Forming to Performing: The Evolution of an Agile Team

1) Darren Rowley, 2) Manfred Lange
First Data Utilities
1)rowleydm@gmail.com, 2) ml@agileutilities.com

Abstract

Using Bruce Tuckman’s [1] “Forming – Storming – Norming – Performing” model of team development we review the effects of change on our team through each stage. We go on to show how in certain cases this change is cyclic and techniques abandoned in one phase may well be re-introduced in another. While tools, techniques and processes may change we observed that despite the cultural change there is one thing that remains constant: The agile principles and values.

1. Project Background

First Data Utilities (FDU) provides customer information and billing systems for the energy and water industry. At the core is a Customer Information System (CIS), which Peace Software marketed as a commercial product for over a decade. As the software has aged over time, the company decided that the system needed to be moved to a new generation of technology. This report refers to a project that evolved the architecture and design of the CIS towards a modern, service-oriented architecture based on mainstream technologies, tools and processes. The business case for the project rested on the existing shortcomings of the product, and a substantial budget was approved for the project.

Starting a new project is always a good point in time to also think about processes, technology, and tools. When this project started the company was using a pure waterfall approach. Processes were very well documented, and there was even a process for changing processes. A lot of things were formalized; there was even a time-code for filling out the time sheet. The majority of the organization was in favor of a significant change. Fortunately some smaller projects with up to 6 engineers and up to 6 months time frame had already served as pilots for the use of agile techniques. This project, called TAM, was the biggest internal project that the company had ever undertaken.

The entrepreneurial spirit in the team made it possible to start the project with a large number of additional risks, including the large-scale roll-out of agile methodologies.

When the project started, the organization – Peace Software – was a small independent company selling one product internationally. Entrepreneurial thinking was typical in the company, and often led to changes in direction for the project. An example included the early decision to no longer pursue a big-bang approach but to instead use a module-by-module process towards building the envisioned target system. The acquisition of Peace Software in August 2006 by First Data will certainly have an impact on the environment as well.

2. The Story

We believe that Bruce Tuckman provides a good model for the evolution of group behavior, which he split into four stages. Our story is structured along Tuckman’s stages (Figure 1), each of which we will briefly introduce.

![Figure 1. Tuckman’s Stages](image_url)

2.1 Forming

The forming stage is characterized as a period of orientation. As the team forms, boundaries surrounding scope, processes and roles will be unclear, team members will be cautious although excited and optimistic about the new opportunity ahead. The new team will be highly dependent on the team leader or coach to provide direction. Members will test the tolerance of leaders and processes, new practices may often be ignored and there will be a hankering for pre-existing standards.

When the project started we faced a number of challenges some of them caused by a major and
traumatic reorganization in March 2005. The agile approach would have been to start small, and increase the team size slowly over time. Instead we had 24 people from day one. The team had never worked together and the combination of people was unusual with half of them being architects with strong personalities. This played a key part in the storming phase of the project.

2.1.1 Testing Tolerance. Testing of the leader’s tolerance was amplified by the direct manager having his desk on a different continent several time zones away rather than being co-located. There wasn’t open resistance but team building was an issue for several months and a particular challenge initially. Leading such a team without direct authority, using influential leadership, was a huge challenge at the beginning.

2.1.2 Working Style. The preferred working styles ranged from academic to pragmatic, from working solo to pair programming, and more. For example, an agile approach towards solving problems was to use time boxed spikes to find a good solution. Some team members preferred a more academic approach, analyzing a problem from many different directions before moving towards a solution.

2.1.3 Providing Direction. The team faced many other challenges including the lack of furniture, equipment, process, and space. Manfred decided that issuing orders wouldn’t provide a fast way to get the team up and running. A scalable approach was required.

The team met in a large meeting room, and Manfred provided A6 index cards to capture tasks. We briefly discussed which large areas needed to be addressed, and then we split up into smaller groups, each responsible for one area. For example one group took care of setting up a team server and putting software in place such as CruiseControl and CVS. A different group took care of organizing office space and furniture for the group. Although the roles were not clearly defined the team members knew within a short time what to work on.

Each group organized themselves. This turned out to be very scalable, and despite the challenges we managed to get the major items sorted within a couple of weeks. Again, directions from the coach helped the team to get organized.

2.1.4 Unclear Processes. At this stage of the project we had little formal processes – another match with Tuckman’s definition of the forming phase. We started to use some of the practices from extreme programming, e.g. stories, small iterations, and review meetings.

The team was still very heterogeneous. Some team members started to use pair programming but it was a challenge to change the culture to one where pair programming would become the norm. The coach supported the team by pairing himself, but provided guidance also with other means such as formal training sessions.

We decided to start with one week iterations which were particularly useful for selecting tools and technologies. At times we tried as many as three similar technologies simultaneously. The first few attempts were often devastating. It took several days before we had the first successful build, and after that it broke very often, too often. Sometimes the success rate of the cruise loop was as low as 30% compared to our target 90% success rate.

2.2 Storming

Tuckman’s storming stage is a time of conflict and polarization amongst the team. Members fail to work as a cohesive unit and the team can become factional and power struggles may emerge. Individuals may resist collaborating with other team members and may focus on personal agendas rather than real issues. The team needs to be focused on its goals and team leaders need to be direct in their guidance. This can be the most difficult time and a team may never leave this stage or suffer significant attrition.

2.2.1. Polarization. After the dust had settled, the focus moved from the immediate needs such as hardware and tools, towards architecture, design, and technology. A hefty discussion, which would turn into a fight at times, developed over the question of whether or not to use transfer objects in the communication between the presentation layer and the business logic layer.

This discussion was further hampered by the fact that half of the newly founded team formerly belonged to the architecture group. Eight months before this project started they created the architecture that would carry us through the next ten years. One faction argued that we should follow the new architectural blueprint then implement our system in line with that five layer architecture. A different faction in the team promoted the idea of starting with a very simple and light-weight architecture and then letting it evolve along the way.

2.2.2. Personal Agendas. In addition to the sometimes heated architectural discussions a few “prima donnas” spiraled off to try technologies that other team members considered “exotic” in the given context. One engineer suggested using Jini, a technology typically associated with embedded systems and we spent many hours discussing the topic. Manfred settled this particular item by putting it on the backlog and giving it a very, very low priority, and a very, very high estimate. Another engineer attempted to use new Java 5 language features in inappropriate
places. Several months later the team had to heavily refactor this part of the code because nobody understood it.

The hottest discussions took place around the user interface technology. Until then the company had deployed no less than 5 different UI architectures. This time the team wanted to use a consistent set of technologies for the entire system. Some people promoted Struts, other were in favor of JSF. The debate was extremely emotional, and Manfred can remember at least two occasions when people angrily left the meeting room, feeling the discussion had taken the “wrong” direction.

2.2.3. Direct Guidance. Somewhere in the middle of the storming phase – about 3 months into the project – and with the direct manager not being co-located, Manfred simply assumed accountability for some decisions. The discussions reached a point where Manfred decided that progress had almost come to a halt. In order to get it moving again he pushed through the three major technologies to be used: Hibernate for the data access layer, Spring for the business logic layer, and JavaServer Faces (JSF) for the presentation layer.

Technology selection by the coach is not an ideal approach. Given the coach’s past experiences, the maturity of the team at the time, and the need to make progress, this was a trade-off against a self-organization. Some people took “self-organizing” as a justification to refuse any kind of leadership or managerial control. This was when Manfred learned that “self-organizing” and “self-discipline” are two sides of the same coin.

2.2.4. Attrition. The storming stage was also characterized by the highest attrition over the two years of the project. One reason was the significant cultural change that was caused by moving from the old waterfall-like world with strict roles to a more merit based approach. Unsurprisingly this was perceived as a threat by some team members. Another reason for the high attrition was that some well-respected executives left the company and some people interpreted that as a sign that the software engineering organization would revert back to the “old” habits. This, fortunately, didn’t happen as it turned over later.

During this stage we also introduced a new hiring process. We basically handed it over to the team. Three senior engineers formed a “recruiting team” and assessed all candidates. We added a programming task, peer interview, pair programming, and a design session to assess applicants. The new approach has a spotless track record as at the time of writing this experience report. Having the team selecting the successful candidate resulted in a high quality and guaranteed the team buy-in.

We also changed our planning and tracking process. Some team members – coming from a waterfall-like environment with tools for every aspect of software engineering – viewed index cards as unprofessional and wanted to use an application. We settled for XPlanner. Once this was set up, the focus of discussions moved away from the tool onto the actual planning and tracking. Story writing became an important topic, and as it was easy to add hyperlinks to stories in XPlanner, the onsite customer started to make use of that feature increasingly to refer to large documents, or screen shots of hand-crafted user interface prototypes. Most stories had a lot of details in them, and they tended to be hard to handle, e.g. to be removed or modified. Manfred wasn’t very happy about this decision, but left it to the team. He tried to give the team as much authority as possible so they would become more and more self-organizing.

2.2.5. Factions. Team interaction was still factional with more people trying pair programming but still some people refusing this practice. Some individuals felt that they were more productive if they worked alone. In some cases this might be true but it would emphasize only an individual’s agenda rather than the whole team’s progress. It was more about local optimization (the member’s productivity) while sacrificing global optimization (the team’s performance).

Likewise interaction between onsite customers and engineers was often characterized by power struggles and conflict instead of collaborating to solve concrete issues.

At the beginning of the storming stage, about six weeks into the project, the team trialed different sprint lengths. We moved to two weeks iterations and also tried a few with three weeks. We had disagreements over the “right” length of a sprint, but settled on two weeks at that point.

The team organized itself along functional areas of the system we were building. We built four groups of similar size, each with a balance set of skills. One group for instance would take care of the billing engine. Another area was collections. Yet another group was responsible for the product infrastructure.

We also had positive experiences in the storming stage. On the testing side we started with HttpUnit but very quickly switched to Selenium. Things further improved when we started working with Rick Mugridge [3] who is the author of FitLibrary, a tool for creating related Fit tables. Rick introduced the idea of executable specifications or story tests. Fitnesse and FitLibrary were added to our tool box. The product managers on our team worked very closely with Rick to develop the application’s domain model.
Pair programming was increasingly adopted. At this stage people sometimes stayed with the same programming partner for several days, which wasn’t ideal but definitely an improvement over no pairing at all.

Test-driven development (TDD) was not yet generally used. There were many reasons given for not fully using TDD at this stage. Old habits were to write code first and then test it – only if time permitted. Another reason was a lack of discipline. This was not intentional but people simply “forgot” to consistently write their tests first. It took the team several months to fully understand the benefits of TDD. With experience TDD helped the team drive the design of the application as well as providing a safety net for refactoring. This practice also helped the developers discover all test cases required for a story.

### 2.3 Norming

During Tuckman’s third stage, the norming stage a sense of trust and cohesion returns to the team. Members agree on rules, processes and values, and teams begin to self organize around decision making. Team leaders need to foster collaboration during this stage usually via hands-on participation.

#### 2.3.1 Cohesion Returns

The start of the norming stage was marked by the team introducing brief decision making meetings, which were usually held immediately after the daily scrum. Topics ranged from items as trivial as where to put curly braces source code through to trying to organize rotation of developers between teams. Developers often used short “show and tell” lessons about new technologies or practices for the code base. People were starting to think about the good of the team over their own goals.

Lead by Darren a small group reverted to some of the lighter processes of the forming and storming stages when they were asked to begin work on a new tactical application, “Revenue Assurance”. The smaller team size and emerging requirements allowed for one week iterations, the use of story cards and a dedicated build for this team only. With just a handful of team members and story narratives consisting of just headlines the team felt XPlanner was too heavy weight. The index cards and a simple tracking spreadsheet complemented each other nicely and the spreadsheet was used to provide a weekly status report email to stakeholders.

The remainder of the group (around 12 developers) converged onto the “collections” module and heavy development in this area ensued. The team had decided to increase the sprint length to three weeks with the view that the longer sprint would allow more room for maneuvering within the sprint.

#### 2.3.2 Fostering Collaboration

Another example of norming was a “fostering of collaboration” when Michael, one of the development leads introduced the planning poker game [2]. With the larger team and longer sprint length, Michael was concerned that the planning meetings would become unbearable, possibly even worse than some of the painful planning sessions experienced during the storming stage. The team readily accepted the experiment; the game became a success and is still in use.

The cohesive team now trusted each other enough that the role of tracker was able to be rotated every sprint. The sprint tracker was responsible for tracking the progress of each story, running the daily scrum and leading the planning meeting, a task made easier by the compulsory requirement to bring donuts.

The team’s stories also started to become smaller and more manageable. The reduction in story narrative was helped by the previous introduction of Fitnesse providing executable specifications. The onsite customers, usability experts, and the engineers increasingly used a jointly developed domain language. Design and implementation of the system shifted more and more towards the domain. Until now, much of testing was done through the user interface using Selenium. We started to move away from that approach by running the story tests directly against the business domain layer thus bypassing the user interface. This also had the benefit of saving precious processing time during the run of the automated test suite.

A lightweight framework was written so that Selenium tests could be generated from java code, this brought about the benefit that both unit tests and functional tests, could share some of the same fixture code and refactoring tests was now easier.

With the increasing size of the test suite and of the ever growing functional code base the build time started to create serious concerns. Our aim was to keep the build time below 10 minutes to enable timely feedback by the automated build system. Multiple selenium streams were run by triggering up to seven VNC browsers at a time and although this definitely had a positive effect on the build time it was still way too slow, finally exceeding 40 minutes. Even worse because of the parallelism the build became quite brittle and too much time was spent either refactoring build scripts or chasing down non-parallel-safe tests.

### 2.4 Performing

During this final stage the team becomes totally focused on the tasks at hand, the team will now be ‘unconsciously competent’ with tools and processes. The team can collectively make decisions without
supervision and leaders can focus on personal and interpersonal development. The team will maintain a high level of collaboration and productivity and does not need to be instructed or assisted.

2.4.1. Totally Focused. The team became totally focused on the task at hand when attention was turned back to the primary goal “build the next generation of Utility CIS products”. In contrast during the norming stage the product had taken a back seat whilst we were striving for perfection with our tools, processes and architecture. In daily scrums during the norming stage, most of the discussion revolved around our target architecture, the testing frameworks and build scripts - we had lost focus on delivery. In the performing stage efforts were made across the board to become more delivery focused. Customers split and trimmed the backlogs of any superfluous functional stories and what remained became the team’s top priority. Senior team members questioned every piece of infrastructure or technology change and asked “how much will it increase velocity”. Infrastructure and architecture changes were blended into required functional stories instead of having non-functional stories. As the team turned its attention to the true job at hand we progressed into a truly performing team.

A mindset change however was not enough, as we still needed to look at our tool and process. The teams reverted to smaller sizes, similar to those in the early stages, and each team concentrated on a separate functional area, as we had already done during the storming stage. The larger collections team split off into two teams of four developers. A lead engineer was appointed for each team and a ‘scrum of scrums’ was held daily to discuss project wide impediments.

Following in the footsteps of the Revenue Assurance team these smaller teams also reverted to lighter tools and processes of earlier stages, the tracking spreadsheet prevailed across the entire program and XPlanner was now only used for its build queue feature. Index cards were now used everywhere and even the purchasing department acknowledged this by making sure that there was always a sufficient supply of index cards on stock. Story sizes were consistently measured in terms of ‘points’. As stories became small enough it was possible to switch to one week iterations. The teams were now experienced with both the domain and the new architecture and became very proficient in terms of estimating and committing to an entire release cycle, and actually delivering on the committed work. The planning meetings became extremely efficient with usually 30 minutes of planning time per week for each team. Sizing and prioritization worked without controversy.

2.4.2. Collective Decision Making. An example of collective decision making involved very old legacy code that nobody enjoyed working with. The team decided to create a roster and rotate people in and out of that area each day, work become bearable and although people still didn’t like it, these tasks were completed.

A high level of collaboration now existed between onsite customers and engineers, with frequent interactions every day. A rule was introduced that when a pair started to work on a story they would talk briefly to the onsite customer as a sanity check that they had a common understanding of the story. Likewise customers were encouraged to test a completed story card immediately, often at the developers’ desks. No story was considered complete unless it was signed off by an onsite customer. There was always an expert in ear shot [4] as customers were co-located to within a few desks of their new teams and engineers were also encouraged to interact with the onsite customers frequently.

2.4.3. Cyclic Change. Many of the practices went through cyclic change, one example were the executable specifications. Story Tests had worked well for the heavily analytical Revenue Assurance product but not so well for some of the simpler business logic and validation scenarios of the collections product. In the storming phase Story Tests were used to great affect with the complex collections credit cycle prediction logic. We reverted to this kind of tactical use only.

Stories were now smaller and easier to understand therefore decreasing the need for clarification via Story Tests. Also, developers found it difficult to swap between TDD using JUnit and an approach of driving the domain through story tests. It is especially difficult for less experienced developers to know when to continue on with a JUnit test. The code base was now very mature and Fitnesse only had limited support for ongoing re-factoring. Our fixture library became very large and we often spent more time maintaining these than writing production code. Finally although there was huge benefit to the customers writing FIT tests the tabular format did not come naturally to them, this resulted in duplication across Story Tests.

2.4.4. Collaboration and Productivity. Team collaboration was also used to bring the build time under control. A separate Cruise Control project for each module was created. Not only did this significantly reduce build times (below 10 minutes) each team could continue to commit if another teams build was broken. Common functionality was put into a “core” project. Any change in the core project triggered building every dependent project, too. We also revisited Selenium tests and refactored them to test UI logic only. Testing business logic and controller tier code (i.e. JSF backing beans) using functional tests
was discouraged, and Story Tests or JUnit tests were used instead. These build and test infrastructure stories were played only because they resulted in a marked improvement in velocity, more exotic measures were discounted in favor of functional delivery.

3. Conclusions

After analyzing the evolution of our project through Tuckman’s four models it became obvious, as show in Table 1, that some of the change was cyclic. Tools and Practices discarded during the forming and storming stages were reintroduced later. Although our team and body of work had grown our processes were once again becoming lightweight.

During the development phase of Team New Zealand's (successful) challenge for the 1995 Americas Cup, management was open to all suggestions from the team. Their only consideration was: "Does it make the boat go faster?” Likewise it was beneficial for us to ask “will this change improve our velocity”.

That said an evolving team - like a child playing with a new toy - may need various rounds of experimentation to advance its learning before we it settle on the best course of action. This experimentation should be fostered through the forming and storming stages as it gives the teams confidence and experiences they may again want to revert to as they move into the performing stage.

We started to realize that there is one common thread that permeates the entire story of the team. We realized that this story is really about cultural change. All of the things listed above (and also those many items we didn't list) are external expressions of a change that happened to the team, to the people, to the human individual. We actually changed our perception of the things we keep constant verses the things we experiment with.

Things we keep constant are essentially intangible. Values and principles are important. It is the tangible things that change such as the office layout, the tools we use for development, the way we use them for development and so forth. While this looks like a simple achievement, in reality it is not. Cultural change is one of the hardest things to achieve.

4. Acknowledgements

We would like to thank our shepherd Ben Hogan for his invaluable and timely feedback and editing advice.

Thank you also to Angela Martin for providing direction.

5. References


<table>
<thead>
<tr>
<th>Timeline</th>
<th>Forming</th>
<th>Storming</th>
<th>Norming</th>
<th>Performing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration Length</td>
<td>1 month</td>
<td>5 months</td>
<td>6 months</td>
<td>12 months+</td>
</tr>
<tr>
<td>Build Time</td>
<td>2 mins</td>
<td>10 mins</td>
<td>40 mins</td>
<td>&lt; 10 mins</td>
</tr>
<tr>
<td># of Teams</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>Planning Duration</td>
<td>30 mins</td>
<td>120 mins</td>
<td>60 mins</td>
<td>30 mins</td>
</tr>
<tr>
<td>Average Story Size</td>
<td>1.5</td>
<td>4</td>
<td>3</td>
<td>1.5</td>
</tr>
<tr>
<td>Tracking</td>
<td>Index Cards</td>
<td>XPanner</td>
<td>XPanner</td>
<td>Index Cards</td>
</tr>
</tbody>
</table>

Table 1: Cyclic Change