
Chapter 1

The next bottleneck in software projects

The world of software development went through quite a shake-up in the last decade. Extreme programming and other early agile development practices removed several huge bottlenecks in the way we were used to working, allowing software teams to deliver faster, better and react to changes more easily. Once these problems were out of the way, people started noticing other issues in projects such as planning and organisation. Looking for a better way to manage projects, agile development teams adopted the Scrum methodology. A whole range of related techniques emerged to remove planning and organisational bottlenecks. Agile development and management practices are now mature and widely accepted and I feel confident that the technical part of building software is no longer a problem. The most important issue now is making sure that we know what we need to build, but this problem is a whole lot harder than it sounds.

Of course, this problem is nothing new. Fred Brooks wrote long ago that “The hardest single part of building a software system is deciding precisely what to build”[2]. Practices used in the creation of traditional system requirements documents and specifications have changed very little over the last decade. Some techniques such as user stories touch upon what we should do to get better specifications and requirements, but they fall short of completing the task to the end. The process of eliciting requirements and communicating them throughout the process to everyone involved is often just described in vague terms and left to teams to work out for themselves. With programming practices and project planning now pretty well sorted, this issue is becoming more and more apparent and many teams realise that they struggle with it. This problem is the next bottleneck to be removed from the software development process.

A nice thing about software development the agile way is that we can easily go back and adjust the system, but making changes is not as cheap as most programmers would like it to be. It costs a lot of money and time. For those of you who would now suggest that using agile practices makes this cheap, don't forget that you are only talking about the technical cost. Agile practices help us cope with clients changing their minds for whatever reason, so that in a sense we are hitting a moving target. There is a huge difference between this and missing a still target and then going back to have another go at hitting it. Even if the requirements don't change, there is still a risk that the project can miss a target if we don't solve the communication problems. Although agile practices help a lot with reducing the risk of failure, they should not and cannot be used to cover up for a project that simply does not deliver what it was supposed to. Disappointing a client is never good, agile or not agile. Gerald Weinberg and Donald Gause suggest that the difference between disappointment and delight is not a matter of delivering software, but how well the delivery matches what clients expected.¹ This was true twenty years ago when they wrote *Exploring Requirements*[3], and it still holds true today.

Matching what clients expect is still a problem, mostly because of communication issues. Individual effects of small communication problems are very hard to detect, so they do not become apparent instantly. Such problems are reflected in lots of small things not working as expected or implied features that simply do not get delivered. Individual issues may have small effects on the project, but their cumulative effect is huge. The reason why changes in development practices in the last ten years have not solved this problem is that most of these changes were driven by developers and I do not believe that this particular issue is a development problem at all. It is a communication problem involving all participants in the implementation team. This is why there is no development practice that can solve the problem, whether or not it demands the involvement of customer proxies and business people.

¹ See chapter 18 of *Exploring Requirements*[3]

We all need to agree on what the target is, even if it moves, and make sure that we all have the same understanding. And by *we* I mean all participants in the process from stakeholders to domain experts, business analysts, testers and developers. The path to success is to ensure that these small communication problems get rooted out instead of accumulating, so that the message gets delivered correctly and completely.

The telephone game

The traditional model of gathering requirements and building specifications is based on a lot of formalising, handing over and translating. Business analysts first extract knowledge about requirements from customers, formalising it into specifications and handing it over to developers and testers. Developers extract knowledge from this and translate it into executable code, which is handed over to testers. Testers then take the specifications, extract knowledge from them and translate it into verification scripts, which are then applied to the code that was handed over to them by the developers.

In theory, this works just fine and everyone is happy. In practice, this process is essentially flawed and the usual result is a huge difference between what was originally requested and what gets actually delivered. There can be huge communication gaps at every step. Important ideas fall through these gaps and mysteriously disappear. After every translation, information gets distorted and misunderstood, magnifying the degree of wrongness of the delivered system. A tester's independent interpretation might help to correct the false interpretations of developers, or it might very well be a completely different misinterpretation of the system requirements.

With agile development processes, the feedback loop is much shorter than in a traditional process, so problems get discovered quickly. However, if agile acceptance testing is not applied, even with other agile practices in place, there is still a lot of scope for mistakes. Instead of discovering problems, we need to work out how to stop them from appearing in the first place.

People seem somehow surprised by this effect even though most of us have encountered it and used to amusing effects in childhood. Antony Marcano draws a parallel between the traditional software development process and the *telephone game*.² In the telephone game (called Chinese whispers in the UK) a group of children stand in line and then the first child whispers a phrase or sentence to the next child. The second child whispers what it heard to the third one and so on. The last child in the line says aloud the phrase or sentence that it heard. It is often significantly different from the original phrase. Although these cumulative differences may have been amusing when we were children; they are not so funny when it comes to solving real problems that obstruct people in doing their jobs. What happens with the traditional approach is that quite a few things get lost in translation. Even when people do read big documents carefully, they don't remember all the details. Most people just digest the text looking for key ideas and guidelines, disregarding everything else.

Relying on people to remember all the details is futile. This is why features rarely get implemented completely and correctly at the first attempt, so that the project relies on testers running through the specification documents and comparing them with what has been developed. This again poses a challenge since testers often get involved only when the party is over and they are expected to understand instantly a complex software system without taking too much of the 'precious' developers' time on explaining things. As tight deadlines are a rule rather than an exception these days, testers don't have a lot of time to learn about the development and system specifications. They are often expected to approve a release in just a few days. This makes it hard for them to really understand what they are testing and severely undermines the effectiveness of testing.

Although big specification documents give the impression of constituting complete and comprehensive descriptions of requirements, in practice (at least the ones I have seen) they often leave out some details that need to be worked out later via e-mail or in a whole series of other documents. The original specification is, in such cases, out of date as

²<http://www.testingreflections.com/node/view/7232>

soon as the development starts. Even with perfectly correct, complete and detailed specifications, there is still a possibility that requirements will change during development. This is a rule rather than an exception for longer projects. Changes get implemented directly without updating specifications, leading to differences between code and system documentation. In this case the specifications are technically incorrect. So if the testers identify a difference between the system and the original documents, this does not necessarily mean that they found a problem. This makes the job of testers even harder, because they cannot rely completely on the specifications.

Such differences between what was asked for and what was developed may also come from unclear or inconsistent requirements. Code is unforgiving with regards to specifying how something should work. On this level, we have to define exactly how the system should behave. Once developers start writing code, gaps in functionality become obvious. Unfortunately, getting to domain experts or business people who wrote the specifications at this point may take a while. In the best case, developers need to get the person on the phone or chase her by e-mail. If the person in charge is an external customer or an executive project sponsor, they might not be readily available to talk and it might take a few days to get the right information. In some extreme cases, it is nearly impossible. During a workshop I organised for a large media company, one of the developers said that their business analysts actually refuse to discuss requirements after they were handed over to development. According to their process, the job of business analysts was done at that point.

Because developers need to specify how the system works in the newly identified case and they need to seek out this information, the time to complete a piece of code increases from several minutes to several hours or several days. When the gaps are finally cleared up, the original specification document again may not reflect the full reality. There is at least some new functionality not covered by the document, or it may even conflict with the original requirements.

Later on, when requests for changes and improvements start coming in, especially if the product has several clients with different require-

ments, changes may break earlier rules or introduce inconsistencies into the system. Depending on regression test coverage, this problem might surface or be noticed only in production.

The problem is, of course, that not even the brightest among us can keep track of all the previous requirements and all the changes in their minds for a long period of time. Realistically, it's not sane to expect someone to remember all 500 pages of a specification document and instantly spot that a new change request might break an earlier rule. This becomes even harder when the people involved in the original project move on to other things and are replaced. With big requirements documents, consistency can be a problem even in the initial phase, since they might be the result of several days of discussions with different people, and are sometimes written by several people.

Imperative requirements are very easy to misunderstand

With agile development processes, it is less likely that business analysts will produce great long documents that caused a small forest to be cut down. Requirements are typically broken down and iteratively constructed and communicated. This does not really make a huge difference with regards to the communication problems that I want to attack. Without agile acceptance testing, agile processes will potentially help us catch a problem sooner because on-site customers provide feedback, but they do not prevent the problem from happening. Requirements are simply subject to interpretation and quite often to misinterpretation. Imperative requirements, which command what people should do without demonstrating it, are especially bad as they are seemingly precise but leave quite a lot of potential for mistakes. We each have our own assumptions that affect the way we understand these statements. The fact that we have heard, read and understood something in English does not necessarily mean that we understood it in the same way.

The story about a series of experiments conducted by Lawrence G. Shattuck with four active battalions of the US Army,³ quoted in Gary Klein's book *Sources of Power*[4], illustrates this problem nicely. The researchers attended a military exercise and listened in while the commanders were giving orders to their teams. During the exercise, they wrote down everything that the teams did. After the exercise, they discussed the actual actions of the teams with the commanders. The results of the research were that actions on the ground matched commanders' expectations completely only in 34% of the cases. While I was reading it, it struck me how this story completely relates to my early experiences in software development. The commanders are customers or business analysts, the orders are requirements or specifications and the teams on the ground are developers and testers. We all sit in the same room, listen to the customers, agree on the requirements and specifications and then go away and develop something that often does not match completely what the customers want. I don't know of any similar experiments actually being conducted with software development teams, but I would not imagine that the result is a lot different. In a sense, the story about the same thing happening in the US Army is comforting because it means that we are not the only ones plagued by misunderstandings and wrong assumptions. Unfortunately, this comforting feeling quickly goes away if you think about the fact that these guys have the biggest guns in the world and that they misunderstand orders twice in every three times.

Are obvious things really obvious?

A very serious problem with requirements is taking obvious things for granted. In *Exploring Requirements*[3], Gause and Weinberg described an experiment from one of their workshops that shows how even the simplest things can be misinterpreted. They showed a seven-pointed star picture to attendees before a seminar, then held a lecture and let people have a coffee break. After the break, they asked the attendees to tell them how many points the picture had. They received

³<http://www.au.af.mil/au/awc/awcgate/milreview/shattuck.pdf>

quite a wide spread of results. According to Gause and Weinberg, everyone knows what a five-pointed star looks like but their star was more unusual, so people remembered it differently and recalled different images, especially after the coffee break. This accounted for the spread. People also understood the task differently, which explained clusters in the answers.

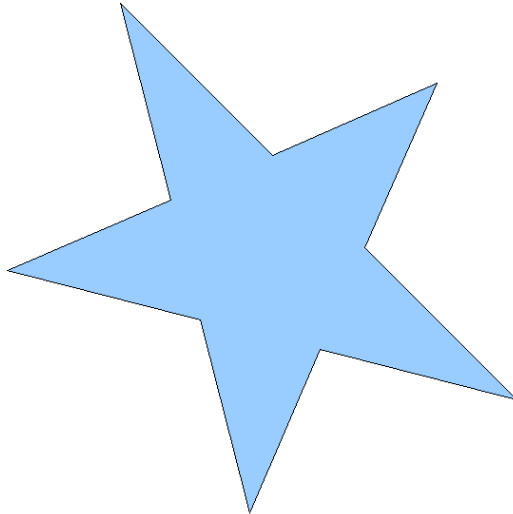
In August 2008, I repeated this experiment,⁴ focusing only on differences in interpretation. During a talk, I handed out index cards, put a picture of a standard five-pointed star on the projector and asked the attendees to write down how many points they saw in the picture. I used an image of a familiar figure (Figure 1.1) and it was on display while people wrote down their answers, so that observation and memory problems would not affect the answers. Any differences in answers could only be caused by people interpreting the task differently. After some initial reluctance and explaining that the question might sound stupid and obvious, but people should answer it anyway, I got more than 40 cards back.

An interesting thing about this experiment is that most people are absolutely certain that there is an obvious answer, because the question seems so simple. The real problem is that although people have an obvious single logical answer in their heads ('the only possible answer'), this answer differs from person to person (Figure 1.2). Twenty five people voted for ten points in the star, counting inner and outer points. The second most popular answer with seven votes was five points, where people counted just the outer points. Six people voted for fourteen – one of the attendees explained that this is probably the ten points on the star and the four corners of the picture. Note that there was no border on the picture, but some people decided to count the edges of the screen as well. There was a single vote for nine points, which can theoretically be explained by five outer points and four corners of the picture. A reader of my blog later suggested that eleven would be a natural answer for someone with a vector graphics background, who would consider that the end point and starting point are different points even though they are physically in

⁴<http://gojko.net/2008/08/29/how-many-points-are-there-in-a-five-point-star/>

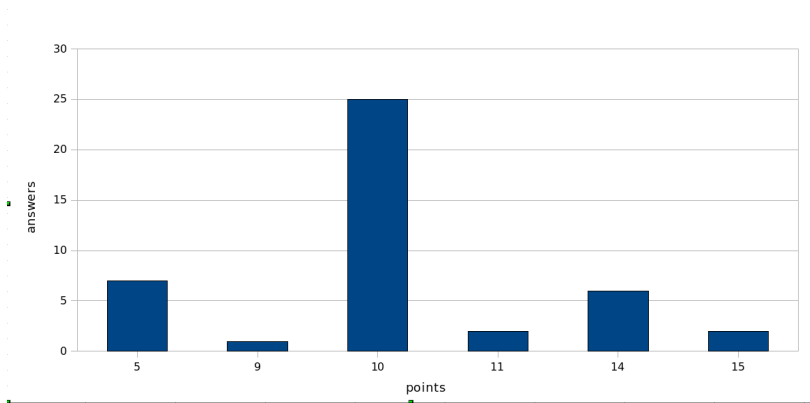
the same place. The two votes for fifteen are still a mystery to me. Bob Clancy suggested that this might be a classic example of the original problem being restated by the individual and then the restated problem solved: if the star is drawn with lines crossing in the inside of the star and then a point counted wherever two lines intersect, people might count the points in the inner pentagon twice (although they are the same as the five inner points).

Figure 1.1. A standard five-pointed star... but how many points does it have?



Some readers are now probably asking themselves what is the right answer. In general, there is no right answer. Or more precisely, all of these answers are correct, depending on what you consider a point. *In software projects, on the other hand, there is a single correct answer in similar situations: the one that the business people thought of.* And for the project to turn out just the way that customers want, this answer has to come up in the heads of developers and testers as well. This is quite a lot of mental alignment. Forty cards are not a sample large enough for statistical relevance, but this experiment has confirmed that even a simple thing such as a familiar image and a straightforward question can be interpreted in many different ways.

Figure 1.2. Results of my poll on points in the star

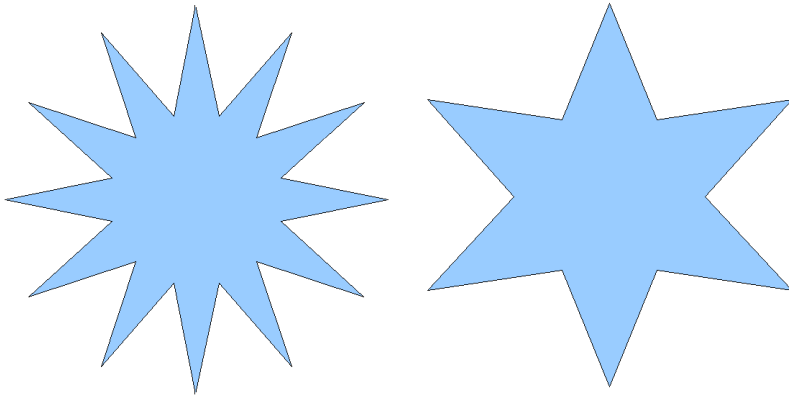


Imagine that you are part of a software project that calculates prices for gold-plating various metal pieces. This may be a contrived example, but let's keep it simple for now. One of the requirements, in its classical imperative form, states that “the system shall let the users enter a diameter and the number of points of a star-shaped metal piece and calculate the price of gold-plating based on materials (total piece surface area using prices in appendix A) and complexity (number of edges using prices in appendix B)”. We have the prices and formulas precisely defined in imaginary appendices and the business analyst has spent a lot of time getting these absolutely clear with the customers, because this is where the money is. Developers should be able to work out the rest easily from the number of points in the star and the diameter and testers should be able to compare the test results to expected results easily. After all, everything is specified precisely and we only have to do a bit of elementary geometry. Right? Well, not exactly.

When requirements finally come to development, things become much more precise because someone actually has to write the code. Bertrand Russell wrote in *The Philosophy of Logical Atomism*[5] that “Everything is vague to a degree you do not realise till you have tried to make it precise”. On most projects even today, writing code is the first time that we try to make the solution really precise. At this point, a developer may have the same understanding of a point as the busi-

ness person making the request, but I would not bet on it. Work out the probabilities from the experiment, and you'll get about a 39% chance for this to happen. A tester will need to verify the result which asks for another mind alignment and brings down the probabilities to 20%. Again, I don't claim that this number is statistically significant and describes a general success ratio, but whatever the precise figures the probability falls exponentially with the number of participants that need to have their minds aligned. Problems like this often don't come to light before development because they are subtle and hidden behind things perceived as more important such as the price of gold-plating per square inch. We think that we don't have to be precise about things that everyone understands during analysis, because it's common sense how to draw a star with a number of points. The rules for the surface area should be clear from basic geometry. Let's disregard all the weird answers and consider just the fact that some people in the experiment counted only outer points, and some counted both outer and inner points. How would you test whether the system works correctly if 12 was given as the number of points? Is the correct star the one on the left or one on the right in Figure 1.3?

Figure 1.3. Which one of these has 12 points?



Things like this, where we feel familiar with the concept and implicitly think that others have the same understanding of it as we do, are one of the core causes of missing the target in software projects.

A small misunderstanding can cost a lot of money

The previous example was invented to demonstrate the point and arguably the misunderstanding is not a big one – the prices will be worked out correctly once everyone agrees on what a star with 12 points looks like. But the devil is in the detail. Wrong assumptions and misunderstandings that we have about the domain might not be huge, but they can still have a huge impact on the software that we deliver. Here is a real world example of how a tiny misunderstanding can cause a lot of pain.

Online poker is big business in UK at the moment, so loads of companies want to grab a piece of the action. Because poker is essentially a group game, potential newcomers are presented with a chicken-and-egg problem. In order for your poker system to be interesting to new players, it has to have lots of other players online. It is very hard to start up a network of players initially. Instead of attempting this, lots of UK bookmakers just make a deal with an existing poker network and then re-brand the software, instantly giving their customers many potential opponents and sharing a piece of the profit of the whole network. UK punters have accounts in British pounds but most poker networks work with US dollars as chips. So the system required a screen that allowed players to convert between UK pounds and US dollars. The requirement for this was to round the converted value to two decimal places, as a poker chip was worth one cent and fractions of cents were not supported. People who have worked on financial systems in the past probably already guess what happened.

This worked fine until one day someone found out how to make money out of it. For the sake of the story, let's say that the exchange rate was 0.54 pounds to a dollar. So 0.01 pounds would convert to 0.02 dollars, but a single cent would convert to a single penny – 0.0054 rounded to two decimals is 0.01. Because of the rounding, you could convert a single penny into two cents, then convert the first cent back

to a penny, then convert the second cent into another penny and end up with twice the money you started with. Yes, it is just one penny more, but the guy who worked this out wrote a script to do the dirty job and apparently took more than ten thousand pounds before the fraud was discovered.

When the news about this broke, business people argued that the amount should have been rounded down and that the developers should have known this, but the developers argued that they received a request to round to two decimals without any specifics. The requirement to round to two decimals sounds obvious and unambiguous, just as did the question about the number of points in the star. In any case, the blame game does not solve the problem. We need to prevent problems like this by ensuring that developers and business analysts have the same understanding of 'rounding to two decimals'. The question needs to be raised before the development starts, not after it is in production, when someone finds a hole in the system.

Fulfilling specifications does not guarantee success

The communication gap on software projects doesn't only cause people to understand things differently. It also causes them to focus on unimportant issues. This is a huge source of problems. I have seen several projects where people wasted enormous amounts of effort building features that they thought were important, but in fact missed the real goals. Working from specifications does not guarantee that the project will deliver the desired business value.

Gilles Mantel organised a workshop entitled *Test-Driven Requirements: beyond tools*⁵ at the Agile 2008 conference held in Toronto in August 2008. At the workshop, four volunteers acted as a customer, a product owner, a developer and a tester for a simple construction project involving children's play bricks and dominoes. (For the sake

⁵<http://testdriveninformation.blogspot.com/2008/08/material-of-tdr-workshop-at-agile-2008.html>

of the story, if you don't know what a product owner is, think of it as a business analyst.) There were two sets of construction bricks and dominoes, placed on two halves of a large table with a large white paper screen blocking the view in between, as shown in Figure 1.4.⁶ A person sitting at one side of the table could not see anything on the other half. The aim of the workshop was to demonstrate how the requirements and testing processes fail when communication is impeded, even with something as simple as a domino construction.

At the start of the workshop, the product owner, the developer and the tester left the room. Mantel put together a simple construction on one half of the table with the customer, aligning the dominoes so that they all fell when a small lead ball was rolled down one of the bricks and bounced against another brick then the first domino. There were lots of other bricks in the construction, but *the primary business goal was to make all the dominoes fall when the ball was rolled.*

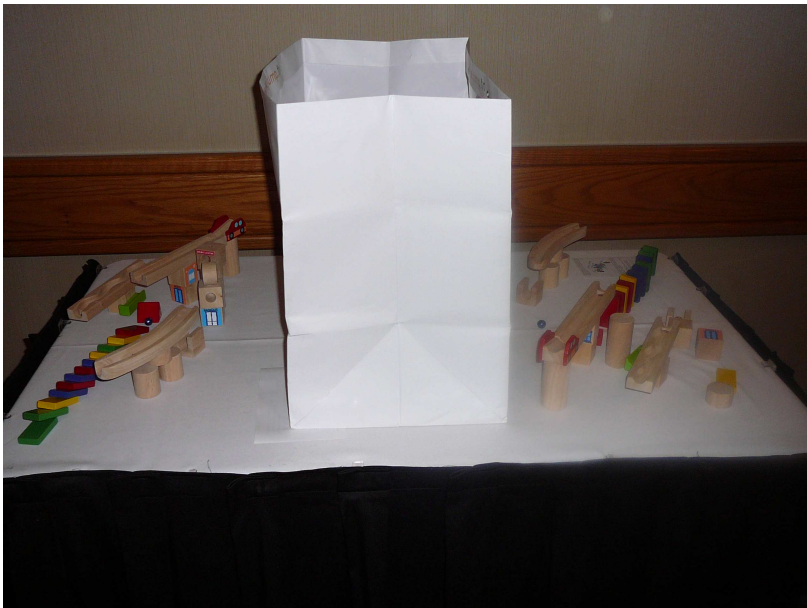
The product owner then walked into the room, looked at the construction and discussed it with the customer. Then the developer and the tester walked into the room. The developer was placed at the other half of the construction table, unable to see the original construction. The tester was placed on a different table, facing the construction table backwards and unable to see what was going on. This was intended to simulate the situation where a product owner acts as a customer representative and testers have no influence over development.

The product owner then started explaining to the developer how to replicate the construction. The developer started building with the bricks on his part of the table and the tester listened in to the conversation. The developer was allowed to ask any questions and the product owner was allowed to explain the construction in any way he saw fit, but the tester was not allowed to ask any questions during the construction. She just listened in and took notes. The product owner led the developer by explaining the shape and relative positions of different building blocks. They discussed at length

⁶The photograph is © Eric Lefevre, used with his permission

differences in colour and number of blocks. When the developer had finished building the system, the tester was asked to validate what the developer had built and approve or reject the release, without looking at the original construction. Based on her notes, the tester found a few issues and refused to approve the release. Gilles Mantel then performed the only test that was really important – he rolled the ball down the first brick. It did not bounce off the second brick and the dominoes did not fall.

Figure 1.4. Domino construction exercise at Agile 2008



The interesting thing was that the product owner never asked the customer about the goal of the construction, so he was not able to communicate anything in this respect. This was not forbidden by the rules of the game, it just never happened. When the exercise ended, the customer said that the ordering of dominoes, alignment of bricks by colour and all the extra bricks on the table were not really important and that he would have accepted the construction if these were different providing the dominoes fell after the ball was rolled.

Unfortunately, the product owner and the developer wasted most of the time discussing exactly these issues and the tester validated the construction based on this discussion.

This exercise demonstrated very clearly what happens with the traditional requirements process, when the important business goals are simply lost in translation and lots of effort is wasted on irrelevant details. The developer had no real chance of fulfilling the business goal because he had no idea what it was.

The tester was just asked to approve or reject the release without really understanding what the project was about, based just on the specification put together by the product owner. So her contribution was left to pure chance as well. She did reject the release, but for the wrong reasons. In this case, it was lucky that the developer did not build the construction exactly as the product owner required, otherwise the tester would have approved the release. The result would have been correct in terms of the specifications, but would have completely missed the business goal.

Describing *how* and *what* but not *why* left the success of the project to pure chance. Instead of producing traditional specifications and requirements, we should really focus more on getting the communication right. Involving developers and testers from the start, communicating business goals to everyone and removing communication obstacles is the way we can take control of projects, and not leave success or failure to pure chance.

Requirements are often already a solution

Classical project requirements focus on *what* but not on *why*. They are effectively a proposed solution to a problem, but do not tell us what the problem actually is. Customers and business people who specify requirements generally have a very superficial view of software, which is hardly surprising as it is natural for them to focus on business

rules and user interfaces and not on the infrastructure to support it. Because business people lack deeper technical understanding of implementation details, their proposed solution is sometimes much more complicated than it needs to be. It is not uncommon for a technical person to suggest a much simpler solution once they know what the problem is.

In 2005, I was involved in building a J2EE-based system for Bluetooth content distribution. It was initially used for pushing small files and text messages to mobile phones. After a few months, the client wanted to add on-demand content pull and distribute large animations and video files. Enabling on-demand content pull required significant system changes, not to mention the fact that a Bluetooth network is not suitable for sending five-megabyte movie files to hundreds of mobile phones at frequent intervals. As always, the client wanted it done as soon as possible, ideally in less than two months. There was no way that such functionality could be implemented and work properly in time. Even if it was possible, such a hack would make future support a nightmare.

We asked the clients to identify the problems they wanted to solve by pulling films to the phones. It turned out that they had an opportunity to sell the system to an art fair, where the visitors would use mobile phones to view a video tour. From this perspective, the video tour had absolutely nothing in common with the original solution, except the idea of software running on mobile phones. We decided to build a small stand-alone application for the mobile devices, which would read all the files from an MMC memory card and would not communicate with any servers at all. This was done in time for the art fair and did not break the architecture of the server.

This example demonstrates how important it is to get technical people involved in the specifications process and to share information about the problems that we are trying to solve, not just the proposed solutions. Traditional ways of defining requirements and specifications only harness the knowledge of a selected few individuals, and don't really use all the brains on the team.

Both traditional and agile development processes expect clients and business experts to specify what the system should do and rely on them to get it right. I do not agree with this. Business experts and clients should definitely be in the driving seat of projects, but I think that much better results can be achieved if developers and testers also have a say in what is to be built.

Cognitive diversity is very important

Getting different people involved in the process of specifying the system is also very important from the aspect of cognitive diversity. When everyone in the group approaches a problem from a similar viewpoint, ideas are reinforced by a kind of echo effect and it becomes really hard to see blind spots. People in homogenous groups often tend to make decisions to minimise conflicts and reach consensus without really challenging or analysing any of the ideas put to discussion. In psychology, this effect is known as *groupthink*.⁷ Groupthink is a serious problem for decision making, not limited to the software world. In *The Wisdom of Crowds*[6], James Surowiecki blamed major US foreign policy mistakes, such as the Bay of Pigs fiasco and the failure to anticipate the attack on Pearl Harbor, on overly enthusiastic plans as a result of Groupthink. Instead of making people wiser, Surowiecki claims that being in homogenous groups can make people dumber (see [6] pp. 341).

Small homogenous groups also exhibit the effects of peer pressure and conformity. Surowiecki quoted an experiment by Solomon Asch, in which he showed groups of people three lines and asked them to identify which line was the same length as a line on a card. Asch lined up people in the group and showed them a range of cards in sequence, asking them to give their answer out loud in the order they were lined up. Unknown to the last person in the line, all the other subjects had been actually told up front which answer to give. For the first few cards, they gave the correct answer and then deliberately started selecting incorrect lines. The last person then often started scrutinising

⁷<http://en.wikipedia.org/wiki/Groupthink>

the picture more, moving around to measure the lines again. About 70% of the subjects changed their answer at least once and one third of the subjects went along with the group at least half the time. Asch repeated the experiment with at least one of the collaborators selecting the correct answer. This immediately encouraged the experiment subjects to say what they really thought and “the rate of conformity plummeted”. This experiment demonstrates the effects of peer pressure and how people are generally reluctant to state their opinion if it is contrary to all the other opinions in the room. It also demonstrates that the situation changes dramatically even when a single different opinion exists.

Surowiecki also states that even small groups of five to ten people can exhibit what he called the wisdom of the crowds, reaching a state where the group together is smarter than any individual in the group. Cognitive diversity and independence of opinion are key factors in achieving this. People should think about the problem from different perspectives and use different approaches and heuristics. They should also be free to offer their own judgments and knowledge rather than just repeating what other people put forward. By getting different people involved in the specification process, we can get the benefits of this effect and produce better specifications.

Breaking The Spirit of Kansas

On 23 February 2008 a B-2 stealth bomber named The Spirit of Kansas took off from Andersen Air Force Base on the island of Guam. By that time, The Spirit of Kansas had more than 5000 flight hours and B-2 bombers were considered highly successful, having taken part in various military campaigns since 1999. Shortly after take-off, the plane started spinning and crashed inexplicably. The aircraft was completely destroyed, with an estimated \$1.4 billions' worth of damage. The two pilots ejected in time and survived the crash with burns. On June 6th 2008, CNN reported⁸ that the bomber crashed because of an issue with control instruments, caused by specific

⁸<http://edition.cnn.com/2008/US/06/06/crash.ap/index.html>

climate conditions over the island of Guam. Moisture caused three out of the plane's 24 sensors to receive distorted data, causing the flight-control system to send an erroneous correction command. The pilots and crew followed the right procedures and CNN quoted Major General Floyd L. Carpenter, who headed the accident investigation board, as saying that “the aircraft actually performed as it was designed. In other words, all the systems were functioning normally”. Carpenter also said that some pilots and maintenance technicians had known about these specific climate issues for at least two years but had failed to communicate them. One of the conclusions of the investigation was that the human factor of failure to communicate critical information was one of the causes of the incident.

Although I was never involved in software crashes of this financial proportion, it was not uncommon to find that someone in the organisation had knowledge that could prevent a problem from happening in the first place, but they failed to communicate it. Facilitating communication throughout the project is a key factor for preventing major failures, but the traditional project model does not encourage this. Testers in particular may be aware of some problems that need to be addressed in development, but they do not get involved in the effort until very late and there is nothing that facilitates the extraction of such key knowledge from them with traditional specifications or requirements. This is not a problem specific to software projects, and lots of organisations struggle with extracting knowledge from their own employees. Lew Platt, a former CEO of Hewlett-Packard, is famously quoted for saying that “If HP knew what HP knows, we would be three times more profitable”.⁹



Stuff to remember

- The traditional specifications and requirements processes now established in the software industry are inadequate and essentially flawed.
- Specifications do not contain enough information for effective development or testing, they are prone to

⁹See *A Measurable Proposal* by Tom Davenport[7].

ambiguity and they only use the knowledge of a selected few individuals.

- Things get lost in translation between customers' original problems and development.
- Teams that do use agile development practices are affected by this problem less than those that do not use agile development practices, but they still suffer from it.
- Unless everyone involved understands the business goals, there is a high risk of missing the target.
- Developers and testers may skip or misinterpret parts of the specifications, and there is no easy way to check which parts are actually affected.
- Requirements are often already a proposed solution to the problem, and do not explain why something is required.
- Gaps and inconsistencies in traditional specifications get discovered only when developers start writing code.
- Change requests may introduce inconsistencies into the software, and there is no easy way to spot problematic rules and affected parts of the system.
- Specification documents often do not reflect the true state of implemented software.
- Customers and business analysts are expected to get the requirements and specifications right and the knowledge of developers and testers is not taken into account.
- The cumulative effects of small misunderstandings cause huge problems, but there is nothing to help us discover and resolve these issues before implementation.
- Matching what clients expect is essentially a communication problem, not a technical one.
- There are some things that are so obvious to the customer that she will never tell you them unless you ask.
