21st International Requirements Engineering Conference (RE'13)

Ongoing Software Development without Classical Requirements

Thomas A. Alspaugh and Walt Scacchi
Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3455 USA
thomas.alspaugh@acm.org, wscacchi@ics.uci.edu

***Abstract*—Many prominent open source software (OSS) development projects produce systems without overt requirements artifacts or processes, contrary to expectations resulting from classical software development experience and research, and a growing number of critical software systems are evolved and sustained in this way yet provide quality and rich functional capabilities to users and integrators that accept them without question. We examine data from several OSS projects to investigate this conundrum, and discuss the results of research into OSS outcomes that sheds light on the consequences of this approach to software requirements in terms of risk of development failure and quality of the resulting system.**

***Index Terms*—open source software; open source requirements; provisionments.**

I. Introduction

In 2002 one of us (Scacchi) published a study of requirements practices and artifacts in four open source software (OSS) development communities Scacchi (2002). This was the first systematic study to show that OSS system and development processes do not rely on what may be termed classical requirements artifacts and processes (Section III), namely those involving problem-space requirements in a document or repository evaluated for completeness and internal and external consistency. Others have since reported similar results German (2003); Noll (2008); Noll and Liu (2010). Yet there are successful, ongoing OSS projects with users numbered in the millions, and hundreds of OSS systems relied on as critical infrastructure, such as GNU/Linux, the Apache HTTP server, the Mozilla Firefox Web browser, the PostgreSQL database system, and the Eclipse development platform to name a few Des Rivières and Wiegand (2004); Mockus et al. (2002); PostgreSQL (2013); Stallman (2007).

From the point of view of a classically trained software developer and requirements practitioner and researcher such as the other of us (Alspaugh), this is unexpected. The broad consensus among software experts and researchers over recent decades has been that devoting appropriate attention to requirements processes and artifacts is essential to project success Brooks (1975); Gause and Weinberg (1989); Jackson (1995); Lamsweerde (2009); Sommerville (2004); van Vliet (2000), and that failure to do so risks undesirable outcomes such as:

- a product that fails to meet stakeholder needs,
- a product that does not exhibit necessary levels of reliability, evolvability, or other software qualities,
- schedule slips and budget overruns, or
- in extreme cases failure to produce any product at all.

How can it be that OSS development produces good software?

In the remainder of the paper we explore this conundrum. We present a motivating example, Brooks's thoughts on the success of Linux (Section II), and elaborate what we mean by "classical requirements artifacts and processes" (Section III), hereafter abbreviated as Classical Requirements, before describing our study (Section IV) and examining the OSS artifacts and processes that appear to serve in the place of Classical Requirements (Section V), using data from our previous work and work reported by others. We find that the overwhelming majority of requirements-like artifacts identified by ourselves and others may be characterized as what we term *provisionments* (Section VI), which state features or qualities in terms of the attributes provided by an existing software version, a competing product, or a prototype produced by a developer advocating the change it embodies. The processes involving these artifacts resemble or in some cases are indistinguishable from the bug reporting, tracking, and response processes found in closed source software (CSS) development. We discuss several contexts in which provisionments appear common and are arguably appropriate: OSS of course, software game mods, and open architecture software ecosystems (Section VII).

Finally, we place our work in the context of related work (Section VIII), discuss several questions of interest (Section IX), and conclude the paper (Section X).

II. A Motivating Example: Brooks on Linux

In reflecting on Raymond's description Raymond (2001) of the open source process producing Linux, Brooks observes of this "marvelously functional and robust operating system" that "for Linux a functional specification already existed: Unix" (Brooks, 2010, page 56). This is a curious statement, since the development of Unix itself displayed characteristics of OSS development including:

- software developed for the developers' own use rather than for an external client and users,
- a strong emphasis on extensibility, and
- no overt requirements artifacts or process preceding development.

Saying that Unix (specifically the Unix kernel) provided the requirements for Linux does not explain the problem; it merely moves it from Linux to Unix. The Unix kernel is marvelously functional and robust, too; was it developed using a functional specification or other Classical Requirements?

If so, supporting evidence is in short supply. Ken Thompson wrote the initial version of Unix in four weeks in the summer of 1969, yet the first edition of the Unix manual was dated 3 November 1971 Salus (1994). Salus notes "the only way you could learn [the Unix system] was to sit down with one of the authors and ask questions" Salus (1994). Ritchie recalls that in 1969 "Thompson, R. H. Canaday, and Ritchie developed, on blackboards and scribbled notes, the basic design of a file system that was later to become the heart of Unix"; not the requirements, but the design Ritchie (1984). We have searched the writings of the creators of Unix and researchers reporting on it for Classical Requirements without finding evidence of it.

It appears that it is indeed possible to produce a marvelously functional and robust operating system without the aid of a functional specification or other Classical Requirements.

Brooks goes on to note, as we and others have, that OSS development works because the developers are users, saying "The whole requirements determination is implicit, hence finessed." He finds no contradiction in ongoing development without Classical Requirements once initial development is successfully complete.

III. Classical Artifacts and Processes

Researchers and practitioners have developed many types of requirements artifacts and many requirements processes. We do not consider any of them in detail here. Instead we focus on three characteristics shared by nearly every such approach with which we are familiar:

1) a requirements document or central requirements repository, defining the system requirements and providing a criterion for whether a particular candidate requirement is or is not a requirement for the system;
2) requirements that are preferentially described in terms of the problem space rather than the solution space; and
3) requirements processes for examining the requirements document/repository for completeness, internal consistency, and external consistency with the domain and stakeholder needs.

These characteristics define what we term in this paper Classical Requirements.

We focus on these characteristics because they figure prominently in many influential requirements approaches and in the requirements practices of working CSS developers we have known or interviewed, and because convincing arguments have been made from them to project success and product quality Boehm (1976); Brooks (1975); Gause and Weinberg (1989); Jackson (1995); Lamsweerde (2009); Sommerville (2004); van Vliet (2000). Brooks famously says Brooks (1987)

> The hardest single part of building a software system is deciding precisely what to build. . . . No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later.

Boehm asserts, supported by data Boehm (1976),

> Clearly, it pays to invest effort in finding requirements errors early and correcting them in, say, 1 man-hour rather than waiting to find the error during operations and having to spend 100 man-hours correcting it.

Lamsweerde characterizes requirements errors as "numerous and persistent" and as the most expensive and dangerous of software errors Lamsweerde (2009). Gause and Weinberg note "Obviously, requirements are important because if you don't know what you want, or don't communicate what you want, you reduce your chances of getting what you want" Gause and Weinberg (1989).

The particular form of the requirements is not material to our work. We note that the prominence and importance of particular requirements artifacts and processes often vary depending on the type of system. Not all are appropriate for development of every system, but many situations can benefit from an appropriately chosen selection of them. Some (overlapping) types and corresponding artifact or process choices might be:

- Embedded systems, in which software is a component of a larger hardware system — a state-based specification;
- Real-time systems that must meet specific often-inflexible timing constraints — a temporal logic specification;
- Critical or high-assurance systems, for which what is required and what is acceptable must be determined with precision and the cost of failure is high — a model-checkable specification and validation by stakeholders;
- Systems that interact significantly with other automated systems — a formalized specification checked for consistency and completeness;
- Systems that play a role in specific organizational processes — stakeholder analysis;
- Systems that address novel problems or address problems in a novel way — processes that encourage exploration of the problem space.

We note that the use of Classical Requirements in these situations and others may be connected to the typical CSS context in which

- the system is produced by a development group for a client outside that group,
- most or all of the system's expected users are also outside that group,
- the developers may or may not have expertise in the problem domain, and
- the system is developed against a budget and a schedule.

The requirements state the expectations and commitments of the client on the one hand and the development group on the other. The client balances the benefits of the specific proposed system against the cost of developing it and the wait until it is ready. The development group evaluates whether the budget, resources, and schedule are appropriate for the work involved.

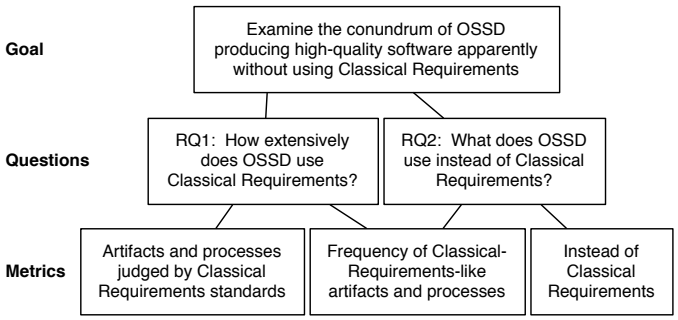

Fig. 1. Goal Question Metric model

The two sides explore, negotiate, and (ideally) agree on a set of requirements. Both sides can then make plans based on specific criteria for acceptance.

IV. Method

A. Research Questions and Metrics

Our goal is to address the apparent conundrum of OSS development (OSSD) that does not use Classical Requirements yet successfully produces high-quality software. We apply the Goal Question Metric approach Basili et al. (1994) to produce a measurement model operationalizing our goal into research questions, and associating each question with data that can be evaluated (Figure 1).

- (RQ1) To what extent do OSS projects in fact use Classical Requirements?
- (RQ2) Where OSS projects do not use Classical Requirements, what artifacts and processes are used instead, if any?

B. Sources of Data

We address RQ1 and RQ2 using data and results from our previous work Scacchi (2002, 2009) and from other published research on requirements in OSSD. For an introductory study we find this appropriate, in place of collection of a new set of data. A first step is to identify such research; there is not much. We used work by Noll and Liu Noll (2007, 2008); Noll and Liu (2010) which provides both analysis and some raw data, and work by German German (2003) providing analysis only. We also examined the data we found while investigating Brooks's statement that Unix provided Linux's function specification, using it primarily to cross-check where possible conclusions we drew from the other data sets. In some cases we followed up on specific data items and examined them in the original context. In a few cases we extended the data with newly-collected data, as for example that shown in Figure 2.

C. Validity

In this subsection we discuss the internal and external validity of the study, and threats to its validity.

1) Internal validity: Internal validity is the soundness of the relationships within a study. Our study examined data and analysis from different researchers, then merged them in order to apply our metrics. We examined original data where possible in order to apply metrics more uniformly. We looked first for overt Classical Requirements, then for requirements-like artifacts and processes, and finally for artifacts and processes that appeared to be used in place of requirements. In order to systematize our study, we coded and categorized each such instance, following standard qualitative practice Creswell (2003).

2) External validity: External validity is the degree to which the results from the study can be generalized. Identification of successful OSS systems without overt Classical Requirements provides an existence proof that software can be successfully developed without it. Other results are more difficult to generalize reliably; for example, the study cannot provide strong support for a hypothesis that Classical Requirements does not contribute to reducing the risk of project failure, nor to increasing the probability that stakeholders will be satisfied. The study also does not provide strong support for hypotheses on the incorporation of OSS development approaches into CSS projects, as our study examines only OSSD data and analyses; these are intriguing and investigation of them remains as future work.

3) Threats to validity: We examined every study we found that addressed OSSD requirements, eliminating any possibility of selection bias; however, the number of such studies is quite small (five), making it more difficult to generalize our results and increasing the possibility that other OSSD projects do not fit our conclusions.

Other practitioners and researchers might apply different standards, for example with a broader or stricter definition of which instances qualify as Classical Requirements. We minimized this by defining Classical Requirements explicitly (Section III) and in abstract terms. This threat affects only RQ1.

V. OSS Artifacts and Processes

A. Requirements-Like Artifacts and Processes

We present several examples of specific requirements-like artifacts and processes we identified in our study. Perhaps the most common requirement-like OSS artifacts are isolated feature requests or bug reports submitted to tracking systems like Bugzilla (Figure 2), and discussed there or on email lists or electronic bulletin boards. An example is this proposal for OpenEMR Noll and Liu (2010):

> You could add a link to the existing superbill page which would open a new browser window/tab with a printable version that meets your criteria. This way, you could leverage existing code and probably not have to add a table. I am thinking of something similar to printable links elsewhere in the program, like in reports and patient report.

A second example is shown in Figure 2. Here a Firefox feature request is being discussed, in conjunction with possible

Fig. 2. Discussion of a feature request in the Firefox Bugzilla, "Bug 797876 — Introduce new API for JS content Profiling"

changes to the implementation and architecture. Comment 4 may be taken as stating a requirement that Firefox provide the Profiler, specifically, and more generally that Firefox provide a specific kind of results (those that the Profiler currently provided, we infer). This fairly explicit requirement is stated in solution-space terms (what Profiler provides) rather than the corresponding problem-space terms; of course, this is probably considerably more compact. The discussion is focused on architecture and implementation issues involved in the requirement. Other requirements are considered only indirectly if at all, for example if the goal of replacing JSD1 with JSD2 + RDP is taken to imply a here-unstated software quality requirement.

A third example is tabbed browsing, a Web browser feature little known not so many years ago, but now so nearly universal that the name "tabbed browsing" has become a token representing a complex of properties and user stories now assumed to be obvious and requiring no explanation. Mozilla/Firefox tabbed browsing appears to have first been proposed in a one-sentence scenario of use ("One thing that I would really want to see is the ability to open a link in the new window in background . . . ") posted to a Mozilla newsgroup, which was immediately followed by a post beginning "Have you tried tabbed browsing [in the Opera web browser]?" Noll (2007). Both these are provisionments; the first cites current system behavior and describes a difference from it, while the second cites another system that exhibits the behavior referred to.

Each feature request or bug report can be taken to imply a requirement, but in themselves they rarely constitute a Classical Requirements artifact. In the examples listed above, as for most requirements-like artifacts we identified, the artifacts are neither integrated into a central requirements document/repository, described in terms of the problem, nor being examined in the context of other requirements. Our study indicates that the OSS projects in question do not use Classical Requirements.

B. An OSS Requirements Document

Our data sources included one example identified as a requirements document: "Firefox2/Requirements"

MozillaFoundation2006-fr (2006), discussed by Noll Noll (2008). The document is interesting to us in two ways.

First, our examination of the document found the items are expressed in general rather than specific terms, as in this representative example "[The system] will be optimized and tuned for general web browsing use cases", with the specifics no doubt proposed, discussed, and agreed on through project mailing lists and discussion boards as in the examples in the previous section. We also note all but one are stated as a difference from the previous Firefox version, using phrases such as "will update" and "will improve", in other words as provisionments.

Second, and perhaps more significant, this is the only presumptive requirements document or repository our research identified in our own searches and in related work on requirements in OSS. While its existence indicates that OSS development can tend toward Classical Requirements, its apparent uniqueness highlights our general finding that OSS development does not make use of Classical Requirements.

VI. Provisionments

As stated in Section I, a *provisionment* is a statement of features or qualities in terms of the attributes provided by an existing software version, a competing product, or a prototype produced by a developer advocating the change it embodies. Most provisionments we encountered only suggest or hint at the behavior or quality in question; the expectation seems to be that the audience for the provisionment is either already familiar with what is intended, or will play with the cited system and see the behavior or quality in question firsthand.

In our study, we saw provisionments being used for requirements or requirements-like artifacts in two ways: either directly as a specification of behavior or quality, or as a starting point in a specification of behavior or quality differing in stated ways from that expressed by the provisionment.

Section V provides examples of both types. Firefox Bugzilla comment 4 in Figure 2 "I think that existing Firebug users would complain if the Profiler is removed or providing [sic] different kind of results" uses a provisionment directly (though stated in the negative), while the OpenEMR proposal uses a provisionment ("the existing superbill page") indirectly as a starting point for a difference ("You could add ...").

A provisionment is distinct from a feature, a quality, a bug report, and similar entities in that each of those is *something to be expressed*, while a provisionment is *a way of expressing something*. In our study we found many feature requests and bug reports expressed using provisionments; OSS project archives appear to teem with feature requests and bug reports, and the majority we examined were expressed using provisionments. Statements of qualities were much less common but were also often expressed with provisionments.

VII. Some Example Contexts

We discuss three contexts highlighting the interplay between requirements, provisionments, and architecture: open source software, here discussed at greater length; software games, some of which are themselves OSS and many of which support modifications that exhibit OSS characteristics, whether the underlying game is OSS or not, and are described using provisionments; and OA systems of complex components, for which provisionments mediated by architectural configurations play prominent roles.

A. Open Source Software

OSS requirements, to the extent that they can be identified, tend to be distributed across space, time, people, and the artifacts that interlink them. OSS requirements are thus decentralized—that is, they are decentralized requirements that co-exist and co-evolve within different artifacts, online conversations, and repositories, as well as within the continually emerging interactions and collective actions of OSSD project participants and surrounding project social world. To be clear, decentralized requirements are not the same as the (centralized) requirements for decentralized systems or system development efforts. Traditional software engineering and system development projects assume that their requirements can be elicited, captured, analyzed, and managed as centrally controlled resources (or documentation artifacts) within a centralized administrative authority that adheres to contractual requirements and employs a centralized requirements artifact repository—that is, centralized requirements. In this way as in others, OSSD projects represent an alternative paradigm to that long advocated by software engineering and software requirements engineering community Scacchi (2009).

By the standards of classical software development and requirements practice, OSS requirements and processes are not satisfactory. Requirements are expressed indirectly at best; they are scattered across mailing lists, discussion boards, and bug trackers rather than collected in one place; they appear to be integrated only in the implementation of the system they refer to; they are almost universally stated in solution terms, not problem terms; once stated and discussed, they rarely appear to be referred to.

An RE researcher or practitioner might well look at dispersed statements such as these and simply conclude that requirements were for practical purposes absent by any reasonable or ordinary standard; if such decentralized, indirect requirements were used for a classical software development project, it would be judged to be at high risk of failure.

One would think therefore that many open source projects should fail—and they do, in large numbers. About 59% fail according to one study Wiggins and Crowston (2010), roughly double the 31% rate at which classical projects are reported to fail according to a 1994 survey The Standish Group (1994). Of course failure means something different for an OSSD project; there is no concept of over budget or behind schedule, and failed OSSD projects tend to wither away rather than being cancelled. Nevertheless, the comparison is startling.

Though most OSSD projects fail to produce a sustained sequence of widely-used software system releases, a substantial number are striking successes. Hundreds of OSSD projects are critical in a number of areas:

- the operation of the World-Wide Web: (the Firefox and Chrome web browsers and the Apache web servers and web services infrastructure);
- interactive software development (Eclipse and NetBeans development environments),
- customer relationship management (SugarCRM),
- database management systems (PostgreSQL, MySQL),
- operating systems (GNU/Linux, Darwin/OSX),
- office communications systems (Asterix),

and many more.

Clearly OSSD processes are capable of producing high quality software systems, despite scanty requirements artifacts and processes. We see the use of provisionments to make statements about the functionality of current and future system versions as one key factor, particularly convenient for an ongoing project producing version after version, each of which is described not in absolute terms but in terms of its differences from the previous one. Others may include developing an (informal) architecture and reasoning about it, in place of developing requirements and reasoning about requirements; using extensibility (see below), developer prototypes, and frequent releases of new system versions to explore the problem space by experimenting with alternative solutions within it; the fact that OSS developers are also users of the systems they develop; and the extensive discussions of system issues and proposals, characteristic of OSSD projects, in online forums that are public and persistently available.

We note that many prominent OSS systems are strongly extensible, with mechanisms by which the core functionality of the system may be extended independently, without affecting the system core. These mechanisms allow end-users to customize their copy of a system to suit there own needs and preferences, and in many cases allow developers to expeditiously prototype candidate provisionments. Examples of extensibility include Unix, supporting the addition of shell scripts, commands, libraries, and device drivers; Firefox, Eclipse, jEdit, and others, supporting the addition of plug-ins; and Firefox and jEdit again, and others, supporting the use of scripting languages. In addition to satisfying the system quality requirement (QR) of extensibility, extension mechanisms can also contribute to the requirement, for project success and continuation, to bring new contributors into the project community. Writing extensions for one's one copy of a system is an easy and appealing first step towards making more substantial contributions to the project that produces the system.

Extensibility and several other quality requirements will be seen to play important roles in games and OA systems too.

Viewing OSSD from a classical RE standpoint, we still note some concerns. Classical RE has approaches for identifying relevant stakeholders, and we see no corresponding practice in OSSD. We are concerned that OSSD projects will tend not to identify stakeholder roles in which the stakeholders are not developers and (for whatever reason) not motivated to come forward and contribute. We are also concerned about the effectiveness of OSSD in exploring the problem space, as opposed to the solution space. If such exploration is occurring, it is doing so inconspicuously.

We also do not claim that developers can easily see into their own goals and needs; they are only human, after all. We note only that what corresponds to elicitation may be more straightforward since the communication step vanishes.

B. Software Game Mods

Many software games are extensible and thus can be modified by their users to produce new games, ranging from simple modifications obviously similar to the host game to others almost unrecognizable as related to their hosts.

User modified computer games, hereafter referred to as *game mods*, are a leading form of user-led innovation in game design and game play experience. Game mods, modding practices, and modders are in many ways quite similar to their counterparts in the world of OSS development, even though they often seem isolated to those unaware of game software development. Modding is increasingly a part of mainstream technology development culture and practice, and especially so for games. Modders are players of the games they reconfigure, just as OSS developers are users of the systems they develop. There is no systematic distinction between developers and users in these communities, except for the many users/players that contribute little beyond their usage and their demand for more such systems. Modding and OSSD projects are in many ways comparable experiments to prototype alternative visions of what innovative systems might be in the near future, and so both are widely embraced and practiced as a means for learning about new technologies, new system capabilities, new working relationships with potentially unfamiliar teammates from other cultures, and more Scacchi (2007).

Game conversion mods are perhaps the most common form of game mods. Most such conversions are partial, in that they add or modify in-game characters, game resources such as weapons, potions, or spells, play levels, zones, landscapes, game rules, or play mechanics. In these cases the conversion can often best be described in terms of provisionments of the host game. More ambitious modders go as far as to accomplish either total conversions that create entirely new games from existing games of a kind that are not easily determined from the originating game, or even parodies that implicitly or explicitly spoof the content or play experience of one or more other games via reproduction and transformation.

One of the most widely distributed and played total game conversions is the *Counter-Strike* (*CS*) mod of the *Half-Life* first-person action game from Valve Software. The *CS* mod attracted millions of players preferring to play it over the original *Half-Life* game. Other modders began to further convert the *CS* mod in part or fully, to the point that Valve Software modified its game development and distribution business model to embrace game modding as part of the game play experience provided by the *Half-Life* product family. Valve has since marketed a number of *CS* variants. As of 2011, Valve Software had sold over 25M copies of *CS* and its descendants Makuch (2011).

Fig. 3. A screenshot of *Chex Quest*, a nonviolent mod of *Doom* (image courtesy of user Vulpis Alba)

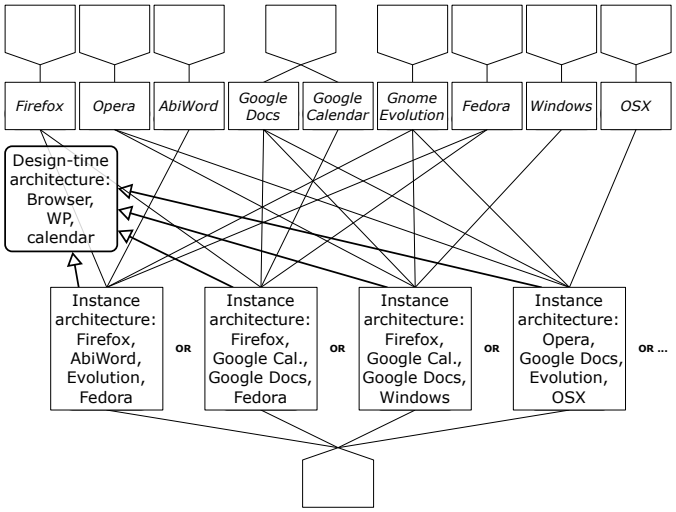

Fig. 4. Ecosystem from which instantiations of the system architecture can be drawn

Other player-modders have produced meta-mods, or mods that can themselves be modded, such as *Garry's Mod* of *Half-Life 2*. *Garry's Mod* has evolved into a modding toolkit used in hundreds of game conversions and producing inventive game play mechanics. Game conversions can also exhibit innovations in game design and re-purposing. The game *Chex Quest* is a conversion of the first-person shooter game *Doom* into a "non-violent" game distributed in Chex cereal boxes and targeted to young people and gamers (Figure 3).

Extensibility to support the creation of mods has become a necessary feature for a successful game.

C. Open Architecture Software Ecosystems

As we note in our previous work Alspaugh et al. (2009, 2013, 2010); Scacchi and Alspaugh (2012b), a substantial number of development organizations have adopted a strategy in which a software-intensive system is developed with an open architecture (OA) Oreizy (2000), integrating components that may be OSS or proprietary with open application programming interfaces (APIs). Such systems evolve not only through the evolution of their individual components, but also through replacement of one component by another, possibly from a different producer or under a different license. With this approach, the development organization becomes an integrator of components largely produced elsewhere and interconnected through open APIs, with shim code added as necessary to achieve the desired result. This approach allows development of large systems of complex components, with relatively little coding needed. Requirements artifacts and processes are not prominent here. Instead, we see a prototyping process and a system described in terms of provisionments rather than requirements.

One reason that reasoning with provisionments is appealing for OA systems is that the integrator cannot choose arbitrary functional capabilities. Instead, there are a limited number of alternative components to select among, and one must simply take what is available. As the components evolve the same situation recurs, in that the functional capabilities may change from version to version, and the integrator must work with what is available. The most straightforward approach is simply to reason based on what the selected components provide.

A second reason is that individual components such as Firefox do not come with Classical Requirements that could be used to reason about requirements for the overall system.

The possible components that can be incorporated into a system define an ecosystem for it. Figure 4 sketches a potential ecosystem for a system composed of a web browser, word processor, email and calendar component, and any scripts and shim code the integrator produces to knit them all together and achieve the desired functionality. If we hypothetically consider the requirements of the composed system, we note that the requirements would necessarily be decentralized, since whatever requirements process we used for the overall system would be independent of that used for each individual component. If we were able to get requirements for each component (which in general is not possible) and integrate them to arrive at requirements for this version of the overall system, this central requirements artifact would last only until the next component version was released, sending the situation back to decentralized requirements.

In practice, integrators appear follow the lead of the developers of the OSS components, and work with provisionments. The acceleration of evolution caused by integrating the independent supply chains for the components currently selected is driving a need to understand decentralized requirements and reason in terms of decentralized provisionments.

VIII. Related Work

A. Requirements in open source development

Scacchi was the first to systematically observe and posit the idea that OSS system and development processes do not rely on producing and review of formal functional requirements

documents Scacchi (2002). Instead, OSS development projects commonly rely on "software informalisms," no matter what the application domain, nor who the developers may be. Such informalisms are rendered within online artifacts like bug reports, messages in a discussion forum, online chat transcripts, etc. that developers use to communicate their interests about different aspects of a system, its development, its user experience, or its need to evolve in some way. He found that OSS requirements often were described after the functionality they prescribe had already been implemented and found to be viable or practical—requirements after the fact. By 2009 Scacchi had identified a set of twenty-odd different types of informalisms in use across different open source development (OSSD) projects, such that a given project might routinely use 5-10 informalisms, with different projects utilizing different mixes of software informalisms so that no specific set seems to dominate Scacchi (2009). The informalisms identified were (a) project email; (b) discussion forums, electronic bulletin boards, and group blogs; (c) news postings; (d) instant messaging; (e) project digests summarizing (a)-(d); (f) usage scenarios as linked Web pages or screenshots; (g) how-to guides; (h) to-do lists; (i) Frequently Asked Questions lists; (j) project Wikis; (k) traditional system documentation; (l) external publications; (m) project licenses; (n) open software architecture diagrams; (o) intra-application functionality in scripting languages; (p) externally developed software modules ("plug-ins"); (q) software modules reused from other OSS projects; (r) project Web sites or portals; (s) project source code Web directories; (t) project repositories such as CVS; (u) bug reports; and (v) issue tracking databases such as Bugzilla. Provisionments may be found in many of these informalisms—especially (a-e), (u), and (v)—but the category of provisionments is orthogonal to them and, we believe, significant in itself.

German described five sources for requirements for the GNOME project, based on his experience as a contributor German (2003). He terms them *vision* (a leader proposes a list of requirements), *reference application* (an outside system is to be imitated), *asserted requirement* (arising from discussions among contributors), *prototype* (an implementation illustrating a proposed feature to be discussed), and *post-hoc requirement* (like a prototype, but offered as a ready-to-integrate implementation of a feature the contributor desires). Provisionments are most closely involved with German's prototypes and post-hoc requirements.

Noll examined the published requirements document for the Web browser version Firefox 2.0, identifying where each of the 14 items was first mentioned, how it was implemented, and why each was initially proposed Noll (2008). Eight were asserted by developers from their personal experience or knowledge of user needs, three were requested by users, and one was driven by a feature in competing browsers. This highlights that although OSS developers are themselves users, non-developer users also play a role in OSS evolution.

Noll and Liu also examined requirements for the OpenEMR electronic medical records project, finding comparable proportions contributed by developers vs. users Noll and Liu (2010). Each feature was briefly discussed in the project's online developers forum, which they characterized as requirements validation and agreement. We found the OpenEMR requirements or features to be more difficult to classify, for example "Support for deleting immunizations", and hypothesize that each acts as a token for the corresponding forum discussion.

B. Requirements and architecture

The close relationship between requirements and architecture suggests that the affordances provided by requirements in classical development may somehow be provided through architectural means in OSSD.

Nuseibeh proposed the Twin Peaks model as an expression of the interrelation of requirements and architecture: problem concerns and solution concerns cannot in general be addressed in sequence, rather needing to be addressed concurrently Nuseibeh (2001). The model conveys a back-and-forth alternation treating both requirements and architecture in increasing detail.

De Boer and van Vliet argue that the traditional distinction between requirements and architecture is misguided, and that there is no fundamental difference between them, saying "architecturally significant requirements [ASRs] are in fact architectural design decisions [ADDs], and vice versa" de Boer and van Vliet (2009). Both are optative statements characterizing what is desired, and by their nature earlier optative statements constrain what later optative statements can be made.

Alspaugh et al. found that of systems with published development artifacts, only toy systems for textbooks have both complete requirements and a complete architecture Diallo et al. (2007). Of the remainder, roughly half had a complete architecture, another quarter had complete requirements, and the remainder had neither. We believe this occurs because requirements and architecture are to a certain degree redundant, so that developers have no need to develop both fully.

All this work suggests that if expected OSS requirements artifacts or processes do not appear to be present, the purposes of those artifacts and processes may be being achieved through architectural means.

IX. Discussion

A. Are OSS Requirements "Good"?

This is a fascinating question to which we have no definitive answer.

In one sense, the answer is "most definitely not". The previous career of one of us (Alspaugh) included work as a developer, team lead, manager, and consultant occasionally called in to help struggling development projects. In each case the struggles could usefully be ascribed to problems with the project's requirements artifacts and processes, in that attacking those problems brought the projects in each case onto a path that could (and usually did) lead to success, and the OSS requirements-like artifacts and processes we examined evoke the problematic ones of those projects.

However, the OSS data we examined in this study was not from troubled projects but from flourishing ones. We conclude

that at least some of the work that Classical Requirements accomplishes is being done in another domain with processes appropriate to that domain; our hypothesis, potentially supported by some of the data we examined, is that some of it is being done in the software architecture domain, through processes that are more what we would expect though here again the artifacts do not appear to be overt.

We note again that CSS bug reports and feature requests and the processes for managing them look much like those for OSS.

B. Centralized vs. Decentralized

Rather than a single central requirements or provisionments repository or document, updated as necessary, OSS projects almost universally appear to use email threads, electronic bulletin boards, and similar sequences of archived interactions as a record of them (and of virtually everything else, it appears).

This choice prevents overall consideration and analysis of the provisionments as a whole. However, it may support a deeper goal for OSSD projects: creating and sustaining a community of contributors. The ongoing conversation, archived online so potential contributors can dip into it to see if interests them, provides an ongoing sequence of nudges to participate and a continuing reinforcement of community membership to those who do participate. This may more valuable and fundamental than any incremental benefits likely to accrue from unifying the information into a single document.

C. Is OSSD efficient?

There does not appear to be data on this question yet. It is not clear that successful OSS projects produce results as expeditiously or more so than CSS projects do; they may well be slower in calendar time or take more person-months. Certainly the importance of schedules and budgets in CSS could drive more efficient development. Brooks notes that one would expect communication to be a more serious bottleneck for OSS than for CSS Brooks (2010), though we note this may be ameliorated by the reduction or elimination of communication between developers and stakeholders, since OSS developers are themselves users and stakeholders.

D. Would OSS Benefit from Classical Requirements?

Perhaps, but the answer is not clear; at this stage, we can only speculate. If the user-developers are identifying stakeholder needs sufficiently well and those needs are addressed sufficiently well by the incremental revisions that appear to characterize OSSD, then probably not. However if the needs would be best addressed by a reconsideration of the problem and a more radical change in the solution, Classical Requirements has advantages to offer.

We note the truism that a new solution to a problem opens the eyes of its users to new problems not previously considered. A product that is evolving at a sufficiently rapid pace (and OSS systems are considered to evolve rapidly) may be obtaining many of the benefits of problem-space requirements processes through solution-space development processes.

E. Are Provisionments Advantageous?

We see an increasing trend of rapidly-evolving systems described and reasoned about in terms of whole-system provisionments, or of component provisionments related through the system's architecture Alspaugh et al. (2009, 2013, 2010); Scacchi and Alspaugh (2012a). This may not only be increasingly typical but also in fact the appropriate approach for reasoning about a stakeholder problem and complex system solution, that is to be implemented by combining complex components. Such an approach manages complexity by reasoning in terms of the capabilities of known (though often themselves complex) components, arranged in architectural configurations in which the capabilities combine to address a problem. It manages ongoing evolution by describing future behavior in terms of differences from past behavior.

F. Are Provisionments Limited to OSS?

No, they are not; we have seen them in our work as professional CSS developers, most prominently in bug reports and to a lesser extent in feature requests where they serve the same purposes as in OSS.

Some professional CSS developers with whom we have discussed this research report that the requirements they work with might frequently be more accurately described as provisionments. And as we noted in Section VII-C, OA system development often appears to be guided by reasoning with provisionments, whether the integrators are an OSS project or a proprietary development group, and with good cause.

As we and many other researchers have noted, there is now far more data available from OSS development projects than there is from CSS projects, to which researchers typically have limited or no access. We recall the challenges we have faced in attempting to get access to proprietary development requirements in order to do research. Based on our results so far, we expect provisionments will be found to be in wide use in OSS development, or even in virtually universal use since they align so naturally with reported OSSD processes. It will be more difficult to assess the degree to which provisionments are used in CSS development, but based on what we have learned, we believe their use is widespread there also.

X. Conclusion

In this paper we examined the apparent contradiction between the success of at least some OSS systems and their lack of what may be termed classical requirements artifacts and processes or Classical Requirements, discussed in Section III. In Section IV we listed four research questions. Here we summarize the answers arising from our study and our examination of related work.

(RQ1) To what extent do OSS projects in fact use Classical Requirements? In the data we examined, Classical Requirements was almost completely absent. We found requirements-like artifacts and some requirements-like processes, but virtu-

ally nothing exhibiting the three characteristics by which we defined Classical Requirements in Section III.

(RQ2) Where OSS projects do not use Classical Requirements, what artifacts and processes are used instead, if any? The most prominent requirements-like artifacts we identified were provisionments (Section VI), statements of features or qualities in terms of the attributes provided by an existing software version, a competing product, or a prototype produced by a developer advocating the change it embodies. These were ubiquitous in the data we examined. The processes were more difficult to characterize; perhaps the most common requirements-like process we saw was the discussion of provisionments in terms of solution-space issues. We hypothesize that architectural reasoning and discussion played a role as well, but did not find strong evidence for it; we may have been looking in the wrong places for that.

In summary, OSS's lack of Classical Requirements results in some of the undesirable outcomes predicted by the broad consensus of software experts and researchers, but not all of them. In some contexts the advantages of OSS appear to outweigh this disadvantage. Further research will be needed to obtain more definitive answers and to provide guidance to making the most effective use of OSS development approaches.

Acknowledgments

This research is supported by grant #N00244-12-1-0067 from the Acquisition Research Program at the Naval Postgraduate School, and by grant #1256593 from the U.S. National Science Foundation. No review, approval, or endorsement implied.

The authors thank the anonymous reviewers of an earlier version of this paper for their careful and insightful comments.

References

Alspaugh, T. A., Asuncion, H. U., and Scacchi, W. (2009). Intellectual property rights requirements for heterogeneously-licensed systems. In *17th IEEE International Requirements Engineering Conference (RE'09)*, pages 24–33.

Alspaugh, T. A., Asuncion, H. U., and Scacchi, W. (2013). The challenge of heterogeneously licensed systems in open architecture software ecosystems. In Jansen, S., Cusumano, M., and Brinkkemper, S., editors, *Software Ecosystems: Analyzing and Managing Business Networks in the Software Industry*. Edward Elgar Publishing.

Alspaugh, T. A., Scacchi, W., and Asuncion, H. U. (2010). Software licenses in context: The challenge of heterogeneously-licensed systems. *Journal of the Association for Information Systems*, 11(11):730–755.

Basili, V. R., Caldiera, G., and Rombach, H. D. (1994). The Goal Question Metric approach. In *Encyclopedia of Software Engineering*, pages 528–532. John Wiley and Sons.

Boehm, B. (1976). Software engineering. *IEEE Transactions on Computers*, 25(12):1126–1241.

Brooks, Jr., F. P. (1975). *The Mythical Man Month: Essays on Software Engineering*. Addison-Wesley, first edition.

Brooks, Jr., F. P. (1987). No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19. Reprinted from *IFIP Congress,* Dublin, Ireland, 1986.

Brooks, Jr., F. P. (2010). *The Design of Design: Essays from a Computer Scientist*. Addison-Wesley.

Creswell, J. W. (2003). *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*. SAGE Publications, Thousand Oaks, CA, USA, second edition.

de Boer, R. C. and van Vliet, H. (2009). Controversy corner: On the similarity between requirements and architecture. *Journal of Systems and Software*, 82(3):544–550.

Des Rivières, J. and Wiegand, J. (2004). Eclipse: a platform for integrating development tools. *IBM Systems Journal*, 43(2):371–383.

Diallo, M., Sim, S. E., and Alspaugh, T. A. (2007). Case study, interrupted: The paucity of subject systems that span the requirements-architecture gap. In *First Workshop on Empirical Assessment of Software Engineering Languages and Technologies (WEASELTech'07)*.

Gause, D. C. and Weinberg, G. M. (1989). *Exploring Requirements: Quality Before Design*. Dorset House, New York.

German, D. M. (2003). GNOME, a case of open source global software development. In *International Workshop on Global Software Development (GSD'03)*.

Jackson, M. (1995). *Software Requirements and Specification: a lexicon of practice, principles and prejudices*. Addison-Wesley, Wokingham, England.

Lamsweerde, A. v. (2009). *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley.

Makuch, E. (2011). Counter-Strike: Global offensive firing up early 2012. http://www.gamespot.com/6328645.

Mockus, A., Fielding, R. T., and Herbsleb, J. D. (2002). Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):309–346.

MozillaFoundation2006-fr (2006). *Firefox2/Requirements*. Mozilla Foundation. http://wiki.mozilla.org/Firefox2/Requirements, accessed 27 Jan 2013.

Noll, J. (2007). Innovation in open source software development: A tale of two features. In Feller, J., Fitzgerald, B., Scacchi, W., and Sillitti, A., editors, *Open Source Development, Adoption and Innovation: IFIP Working Group 2.13 on Open Source Software*, pages 109–120. Springer.

Noll, J. (2008). Requirements acquisition in open source development: Firefox 2.0. In Russo, B., Damiani, E., Hissam, S., Lundell, B., and Succi, G., editors, *Open Source Development, Communities and Quality (IFIP — The International Federation for Information Processing)*, pages 69–79. Springer-Verlag.

Noll, J. and Liu, W.-M. (2010). Requirements elicitation in open source software development: a case study. In *3rd International Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development (FLOSS '10)*, pages 35–40.

Nuseibeh, B. (2001). Weaving together requirements and

architectures. *IEEE Computer*, 34(3):115–117.

Oreizy, P. (2000). *Open Architecture Software: A Flexible Approach to Decentralized Software Evolution*. PhD thesis, University of California, Irvine. http://www.ics.uci.edu/~peymano/papers/thesis.pdf.

PostgreSQL (2013). About. http://www.postgresql.org/about/, accessed 30 March 2013.

Raymond, E. S. (2001). *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly, revised edition.

Ritchie, D. (1984). The evolution of the Unix time-sharing system. *AT&T Bell Laboratories Technical Journal*, 63(6):1577–1593.

Salus, P. H. (1994). *A Quarter Century of UNIX*. Addison-Wesley.

Scacchi, W. (2002). Understanding the requirements for developing open source software systems. *IEE Proceedings—Software*, 149(1):24–39.

Scacchi, W. (2007). Free/open source software development: Recent research results and emerging opportunities. In *6th Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2007)*, pages 459–468.

Scacchi, W. (2009). Understanding requirements for open source software. In Lyytinen, K., Loucopoulos, P., Mylopoulos, J., and Robinson, B., editors, *Design Requirements Engineering: A Ten-Year Perspective*, pages 467–494. Springer-Verlag.

Scacchi, W. and Alspaugh, T. A. (2012a). Designing secure systems based on open architectures with open source and closed source components. In *International Conference on Open Source Systems (OSS 2012)*.

Scacchi, W. and Alspaugh, T. A. (2012b). Understanding the role of licenses and evolution in open architecture software ecosystems. *Journal of Systems and Software*, 85(7):1479–1494. http://www.sciencedirect.com/science/journal/01641212/85/7.

Sommerville, I. (2004). *Software Engineering*. Addison-Wesley, 7th edition.

Stallman, R. (2007). Linux and the GNU system. http://www.gnu.org/gnu/linux-and-gnu, accessed 30 March 2013.

The Standish Group (1994). The CHAOS report.

van Vliet, H. (2000). *Software Engineering: Principles and Practice*. John Wiley & Sons, second edition.

Wiggins, A. and Crowston, K. (2010). Reclassifying success and tragedy in FLOSS projects. In *6th International Conference on Open Source Systems*, pages 294–307.