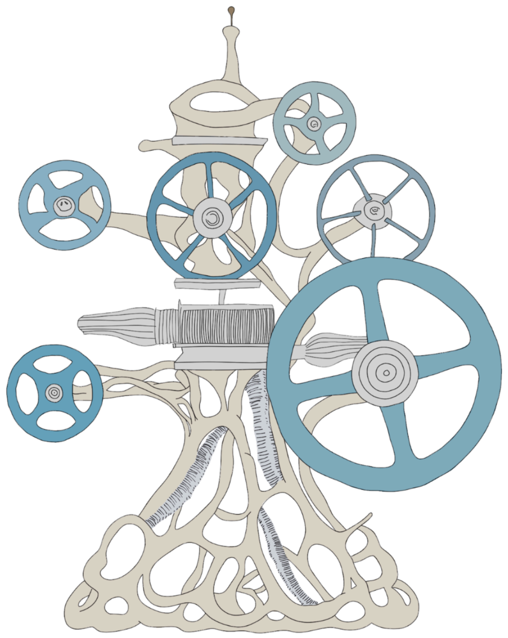
Regular Expressions Machinery

The Illustrated Guide



Staffan Nöteberg

To inquire about booking the author for podcasts, training sessions, and speaking engagements at conferences, please contact him directly at staffan.noteberg@rekursiv.se

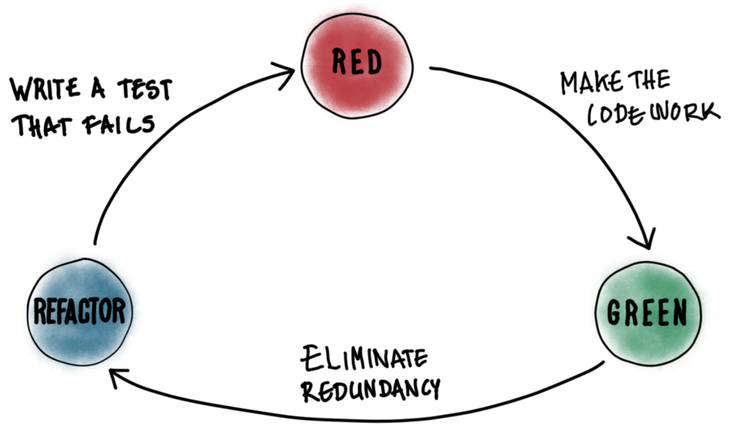
PDF ISBN:	978-91-989983-0-6
EPUB ISBN:	978-91-989983-1-3
Book Version:	V1.0—January, 2025

Part IV: Test-Driven Regex Development

In this part, you'll learn about a technique called test-driven development (TDD) and how you can benefit from it when developing and maintaining your regexes. TDD helps you write better code, especially when you work with regexes.

With TDD, you ensure that your regexes execute as intended—both today and in the future—and that they are easy to maintain and update. It's a method that I think you'll find very useful, and it'll save you a lot of time and headaches in the long run.

Wishful Thinking and Test-Driven Development



Wishful thinking in programming involves assuming you already have the functions or code snippets to solve parts of a problem, even if you haven't written them yet. For example, you might write calls to an API that doesn't exist. It's a mind hack that gets you to "begin with the end in mind." Instead of getting bogged down in implementation details, you can look at the bigger picture and focus on what the client code of the yet-to-be-written code really needs.

Imagine you need to write a regex that validates that a string seems to be an email address. Even though you won't follow everything in the RFC 5322 standard, there are still several things to check:

- There is exactly one `@` symbol.
- There is at least one period to the right of the `@` symbol.
- There are alphanumeric characters to the left of the `@` symbol.
- Etc.

This creates a list of requirements. For each requirement, we'll write down examples of correct as well as incorrect email addresses. Let's start with examples of "There is exactly one `@` symbol."

- `abc@abc.com` (pass)
- `abc.abc.com` (fail)
- `abc@abc@com` (fail)

Great! Now, let's test these examples in IRB. First, we must make sure that the unit test framework is present.

```
require "test/unit/assertions"
include Test::Unit::Assertions
```

With the test framework, we can test the truth of statements by calling the `assert` function. This is where wishful thinking comes in. We simply pretend—that is, "wish"—that the `is_email` function already exists and that it validates that its input has exactly one `@` symbol.

```
assert is_email('abc@abc.com') #=> undefined method `is_email'
```

Ouch! Or ... good! The message "`undefined method 'is_email'`" tells us that `is_email` doesn't exist, which we already knew. We start by writing the simplest possible `is_email` function to get rid of the "`undefined method`" error.

```
def is_email(email) false end
```

Now we can run our test again.

```
assert is_email('abc@abc.com') #=> <false> is not true
```

Since `is_email` always returns `false`, the test fails and gives us the logic tautology "`<false> is not true.`" The easiest way to make it pass is through a boolean makeover: always return `true` instead of always return `false`.

```
def is_email(email) true end
```

However, now that the function always returns `true`, checking an inadequate address example with `assert_false` will cause problems.

```
assert is_email('abc@abc.com') #=> nil
assert_false is_email('abc.abc.com') #=> <false> expected but was <true>
```

Our positive test—the old one—passed, but the negative test—the new one—fails because `is_email` always returns `true`. Sorry, IRB! We promise to do better by writing an improved implementation of the function.

```
def is_email(email) /.*@.*/ === email end
```

This regex checks if the input string contains an `@` symbol. Now both tests should pass.

```
assert is_email('abc@abc.com') #=> nil
assert_false is_email('abc.abc.com') #=> nil
```

Wow! Both tests pass—that is, return `nil`— and we've just developed a function embryo using wishful thinking and TDD. By isolating our regex into its own function, we can test it—and even regression-test it—without other potential sources of error.

Let's continue with the third example.

```
assert_false is_email('abc@abc@com') #=> <false> expected but was <true>
```

When we adjust `is_email` to make our third test pass, the first two tests must still pass. We can modify the regex to ensure it matches strings with exactly one `@` symbol. Anchors and negated character classes do the trick.

```
def is_email(email) /^[^@]*@[^@]*$/ === email end
assert is_email('abc@abc.com') #=> nil
assert_false is_email('abc.abc.com') #=> nil
assert_false is_email('abc@abc@com') #=> nil
```

Yes! All three tests pass.

In its purest form, TDD consists of three steps, often called Red–Green–Refactor.

1. **Red:** Write a test that assumes functionality you don't have yet.
2. **Green:** Write the code that makes the test in step 1 pass.
3. **Refactor:** Improve the structure of the old and new code.

If it feels backward to start by writing a test for code that doesn't exist—that is, step 1—you can think of the test as a requirement specification.

Two Takeaways

- In wishful thinking, you assume you have the necessary functions or code snippets to solve parts of a problem, even if you haven't actually written them yet.
- TDD is a method in which you start by writing tests for code that doesn't exist yet, which helps drive a minimal API with needs-driven implementations.

Get a Grip on the Regex Machinery

To effectively use regular expressions, you need to understand how the machinery works under the hood. It's about taking control of the search process, controlling how the pattern is matched, and thus getting both faster and more accurate results.

In this illustrated guide, you gain precisely that understanding.

You can even get started without any prior knowledge of regular expressions. Before you know it, advanced tools like reluctant, lookbehind and nondeterministic finite automata will be at your fingertips as you write efficient and elegant regexes with ease.

This book presents complex concepts in a simple and visual way, with clear examples and practical applications. Whether you are a programmer, data analyst, or just want to get better at text processing, this book will take your knowledge to the next level.

Staffan Nöteberg is a bestselling author passionate about helping people become more efficient and focused. He is the author of these popular books:

- Pomodoro Technique Illustrated: The Easy Way to Do More in Less Time
- Monotasking: How to Focus Your Mind and Be More Productive
- Guiding Star OKRs: A New Approach to Setting and Achieving Goals

With a background in software development and an interest in productivity, Staffan combines his expertise with an ability to explain complex topics in an easy-to-understand way.

PDF ISBN: 978-91-989983-0-6

9 789198 998306

EPUB ISBN: 978-91-989983-1-3

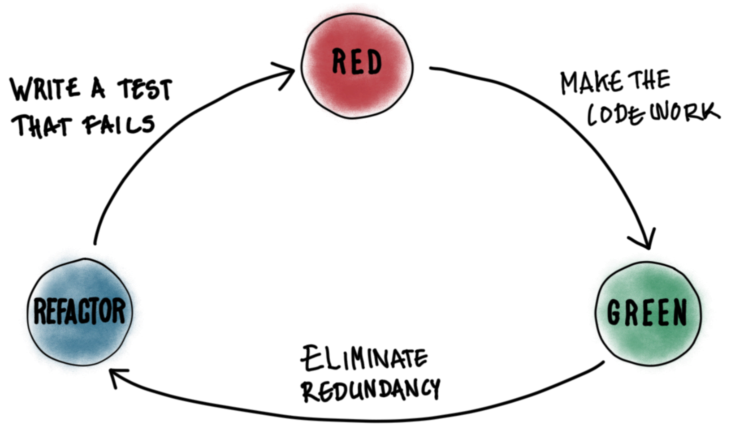
9 789198 998313

Part IV: Test-Driven Regex Development

In this part, you’ll learn about a technique called test-driven development (TDD) and how you can benefit from it when developing and maintaining your regexes. TDD helps you write better code, especially when you work with regexes.

With TDD, you ensure that your regexes execute as intended—both today and in the future—and that they are easy to maintain and update. It’s a method that I think you’ll find very useful, and it’ll save you a lot of time and headaches in the long run.

Wishful Thinking and Test-Driven Development



Wishful thinking in programming involves assuming you already have the functions or code snippets to solve parts of a problem, even if you haven't written them yet. For example, you might write calls to an API that doesn't exist. It's a mind hack that gets you to "begin with the end in mind." Instead of getting bogged down in implementation details, you can look at the bigger picture and focus on what the client code of the yet-to-be-written code really needs.

Imagine you need to write a regex that validates that a string seems to be an email address. Even though you won't follow everything in the RFC 5322 standard, there are still several things to check:

- There is exactly one `@` symbol.
- There is at least one period to the right of the `@` symbol.
- There are alphanumeric characters to the left of the `@` symbol.
- Etc.

This creates a list of requirements. For each requirement, we'll write down examples of correct as well as incorrect email addresses. Let's start with examples of "There is exactly one `@` symbol."

- `abc@abc.com` (pass)
- `abc.abc.com` (fail)
- `abc@abc@com` (fail)

Great! Now, let's test these examples in IRB. First, we must make sure that the unit test framework is present.

```
require "test/unit/assertions"
include Test::Unit::Assertions
```

With the test framework, we can test the truth of statements by calling the `assert` function. This is where wishful thinking comes in. We simply pretend—that is, "wish"—that the `is_email` function already exists and that it validates that its input has exactly one `@` symbol.

```
assert is_email('abc@abc.com') #=> undefined method `is_email'
```

Ouch! Or … good! The message "`undefined method 'is_email'`" tells us that `is_email` doesn't exist, which we already knew. We start by writing the simplest possible `is_email` function to get rid of the "`undefined method`" error.

```
def is_email(email) false end
```

Now we can run our test again.

```
assert is_email('abc@abc.com') #=> <false> is not true
```

Since `is_email` always returns `false`, the test fails and gives us the logic tautology "`<false> is not true`." The easiest way to make it pass is through a boolean makeover: always return `true` instead of always return `false`.

```
def is_email(email) true end
```

However, now that the function always returns `true`, checking an inadequate address example with `assert_false` will cause problems.

```
assert is_email('abc@abc.com') #=> nil
assert_false is_email('abc.abc.com') #=> <false> expected but was <true>
```

Our positive test—the old one—passed, but the negative test—the new one—fails because `is_email` always returns `true`. Sorry, IRB! We promise to do better by writing an improved implementation of the function.

```
def is_email(email) /.*@.*/ === email end
```

This regex checks if the input string contains an `@` symbol. Now both tests should pass.

```
assert is_email('abc@abc.com') #=> nil
assert_false is_email('abc.abc.com') #=> nil
```

Wow! Both tests pass—that is, return `nil`— and we've just developed a function embryo using wishful thinking and TDD. By isolating our regex into its own function, we can test it—and even regression-test it—without other potential sources of error.

Let's continue with the third example.

```
assert_false is_email('abc@abc@com') #=> <false> expected but was <true>
```

When we adjust `is_email` to make our third test pass, the first two tests must still pass. We can modify the regex to ensure it matches strings with exactly one `@` symbol. Anchors and negated character classes do the trick.

```
def is_email(email) /^[^@]*@[^@]*$/ === email end
assert is_email('abc@abc.com') #=> nil
assert_false is_email('abc.abc.com') #=> nil
assert_false is_email('abc@abc@com') #=> nil
```

Yes! All three tests pass.

In its purest form, TDD consists of three steps, often called Red–Green–Refactor.

1. **Red:** Write a test that assumes functionality you don't have yet.
2. **Green:** Write the code that makes the test in step 1 pass.
3. **Refactor:** Improve the structure of the old and new code.

If it feels backward to start by writing a test for code that doesn't exist—that is, step 1—you can think of the test as a requirement specification.

Two Takeaways

- In wishful thinking, you assume you have the necessary functions or code snippets to solve parts of a problem, even if you haven't actually written them yet.
- TDD is a method in which you start by writing tests for code that doesn't exist yet, which helps drive a minimal API with needs-driven implementations.