Holding Our Feet to the Fire

Clarrus Consulting Group Inc. 7770 Elford Avenue Burnaby, BC Canada V3N 4B7 **T** +1 (604) 540-6718 www.clarrus.com

Abstract

In recent times, there have been several movements in software development that appear to suggest that we can defer or eliminate many practices that apparently slow down our ability to develop our products.

As we have all seen in this industry, though, as movements such as Agile or Lean gain popularity, there is much that is lost in the translation to the masses, and these innocuous statements become embraced a little too tightly. There are claims of universality from all corners of the methodology debate, and that ‘barely sufficient methodology' often becomes ‘insufficient’, with disappointing or disastrous results.

In this presentation, Jim suggests that a balance is required. There are some practices that need to be considered for any project, as the cost of deferring or ignoring them becomes extremely costly to the organization, and results in less delivered value. Balancing appropriate weighting of these practices with appropriate management of change becomes the optimal way of driving a project to successful completion.

Jim describes different types of practices to be considered, approaches to recognize reasonable application along the way, and identifies the practices that we absolutely need to move forward in the lifecycle to ensure that success means creation of the value we actually intended to deliver.

Holding Our Feet to the Fire

In recent times, there have been several movements in software development that appear to suggest that we can defer or eliminate many practices that apparently slow down our ability to develop our products. For example:

- Jim Highsmith has advocated a 'barely sufficient methodology' [1],
- Ken Schwaber and Mike Beedle suggest that 'Teams are put in a time box and told to create product', and assert that 'keeping models synchronized with code is a maintenance burden in any organization' [2].

As we have all seen in this industry, though, as movements such as Agile or Lean gain popularity, there is much that is lost in the translation to the masses, and these innocuous statements become embraced a little too tightly. There are claims of universality from all corners of the methodology debate [3], and 'barely sufficient methodology' often becomes 'insufficient' [4], with disappointing or disastrous results.

A balance is required. There are some practices that need to be considered for any project, as the cost of deferring or ignoring them becomes extremely costly to the organization, and results in less delivered value. Balancing appropriate weighting of these practices with appropriate management of change becomes the optimal way of driving a project to successful completion. The trick, of course, is to understand when to make those decisions.

When are we better served to hold our feet to the fire?

There are a couple of areas that lend themselves to conscious decisions early on, regardless of the project lifecycle, the size and shape of the team, or the relationship with the customer. While mature teams with depth of experience might apparently be able to breeze through these decisions quickly, it is still critical to ask ourselves if we have really nailed these elements before we proceed on any new project.

No decision or shared understanding should ever be considered cast in stone, regardless of when it is determined in the project lifecycle. Early agreements need to be seen as our best understanding at the time, and future events always involve the possibility of invalidating an early assumption. While some would argue this means deferral is better, these decisions described here, made early, serve to reduce the risk of conflict downstream by serving to align the stakeholders in the same direction.

Overall, there are five core elements that should be consciously considered on any project. While in many mature teams this consideration may be straightforward, to ignore these elements in the name of quickly creating product can have devastating results:

---

[1] Jim Highsmith, Agile Software Development Ecosystems, Addison-Wesley, 2002

[2] Ken Schwaber, Mike Beedle, Agile Software Development with Scrum, Prentice Hall, 2001

[3] Barry Boehm, Richard Turner, Balancing Agility and Discipline, Addison-Wesley, 2004

[4] Alistair Cockburn, Agile Software Development, Addison-Wesley, 2002

Intentional management of the team dynamics,

Alignment of the group with a shared vision,

Explicitly identifying the value to be delivered on the project,

Quantifying the quality of the product to be built, and

Analyzing the system as extensively as possible.

The first element centers on how the people involved in the project will work together. In every group I work with, when asked about the characteristics of their best project experience, the vast majority of the responses center on relationships: good, positive communication, respect and trust are always evident.

While some teams may have the luck or serendipity to start out with strong relationships, most are well served to take the time to build and reinforce strong relationships between one another. Even for those that start out fine, the fact that push will usually become shove on projects means that the apparently strong relationships can quickly erode when the pressure rises. Team members need to understand each other's goals, needs and motivations, need to agree on mechanisms for resolving disagreements respectfully, need to understand how to best relate to one another.

The remainder of these elements critical to project success aligns the team to ensure everyone is working on the same project. A shared vision for the project, clear completion criteria in terms of value to be delivered to the customer and expression of scope that involves a quantified quality definition are critical before moving forward - think of this as an agreement on where the finish line actually is.

Note that this does not necessarily mean a comprehensive, exhaustive definition of scope. For those projects where there will be plenty of discovery downstream, we should still strive to identify where in the project we will be making these deferred decisions.

There is value in confirming our assumptions about what we believe we all know, and we need to know what we don't yet know.

Consciously Build Effective Teams

In working with a wide variety of teams in both industry and academia, there is often discussion of what needs to be captured in a persistent form. There is always resistance to documentation of any kind on projects, but there is one document that turns out to be the most relevant to the success for many teams. It is their team agreement.

For reasons that are not surprising, many people see team agreements as a waste of time, a useless document that is full of flowery language, and can be dismissed along with many vision statements that are found in lobbies around the world. Indeed, for some teams I work with, this viewpoint is not far from the truth. It is seen as an interesting diversion and generates some fun discussion about personalities, but it is not seen as the real work to do: the assignments due to be graded.

From my experience, though, there is a clear correlation between a respect for their team agreement and how well teams perform on their projects.

We use the Tuckman model[5] for team development, which many people recognize as the Forming/Storming/Norming/Performing model. Usually, the teams progress through the team agreement with a false sense of security that they have

[5] Bruce Tuckman, Developmental Sequence in Small Groups, 1965

quickly and painlessly managed to get to the performing stage as a team. That dreaded storming stage never seemed to appear at all.

What has actually happened, in most cases, is that the team has mistaken this initial team agreement as final, and any underlying issues tend to stay there, lurking, often brought to the surface later in the project.

This raises a couple of issues that teams need to address. It is clearly insufficient to hack together a team agreement and think the job is done. No team can assume they are faring well, that they are a well-performing team, until they have managed to get through a phase of conflict unscathed. Until then, and possibly afterwards as well, there remains a significant risk that the team can collapse with conflict.

If the team agreement is intended to help a team manage through crisis, there are a couple of things that should be discussed and agreed upon that are not often dealt with.

Feedback needs to be open and honest, of course, and team members need to be able to serve this up in a face-to-face manner. This feedback should include the strengths as well as the challenges, which should include some discussion of how these may be resolved. In other words, we need to avoid purely negative feedback, it needs to be provided with a goal of strengthening the team overall.

The discussions around the team agreement should include how to give reasonable feedback, but how to take feedback as well. Remember, the team agreement is simply the minutes of the important teaming discussions that you have had, not the production of a document. When we receive feedback, we need to see it as merely a data point. We can choose to be offended by this data, we can choose to 'shoot the messenger', or we can try to find the kernel within the data that supports improved relationships.

These discussions are never over at the beginning of the project when we have produced that document. In fact, if you think you have finished with your team agreement by producing a document rather than having the deep discussions, be prepared to deal with conflict sometime in the future, when your team really does run into trouble. This advice holds true for any document you produce, of course.

Work to a Compelling Vision

A critical component to sustained team success is finding a way of keeping everyone jazzed.

We all hear about amazing places to work, and Google comes up in many of these conversations. They have flexible work conditions, great perks, even one day a week where employees focus on whatever they want to, which has been the fruitful source of many of their products or research areas. Done right, everyone wins, and Google's success is at least partially a testament to this approach.

If we miss one component of what has been successful for others, though, things won't work out as we had hoped. In one shop I have worked with, they had set up what seemed to be a great work environment – totally flexible work hours, a comfortable workplace in one of the city's trendiest areas. They even provided the incentive to the troops to be creative and come up with great new ideas.

What happened was the opposite of what was intended. While not everyone took personal advantage of the flexibility, there was enough of a drop in energy that the net effect from the group was disappointing. Product wasn't coming through as expected. Few of those new ideas materialized. In the end, it turns out a couple of people were let go.

What went wrong? If we take a look at what Google stands for, there are a couple of statements that may be relevant. Buried deep in their Philosophy is the statement "Give the proper tools to a group of people who like to make a

difference, and they will." While it sounds like this group did what they could to provide the proper tools and environment, the desired result didn't materialize. Granted, their pockets are not quite as deep as Google's, but could the challenge be in that phrase "who like to make a difference"?

Stop right there. I'm not suggesting that they were a bunch of deadbeat employees that don't like to make a difference. Intrinsically, we all want to make a difference in what we do, but we all also go through stages of lower energy.

More accurately, alongside the great work environment, there needs to be a fresh, exciting raison d'être, something that motivates and inspires us to greatness. The second relevant statement from Google's site is their mission statement itself: "to organize the world's information and make it universally accessible and useful".

Few companies have such a compelling vision. While I don't think we can all have such strong visions for our existence, we need to develop the best vision we can, and it needs to be something that the entire team buys into. It has to be something that jazzes everyone in the group, something that inspires them to make a difference. Whether it is inspiration to defeat a common enemy, or altruistic and world-saving, there should be no question about what you are all trying to achieve.

A good workplace is important, to be sure, but the motivation needs to be more than a request to get your work done. A strong vision can truly jazz a team, it is the difference between the workplace being a job and a joy.

Identify the Value to be Delivered

It can be difficult enough to get a project team to focus on delivery of value when we are starting a project; it is all that much tougher to remain focused on this prize as the project plays out. One of the main reasons for this is that the tools we use to manage projects tend to divert our focus elsewhere.

It is easy and commonplace to frame a project in terms of cost and schedule. These are two dimensions that we are all familiar with, and the traditional emphasis of these on projects reinforces these often shortsighted habits. While these are important, they pale in significance to the critical dimension of delivered value.

Yes, we want to complete the project on time and not lose our shirts doing so, and on some projects, these dimensions can be absolutely rigid. If we haven't made the stakeholders' world better in some way, if we have not delivered value, we have not done our job.

In working with project teams, setting expectations for value to be delivered can be one of the most difficult things we do. We need to do to so as clearly and precisely as possible, which unfortunately requires more thinking and collaboration than most teams are accustomed to at this early stage.

We need to define value in terms that are relevant to stakeholders. Are we intending to reduce their costs by x% in the next 12 months? To increase market share or improve their quality of outputs or reduce their risk? We need to agree on expectations of how their lives will be better, and we need to be specific, measurable, and time based. All these objectives need to be relevant to the stakeholder needs, and they need to be attainable within our cost and time constraints.

This expressed value to be delivered is a precursor to capturing the user stories or traditional requirements of our systems, and serves as a basis to determine whether any of these user stories or requirements even make sense as part of the project: both initially, and for those proposed downstream.

The weaker we are at crisply defining expected value, the easier it is to fall astray as the project progresses. More importantly, the easier it is to artificially declare project success at the end of the day: "I managed to spend all your

money in the time allotted!" can be all the good news we can muster on a project with a poorly structured expression of delivered value.

Even Earned Value analysis really has nothing to do with delivering value: it would be more aptly named Spent Budget analysis, as an entire project can progress very well by these standard reports, but fail to provide any value. The 'value' in Earned Value, is assumed to have been included in the definition of the activities to be performed, but is not actually measured: we accrue costs and effort expended, value is inferred. The same argument holds true for user stories.

We need to define value up front, to be sure, but we also need to remember that whenever we are thinking about progress, usually in terms of cost and time, we need to continuously ask ourselves: are we making progress towards our goal of providing value?

This is true for all of the activities we originally envisioned when we put our original schedule together, but it also remains important when we address change. Indeed, this becomes an important differentiator that allows us to manage change effectively: if a proposed change does nothing to maintain or improve the value delivered to the stakeholders, we have a strong argument for not including it in the project.

This alone is more than enough reason to invest the time to set expectations about delivered value on a project. The cost of the effort to do so will usually be more than made up if we can use it to reject a single proposed change that would otherwise diminish the value we deliver.

Quantifiably Specify Quality Needs

For all the companies I have worked with as an employee or as a consultant, whether they suggest they are agile or not, there have always been issues in the area of understanding the scope of work.

One of the recurring areas where many teams fail to specify the needs of what they are about to build or buy, is the area of quality. Everyone I have worked with suggests that quality is of utmost importance, yet most fail to precisely quantify what quality means in the context of their product.

There are many reasons for this deficiency. Some take the term quality to align with qualitative, and therefore don't grasp that these can and should be expressed in a quantified fashion. Unfortunately, suggesting that a system needs to be fast or user-friendly or secure just doesn't cut it when we are trying to test against these terms.

Others struggle to make the leap from the taxonomies that generally cover the landscape of quality, but are not easily expressible in quantified terms. It can take a great deal of time in the school of hard knocks to recognize that user-friendly needs to be expressed as a combination of a wide range of quantifiable ideas: acceptable response times and feedback, adherence to GUI standards, consistent error-handling mechanisms and a host of others.

Again, most teams don't do a good job in this area, and the results are often either accidental adequacy, or more often a trip back to the drawing board.

Within agile teams, there is a notion of an architectural spike: analyzing just enough of the necessary architecture of the system that will allow you to progress forward with the current set of stories so that the risk of major refactoring is minimized. This is a powerful concept that can work very well for the functional considerations of the system being built.

I would suggest that there is a sister spike that should take place around the same time: a quality spike can give us just enough insight to ensure that we are building to meet the quality needs of our system.

Indeed, if we perform a reasonable quality analysis at this point in the game, agile practices actually put us ahead of the curve when compared to traditional approaches. If maintainability is a quality attribute that is of interest to us, then the principles of a shared code base, ongoing refactoring and frequent releases play to our favor. These practices can be expressed as requirements that support our building a system that is maintainable.

Similarly, test-driven development and a unit test infrastructure support the testability of the system, as does the practice of frequent releases. Refactoring supports reusability, the list goes on. For those quality attributes that are of interest to the development organization, agile practices serve to inherently produce a higher quality system.

For those quality attributes that are of interest to the end-user, though, there is value in performing that analysis as a quality spike. As these attributes are strongly domain and application dependent, and will vary in importance dramatically, we cannot simply rest on our development practices to ensure that we meet these quality needs.

We need to sit down with our customer and decide which attributes are most important, or even relevant, and translate these into a series of measurable characteristics that we can then build to as we flesh out our stories. We do the best we can at this time, remembering that even if it is a spike, it can generate tremendous value.

Many of these attributes - such as security, usability, interoperability, and others - weave throughout all of the stories we are building, and consciously articulating these alongside our architectural spike gives us a much higher likelihood of efficiently meeting our goals. Indeed, some of these may be critical enough to drive stories of their own, stories we might otherwise only consider much later in the game.

Reasonably Analyze the System Early

Generally, when we talk about software testing, this is seen as the dynamic execution of the software or system and peering into the code, rather than any other form of validating or verifying the system. Not static analysis of code, not peer reviews or inspections.

This trend of greater emphasis on this type of software testing is not good for the industry, and the fact that most companies I have worked with do not make a distinction between QA and testing compounds the problem.

Don't get me wrong; being effective at testing software products is not a bad thing. Indeed, given the way most companies develop software, we had better be good at finding all the bugs in our products. The challenge here is not testing; it is the emphasis on exhaustive testing over more efficient analysis techniques earlier in the lifecycle.

Let's pull out some stats from the industry. Numerous studies show that half or more of all defects found on software projects have their root in a requirements issue. One study at TRW [6] indicated 54% of all errors were found after unit testing, and 83% of these were requirements and design-based errors. Think critically about the defects that have been logged for your products, I would expect you to see a similar trend.

Layer on top of that a study with the US Navy [7] that indicated that incorrect facts accounted for 49% of requirements errors, and an additional 31% were errors of omissions. Having worked on defense projects as well as a wide range of projects in other sectors, I'd say that it is safe to assume that those 'errors of omission' could easily double in most shops.

---

[6] TRW study as reported in Alan M. Davis, Software Requirements: Objects, Functions and States

[7] The US Navy's A-7E project, as reported in Alan M. Davis, Software Requirements: Objects, Functions and States

One more data point that has been reflected in numerous studies, including one from Hewlett-Packard [8]: a requirements-based flaw can cost over 100x to fix after the system has been deployed, compared to the cost of fixing the problem closer to when it was injected into the system. The problem is, many shops first validate their systems only when the code is running.

The point is this. Throw all these data points together, and it is pretty clear that we spend way too much time cleaning up the spills that never should have happened in the first place.

I see a trend developing that more organizations are leaning on testing to actually figure out what their requirements should have been. In the recent past, the following points came out in discussion with various companies:

One shop identified numerous defects for non-typical issues around data that should have been considered as alternatives and exceptions in their use-case analysis.

In the same place, a tax lawyer indicated that she gets called into projects when a problem has been discovered, rather than identifying the relevant tax-related business rules up front and developing the system to accommodate them accordingly.

Another shop found a particularly nasty bug where one data element was implemented in slightly different ways in different locations throughout the system. A data dictionary might have come in handy here.

Then there is the company that only realized when integrating the whole system together that the system took minutes for what should have been a very fast response time. And the company that arrived at the integration stage to find they don't know how to test what they had built in the first place. Oops, maybe an analysis of quality attributes would have set reasonable expectations for the design.

It would be easy to come up with other examples for quite a while, and you would likely have stories to contribute as well. All of these issues identified above were discovered through dynamically executing the code, and all issues were fixed in the code. Until these points were raised in the context of requirements discussions, the leap had not even taken place that all of these were actually requirements-based defects in the first place.

It sadly reminds me of an old Calvin and Hobbes cartoon, where Calvin asks his dad how they determine the load limit on bridges. His dad replies that "they drive heavier and heavier trucks across the bridge, and when the bridge breaks, they weigh the truck, and that's the limit!" Absurd, but that's effectively the approach in many software shops today.

It is easy to see why testing is leaned on so heavily in the industry. Even the most novice person on the team can be seen as productive by poking at applications and quickly find issues with the system. Testing has traditionally been the point where we get the bugs out of the system, and it is the teams and individuals that magically pull a bug-ridden application to a state where it can be shipped that are often seen as heroes. In some ways, test-driven development takes this to the extreme: tests are crafted in advance of the code.

Isn't that what those requirements and design stages we rush through are supposed to accomplish? While I respect that good testing is a mature discipline and a skilled tester can wring out many of the most elusive defects, the point remains that everyone is better off if the defects weren't injected into the system to begin with.

---

[8] Robert Grady, "Applications of Software Measurement Conference," 1999

The industry as a whole has far greater opportunities available in getting better at analysis, not in more testing or even more effective testing. We need to figure out our equivalent to those bridge load limits in advance, and build systems that meet these specs.

Good, effective analysis is a matter of selecting which techniques to use that will give us the most valuable perspectives of what we are trying to build. Use cases alone are insufficient, as are 3×5 index cards. Having a customer embedded with the team gives us timely opinions and answers, but does nothing to make us more effective at understanding whether we have appropriately analyzed the problem.

We need to pick the right techniques, as each distinct view peels another layer off the problem, and if sequenced correctly, no step is difficult on its own. The challenge is in picking those appropriate techniques, and very few groups consciously choose how they will attack the analysis challenge for each project. Almost all standardized approaches lean towards specific analysis techniques or deferral of these important decisions downstream, rather than ensuring the team is equipped with a wide range of techniques required to do effective analysis, and the judgment required to select the right ones for the problem at hand.

Actually, this higher level strategizing is a technique that effective testers do well, but in the context of exercising the product that has been built rather than proactively developing the product correctly in the first place. The skills and attitudes that have already been developed in effective testers are the same required for effective analysts, except that this need is in an under appreciated phase of building our products.

Make the shift to deciding what you need to build and how you will build it rather than discovering the omissions after the fact. You will need far fewer test resources or use them more efficiently and strategically earlier in the project, you will reduce risk and uncertainty on your projects, and you will ship better products faster.

Dangers in Taking Short-cuts

When you get right down to it, we spend a lot of time every day making decisions. From the clear decisions of how we are going to spend our time, whether we are going to do this or that, to decisions that are not as consciously performed, like how we'll make that first coffee in the morning or choosing our route home in the evening.

On projects, the decisions we make on a daily basis determine the outcome of the project. Some have more impact than others, of course, but like a butterfly flapping its wings in the Amazon, many of our decisions have far greater impact than we might think.

Even more than those small decisions we do make, though, it is the decisions we often neglect to make on our projects that can have the greatest impact at all. As we all know, it's unlikely that this impact will be positive.

Many of the decisions we don't make fall under the category of assumptions.

When I set a group loose on an exercise during training, I rarely get a question about my instructions, even after I raise this very point. Far more often, though, when it is time to debrief the exercise results, many of the questions I have to field are actually about details that weren't even understood up front. The group has forged ahead with implicit assumptions.

The irony that these people have forged ahead without first clarifying the requirements is not lost on me.

We all go through life, everyday, supported by a large number of assumptions. We develop routines, standard ways of doing things to reduce the number of decisions we need to make. While chances are that these decisions were appropriate as a basis for our routines when they were formed, there's a good chance we are running on a bunch of

decisions that are now, over time, seriously flawed. It wouldn't make sense to always question everything we do, but at least once in a while, we should step back and consider if our assumptions are still valid.

On projects, one type of decision we don't make are the ones where we simply choose the first good looking option that comes to mind.

We take the first words out of our end-user's mouth as gospel rather than analyzing whether this is indeed an appropriate requirement. Our system design is usually not the one chosen after comparing a range of potential options and selecting the best, but more likely the first one that pops into our head. I would guess that few of us have given much thought as to whether a bubble sort or a selection sort would be more appropriate in a given situation after we had to answer that question on an assignment back in school.

We look back on these decisions and lament "why hadn't I thought of that?" It is almost always better to select from a range of possibilities that we have generated than to just go with the first idea we get that might work.

Another type of non-decision occurs when we delay making a decision until we have run out of options, and there is only one way left to deal with an issue. This is often forced on us during projects as we discover late in the project issues that were injected in the early stages, but have been lying hidden, typically because we haven't hunted them down.

If we find a requirements flaw early, we have a wide range of alternatives for solving the problem. When we catch the same problem a week before we ship, it's far too late to adjust the architecture to address the issue. Deferring decisions to the last possible moment is not an effective time management strategy.

Finally, there are those decisions that are made without the right people involved in the process. Requirements or User Stories written by an analyst on his own, without consulting the developers for feasibility, the testers for testability, or sometimes even the customer for…well, the information simply cannot address all those different perspectives if written in a vacuum. The apparent cost savings of not involving the right people becomes far more expensive as these people discover the flawed assumptions.

Flawed assumptions, taking the first thing that will work, running out of options by deciding too late, and not involving the right people in the decision-making process: each approach can be devastating to your project, and we are all guilty of these on occasion.

Building a Structure for Safe Deferral

If we consciously focus on effective team dynamics and a strong shared vision, and clearly identify the intended value to be delivered and use this to build the clearest possible understanding of the structure of the desired product, we get the best of both worlds: strong team coordination and the agility to provide the best possible solution to the customer. We develop an infrastructure that allows us to efficiently and effectively defer other decisions to that 'last possible moment'.

By reasonably holding our feet to the fire, by making the right decisions earlier in the lifecycle, we end up with toasty little toes. If we defer or avoid these decisions, we often find that the fire is still there, and there is a good chance we'll get burned.

This paper was originally published at the Pacific Northwest Software Quality Conference, Portland, Oregon, 2009

About Jim Brosseau

Jim has been in the software industry since 1980, in a variety of roles and responsibilities. He has worked in the QA role, and has acted as team lead, project manager, and director. Jim has wide experience project management, software estimation, quality assurance and testing, peer reviews, and requirements management. He has provided training and mentoring in all these areas, both publicly and onsite, with clients on three continents.

He has published numerous technical articles, and has presented at major conferences and local professional associations. His first book, Software Teamwork: Taking Ownership for Success was published in 2007 by Addison-Wesley Professional. Jim lives with his wife and two children in Vancouver.

About Clarrus

It is often the same class of challenges that are the root cause of most business inefficiencies. A key aspect of these challenges is that they manifest themselves differently in every organization. That's where we come in.

Clarrus is a down-to-earth, value-driven company with a focus on improving project teams and the projects they work on, across all industries.

Clarrus is unique in that it understands the interplay of human dynamics and project mechanics. Our consulting services and workshops bring a practical and proven approach to increasing the effectiveness and satisfaction of your project teams and delivering results.

Clarrus believes in the principle of teaching people to fish. Our approach nets its richest results in complex environments where multi-disciplinary teams bring diverse perspectives to the table. But, we also help small teams thrive, too. In a nutshell:

- We harness the best of best project practices and apply only what's needed for tangible short- and long-term results.
- We create an environment of trust so our sometimes tough questions can open the doors to peak performance.
- We provide inventive and practical ways for your teams to approach challenges.
- We secure the engagement of your project teams by focusing on issues where the impact is greatest.

Contact us today.

- On the web at http://www.clarrus.com,
- By e-mail at info@clarrus.com, or
- By phone at +1 (604) 540-6718.

January 1, 2010 - Version 1.0