We had a deep conversation with an ex-developer from Expedia advertising group, to see practically what software development strategies are adopted to help development work in big IT industries.

I. Legacy Code

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].

II. Continuous Delivery

Continuous Delivery is a popular strategy adopted by most modern software development industries. It is, as described by Jez Humble and Dave Farley (2010), the process of get changes into production or into the end users in a reliable way. Changes by developers include adding new features, change configuration, bug fixing, and they have to be broken down into smaller parts that can be delivered respectively [**?**]. Getting changes into production requires the code to be sustainable, which means that the code should always in a deployable state. To achieve this, it requires automated software building, testing, and deploying, but we only deliver what is potentially profitable and beneficial to the business. Adopting Continuous Delivery allows developers to get feedback from market in time, which improves software quality and releases better products.

In our case study of Expedia, the develop team mainly work under principle of Scrum, which is a development methodology for agile software development, requires a PO, a Scrum Master, and development team. It aims at responding to emerging requirements, and to adapt to evolving technologies and changes in market conditions [**?**].

As for project management part, some teams in Expedia rely completely on Jira, an issue and project tracking software, while others make a combination of Jira and Mingo, which is also a project management software. He introduced how development teams make use of Jira as an example. Since teams are facing various requirements every day, both from project details or software developers. Requirements contain a general problem description, an expectation for the problem, or a specification of outcoming product.

Different requirements can be categorized into epics, stories, or tasks, based on size and complexity of requirements in descent order. Each of epics, stories, and tasks is assigned to different Scrums, through the "tasking meeting", which allow developers understand the task size, what is requested and what is expected.

A fixed bi-weekly Sprint is also a component in Scrum in Expedia. Previously, because of the fact that Expedia website updates have a weekly release strategy, any bug fix and small changes have to be conduct at the end of that two weeks, and developers usually address everything discussed on Sprint within a small period just before the deployment, which is not efficient for product development. Specifically, He told that the deployment day for Expedia is Wednesday weekly, and there is a "cut-off time" on Tuesdays that every change made by that time will be deployed to the latest version of their website. This strategy has an obvious drawback that if some new features have to be added, or some emerging bug fix need to be conducted, which is so important that cannot wait until next deployment day, then it becomes a catastrophe. That bug needs to experience a complicated process of fixing, integration, approved by manager and project director, and finally build the implementation from head to tail before deployment. Normally small sub-deployment won't be approved by director, instead the team will perform some security tricks such as shut down that particular service for a week by switching flags.

After using continuous delivery, such bug fixing problem becomes a lot easier. Take their mobile web page for instance, developer only needs to push a change to the git repository and waits until that change to be deployed. The build pipeline can automatically monitor every commit that developer pushes and begin building. Afterwards, it will conduct automated tests that the team set in advanced, such as unit test, integration test, and squash test (simulating user clicks). Finally, it will perform automated deployment, and the change will be push to cloud and allow user to see the latest Expedia mobile page immediately. It is apparent that using changing into such software development strategy is potentially risky, as if someone's push causes small problems, it will directly deliver to user's hand and more severely affect functionality from other parts. Therefore, it requires every developer to bear a high responsibility that every single commit they push represents the entire Expedia company. Specifically, it requires the code in their hands should be production ready at any time and take ownership of what they are doing. There are also push review team to test code in an extremely careful manner before pushing to examine potential possibility about performance decay. Moreover, there are various monitoring parameters recorded on pipeline, such as webpage loading speed. If such parameters decrease dramatically after a particular push, there will be a caveat prompted to reminded that developer to revert if necessary. If every developer can be carefully enough and make sufficient test before push and every team works stably, then it is a highly efficient and robust developing system.

References

[1] S. Mullen, **'Legacy' isn't A Bad Word** https://samuelmullen.com/articles/legacy_isnt_a_bad_word/ *Date Accessed: 5 March 2019*

[2] M. Feathers, **Working Effectively with Legacy Code** http://www.netobjectives.com/system/files/WorkingEffectivelyWithLegacyCode.pdf *Date Accessed: 6 March 2019*

[3] M. Fowler, **Component Testing** https://martinfowler.com/articles/microservice-testing/#testing-component-introduction *Date Accessed: 6 March 2019*

[4] B. Sourour, **How to Conquer Legacy Code** https://medium.freecodecamp.org/conquer-legacy-code-f9e23a6ab758 *Date Accessed: 7 March 2019*