# Case Studies in Just-In-Time Requirements Analysis

Neil A. Ernst, Gail C. Murphy
*Department of Computer Science*
*University of British Columbia*
*Vancouver, Canada*
*nernst,murphy@cs.ubc.ca*

***Abstract*—Many successful software projects do not follow the commonly assumed best practice of engineering well-formed requirements at project inception. Instead, the requirements are captured less formally, and only fully elaborated once the implementation begins, known as 'just-in-time' requirements. Given the apparent disparity between best practices and actual practices, several questions arise. One concerns the nature of requirements engineering in non-traditional forms. What types of tools and practices are used? Another is formative: what types of problems are encountered in just-in-time requirements, and how might we support organizations in solving those problems? In this paper we conduct separate case studies on the requirements practices of three open-source software projects. Using an individual task as the unit of analysis, we study how the project proceeds from requirement to implementation, in order to understand how each project manages requirements. We then comment on the benefits and problems of just-in-time requirements analysis. This allows us to propose research directions about requirements engineering in just-in-time settings. In particular, we see the need to better understand the context of practice, and the need to properly evaluate the cost of decisions. We propose a taxonomy to describe the requirements practices spectrum from fully formal to just-in-time.**

***Keywords*-requirements, analysis, agile**

## I. Introduction

Both industrial experience reports[1] and academic research (such as [1]) have identified a significant set of software projects for which traditional notions of requirements engineering (RE) are neither appropriate nor useful. In these settings, requirements still exist, but in different forms than what requirements textbooks typically characterize as best practice. These requirements approaches are characterized by the use of lightweight representations such as user stories, and a focus on evolutionary refinement. This is known as *just-in-time* RE [2]. We conducted three case studies to study 1) how these approaches are used in practical settings, including what tools and methods are represented; and 2) the types of problems, if any, encountered by such projects in managing requirements.

In contrast with just-in-time RE, many other approaches to RE, such as IEEE-830 ("Recommended Practice for Software Requirements Specifications"), involve heavyweight process and tooling approaches, demanding extra time of the development team in terms of data input or management. We call these approaches *up-front* RE. These approaches promise better estimation and lower cost changes, in return for an investment in time and effort for modeling the requirements. This imposes a substantial burden on the development team, one which is often seen as too costly (even while acknowledging the cost in the long-term may be lower). This is why requirements management tools such as IBM's DOORS or BluePrint are most often used on projects where traceability from requirements to implementation is necessary for regulatory reasons (e.g., ISO/WD 26262). Many projects do not fall into this category [3].

We describe an investigation we conducted into the just-in-time RE practices at several large-scale software projects—Mozilla, Lucene, and CONNECT. Expanding on the work of Scacchi [1], [4], we conducted case studies for each project using an exemplar individual requirement to understand how requirements were managed in that project, and what problems, if any, were encountered. This paper makes the following contributions:

- Provides insight into the requirements practices at three large software projects;
- Details some problems and challenges those practices cause;
- Suggests a taxonomy of requirements practices to support comparison;
- Outlines the need for measuring the present and future value of a requirement;
- Highlights future research directions for better understanding just-in-time requirements.

## II. Empirical Investigation

### A. Methodology

As case studies we picked three projects which have open development environments and also have some form of requirements available. These projects are fairly long-lived, reasonably stable and very successful (they deliver high-quality software to millions of users). To unify the discussion, we note the difference between requirements and tasks. *Requirements* refer to higher-order organizational constructs, including features to be added to the project, agile epics and user stories, improvements to software quality, and

[1]E.g., http://www.agilemodeling.com/essays/examiningBRUF.htm

978-1-4673-4365-7/12/$31.00 © 2012 IEEE

major initiatives such as paying down technical debt. *Tasks*, by comparison, are individual work items which compose a requirement, are commonly assigned to individuals, and in many cases are formally represented in issue trackers and code repositories. Ko and Chilana call them "concrete, actionable issues" [5]. Organizations have different interpretations of how this distinction is made. We give examples of this division below.

For each project, we began by studying the project's tasks, as represented in an issue tracker and source code repositories, to gain a sense for how development proceeded at the task level. This allowed us to select a requirement, composed of some of these tasks, which seemed representative of the project's work practices. We traced that requirement throughout the project artifacts (including mailing lists, textual documentation, and other sources). This approach gave us data to create a narrative surrounding that requirement and how it was initially created, divided into tasks, and eventually implemented. It should be noted that only one of these projects discussed 'requirements' as individual entities: the terms *features* and *user stories* were much more common.

It is important to contextualize the systems being studied, since software projects can be vastly different. For each case we do so with the model of Kruchten [6]. He proposes the following dimensions:

- Age of System – older systems have more constraints.
- Rate of Change – is the system in a dynamic ecosystem, or are customers tolerant of infrequent releases?
- Governance – how is the project run? Does it allow for outside contributions via open repositories? Is management hierarchical, distributed, fragmented, . . . ?
- Team Distribution – in particular, is the team co-located or distributed?
- Size – either in lines of code, function points, team size, etc.
- Criticality – are there any safety or security implications? What regulations govern the project?
- Business Model – is this software the company's primary source of revenue, or is it an internal project?
- Stable Architecture – are we adding on to an existing architecture, or must we redesign that as well?

Below, we reference many project-specific artifacts, such as the task id LUCENE-2215. We list the source for these artifacts (e.g., the web link) at http://goo.gl/u1SXZ.

### B. Mozilla

Mozilla is a not-for-profit organization devoted to "building a better Internet" whose chief product is the Firefox web browser. Firefox 1.0 was released in 2004. The system is highly dynamic, with releases every six weeks. Governance is corporate and not-for-profit. As of Firefox 2.0, a majority of developers are located in Silicon Valley, with the remainder widely distributed [7]. Mozilla employs approximately 500 people. The software is not safety-critical, but is used by millions of people around the world. The architecture is stable, but the problem space is highly dynamic with several competing products, including browsers from Google, Apple and Microsoft.

We chose a Firefox feature to investigate, the IonMonkey compiler improvement. IonMonkey is dedicated to completely re-writing the existing compiler in Firefox to drastically improve Javascript performance. The Javascript component of Firefox contains over 696 kLOC (thousands of lines of code), of which 360 kLOC are Javascript and 165 kLOC C++. The IonMonkey-specific code is 57 files with 27.5 kLOC of C++, although the project modifies files outside of these files.

*1) Requirements Tools and Practices:* Mozilla is an open organization that defines its missions and goals through a wiki, hosting the roadmaps for 'what we will be doing' in the coming year. These roadmaps then define a more detailed set of features which the project will focus on for the coming year. These are lists of high-level requirements and features that the team would like to work on. These roadmaps are developed by Mozilla internally, but released for comment on the mailing lists so anyone can get involved. Each roadmap has a product owner who oversees the roadmap and leads development focus. The focus features are assigned to a feature team, consisting of Mozilla employees with relevant expertise (e.g., for IonMonkey this consists of Javascript and language experts, testers, and performance engineers). Community members (i.e., not Mozilla employees) can contribute to the feature. In our study of the IonMonkey case, non-employees were largely bug-reporters and testers, as reported in ([8]).

We chose as our unit of analysis the high-level feature (**"Improve JS performance by building a new compiler"**). This is broken into meta-bugs represented in the Bugzilla task tracker (**"IonSpeed: make us fast"**). These tracking bugs are linked to individual tasks using Bugzilla's dependency tracking mechanism, and then, when completed, closed with a manual reference from a commit in the project's Mercurial repository.

*2) Observations:* There is no iteration in the sense of Scrum sprints. Instead, broad leeway is given to a feature team to determine how to implement the feature. Because the high-level product is fairly well-understood at this point, and the developers are also customers, it is reasonably clear what constitutes 'done' for a given feature (i.e., it is reasonably clear what 'improving JS performance' looks like, and there is a website dedicated to measuring performance). A recent shift to frequent product releases (every 6 weeks) sets clear deadlines for the team. Implicit is the recognition that the developers are world experts on the feature, and that finishing the feature will involve some amount of experimentation. This is similar to "lean startup" ideas of [9]. Experiments are tracked and measured against a baseline. For example, bug

578225 tracks 'small wins' and bug 578133 tracks tasks that 'make us fast'. We see this as just-in-time RE because the requirements are only fully understood once the experiments are concluded.

The feature-metabug-bug-commit process is occasionally violated, and it is difficult to determine the current status of a feature without asking individual developers. While there are tracking pages, and weekly status meetings, these are either infrequently updated (in the case of the former) or too detailed (the latter). References to tasks are found in many locations. For example, "a lot of other functions are called only a few times, it's faster to just run them in the interpreter. See bug 631951 for more information." This is a reference to a Bugzilla id from a newsgroup. That suggests that this bug and this newsgroup post are related.

### C. Lucene

Lucene is a project in the Apache Software Foundation ecosystem. It supports full-text search and indexing, and is widely used in mission critical scenarios, including searches on Wikipedia.org. Currently, the Lucene-core code base consists of 363 kLOC of Java. Projects in Apache are governed by the Apache project management policies. In Lucene, these take the form of a project management committee (PMC), composed of 23 people, and 37 committers. Access to either group is through merit, typically by contributing patches and bug reports. The PMC does not act as change control board, but rather as a facilitator for the legal requirements of delivering software—copyright rules, trademarks use and so on. Typically, though, PMC members are the more senior members of the project. Lucene uses lazy consensus for code review. A few positive votes is sufficient to commit the code, and negative votes should explain why. Developers are widely distributed around the world.

*1) Requirements Tools and Practices:* Since Lucene is a flat organization driven by merit, decisions on future directions are driven by consensus. Many requirements arise organically, particularly since the developers are all domain experts, constantly thinking of new ideas to improve Lucene. These are created at first as rough outlines of what might be useful. They are fully elaborated just-in-time, when someone begins to work on the issue. The common thread among the committers is a passion for the technology. Some requirements arise from experience with clients or user requests, while others are spontaneous discussions on mailing lists or on IRC (LUCENE-2308). A key place to outline future plans is at developer conferences such as Lucene Revolution. Held annually, one session is a panel discussion on future plans, where anyone can participate: "any and all Lucene/Solr committers attending the conference are invited (and encouraged) to be on the panel as well".

Features are released based on readiness. A feature freeze is announced a few weeks before the target date, and at that point any tasks not blocking release are re-targeted to the next major release: "[the release manager] will move all jira issues out of 3.6 unless they are marked blocker bugs." New features and enhancements are added to the issue tracker as they arise.

We focused on one issue in particular, LUCENE-2127, which is related to how Lucene handles large result sets. The initial impetus for adding this feature was a post on a mailing list. The reporter, a Lucene committer, attended a presentation where he learned of possible performance improvements. He reported this as a new feature on Jira, and began experimenting with the approach. Another committer saw the issue and informed the community of his alternative approach, LUCENE-2215. The community coalesced around this approach, which was committed and released.

*2) Observations:* Like Mozilla, there is no clear software process being followed, aside from code freezes prior to release. Requirements arise organically, and the long-term direction for the project is a combination of new ideas and features which are finally ready. Individual developers take on the responsibility for explaining new features to the community. Planning and discussion nearly all go through Jira, but IRC and face to face meetings are useful in crystallizing consensus about the direction. Jira provides good tools for managing the tasks involved in a release, but the connection between individual tasks and high-level requirements is less clear, and mostly captured in the knowledge of the individual committers. A Jira 'feature' task is still a relatively low-level, actionable unit of work.

### D. CONNECT

CONNECT is an initiative of the U.S. federal government to interconnect health-care information, beginning with governmental agencies, but ultimately other private organizations. It is licenced under the BSD licence, with the intent that other vendors adopt the code and make commercial software and support. CONNECT is developed using a modified agile approach. They hold sprints (83 to date) and make fairly frequent releases. There are 45 separate committers.

*1) Requirements Tools and Practices:* The project collects requirements from the key stakeholders, some of whom include the Department of Defence (DoD), Federal Housing Authority, and the Centre for Medicare and Medicaid Services (CMS). The project therefore has multiple important customers, and since it concerns health information, must abide by many governmental regulations. The development is contracted to private consultants and developed in several different offices. Periodic code sprints bring all developers together to sync. A change control board (CCB) made up of the cooperating agencies governs the process of prioritizing development.

We chose to examine the high-level requirement that CONNECT support logging functionality. This feature is

realized in (among others) bug tracker issue GATEWAY-472, which is in user story format and titled "As a CONNECT implementer, I shall have the ability to get more comprehensive performance logging and metric data (counts and duration) using improved logging in CONNECT". The source of this issue is a set of requirements from various stakeholders, including CMS and DoD, that demand CONNECT support logging for auditing and error detection. Indeed, the issue itself notes that this item 'consolidates' eight agency requirements. These agency requirements are prioritized in CCB meetings, where the agencies assign importance values from 1-5 to the user story, and a set of stories are selected within a certain constrained number of story points (an estimate for sprint effort). This particular issue is dormant, and superseded by other, newer issues, including GATEWAY-2222, improved logging.

*2) Observations:* All agencies have clearly developed a list of requirements beforehand, which are then amalgamated using the CCB. However, because the development is iterative, certain stories are left for future sprints, and the requirement is not fully described until it becomes part of an active sprint, as needed. Many issues are tracked in multiple spots, such as the JIRA issue tracker or Excel spreadsheets. They operate at multiple levels of abstraction, and are added by different users. There is unclear language in the issue tracker: "supports/supported by", "depends/depends on", and so on. JIRA uses a particular ontology of software development, anchored in the Epic/User Story/Task/Bug hierarchy, but this is not the same as the breakdown CONNECT uses, which includes separate requirements labels (e.g., REQ-097, EST009 are both names for the logging requirement). Thus the JIRA fields are a strange melange of JIRA labels and CONNECT language. For example, in the Description field, CONNECT developers had added:

> QA: No new tests required, Validation Suite should suffice.
>
> CC: Codereview, checkin, validation suite passes.

It should be noted that JIRA supports full customization of the field labels and organization, which has been done to add the "Epic" issue type, but nothing more.

## III. Results and Discussion

Based on the investigation, we now return to our original questions regarding the nature of requirements processes in these organizations, and the problems encountered. To do so, we generalize our observations for all three projects into requirements practices which were positive, and aspects which were negative. We make these generalizations based on practices which we observed were common to all projects.

### A. Common Practices

These practices tend to emphasize the just-in-time nature of the requirements process at each organization. The underlying objective was to perform just enough requirements engineering to deliver valuable products without burdening the development team. The short nature of our observations do not allow us to comment on any long-term penalty such practices might incur, such as inadequately preparing future changes or maintenance might be required. However, these projects are long-lived and successful, so it seems unlikely to be an issue.

**Just-in-time requirements**. Each project proceeded from a list of objectives for the next year to a set of features the next release ought to contain, rather than an up-front RE approach with detailed plans and estimates for what should be done. Requirements were instead first sketched out with simple natural language statements, and then fully elaborated during development, what we call just-in-time RE. This elaboration was done either ahead of each sprint by a CCB (CONNECT) or as developers worked on high level features and enhancements (Mozilla and Lucene). In Mozilla, for example, experimentation was used to drive feature implementation, rather than an up-front functional specification.

**Feature-driven**. All three projects were primarily focused on delivering new features with each release. For example, change logs for Lucene emphasize the new features which form part of that release. This makes it very clear what is important for developers to work on. These features also easily support fluid team structures. On the other hand, there is less opportunity to understand how these features fit into a greater whole (see 'big picture' below). It is only when the features are being developed that they are fully understood and described.

**As-needed traceability**. Traceability was implemented where it was necessary. Rather than using a traceability matrix linking requirements to source explicitly, traceability links were defined using the tools available. In Bugzilla, this was manifested as dependency links between meta-bugs and specific tasks; in Jira, as dependency links between tasks. All projects followed the practice of prefacing a commit message to the source repository with a task ID number. The advantage to this approach is that 1) developers actually do it, and 2) it seems to provide sufficient information to understand how something was developed. The disadvantage is that it is heavily tool-dependent for the automatic link generation.

**Exploratory and iterative development**. Mozilla and Lucene use an exploratory, release-based approach to development, which places the development teams in full control of the software feature release. This allows teams the relative luxury of deciding when a feature is ready, to work on tasks they feel are important (as needed), and to define what 'done' means. In CONNECT, the sprint approach ensures that development is relatively agile and adapts to changes in stakeholder requirements, which frequently arise due to the poorly understood nature of the domain.

**Community-mindedness**. All three projects are open-source and welcoming to new developers (in the form of bug reports, questions, code). There is frequent and meaningful collaboration with other participants, which helps with getting immediate feedback on new ideas and approaches, which might eventually become requirements or features. This seems to partially satisfy the role of Onsite Customer in agile software methodologies. In several places new ideas came from translating academic papers into code, and then getting feedback from the original authors.

### B. Departures From Accepted Best Practice in RE

We now detail some practices we found in our case studies which seem to diverge from best practices in RE research. These suggest longer-term problems which may arise.

**No explicit 'Big Picture' thinking.** Many RE methodologies, such as goal-oriented analysis in KAOS [10], suggest beginning with organizational objectives and decomposing those into sub-goals in order to wholly capture the purpose of the software. Mozilla and CONNECT both had long-term objectives, but these were loosely modelled and subject to change. Lucene had no clear long-term goals beyond a sense that non-functional requirements were important (handling bigger data, faster results searching). Development tended to proceed in bursts, with the best and most popular ideas getting the focus. It wasn't clear, for example, who was tasked with refactoring code to remove technical debt.

**No separate prioritization phase**. RE is characterized as proceeding from elicitation to analysis, with prioritization an important part of analysis [11]. However, prioritization at Lucene and Mozilla relied on the interests of the developers; at CONNECT it was done by a CCB. The developers on Mozilla's Javascript team also acted as domain experts, while at CONNECT the domain is poorly understood and the problem is quite novel. The disadvantage for Mozilla is that you only find requirements that the developers surface themselves; at CONNECT the problem is that prioritization is done by each stakeholder individually, and then complex negotiations ensue.

**Unclear feature provenance**. Traceability is a major objective of RE practices [12]. However, while makeshift traceability existed in our case studies, it was usually unclear where a particular feature originated, or how to discover this. It was often very clear where *tasks* originated, e.g., via bug reports, but features were less obvious (recall that a task is actionable). For example, the IonMonkey project at Mozilla has evolved from other Javascript projects, from competition with other browsers, and from developer interest. Amalgamating the various efforts that somehow touch on IonMonkey is not possible without extensive developer knowledge. The challenge is in identifying, at a project level, what success criteria are when it is not clear what the feature is delivering.

**No repeatable elicitation**. Explicit models documenting who originated a particular requirement are advocated in [13]. In these case studies, however, there is no good way to understand how much elicitation was conducted or what value a feature exhibits. This implies that some features serve many users, and are highly valuable, while others might serve very few. In Lucene, the decision to implement a feature is driven by perception that this is a cool feature that *should* help the project, but there is no analysis of the overall importance (which ties into the lack of big picture thinking). In CONNECT, most functional requirements come from the stakeholders, such as the Veterans Administration, but there is no good way of evaluating that requirement's importance outside the periodic prioritization meeting. Instead, there is an assumption that the single stakeholder representative is valuing that feature using the same criteria as the other stakeholders.

### C. Threats to Validity

We selected these cases in an ad-hoc fashion, since they seemed representative based on our understanding of the project, and served to illustrate the phenomena of just-in-time RE which we, and others, have seen in many projects. However, a more rigorous approach is necessary in the future to provide more confidence in the observations. We do not propose that our observations and suggested directions hold for all other projects. It is unclear how these cases generalize, but the fact that three major projects use just-in time RE suggest there is something worth understanding. However, to increase our theoretical generalizability more case studies would be helpful.

An internal validity threat is that investigating these issues without the assistance of the participants involved may lead to a major oversight when something pertinent was not captured in the various archives we looked at. For example, Aranda and Venolia [14] classified the *levels of data collection and analysis* in repository mining using a four level scale. Level One collection is typically automated retrieval and analysis, and Level Four includes stakeholder interviews. We characterize this study as Level Three, since we use *human sense-making* to understand the nature of the requirements under study.

## IV. Future Directions

These projects are all successful, and yet they are not following many conventional RE practices (e.g., those described in [15]). We now return to our earlier questions about practices and problems.

### A. Understanding Requirements Practices

We are interested in the types of practices and tools which these projects use. The larger question is about understanding the appropriate degree of requirements representation

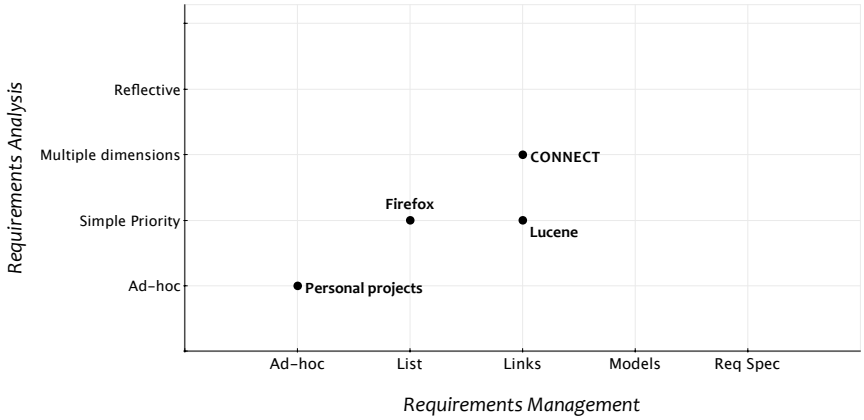


Figure 1. Dimensions of requirements practice, including a possible categorization of our case studies.

and analysis support that software projects require. Requirements are not treated identically across projects. Instead, there are at least two dimensions across which we classify requirements practices, which we represent in Fig. 1. One is **requirements management**, or level of formalization. Arguably a fully managed set of requirements supports traceability and conformance checking (e.g., did we deliver what the customer wanted?). While we might be tempted to conclude that more heavyweight tools are more suitable for more formal management, it is certainly possible to customize wikis, hypertext, and spreadsheets with complex management functionality. Our progression through the dimension of management begins with ad-hoc or unmanaged requirements (if they exist at all); then simple lists of requirements, as in a contractual specification. Mozilla's wiki pages and bug tracker manage requirements this way. Requirements in lists are typically in natural language. Following from that is the representation of requirements using untyped links with other software tools, which we see with Jira and the Lucene source repository. At the next level, requirements are managed using hierarchical models, including feature models. There is a notion of decomposition of requirements and dependencies on other requirements, i.e., requirements models become complex independent entities in their own right. There is nearly complete traceability from the original elicitation documents all the way to the source code, which we typically see in highly-regulated environments. Finally, we have requirements as specification: requirements are living documentation that are used as acceptance or end-to-end test components.

The other dimension of requirements practice is **requirements analysis**, or level of strategic planning. This describes a project's use of requirements (however represented) in the software design and planning phases. To begin with we have ad hoc planning (or lack of planning), where plans are concocted based on past experience or as needed. Requirements are often unused or not collected. The second level uses simple prioritization. Plans and analysis are ad-hoc and as needed. Documentation about plans exists, but is typically out of date. Prioritization is based on binary comparisons over a single variable. Both Lucene and Mozilla made decisions this way. In the third level, planning is more extensive and requirements are costed or valued on multiple dimensions (time, value, etc.) in order to inform the plans. To some extent CONNECT considered extra dimensions (at the stakeholder level), which is why it appears here. Advanced practitioners might use techniques such as formal logic, as in [16]. Finally, we have reflective analysis. This encompasses the previous level, plus the refinement of future analysis based on past experiences.

To return to the question of "appropriate degree of RE" we introduced above, the purpose of this taxonomy is to characterize two dimensions of an organization's RE practices. In some cases, it may make sense to begin by insisting that an organization managing requirements at Level 1 move up to higher levels. However, this is not always the case: in some instances it may be inappropriate, since the project does not demand that level of requirements management. For our case studies, we can conclude that they are doing 'appropriate RE' if our measure is project success. It is difficult to establish counterfactuals, of course, but it does seem that a slight increase in formality or usefulness of requirements should help all three organizations. For example, we might be able to centralize Mozilla's scattered set of requirements and tasks. It is an open question whether more formality or centralization would be beneficial.

We therefore propose two research questions. One, given a project in the lower left-hand quadrant of Fig. 1, what advantages (in terms of productivity, cycle time, and others) could be gained from improving along either dimension, i.e., moving from just-in-time requirements to up-front requirements. Secondly, what tools, processes, and technologies might ease that transition? A major risk of introducing new tools or processes is that they are used once and then

abandoned, as seems to be the case for Mozilla's wiki-based approach to feature prioritization. We also note that other RE practices, such as validation and negotiation, were not examined in this paper, but would be interesting to examine explicitly.

### B. The Role of Tools in Just-In-Time RE

We observed another practice, a heavy reliance on tools, such as issue trackers (JIRA, Bugzilla), mailing lists, and source code repositories (Git, Subversion). For all three projects these tools are critical to their just-in-time RE practices. In Lucene, for example, all conversation about development tasks is captured in the Jira repository. However, conversations about features do not always get captured in a tool, and this impacts the participation of outside users. Furthermore, since the organization depends on these tools, any opportunities for improvement are limited to what the tools provide, and what parts of the tool people actually use. It seems clear that Bugzilla's lack of external integration with commit messages, for example, is a major hindrance to tracking outcomes, short of external tools to link issues to code commits. As with the previous section, research to understand how and why these tools are adopted is important.

### C. Understanding Net Present Value

One of the questions that is very difficult to answer is the opportunity cost of a given decision. In other words, by working on feature A, what opportunities are we missing now, or in the future, by not working on feature B? Sullivan et al. [17] provide a possible framework (net present value and real options) for modeling these decisions. Organizations like the ones we studied here do not seem to pay much attention to these questions. Instead, they are opportunistically working on features that seem at that moment to be the most important—hence just-in-time. The research question is to understand where the optimal point is between over-analysis and missing opportunities. This is tricky: measuring what was not done is very difficult, as is valuing decisions in uncertain domains. Perhaps the past performance of the organization in making these decisions can be a guide here. A related question concerns the enabling factors of just-in-time RE: those project characteristics which make this possible. These might include well-managed teams, good access to customers or other decision makers, flexible schedules, and others.

In future studies we would like to explore the use of several research techniques (including interviews, observations, and data mining) to better triangulate actual practices.

## V. Related Work

Scacchi [18], [4] has written about the use of requirements in open-source software (OSS). He argues that requirements as they are traditionally viewed simply do not exist in that format, since there are no external customers in most OSS projects. While perhaps true earlier, many of the projects that are technically 'open' are not necessarily free from "the constraints of budget, schedule, and project management [18, p. 1]", as he says. Mozilla, for example, is a commercial not-for-profit with clear budgetary and schedule constraints. In that sense, therefore, his characterizations of OSS might better be attributed to the domain most of the projects he studied operated under, rather than their openness.

Gallardo-Lopez and Sim [19] conducted a case study of a single company in the financial services sector. They looked at requirements validation, and noted that these agile requirements were often composed of a written and tacit component, which meant testing was challenging. The notion of agile requirements is analogous in many ways to just-in-time requirements, although we describe the characteristics of this form of RE in more detail.

The notion of just-in-time RE is supported in the industry literature (i.e., [2], consulting firms like Forrester, tech bloggers like David Anderson or Martin Fowler, and content aggregators like DZone or HackerNews). Although anecdotal, this literature provides insight into what practitioners are doing.

The Business Analyst Body of Knowledge (BABOK) [20] is a guide for business analysts, most of whom spend a large amount of time doing 'requirements'. The approach taken in the main BABOK text is strict up-front requirements, including estimation and planning. A proposed addendum, the Agile extension to the BABOK [21], adds more iterative refinement of requirements to the BABOK, which supports our concept of just-in-time requirements.

As a specific example, consider TargetProcess[2], a company building enterprise project management software. They used user stories and estimation using story points and subsequently shifted to just-in-time requirements, manifested as conversations with the product owner and implementation team when necessary.

Rather than upfront planning, there are very high-level Epics.Behavioural tests are written before the user story is begun, also called *behavior-driven development* [22]. This organization supports our understanding of just-in-time RE being composed of a management component (here they use user stories and whiteboards) and an analysis component (here relatively ad hoc).

Lormans et al. [23] presented results from a small-sample survey on RE practices. They observed that provenance and coverage of requirements was useful to practitioners, supporting our taxonomy dimensions of management and analysis, respectively. They then worked on a tool to support automatic traceability recovery, but our work suggests that such a tool is doomed to failure without considering the context of the software project.

[2]http://www.targetprocess.com/articles/agile50months

In many projects, developers face ongoing, continuous requirements and feature changes, where just-in-time analysis in necessary. Research in product management, the next release problem, and software product lines and roadmaps has looked at the problem of identifying which set of features are appropriate for which release, and how best to cope with the uncertainty involved, e.g. [24].

## VI. Conclusion

In this paper we discussed the concept of just-in-time requirements analysis. We showed how this notion is manifested in three separate software development projects. From the detailed investigation of specific requirements, we generalized to practices which were shared, and some which contradicted academic best practices. We then proposed some research directions, including a taxonomy for how requirements are managed and analysed. Our longer-term research objective is to deliver the clear value of well-defined requirements management processes, such as optimization, while respecting (and working within) existing work practices.

## References


[1] W. Scacchi, "Understanding the requirements for developing open source software systems," *IET Software*, vol. 149, no. 1, pp. 24–39, 2002.

[2] M. Lee, "Just-in-Time Requirements Analysis The Engine that Drives the Planning Game," Agile Alliance, Tech. Rep., 2002.

[3] J. Aranda, S. M. Easterbrook, and G. Wilson, "Requirements in the wild: How small companies do it," in *RE*, Delhi, India, Sep. 2007.

[4] W. Scacchi, "Understanding Requirements for Open Source Software," in *Design Requirements Engineering: A Ten-Year Perspective*, ser. Lecture Notes in Business Information Processing, Springer Berlin Heidelberg, 2009, vol. 14, pp. 467–494.

[5] A. J. Ko and P. K. Chilana, "Design , Discussion , and Dissent in Open Bug Reports," in *iConference*, Seattle, Feb. 2011, pp. 1–8.

[6] P. Kruchten, "The Frog and the Octopus – Experience Teaching Software Project Management," in *Canadian Engineering Education Association*. St. John's, NF: IEEE, Jun. 2011.

[7] C. Bird and N. Nagappan, "Who ? Where ? What ? Examining Distributed Development in Two Large Open Source Projects," in *MSR*, Zurich, Jun. 2012.

[8] M. Zhou and A. Mockus, "What Make Long Term Contributors: Willingness and Opportunity in OSS Community," in *ICSE*, Zurich, Jun. 2012, pp. 518–528.

[9] E. Ries, *The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*. Crown Business, 2011.

[10] A. Dardenne, A. van Lamsweerde, and S. Fickas, "Goal-directed requirements acquisition," *Science of Computer Programming*, vol. 20, no. 1-2, pp. 3–50, Apr. 1993.

[11] A. Herrmann and M. Daneva, "Requirements Prioritization Based on Benefit and Cost Prediction : An Agenda for Future Research," in *RE*, Barcelona, Sep. 2008, pp. 125–134.

[12] O. C. Z. Gotel and C. W. Finkelstein, "An analysis of the requirements traceability problem," in *RE*, Colorado Springs, Colorado, 1994, pp. 94–101.

[13] S. M. Easterbrook and B. Nuseibeh, "Using ViewPoints for Inconsistency Management," *IEE Software Engineering Journal*, vol. 11, no. 1, pp. 1–25, 1996.

[14] J. Aranda and G. Venolia, "The secret life of bugs: Going past the errors and omissions in software repositories," in *International Conference on Software Engineering*. Vancouver: IEEE, Sep. 2009, pp. 298–308.

[15] K. Pohl, *Requirements Engineering: Fundamentals, Principles, and Techniques*. Springer, 2010.

[16] N. A. Ernst, A. Borgida, J. Mylopoulos, and I. J. Jureta, "Agile Requirements Evolution via Paraconsistent Reasoning," in *International Conference on Advanced Information Systems Engineering*, Gdansk, Jun. 2012.

[17] K. Sullivan, P. Chalasani, and S. Jha, "Software Design as an Investment Activity: A Real Options Perspective," in *Real Options and Business Srategy: Applications to Decision Making*, L. Trigeorgis, Ed. Risk Books, 1999, pp. 215–262.

[18] W. Scacchi, C. Jensen, J. Noll, and M. Elliott, "Multi-Modal Modeling of Open Source Software Requirements Processes," in *International Conference on Open Source Software*, vol. 1, Genoa, Italy, Jul. 2005, pp. 1–8.

[19] R. E. Gallardo-Valencia and S. E. Sim, "Continuous and Collaborative Validation: A Field Study of Requirements Knowledge in Agile," in *International Workshop on Managing Requirements Knowledge*. IEEE, Sep. 2009, pp. 65–74.

[20] IIBA, *A Guide to the Business Analysis Body of Knowledge*. International Institute of Business Analysis, 2011.

[21] International Institute of Business Analysis, "Agile Extension to the BABOK Guide," International Institute of Business Analysis, Tech. Rep. November, 2011.

[22] R. C. Martin and G. Melnik, "Tests and Requirements, Requirements and Tests: A Möbius Strip," *IEEE Software*, vol. 25, no. 1, pp. 54–59, Jan. 2008.

[23] M. Lormans, A. Deursen, and H.-G. Gross, "An industrial case study in reconstructing requirements views," *Empirical Software Engineering*, vol. 13, no. 6, pp. 727–760, Sep. 2008.

[24] C. Li, J. Van Den Akker, S. Brinkkemper, and G. Diepen, "Integrated requirement selection and scheduling for the release planning of a software product," in *International Working Conference on Requirements Engineering: Foundation for Software Quality*, Essen, Jun. 2007, pp. 93–108.