Abstract

This experience report describes the challenges and initial steps taken to move a large development team from traditional waterfall test phases to agile test methods. In late 2007, our development team started the transition to incorporate agile practices of user stories, daily scrums and time-boxed iterations. This paper focuses on the changes in test strategies necessary to support agile.

1. Introduction

An ideal scenario to introduce agile development would be to start with a small, co-located team of people to create a new product. This was not our situation. Our product is a graphical user interface client built on Java, the Eclipse framework and C/C++. Our product has millions of lines of code, some of which are 20+ years old, and it runs on six operating systems. The team is comprised of several hundred developers and testers on component development teams globally distributed. The test teams are integrated into the development teams.

In 2007, we decided to move to agile to get frequent customer feedback during our development cycle. Previous releases were based on a waterfall-based development model with long test cycles. A high percentage of tests were executed manually. Development and test phases were separated by detailed entry and exit criteria.

With the new agile project, our teams were formed based on product initiatives. Initiatives represent work for the release. Teams scoped initiative work through user stories and time-boxed iterations. Early product builds were provided to selected customers and business partners on a monthly basis to enable feedback on the new product capabilities.

Transitioning to agile development with such a large legacy product and globally distributed organization has its challenges.

2. Problems and Goals

Although there were many challenges for us transitioning to agile, the creation of new user stories was not among them. We were able to implement new code for user stories under stronger guidelines of unit testing and a fuller breadth of test coverage before delivering into the build. Automating new user stories was relatively easy because the work was well defined and the teams were focused and staffed.

The largest challenge was around regression testing. How could we handle regression testing of hundreds of functional areas across all six operating systems, maintain stable builds and maintain product quality while still delivering new functions and innovations? The only solution is to have a robust and reliable automated regression test.

It became immediately obvious from our initial data collection that our automated regression test inventory was inadequate. We needed continuous regression testing in the mainline build but also needed to test new user stories with the new innovations for the release. We needed to focus resources on automation development and move from a manual testing model to an automated model.

With that said, automation cannot replace the experience or knowledge of test engineers. Testers need to be focused on doing more complex testing, designing tests for new features, or improving tests for existing features. Manual testers perform interactive tests that can’t always be automated. They may focus on one area and then refocus on another area if they are suspicious of behavior changes. Or, if they get a new idea while they are testing, they change the test. This is when many of the bugs are found.

Our goal was to replace the repetitive manual regression test execution with automated tests and use the test engineers for exploratory testing, test design, automation development, and executing end-user scenarios.
The problems we needed to solve included:

- Lack of adequate automated regression tests
- Shortage of test resources for new user stories and regression testing simultaneously
- Previous test strategies were based on long regression test cycles
- Previous releases had testing commence after development had completed most of the design and coding
- Previous releases had automation development start after product development had completed making it virtually impossible to build testable hooks into the product
- Shortage of skilled automation developers
- Shortage of skilled testers to execute automation and analyze the results
- Testers multi-tasking on patches, main releases, maintenance releases at the same time

Goals were defined to address the problems:

- Define new agile test and automation strategies
- Decrease repetitive manual regression testing
- Increase manual testing for new functions and more complex testing
- Staff an automation development team to increase the functional automation regression coverage
- Increase automated regression tests to 80% on high priority feature areas
- Execute a weekly automated regression test
- Develop reliable automated tests that execute in a central automation framework
- Build testable methods into the product to enable automation development as new functions are added
- Enhance automation development environment to incorporate standard software engineering development methods
- Increase tester automation skills for automation development, execution and results analysis

3. Test Strategy

The basics of the test strategy are to align the test strategy with the business value proposition for the release. Testing is performed to identify business risks by exposing defects. The test strategy is designed to find the most important defects as early as possible with the lowest cost.

Function and system testing are a balance of scripted and unscripted prioritized tests. We determined the balance of test resources, coverage, and time based on risks. The test execution sequence was prioritized so that high-risk areas were tested first so rework or redesign could be identified as soon as possible.

3.1 Test Design Pairing Strategy

One of the basic agile principles is pairing people to work together. For test, it’s critical that test design is done with product developers and not in isolation. Developers who write the code and testers who test the results worked side-by-side to determine the right test coverage. Pairing the development architects and the test architects to determine the test requirements make for a more solid test coverage and execution plan.

Using a Test Design Pairing approach helped to build a solid working relationship and clarity on what tests are required and the test execution sequence. With this approach, there was a stronger correlation of the testing to the expected results because plans were formulated together rather than in isolation.

A pairing approach was a challenge for our globally distributed teams. There were planning meetings that were either late at night for one team or early morning for another. Once the initial test designs were discussed, we used database postings for test case reviews so a remote team could review at a more convenient time.

3.2 User Story Test Strategy

User stories were a new agile practice for the team to learn. User stories were written in relation to an end-user character or persona. They focus on the end-user capability and are intended to be small and simple so that estimations for the development and test work are easier and accurate. For example, Danielle can drag one or more e-mail messages or conversations from her Inbox to her sideshelf.

Transitioning the team to user stories was a fun part of the project. All of the team members participated in the iteration planning meeting, including members from user design, development, test, and documentation development. The product manager represented the customers and provided input to make sure we were planning the right work. Together we prioritized the user stories and in several instances broke them into smaller stories so they were more modular. While developing and estimating the user stories, the entire team seemed to gain insight into the aspects of designing tests to support the new functions.
The interactive discussions were effective in identifying the right coverage and scoping the tests which might otherwise been minimized or omitted from the estimates.

During the iteration planning meetings, the team members were given a Test Planning Worksheet to ask questions and provoke thought around the test coverage during the user story discussions. It helped the team think about the test configurations and various testing types such as function, error, load, stress, migration, performance, globalization testing to name a few. The worksheet helped to make the estimates more accurate by evaluating a broader test perspective rather than just a functional aspect.

For user stories, exploratory testing was encouraged and adopted. The testers worked with the developers on a daily basis to provide feedback on new functions, and retest bug fixes while building the test cases in parallel. As the test cases were developed, they were reviewed immediately and test cases that were out of scope were eliminated.

During the iteration, the team was required to test on all supported operating systems. As the project evolved, we developed checklists for iteration reviews. We continued to adjust the checklists as we gained more experience with iterations.

In addition to testing the new functions, the team was also responsible for regression tests of adjacent functional areas. Initially, the test estimates that were done in the iteration planning were not sufficient and in a few cases we ran out of time. The test work for the user stories were not completed by the end of the iteration so they were not delivered into the main build. During one iteration, the developers helped to execute the test cases on a few platforms in order to meet the date. The estimations steadily became more accurate after doing a few iterations.

### 3.3 Regression Test Strategy

The basic purpose of the regression test is to ensure that functions in previous deliveries are not adversely affected by new code. The system must continue to function correctly after the changes have been implemented.

The regression strategy for the previous release did not have specific goals called out for test results to the developers (feedback times). With a greater focus on continuous testing, regression definitions were defined as they relate to the execution frequency and feedback time to developers. Simply put, the more frequent the execution, the quicker to identify regressions in target areas. Since frequent execution is a necessity, critical functions were identified and automated.

Regression levels were defined as shown in Figure 1. The regression levels provide clarity on the frequency of test execution and coverage with expected feedback time to developers.

<table>
<thead>
<tr>
<th>Level 0: Unit Tests</th>
<th>Feedback = minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 1: Build Checkpoint</td>
<td>Feedback &lt; 30 minutes</td>
</tr>
<tr>
<td>Level 2: Daily Build Test</td>
<td>Feedback &lt; 1 hour</td>
</tr>
<tr>
<td>Level 3: Feature Smoke Test</td>
<td>Feedback &lt; 2 hours</td>
</tr>
<tr>
<td>Level 4: Weekly Regression Suite</td>
<td></td>
</tr>
<tr>
<td>Level 5: Full Regression Suite</td>
<td>Feedback = 4 weeks</td>
</tr>
</tbody>
</table>

![Figure 1. Regression Test Levels](image)

Each level incrementally adds functional test coverage and platform coverage. As the regression level number increases, the feedback time to developers increases due to the increased test execution time.

Regression levels 0 – 4 are automated tests. Level 5 is automated and is supplemented with manual testing. Manual testing remains critically important and should not be eliminated. Manual testing should still be planned for exploratory testing, more complex testing or areas that were not cost effective to automate. The test coverage was prioritized for each regression level with consideration to the feedback times.

- **Level 0 Unit Tests** are executed as code changes are made in the corresponding source files. Feedback to the developers is in seconds or minutes.
- **Level 1 Build Checkpoint** is an automated test that runs on a single operating system. It has limited test coverage that exercises the fundamental infrastructure areas. Feedback to the developers is within 30 minutes.
- **Level 2 Daily Build Test** is automatically executed for each build that is produced from the build system. Functional test coverage extends from infrastructure to the application layer and executes on each supported operating system. Feedback to the developers is within 1 hour of build availability.
• **Level 3 Feature Smoke Test** is an automated functional test that is executed on each supported operating system. It has more extended coverage for each application. Feedback to the developers is within 2-3 hours.

• **Level 4 Weekly Regression Test** is a series of automated test suites that are cycled through each supported operating system over the course of a week. Test coverage extends to 80% of the prioritized functional product areas. Feedback to the developers is in days and is dependent on the prioritization of the test suite execution.

• **Level 5 Full Regression Suite** is a combination of automated test and manual tests. The automated tests are more extensive and may require specific configurations for destructive tests. Feedback to developers is in days or weeks depending on the execution sequence.

4. **Weekly Automated Regression Test**

With the agile practice of continuous testing and a higher investment in automation, the selection of the tests that are included in the weekly regression test coverage is critical. The automated test coverage needed to have enough value to justify the costs of automation development, maintenance, and execution.

The goals of the Level 4 Weekly Automated Regression Test included:

• Provide feedback on results of the level 4 automated tests each week
• Identify product regressions more efficiently by eliminating expensive tedious, repetitive testing
• Test multiple operating systems in parallel
• Increase development capacity enabling testers to test a higher quantity of new user stories
• Enable testers to do more exploratory and complex testing

To identify the coverage in the automated regression suite, a risk assessment was performed to help determine potential quality issues and to prioritize the automation development work. Each component team was presented with the data for their product area to help determine the importance of coverage and priority for automation development.

The risk assessment criteria used to determine coverage in automated test suite included:

• **Business Risk.** Strategic and competitive capability and high-usage functional areas.

• **Development Risk.** Product areas with new functions, high complexity such as multiple integration points and external dependencies.

4. **Source File Risk.** Functional areas with high quantity of code changes or with multiple developers editing the same source files.

• **Defect Analysis Risk.** Analysis of defect volumes and regression volume from the previous and current release.

• **Manual Regression Test Volume.** Analysis of the quantity of manual tests in a functional area.

Source code file analysis was performed to identify files that changed frequently during the last 90 days of the previous product release as illustrated in Figure 2.

![Source File Analysis](Image 270x40 to 350x60)

Figure 2. Source File Analysis

The data indicated high-change rates on some source files, thus increasing risk of regressions in the corresponding functional areas. The source files were analyzed to identify the functional areas that were susceptible to regressions. Another dimension to the source files were the number of developers modifying the same file. These presented a risk for regressions so those areas were given a higher priority.

In the future, we will incorporate code coverage tools for C/C++ and Java. We intend to measure the coverage as the automated suite grows. Another future goal is to have the test suites automatically selected and executed based on the source files that were modified in the build.

Another parameter considered was the volume of manual testing. Functional areas that had a high amount of manual tests across multiple operating systems were prioritized for automation ahead of areas with a smaller volume. The automated tests provided almost instant relief to the manual test efforts upon completion of development.

A detailed defect analysis identified functional areas with high volume, high regressions, and customer-reported defects. Each component team received 4-6 charts of defects within the current release, the previous release and the last two releases. Automation for high-volume defect areas was prioritized ahead of
other functional areas. In Figure 3, the histogram represents the defect analysis for one component from the last two product releases.

![Component Defects by Functional Area](Image)

**Figure 3. Component Defect Histogram**

In summary, each component team used the data sets to help direct the priorities and gain agreement on the coverage of the weekly regression suite. This list became the input for the automation development priorities. The prioritization is crucial to agile and continuous testing since we don’t have time to test everything at the end of the project, like a waterfall method.

5. Automation Execution Architecture

The automated regression tests are executed in a Central Automation Framework (CAF) as shown in Figure 4.

![Central Automation Framework](Image)

**Figure 4. Central Automation Framework**

The Central Automation Framework is our end-to-end automation solution. It is comprised of a series of systems and services using virtual operating system images. New builds are automatically deployed to the CAF from our build systems through IBM’s open-sourced framework called STAF/STAX (Software Testing Automation Framework/Execution Engine).

Once the new build is deployed, automated builds tests are executed on a series of virtual operating systems. STAF services send email notification when there is a new build and when the results are available for evaluation. The STAF/STAX framework facilitates the distribution of the automated test runs and allows testers to monitor the progress and results.

The CAF makes it easier to cope with frequent builds and multiple operating systems. The framework reduces capital expenses because each tester does not need a dedicated system for each operating system. This represents a significant cost savings to the business. However, the test architecture comes with a significant investment in hardware, software, monitoring, maintenance, and administration staffing to support it.

The automated build tests executed within the CAF have reduced the manual testing costs by 67% in six months as illustrated in Figure 5.

![Monthly Cost of Manual Build Acceptance Testing](Image)

**Figure 5. Manual Build Test Cost Reduction**

The cost reduction is a way to show the return on automation investment. The resources that would otherwise have been performing the manual build tests were refocused on more advanced testing. The same is true for the automated regression suite. The savings in repetitive manual regression testing afforded us the ability to redirect the test resources to other important test work.

The reliability requirement for the CAF is high because if the system is not available for any reason, the cost is considerable. The fallback for the team is to execute automation on the tester’s local desktop or to perform manual testing. Therefore, the system management of the CAF requires that it be treated like a production system with formal procedures and guidelines for scheduled downtime.

Level 1 and Level 2 automated tests disqualify bad builds at a lower cost. The expenses continue to decrease as the reliability of the test architecture improves and additional automated tests are added.
The automated tests are continually strengthened to detect basic failures in the software quickly with lower maintenance.

6. Automation Development Strategy

In the past, our automation development was an afterthought. It was important, but automation development would proceed throughout a release cycle only to be applied at the end of a release. With agile, we needed continuous improvement and enhancement of the automation. To do so, we had to adopt agile methods to our automation development itself.

Our basic philosophy for automation development is to use standard software development practices to build change-tolerant automated test suites. Software development standards include technical design documents, source control, coding guidelines, best practices, checklists, and code reviews. Design reviews and peer code reviews have helped to make more effective automation designs and more robust automated test cases. Submission checklists have decreased the number of automation failures.

The senior management team recognized the urgency for our team to increase our automated regression suite and funded an automation initiative. The initiative was funded with resources to support the following goals:
- Build a reliable automated functional regression suite for Level 4 regression
- Use common tools and a common automation development infrastructure
- Simplify the automation development to increase test case development velocity
- Simplify the infrastructure by eliminating redundancies
- Standardize on design methods
- Simplify automation logging and results for easier debugging and result analysis
- Increase the automation development and execution skills across the team through education sessions

The automation initiative is managed through iterations and has stakeholder review meetings like all other initiatives. Each iteration has a planning session and delivers a list of user stories. The user stories are estimated and prioritized. The team is responsible for the deliverables by the end of the iteration. The automation initiative is managed through daily scrums, which have been quite effective in resolving technical problems. Evaluation results, retrospectives and stakeholder review meetings are held at the end of each iteration with a demonstration and summary of the results.

To communicate the status of the automation development to the stakeholders, a burn down chart was produced as illustrated in Figure 6. Team members estimated their automation velocity. Based on the staffing and experience, the burn down rate was projected.

The burn down rate is on track primarily due to the daily scrum meetings and iteration management method. This and the stakeholder review meetings helps to keep the team accountable for the deliverables.

![Figure 6. Automation Burn Down Chart](image_url)

7. Automation Design Strategy

Our product under development is a Graphical User Interface (GUI). GUI automation tends to be expensive to maintain and execute. As the product continues to undergo changes during development, the automation tends to be fragile and can “break” frequently. The automation has to be modified to match the new, intended product designs. The heavy cost is in the maintenance to keep the automation in sync with the product.

The automation design strategy takes into account that GUI automation is more susceptible to failures and has to be change-tolerant. Whenever possible, the team is implementing automation through Application Programming Interfaces (APIs) to get to specific test states. For example, if dialog boxes are used to get to a certain test state but are not the focus of the test; an
API can be used to bypass them. Since the dialog boxes are not the things being tested with this particular test, we do not need to go through the GUI when manipulating them. Instead, another automated test, one that focused on the dialogs themselves, would identify any errors in the GUI of the dialog boxes.

Using APIs in this way increases the reliability of the automated test. If for some reason the dialog boxes were redesigned or had a defect, other tests are not blocked from executing. Using APIs also increases the performance of the test execution substantially. For example, in one automated test, implementing the API approach reduced the execution time from 80 seconds to 2.5 seconds.

By no means do the use of APIs as part of the GUI automation solution reduce or eliminate GUI test coverage. Each and every GUI component is thoroughly tested and has correlating automated tests. APIs are used to improve reliability and performance and in some cases reduce redundancy in the automated scripts.

Another dimension to automation development is building testability into the product. One of the problems that we had using the waterfall development model was that it was virtually impossible to have methods added to the product to support automation because we were already at code freeze. And often, the development culture we operated in did not always see automation hooks as important as other development work.

For this agile release, we have made significant progress with numerous successes in building testability into the product. The best way to summarize our success with product testability is to include a quote from one of our automation developers:

“Here's the thing that is so great about agile. I'm writing automation for a new user story. There were some objects that I couldn't get at with the automation. So the developer made the methods I needed to get the object. Now I'm writing a wrapper for that object. If this had been done in a waterfall model, I would never have the chance to have the developer add that.”

There are still obviously some big challenges to deal with daily. How do we avoid automation rework? How can we plan for it more effectively? Rework is almost unavoidable but using a design strategy with APIs can make it more affordable. We are working to get automation for adjacent functions executed within the iteration and before it is delivered to the mainline build to avoid automation breakage.

8. Automation Skills

Another challenge we had concerned automation skills. We evaluated the team and the results clearly showed we needed to put a stronger program in place to enhance our team automation skills. Automation skill levels were identified and goals were set to produce training programs and year-end organizational goals.

Automation Skill Levels:

- Skill 1: Analyze Results from Automated Tests
- Skill 2: Automation Execution via Web UI
- Skill 3: Automation Execution via Desktop
- Skill 4: Test Case Automation Development
- Skill 5: Automation Framework Design
- Skill 6: Architecture and Infrastructure Design

The education program will continue for the rest of the year. Each session is being designed and delivered incrementally. We have a global team so we need to deliver the training multiple times to accommodate different time zones. For the end result, we anticipate increasing our overall development capacity because of the investment in the automation inventory and the team automation skills.

Beyond formal training programs, we instituted a program of mentorship, which has been quite successful. Each team member is assigned a mentor that is one skill level above. If the mentor is unable to resolve the problem or question, he or she goes to the next level above until there is resolution. This not only helps the mentee, but the mentor. Not only does it reinforce the mentor’s knowledge, but, if the mentor doesn’t know an answer, it forces the mentor to go to his or her mentor to resolve it. This lets knowledge trickle down from those with the highest skills to multiple people at once.

9. Conclusion

The key to testing in an agile model is to determine the right priorities. Test priorities must be agreed to by the whole team and not in isolation. They must be defined for regression tests, user stories, automation development, and test execution. Empirical data and risk analysis is crucial for defining the right priorities for the project.

We have definitely made progress moving to agile. We continue to learn and make improvements to determine what works best for our team. We continue to address problems. For example, some team members were frequently frustrated as they struggled with priorities. A person would get assigned to an agile project initiative with one set of priorities but would get conflicting direction from their line manager.
People also seem to still have their “day jobs” and their existing job responsibilities. Most people were not assigned 100% to an initiative because of his or her existing tasks. Typically people had 50-70% of their time for the initiative work.

We were successful in delivering monthly customer builds although it was challenging. We are still struggling with manual regression testing and will continue to increase our automation inventory to solve that problem.

Transitioning to agile gave us the urgency to move to a more automated test model. Without automation, continuous testing would not be possible with the size of the product and the number of platforms.

It became evident that test automation development requires the same vigor as product development. Automation development requires standard software engineering practices. Automation must be designed for reliability and performance.

The automation skills and mentoring programs have been effective. Overall, the team skills are improving significantly. We are increasing the automation developers and automation execution staff across the team. By year-end, the goals are to have 60% of the team doing test case development and 95% of the team executing and debugging automation.

User stories were very effective in terms of making the whole team understand what we were delivering in terms of the customer value and what the corresponding test design and effort involved.

The daily scrum has been effective for resolving problems for the automation initiative. We feel that it’s not “just another meeting”. It’s a very valuable part of the agile model and has been extremely beneficial in team collaboration, keeping everyone focused, solving problems, and delivering results.

It will take us a few more releases to improve and optimize on our agile implementation. We may add additional agile practices like Test Driven Development (TDD) and more pair programming. For future releases we will be focusing on optimizing the test execution model and continuing to find the right balance of test coverage and time, to deliver a quality product.

10. References


