Reducing Risk in Software Projects Using Behavior-Based Requirements

Jeffrey S. Davidson

Abstract

Requirements are the key to implementing the vision of a business or client. Software project failures are a significant factor to lost capital and operational expenses, lost time, and eventually lost opportunity and revenue. While project failures have been well documented, less attention is paid to the similar costs from unused and underutilized features within software projects.

With upwards of 45% of software features never used and an additional 32% rarely used it is time to pay attention to the core needs and jettison the wasted effort, expense, and code that is bogging down project success.

A key methodology in correcting this problem is Behavior Driven Development (BDD). Originally developed to help developers understand the business needs, it has grown beyond its roots and is capable of making a significant impact on curbing excess demands, requests, and gold plating.

Often seen as solely an Agile software development technique, this toolset provides insight into the core functionality of a software product and can lead to significant improvements in user experiences by removing unnecessary functionality before it becomes embedded in modern and future systems.

Introduction

Understanding the goals and objectives should be the first step of every project. In traditional project delivery, the outcomes and quality of a project stems from a firm understanding of the basics, i.e. time, scope, and cost. While each of these three may be fixed outside of the project, the final deliverables depend on how well they are understood and implemented within the project.

When discussing the success and failure of software projects, there are more references to the CHAOS Report by the Standish Group than any other metric. I want you to look at the following

table and understand software projects now, and apparently always have, do not deliver on their initial objectives. Compared to their objectives, 63% of projects are challenged or fail! This number would be shocking if it was a one-time event, but it's not. It is just the latest number in a 20-year trend line.

Table 1: Standish project benchmarks over the years[1]

Year	Successful (%)	Challenged (%)	Failed (%)
1994	16	53	31
1996	27	22	40
1998	26	33	40
2000	28	49	23
2002	34	51	15
2004	29	53	18
2006	35	46	19
2009	32	44	24
2011	37	42	21

Further, Standish goes on to claim 45% of features are never used. Jim Highsmith reports[2] this number is over 50%. Half of the functionality developed in software projects is wasted.

Forrester Research Report, "Corporate Software Development Fails to Satisfy on Speed or Quality," (April 11, 2005) states,

> Corporate development shops continue to disappoint: A fall 2004 Forrester survey of 692 technology influencers—those who hold the information technology (IT) purse strings—indicated that *nearly one-third are dissatisfied with the time it takes their development shops to deliver custom applications*, and *the same proportion is disappointed by the quality* of the apps that are ultimately delivered. *One-fifth of respondents are unhappy on both counts*. [Emphasis added]

The problem goes deeper though. Eveleens and Verhoef quote Standish,

> "Standish defines a successful project solely by adherence to an initial forecast of cost, time, and functionality. The latter is defined only by the amount of features and functions, not functionality itself. Indeed, Standish discussed this in its report: '*For*

[1] Standish Group, CHAOS Reports, http://blog.standishgroup.com/.

[2] Jim Highsmith, "Beyond Scope, Schedule, and Cost: The Agile Triangle," http://jimhighsmith.com/2010/11/14/beyond-scope-schedule-and-cost-the-agile-triangle/.

challenged projects, more than a quarter were completed with only 25 percent to 49 percent of originally specified features and functions.'"[3] [Emphasis added]

Summarily, we have failed projects, we have significant portions of functionality sitting unused, we have unhappy executives, and we have challenged projects delivering less than half the functionality.

There are many ways to run a failed project, but only a few ways to run a successful one. Some of the recurring and significant contributors to these problems are centered on understanding the scope of the problem and solution, involvement of key personnel with the project team, and defining detailed requirements of what needs to be built.

Capers Jones, the pre-eminent expert on software quality stated,

> "Although clear requirements are a laudable goal, they almost never occur for nominal 10,000 function point software applications. The only projects I have observed where the initial requirements were both clear and unchanging were for the specialized small applications below 500 function points in size."[4]

This is the crux of why I propose the standard formats for eliciting and communicating requirements are deficient and need to be revisited. Software delivery project problems will not cease when we have a better understanding of software requirements, but we can start making radical improvements to the decrepit state of affairs in today's projects.

Thinking like an investor

Chris Matts proposes there are only three reasons for developing software; making money (or expanding the organization's mission), saving money, or protecting money, including risk avoidance or compliance dictated. Consequentially, every software development project is an investment.

---

[3] J. Laurenz Eveleens, and Chris Verhoef, "The Rise and Fall of the Chaos Report Figures," *2010 IEEE SOFTWARE*, Jan/Feb, pp. 30-36.

"Standish defines a project as a success based on how well it did with respect to its original estimates of the amount of cost, time, and functionality." Eveleens and Verhoef argue CHAOS Reports measure software projects in comparison to their initial estimates and other factors. As such, they may not be accurate due to problems in the initial estimate.

This paper does not dispute problems with initial estimates, but rather presumes the sponsors of such projects expected them to deliver as promised, even if the estimates are poor.

[4] Capers Jones, Software Engineering: State of the Art in 2005, http://twin-spin.cs.umn.edu/sites/twin-spin.cs.umn.edu/files/STATEOFART2005.pdf.

Jones states large projects, "a size of 10,000 function points is roughly equal to about 1,250,000 statements in the C programming language." Further information on estimating code based on function points may be found at: http://www.qsm.com/resources/function-point-languages-table.

Gene Kim and Mike Orzen "calculated the global impact of IT failure as being $3 trillion (US) annually."[5] This includes $100 billion of waste just from S&P 500 companies and at least $250 billion in failed IT projects. Software delivery projects are failing.

A primary issue is many project failures is Project Managers, Business Analysts, and project teams understand the list of features to be delivered, but do not understand the project's purpose. Many individuals are good at running calculations to determine Cost-Benefit Analysis and Return-on-Investment. Even more teams know where to find the list of features to be built and tested. What teams do not have a deep understanding of is why the business investment is being made and how the organization will benefit from it. They do not shepherd the project as if their own capital was at risk.

The truth of every software project, despite the thinking of so many of my peers, is the value of an IT system is entirely based on the expected outputs or outcomes. Systems are not built because they are esthetically pleasing or to meet the experiential design goals.[6] Rather, IT systems are built to make, save, and protect money.

Over the last decade a number of tools have been developed to assist development teams with grasping how to approach problems based on this mindset. The first of which is Feature Injection.[7]

There are three steps to feature injection:

1. *Hunt the value* – Build a model based on your desired results
2. *Inject the features* – Use the model to decide what features pull us toward the value
3. *Spot the examples* – Use stories to find variants to the happy path and have a conversation

Step 1: *Hunt the value*

The value of a software project needs to be boiled down to a well-understood model. Models should be specific and targeted towards the desired business value. This value must be both clearly defined and communicated to the entire project team. Communicating specific goals and the delivery outcomes expected aligns the team with the organization.

---

[5] Michael Krigsman, "Worldwide cost of IT failure (revisited): $3 trillion" http://www.zdnet.com/blog/projectfailures/worldwide-cost-of-it-failure-revisited-3-trillion/15424.

[6] I wish enterprise systems were more pleasing to the eye and had better experiences for their many users. Michael Krigsman, has a good discussion about how enterprise software is currently directed at management over users in "Enterprise software under attack," http://www.zdnet.com/blog/projectfailures/enterprise-software-under-attack/14709 resulting in poor user adoption and reduced benefit from those systems where users can make a choice.

[7] Further information can be found at http://www.infoq.com/articles/feature-injection-success.

Models require only a concise statement about the output or outcome and how this will be achieved. It does not require a significant document and can often be conveyed in a single paragraph.

Step 2: *Inject the features*
It is by looking at a project through the lens of what value needs to be created we can spot the features that must be added, or injected. This list of features should be only those items that pull us towards the greater value. This is not the same activity as recording a list of desirable or demanded features. Rather, this becomes a focused set of features, driving towards specific outputs defined within the model.

Step 3: *Spot the examples*
Examples are the real world test of what a system needs to be responsive towards. Using the steps above will find most paths to a positive output (value). This step is focused on both ensuring negative paths are understood and the system can be validated as successful. Examples become the communication tool to ensure the system is adequately built.

Of course, the unstated fourth step is to *follow information smells*. That is, look for gaps in understanding, particularly around the information and data being used. This step allows you to both confirm your models completeness and discover what may be missing. When you find something that may be missing, whether a feature or example not adequately covered by the existing set, repeat the above steps to fill in the hole.

Communicating through examples (Behavior-Based Requirements)
The second tool I want to introduce to you is a structured, natural language for learning, writing, and communicating requirements. I want to re-introduce you to storytelling.

> "Storytelling is among the oldest forms of communication. Storytelling is the commonality of all human beings, in all places, in all times. " Rives Collins, *The Power of Story: Teaching Through Storytelling* [8]

The oldest written story is The Epic of Gilgamesh[9], from the land of Ur. It was likely written about 2,700 BC. The actual history of storytelling is unknown. No one knows if the first stories were told to calm an upset tribal member about a sudden storm or why catastrophe occurred on the most recent hunt. No one can say the first time a story was told to explain someone else's behavior. What we can say is storytelling evolved to give meaning and purpose.

[8] http://www.goodreads.com/quotes/33388-storytelling-is-among-the-oldest-forms-of-communication-storytelling-is

[9] http://www.sparknotes.com/lit/gilgamesh/

Example: Storytelling

I want you to imagine being alive three millennia ago, when written language was rare. You live in a tribe with your family and friends. There is nothing anyone in the tribe can do to surprise you. You know everyone else too well. You have been through too much together to be very surprised by anyone. You were born into this tribe and if you had to, you would die for this tribe. You know and trust everyone else in the tribe. And they trust you.

Tonight your tribe is sitting around after a great hunt. You and your tribe have killed a bison and tonight you feast. After the feast, and the children have started to settle down, you sit around the fire. You get quiet and still, but you are excited. Expectant. Everyone knows what's going to happen next, the tribal elder is going to start telling a story.

We use stories to communicate. We don't communicate in facts. The tribal elder didn't say, "Run. Many hills. No water. Throw. Parabolic arc. Dead. Eat now."

No, the tribal said, "The winter was long and it has been many moons since our bellies were full. Yesterday we learned of a great beast, great enough to feed us all. Early this morning we woke and ran from when the sun came up until it was at it's highest in the sky. We are so glad the beast was moving slower than Ugg runs! We were incredibly thirsty, but knew the right shot would bring us victory! Sam pulled back his arm and threw his best spear, the best throw of his life. It was amazing and now we eat like the gods."

And the reason the tribal elder tells his stories with a setting, a sequence of events, and a conclusion, is because that is how we are wired. The human race has been telling stories around the campfire since time we first grunted at each other. The first record of telling stories may be the sons of Cheops entertained their father with stories, but we have been telling stories since time immemorial because it is what works best.

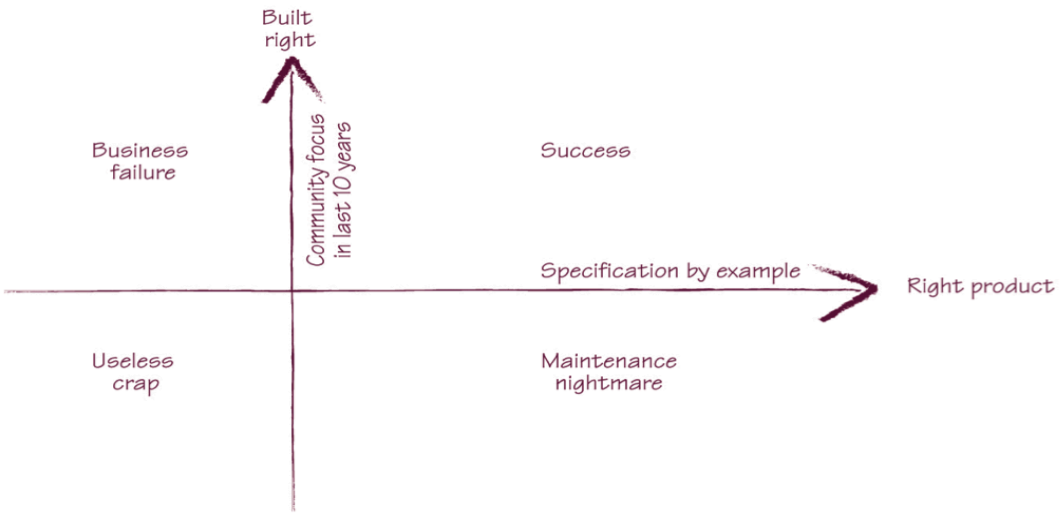
As part of a team, we need to understand and acknowledge the human species is wired to communicate using stories. To not tell stories is to deliberately hurt the chance of your teammates, your organization, and your customers understanding. It becomes simple, telling stories are good. Not telling stories is harmful.

The Wrong Focus

Our focus over the last twenty years has been to build correctly. Teams spend a great deal of money and effort to obtain and master the latest tools. Yet despite significant investments, despite significantly faster workstations, despite untold dollars spent on training, the products we deliver are barely beyond what they were in 1994.

Our focus on the building process is only part of the solution. The time has come to focus on building the right product. Gojko Adzic presents the following chart in his latest book, *Specification by Example*:

Specification by Example[10]



What I am proposing is stories are a key tool in a new focus, a focus on building the right product.

This is especially true when we deal with large, complex subjects. Rich Hickey recently presented "Simple Made Easy."[11] His points are especially poignant for teams striving to make a difference with the software projects. They were:

- We can only hope to make reliable those things that we can understand
- We can only consider a few things at a time
- Intertwined things must be considered together
- Complexity undermines understanding

Building large software projects is a difficult endeavor. We needed to spend the last few decades ensuring we have the proper tools for the job. Success requires more than just tools. Success will come when we take the time and effort to enable greater understanding. Storytelling is the means to achieve this goal. Using behavior-based requirements uniformly promotes this across all types of projects. They are simply a means to capture the functionality of a system as described through fine grained, focused bits of behavior. These small behavioral examples are told in a story format, allowing easy access to readers of all kinds.

---

[10] Gojko Adzic, *Specification by Example*, http://specificationbyexample.com/.

[11] Rich Hickey, "Simple Made Easy," *StrangeLoop 2011*, http://www.infoq.com/presentations/Simple-Made-Easy.

Structure of Stories

The structure for our behavior-based requirements is:

1. *Context* – You and your condition
2. *Event* – What action do you perform?
3. *Outcome* – What is the observed response?

When writing these requirements, they should be captured using the following language:

1. Given … (context)
2. When … (event)
3. Then … (outcome)

After using this technique for a couple years, I have specific recommendations for effectively writing behavioral requirements. First, while the requirements need to be cover exact behaviors, they should be design agnostic. It is not appropriate to include language implying either user interfaces or system architectural decisions within the statements. This level of detail takes practice to master as it is typically less than included within traditional requirements, i.e., "The system shall…."

Second, the language used needs to be natural and not stilted, contrived, or technical. If you cannot get an executive, a data entry clerk, and your grandmother to understand the same sentence, you may not be writing clear enough. Read your statements out loud. If they sound more like a novel than the technical instructions for programming your electronic clock, then you are probably on the right path.

Third, the language needs to use business terms. This means the storytellers (users, executives, and subject matter experts), the people documenting the stories, and the team developing the stories will come to understand more about the business domain. This is a good thing and in keeping with delivering value over a list of features.

Fourth, amplify and reinforce the stories with testable data. For developmental and validation purposes, this proves the delivered code is correct. The story is correct. The understanding is correct.

Power of Stories

There is an underlying presumption we have both misunderstandings and unknown unknowns inherent in existing techniques. In seeking to both document those places and fill them with understanding, the practitioner asks about usage scenarios. These stories are filled with behavior and from this behavior design patterns emerge. The end result of using stories and examples correctly is a shift towards better elicitation techniques.

The simplicity of the grammatical structure belies the power this technique brings over the course of a project. Converting needs and actions to words is inherently messy. Our language is inexact and not designed to convey preciseness. Yet we need a level of precision and accuracy in our project communication. This technique allows us understand and validate finite behaviors in a large context.

This technique also encourages the discovery of what needs to be done to achieve project objectives. Using the discovered requirements throughout the project allows for the understanding project teams have been missing. No longer will one level of understanding be locked in a document team members do not read or have access.

Conclusion

Based on the rates of software delivery project success, your projects should already have a risk calculation around being challenged or outright failing. With a proper and universal understanding of the scope and goals, this risk will be reduced. I propose using behavior-based requirements as a key technique for achieving your common understanding, even it were to raise your project cost or timeline.

This technique is not meant as a direct challenge to the traditional requirements elicitation and documentation. Rather, this is a refinement of how requirements are drawn out and shared. This technique reduces requirements risk in a manner existing requirement structures cannot.

Using this technique we begin to literally capture conversations. And in capturing, we are sharing instead of merely interpreting. The need for business interpreters can be supplanted with understanding. The roll of interpreter and gatekeeper changes to one of communicator and bridge builder. Rather than attempting to push information uphill to a team, understanding can be shared and pulled as it is needed. Sponsors and project teams have concrete and comprehensible proof they are working towards the same ends.

> "*BDD is a second-generation, outside-in, pull-based, multiple-stakeholder, multiple-scale, high-automation, agile methodology. It describes a cycle of interactions with well-defined outputs, resulting in the delivery of working, tested software that matters.*"
>
> Dan North @ Agile Specifications, 2009

Licensing

Author

Jeffrey Davidson, CSPO, PMC is a Principal Consultant with ThoughtWorks, a leading international Agile development firm with a passion to improve how businesses design, build and evolve software. He regularly consults in the transportation and finance industries. He also serves as the current President of IIBA Dallas Chapter.

Jeffrey has had many titles and pseudo-titles, including Director of Business Analysis & Quality Assurance for UTI, Business Analyst for Dell Financial, Systems Engineer for Raytheon, and Product Manager for Ebay.

Jeffrey's multiple contact points are probably easiest to find by checking out his profile on LinkedIn, http://www.linkedin.com/in/jeffreydavidson or his personal blog, http://goodrequirements.com/.

My very favorite resources on this topic are available at http://goodrequirements.com/bdd/.