Software Development Process Changes in the Telecommunications Industry

John Kevin Doyle
Benedictine University
Lisle, IL 60532 USA

Robert S. Janek
Lucent Technologies, Inc.
Naperville, IL 60540 USA

M. David Long
Tekelec, Inc.
Richardson, TX 75081 USA

INTRODUCTION:

The tremendous changes in the telecommunications business in the last several years drove changes in the software development processes of telecommunications equipment providers. We compare changes in these very large projects, in two companies, with those proposed in the Theory of Constraints / Critical Chains [1], [2], Extreme Programming [3], [4], and Agile [5] development models.

The 2000s have been a time of significant challenge in the telecommunications equipment business. Telecommunications service providers have excess equipment capacity. Many are waiting for next generation telephone switches that will simultaneously lower operating costs and enable additional revenue generation. The large service providers have drastically reduced their capital and expense purchases. Many small service providers, particularly the dot-coms, went bankrupt; much of their equipment is on the secondary market, at a fraction of the original cost. Thus the equipment market has significantly shrunk, and the equipment providers have been reducing expenses, while continuing to deliver software and hardware equipment at the high quality level required by the service providers. This drove many changes in the software development process. While the process changes are reported in two telecommunication equipment development organizations, the changes are applicable in any product development organization.

THE BASELINE:

We describe the organizational structure and software development processes used at Lucent Technologies in mid-2001. This is then used as a baseline with which we compare the current processes, at Lucent and at Tekelec.

Baseline organization:
Lucent Technologies was spun off from AT&T in 1996, as a telecommunications equipment design and manufacturing firm. The baseline organization in this paper is Lucent Technologies' telephony switching division, whose primary product at the time was the Lucent Technologies 5ESS®. By mid-2001, the 5ESS organization included thousands of developers, testers, project managers, and others.

The primary organizations involved in the software development process were development and business. Development was organized by function, with multiple departments focused on requirements generation and maintenance (called systems engineering), on project management, on software development (design, development, and unit test), on software construction (load building), on test environment (planning, delivery, and maintenance), on integration and system test, and on process management and other quality-related activities. Development's primary external interfaces were to business and directly to significant customers.

Business was primarily organized into departments focused on customer regions (e.g., North American customers) with a small coordination department. These departments performed what is traditionally viewed as product management. Their primary external interfaces were to development and to sales.

Baseline process:
In mid-2001, the software development processes consisted of Front End, Software Design, Software Development, Software Construction, Project Management, and Test.

In the *Front End* component, the tasks were Release Planning, Requirements Generation, Project Commitment, and Project Planning. *Release Planning* determined the baseline feature contents of major releases (an annual release was approximately the maximum that customers could afford/apply). The release content was primarily a business decision, informed by initial feature cost estimates from development. Changes in estimates, or differences between estimated and actual costs, and changes in the business priority of various features would change the targeted feature contents over time. Release planning also determined the release schedule, staffing required, a business case, and (sometimes) a customer offer or response to a customer RFP, RFI or RFQ.

The *Requirements Generation* activity took the high-level feature definition, the existing documentation for the overall system, industry standards, contractual obligations, regulatory requirements, modification requests

targeted for this release in the area of this feature, and generated and formally reviewed a requirements document. Requirements generation was principally performed by members of system engineering departments, with review as appropriate from others.

The project management departments performed Project Commitment and Project Planning. The information generated by the release planning step was somewhat abstract – "it should take three people three months to develop and test this feature". The *Project Commitment* step was to take the abstract planning information for all features in a project, compare the staffing needs with staff availability, and build a project network [7] which developed all the features. In the late 1990s, Critical Chains planning [2] was introduced, and the committed delivery date was the end of the project buffer. The *Project Planning* activity was to flesh out the project network to a complete project plan, adding risk management, process deviations, delivery mechanism planning, and test environment scheduling. This plan was formally reviewed, and this formal review often served as a project kickoff.

In the *Software Design* component, the tasks were to develop a High-Level Design Outline and High-Level Design document. The former was a viewgraph-level outline of the software design, covering decomposition into modules, functionality of each module, inter-module interactions and interactions between the modules and the system of which the feature was a part. The latter detailed the information outlined in the former, and was formally reviewed.

In the *Software Development* component, the activities were to *Develop*, *Code Inspect*, and *Unit Test* the modules. The detailed processes used were specific to the particular code areas. Formal code inspection was required. Unit tests were expected to cover every leg of code. The software design and development was performed by members of the software development departments. In telecommunications applications, a significant proportion of the software is "data" – customer configurable, recent change, office dependent, equipment configuration, system generation, etc.

In the *Software Construction* component, the tasks were Load-Line Planning, Load Building, Simulator and Laboratory Bring-Up, and Delivery Information Generation. These activities were performed by members of the Software Construction department. In *Load-Line Planning*, software construction and "bring-up" resources were identified, and load building schedules determined, including code submission and load availability dates. In *Load Building*, software submissions from all developers to a particular load were combined and built, and the source code control databases updated and maintained. In *Simulator and Laboratory Bring-Up*, the executables were installed on the simulators and laboratories, problems identified, fixes generated (typically with help from software developers), fixes installed, "bring-up" tests run and passed, and the environments made available for use. Finally, in *Delivery Information Generation*, delivery documents and media were generated to allow transmission of the constructed software to manufacturing centers, for physical product production.

In the *Project Management* component, the activities were Execution, Change Control, and Project Completion, and all were performed by members of the Project Management departments. *Project Execution* involved monitoring the project plan, including the network diagram, staffing plans and reality, and identified and emergent risks, and taking corrective action as required. The project management model used was Critical Chains, so buffer recovery was a principal focus. *Change Control* was typically performed by a committee, chaired by the project manager, which assessed every proposed change to the project and decided to accept a change proposal with no changes to the project plan, accept subject to changes to the project plan, defer, or reject the change. This always involved obtaining business impact, resource availability, cost and schedule estimates from others. Finally, in *Project Completion*, the project manager archived the project network and plan (for comparison in estimation and planning of future projects) and developed a project retrospective.

The two major groups of activities in the *Test* component were Test Environment and Testing. The Test Environment tasks were Laboratory Planning, Laboratory Engineering and Operations, and Simulator Planning and Development. The (direct) Testing tasks were Requirements Test Planning, Requirement Tests Execution, Regression Test Planning, Regression Test Execution, and Problem Report Fix Verification. All of these activities included hardware, software and data aspects of the laboratory and simulator environments.

Laboratory Planning involved assessing project needs, and forecasting and acquiring laboratory equipment. The *Laboratory Engineering and Operations* activity configured specific laboratories for use by specific projects during specific periods, and provided first-level laboratory problem resolution. The *Simulator Planning and Development* task determined what changes and enhancements were needed in the simulator environments, and engineered and maintained specific instances of the simulators to model specific hardware / software / data configurations.

Of the direct testing tasks, the *Requirements Test Planning* activity designed and developed tests which would verify all requirements, and planned the detailed testing schedule, including staff and laboratory/simulator resources. *Regression Test Planning* designed and developed tests which would verify existing feature functionality, and planned regression testing schedule, including staff and laboratory / simulator resources. The *Requirements Test Execution* and *Regression Test Execution* activities executed these tests, identified problems (in the software under test, the laboratory/simulator configuration or implementation, or the tests), resolved the problems (with assistance from others as appropriate), and re-executed the tests, repeating this as necessary. The *Problem Report and Fix Verification* activity verified

fixes delivered in the software under test, to problems which had been identified by development, by customer service, and by customers.

PROCESS CHANGES IN LUCENT TECHNOLOGIES:

In addition to the process changes driven by the usual pursuit of increased efficiency and speed to market, the significant downturn in the telecommunications equipment market forced further changes. A project with thousands of developers was downsized by an order of magnitude. The level of process control should be proportional to the number of people in an organization. As the number of people decreased, we needed to reengineer these controls and procedures to be appropriate for the new organization.

Front End/Project Management:
The front-end organization was merged into project management. All of the activities were project management activities, but the sheer size of the team in the 1990s necessitated two departments. The resultant benefits, in reduced handoffs and simpler communication, are apparent.

It's interesting to note the momentum in behavior. After years of automating to reduce costs, there was still a tendency to identify significant tool development work to further automate process checks. This is an effective way to reduce costs in a growing or stable company. We contend, however, that when shrinking, an organization generally needs less controls and more informality. Our challenge to right-size processes becomes not just to attack the technical problem, but also to overcome the mind set momentum that had been pushing in the other direction for a decade.

For example, databases and tools that produced automated reports were excellent in supporting an organization of hundreds of product managers and thousands of developers. Filling out forms and populating databases is a cumbersome way to communicate to a few key people. In a small organization, a telephone call handoff is more efficient.

Removing some of these tool steps, for example during project estimation, has become the new target. Roles were eliminated so that the product manager hands off directly to the project manager, who estimates the project. Similarly, driving action via a myriad of reports has been discarded and email or phone is used for the handoff. The tool that sent email automatically to alert dozens of people of a new feature estimation is no longer helpful.

Estimation of new product features was perhaps the biggest surprise. With the reduced overall organization size, and fewer handoffs, one naturally expects faster execution in development. Estimation was to a large extent based on 'Yesterday's Weather' [3]. With the rapid change in the organization size, history was predicting what turned out to be overestimates of project duration and cost. Over twenty years, the organization had built an ability to estimate projects very accurately. This estimation process needed to be recalibrated to factor in the significant increase in speed of working with a smaller team. Subsystems that formerly spanned several groups or departments were now supported within a single work team. The end result was that feature development velocity increased significantly.

In a small organization, estimation and commitment databases can be quite simple. There is no need for tools connected to these databases, to produce reports or to help to assure process integrity. However, there are limits to how much can be shed even if it is no longer ideal. The commitment process was perhaps the biggest demonstration of this. A database had been developed to keep track the dozens of load lines and thousands of options for customers. With features in progress at all times, the only way to assure compatibility was to keep track of these options as soon as they came into existence, during the commitment process. So if a customer in Europe was getting a feature that treated Caller Id one way, and an Asian customer was getting a different version, the database kept track of the customers and their options. This way, when the European customer upgraded from one release to another, we could assure that the new release wouldn't deactivate features the customer had purchased over the years. Also, as a customer received new features, appropriate regression testing would be planned. So an options database, beginning at commitment time, was beneficial. With less code being produced and fewer streams, a simpler database would have been workable. However, re-architecting this complex tool for a mature product was infeasible.

A significant process change was the introduction of Theory of Constraints for project management of development [2]. Although underway at the baseline point, continuous refinement of the implementation has yielded large gains. Whether applied to a small or large organization, the benefit in interval, throughput, and control of a project is evident. Managing buffers, taking advantage of early finishes from the previous tasks, minimizing multitasking, and having a critical chain with a constraint that doesn't bounce from area to area with each new feature, were large benefits. It continues to be embraced in the new organizations as well. With 350 to 400 TOC projects annually, we believe Lucent delivers the largest number of such projects in the world.

The commitment to quality has not changed, although the way it is accomplished has changed. Now, a distributed model is used in which each area is responsible for doing root cause analysis and driving corrective action. A small core quality team exists for the purpose of tying these pieces together and for maintaining ISO 9000/TL 9000 certification. The result is fewer people coordinating and more people personally taking action to improve or maintain quality.

Software Development:
As next generation telecommunication product lines were started during this time, new organizations were created to support these. Although the staff for these new organizations came from the once large project, it was interest-

ing to see how quickly and happily the staff relished their freedom and abandoned the desire for process controls. This is understandable: to produce a high quality, on time project in a large organization, procedures and controls were needed to allow thousands of developers to successfully submit code on the same load line or on a dozen parallel load lines. With a small team, these same developers need a simpler process. This supports the Agile principles that larger teams need heavier methodologies, as was the case with 5ESS, and that excess methodology weight is costly, and thereby to be avoided in the smaller organization [5].

Start ups of new product lines during this period did not implement the checks and controls in place in the large organization, choosing face-to-face communication instead, in line with another Agile principle that this is the fastest channel for exchanging information [5]. In the large organization, to make a change in code that affected data, a tool notified a core set of developers, who each approved the code prior to submission. The hundreds of developers in the 1980s and 1990s and constant learning curves from the influx of new developers made this necessary. While appropriate when introduced, and still in use, this tool is now being reevaluated. The balance between assuring consistency between developers, and the flexibility and speed to submit code is being reassessed.

In other cases, the process did not change, but the execution of the process did. For example, document reviews and code inspections are still required. With smaller teams, individual peer reviews have become more prevalent. Although years of data show that meetings are more effective for finding faults (due to the synergy effect), the team size allows closely working together and individual reviews have been adequate in more cases. Switch availability over 99.9999% is still produced in the field.

With thousands of developers, manually monitoring quality execution (e.g., determining if staff are giving in to the temptation of skipping a code inspection) became a large job. A tool was created to make sure a code inspection record existed before allowing code submission. But in the new, relatively small organizations, this can again be handled manually.

As the organization shrunk, considerable training was needed due to the collapsing of jobs and the learning curve as staff picked up work. In addition, training was needed as the team was growing in other regions of the world. This globalization during a shrinking period presented managers with significant challenges. Experienced people were training staff in other countries, while they themselves took on new responsibilities. It became obvious which tools and processes were the least user friendly during this time. For example, the volume of internal help requests for the commitment database peaked. This was a challenge, because when trying to maximize product feature content as staff declined, user interface enhancements to internal tools are low priority.

Software Construction:
We moved away from a central software construction team. This may seem counter-intuitive, since a central team eases personnel coverage/backup concerns, and generally ensures staff and computer utilization is kept high. However, we decided that a distributed load building model better met our needs. Each organization gained flexibility and speed from load building catering to that specific organization's needs. This is particularly beneficial in the next generation organizations as code development has migrated toward the XP principle of small, frequent releases [3].

Test:
In the drive for more efficiency, with consequent lower costs and shorter time to market, another change implemented since the early 2000s was the use of Cross Functional Teams for a development project. The idea was not new; however, true participation by system testers and customer technical support personnel has historically not occurred until the feature arrived at their door. In the early 2000s, these Cross Functional Teams became a reality. System testers and customer technical support members joined with coders and system engineers in the requirements and design stage for reviews, and began to write test plans at this early point, supporting an XP principle [4]. This reduced disconnects late in the project, and reduced the overall testing effort.

Conclusion:
The resulting 5ESS organization is quicker to release quality features to meet customer needs. Building on the years of process and quality management, coupled with the current market needs, 5ESS is well positioned. The newer, smaller teams on the next generation of telephony switches have strong quality and process heritage, and achieve fast time to market, with smaller processes.

PROCESS CHANGES IN THE TAQUA BUSINESS UNIT OF TEKELEC:

History and Introduction:
Taqua Systems was founded in 1998 to build a low cost, compact, Class 5 switch. The number of developers and testers was small, and venture capital funding was based on delivering many rapidly developed prototypes. The software development processes were initially ad hoc. In 2001, the first customers installed switches in small, but live, central offices. Product quality was as expected for an initial release. By the end of 2001, due to the economic climate and its impact on venture capital, Taqua Systems ran out of money.

In January 2002, Taqua Inc. was recapitalized with new venture capital, and new leadership was brought in to improve product quality and delivery performance. The software development and testing processes needed dramatic improvements, and the initial focus incorporated TOC/CC into the organization structure, product management, and project management. The strategy was to evolve the processes through practice, and not spend a lot of effort on process formalism or in process "meddling" [8]. As DeMarco and Lister [9] observe

"Voluminous [methodology] documentation is part of the problem, not part of the solution" and "The total of all standards imposed should be described in no more than ten pages".

The discipline of the development and testing organization matured and improved as a result of root cause analysis and release postmortems. The quality improvements were typically implemented via automated controls, or by simply communicating to everyone on the team that we were going to change a particular way of doing things, rather than by adding additional text to process documentation. The results were dramatic improvements in product quality with each subsequent release as well as dramatically increased development throughput (eight major releases were delivered in 18 months). The processes as executed were also very repeatable.

In April 2004, Taqua was acquired by Tekelec, and the product renamed the Tekelec 7000 Class 5 Switch (T7000). Using these processes and practices, the T7000 R&D team has produced a 99.999% available Class 5 switch, in service with over 100 customers.

Front End:
Release Planning is managed by the Project Office and brings input from Engineering (hardware, software, and systems), Product Line Management (PLM), and Sales, with PLM the final authority. The release scheduling targets two or three releases per year, although additional incremental releases are possible, if sales opportunities exist.

The release candidates are managed through Release Candidate forms which capture market requirements, business opportunity, rough order of magnitude development estimates, and an abbreviated business case. These release candidates are prioritized by PLM for a particular release, and further estimation is performed.

The product roadmap typically looks two years into the future. The contents of the roadmap are re-evaluated as each release is estimated and committed (e.g., to move unselected features to later releases). The roadmap primarily includes "larger" market features that would require the majority of development resources to address. The market features for a release define the "theme" for a release.

From an Engineering perspective, the initial release planning provides a prioritized list of features which need additional estimation and brainstorming. The list is initially pared to roughly twice the available development resources (i.e., if there are fifty candidates, and the first ten over-commit Engineering by 2×, then these first ten get additional estimation). The 2× idea is that by providing better estimation on these items, the final priority of the features at release commitment may differ. This is very similar to the XP "planning game" [3], [4].

The final selection of features is matched to the resources available in development. Engineers then are identified by name (and skill pool match) and *exclusively* dedicated to a feature being developed over the life of requirements, software development, and testing. This is a TOC/CC concept [2]. After feature testing, the development team is available and redeployed to other features in this or future releases. Some of the developers will follow the release into technical support, where they will stay for roughly one year.

Feature progress is tracked at a macro level by the Project Office (at weekly Core Team meetings) and is project managed by the responsible software development Director.

Requirements Generation:
Requirements are produced by Systems Engineering (in the PLM organization) and use customer interaction and standards (ITU, Telcordia, IETF, etc.) as input. The requirements are "tagged." System level interaction is typically described in terms of message sequence charts.

Software Design:
Both Data Design and High Level Design are performed and documented by the feature team as part of High Level Design (typically via whiteboard presentations and working discussions). The high level design includes object definition. Objects are defined by procedural behavior (a calling feature/service described by a state machine) or as persistent data (with additional dynamic attributes). Designs (and their coding) are described via state machines. The high level designs include message sequence diagrams and state models. Object definitions, state machines, states, and events are described in the high-level design documents.

Software Development:
Data and code are produced at the same time. There are coding and data modification rules (e.g., all static variables must be explicitly initialized, new attributes and enumerations are added to the end of existing structures, etc.). There are a number of static code checkers / analyzers in place, which check for problem areas found in the past. These were developed in-house, by developers supporting the field.

Code inspection is done through formal meetings. The inspection is done with a laptop and a projector on a screen, with the author serving as "reader." The moderator also serves as the recorder. Alternatives and solutions are discussed at the meeting.

White Box testing using a software simulator is done prior to code inspection. White Box testing is performed using a software simulation of the T7000, which can run on any PC (laptop, desktop, at home, etc.). The developer must assure that his code does "no harm" to the mainstream. Automated No Harm (regression) testing is performed prior to approval of code submitted to the mainstream code base.

Software Construction:
The basic tenet in software construction is "there is one and only one code stream". So, as a rule, the next release's branch is only created after a release moves to Controlled Introduction (CI). There have been small exceptions to this (an earlier than CI branch for the next release), but the merge back to a single stream occurs as

the first order of business when the previous release is declared CI.

Load build frequency depends on code inflow rates. At the beginning of a release, there are weekly builds. When the code submission rate increases, builds are two or at most three times a week. In some cases, a build (and associated "no harm" testing) is reserved for a particular feature. The idea in all cases is to keep the code delta per build at a low level, to reduce complexity and the likelihood of breakage. This is both a TOC/CC concept [2] and an XP concept [3].

Load building is automated, with one person supervising the activity. A full system build takes three hours. Release notes are automatically generated as part of the code extraction process, along with the NCSL counts, and distributed to all engineers.

After the load is built, overnight testing determines a "stability index." If the index is reduced (which rarely happens), all submitting developers are brought in to determine the cause and implement a fix. All other code submissions and no harm testing are halted until this is resolved. This is similar to the Microsoft process described in Brooks [6].

Test:
The software test management team plans needed test environments, based on the features on the release roadmap. The resulting plan is implemented by a lab technician.

Feature tests are reviewed and approved, and placed into an on-line test management system that is the sole manual testing tool used by testers in the lab. The test scripts are written so that it is easy to automate the tests. Some features utilize 100% automation of their test plans.

During feature test execution, if a test fails, all testing on that feature or switch halts until the developer finds / debugs / understands the failure. The philosophy is to not create a mountain of change requests for a feature under test (i.e., the testers are not held to an expectation on test execution speed). The last one across the finish line (developer or tester) is the one that declares the finish time. This is the famous "we're not at the top of the mountain until Herbie gets here" story from [1] – a TOC/CC concept.

A new "fast feature team" is being developed, and will utilize pair-wise development [4] and small features [3], both XP concepts. The XP "write the test before the design and coding" concept is also attractive and will be considered.

Conclusion:
A small, highly-focused development organization is an ideal setting in which to utilize the principles of the TOC/CC, XP, and Agile development models. Taqua's success in product development, addressing customer needs, and field quality, shows these models are well suited for this environment.

CONCLUSIONS:

In both the Lucent organization and in Taqua Systems, major industry and market changes, coupled with significant internal changes, drove changes to the software development processes. The resulting development models are surprisingly similar, although one organization moved from a less formal to more formal process, to address needed delivery fidelity and quality improvement, and the other organization in the other direction, to address changing customer buying levels. In the latter organization, many process steps were reevaluated to ensure the original precipitating reasons continued to apply, and that the steps were still appropriate ways to address the problems. Many of the changes in both organizations are consistent with those discussed in the TOC/CC, XP and Agile development models.

Virginia Satir's change model [9] posits that change never proceeds from the old status quo to the new status quo. Instead, change proceeds from the old status quo to chaos (induced by some foreign element), then to practice and integration (induced by some transforming idea), then to the new status quo. The foreign element can be an outside force, or the recognition that the world has changed. This fits exactly the experiences reported in this paper: business stress drove process changes that are similar or identical to those proposed in TOC/CC, XP and Agile. The organizations on which we report are more productive and energized than before the change.

REFERENCES:

[1] E. M. Goldratt, *It's Not Luck*, North River Press, Great Barrington (1994).

[2] L. P. Leach, *Critical Chain Project Management*, Artech House, Boston (2000).

[3] K. Beck and M. Fowler, *Planning Extreme Programming*, Addison-Wesley, Boston (2001).

[4] D. H. Steinberg and D. W. Palmer, *Extreme Software Engineering, a Hands-On Approach*, Pearson/Prentice Hall, Upper Saddle River (2004).

[5] A. Cockburn, *Agile Software Development*, Addison-Wesley, Boston (2002).

[6] F. P. Brooks, Jr., *The Mythical Man-Month: Essays on Software Engineering*, Addison-Wesley, Reading (1995): 90-92.

[7] R. K. Wysocki and R. McGary, *Effective Project Management*, 3rd ed., Wiley, Indianapolis (2003).

[8] W. E. Deming, *Out of the Crisis*, MIT Center for Advanced Engineering Study, Cambridge (1982).

[9] T. DeMarco and T. Lister, *Peopleware: Productive Projects and Teams*, Dorset House, New York (1987): 116.