The best of adaptive and predictive methodologies: Open source software development, a balance between agility and discipline

International Journal of Information Technology and Management
Volume 11, Number 1-2/2012, Pages 153-166
DOI: 10.1504/IJITM.2012.044071

Chitu Okoli
John Molson School of Business
Concordia University
1550, boul. de Maisonneuve West, Montréal, QC H3G 1M8, Canada
Chitu.Okoli@concordia.ca

Kevin Carillo
School of Information Management
Victoria University of Wellington
PO Box 600, Wellington, New Zealand
kevin.carillo@gmail.com

Chitu Okoli is an associate professor in the John Molson School of Business at Concordia University, Montréal, Canada. His primary areas of research are open content, open source software, and applications of the Internet in developing countries. In the past, he has also researched strategic uses of the Internet, as well as various other topics in information systems. His main subjects of teaching have been management information systems, business data communications, object-oriented programming, and Web development. He earned a Ph.D. in 2003 from Louisiana State University, Baton Rouge, USA.

Kevin D. Carillo is currently a Ph.D. candidate at Victoria University of Wellington, New Zealand. His research interests focus on e-commerce and open source and open content communities. He earned an M.Sc. in Business Administration in 2006 from Concordia University, Montreal, Canada.

The best of adaptive and predictive methodologies: Open source software development, a balance between agility and discipline

Abstract

Open source software development (OSSD) is a promising alternative for synthesizing agile and plan-driven (e.g. waterfall) software development methodologies that retains most benefits of the two approaches. We contrast the traditional systems development life cycle approach, more recent agile software development methods, and OSSD. We compare the first two approaches with OSSD, highlighting its synthesis of benefits from both, with unique benefits of its own, offering solutions to areas where the other methodologies continue to face difficulties. OSSD is highly responsive to user needs, and draws talent from a global team of developers. OSSD is a low-risk methodology with potentially high return on investment. While not appropriate for all applications, especially those where the needed applications are extremely idiosyncratic to one company, it is nonetheless a valuable asset in an organization's portfolio of software development solutions.

Keywords: Software development methodologies, open source software development, agile software development, systems development life cycle, waterfall model, extreme programming.

As technology constantly reshapes work environments at a dramatically pace, software development is increasingly subject to conflicting forces due to ever-increasing uncertainty. On one side, such turmoil in the business environment constrains users to iteratively redefine their needs whereas on the other side, software development companies search for stable planning to allocate their resources and to efficiently control software production processes in order to meet customers' expectations. To answer both needs, the software development field has been prolific in introducing innovative methodologies for the last 25 years that can be arranged in an "adaptive-to-predictive" continuum (Barry W Boehm & Turner, 2004, pp. 165-194). However, only a small minority survived to be used today. Barry Boehm, a major contributor in the field of software engineering, places the widely-accepted plan-driven and agile methods on the two extremes of the planning emphasis spectrum. Many real-world examples argue for and against both methodologies leading to an ideological battle between fervent proponents of both sides. However, the resolution may not reside in one of the extremes but rather in a combined approach, as Barry Boehm (2002) suggested:

> Although many of their advocates consider the agile and plan-driven software development methods polar opposites, synthesizing the two can provide developers with a comprehensive spectrum of tools and options.

Open source software development (OSSD) appears to be a particularly attractive candidate for a solution, as it combines and integrates the main strengths of agile and plan-driven methods. Based on Boehm's analysis, Warsta and Abrahamsson suggested placing the OSS paradigm between the agile and plan-driven approaches (Abrahamsson, Salo, Ronkainen, & Warsta, 2002). They build on Boehm's (2002) framework that compares the methodologies at the organizational level by focusing on developers, customers, requirements, architecture, refactoring, size, and primary objective.

Rather than the focusing on the stated purpose of each of the three methodological approaches, we believe it is valuable to contrast agile, open source and plan-driven methodologies from the pragmatic perspective of their differences in the various stages of a software development project. We believe that practitioners could benefit from such an analysis by identifying the key sensitive factors during project development. To present this analysis, we first review the traditional systems development life cycle (SDLC) approach in section I. In section II, we then review more recent agile software development methods (which we will refer to as "agile methods") that try to resolve some of the ongoing problems, and then introduce open source software development in section III. The major part of this article, section IV, follows with a comparison of the first two approaches with OSSD, highlighting its synthesis of benefits from both, with unique benefits of its own, offering solutions to areas where the other methodologies continue to face difficulties. We conclude in section V with a summary of the benefits of OSSD.

I. Plan-driven approach: the SDLC methodology

Among the plan-driven methodologies, the traditional **systems development life cycle** (SDLC) is the most widely used conceptual model in project management to describe the

stages involved in an information system development project (Blanchard & Fabrycky, 2006; Jessup, Valacich, & Wade, 2008). The "waterfall" model is the most popular version of the SDLC, but not the only form (Royce, 1970; Barry W Boehm, 1988; McConnell, 1996). The SDLC approach involves a number of systematic stages (usually four or five, depending on how the stages might be categorized) whose goal is to thoroughly understand users' needs, craft a solid design to meet these needs, and implement a functional system that satisfactorily fulfils the needs. The stages are: **identification and planning**, where the project is justified; **analysis**, where user and project needs are understood in detail; **design**, where a thorough, detailed specification of the solution is created; and **implementation**, where the programming, testing, and installation are executed. Some add **maintenance** as a fifth stage that continues for the life of the system (Jessup et al., 2008).

The traditional strengths of this approach have been its ability to manage very large projects, its attention to quality assurance, and its long-term scalability (assuming a properly executed design stage). However, it has several well-known shortcomings. First, the SDLC depends on the assumption that the users' requirements are understood from the outset, and that they will not change significantly during the process of development, nor soon afterwards. When these assumptions do not hold (and they often do not), this approach can result in working systems that do not correspond to users' present and actual needs.

Another fundamental problem is simply that the process takes too long from commencement to when the end-users can begin using the system. While the length of the project depends on its size and the available resources, SDLC projects are designed such that they developed systems are not available to users until the process is completed, which could take from several months to a couple of years. This implicitly means that SDLC projects have a relatively high risk of finishing over budget and behind schedule, or not finishing at all. The longer the project lasts, the more likely that user needs will change—often quite dramatically—before the completed system is delivered, thus not satisfying their needs when eventually delivered. The cost of large software projects is notoriously difficult to accurately estimate, and extended time frames aggravate this effect (Jones, 2007; Lederer & Prasad, 1992).

II. Agile software development

A number of development methodologies have been formulated to resolve many of these shortcomings of the plan-driven approach, including prototyping, rapid application development, and object-oriented analysis and design, among others. One particularly notable class of solutions that emerged in the mid 1990s has come to be called **agile software development**, which we will refer to here as "agile methods" (Wikipedia, 2008a; Abrahamsson et al., 2002; B. Boehm, 2002; Cunningham, 2001). While Extreme Programming (XP) (Beck, 1999) is the best known of these methods, others such as Scrum (Ambler, 2008) and the Dynamic Systems Development Method (DSDM) (Stapleton, 1999) hold to a similar philosophy. Agile software development methods are characterized by four key characteristics, outlined in the Agile Alliance's "Manifesto for Agile Software Development" (Manifesto for Agile Software Development, http://agilemanifesto.org).

1. **Individuals and interactions over processes and tools:** Generally, programmers do not work individually in their cubicles, alone with their computers. Rather, they work closely together (in XP they program in pairs), and review each other's work in a high-interaction environment.
2. **Working software over comprehensive documentation:** Traditional development might not produce any software for several months or even a year or two. It uses careful documentation as its primary evidence of productivity and basis of accountability. In sharp contrast, agile methods release working software (though perhaps very rudimentary at first) in regular periods of two to six weeks, and work is centred on this "live documentation".
3. **Customer collaboration over contract negotiation:** Agile methods heavily emphasize customer (end user) collaboration throughout the life of the project. XP even insists that a non-technical customer representative should be a permanent, full-time member of the project for its entire lifespan.
4. **Responding to change over following a plan:** Agile methods de-emphasise lengthy periods of analysis and design, which usually take half the time of an SDLC project. Rather, they get right to work, churning incrementally functional releases (not just prototypes) every three or four weeks. They expect customer needs (or desires) to change often, and rapidly change direction to accommodate the customer.

As their name implies, agile methods emphasize rapid development and high flexibility through proven practices such as test-driven development and refactoring. Instead of carefully laid-out designs, agile methods use short "timeboxes" of focused work to produce working, thoroughly tested releases, terminating the production process only when the client is satisfied. Because they don't have a huge fixed investment in a lengthy design process, they are flexible enough to change direction as quickly as customer needs might change, so they lend to high user satisfaction.

However, because of certain features of agile methods, they have thus far been limited to smaller-scale projects with a certain limited set of characteristics. They have not been widely used for projects with more than 20 developers, nor in those where the developers are geographically distributed. The emphasis on high physical interaction between developers, where everyone participates in every module of the project is hard to achieve for very large projects, and impractical when the developers are not geographically collocated.

Furthermore, it is questionable whether the de-emphasis of careful planning can produce systems that are sufficiently reliable for mission- or life-critical applications. While agile methods do emphasize heavy testing during the release production month to month, it is questionable if their testing is as rigorous as that in a carefully conducted traditional plan-driven approach. The nature of some system applications is such that they cannot afford to wait for production use to discover hidden flaws.

A third, very practical concern about the feasibility of agile methods is that they require a radically different organizational culture to be completely executed. Developers must adapt to working very closely together—in fact, many programmers' first reaction to XP's idea of

coding in pairs is, "That is literally impossible." The organizational clients must also adapt to heavy non-technical user involvement on the project, and to receiving incomplete increments and responding constructively to them. While none of these is as impossible as it seems, changing organizational culture is by far the toughest challenge in implementing agile methods, and the primary reason why any half-hearted attempt at implementation would invariably flounder. They require full commitment to the philosophy and methodology in order to realize benefits.

III. Open source software development: a productive bazaar

Open source software development (OSSD) continues to puzzle outside observers even as its importance and influence steadily grows (Fitzgerald, 2004; Hann, Roberts, Slaughter, & Fielding, 2002; Kavanagh, 2004; O'Reilly, 1999; Spinellis & Szyperski, 2004; Wikipedia, 2008b). Relying upon an alternative approach to developing and distributing software, open source communities have been able to challenge and often outperform proprietary software by often enabling better reliability, lower costs, shorter development times, and a higher quality of code. Behind the software is a mass of people working together in loose coordination: "No quiet, reverent cathedral-building here—rather, the Linux community seemed to resemble a great babbling bazaar of differing agendas and approaches … out of which a coherent and stable system seemingly emerges only by a succession of miracles" (Raymond, 2001).

In contrast to the sacred cathedral-like software development model that gave birth to most commercial and proprietary systems, such bazaar-like communities seem to base their success on a pseudo-anarchic collaboration of developers. The basic definition of open source software as expressed by the Open Source Initiative (www.opensource.org) goes far beyond the notion of merely gratis code. It encompasses broader issues such as distribution and licensing that stipulate free exchange and modification rights of source code. The key tenets involve:

- Both source code and compiled binaries must be made available at no financial charge. The original source code must be complete.
- Users must be permitted to modify the program and redistribute their modifications.
- The license may not discriminate against any person or group, and must permit any area of application (in particular, commercial reuse must be permitted).

In addition, some open source licenses, such as the popular GNU General Public License, include a controversial "copyleft" provision that requires that any distributed derivative works be released as open source with copyleft. Protagonists call this feature "share and share-alike", believing that it promotes communal sharing of software as a public resource; critics consider it a poison pill that restricts the commercial use of open source software. However, many open source projects are licensed without such a requirement. Although there have been some questions about the legal validity of open source licenses and accusations of violation of intellectual property rights in open source projects, the cases that have been tested in courts have consistently affirmed the legal status of this model (Rosen, 2004).

It could be argued that OSSD is not actually a different development methodology, mainly from the perspective that an OSSD project could follow a more plan-driven development process or a more flexible process, employing elements of agile methods. However, the fundamental difference lies in the key defining characteristic of open source development: the software is built by various developers from various organizations who contribute sections of code according to their own interests and preferences. Both plan-driven and agile approaches maintain full control over the programmer resources, and direct precisely what parts of the project the programmers would work on, and how they would work on them. In contrast, with OSSD, a significant portion of the project cannot be placed under the structure of either a plan-driven or an agile approach, since the project does not control how the volunteer contributors function; it can only request and suggest. Indeed, for OSS projects that are mainly sponsored by an enterprise or not-for profit organization (such as the Mozilla project) that pays its own developers, the core aspects that the sponsor-paid developers work on could—and often does—follow a plan-driven, or maybe even agile, approach. However, this paper focuses on the open source collaboration aspect of OSSD, which is its distinguishing characteristic.

IV. Comparison of software development methodologies

OSS is far more than just freeware—in fact, most enterprise-grade OSS is not acquired free of charge. The open source software development (OSSD) model, when compared to plan-driven and agile methods, provides intriguing solutions to many of the challenges that organizations face in software development. While by no means a panacea to all the development ails we have outlined above, this approach to software development by a community of developers who produce a common public pool of software resource introduces fresh solutions to long-standing challenges. Using the framework of the SDLC, we will compare and contrast the solutions that the SDLC, agile methods, and OSSD offer (summarized in Table 1).

A. System identification, selection, and planning

The differences among the three methodologies are most evident in their approach to the software development problem. To begin with, the SDLC will not engage on a project without careful assessment of project feasibility, risk, and return on investment (ROI) analysis. However, these are very difficult to accurately estimate; thus the riskiness of the project is usually quite high. One thing is certain, though—SDLC projects are always quite expensive due to their lengthy analysis and design stages, and it usually takes one to three years to produce a complete, working product. In sharp contrast, agile methods deliberately avoid lengthy project feasibility analysis in favour of getting to action. Since there are no large up-front investments, it is not overly costly to back out if things don't turn out well. Agile projects advance rapidly, often producing a working (though incomplete) product within a month, followed by monthly working releases, and producing the final product in six to twelve months.

Table 1. Comparison of the System Development Life Cycle, agile methods, and Open Source Software Development

	SDLC	Agile methods	Open Source
System identification, selection, and planning			
Project feasibility, risk, and ROI analysis	Carefully considered, but difficult to accurately estimate; thus high risk	Deliberately not carefully considered—relatively easy to back out, since there are no large up-front investments	Relatively low investments required, so low risk: losses tend to be low and ROI tends to be high, but benefits are proportional to resource contributions
Financial cost to developing enterprise	Very high due to lengthy analysis and design stages	Moderate—comparable to cost of a small traditional project	Low due to free development contributions by outsiders
Time frame	Relatively lengthy period to first working (but complete) product	Very brief to working (incomplete) product, with monthly releases; relatively shorter period to final product	Lengthy period to become significantly large; then incremental releases as often as weekly; product continuously improved throughout its lifespan
Number of developers	Particularly designed for large projects with more than 20 developers	Typically less than 20	Participating organization contributes any number they want; entire community might have anywhere from just a few up to hundreds of developers
Organizational culture required	Centralized and well-organized	Decentralized and highly flexible	Has worked well with both centralized and decentralized cultures
Product ownership	Proprietary to developer organization	Proprietary to developer organization	Publicly owned source code—attracts outside contributors
System analysis			
Responsiveness and flexibility towards client needs	Carefully considered, but inflexible once set	Highly responsive and extremely flexible to changes	Highly responsive and generally flexible to changes
System design			
Handling of design complexity	Handles complex designs well	Deliberately "refactors" to avoid complexity: simpler projects are easier to manage	Open source communities self-organize very well for complex projects
Solid design	Particular focus—spends much time to ensure good design	Particular weakness, but flexible enough to restart often; compensates by responsiveness to users' changing needs	Varies widely from project to project, but low cost to restart, though this can waste a lot of time
Scalability for future upgrades and extensions	Depends on good design, which is presumably the case	If poor design decisions were made, flexible enough to restart from scratch, though not without cost	Extremely scalable and extensible due to open source licenses
System implementation			
Geographically distributed development	Can be accommodated in implementation design	Generally, not a feature of this methodology, though sub-teams could exist	Intrinsic and essential feature of this methodology
Quality assurance	Thorough testing built into the process	Continuous testing, but of questionable reliability	"Given enough eyeballs, all bugs are shallow"—continuous, rapid, incremental error detection and correction

OSSD, however, has significant advantages over both traditional plan-driven and agile approaches. Fundamentally, because organizational participants in OSS projects make relatively low investments, their risk is correspondingly low: losses thus tend to be low and ROI tends to be high. This does not mean that it is all gain for no cost, though: actual benefits are directly proportional to the number of developers the organization contributes to the project, since these developers will focus on pieces of the software most valuable to the organization, while benefiting from the other general parts that other contributors focus on.

It is important to note though, that while agile methods specifically target high speed, it is not possible to put a whip behind volunteer contributors to an open source project. Thus, OSSD does not move particularly quickly, especially in the early stages of the project when only a few individuals or organizations are interested. They slowly gain momentum, and then only pick up steam when they have attained a minimal level of usability and value. This process can take two or more years—often longer than the SDLC process. This process will usually take over a year when starting from scratch. For example, Linux took around two years from inception (1991 to 1993) to its first commercial release (Slackware)—a significant milestone, indicating that it had become good enough to be worth paying for, which could arguably be used as an indicator of "success." However, the Apache web server took just over a year to go from zero to become the Web's most popular web server (1995 to 1996). Other than these wildly popular examples, most successful OSS projects might take a year or two to attain critical mass of developer support. However, once they attain a critical point where they are valuable in their own right, albeit imperfect, the project size mushrooms exponentially and the most valuable products can charge ahead precipitously.

There are a number of important considerations in deciding on which systems development methodology is most appropriate. The SDLC is particularly designed for large projects with more than 20 developers, and works with an organizational context that is centralized and well organized. Agile methods, on the other hand, typically involve fewer than 20 programmers because of the necessity of a high degree of interaction, and they require a decentralized and highly flexible organizational structure to work effectively. OSSD has worked well in organizations with both centralized and decentralized cultures; considering that the development actually goes on outside of the organization, the approach has succeeded with widely diverse organizational cultures. For example, the software of Netscape Communications, a company with a typical dot-com culture, is based entirely on the open source development; while IBM, a relatively traditionally-cultured organization, invests billions of dollars and hundreds of developers on Linux, and makes enormous profits doing so. However, with OSSD, there is the need for a significant mental paradigm shift from the concept of exclusively owning all software produced by the organizations developers to the concept of shared ownership—the very characteristic that permits an organization to attract outsiders to create software for its benefit at little or no cost.

B. System analysis

The main purpose of system analysis is to assess what the client needs to make sure that the system meets those needs. The SDLC, on one hand, spends a lot of time carefully

investigating and understanding user requirements. On the other hand, a major limitation is that once these needs are identified and set, they are inflexible after the system design and implementation begins, since "scope creep" would quickly kill an SDLC project. Unfortunately, for projects that take as long as a year to design and implement, user needs almost always change at least to some degree by the time the project is launched. In sharp contrast, agile methods incorporate continuous client input and extreme flexibility to changing needs—this is normal and expected, and is built into the methodologies. Similarly, OSSD is highly responsive to client needs since the client *is* the developer—that is, in a commercially developed OSS project, the user organization contributes its own developers dedicated to their piece of the project (we do not refer here to purely volunteer-based OSS projects, which understandably have a high rate of failure and abandonment). Because OSS projects give the full source code to any participant, even if the entire project might not move in a direction they would prefer, they have the flexibility to extend their own piece of the code to meet their precise needs. In OSSD, this is in fact the norm. In fact, that alone, apart from the low cost of development, is one of the primary reasons of participation in OSSD, since the user organization has full control over the software on which they depend.

C. System design

In designing a system, the SDLC is well suited for handling complex designs because of its systematic approach towards the design process. This yields systematically thought-out designs when this phase of the project is properly executed, and ensures a software platform that the organization can build upon for many years of upgrades and extensions. In contrast, well-crafted designs are not a particular hallmark of agile methods, which can sometimes lead to costly restarts when the software needs to be upgraded down the line. However, this point of neglect is by design, so to speak: the agile philosophy holds that complexity in design is in itself a bad thing: it unnecessarily complicates the problem. Thus, when an agile project encounters an extremely complex design problem, it will "refactor"—that is redefine and restructure—the problem so that a simpler solution or a number of simpler solutions can satisfactorily tackle it. This does mean that sometimes many starts and restarts are necessary, which would be intolerable in an SDLC project, but is normal and expected in an agile project, though not without cost.

Open source communities, however, are remarkably effective in self-organizing for very complex projects. Contrary to the common perception, not all participants in OSSD are programmers hacking out code. Some senior programmer participants serve primarily in a "software architect" role (this is Linus Torvald's primary role today in Linux development, though he does still codes (Joe Brockmeier, 2007)), and they spend a lot of time together to carefully map out the long-term design of the software platform. As we remarked earlier, OSS projects are not quick processes—the major ones are ongoing incremental projects with long timeframes in view. However, smaller projects often do not have the benefit of a good designer on board; thus it is quite common for OSS projects, after they have reached the critical point of sufficient value and public attention, to completely restart from scratch to have a platform that can be solidly built upon indefinitely. This was the case with the Mozilla project, that finally gave up on the code base that Netscape contributed to it after a couple

years of frustrating revisions, and created a completely new code base upon which they have been building Internet client software since. Moreover, with the liberty given by open source licenses, there is no administrative red tape to hinder innovative design decisions—if anyone disagrees with the current direction of the project, they are free to take the code and launch out in a bold new direction. This has been the case of the open-source Unixes: FreeBSD, OpenBSD, NetBSD, and DragonFlyBSD being branches or "forks" off the Berkeley Software Distribution version of Unix.

D. System implementation

In the actual implementation (programming, testing, installation and deployment) of a system, there are two aspects we will discuss that highlight valuable and desirable features of OSSD. First, there has been increasing attention to the merits of geographically distributed software development. While this is often implemented for cost-savings, such as off-shoring development to low-income countries, we prefer to emphasize the value of drawing from a wider pool of developer talent who could not otherwise be collocated for development due to the impracticalities of relocation and immigration. In addition to having a larger pool to draw from, there is the benefit that programmers located in the different countries and cultures in which the eventual users of the software are situated would be more likely to be cognizant of the local needs, and could enrich the project with their local insights.

We would consider the SDLC generally neutral regarding distributed development—while not explicitly conceived for distributed development, it is well able to accommodate this. However, this approach does not generally work for agile methods. Indeed, physical contact is doubtless the richest form of communication, and agile methods strongly rely on high levels of physical interaction among developers, and between developers and users. Unfortunately, they thus forego the also valuable benefits of distributed development, which have no place in agile methodologies. In the case of OSSD, however, distributed development is the nexus of the methodology, being an intrinsic and essential feature. Indeed, although the open source philosophy has existed since the 1960s, it was not until the rapid rise of Internet usage within developer communities in the late 1980s that OSSD was able to really blossom through this cheap and practical medium for communication.

A second, very important consideration in the implementation of systems is on quality assurance of the final product. The SDLC is generally acknowledged as a reliable method for producing high-quality, even mission-critical or life-critical systems. Of course, quality always depends on the execution of the final product testing, but the SDLC has thorough testing of the system built into the process. Agile methods heavily depend on continuous testing, but they take a just-enough approach of creating programmatic tests of the functionality of each monthly release—if their tests pass, they consider the software releasable. In addition, the end-user members of the development team use the software in real life, as a kind of continuous beta-testing. However, this approach is generally less rigorous than the thorough testing regime of the SDLC, and few organizations would bet their company on such software.

OSSD in a sense takes the best of both approaches, but adds to it the benefit of hundreds or thousands of ongoing active beta-testers. The open source philosophy of testing is encapsulated in Linus's Law (named after the creator of Linux): "Given enough eyeballs, all bugs are shallow" (Linus Torvalds, 2001; Raymond, 2001; Wikipedia, 2008c). With enough developers and users continuously using the incrementally produced software and constantly working with the raw source code, bugs are identified continually, fixed rapidly by any developer, and the corrected versions are released as often as weekly. Because of the high responsiveness of open source communities to users' input, ordinary users are accorded the influence of trusted beta-testers, and their input is considered and frequently incorporated. The effectiveness of this approach depends entirely on the size of the developer and user community, but with projects with as few as 15 to 25 developers and with hundreds or thousands of users, it is extremely effective in producing highly reliable code.

However, one concern for mission- and life-critical applications is that open source licenses almost universally disclaim all guarantees of fitness for any particular purpose, meaning that there is no one to sue if something should go wrong. And it has to be that way, since a problem could potentially be introduced by any one of hundreds of developers. Thus, concerned user organizations must judge for themselves the quality of the code, with little legal recourse in the case of serious problems. That said, one commercial strategy with open source software is the simultaneous release of a paid commercial version backed by the open source software sponsor's own developers. This is the approach taken by Sun Microsystems's open source acquisitions StarOffice/OpenOffice.org office application suite, MySQL database engine, and Virtualbox virtual machine, as well as a number of Linux distributions such as Red Hat, Linspire and Xandros. Thus, concerned buyers have the option to pay for the comfort of having a corporate backer of their open source software.

V. Summary of OSSD benefits

Most information systems managers think of open source only in terms of being an interesting type of freeware (which in fact it often is not, especially for enterprise-grade applications). Few have even considered that it might possibly be a feasible systems development methodology that is flexible enough to meet the challenges they face in their software development projects. Inheriting characteristics from both predictive and adaptive methodologies, OSSD is an appealing alternative to the agile and plan-driven views that feature opposing benefits and shortcomings in comparison with each other. OSSD is not a compromise position between these two approaches, but rather a third perspective that retains most of the benefits of the two, while offering several unique benefits of its own.

OSSD is a relatively low risk proposition with low potential losses (because of the relatively low investment), but with correspondingly high potential return on income. While new participants are best off joining existing projects that have been around for a few years, they should also considered releasing existing internal software projects to the OSS community to solicit outsiders to contribute to their projects. And one nice thing is that while certainly requiring a major change in mindset, there is no need to attempt a revolution in organizational

culture to benefit from OSSD, as would be the case with a switch from plan-driven to agile methodologies.

For companies that mainly outsource their development or acquire finished software, OSS is an important option due to its high responsiveness to user needs—at the worst, they could simply hire a few developers to extend and customize the pieces that are important to them. Thus, there is little fear of being stuck with software that cannot be extended due to lack of continued vendor support.

Furthermore, open source software benefits by drawing talent from a large number of developers distributed throughout the world—more than half of open source developers are outside the United States and Canada, and 25% are from outside the Western world (Ghosh, Glott, Krieger, & Robles, 2002; Lakhani, Wolf, Bates, & DiBona, 2002). Not only does this give access to a large talent pool with global perspectives, but it also ensures a continuous rigorous quality assurance process.

VI. Limitations of OSSD and ongoing questions

Of course, OSSD is not appropriate for all situations. Specific projects may be better suited to either agile or plan-driven methods due to particular product specifications and characteristics. For instance, OSSD would be little help if an organization has a very particular need in software that is not a need shared by outsiders—contributors would not be forthcoming. Moreover, for new small projects, OSS takes a relatively long time to really kick off. While there might be a handful of outside contributions in the early stages, there generally needs to be a demonstrated base and demonstrated need before outside contributors would come flocking—this stage generally takes a few years to arrive, the exceptional runaway OSS successes notwithstanding.

Furthermore, sound strategy indicates that an organization should be very cautious before releasing the code of systems that are sources of competitive advantage. However, with an appropriate open source business model, companies such as IBM (Linux), Red Hat (Linux), and Sun (Java and MySQL) depend heavily on open source software for some or all of their core sources of value. Thus, OSSD should be considered as just one approach among others.

There remain some ongoing questions about how OSSD can be harnessed to reap the benefits of the SDLC methodology and agile methods. Although this paper has laid out the potential of OSSD, by its approach and structure, to implement the benefits of these other approaches, it is not clear how these could be systematically incorporated in an OSSD project. Most likely, different parameters of a project (such as number of end-users, number of programmers available, budget, and programming culture of core developers) would affect which elements of the SDLC or agile methods could be most effectively appropriated. Similarly, there would be need of research to compare the success rates of SDLC project, agile projects, and OSSD projects, based on common parameters. Success would be measured by metrics such as user satisfaction, monetary costs, and time to launch satisfactorily useable implementations. Because of the significantly different philosophies of

these approach, some standard metrics such as completion rates ("completion" of a project is defined quite differently among the three approaches) and original requirement attainment (this is more and SDLC goal; agile methods and OSSD treat requirements as moving targets) would not be suitable measures across methodologies.

Fortunately, OSSD is flexible enough for any organization to choose to use it in some cases and not in others, as needed. For its many benefits, organizations with software development needs should seriously consider incorporating it into their portfolio of solutions, and particularly as an answer to the need for hybrid approaches of agile and plan-driven methods. As projects tend to require a mix of typical agile and plan-driven characteristics, OSSD is an appropriate solution to fulfil the balance between agility and discipline.

References

Abrahamsson, P., Salo, O., Ronkainen, J., & Warsta, J. (2002). *Agile software development methods: Review and analysis*. VTT Technical Research Centre of Finland.

Ambler, S. (2008). "Scaling Scrum - Meeting Real World Development Needs. *Dr. Dobbs*. Retrieved August 15, 2008, from http://www.drdobbsonline.net/architect/207100381.

Beck, K. (1999). *Extreme Programming Explained: Embrace Change*. Boston, MA: Addison-Wesley.

Blanchard, B. S., & Fabrycky, W. J. (2006). *Systems Engineering and Analysis*. Prentice Hall.

Boehm, B. (2002). Get Ready for Agile Methods, with Care. *Computer*, *35*(1), 64-9.

Boehm, B. W. (1988). A Spiral Model of Software Development and Enhancement . *Computer*, 61-72.

Boehm, B. W., & Turner, R. (2004). *Balancing agility and discipline: A guide for the perplexed*. Boston: Addison-Wesley.

Cunningham, W. (2001). *Manifesto for agile software development* (Vol. 2005).

Fitzgerald, B. (2004). A critical look at open source. *IEEE Computer*, *37*(7), 92 - 94.

Ghosh, R. A., Glott, R., Krieger, B., & Robles, G. (2002). *Free/Libre and Open Source Software: Survey and Study*. Report of the FLOSS Workshop on Advancing the Research Agenda on Free/Open Source Software, European Commission.

Hann, I. -., Roberts, J., Slaughter, S. A., & Fielding, R. (2002). Economic Incentives for Participating in Open Source Software Projects. In *23rd International Conference on Information Systems*. Barcelona, Spain.

Jessup, L. M., Valacich, J. S., & Wade, M. R. (2008). *Information systems today: Why IS matters* (2nd ed.). Toronto: Pearson Education Canada.

Joe Brockmeier. (2007, January 19). Linus and Andy together again: Day three at Linux.conf.au. *Linux.com*. Retrieved August 14, 2008, from http://www.linux.com/feature/59670.

Jones, T. C. (2007). *Estimating Software Costs*. McGraw-Hill Osborne Media. Retrieved August 14, 2008, from http://portal.acm.org/citation.cfm?id=1199222.

Kavanagh, J. F. (2004). Resistance as Motivation for Innovation: Open Source Software. *Communications of the AIS*, *13*(36).

Lakhani, K. R., Wolf, B., Bates, J., & DiBona, C. (2002). *The Boston Consulting Group Hacker Survey*. Boston Consulting Group and Open Source Developers Network.
Lederer, A. L., & Prasad, J. (1992). Nine management guidelines for better cost estimating. *Communications of the ACM*, *35*(2), 51-59. doi: 10.1145/129630.129632.
Linus Torvalds. (2001). Prologue. In *The Hacker Ethic* (1st ed., p. 256). Random House.
McConnell, S. (1996). *Rapid Development: Taming Wild Software Schedules* (1st ed., p. 680). Microsoft Press.
O'Reilly, T. (1999). Lessons from open-source software development. *Communications of the ACM*, *42*(4), 33-37.
Raymond, E. (2001). *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. Sebastopol, CA: O'Reilly.
Rosen, L. (2004). *Open Source Licensing: Software Freedom and Intellectual Property Law* (p. 432). Prentice Hall PTR.
Royce, W. (1970). Managing the Development of Large Software Systems. In *Proceedings of IEEE WESCON* (Vol. 26, pp. 1-9). IEEE.
Spinellis, D., & Szyperski, C. (2004). How is open source affecting software development? *IEEE Software*, *21*(1), 28 - 33.
Stapleton, J. (1999). DSDM: Dynamic Systems Development Method. In *Technology of Object-Oriented Languages and Systems, 1999. Proceedings of* (p. 406).
Wikipedia. (2008a). Agile software development. In *Wikipedia*. Retrieved August 14, 2008, from http://en.wikipedia.org/wiki/Agile_software_development.
Wikipedia. (2008b). Open source software. In *Wikipedia*. Retrieved August 14, 2008, from http://en.wikipedia.org/wiki/Open_source_software.
Wikipedia. (2008c). Linus' Law. In *Wikipedia*. Retrieved August 14, 2008, from http://en.wikipedia.org/wiki/Linus's_Law.