Legacy System:

A legacy system is outdated computing software and/or hardware that is still in use. The system still meets the needs it was originally designed for, but doesn't allow for growth. What a legacy system does now for the company is all it will ever do. A legacy system's older technology won't allow it to interact with newer systems.

The reasons are varied as to why a company would continue to use a legacy system.

Investment: Although maintaining a legacy system is expensive over time, upgrading to a new system requires an up-front investment, both in money and manpower.

Fear: Change is hard, and moving a whole company —or even a single department — to a new system can inspire some internal resistance.

Difficulty: The legacy software may be built with an obsolete programming language that makes it hard to find personnel with the skills to make the migration. There may be little documentation about the system and the original developers have left the company.

The key points of Working Effectively with Legacy Code (Michael Feathers' key points)

1. First, add tests, then do your changes

The challenge with changing existing code is to preserve the existing behavior. When code is not tested, how do you know you didn't break anything? You need **feedback**. Automated feedback is the best. Thus, this is the first thing you need to do: write the tests.

2. Adding tests: the Legacy Code dilemma

Before you change code, you should have tests in place. But to put tests in place, you have to change code. This is the paradox of Legacy Code! You should perform minimal, safe refactorings. Change as little code as possible to get tests in place.

3. Identify Seams to break your code dependencies

Adding tests on the existing code can be challenging. the code you want to test can't run because it needs *something* hard to put in the test. Sometimes it's a database connection. Sometimes it's a call to a third-party server. Sometimes it's a parameter that's complex to instantiate. Usually, it's a complex mix of all that. To test your code, you need to **break these dependencies** in the tests.

JS Copy

```
export class DatabaseConnector {
  // A lot of code…

  connect() {
    // Perform some calls to connect to the DB.
  }
}
```

JS Copy

```
class FakeDatabaseConnector extends DatabaseConnector {
  connect() {
    // Override the problematic calls to the DB
    console.log("Connect to the DB")
  }
}
```

4. Unit tests are fast and reliable

Michael Feathers gives a clear definition of what is NOT a unit test.

In short, your test is not unit if:

- it doesn't run fast (< 100ms / test)
- it talks to the Infrastructure (e.g. a database, the network, the file system, environment variables…)

Write a maximum of tests that have these 2 qualities. How you call them doesn't matter.

5. Characterization tests

   Before you can refactor the code, you need tests. But writing these tests can be challenging. Especially when code is hard to understand.

   "A characterization test is a test that characterizes the actual behavior of a piece of code."

   Instead of writing comprehensive unit tests, you capture the current behavior of the code. You take a snapshot of what it does.

1. THE SPROUT TECHNIQUE

1. Create your code somewhere else.
2. Unit test it.
3. Identify where you should call that code from the existing code: the *insertion point*.
4. Call your code from the Legacy Code.

2. THE WRAP TECHNIQUE

When the change you need to do should happen before or after the existing code, you can also *wrap* it.

1. Rename the old method you want to wrap.
2. Create a new method with the same name and signature as the old method.
3. Call the old method from the new method.
4. Put the new logic before/after the other method call.

That new logic, you can test.

Why? Because the old method is a Seam you can alter in the tests.

Let me remind you of the “how to tackle Legacy Code” recipe:

1. Identify change points (Seams)
2. Break dependencies
3. Write the tests
4. Make your changes
5. Refactor