

Teamwork Does Work!

By Jim Putka

Introduction

In 1993, we at USA Group chartered a project to replace its 15-year old student loan guarantee system. The new system was named EAGLE II and recently entered production.

USA Group is driven by its software systems. As a consequence, software development is strategic to the company's success. We've had significant experience in creating and maintaining large systems. However, our experience in creating those previous systems left us looking for a better way.

If you've experienced:

- Friction between application developers and testers.
- Poor quality code delivered to test.
- Poor quality code in production.
- Knowing the what for improving software quality but not exactly the how.
- Wishing you knew how to sell this software quality thing to the organization at large.

Success is achieved through people. Large projects require large numbers of people working in teams. Successful teamwork is absolutely required. We've found that the solutions to problems above are rooted in successfully structuring the environment your teams operate within. What follows is our story of structuring the EAGLE II environment for successful software development teamwork.

Background

EAGLE II is large. We were genuinely surprised. It turned out to be 7.8 million lines of code. The count of individual components follows:

- 1460 programs
- 391 maps
- 317 copybooks
- 1035 procs
- 1024 JCL members

As stated above it is USA Funds' and other guarantors' student loan guarantee system. As you've probably guessed from the component types, EAGLE II is a mainframe system with on line and batch processes.

We conducted development work within a formal SDLC with defined phase work products. These defined work products were essential to progress in promoting teamwork and quality. A commercial CASE tool supported the development effort.

Goals, Objectives, and Strategies

The overall goal was to deliver a higher quality product with higher quality's attendant benefits. This goal gave rise to two process change objectives. The first was to test early and test often. The second was spread the responsibility for quality across the development organization. As it turned out, we implemented four strategies for reaching our goal through realizing our objectives. They follow below:

- OODA – how we sold the process change objectives
- The Gate – how we put in place one measure to share the responsibility for quality
- The Bridge – how we built teamwork while maintaining accountability
- The Washing Machine – the power of iteration

Let's now take a more detailed look at each one of these four strategies.

OODA

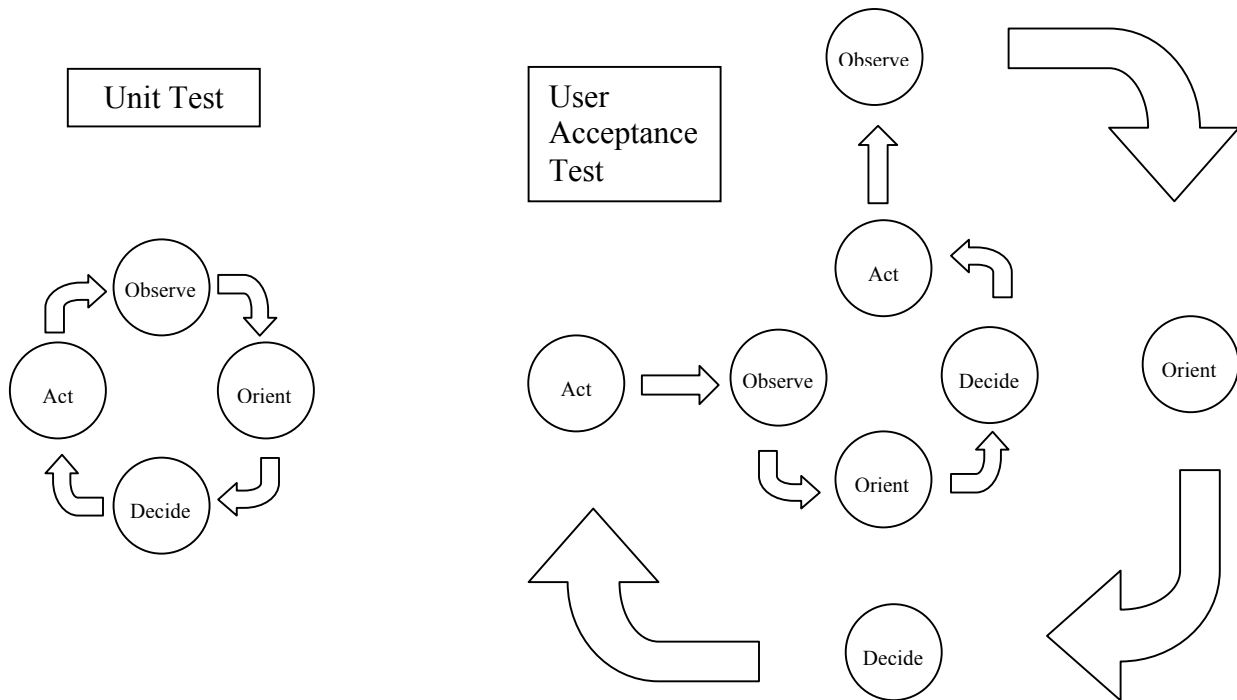
Change sometimes requires pain. We had at least one of the prerequisites. People believed change was needed for EAGLE II. People, however would not accept change without understanding the approach and benefits. To give change a reasonable chance to succeed, a critical mass of support needed to be built. A simple and visual explanation for testing early and often would need to be found.

Sometimes inspiration comes from when least expected. I happened to reading a book on the Desert Storm air war at this time. The book explained some of the US Air Force's history in developing the strategy and equipment employed in Desert Storm. One concept caught my attention – OODA. OODA is an acronym short for observe, orient, decide, and act. It is alternatively known as the Lewis cycle. During the Korean War OODA was developed to provide a conceptual understanding of fighter combat. To succeed and stay alive, a fighter pilot must be the first to observe an opponent, orient toward the opponent, decide what to do, and then act on the decision. Successful air to air combat requires operating your OODA cycle within (or faster) than your opponent.

It occurred to me how useful the OODA cycle would be in providing an easy to visualize and remember reason for testing early and often. It was also helpful that OODA rhymed with Yoda, the well-known character from Star Wars. Making such a connection aids the retention of new ideas. I prepared a presentation for an all EAGLE II project team meeting to build business and development support for making the investment needed to test early and often.

When testing or inspecting, one must observe the error, orient to the cause, decide on what to do, and then act. The effort and speed of the OODA cycle is dependent on the nature of the testing or

inspection. There is a large difference, easily illustrated between removing errors at the unit level and removing them during user acceptance testing.



Note the two OODA cycles in operation during user acceptance testing. The outer cycle represents the user acceptance test team finding, researching, and reporting possible errors. The inner OODA cycle shows the developers reacting with their own OODA cycle. The effort, time, and cost involved in resolving possible errors during a late testing cycle become readily apparent.

The presentation also included explaining that OODA was not Yoda's evil twin brother, the US Air Force connection to OODA, and the part OODA played in winning the cold war helped cement testing early and testing often in our corporate psyche. In fact, after the presentation, groups could be heard chanting OODA leaving the auditorium and for a year I was known as Dr. OODA. The foundation for testing early and often had been laid for the next steps.

The Gate

We've had a history of defects entering user acceptance testing that should have been found and corrected during unit test. In some cases the number had been large and thrown the effort into chaos. Performing quality unit testing was the obvious solution. Everyone reacts to measures. For our developers, historically the *only* measure had been progress toward the implementation date. EAGLE II was falling into the same trap. The speedy completion of work products was all

that counted. Unit testing is the last task in completing a program and was often sacrificed to move the product to user acceptance test. Change was needed, but how?

In hindsight, the answer was obvious. At the time it was not so clear. In a meeting to plan the user acceptance test, staff planning was being discussed. The team had just completed a user acceptance test cycle several months earlier. It became readily apparent during the discussion that most of our staff time was being used to research, log, and monitor test incident reports. A relatively small percentage of time had been used for actually executing the tests. I asked those in attendance for an estimate of the percentage of failed test cases compared to total test cases. The planning staff estimated that 25% of the test cases would fail. Some believed the number would be much higher. Our user acceptance test staff of 20 would need to triple in order to complete in the allotted time. Here was an attention grabbing number and a potential lever!

I met with the leader of the IT organization. We discussed the situation, our historical performance, and the need to radically expand the user acceptance test team. He asked what failure rate we could tolerate since hiring 40 people was not in the cards. I stated a 10% failure rate was tolerable and we had automated verification software in place for the first subsystem to quickly measure the percent of cases failed. I suggested that we refuse to accept the software for user acceptance test if the failure rate through automatic verification was greater than 10%. He agreed. I confirmed that we would likely miss our dates if we kicked the software back to the development teams. He stated that something had to change in light of the high cost involved in debugging during user acceptance test. (This presents a key lesson to remember. Exposing the cost of poor quality can be used to change the development process.) We had another measure besides *the date*, a single measure of quality that would dramatically change the course of the project.

We communicated the new criteria for entry into user acceptance testing. The effect was electric. A development team leader appeared in my office stating that the entry criteria for user acceptance test was much too severe and could not be met by his team. The rules had changed and some of the development staff were alarmed and upset. We substituted a people problem for a process problem. I asked the development team leader if it would help if we gave his team the test cases we were using. I asked if we worked together if it would help. He responded with a tentative yes. We succeeded in making quality the joint job of developers and testers! We now needed a process and people to improve the quality of the unit and integration testing.

The Bridge

The time arrived for genuine teamwork. The goal of improving the quality of the software delivered to user acceptance test was set. The development teams needed help. Unit testing skills varied greatly from developer to developer. The quality of unit test plans varied greatly, if they existed at all. No method of tracking unit test results existed. Thus was born the role of test coach. We associated a test coach with the major development teams. They were to be the bridge to take the development teams successfully into user acceptance test. The test coach embodied the commitment that the developers and testers were united in the effort to deliver a quality system. We would not use the new entry criteria negatively, but positively as a joint goal.

The test coach asked for test plans and assisted in their construction. Group reviews of the unit test plans were conducted with business, development, and test staff in attendance. In this way joint agreement was reached on the criteria for successful testing. It also proved useful in discovering hidden requirements. Each developer assumed responsibility for testing their code. The test coach received notification of unit test completion. In the case of on lines the test coach would re-execute the unit test plan. Batch programs unit test execution was documented. The test coach logged errors found on the development LAN. Individual developers were coached to improve their unit testing skills.

Very importantly, the test coaches provided a picture of unit test progress. We knew the status of each program. We knew which programs were untested and the quality of those tested. Program quality was now exposed. We added to the entry criteria for user acceptance test. We now stated that no unit with critical errors would be accepted. Since the status of each program tested existed on the LAN, the development team leaders began to use the database to monitor the quality of their units. It also gave the development team leaders the tool to improve program quality prior to the creation of the monthly reports. The test coaches kept the teams focused on quality while working hand in hand with them.

The Washing Machine

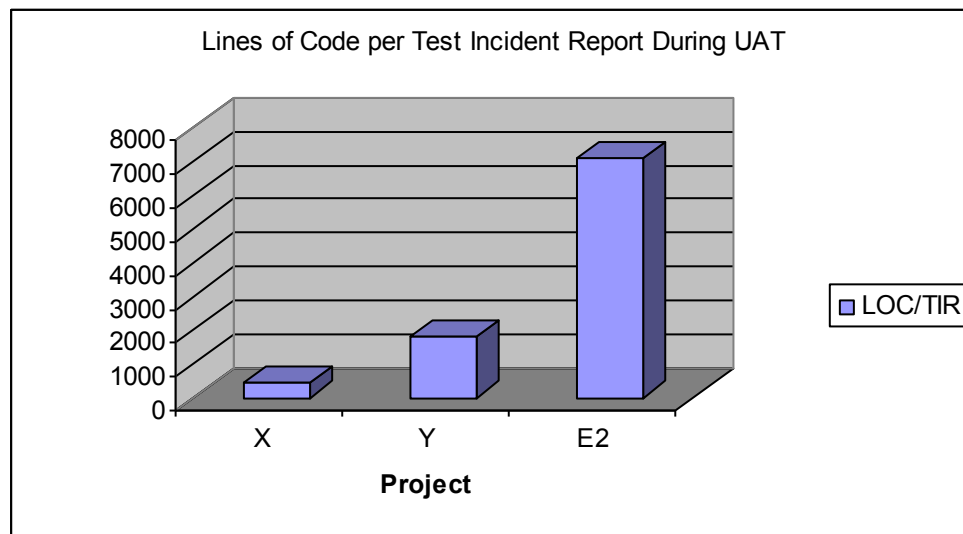
A brief history lesson may be of use. During WWII in the spring of 1943 a bulge in front lines of the eastern front centered on the Russian town of Kursk. The Soviets knew of the German army's penchant for attacking bulges at their base, pinching the bulge off, and thereby isolating troops and equipment for eventual destruction. This happened to be what the German army planned and the Soviets learned of German intentions. The Soviets prepared with a vengeance for the upcoming German attack. They built multiple lines of defense studded with men and equipment. Soviet planners knew no single line of defense would be sufficient to stop the concentrated German attack. However, they did know that penetrating a defensive line would cost the German attackers troops and equipment, so there would be less of each to attack the next line of defense. With many lines of defense, the attack would be greatly weakened. Another advantage would be the identification of where the weight of the German attack fell. The Soviets planned to then counter attack the weakened German forces. The Germans lost the battle of Kursk and were thereafter always on the defensive until the end of the war. Again, another lesson from military history with import for software quality.

Software engineering literature contains discussions of the Rayleigh curve. The idea behind the Rayleigh curve (and other reliability growth curves) is similar to the Soviet multiple lines of defense in the battle of Kursk. The software development life cycle is deliberately split into multiple phases. Each phase contains produces certain unique deliverables such as requirement documents, designs, and programs. Associated with each phase are quality activities such as reviews, inspections, and tests. These quality activities are designed to detect and correct defects created during the current phase and from previous phases. The goal is for no defect to cross a phase boundary. With good quality activities, the reliability of the software grows throughout the software development life cycle. This approach is also similar to washing machines with multiple rinse and spin dry cycles. Each rinse and dry cycles contributes to removing an increasingly greater percentage of the dirty water left behind by the wash cycle.

During EAGLE II we took advantage of conducting reviews, inspections, or tests on that phase's critical work products. We conducted extensive group reviews of early life cycle work products such as requirement documents and models. In similar fashion, the application teams during construction performed unit and subsystem integration testing. These tests were monitored for quality and as stated earlier, quality targets had to be met for entry into user acceptance test. Uniquely, we took the step of phasing user acceptance and system testing, building up to the full system, then to a pilot and then to final implementation. In this fashion, sub-systems and the entire system was tested multiple times in user acceptance and system testing. The testers and development staff worked very closely together treating these tests as near production. We found and eliminated defects and very importantly practiced implementation tasks. To get software really clean, take a tip from your washing machine and split your testing across multiple and phases and where practical, multiple times within a phase.

The Results

We experienced a dramatic increase in ratio of lines of code per test incident in software entering user acceptance test. In fact compared to earlier projects, EAGLE II shows a nearly 17 x improvement.



We also saw significant improvements in the quality of the software entering production. The results of the first 2 months of production follow:

- 165 defects were found
- 7,800,000 lines of code / 165 defects
- 47.2 KLOC / Defect
- Dramatic improvement evident the first day
- Smiles

The last bullet point deserves some more explanation. Most gratifying was our customers' reaction to the quality of the system. During training, the functionality and reliability of the system was on display. Those customers were so pleased they stood up and cheered! This is a true story. Teamwork does work!

Summary

Successfully completing large projects requires a high degree of teamwork. Project sponsors and project managers must structure the environment to encourage and promote the needed teamwork. Successful structuring means identifying your goals and objectives. We identified two objectives that we believed would lead to improved software quality. The first was to test early and test often. The second asked for quality to become the responsibility of everyone in the organization. We took steps to sell the organization on why this was a good idea – OODA. Development and testing management worked together to define and measure for entry into user acceptance testing – the gate. Test coaches were assigned to assist the development teams improve unit quality – the bridge. Lastly, we structured all quality activities to take advantage of the power of iteration – the washing machine. It worked for us, I hope it can work for you.