# A(N) (RE-)INTRODUCTION TO TEST-DRIVEN DEVELOPMENT

We're going to begin at the beginning. Some of you will have the impulse to skip this document.

Please don't.

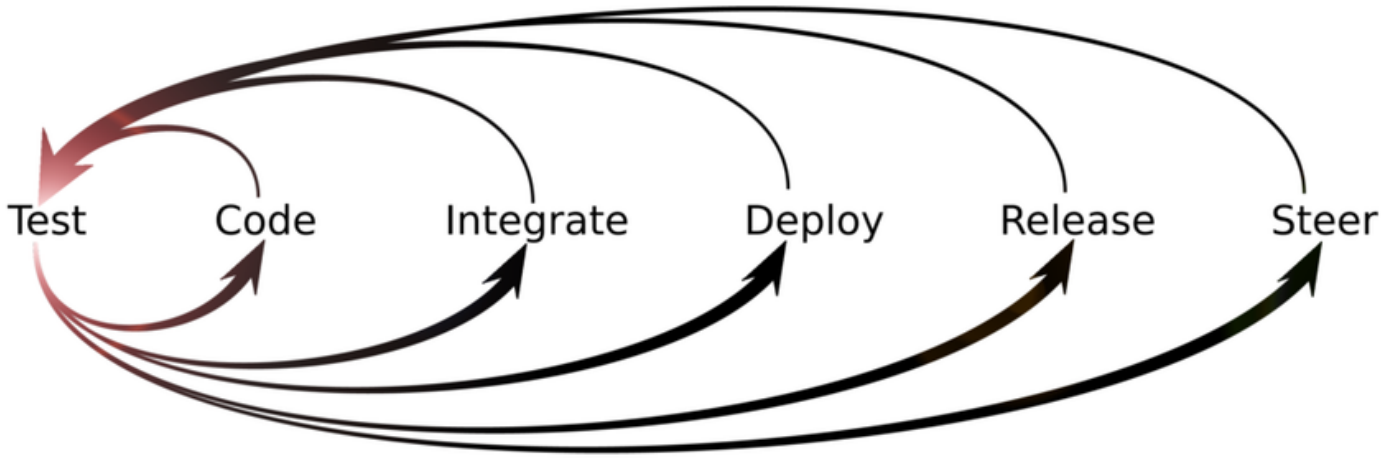
## OVERVIEW

I ask you to do the following:

1. Read this entire article. I've shortened it significantly since the first time I wrote it.
2. Do the exercises.
3. Read some of the references.
4. Ask questions in the comment thread for this lesson.

## LET'S TAKE IT FROM THE TOP

In the beginning, we had this:




This was the first authoritative diagram of the emerging discipline of test-driven development. Not much has changed in the decades since then, except that TDD has narrowed its focus to the programmer's experience of **incrementally delivering features while guiding the design to evolve**. This diagram describes what we now think of as the heart of Extreme Programming.

You'll notice something missing from this diagram: the word "design". We mostly assumed that design happened while writing code. It happened continuously. That's the focus point of this course: **TDD as a way to do incremental, iterative, evolutionary design**. Not only does the programmer design the system incrementally, but they gradually become increasingly confident in deferring design decisions until they absolutely have to be made. This makes the experience of delivering features significantly more pleasant for everyone.

So how does one *do* TDD?

## THE WORLD'S SIMPLEST INSTRUCTIONS FOR TDD

I didn't invent them. Ward Cunningham codified them in his wiki.

> Where you code, alternate these activities:
>
> 1. add a test, get it to fail, and write code to pass the test
> 2. remove duplication

At its core, this is enough. Everything else we discuss in this course is a refinement of these two instructions. In fact, if you follow these two instructions diligently and pay attention to what happens to you when you do, then you don't need this course.

Too late! You're already here.

As it turns out, **the truly interesting and valuable parts of TDD** lie in the various consequences of following these two instructions. And those interesting and valuable parts are not entirely obvious to everyone all the time. And it depends how you practise.

Good news: in this course, you'll see examples of how to practise, what to practise, and what it will mean. If you need more, join The jbrains Experience to receive personal guidance on how to navigate the details hidden within these two instructions.

In particular, as you progress through this course, you might notice some of the following:

- Increased confidence that your code behaves the way you intended.
- Lower cost of failure due to taking smaller steps.
- Changing your perspective from "I just need to fix a few bugs" to "I just need to add one or two more working paths".
- Writing less code while delivering more features.
- Constantly cleaning up little messes, but rarely being distracted with big messes.
- Changing code confidently, gradually, and with less stress.
- Changing code more aggressively.
- Making steadier progress.

You won't notice all these things right away, but with continued practice, you'll probably notice most of them eventually. You might even come to take some of them for granted!

# START!

Before you write your first failing test, you need some tools. Not much, but some.

To start, you'll need to be able to run one failing test. Next, if you can rerun all the tests with a single command, that's better. It's even better if you can rerun all the tests every time you save a change to your code.

You'll also want to be able to commit your changes to version control any time the tests all pass. As of 2021, most people use `git`, but any lightweight (fast!) version control system would work very well: `darcs`, `bzr`, `hg`, whatever makes you happy.

# EXAMPLE TOOLS

When I work in Java, I tend to use these tools:

- IntelliJ IDEA
- JUnit
- git + gitlab or github

- a pen, a stack of index cards, and maybe a notebook

When I work in Purescript, I tend to use these tools:

- kakoune
- Purescript-spec
- https://pursuit.purescript.org/ because I always need to look up library functions
- git + gitlab or github
- a pen, a stack of index cards, and maybe a notebook

And recently, at least in late 2021, I have grown fond of Zora for writing and running tests in (plain) Javascript. (Of course, the tools in Javascript change every 18 minutes, so this will be out of date by the time you read it.) And I think rspec remains the standard for Ruby.

You have enough tools if you can create a new project, write a failing test, then run it and watch it fail. Take a few moments now and set that up in your favorite programming environment.

# EXERCISE: ADD FRACTIONS

As a warm up, trying adding some fractions. With this exercise, **you'll start to establish the habit of writing the test first**. If you don't know how to add fractions, then search the web for an explanation. Wikpedia will probably give you everything you need. You'll build a library function that adds fractions as exact values—no converting to floating-point numbers! For example: 1/8 + 3/8 = 1/2.

And **that's your first test**!

## A LITTLE BDD: TALKING IN EXAMPLES

When discussing a feature, product directors (also known as product owners, although I don't think they truly *own* the product) and programmers need to be able to clarify what they expect from the system. **I encourage them to talk in examples**. Examples clarify. They are concrete. And they can become automated tests, if you want.

That's why I started with an example: 1/8 + 3/8 = 1/2. This is a complete example, because it describes inputs to an action and the expected result. You can turn this into an automated test in your favorite programming language.

```
// In Javascript, using Zora
const fraction = (numerator, denominator) ⇒ ...;

test("a simple example", (t) ⇒
     t.eq(fraction(1, 8).plus(fraction(3, 8)), fraction(1, 2));
);
```

Automating the test is by far the least important part of this work. What matters is **articulating the example so that everyone agrees on what we want the system to do**.

Let me jump to the end. Here are some constraints and hints for adding fractions:

- Exact values only; no converting to floating-point numbers. 1/4, not 0.25.
- Make sure that equal values are equal: 1/2 = 4/8 = 89/178 = -56/-112. (And don't forget `hashCode()`!)

- Express fractions as *improper* and not *mixed*. That means 7/2 and not "3 and 1/2".

That's it. It's certainly enough to get started.

# DO THIS NOW!

Ship a module that adds fractions exactly. If you want to go further than that, do the other three basic arithmetic operations: subtract, multiply, divide. Or write a parser for a calculator so that the string `"1/4 + 1/2"` returns the value 3/4.

“Ship”?

Yes. Ship a module that I could download and use in my application. Do whatever the means for your programming environment: package a JAR file, build an assembly, publish a gem, register an npm package… whatever it takes. I should be able to include, import, or require your library and use it to add fractions.

In fact, maybe even start by shipping an empty package/library/whatever, then teach it to add fractions.

Once you can ship and empty package, make a test list.

# TEST LIST

**Your mind is probably racing** with ideas of tests that you will need to write. Good! Write them down. I tend to write them down quickly onto an index card. It’s enough to write just enough words or numbers or symbols to remind you of the test you need to write. Sometimes you’ll write the exact test (“1/2 + 1/2 = 1”) and sometimes you’ll write a few words (“reject 0 denominator”). Whatever is rushing to your head, write it down.

Your goal, when writing a test list, is to **get ideas out of your head quickly**, so that afterwards you can decide where to start, then write a failing test. That’s it.

You’re not building a perfect plan. You’re not deciding in advance all the tests you’ll need. You’re getting ideas out of your head so that you can **focus on one test at a time**. That’s it.

# SOME HINTS

I tend to work in stages when I write my test list:

1. Get ideas out of my head.
2. Work towards the simplest test possible.
3. Add examples of what can go wrong.

Try it!

For example, I thought of 1/8 + 3/8 = 1/2, but then simpler than that would be 1/5 + 2/5 = 3/5, because the denominators are the same, but the most natural way to compute the answer already produces 3/5 without worrying about the fact that 1/2 = 4/8. It’s a simpler test, because it requires less work to make it pass. I can simply compute 1 + 2 = 3, copy the 5, and get 3/5.

After I get ideas out of my head, **I look for simpler tests until I find the simplest possible test**. And usually, I start writing code with the simplest possible test. (What’s the simplest possible test for adding fractions?!)

# VERSION CONTROL

When you write code, do this:

1. Write a test, run it, and watch it fail.
2. Make the test pass *and keep the previous tests passing*.
3. **Commit your changes to version control.**

3. **Commit your changes to version control**.
4. Repeat until there are no more failing tests to think about.

By committing your changes to version control, you have **the world's best undo button**. If you make a mistake, you can easily roll back to the last time everything worked. You are always free to commit changes when all the tests pass. I recommend *against* committing code when the tests fail. I would like to have the confidence that **I can check out any version of the code base and it will work**. It might not do everything the customer asked for (yet), but it will work.

You don't have to do this, but it helps.

## QUICK SUMMARY

1. Follow the steps of TDD strictly and diligently. Go slowly. Pay attention to what's happening. **Pay attention to how you feel about what's happening**. When you feel something strange, ask for help. (This is where the The jbrains Experience is especially valuable.)
2. Start with a Test List, then pick one failing test and start following the steps. Get your ideas out of your head first. **Don't weigh yourself down with distracting thoughts** about what you might need to do next. Write it down, get it out of your head, and focus on one thing.
3. When you can't think of any more failing tests, you're done. **Stop**. There is nothing more to do, at least for now.

## ONE MORE THING

Some programmers have trouble managing their time. If this is you, then consider working in short episodes of 25-40 minutes each. Set a timer, start working, let yourself focus, and then when the timer sounds, stop. Write down whatever is in your head, including which test is the next one to make pass, then **walk away from the keyboard**. Pour some coffee or tea or whatever will make you feel better. Come back and continue when you're ready. After practising this for several months, I noticed that I could "get into the flow" much more quickly than I used to be able to do. Maybe you'll experience the same.

## GO!

Add some fractions! Enjoy! If you're also in the The jbrains Experience, then share your experiences and impressions in Chat and in the Forum whenever you like. Good luck and have fun.