eXtremely Distributed Software Development

white paper

TechnoPark Corp.
470, 555 Bryant St,
Palo Alto, CA 94301
(860) 506 5536

In this document we introduce eXtremely Distributed Software Development (XDSD) methodology and explain how it reduces risks and improves quality.

Are We Getting Any Better?

Jeff Atwood said in his blog four years ago: “The history of software development is a tremendous success. Just look around you for evidence of that. But that success has a long, dark shadow that we don’t talk about very much: it’s littered with colossal failures. What’s particularly disturbing is that the colossal failures keep recurring year after year.” Sadly, since then we have not been getting much better.

CHAOS (2010), the latest report by Standish Group based on the analysis of 70,000 IT projects, shows that the industry “failure rate” is over 68% and is not decreasing. Cerpa and Verner (2009) gives a representative summary of other recent studies, which mostly agree that software quality is not improving, but getting worse every year.

“Billions of dollars are wasted each year on bad software.”
– Bob Charette

Charette (2005) says that while billions of dollars are wasted each year on bad software, few IT projects truly succeed.

Is there a bad pattern associated with projects falling short? Do such cases look similar to each other? In this white paper, we explore those questions and provide an

example of a process that does work.

How It Happens

Imagine that you have a business process based on paper documents, and you discover a strong market demand to make them electronic and available over the Internet.

So you find and hire a reputable software development company with good references and the readiness to make your project happen for a flat fee of $180 per hour. The developer gives a precise and accurate estimate of the project size, equal to 2585 hours, making the total project budget equal to $465,300.

In a few months and after a number of payments made to the developer, you see a demo version of the product. You like it; however it misses a few important features which are critical for the business. The developer documents your suggestions and comes up with a new budget. You approve it and wait for the next demonstration, which happens in a few months.

After the 13th demonstration and four years of work, the budget has increased and $3.1 million has been paid to the developer. The system is still not finished and has not been delivered to your end-users.

A reality check reveals that there is no system in place,

but rather a large amount of messy documentation and software components not compatible with each other. You hire another vendor to get an independent oversight, which reviews the existing artifacts and comes up with a new project budget of $1.8 million, claiming that “it has to be redone from scratch”. However, at this point you are out of time and the budget has run out.

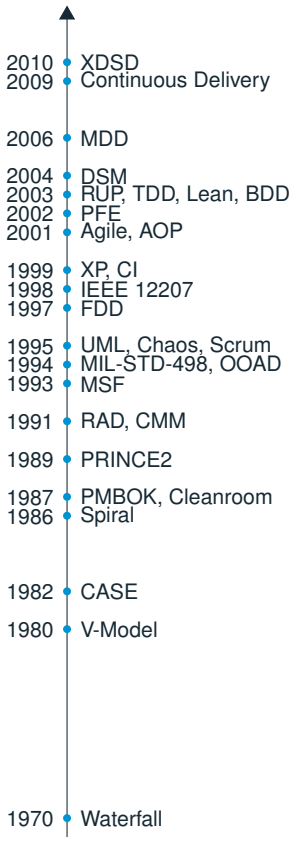
“...independent oversight reveals that there is no system in place”
– audit report

Don’t be surprised if this story sounds familiar. This is not fiction, but rather a summary of what William C. Thompson (2003), New York City Controller, documented in his audit of a project once outsourced by the NYC Department of Health and Mental Hygiene to IBM Corporation.

If IBM, a global corporation, makes such mistakes, how can we avoid them? Is it just impossible due to the “essential complexity” of software?

What History Says

Since the 1960s, when software engineering as an independent profession was born, the success factors of software projects concerned their sponsors. There were two principles discovered in the industry and emphasized by Brooks (1995). First, there is no “silver bullet”. Second, if a “silver bullet” does exist, despite principle one, then the solution is a discipline (also known as formal methods, process, methodology, framework, best practices, etc.).



Source: Wikipedia.org

In most cases, a disciplined development process is more important than effective algorithms, robust architecture, talented engineers, team morale, or open communications (Cooke-Davies, 2002; Emam and Koru, 2008; Sauser et al., 2009).

During the last 40 years, there were many such methodologies introduced which were more or less effective for their particular applications, from Waterfall in 1970 to Continuous Delivery in 2009.

However, despite all of these “proven” formal methods, software project sponsors continue to suffer from cost overrun, missed delivery, and incomplete features.

At the same time, being a software architect is one of the highest paying and least stressful jobs in America, according to CNN Money. Is this fair? How can we leverage this reality to better benefit project sponsors?

There Is An Alternative

To find a solution, we just need to look around. While the situation with commercial software projects is dramatic, open-source industry experiences are quite the opposite. Goldman and Gabriel (2005) presented how free open source software (FOSS) products flourish with great success in every business domain. Mozilla Firefox, Apache HTTP Server, OpenOffice, MySQL, Linux, and

Eclipse are just a few visible examples.

Open source teams tend to deliver quicker and with higher quality. Their end-users experience fast and motivated responses from product designers and developers. OSS products effectively understand and fulfill user requirements. Moreover, in most cases exceptional products are being developed with extremely small budgets.

> “Recent case studies provide very dramatic evidence that commercial quality can be achieved/exceeded by OSS projects.”
> — V. Valloppillil
> Microsoft, 1998

What are the success factors of open source projects, which are absent in their commercial counterparts? There are many of them, as shown by Gurbani et al. (2006) and Herbsleb and Mockus (2003). The most significant are motivated contributors, immediate delivery of a product to end-users, and a bug-friendly development environment. In a typical brick-and-mortar for-profit organization, these factors rarely if ever exist.

During the last two years we have been developing and experimenting with a methodology called eXtremely Distributed Software Development, that applies the most important components of OSS success to commercial teams, without ruining the fundamentals of modern business administration.

XDSD guarantees commercial projects what they usually tend to lack, i.e. manageability, quality of code, high team morale, and motivated user community. Next, we show how XDSD was successfully implemented in a recent commercial web project developed by our TechnoPark Corp. team.

An Ideal XDSD Implementation

TechnoPark Corp. started this project with very little information from the customer. There were no formal requirements, just a USPTO Patent Application for a new computer-mediated communications method.

We were contracted to architecture, design, and implement a workable web2.0 system. Moreover, the top priority non-functional requirements included interface usability, source code maintainability, and extreme scalability. We were constrained to only spend up to four months and $50,000 to produce an extensively tested product ready for a promotion campaign.

The customer planned to invest his own money into a workable prototype and then raise venture capital for marketing and business development.

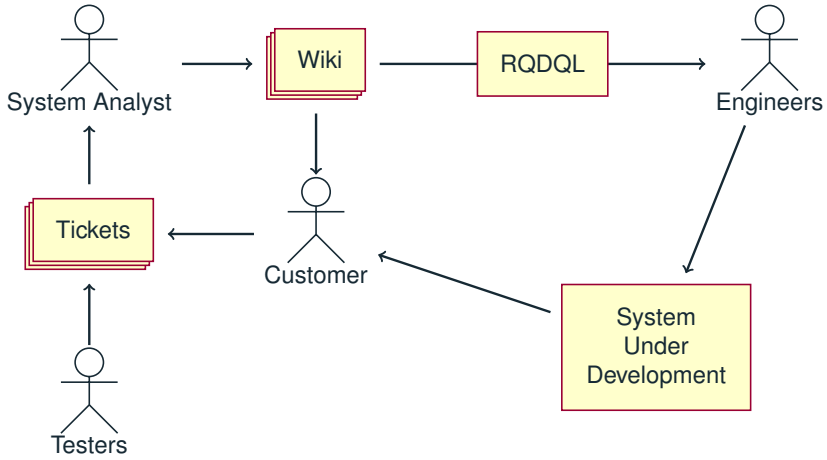
Skills	Ppl	Location	Price	Hours
Web performance architect	1	USA, CA	$130	20
PHP architect	1	USA, NY	$80	50
PHP programmers	2	Poland	€18	380
System analyst	1	Poland	€15	80
Requirements reviewers	2	USA, NY	$40	70
Manual testers	3	China	$6	200
Code reviewers	3	Germany	€40	40
Deployment engineer	1	Russia	$20	20
Interface designer	1	Germany	€40	40
Graphic artist	1	UK	£50	40
Performance testers	2	Belarus	$22	200
Total	19			1140

We found and built a distributed team of narrow-skilled engineers, all of them working remotely from different continents. This is how we planned to spend our budget among 19 people, and the plan proved to be correct.

With every project participant, we signed an individual contract obliging him/her to work according to our rules, with certain awards and penalties.

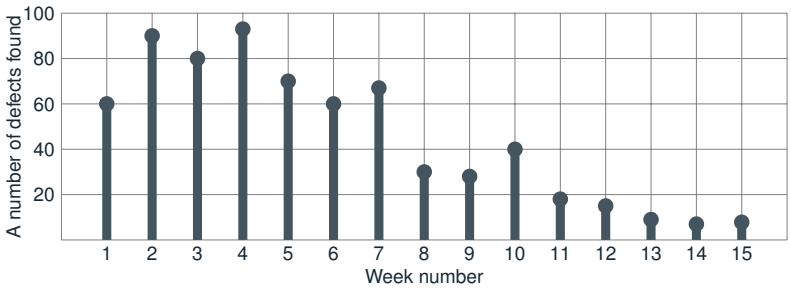
According to these rules, everybody got paid for verified deliverables.

In order to keep our programmers, testers, management, and the customer in-sync in regards to the product scope, we documented requirements in a “wiki”. At the same time, everybody was free to express their ideas and concerns in online tickets.



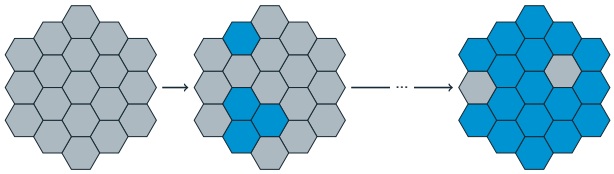
The system analyst summarized the information from tickets and made changes to wiki pages. RQDQL, our proprietary instrument, validated these requested changes for consistency and integrity, and delivered to the engineers in plain text and UML 2.0.

During the project course, we made over 600 changes to the specification. Most of them were made in parallel with programming and testing. The next graph shows the dynamic of changes registration, week by week.



During the project lifecycle, the customer took an active participation in discussions about system functionality with the system analyst, programmers, and testers.

The first version of the system was delivered to the customer and manual testers in nine days after the project began. It was a workable software product, with very limited functionality implemented. During the next 16 weeks, the engineers made over 8000 alterations (check-ins), releasing new versions 10 to 20 times a day. We used fazend.com, a hosted continuous integration platform developed and maintained by our team.



Empowered with Puzzle Driven Development paradigm, the team constructed the software as a puzzle mosaic. Every day we made many visible and tangible micro-steps, continuously approaching the final goal.

The deployment engineer was responsible for keeping the source code consistent and robust from day one. At the end of the project, the system included 29000 lines of code extensively documented and covered by unit tests for 85%.

The quality of product was planned and controlled by means of pro-active bugs classification and enumeration, as originally suggested by Myers (2004). We focused our

testers on the forecasted amount of bugs we were going to fix in the product.

According to the signed contracts the testers were paid for the bugs found and documented, and they showed outstanding results. There were 700 bugs registered of different severities. 97% of them were fixed before the end of the project, and the product was delivered on time and on budget, with all specified features.

The Next Step

At TechnoPark Corp., we keep researching and developing new processes and ways to develop software. There are a number of academic and industry papers we have produced during the last two years that emphasize certain aspects of our achievements, mostly by Bugayenko (2009a, 2010, 2009b).

We look forward to a new software development project with you that uses the XDSD methodology. For more information or to setup a meeting, please call us today at: (239) 935 5429.

References

Brooks, F. P. (1995). *The Mythical Man-Month: Essays on Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition.

Bugayenko, Y. (2009a). Quality of code can be planned and controlled. In *Proceedings of the 1st International Conference on Software Validation and Verification (VALID)*, Portugal, Porto.

Bugayenko, Y. (2009b). Quality of process control in software projects. In *IWSM/Mensura 2009 International Conference on Software Measurement, Software Process and Product Measurement*, Amsterdam, Netherlands.

Bugayenko, Y. (2010). How to Prevent SVN Conflicts in Distributed Agile PHP Projects. *php|Architect*, 9(8).

Cerpa, N. and Verner, J. M. (2009). Why did your project fail? *Communications of the ACM*, 52(12):130–134.

CHAOS (2010). CHAOS summary for 2010. Technical report, The Standish Group International, Inc.

Charette, R. N. (2005). Why Software Fails? We waste billions of dollars each year on entirely preventable mistakes. *IEEE Spectrum*, pages 42–49.

Cooke-Davies, T. (2002). The “real” success factors on projects. *International Journal of Project Management*, 20(3):185–190.

Emam, K. E. and Koru, A. G. (2008). A Replicated Survey of IT Software Project Failures. *IEEE Software*, 25(5):84–90.

Goldman, R. and Gabriel, R. (2005). *Innovation Happens Elsewhere, First Edition: Open Source as Business Strategy*. Morgan Kaufmann.

Gurbani, V. K., Garvert, A., and Herbsleb, J. D. (2006). A case study of a corporate open source development model. In *Proceedings of the 28th International Conference on Software Engineering*, pages 472–481.

Herbsleb, J. and Mockus, A. (2003). An empirical study of speed and communication in globally distributed software development. *IEEE Transactions on Software Engineering*, 29(6):481–494.

Myers, G. J. (2004). *The Art of Software Testing*. John Wiley & Sons, Inc., 2nd edition.

Sauser, B. J., Reilly, R. R., and Shenhar, A. J. (2009). Why projects fail? How contingency theory can provide new insights – A comparative analysis of NASA’s Mars Climate Orbiter loss. *International Journal of Project Management*, 27(7):665–679.

William C. Thompson, J. (2003). Audit Report on the Development and Implementation of the Electronic Death Registration System By the Department of Health and Mental Hygiene. Technical Report 7A03-073, Office of the Controller, Bureau of Financial Audit, New York, NY, USA.