Engineering Quality while Embracing Change: Lessons Learned

M.C. Marinovici, H. Kirkham, K.A. Glass, L.C. Carlsen
Pacific Northwest National Laboratory, Richland, WA
{cristina.marinovici, harold.kirkham, kevin.glass, leif.carlsen}@pnnl.gov

Abstract

In an increasingly complex technical environment, failure is accepted as a way of maximizing potential, a way of growing up. Experience can be utilized to improve designs, advance product maturity, and at the same time, can increase team’s training and education. It is not enough to understand the development tools to ensure a project’s success. Understanding how to plan, measure, communicate, interact, and work in teams is mandatory to make a project successful. A manager cannot enforce a process of good communication between team members. Project teams have to work together in supporting each other and establish a constant communication environment. This paper presents lessons learned during the development process of operations research software. The team members have matured and learned during the process to plan successfully, adapt to changes, use Agile methodologies, and embrace a new attitude towards failures and communication.

1. Overview of Paper

The need to embrace change becomes an accepted fact of life in software development environments. Any software project is subjected to continuous changes as it supports evolving user needs. Diverse factors make people change their minds, and these changes affect the development process.

As a first step in adapting to change, the software team may shift gears, self-organize and adopt the progressive, iterative and incremental development approach known as Agile. Since the standard methods may not work, the team might experiment in adjusting the methods until they find what works best for them. At this point, the knowledge gained can be utilized to improve the project’s state of art, practice, and the team’s education and training.

In this paper, the reasons for change in some software projects are identified and their effect on the team’s activities are examined. Therefore, in Section 2 of the paper a brief description of a particular scientific software project and its development path are presented. In Section 3 and Section 4, the software engineering is detailed and analyzed, pointing out the changing points. The final part of the paper summarizes the lessons learned. Readers who develop software for scientific projects may find similar situations in their projects and may be able to use our learned lessons in their own development practices.

2. Introduction: the project

The growing penetration of intermittent resources, such as wind and solar power generation, makes planning and operating the electric grid more challenging than ever. Decisions must be made at time-scales ranging from minutes to years. In operation, power grid operators adjust generation to balance the continuously changing load. In the past, the load on the power system has varied with customer needs, but the generation has been completely under the control of the power system. With intermittent renewable generation, not only is the load subject to being changeable, so is some of the generation. Operators have to maintain a state of balance by monitoring system frequency, by starting and controlling generator units over a time-horizon of hours, and doing so at minimum cost.

To address the challenge, operational research software was planned. The aim of it was to allow a user to quantitatively analyze the impacts of resources with unpredictable supplies and to develop strategies to deal with how this unpredictability affects operations. The model can be used to evaluate an existing or future generation mix and take into account the effect on operations of unpredictable supply of various resources.

The software was divided into two parts: the operational problem and the “model.” The operational problem is one of optimization and it is divided in three sub-problems, each an optimization,
at a different time-scale. The optimizations at longer
time-scales are used as inputs to shorter scale
optimization. This approach, which emulates the
behavior of actual power delivery operations, makes
the model more detailed as execution proceeds.
The “model” is the set of calculations that play no
role in the optimizations, but allow calculation of
various performance measures, or (for example) the
carbon produced in the system operations.

Looking retrospectively at the project, one can
see that from the beginning, the project exhibited
some of Mike Cohn’s problems of project
management [2]. The project tried to do too much for
the resources and schedule available, it had an
unrestrained craving for features, and it believed it
knew everything about how to build the product.
The software development for a scientific community is
challenging for software engineers and poses two
major problems: requirements development and
understanding between the engaged parties [3].

At its start, the project was designed as a set of
separate functional blocks prototyped using
MATLAB routines. Usually scientists start by
developing a prototype and when the complexity of
the product requires higher expertise, it had an
unrestrained craving for features, and it believed it
knew everything about how to build the product.
The software development for a scientific community is
challenging for software engineers and poses two
major problems: requirements development and
understanding between the engaged parties [3].

A direct consequence of the improved
understanding was to change the solution method to
an agent-based simulation. Given the new approach
and the required problem sizes, the development
team chose C++ language and the commercial
optimizer solver IBM ILOG CPLEX to implement
the simulation. At this point, the development of
software requirements construction began. Once the
software engineering process was started, it became
evident that the scale of the system had been underestimated in the original planning.

New management was introduced, and it was
realized that one of the three “iron triangle” critical
factors – scope (features, functionality), resources
(cost, budget) and schedule (time) – should be varied
in order to have a successful project of a good quality
[4]. The decision was made to de-scope. The iron
triangle became a more elastic triangle.

Consequently, the outside observer (non-
participant) perceived a slowing of progress, and
pressure was brought to cut corners and produce a
“scaled-down” version of the product. In fact, this
pressure resulted in a reduced-capability model that
was reasonably functional, but which could not
readily be taken to complete the overall task.

As the development proceeded, new require-
ments were added to the original problem, which
inevitably altered the earlier efforts. The lack of a
complete set of requirements made the code
inflexible as new requirements were introduced. At
this point, it was understood that requirements are
what defined the work going on in the project, and
their definition became top priority. External factors
d dictated changes in the way the things were thought
to be and influenced specifications, making the
requirements a living document. Dealing with
specifications that are continuously improving means
continuous software changes. Accepting changes was
implicitly making the delivery target slip.

Looking at the motivations behind the software
changes in our project, the same reasons as the ones
listed by Edberg and Olfman in their paper can be
identified [5]:

- external changes - changes required to meet
  some external environment or outside-the-
  organization needs. This sort of change or
  new requirement might be at a high level
  (sponsor) or lower;
- internal changes - changes required because
  of company changes, such as management
  changes or restructuring;
- technical changes - required to meet new
  technical demands;
- learning changes - due to gained knowledge
  as a result of individuals or group learning,
  (i.e., second system development – improved
  version of the old system).

During the development of the simulator,
another reason for software change surfaced. There
was a communication problem. Team members did
not have a common vocabulary and did not have a
shared perspective on the software development
process. Hence, misunderstandings arose related to
the software’s feature set, priorities, architecture and
structural requirements. These misunderstandings
caused several false starts and unproductive project
tasks. Clearly, team internal communication affects
the productivity of the project and the knowledge
exchange process is reduced.

In the following section each of these points will
be analyzed in detail.
3. Challenges in the Engineering

As in any evolving process, in software engineering new technologies and practices are adopted, while the old ones are abandoned and even deprecated. It is considered that software engineering can be divided into ten sub-disciplines [6] as shown in Table 1.

Table 1. Software engineering sub-disciplines

<table>
<thead>
<tr>
<th>Discipline</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software requirements</td>
<td>elicitation, analysis, specification, and validation of requirements for software</td>
</tr>
<tr>
<td>Software design</td>
<td>the architecture, components, interfaces, and other characteristics of a system or component definition</td>
</tr>
<tr>
<td>Software construction</td>
<td>create working, meaningful software through a combination of coding, verification, unit testing, integration testing, and debugging.</td>
</tr>
<tr>
<td>Software testing</td>
<td>dynamic verification of the behavior of a program on a finite set of test cases, against the expected behavior.</td>
</tr>
<tr>
<td>Software maintenance</td>
<td>activities required to provide cost-effective support to software</td>
</tr>
<tr>
<td>Software configuration management</td>
<td>systematically controlling changes in configuration, maintaining the integrity and traceability of the configuration throughout the system life cycle</td>
</tr>
<tr>
<td>Software engineering management</td>
<td>planning, coordinating, measuring, monitoring, controlling, and reporting activities to ensure that the development and maintenance of software is systematic, disciplined, and quantified</td>
</tr>
<tr>
<td>Software engineering process</td>
<td>definition, implementation, assessment, measurement, management, change, and improvement of the software life cycle process itself</td>
</tr>
<tr>
<td>Software engineering tools and methods</td>
<td>computer-based tools and methods which impose structure on the systematic software engineering activity</td>
</tr>
<tr>
<td>Software quality</td>
<td>degree to which a set of inherent characteristics fulfills requirements</td>
</tr>
</tbody>
</table>

Not all software engineering endeavors can be divided this way; the needs of each project define the application criteria and boundaries. In what follows, the hurdles of the project engineering are detailed, focusing on the sub-disciplines subjected to continuous changes.

The software requirements definition was the first serious system engineering on the project. Initially requirements were written in natural language. Each of the requirements was put into one of the below four levels:

- At Level 1 were requirements that derived directly from the sponsor. For example, the model had to be accessible to users who were outside our laboratory.
- Level 2 collected together the requirements that the user might have on the system. For example, the ability to retrieve any calculated value (i.e., any parameter) after running the model.
- Level 3 requirements were system requirements, applying to the entire system, and typically having to do with the design. An example of a level 3 requirement is that the user shall be able to calculate reserve requirements according to a standardized method.
- At level 4, the implementation is specified in more detail. For example, there are level 4 requirements that are applied to operations, unit commitment, economic dispatch, generation and so on.

While the requirements writers (power engineers) viewed them as acceptable, the software team viewed them as ambiguous or incomplete. The process of trying to resolve the communications gap and clarify the requirements revealed things that were not accounted for. Therefore, specification changes were inevitable.

Due to the time and budget constraints, implementation of the full-scale system requirements was perceived as a difficult task. The knowledge embodied in the team is easier to materialize following an Agile methodology, than a traditional one [4]. Therefore, employment of the Agile methodology allowed the team to embrace change, and not try to prevent or fight it. In this light, the requirements and test scenarios were linked together, exemplifying the abstract description of rules and conditions. To keep up with the project scope and deliverables, prioritization techniques were applied. When a requirement came in, it was analyzed and prioritized; its value was the most important thing for the development process and the coming-in time does not matter. Based on IEEE Std 830-1998 [7], the
initial prioritization follows the semantic rules presented in Table 2:

<table>
<thead>
<tr>
<th>Priority</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>The requirement is a <em>must have</em> of the on-going construction (essential)</td>
</tr>
<tr>
<td>2.</td>
<td>The requirement is <em>nice to have</em> in the current release (conditional)</td>
</tr>
<tr>
<td>3.</td>
<td>The requirement is <em>postponed</em> to a later release (optional)</td>
</tr>
</tbody>
</table>

As a subsequent step, the requirements were prioritized based on their risk-value and impact on functionality. By functionality, the prioritized work items are viewed as options and the options are managed as a pool. An example of possible pools, adapted from Scott Amber [4] is presented in Figure 1. Stakeholder input is used for defining and selecting the high priority value work.

The software design process was also subjected to change. The initial design had called for a system that would run on a high-performance computer system. Users of the system would specify a model from a remote workstation. A GUI was developed to aid the user in model development. Simulations results, which were expected to store up to 1 TB of data, would be shipped to users for remote analysis.

The system was also expected to use third party data to generate historical and projected data. To support this requirement, the project had to provide networked access to the system. In addition to the remote user requirements, there were requirements to support several different model types. During the course of development, other requirements were added, and then removed from the core requirements.

Improved comprehension of the modeled system, and recognition of scope creep, required a change in whole system design. The project focus shifted toward implementation of the calculation engine with more modularity and flexibility for the final product. A “Version 2” got on the drawing board. The software design in Version 2 was changed. The project experienced the second system effect [8]: a smarter system was developed, based on the initial prototype. Once the development of the smarter system started, many fixes and improvements from the old system were forced in, and the chain of changes continued.

The software development community estimates that about 40% of the development budget is used for code rework due to poor requirements or not getting the requirement “right” in the product design [5]. The software development team’s lack of power system knowledge made the translation of the requirements and tacit power-system practices into software difficult. There were even cases where the interpretation of a requirement was incorrect, and therefore required changes later on.

When the second system implementation started, it was clear to the project team that communication was one of the major project problems. This was a project team observation, where project team means the whole group, including the developers, testers and power-system engineers. In order to overcome this hurdle, the project team decided to follow the Agile methodologies: adopted iterative and incremental development, followed the scrum framework and started pair programming sessions.

This change improved the team dynamic and its communication: the power engineers were working together with the developers and testers to properly translate the requirements into software. Migration from the initial system to the second one also helped individual knowledge to be transferred to the whole group. In many cases, the power-systems engineers discussed the problem to be modeled with the team, helping to choose the optimal approach.

This type of knowledge sharing can be viewed as a way of saving other’s time. Technical literature surveys provided evidence that pair programming is almost similar in cost with the single person programming [9], if you take into account the quality of the produced code. In the case studied here, a productivity increase by employing pair programming was observed; the continue peer-review of the software generated more flexible, cleaner and high quality code. All the team members remain in contact with each other and are aware of the project needs and directions.

In our case, the team gained experience in risk assessments and was able to prevent failures by thinking ahead in the development process. As the system complexity increases, the risks grow proportionally, but some of the failure cases can be
caught earlier if knowledge is made available [10].
As a result of pair programming and collective efforts the defects life-time had reduced visibly compared with the period before Agile methodologies adoption. This fact was supported by studying the life time of a series of defects during a predefined time frame. The graph in Figure 2 shows how applying Agile methodologies improved the average defect time span.

The project had to deal with internal changes too: the management changed three times and the project team went through multiple re-structuring drills.

From time to time, in order to keep up with the project deliverables, the addition of more resources to the project was considered as a solution. It is a well-recognized fact that the benefit of resources increase is not a linear operation; people need time to get up to speed and their productivity is different from person to person. Adding people to a project that is already late may not speed up the process, instead it may slow down even more. A bigger team will require more channels of communication, as presented mathematically by Brooks’s formulation [8]: for \( n \) team-mates, it is necessary to have \( n(n-1)/2 \) channels of communication. In the case of this simulator, the increased manpower represented a “surge.” The long-term picture was one of decreasing manpower.

The project had well defined software testing and software quality strategies, but with so many deviations and changes in the development path, the testing strategy changed too. Shifting the development towards Agile methodologies, where coding and testing are part of the same process, testing became a way of ensuring continuous progress and the quality of the product.

Testing practices from all Agile testing quadrants [11], [12] (see Figure 3) were employed during the project life-cycle\(^1\). The Agile testing quadrants, put together, define the multiple dimensions of the testing problem. The testing process can be viewed from the application or developer point of view, also from the way the things are built or from the point of view of final product and its quality [12].

![Figure 2. Defects life-time curve](image)

**Figure 2. Defects life-time curve**

The software team is likely to be concerned mainly with the bottom half of the chart, while the power system team would be more concerned with the top. In our project, unit testing (lower left quadrant) and performance testing (lower right) were almost exclusively the purview of the software team. The power team was more concerned with functional tests and scenario development.

It is a well-known adage in mission assurance that you cannot “test-in” quality. Quality is something that has to be built in, and software is no exception to that. In terms of the four quadrants shown in Figure 3, the two quadrants on the left can be viewed as the more “creative,” and the likely to contribute to (rather than simply to discover) quality. What this meant for our project was that involving both the software team and the power team in this work was very beneficial.

Since our software product is a complex system, a powerful and very useful approach to testing was represented by exploratory testing. James Bach’s definition of exploratory testing - “the simultaneous learning, test design and test execution” - states perfectly our testing strategy [13]. In many cases little was known about the proper and desired behavior of the software product and exploratory testing was performed. It helped us to define the next step in testing and go beyond the obvious.

The software maintenance and configuration management did not experience too many major deviations. Everything was built respecting the principles of continuous integration: each check-in into the version-controlled repository triggers a

\(^1\) The Agile testing quadrants were introduced by Brian Marick, improved by Lisa Crispin and Craig Brown. The version presented above is based on these Agile testing quadrants [11] but with our own descriptions.
software build. To assure the high quality of work, alignment with the refactoring principles was sought. Every time minor mistakes were found in the code, they were fixed, as part of avoiding the 90% syndrome [14] and the accumulation of bugs.

Focusing on maximizing customer value, the engineering management and engineering process activities were exposed to changes associated with the migration towards Agile and lean software development. While the development team worked on delivering prioritized, valued product features, the management tried to determine metrics around value in the form of test coverage, team velocity, Agile adoption assessment, defect counts, and definition of done. In this effort, the management tried, but was not always successful in avoiding the seven deadly sins of project management [2]. For the whole project team, it was clear that understanding and implementing software engineering principles and other best practices is just as important as specialized knowledge.

4. The Lessons Learned

In hindsight, the project engineering approach could have been handled better. Each type of project requires a different engineering approach. Making mistakes is helping us progress and as James Dyson said “instead of being punished for mistakes along the way, learn from them” [15]. The falling down is part of growing up [1]. A summary of lessons learned while “growing up” is presented in Figure 4.

Better initial understanding of customer’s needs save hours, days, weeks and even months of effort further down the road. Even if the process seems slow at the beginning, a significant increase of speed will be seen at each milestone [16].

Failure to solicit feedback and learn from the customers and teammates creates communication problems, delays and, in the end, unsuccessful products. Communication breakdown should be minimized across project team and all the barriers that affect communication flow should be resolved as they appear. All team members should assure and establish an environment of good communication.

Communication is also the key for avoiding “muddy requirements” [17], by allowing transformation of tacit knowledge, ambiguous assumptions and beliefs into precise specification formulations. The “mud” comes from the different knowledge of the application subject experts and the software experts: communication is the only solution. Engineers must create a functional specification from the marketing concept proposal to document specific features for implementation. The functional specification document should be customer reviewed and validated.

In the software world, project changes (requirements, design strategies etc.) are inevitable. They should not be fought; they should be regarded as ways of assuring quality and progress. The team gains experience in assessing risk and preventing failures. Increase of resources at critical times does not always speed up the progress. Working at a sustainable pace is the key for reducing overtime and product quality surprises. The team velocity shall be used for the incremental workload planning and project duration estimation. The use of team velocity will eliminate opaqueness [2] from the development process and will keep management informed on the project timelines and progress.

Making use of the existing techniques, tools and methods can improve the project development work, it can speed it up by making the activity systematic and ultimately more likely to be successful.

5. Conclusions

There are multiple factors that determine the success of software projects of the same magnitude. Each software project is different and no single method can be prescribed for success. During the development stage, any methods that prove unreliable should be replaced with more dependable ones. Adoption of Agile and lean software development methodologies are ways to provide flexibility and adaptability to change.
Attitudes of risk avoidance and fear of failure do not allow us to consider all the possibilities and alternative solution to our problems. But embracing change and accepting failure is a good way to learn and innovate. Instead of trying to prevent changes completely, the team should work together to manage the changes effectively, while not affecting the project timelines and budget. Throughout the project development discussed here, experience was gained by learning from each other and from our mistakes. The team feels duty-bound to share the lessons learned and project’s reflections with anyone interested in listening.

References


