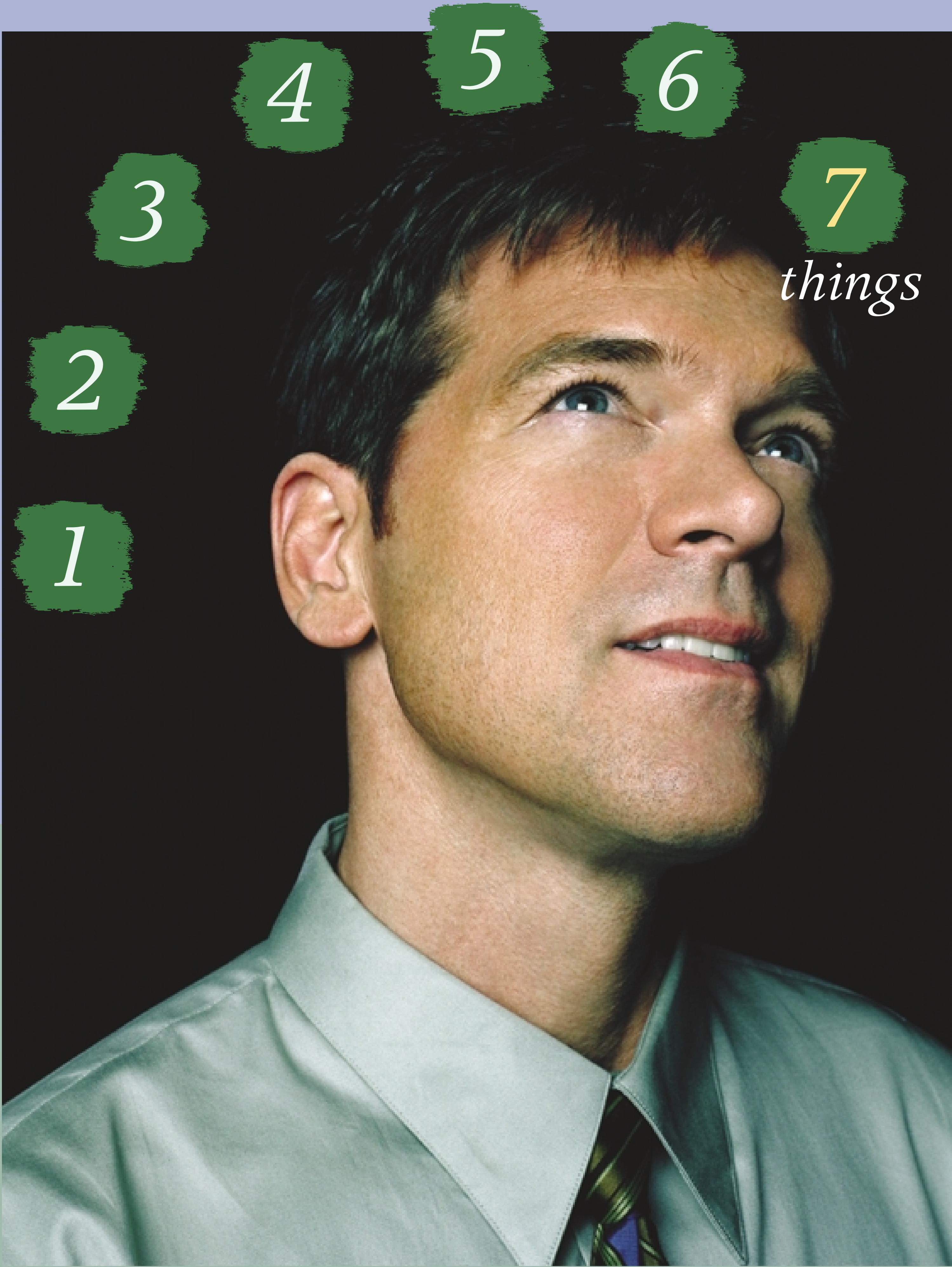
1
2
3
4
5
6
7
things

project customers can do to turn a good product into a great one.

Want Better Software? Just Ask.

by Mike Cohn

PROJECT CUSTOMERS EXIST IN ALMOST EVERY ORGANIZATION BUT OFTEN GO by a different name: product manager, product owner, product champion, and so on. Whatever you call him, a Project Customer is the person in the development process who represents the real users and customers of the software. The Project Customer works with real users and customers to understand their needs. He determines exactly what features will be in the software, communicates users' needs to developers, and coordinates scheduling decisions between the technology group and the rest of the organization. The Project Customer is the driving force behind what gets built into the software.

I'm a developer, but I've worked with enough good (and bad) Project Customers to know that a good Project Customer can turn a good product into a great product—and a bad Project Customer can sink a project

Info to Go

- Set clear, elevating project goals, and expect great things.
- Provide guidance, support, and recognition.
- Prioritize and reprioritize the work as user stories that developers can estimate.

ANDREAS KUEHN/GETTY IMAGES

faster than any bad technology decision. So how can you make sure you're one of the "good" guys? Follow these seven simple rules.

1 Collaborate

If we developers aren't 100% sure you're with us, then we may think you're against us. You need to establish right from the start that as much as any programmer or tester, you are part of the team. Watch how you treat the developers. Are you respectful of our time, or do you cancel or not show for meetings? Do you pass work on to us that you could do, such as answering requests for information from prospects? Do you say, "I need you to finish by May" or "You'll need to work this weekend"? Or do you join the team and say, "We need to figure out how to get this done by May" or "It looks like we'll be here this weekend"?

We need to see you as an authority on what users want and, more importantly, what they value. Demonstrate that you understand the product and how it will be used. Master the software. I've seen too many teams where the Project Customer—who should know the software better than anyone else—has one of the developers give all the demos to prospects. Developers resent this. We know that usually you're having us give the demos because you don't know the software well enough to demo it yourself. Learn the software, and then use demos as an opportunity to show us how well you know the software.

You might want to move as close to us as you can get. Many of the agile processes, such as Extreme Programming and Scrum, advocate that you move from your office or cubicle to sit right alongside the development team. I won't go so far as to say this is necessary, but it sure does help.

2 Communicate the Vision

Software developers need to know why a project is being undertaken. The better we understand the project vision, the better able we will be to help the organization achieve it. We look to you to establish and then communicate the vision. We need what authors Larson and LaFasto call "a clear, elevating goal."

The best example of a clear, elevating goal was President Kennedy's call to land a man on the moon before the end of the 1960s. It was clear: Everyone would know if the goal had been met or not. It was elevating: It is easy to imagine the excitement of being on a team of such

Working in the Dark

In 1994, Windows had not yet completely taken over the PC desktop. There were still plenty of DOS applications around, and they often performed better than their Windows equivalents. However, it was apparent that desktop applications were moving toward Windows. I was working in a company that had a team rewriting its bread-and-butter application for Windows. I wasn't on that team, but I had worked with the developers on a previous project.

Initially, the team completely bought into the vision of the project. It wasn't unusual for me to come into the office at 6:00 AM and find at least one person on that team already hard at work. Eventually, though, their enthusiasm waned. One day, two developers came to talk to me. They told me that their Project Customer had instructed them to leave out absolutely all error handling, do no unit testing of their code, stop doing code inspections, and so on. In other words, do shoddy work. These instructions went beyond the usual "speed up a lot and let quality suffer a little." He was telling them to *ignore* the quality of their work. These developers were concerned about the thousands of users who would eventually buy the software as a replacement for the DOS system. They had been coming in early so they could achieve the productivity their Project Customer wanted, while also coding the system to their personal quality standards. They asked if I'd talk to their Project Customer, my peer, and see if I could talk some sense into him.

I agreed but was shocked at what I heard from him. He had no intention of ever selling the product these two developers were working on. He was convinced that he could keep the customers on the current DOS platform for a few more years. He planned to show the "new version" at an upcoming users' conference, promise that it would be coming soon, then entice customers to sign three-year contracts with the company. He planned to start a real project to develop a true replacement system during this time, but he felt he needed to show a nearly-complete system in order to lock customers in. His analogy for this ruse was to imagine our customers walking around the set of a Hollywood Western. They'd see the facades of various buildings, and it would look like everything was real. But if they looked too closely, they'd see that it was all an illusion. The Project Customer had gone so far as to give the project the code name "Dodge City."

Of course, the developers knew nothing of this and were putting long hours into a product they thought was necessary for the company's survival as the world migrated toward Windows. I convinced the Project Customer to come clean with the developers. They were glad to finally understand the true vision for their product, but obviously they never again trusted anything they heard from that Project Customer. The real shame in this story is that if the true vision had been laid out for these developers, they would have worked just as hard, but their efforts would have been directed at the real goal of the project.

historic importance. Our project goal should be just as clear. Of course, the goal does not have to be quite as elevating as putting a man on the moon, but it should be something we want to accomplish either because the goal itself is meaningful or because of the challenge represented by achieving it.

Here are some examples of clear, elevating goals:

- The product will win a "Product of the Year" award from the magazine covering that industry.
- The product will reduce call duration in our call centers by three minutes per call.
- Make the product so simple to use that we can cut training time from three days to half a day.

Here are two examples of goals from real projects I was on that the team did not find elevating:

- The product will be released with all planned features by 30 May.
- Release the product by September so that we can go public.

The goal of releasing a product by a specific date failed to motivate us, because there was no reason why 30 May was any better a release date than 1 June. Imagine if President Kennedy had instead worded his goal as "put a man on the moon before 15 June 1969." That date has far less impact than "by the end of the decade."

That doesn't mean all date-based goals are bad. As part of preparations for Y2K, for instance, I managed a team that was focused on updating a health care application and deploying it to hundreds of hospitals well in advance of 1 January 2000. That team had a deadline-based goal that it found clear and elevating. Team members knew there would be consequences if a customer remained on the old application a day too long. Date-based goals can be elevating (they are always clear), but the date must have significance.

The goal of releasing a product by a certain date in order to initiate a public offering didn't work well because "make a lot of money" is not an elevating goal for many developers. Making money and going public can be strong motivators and valid goals but, in general, they are not elevating.

Most developers will bend over backwards for a project with a clear, elevating goal. As the Project Customer, you are responsible for making sure the project has one. Once you understand what that goal is, you are responsible for articulating it to the project team. If you can't articulate it, don't start the project until you can. According to Larson and LaFasto, the lack of a clear, elevating goal is the most frequently given reason for why teams fail. (For more on team success factors, see this issue's StickyNotes at www.stickyminds.com/bettersoftware.)

3 Set High Expectations

Have high expectations of us. We want you to give us challenging problems—make this run ten times faster or do that in one-fourth the memory. Developers thrive on these types of challenges when given the freedom and time to pursue solutions. Asking you to have high expectations doesn't mean we're asking for impossible deadlines. Most developers will shrug off an impossible deadline and let the solution take as long as necessary. We don't want "stretch goals" just so you can see if you can make us develop software faster. When we develop beyond a certain speed, we take shortcuts that come back to haunt us (and you). Very few products are worth doing at high speed, and all end up costing far more over the next few versions.

Most developers will shrug off an impossible deadline and let the solution take **as long as necessary.** We don't want "stretch goals ..."

Expect to be given working code at least once a month. Think about the project you've been involved in that had the worst schedule slip. Was it a new 1.0 development effort, or was it an incremental point-release such as 3.1? Most likely it was a project aimed at producing a new version 1.0 product. It is easier to add features to a known, stable product than it is to create a new product from scratch. This means we should never let a product—even a new one—be more than a month away from that perfect, shippable state. Naturally I'm not advocating that every project should be shipped once a month. But you should expect the developers to pull the project together at least that often. Insist that once a month we give you a demo (or let you go hands-on with the software, if you prefer) of the new functionality that was added during the month. Remember that the product should be good enough that you could ship it if you wanted to—meaning it's been coded and tested.

Only trust what you can run. Like Frodo with the ring, we're tempted by the evil within us. There is a part of us that would love to sit around for a month and just think, think, think about the perfect way to design some complicated aspect of the system. Naturally, some of this thinking is necessary, and we need time to do it, but what you can't do is let us count it as progress. Don't let us tell you things such as "We're done with analysis and design, so we're about halfway there." We're half done when half the code is 100% written and 100% tested.

Make us prove it. Every time I turn on my computer, it runs the familiar power-on self-test to check that the hardware is working properly. Imagine how nice it would be if your software did the same thing. What if every night a large suite of automated tests were run against the product, and the results were in your email in the morning? We can do that. Insist that we do.

4 Know Your Priorities

A couple of years ago, I was talking to a colleague, and I made the comment that a particular practice "would be good for time-constrained projects." He challenged me, "Show me a software development project that is not time-constrained." I laughed and said, "You're right. There's no such thing."

All projects are time-constrained to some extent, so prioritize the functionality you want built into the software. If you say, "Everything is top priority," then when the project runs out of time, you can expect a random set of functionality, as we won't have known what was truly the most important piece to finish first. One of the best prioritization techniques comes from the DSDM process and is known as the MoSCoW rules:

- "Must have" features are fundamental.
- "Should have" features are important but have a short-term workaround. If the project has few time constraints, these features are normally mandatory.
- "Could have" features can be left out of the release if time runs out.
- "Won't have in this version" features are desired but acknowledged as needing to come in a later release.

We also need a way to document which features are more important within these categories. We can prioritize features along many dimensions. For example, we can use technical factors:

- The risk that the feature cannot be completed as desired (for example, with desired performance characteristics or with a novel algorithm)
- The impact the feature will have on other features if it is deferred (We don't want to wait until the last iteration to learn that the application is to be three-tiered and multi-threaded.)

However, you may have your own set of factors that could be used to prioritize:

- The desirability of the feature to a broad base of users or customers
- The desirability of the feature to a few important users or customers
- The cohesiveness of the feature in relation to other features (For example, "zoom out" may not be a high priority on its own but may be treated as such because it is complementary to "zoom in," which is a high priority.)

There is an ongoing debate in software development about whether a project team should go after the riskiest parts first or the most valuable parts first. Probably the leading proponent of risk-driven development has been Barry Boehm. The leading advocate of doing the "juicy bits" first has been Tom Gilb (see this issue's StickyNotes for more).

Projects should definitely err on the side of doing the juicy bits first; that is, first do the parts of a system that deliver the greatest value to the intended users of the system. However, this does not mean we can ignore risk. As the Project Customer, you are the one who must make the decision after considering input from us.

Additionally, without knowing the relative cost of each feature, you cannot prioritize them. It is a priority for me to drive a Ferrari until I see the price tag, at which point sending my daughters to college is a much higher priority. Similarly, you may think of some features as "must have" until you consider the cost. Don't prioritize until you know the expected cost, in development time, of each feature.

Who gives those estimates? The developers. Don't force deadlines on the developers; instead, allow them to estimate each feature. Then collaborate with the developers so that the highest-priority work is done first.

5 Tell Us Stories

If you want us to give you an estimate for each feature, it is important that you provide us with features we can estimate. Long lists of "The system shall …" statements are not amenable to individual estimation. It is too hard and too unreliable to estimate statements such as:

- The system shall accept passwords between six and ten characters long.
- The system shall require passwords to contain both alphabetic and numeric characters.
- The system shall require the user to change her password every ninety days.

Features described like this are often too entwined to be estimated independently and effectively. We need to hear about features in chunks that are big enough to understand but small enough to estimate. Extreme Programming has introduced the practice of user stories (see this issue's StickyNotes), which perfectly meet this need. A user story is a short description of functionality that will be valuable to either a user or a purchaser of a system or software. User stories are traditionally handwritten on paper note cards because of their low-tech elegance. Some typical user stories for a job-posting and search site are:

- A user can add a new résumé to the site.

If you want us to give you an estimate for each feature, it is impotant that you provide us with features we can estimate.

- A user can edit a résumé that is already on the site.
- A user can remove her résumé from the site.
- A user can mark a résumé as inactive.
- A user can search job openings.

Unfortunately, programmers cannot take a typically vague user story from a note card and use only that to develop software. The story cards are really reminders for the Project Customer and the developers to talk about a feature. The story card may be the most visible part of a story, but the most important part is the conversations that take place to refine the story and communicate the desired behavior of the software.

The user stories you write for us should be tied to users' goals and lead to achieving the clear, elevating goal of the project. A good Project Customer will not just rattle off a list of stories; she will be able to relate those stories to workflows and knowledge of how users will interact with the system.

A big advantage of user stories is how easy it is to use them with different levels of precision. A project may begin with a list of high-level, large stories (known as "epics") and then refine these as needed into stories that are smaller and easier to work with. Rather than forcing a project to begin by identifying all requirements in great detail, projects can begin with a mixture of epics and smaller stories. Features that won't be started for weeks or months can be left as epics; features that are prioritized for early development can be refined into smaller stories.

6

Change Your Mind

We ask you to prioritize the features at the start of the project, but we also expect you to change your mind. On a project lasting more than a couple of months, it is unrealistic to expect an organization's priorities to remain constant. As a matter of fact, as you learn more about how and to whom we'll be selling our product or service, and as we collectively learn more about the software we're building, you *should* change your mind about priorities.

For example, suppose we're halfway through a project and an opportunity presents itself for us to sell the half-finished product to an initially small set of customers. That might be worth considering. Alternatively, suppose that a fierce competitor announces some new blockbuster feature. Adding that feature to our product may become a higher priority than much of the other remaining work.

Indecisiveness and random changes are annoying, but changes based on new knowledge are an important interruption. In fact, if we've established a development process that meets the high expectations you *should* have of us, our ability to respond to change can become a competitive advantage for our organization.

7

Provide Support and Recognition

On many projects, you represent the entire outside world to us. Give us support and recognition, and we'll be the team you need and want. Withhold these, and we're likely to become bitter and resentful. Support us by making sure we have what we need to keep us productive. Sometimes that means we need more space, or space that's quieter, or space where we can be noisier and have lots of conversations. We may need team rooms or larger whiteboards. We may need the air conditioning left on after 6:00 PM because some of us come in late and work late. Support us through your words and your actions. We know we can't always have what we want, but let us know that if you could get it for us, you would.

Recognition can take a variety of forms, and you need to choose one that is appropriate to the team and its accomplishments. Find out how individuals prefer to be recognized. Some employees crave public recognition, such as being singled out for praise at a company-wide meeting; others dread it. Don't save recognition for the end of a project; being recognized at the end is nice but doesn't do anything for morale in the middle of a project. Remember how you should expect to see working code from us at least once a month? Use those occasions as opportunities to provide recognition. While we're showing you the demo, toss out a compliment or two on pieces you really like. Or if it's appropriate, praise the project in an email to your boss, and copy the team.

Larson and LaFasto make the interesting claim that insufficient support and recognition are most likely to become a problem for both poorly performing teams and teams doing extremely well. A poorly performing team often feels that it cannot achieve its goals without additional support. A team doing extremely well often feels it is getting less recognition than it deserves. So pay extra attention to both ends of the spectrum.

Conclusion

None of the rules I have described require any special skills, training, or domain knowledge. Anyone can do them. Work as part of the project team, prioritize and re-prioritize the work, tell us user stories, give us a clear, elevating goal, set high expectations for our performance, and offer your support and recognition. We'll reward you with superior results. **{end}**

Mike Cohn (mike.cohn@computer.org) has twenty years of experience developing software and is the vice president of engineering at Fast401k. He is a founding member and on the board of directors of the Agile Alliance. Some ideas in this article are taken from his most recent book, User Stories Applied: For Agile Software Development.

Sticky Notes

For more on the following topics go to www.stickyminds.com/bettersoftware

- Factors for team success
- Risk vs. value
- User stories