Requirements Determination is Unstoppable: An Experience Report

Daniel M. Berry
Cheriton School of Computer Science
University of Waterloo
Waterloo, ON, Canada
dberry@uwaterloo.ca

Krzysztof Czarnecki, Michał Antkiewicz, Mohamed AbdElRazik
Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, ON, Canada
{*kczarnec,mantkiew,m2abdelr*}*@gsd.uwaterloo.ca*

***Abstract*—The paper describes the quotations gathered during interviews and focus groups during a consulting engagement to help the client improve its requirements engineering (RE) process. The paper describes also a model of the software lifecycle derived from a Michael Jackson quotation, a model that explains about 95% of the quotations that we gathered. In particular, it explains why basic requirements determination is unstoppable and how management attempts to stop RE lead to the phenomena that are described by the quotations and less than optimal requirements specifications.**

***Keywords*-company's RE process, continual requirements determination, incomplete requirements specifications, late requirements determination, premature termination of RE**

I. Introduction

In a recent consulting engagement, we visited a company, X, that was not satisfied with the progress and results of its software development projects. X is a large company with a well-managed IT department that had delivered award-winning solutions. Nevertheless, the management of X and its IT department understood that software development was not going as well as it could. The management understood that X's requirements engineering (RE) process needed some improvement and asked us to help them identify the problems with the process.

To get to the root of X's requirements process problems, we conducted semi-structured interviews with some of its software development personnel, including its requirements analysts (RAs), its software architects (SAs), its programmers, and its quality assurance testers (testers). We conducted focus groups with some of these interviewees. We collected over 23 hours of recordings, capturing 40 people's words. There were 18 interviews and 5 focus groups.

After recording and collecting these interviews and focus groups and transcribing some parts of them, we were trying to cluster the individual statements into equivalence classes so that we could report to X the set of observed problems and the reporting frequency of each problem. In fact, we ended up with over 700 different clusters. However, we noticed that the statements were telling a story, a story of premature management-dictated termination of the requirements process in order to move quickly on to deadline-driven development. Each of the more experienced of us had seen the story before, and one of us had even participated in a project that suffered the story.

We therefore wrote up and showed to X's management a model of the software development lifecycle that explains the story. This model, built out of what the RE community has learned since the 1980s, explains about 95% of the statements we heard. Section 2 lists some of the quotations we gathered. Section 3 describes the model explaining the quotations in Section 2. In fact, this model is not the complete model we developed. Space limitations prevent us from giving more quotations and more of the model. We selected a subset of the quotations and a submodel that explains the selected quotations. Sections 4, 5, and 6 discuss implications of the model on X's RE process, and Section 7 positions X's RE process within the model. Finally, Section 8 reports some of the comments we received from X's management after our presenting the model.

Some of the quotations in Section 2 are directly from the interviewees and some are from one of us summarizing what the interviewees were saying. Each quotation is labeled by an indication of who said it. In each case of a summary by one of us, in the original transcript, the quoted sentence was followed by a confirmation by a group member or the interviewee that the one of us understood what the group or the interviewee meant.

To maintain anonymity, each proper name, a company's or a person's, is replaced by a different letter, e.g., "A" and "X". To maintain X's privacy, some words of the quotations are changed, specifically to use a more common term in place of an equivalent company-specific term, e.g., with the name of a person's role; these words are italicized. Also occasionally a pronoun phrase is replaced by its referent; these replacements are enclosed in square brackets.

Each occurrence of a role name may or may not be different from other occurrences of the same role. When it is intended that several occurrences of a role name represent the same person, a letter serving as a proper name is introduced to name that person.

II. Quotations and Paraphrases

The quotations and paraphrases are divided by categories.

A. The Main Complaint:

RA: "So there's never money or time put into project for ... keeping [the requirements and design] documentation up to date."

RA: "Give us enough time."

RA: " ... the major problem our *RAs* have is time.... To do requirements properly, you need time to collect data about the current state and to do analysis and document. In the majority of time, the timelines that are set by the executives or other groups do not give our *RAs* enough time to do the job properly...."

RA: "We don't have the time."

RA: "We can't control the timelines that are set by upper management."

Tester: "... Time to market.... Get something working out there quickly.... They are trying to shrink our testing time."

B. The Effects of Not Having Enough Time for RE:

1) Blame:

RA: "Another problem we have is that a lot of times, the finger comes wagging down to the *RA* about a lack of quality of the requirements [that they delivered by the deadline]."

2) Creep:

SA: "*The Programming Team* codes too early, while the documents keep changing. Enough time for *requirements* analysis results in good requirements and has very positive impact on the project."

Tester: "The *managers* have to make their decisions earlier, everyone should participate in JAD [Joint Application Development] sessions, everyone has to see all the output of the sessions, all the minutes, there has to be adequate time for programmers to create the *design specifications* and there has to be enough time for everybody to do the proper review. None of that can happen if the requirements and scope are changing — scope creep."

3) Programmers Guess at Requirements:

Programmer: "Development builds the system guessing [what the requirements are]."

C. Enhancement or Descoping to Deal with Creep:

One of us: "So what do you think about: When the time of requirements elicitation is so short, how much time is required later when they are *developing*, *testing*, or even while in *deployment*?".

RA: "Nine times in ... 10 [, it] ends up as an enhancement, or [it] becomes de-scoped because you can't deliver everything the client wants. So de-scope [it], and it becomes part of another phase or part of enhancement or dropped completely, depending on cost."

D. Change Requests and Stealth Mode Changes

One of us: "The *Change Request* process is perceived very negatively. Raising a *Change Request* is perceived as negative because it implies that somebody did not do his or her job ... properly. Consequently, some system changes are performed in stealth mode, which avoids raising *Change Requests*. In this mode, changes are incorporated into the existing and compatible current work without creating a *Change Request*, code is updated, and the frozen requirements are updated silently, or code is updated without updating the requirements."

RA: "*The quality assurance group* finds out about some undocumented changes, very infrequently."

E. Effect on Testing:

Tester: "Staggered code delivery is a problem. Ideally, all code should come to *the testing group* on day one to enable planning. In reality, code comes in batches and constant regression is necessary. Testing of code batches is necessary because of staged delivery to the branches. It is a ripple effect of *RAs'* not getting enough time for requirements and the *management* finalizing the scope in time for *implementation*.... *The testing group* is forced to commit the timelines too early based on the initial requirements. *The testing group* always suffers the most and has to do the largest amounts of overtime, 250%, and including weekends. The reason for the fixed date is that it is specified in the *development plan* created based on very early and high-level requirements which is approved and provides only a little bit of contingency."

F. When Are Requirements Determined?:

A conversation between a programming leader and one of us:

Programming Leader, L: "... we got the requirements [document], and you know we realize that we have 5% of the actual requirements in the document even though [the RAs] were sure that those are the requirements."

One of us, U: "Because [the RAs] didn't realize the underlying complexity?"

L: "Yes, and [the discovery process] was very much on our part ... as well, because we didn't know the technology ourselves."

U: "This is the prime example of [the requirements problem]. At some point [before] the end of [a] project, all requirements are decided. Sometimes more of them are decided at the beginning; then some of them are decided later on. Then, the question is who makes the [requirements] decisions? Of course, what you are saying is that you always have to go back to [the clients] and ask them. So, you will be discovering new requirements [during] development, but you will be always relying [on the clients to] make the decisions."

L: "I think so."

U: "So [the clients] do the decisions. They do 100% of the decisions."

L: "Exactly!"

U: "But [the RAs] don't discover 100% of the requirements. So, now there is a difference. It is very interesting. We haven't heard this case before. So you were saying, in this case, maybe 5 – 10% requirements were [discovered] upfront, and then you had to discover the [remaining] 90%?"

L: "We had to discover a lot, and we had .... I don't know how many. We had [about 100 and something] project change requests ..."

III. Model Explaining the Quotations

The model described herein is based on what the field of RE has learned, and it explains what is happening to the people we interviewed and thus what they were saying to us. In the rest of this paper, what RAs do is called RE, for "requirements engineering" instead of "requirements analysis", both to match the literature, and to have an acronym for what RAs do that is different from "RA". Occurring many times during RE is instantaneous requirements determination (RD), i.e., the making of some decision even as simple as adding, deleting, or changing one word. It is possible to determine a requirement without having done any analysis.

Throughout the transcript is voiced the complaint that management does not give the RAs enough time to do the RE, in particular to determine the AS-ISs and the TO-BEs. Basically, the RAs do not get enough time to determine the requirements for the systems to be built.

Management seems to be operating under the assumption that it can control how much RD is being done. The reality is that there is no escaping doing enough RD to write the code, and if not enough time has been allocated to allow the RAs to do enough RE before writing a requirements specification (RS) to be given to the programmers and testers, then the programmers and testers will do the additional needed RD as they do their jobs.

In other words, there is always enough time to do the needed RD. The project makes enough time whether management has allocated it or not.

Michael Jackson [1] once said that "Requirements engineering is where the informal meets the formal." Figure 1 shows the project timeline, with the raw client ideas on the far left and the code and test cases on the far right. A test case is an input and its expected output.

The client ideas are necessarily informal. They may not have even been put into words. They are just thoughts. The code and test cases are necessarily formal, not just because a program is as formal a language as any mathematical notation, but also because the meaning of any utterance in code and in input date for a program is fully defined, in the first case, by what the machine code generated by the compiler does, and in the second case, by the output generated by the executing program reading the input.

Somewhere along the project timeline, the informal client ideas must be converted into something formal enough that the code and test cases can be written. Short of not writing any code or test cases, this conversion is inescapable. Note that to do this conversion, the requirements for the code must be understood. One cannot write code without knowing what the code is supposed to do. One cannot write a test case without knowing what output to expect from the code's reading its input. That determining requirements at some point is unavoidable is one point of the Jackson quotation.

There are two extremes about when this conversion happens:

1) at the far right, during coding and test case writing. Each programmer determines for herself what the code she is writing does at each point in the coding that a decision about the code's behavior must be made. Each test-case writer determines for himself what the expected output is for the considered input. In this case, the code and test cases end up being the RS. We call this extreme "RD during coding".
2) at the far left, when the client is presenting her raw ideas to RAs. The client and RAs discuss the system thoroughly, with the RAs asking the client what she means any time they do not understand what they have heard so far. Not understanding includes the RAs' realization that there are details missing from what they understand. The output that the RAs produce is a RS expressing as precisely as possible the requirements for the system that they believe the client requires. Ideally, the programmers should be able to write the code directly from the RS without having to make any requirement decisions, and the test-case writers should be able to write each of a covering set of test cases from the RS without having to make any requirement decisions. We call this extreme "upfront RE".

In between these extremes, there are countless other places in which some project personnel may make a requirements decision to fill in on details that are not captured in

Figure 1. Project Time Line and Requirements Engineering

whatever RS is generated by the RAs after whatever RE they have managed to do.

Thus, no matter what, if client ideas have been converted to code and test cases, requirements had to have been determined.

In most system development projects in most organizations, there are more coders and test-case writers than RAs. We would guess that about a half an order of magnitude more coders and test-case writers than RAs is typical. On the assumption that there are more coders and test-case writers than RAs:

1) The number of people doing RD in RD during coding is larger than in upfront RE.
2) The total time spent doing RD in RD during coding is larger than in upfront RE.
3) The number of uncoordinated, independent requirements decisions being made in RD during coding is larger than in upfront RE.
4) The people making the decisions in RD during coding have interests other than those of the client: each programmer tries to simplify her coding and each test-case writer tries to minimize the number of exception cases to be generated. Therefore, the number of decisions that are made according to what the client wants in upfront RE is larger than in RD during coding.

The questions to ask about these kinds of RE are:

- Which kind of RE is more likely to yield higher quality, robust, non-brittle code?
- Which kind of RE is more likely to yield code that meets the client's requirements?
- Which kind of RE is more likely to yield high quality code that meets the client's requirements with a lower expenditure of people time?

There are very early and recent data that show that a large majority, from 70 to 85%, of all defects found in running software can be traced back to RE [2, 4] and that 70–85% of total project costs are rework due to requirements errors and new requirements [4]. There are data, a sample of which is shown in Figure 2 [3], that show that the cost to fix a defect grows dramatically with the lateness of the lifecycle in which the defect is detected and that fixing a defect after delivery costs two orders of magnitude more than fixing it it at RE

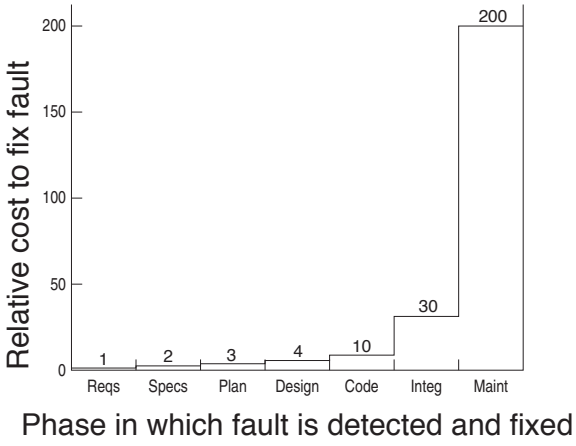

Figure 2. Cost to Repair Defects Over Lifecycle

time. The later a defect is repaired, the more artifacts and the more of each artifact mentions the defect, and therefore the more artifacts and the more of each artifact must be fixed when the defect is fixed. Therefore, to reduce defect repairing costs and time, it is essential to make RE strong enough to detect all errors during RE time so that the RS goes out with zero defects. Many a Change Request at X represented the expensive repair of one or more requirements defects during a late lifecycle stage.

Moreover, sometimes, the programmers are in a continent far away from Canada, making direct communication with the client even more difficult and making the completeness of the RS all the more essential.

IV. Agile vs. Waterfall Lifecycles and RE

When one uses an Agile lifecycle, the entire line in Figure 1 is very short, e.g., two or three weeks. In such a situation one can afford to keep the client accessible on a moment's notice to ask him or her questions[1]. In this situation, it is OK and expected that the programmers writing the code

[1]Of course, keeping the client around for all iterations in an Agile development can be very expensive, but a different client representative can be present for each iteration, as a function of the domain expertise needed for the iteration.

will ask the client questions that resolve requirements. In other words, the programmer is doing RE during coding. The main difference between this Agile situation and the typical waterfall situation is that the scope of what is being implemented is small enough that it can be fully understood and implemented in the short period available. This small scope makes it feasible to have the client available on short notice and that availability, in turn, makes it feasible for a programmer to do RE by just asking the client what the program being implemented should do in the situation that the programmer has identified. On the other hand, the X manager that engaged us does not believe that an Agile lifecycle can work effectively because it is often used as an excuse not to do full RD, since things can always be fixed in a later iteration.

In a typical Waterfall situation, the scope of the program is much bigger. The much bigger scope makes it infeasible for the client to be available on short notice for the entire development. Hence, there is a distinct RE phase in which RAs and the client jointly determine the requirements thoroughly enough to produce a RS that allows the programmer to code without having to ask questions of the client. When the RE phase is cut short by external management decision, then the programmer is left with an incomplete RS. The programmer ends up having to determine requirements in order to proceed with coding, which, in the best case, introduces delays and in the worst case, results in incorrect requirements being determined.

V. Requirements Creep

Requirements creep (creep) is new requirements that are not derivable from what is already known. According to this model, it becomes apparent that not all of what is called "creep" is really creep. Some of the apparent creep is a simple consequence of the prematurely ending RE and this apparent creep amounts to the completion of an incomplete RS. The true creep is *unavoidable* creep, and the apparent and not true creep is *avoidable* creep.

Unavoidable creep is creep that comes from just not knowing what the software is supposed to do. Prototyping and iterative development are intended to deal with this category of creep. This creep is unavoidable because no amount of RE is sure to find unknown requirements. They need to be discovered by users using some version of the system, during which time their need becomes apparent. These are the kind of requirements for which one says "IKIWISI" ("I'll know it when I see it.")

Avoidable creep is creep that comes from cutting RE short, and one does not have a complete RS that allows the programmer to program without determining requirements. The requirements discovered by the programmers are called "creep" because they appear to be new requirements; and indeed they are. However, they would have been found during RE had RE been allowed to proceed to its natural conclusion.

It would be useful to analyze some project histories to determine what percentage of what was thought to be creep is of each category. Our bet is that for X, and for most institutions that cut RE short, a majority of the creep that occurs is avoidable.

It seems to us that each lifecycle model has its place and for any development, there is a better lifecycle for it. We believe that when requirements are not well understood because the software being developed is for a totally new domain, much of the creep will be unavoidable, and an iterative or Agile lifecycle will help tame that creep. We believe also that when requirements can be well understood because the software being developed is for a well-known domain, a Waterfall lifecycle with upfront RE carried out to its natural conclusion will help avoid avoidable creep.

The key is that whatever lifecycle is chosen, its RE must be allowed to run its course, and the client must be available for consultation the whole time of RE. If a waterfall lifecycle is followed, then if RE is not allowed to run its course, then RD continues anyway, but

1) the wrong people do the RD,
2) they do not have access to the client, and
3) the newly discovered requirements are called creep when they are really only the requirements that were not found by the time RE was cut short.

If an Agile or iterative lifecycle is chosen, then the development must have a scope that is small enough that the client can be readily available for questioning during the whole development. The programmers are doing RE as they write code during the whole development, and the programmers ask the client to resolve all requirement issues. Thus,

1) the right people do the RE,
2) they have and make use of access to the client to ask questions, and
3) there is no avoidable creep.

Of course, there is unavoidable creep in the sense that each development deals with some requirements that were not considered before, that emerged as a result of the client's seeing how a previous development's product works.

One reason cited for not being willing to spend more time on RE is that there is no apparent end in sight for continuous RE, especially once the need for an iterative approach is identified to allow X's software to keep up with the ever-changing market.

On the other hand, it is recognized that a lot more can be done than is currently done, especially to discover those missing requirement and requirements defects that are eventually found during coding of the RS and that existed at the time RE was terminated. These late-discovered requirements are thought to be true creep, but are really avoidable creep.

The idea is to recognize that there are two kinds of analyses going on during RE:

1) one to determine the scope, i.e., feature set, of the system to be built, and
2) one to determine the details of requirements within any given scope.

There is no end possible for the first kind of RE. One can always add more features to any scope and one can always find variations of any scope that achieve the same functionality. However, deciding whether any scope is right requires building that scope and letting users have a go at it. So, it is necessary to choose a scope based on whatever information is currently available and to resist temptations to modify it or add to it.

However, there is no excuse to proceeding to implementation until all of the requirement defects of the chosen scope have been discovered by thorough RE and until the RS is such that a programmer can code the software without having to ask questions or to make requirements decisions. Proceeding before these details have been worked out creates a situation in which RD is done by the wrong people, too many times, redundantly, inconsistently, and taking more time than needed and in which correcting defects is done at a significantly higher cost than needed.

VI. How To Know When to Stop RE

The model carries in it a criterion that allows determining when to stop doing RE on any given scope: RE is finished when the RS that is produced is sufficiently detailed and validated that (1) a programmer may code the specified software without having to decide requirements and without having to ask questions and (2) a tester can write test cases for the specified software without having to ask questions. That is, RE is done when the resulting RS is self contained.

In order for the RAs on the RE team know when the RS meets the criterion, the RE team needs to have at least one person from the eventual coding team and at least one person from the eventual testing team as members to inform the team when the RS meets the criterion. It is our experience as programmers and testers that programmers and testers instinctively know when this criterion has been met for any RS.

VII. Clarification About X's RE

Lest the reader get the wrong impression about X's RE process, it is necessary to state that everyone at X understood the implications of Figure 2 and the importance of doing as much upfront RE as possible. Upfront RE was the norm, but it was being terminated prematurely in many cases. The quotations we saw indicated that the typical premature termination was motivated in large part by the pressure of the impending agreed-to final delivery date. Other quotations indicated that some feared that if upfront RE were not terminated by decree, it would continue indefinitely, i.e., some people did not understand how to know when RE is done.

VIII. Client's Comments About Our Model

After completing our report giving the clustered quotations and their frequencies and describing the model, we met with some of X's vice presidents (VPs) and senior VPs to give a short 15-minute presentation. We believed that the VPs were expecting us to present a summary of the data, showing how many times each unique idea was mentioned in the interviews and focus groups. During the planning for the presentation, we realized that what we thought they expected would be nothing new, bore them, and turn them off from listening to us. In fact, we thought that the reason that they limited our presentation to 15 minutes is that they were trying to limit, in advance, how long they would have to sit through a boring presentation. We gambled that presenting the model would pique their interest. So, we used the 15 minutes to present only the full model.

The gamble seemed to have paid off. The VPs seemed surprised at what we presented, perhaps having never seen anything like it before. The two X people who engaged us said that the VPs had certainly never seen Figure 1 described this way before, but they had seen and had internalized Figure 2.

After the presentation, the VPs started to discuss the model. The discussion was so good that it was apparent that they found the model to be novel and interesting. In any case, they said that the consulting work that we did was valuable to them, because they got an independent assessment and summary of what they already intuitively knew, but the models provided them with a tool with which to reason and make decisions about RE practices.

After the meeting, one of the VPs approached us to say that we had hit the nail on the head.

We know that what we observed in Company X is very common. Moreover, we suspect that it is worse in most companies, because as mentioned, X's IT department was particularly well managed. Indeed, our own limited personal experiences in industry support that suspicion.

Acknowledgment

Daniel Berry's work was supported in parts by a Canadian NSERC grant NSERC-RGPIN227055-00. Krzysztof Czarnecki's work was supported in parts by a Canadian NSERC grant NSERC-RGPIN262100-07.

References

[1] M.A. Jackson. Problems and Requirements. Proceedings of the Second IEEE International Symposium on Requirements Engineering, pp. 2–8, IEEE Computer Society Press, March 1995

[2] B.W. Boehm. Software Engineering Economics. Prentice-Hall, Englewood Cliffs, NJ, USA, 1981

[3] S.R. Schach. Software Engineering, Second Edition. Aksen Associates & Irwin, Boston, MA, USA, 1992

[4] K. Jackson. Tutorial on Requirements Management and Modeling with UML2, Proceedings IEEE International Conference on Software—Science, Technology and Engineering, 2003