch the new test fail and the fifth step states that we need to refactor the code to remove duplication. Sometimes programmers just do not perform all the steps of Kent Beck's description.

TDD is a simple technique since it only has few steps to be followed. However, in practice the steps are not that easy to follow as programmers need to be very disciplined. Kent Beck says:

I am not a great programmer. I am just a good programmer with great habits.

If you would like to learn more about how to do TDD and how to avoid common mistakes to cultivate good habits, please check out my Test Driven Development in Ruby course at www.rubyplus.com.

Why is TDD Difficult to Learn?

Kent Beck says:

TDD is not a testing technique. It's an analysis technique, a design technique, really a technique for all activities of development.

TDD is not a substitute for thinking. You as a developer make all the design decisions. It's not a replacement for design skills.

One way to make TDD easier to learn is to understand how it works. Start with a trivial, one class example to learn the basics of the red-green-refactor process, like implementing a Stack, but then stop and think about what the process is doing. I demonstrate this in my course 'Test Driven Development in Ruby'. Visit www.rubyplus.com for more details.

By using the small but precise nature of the Coding Kata to practice these skills separately, you can get over this difficulty and hone your TDD skills.

How TDD Separates the What from How?

TDD splits the design activity into two phases. First, we design the external face of the code, i.e., the API. Then we design the internal organization of the code. The external face of the code, i.e., the API is the 'What'. The design of the internal organization of the code is the 'How'. This is an useful distinction because you can vary the implementation without breaking the tests.

Problem Solving Skills

Why Problem Solving Skills?

As software developers, we are in the business of solving problems. Even if you know how to write a test, if you don't have problem solving skills then you will not be able to complete coding a solution for a given problem.

Albert Einstein said:

If I had an hour to solve a problem and my life depended on the solution, I would spend the first 55 minutes determining the proper question to ask, for once I know the proper question, I could solve the problem in less than five minutes.

Albert Einstein said he would spend 55 minutes defining the problem and alternatives and 5 minutes solving it. So this leads us to the next question.

How Does Problem Solving Skills Fit into TDD?

In 1945, George Polya wrote a book 'How to Solve It - A New Aspect of Mathematical Method.' This book discusses the problem solving in the context of mathematics. The process he describes is applicable to software development.

The four phases he describes in his book are:

1. Understand the Problem
2. Devise a Plan
3. Carry Out the Plan
4. Look Back

Understand the Problem

- Read the verbal statement of the problem
- Draw a figure
- Draw a table

Devise a Plan

We have a plan when:

- We know what needs to be done to solve the problem
- Conceive the idea of the solution
- The plan gives a general outline

Carry Out the Plan

The details fit into the outline

Look Back

- Reconsider and reexamine the completed solution
- We can improve any solution
- We can improve the understanding of the solution

These four phases relate to each other in the following way:

- Understand the Problem ⇨ Problem Domain Analysis
- Devise a Plan ⇨ Solution Domain Analysis
- Execute the Plan ⇨ Write the Test
- Look Back ⇨ Refactor

Why do we Need to Separate these Four Phases?

In his book 'The First 20 Hours - How to Learn Anything Fast', Josh Kaufman says that in order to acquire a new skill, you must break it down into sub-skills.

So the sub-skills required to do TDD are:

1. Problem Domain Analysis
2. Solution Domain Analysis
3. Designing Test Cases
4. Writing Tests First
5. Refactoring

We can discuss and practice these sub-skills independently of each other. This also helps when you get stuck during a TDD session. You can always step back and see which phase can help you get past an obstacle.

We do problem domain analysis to gain an understanding of the problem. We do solution domain analysis to devise a plan. We test our plan by writing the test first and derive the solution using tests. We look back in the refactor step.

How Do You Analyze the Problem?

Before you start writing any code, you can apply the problem solving skills that I explain in more detail in my Test Driven Development in Ruby course. Visit www.rubyplus.com for more details.

How Much Analysis is Enough?

We need to avoid analysis paralysis. You apply the tools that I teach in my course just enough to give you a big picture view and confidence to start writing the test first.

Basics of Test Driven Development

Designing Test Cases

Why Design Test Cases?

We need to design test cases before we start writing tests to avoid impasse during test-driven session. We want to minimize the red time. What is red time? It is the time taken to fix errors and failing tests. We expect some time in red. But we need to minimize the time spent in red. Why? Because we want to keep making progress whether it is learning something about the code or finishing a feature. Ideally, the only time the red is acceptable is when the test is failing for the right reason.

How Many Test Cases Do We Need?

TDD's view of testing is pragmatic. In TDD, the tests are a means to an end. The end being the code in which we have great confidence. My belief is that confidence is a subjective thing and it varies from one developer to another. It cannot be measured. So here is a useful checklist that you can use:

- Positive Case
- Negative Case
- Bad Inputs
- Boundary Conditions

This is a systematic way to make sure you have sufficient number of tests instead of depending on confidence.

What Should Be the Sequence of Tests?

- Starter Test
- Next Test
- Story Test

What is a Starter Test?

Which test should you start with? Start by testing a variant of an operation that doesn't do anything. This is usually the degenerate case. We need to focus on solving one problem at a time. If we begin with a realistic test, it will leave us too long without feedback. Red-Green-Refactor loop should be in minutes.

You can shorten the loop by choosing inputs and outputs that are trivially easy to discover. If you are familiar with the problem and confident that you can get it working quickly, then you can write a realistic test.

What is a Next Test?

Which test should you pick next from the list? Pick a test that will teach you something and you are confident you can implement. Each test should represent one step towards your overall goal.

What is a Story Test?

You end with a Story Test. This is the acceptance test. We know we are done when this test passes. The code is generic enough and solves the entire problem.

To learn about how to design test cases, check out my course 'Test Driven Development in Ruby' at www.rubyplus.com.

Assertion

What is an Assertion?

How do you check that tests worked correctly? Use boolean expressions to automate the human judgment about whether the code worked. When the assertion evaluates to true, it means the test worked. We need informative error message for all assertions.

How Many Assertions Can You Use in a Test?

Usually one assertion per test. The test should be very focused and test only one thing. You can break this rule by using multiple assertions but when that test fails, it should fail for only one reason. No other test should fail for the same reason. If we follow this rule, we will be able to quickly fix the problem. Because we can quickly track the cause of the problem.

Canonical Test Structure

According to the dictionary, the term canonical is defined as : relating to a general rule or standard formula. In the context of writing a test, there are three steps, they are:

1. Given
2. When
3. Then

The Given is the first step. This is the precondition. The system is in a known state. The When is the second step. We exercise the system in this step. Then is the third step. This is the post-condition. We check the outcome is as expected in this step.

Instead of thinking about : How do I write a test? Ask yourself the following questions:

1. What is the given condition?
2. How do I exercise the system under test?
3. How do I verify the outcome?

The answers to these questions will help you write the test. These questions correspond to each step in the Canonical Test Structure.

For example, if you have a class called Car, you need to have fuel in order to drive the car. The given condition in this case is that it has fuel. The drive() is the behavior you are testing. So, you invoke drive() method on the instance of a car in order to exercise the system under test. In our case the system under test is the car. When you drive, you expect to travel. So, you can verify the outcome by checking the distance traveled in the odometer.

Minimal Implementation

Why Minimal Implementation?

We force ourselves to write only enough code to make the test pass. Because our goal is simplicity. We want to prevent unnecessary complexity.

Why do we Aim for Simplicity?

Because it will be easy to maintain. In a maintainable system, tests will be easy to understand, modify and extend. Keeping your code ready for unexpected changes is about simple design. The only thing you can bet on tomorrow is change.

Ways to do Minimal Implementation

In practice, there are two ways to help you write minimal implementation. The first way is to do pair-programming, your

partner might suggest a simpler solution. The second way is to use Fake It Till You Make It.

The idea is simple, you don't think through every possible edge case in your head up-front but rather let the tests tell you what's needed for each step along the way.

Getting it Right

Let's take a look at some common mistakes made by developers. Here is list of common mistakes for each step of the TDD.

Quickly Add a Test

1. Not picking the simplest test case as the first test.
2. Not picking the next simplest test case as the second test and so on.
3. The need for writing a complex test scenario.

Run All Tests and See the New One Fail

1. Do not watch the test fail. Directly implement the feature.
2. Not mutating the code when the test passes without failing.

Why is not watching the test fail a mistake? This is a mistake because when you write code and then run the test, you don't know whether the new code you added is the reason for the test passing. So by making sure that the test fails before you write the code, you can be certain that the new code you added is responsible for making the test pass.

Make a Little Change

Another common mistake is not implementing the simplest thing that makes the test pass. This is the step that is focused only on 'that works'.

Run All the Tests and See them All Succeed

Run only the current failing test. Why is this a mistake? Why run all the tests? We run all the tests to make sure we have not broken anything. We also get a big picture view of the evolving design when we run all the tests and it passes.

Refactor to Remove Duplication

1. Forget the refactoring step.
2. Do not refactor the test code for readability.
3. Refactoring when in red state.
4. Refactor some other piece of code while working on a test.

Refactoring step is focused on design. This is the step that is focused on 'clean code'.

Common Beginner Mistakes

1. Code reflecting the data set used in tests.
2. Multiple assertions in a test.
3. Redundant tests.
4. Overlapping tests.
5. Forgetting to test negative cases and boundary values.
6. Testing too many things in one test.

7. Not updating the tests to reflect the current understanding of the system.

The data set you pick must be minimal and drive the evolution of code to become more generic. Robert C Martin says:

As the tests get more specific, the code gets more generic.

We need to keep this in mind as we write tests to drive the design.

Techniques in TDD

Introduction

We will see three different techniques for Test Driven Development. They are:

1. Obvious Implementation.
2. Fake it Till You Make It.
3. Triangulation.

Obvious Implementation

In Obvious Implementation, we type in the real implementation, if it's obvious and can quickly make the test pass. We use this technique to implement simple operations.

Fake it Till You Make It

You hard code constants to make the test pass, gradually generalizing the code using variables.

When you have a failing test, the first implementation returns a constant. Once the test is passing, we gradually transform the constant into an expression using variables. This approach can teach you something that you don't know. If you did not properly implement the test, you can fix the test. So fake implementation can teach you that written test is wrong. This avoids investing your time in the real solution to find out.

There are two positive effects that make this technique powerful:

1. Psychological
   The green bar feels good. We know where we stand when we are in green state. We can refactor with confidence.

2. Scope Control
   We avoid imagining future problems. Starting with one concrete example and generalizing from there prevents you from prematurely confusing yourself with extraneous concerns. You can do a better job of solving the immediate problem because you are focused. When you implement the next test case, you can focus on that one, knowing that the previous test is guaranteed to work.

There are two core things that we need to pay attention to when using this technique:

1. Start with the simplest implementation possible, which usually is returning a literal constant. Then gradually transform the code of both test and implementation using variables.
2. When doing it, rely on your sense of duplication between test and fake implementation.

Triangulation

In mathematics, Triangulation is the process of determining the location of a point by measuring angles to it from known points at either end of a fixed baseline. If two receiving stations at a known distance from each other can both measure the direction of a radio signal, there is enough information to calculate the range and bearing of the signal. This calculation is called triangulation.

By analogy, when we Triangulate, we only generalize code when we have two or more examples. We briefly ignore the duplication between test and model code. When the second example demands a more general solution, then and only then do we generalize.

Triangulation is the most conservative way to drive abstraction with tests. Because it involves the tiniest possible steps to arrive at the right solution. There are two characteristics : indirect measurement and using at least two sources of information at the core of TDD Triangulation. Basically, it says:

1. Indirect Measurement
   Derive the design from few known examples of its desired external behavior by looking at what varies in these examples and making this variability into something more general.

2. Using at least two sources of information
   Start with the simplest possible implementation and make it more general only when you have two or more different examples. These examples are tests that describe the desired functionality for specific inputs. Then new examples can be added and generalization can be done again. This process is repeated until we reach the desired implementation. Robert C. Martin developed a maxim on this, saying :

   As the tests get more specific, the code gets more generic.

Usually, when TDD is showcased on simple examples, triangulation is the primary technique used, so many novices mistakenly believe TDD is all about triangulation.

You can use Triangulation when you are not sure about the correct abstraction. It can also be used when you are not sure how to refactor. Kent Beck says:

If I can see how to eliminate duplication between code and test and create the general solution, I just do it. However, when the design thoughts aren't coming, triangulation gives you a chance

to think about the problem from a slightly different direction. What axes of variability are you trying to support in your design? Make some of them vary and the answer may become clearer.

These techniques are illustrated using coding demos in my course 'Test Driven Development in Ruby'. Visit www.rubyplus.com for more details.

About the Author

Bala Paranj

Bala Paranj has a masters degree in Electrical Engineering from The Wichita State University. He has been working in the IT industry since 1996. He started his career as Technical Support Engineer and became a Web Developer using Perl, Java and Ruby. He has consulted for companies in USA, Australia and Jamaica in finance, telecommunications and other domains.

He has professionally worked as a developer using TDD and pair programming for startups. He is the founder of Silicon Valley Ruby Meetup. He has been organizing Ruby, Rails and TDD related events since 2007. He has run TDD Bootcamps and TDD tutorials for Silicon Valley Ruby Meetup members for more than 3 years.

The course is the result of the feedback from students who have taken his TDD bootcamps and tutorials. He published an article in JavaWorld in 1999 on Command Pattern. Java Tip 68: Learn how to implement the Command pattern in Java. He is the content creator for the Whizlabs OOAD test simulator. This is the exam simulator for students preparing for IBM 486 Object-Oriented Analysis and Design with UML. He is also the author of self-published book on Ruby on Rails, Rails 4.2 Quickly, Essential SQL, Essential Git and Test Driven Development in Ruby.