There are many definitions of legacy code, such as the explanation from Wikipedia that *legacy code is source code that relates to a no-longer supported* [1] *or manufactured operating system or other computer technology*. Back in 2002, Michael Feathers introduced a definition of legacy code as code without tests in his article *Working Effectively with Legacy Code* [2]. Here, we summarize that legacy code is the source code of an earlier version of the system written by the previous developers, or the code that lacks the corresponding complete documentation and testing. In general, these codes are difficult to handle and maintain but still have some reference value for the current development system. In real industrial software projects, writing legacy code can hit some important deadline due to a range of business reasons. So how to get started with legacy code and effectively modify and apply it is pretty important for project developers. For this purpose, our group interviewed relevant developers of a software technology company from America to learn about the application of legacy code in real projects.

In the interview, we learned that before using legacy code, programmers should first confirm the requirements of the project and fully understand the system development framework, business functions and database architecture. In most cases, project developers can draw diagrams to describe the internal connections amongst components of the system and determine which external systems the legacy component interacts with. Furthermore, developers should compare and analyze the technological difference between legacy systems and systems under development - e.g. which technology stacks no longer meet modern development standards and what sections of legacy codes need to be removed as soon as possible. Developers should take responsibility of what they build and maintain; in the modules they are responsible for, programmers should routinely identify potential segments of legacy code that needs to be modified or rewritten entirely. Often the best way to start on tackling a piece of legacy code is not to examine the code alone, but rather to hold a netting with the original owner. Another useful approach to understand legacy code is to create unit tests or analyze current ones. In real project development, such unit tests are often included in the larger and more important integration tests. The programmer can check if the legacy code could still pass the unit tests and obtain the expected result first. After making changes or commits to the legacy code, the programmer could simply run the same test cases to examine if the test still successfully runs. If the test fails, it means that the code after refactoring is not equivalent to the legacy code. At this point, the programmer should determine the reasons why the new code fails to satisfy the tests - especially if there are boundary cases that are missed, or if there are specific requirements that are not met.

However, in practice, there is no guarantee that unit tests will be created for legacy code. The main reason is that, on the one hand, for a legacy system without any automated testing, it usually means that its internal design coupling is very high. On the other hand, the development of legacy code often requires large-scale reconstructions to better maintain the system and make it easier to add new features. Based on this situation, even if unit tests are added to the system, subsequent reconstructions will cause the unit tests to be modified again. One way to solve this problem is to create component tests for legacy code. Compared with unit testing, component testing focuses on business behavior rather than code implementation details [3]. Therefore, when the details of the code implementation change, it will not be affected. For example, when testing a user-registered API, you can test that the response to the successful scenario of the registered API is correct and send a confirmation email to the user instead of providing multiple user name use cases to the API and testing which user names are legitimate (those should be covered by unit tests of the test user name validator).

As a developer, we need to be humble and respect legacy code and the developers who wrote it previously. Never judge legacy code or change it until we have taken the time to fully understand it. The best way to learn legacy code is to start at the user interface level, then back into the code [4]. For example, we can pick a single user flow, such as logging in, placing an order, writing a review. Go through the flow as an end user. Then starting with the user interface code to follow each step on back, all the way to the database. Besides, when fixing legacy code, programmers should make the minimum variable change [4] first. Prefer small, incremental improvements over wholesale re-writes or changes. This means programmers should make the least disruptive change that complete fixes the problem before attempting to clean and refactor any code [4].

Reference

[1] Samuel Mullen, ‘Legacy’ isn’t a bad word,

<https://samuelmullen.com/articles/legacy_isnt_a_bad_word/>

[2] Michael Feathers, Working Effectively with Legacy Code, Upper Saddle River, N.J.: Prentice Hall, PTR 2004

[3] Martin Fowler, component testing,

<https://martinfowler.com/articles/microservice-testing/#testing-component-introduction>

[4] Bill Sourour, How to conquer legacy code, <https://medium.freecodecamp.org/conquer-legacy-code-f9e23a6ab758>