# The Far Away BA: Business Analysis Tips Tor Pulling It All Together Without Being There At All

**Jourdan Arenson**

arensonj@comcast.net

## Abstract

For Business Analysts who help build in-house IT systems at mega-global corporations, system requirements are usually not simply there for the "gathering." Often the Business Analyst (BA) has to lead an expedition of requirement discovery across the continents to drive out the needs of stakeholders in Europe, Asia and the Americas.

If the organization has a chaotic history of mergers and hasty system integration, it is likely business partners and users are overworked and overwhelmed by the complexity of their own systems. The enhancement list may include more projects than the Dev team can possibly look at, much less build. Yet the BA may need to track enough details on all projects in the pipeline, so the Dev team can get working on any one.

In this fragmented, distracted corporate environment somebody needs to organize the vision, keep the focus, root out the issues, and drive the design forward. How can the BA provide this kind leadership without any official management authority? How can the BA stay top of all the dynamic complexity? How can the BA guide a distributed team toward the consensus and collaboration needed for successful, efficient software engineering?

This paper gives observations from a Business Analyst who has faced all these challenges, with a couple of twists not typical of BAs at large corporations: He's worked from home full time for the past 12 years. And, for the past three years, he's worked in an Agile Dev team.

## Biography

*Before getting into IT, Jourdan worked as a journalist and marketing copywriter. In 1995, he switched to tech writing ("because you never need to lie in tech writing") and got a job documenting foreign exchange trading systems at Bank of America in San Francisco.*

*After taking a course in Unified Modeling Language, he decided to become a Business Analyst. And after a couple mergers where his business partners and Dev teams were broken up and distributed around the globe, he was able to convince his managers that he could move to Eugene, Oregon and work from home as a Business Analyst.*

*He's spent the past 12 years on conference calls helping colleagues in the global market centers of Chicago, New York, Charlotte, London and Bangalore build the Bank's web-based foreign exchange trading applications. During those calls, nobody else suspected he was looking out the window at the chickens in his backyard.*

# Why I Like Engineering

I like the term "software *engineering.*" I think engineering is a wondrous human ability. Engineering combines our talents of social collaboration and technical ingenuity. Humans are good at working together to design and build complex systems to meet our goals.

In an engineering project, people work together to build a system designed to achieve a stated purpose. The system could be a spacecraft, a computer chip, a bridge, or application software for a global corporation. The complexity exists on many levels: requirements management, business algorithms, technical design, logistical coordination, and much more. In the final system, all these diverse elements must function together to meet the stated need.

As a Business Analyst (BA) in software engineering I am on a team that builds systems to meet business goals, but as the BA, I don't have to build the systems, and I don't have to come up with the goals. Instead, I support the Business Team (Biz)—who decides what needs to be built—and I guide the Development Team (Dev)—who decides how to build it. My responsibility is to help the teams articulate and agree on what we are going to build.

Software engineering is about managing complexity and collaboration. As the BA, I want to: 1) make the collaboration as efficient and pleasant as possible, and 2) make the required complexity as clear and simple as possible.

While I work on technically complex systems, managing collaboration is the higher priority for me. This is because I work at a large global corporation where my team members are spread out all over the world. Because team members are not physically together, it can be hard to build good relationships. With my unique role between the two sides, I have the opportunity and duty to foster collaboration.

In this paper, I would like to share some Business Analysis tips I've found on managing the complexity and collaboration of software engineering projects.

# Engineering Consensus

Since engineering is about collaboration, how smoothly a software engineering project goes largely depends on the quality of relationships between the teams. When team members are not together in one location, it is easy for relationships to be weak. It is easy to mistrust others. It is easy to avoid problems, point fingers and be defensive.

When the Biz-Dev relationship is poor, there is usually some power differential going on. If the business has the upper hand, the BA and Project Manager (PM) may be openly aggressive, intimidating Dev into agreeing to deliver more software, faster than they can handle.

Other times, the Dev, BA and PM may be passively aggressive, forcing the business to follow excessive bureaucratic process which slows down the rate at which they deliver software.

I have experienced both and I have found that the worst thing I could do was this:

1. Work exclusively with Biz to define all the system requirements in all their complexity.
2. Write it up in a big document.
3. Set up a meeting with Dev and show them the document.
4. Say "Build this."

Whenever I've done that, the Biz-Dev relationship took a dive toward one of those extremes.

The Agile mindset definitely helps, but prior to moving to Agile I followed a principle during requirements gathering that always seemed to result in better relations between the Biz and Dev. The principle is:

**Build consensus step by step, moving from the general to the specific.**

Let me illustrate this in reverse terms. When I described the worst thing I could do, it was going from the general to specific all on my own, *without* building consensus. I would have all the system requirements in my head or on paper. Even if I had it right—which I usually did not—others would challenge my model, if only because they didn't have input into its evolution. I discovered that it is very difficult to get buy-in on "fully baked" requirements.

The best way to get buy-in is to first establish it at a general level, and then establish it again and again, at each more detailed level. There is often a bit more running around and chasing people down—but it is less time consuming than trying to sell fully-baked requirements because people participate in the evolution of the vision of the final result.

It starts with building consensus on the "general." Let's say we're building a new system (as opposed to enhancing an existing one). Building consensus on the general means getting the Biz and Dev to agree on what is the general scope of the initial release.

Note that this is *my* first goal: the general scope of the *initial* release. It may not be the first goal of others. Often the business wants to communicate a grand vision for the system. So I don't press too hard at first. I document a high-level features list, adding whatever is proposed, but in the back of my mind I'm thinking: "What are we going to build first?"

Eventually I start to press the point and say something like: "While it is good to discuss everything we eventually want to build one day, if we don't agree on what the Dev team should build first…they won't build anything."

Here I take the feature list and use that as the basis for a Scope Document. I keep it pretty high-level: a listing of screens, features, interfaces to outside system, workflow, and user roles. I want to make sure I have every major feature and dependency on the table and up for discussion.

Dev is going to use this to decide what they are comfortable taking on for the first or next release. Biz can use it to determine if the proposed set of features meets their business goals.

Once I have a comprehensive list of features for a release, I press for consensus at this more general level *before* moving to specifics. I want to hear Biz and Dev agree: "Yes, this is what we want to put our arms around in our release."

I don't want to be too rigid in this, because there are invariably features that Biz would like in the next release that will not be built in time. So I always have sections where I note features that are "nice to have for first release" or "to be prioritized for future release." I try my hardest to never say "no" to Biz. Instead, I want to say "yes, for a future release."

Eventually I get consensus at the general level, and move down to build consensus on specifics. I try to start with the feature that has the most risk of being more complicated than everybody thinks. It could be due to missed or poorly defined requirements. Or it could be something that the Biz wants from the system that might be difficult to implement.

In this, I'm looking for features that may impact the "scope consensus" we've established. This is how I try to stay in front of scope creep. It's not that scope creep is always bad, it is often necessary to expand scope to address some unexposed complexity that you *have* to account for. But I try to be the first one who spots it, so I can proactively manage it as pleasantly as possible.

Let's say I uncover a feature with newly exposed complexity. The fact that we previously had consensus on scope helps, because I can loop back to that. I say something like, "Gosh this is not what we had in

mind when we agreed on scope. Let's go back and re-confirm our consensus on the scope." The memory that we once agreed seems to make it easier to negotiate scope creep.

In addition to managing scope creep, the Scope Document helps when the team is lost in the weeds of an individual feature. In discussing an individual feature, we may lose track of what the system is trying to accomplish. Going back up to a more general level helps set the context for what details are the most important.

The Scope Doc is also a tool for expectation management. In many cases, business folk don't have a sense of the amount of time and effort needed to build a system, so it can be hard for them to have realistic expectations. This document establishes a fixed point that everybody has agreed will be a useful and realistic accomplishment. If the Dev meets or surpasses what is outlined in the Scope Doc, the business will have their expectations met.

"Build consensus step by step, moving from the general to the specific" works in other situations. It is a great way to organize a meeting agenda and it is a good idea to have in the back of your mind as you run a meeting. I admit it is a fairly obvious, common sense principle, but I have found that if I use this principle when I get rushed or flustered it makes it clearer where and how I should be guiding the team.

# Advocate for the Business Team

As mentioned, poor Biz/Dev relationships stem from a power differential. When Biz has the upper hand, Dev can be forced to take on more than it can handle. And sometimes Dev can use unnecessarily complex processes to fight back. It is the role of the BA to advocate for each side. Let's begin with the role of advocate for Biz.

The best thing I can do for my business partners is to elicit reasonable, buildable, coherent, comprehensive, system requirements that will efficiently meet their business needs.

- "Buildable" means that what they are asking for has a reasonable chance of being built, given the resources and the present condition of the system they are enhancing.
- "Coherent" means that there is a logical consistency to the requirements. I want to show that the system can work on paper, and there are no logic flaws, contradictions or conflicting requirements.
- "Comprehensive" means that we are asking for everything that needs to be done. This is often the hard part. Biz can easily give requirements for that sweet feature that they want, but they forget the boring supporting features that are needed to achieve them such as, user administration or static data entry functions.
- "Efficiently meeting business needs" means that we specify a system that meets their needs and can be built in the most cost effective way: not too much and not too little.

If it all comes together well, I want to help Biz present the Dev team with elegant requirements. We want the Dev team to look at the requirements and think: "That's cool, we want to build this."

I try to do this with all business partners, and over the years I have had great ones and I have had terrible ones. The good ones know their systems, understand the level of effort for enhancements and are reasonable negotiators. My business partner over the last couple years has been the best, and I am very grateful.

Not everything is perfect, however. We sometimes have "off-site" requirement conferences where lots of stakeholders gather in a conference room and lock themselves in for multi-day requirements sessions. Even if all the folks are smart and reasonable, people from different departments with different responsibilities will have competing priorities, and the discussion can often get heated. The principle for this situation is:

**Wait for all the hot air to condense, show what's left, and build on that**

I don't mean to imply that everything my colleagues say is hot air, but a lot of hot air gets blown around in those off-sites. The principle means that I don't try to force all comments into structured format. I certainly do *not* type "1.2.19.1.10 The system shall…" every time somebody says something.

The hot air has to blow and it may be chaotic at times. I let them talk and talk and talk—maybe even the entire first day—but when the time seems right I try to begin the process of condensation. How does it condense? Hot air will always condense under the pressure of having to specify pertinent details.

It's hard to describe when to intervene, but experience helps. There are times when I can tell the conversation is going in circles because there is no clarity or consensus on one particular aspect of the system. I can tell that if folks defined and agreed on the specifics of that part of the design, they'd have a more solid basis on which to continue discussion.

This is the time to pull out a BA modeling technique from the toolbox, something unambiguous that will crystallize and distill the aspect of the system everyone needs to agree on.

Let's say the team is talking about a workflow where different roles have different functions. Most folks will be focused on defending the processing responsibilities of their own group. The conversation bogs down or starts going in circles, and I realize that discussion will improve if we agree on the role names and the logic of who can have what roles.

I ask folks to help me mock up the user entitlement screen that will be built to assign roles. We write out the role names and draw in check boxes and radio buttons to illustrate the logic of what roles one person can have. It's not about designing a screen, but about agreeing on common terms for the roles and the logic for assigning them.

With this crude mockup of the screen on the whiteboard, we now have a concrete hypothesis that can be tested as we continue to discuss who does what in the workflow. It provides a tentative consensus on some specific details, and anchors the conversation to those details. "OK, if these are the roles we have, then how would the workflow go?" Maybe the original hypothesis on the roles is all wrong but at one point we agreed on them, and we can go back together and agree on how they have to change.

The BA modeling techniques I use most for this are simple things I can draw on a white board: screen mockups, business domain class diagrams, workflow diagrams, and sequence diagrams. Sometimes I only need to model one aspect and get consensus on that to advance the discussion. Often this is a critical but mundane aspect of the system, such as agreeing on the user role names.

Personally, I feel I provide more value to my business partners this way, as compared to mindlessly writing down everything everybody says like a stenographer.

I mentioned some bad business partners. The worst was one who didn't understand the system he had, would give me requirements one day, completely forget about them the next, and was the kind of negotiator who would literally pound his fist on the table in IT planning meetings. (I do mean literally. I was on the conference call at home and could hear it pounding on the table over the speaker phone.)

But there's another business partner that can be more dangerous than the fist-pounding idiot who everybody knows is an idiot. This is the business partner who isn't as smart as he thinks he is, but still has it right half the time. This is a dangerous combination. A person like this is wrong half the time, but he thinks he is right *all* the time.

If they are ambitious, people like this can be especially dangerous, because they can be very convincing to senior managers, even though what they are saying can be dead wrong.

With this kind of person, it's best to make it look like everything is their idea, and every idea is a good idea…until they themselves see it is a bad idea. Maybe they have a requirement that is logically impossible, will lead to the users into a dead-end in the system, or will cause a huge data maintenance nightmare.

I let them explain it me, and then I model it on my screen or on a whiteboard so they can see what would happen if we were to actually build that. They have to see for themselves that the program they are asking for will not give them the output they want. They have to see how garbage going in leads to garbage coming out. When they see that, they will change their mind.

It can take a lot of patience and extra work. But it keeps the focus on collaboration and the relationship on good terms, so it is worth it.

# Advocate for Development Team

I have always considered myself more of a member of the Dev team and identify with them more than the business or project management side. I think I have had very good relations with my Dev partners. The few exceptions were when I was under the thumb of a nasty Project Manager who acted as a drill sergeant and used me to try to beat up or get evidence against the Dev team. But even in a hostile political environment, the BA has so many ways to help Dev be successful that they generally have seen me as an ally. Here are some tips I've learned over the years.

### Learn the level of prep your lead developer wants

The first time I work with a lead developer I spend time questioning him on the level of detail they want to see from me. I do this *before* I spend too much time preparing anything. It's surprising how much variation there can be among developers on this point.

For example, one lead developer looked at my requirements doc and turned me away because I had "already designed the system." He said: "You need to tell me the 'what' not the 'how.'"

Later on another project with a different lead developer, I was turned away because I had *not* designed the system, at least to the level that he wanted. He said: "You need to go back and find out what they want."

In the first case, the developer was something of know-it-all control freak. He didn't think anybody should make decisions besides him. In the second case, the developer was simply over-worked. He didn't have time to take on the project and so he was sending me away to do more leg work. In both cases I could have had a talk with them to figure what level of prep they wanted, before going to them with any specification. Once we established this, things went fine.

### Like a tourist in a foreign land, learn enough programming language to get by

As a BA, I sometimes feel like a tourist who travels between two foreign countries: the land of business and the land of developers. I have always been interested in foreign languages, so I've found it fun, interesting and rewarding to learn about programming languages.

Note that I didn't say "to learn a programming language," I said "learn *about* programming languages." I never expect to write production code, but—like a knowledgeable tourist—I have learned enough to make my life easier, and to communicate more precisely with the developers in their language. I bought a Java book to learn about the basics of object oriented-programming: how classes have attributes and methods, how methods take in some data, process it, and return some other data. I learned the basics of relational databases and SQL: tables, columns, rows, views, and queries. I have learned enough about Perl and Visual Basic to allow me to search the internet and find little programs to help me automate tasks. I can't write the programs from scratch, but I know how to steal somebody else's code and make it work for me.

This has given me confidence when talking with developers about their production code. When developers talk with me, they know they don't have to go into programming basics to explain what they

are doing. And when I give them requirements, I have a good idea of what they face when coding. This has helped me advocate for Dev while working with business partners on requirements. I have a better sense of what Dev has to do to build a feature, what is hard, what is easy, and what is impossible.

Now if I think something is easy I'm careful not to promise. I'll say: "I'm not a software engineer, but I think that should be easy." If something seems hard or impossible I can help Dev by setting business expectations.

At the same time, like a savvy tourist who knows a little bit of the language, I won't be fooled so easily. If I find a developer giving huge estimates for something I recognize as a trivial coding change, I can stand up to them with some confidence.

Plus, learning about programming is fun. There is a pleasure to reading simple, elegant code. A programming task is like a brain teaser. In most cases, I could never solve the problem myself but it's cool to see how somebody else solved the problem with a few clever lines of code.

**Faire La Mise En Place**

*Mise en place* is a technique that comes from the culinary arts. It translates to "put in place." I've found developers love it if I can help set the *mise en place* for their coding.

Julia Child explains: "all well-trained chefs prepare their *mise en place,* the professional culinary term for setting all the food and equipment needed out on the work space and prepping them before beginning to cook. It is absolutely necessary in the fast and hectic atmosphere of a restaurant kitchen…When everything is there, it makes cooking so much faster and easier—and you don't forget to add any special ingredients" (Child 1995, 278).

In a software engineering context we might say: "When everything is there, it makes *coding* so much faster and easier—and you don't forget to add any special *requirements.*"

As a BA, I am always on the look-out for tasks I can do to set the developer's *mise en place.* Generally, these are not the normal requirements artifacts, but more specific analysis tasks that I can see the developer would have to do before actually writing code.

A recent example was a requirement for our system to automatically display a bank's address when a client typed in an account number at that bank. The bank could be almost anywhere in the world. This data was provided by a vendor via giant fixed-width text files. The vendor's documentation was poor, but the information we needed was in there.

I could have left the requirement at that and let the poor developer figure out where the data was, what needed to be loaded into our database, and how to retrieve individual addresses. Instead, I sifted through the documentation and made an Excel file with annotated rows and columns that illustrated how the data was to be mapped. This was like washing, peeling and dicing all the onions, carrots and potatoes. The Dev could have done it himself, but with the annotated Excel file, he didn't have to. He could start cooking (coding) right away.

# Agile means never having to say you're certain

At my job, I've been lucky to be in a group doing Agile development processes. Now, some may say that a huge bank could never be hip enough to do *real* Agile development. That may be true; we may not be doing real Agile, but whatever we are doing has made my team quicker and more responsive. Compared to other Dev teams in the bank that are tied down by grueling bureaucratic processes, our team delivers faster, more accurately, and with a fraction of the people.

The difference is obvious to everyone who works with us. One very senior (but not very tech-savvy) business executive asked if our team could develop a major enhancement to a high-profile online banking application owned by a completely different development team in a different division of the bank. My developers had no access to this application's development environments or code repository. There was no way we could change or do anything to this application. But this senior executive asked if my Dev team could "build the enhancement and then send it to him."

While I'm a big fan of Agile, I'm not an expert on Agile, so I'm not going to talk about techniques. Instead, I would like to highlight how Agile supports Biz-Dev collaboration and avoids the confrontation that often comes from a waterfall process.

First, let's look at how the waterfall process often leads to confrontation. It begins with what you might call "The "Waterfall" Conversation."

> **Business Partner:** *(After working with BA to draft massive requirements document)* This business requirements document represents what I want you to build.
>
> **Dev Team:** *(After filling in project plan)* This project schedule promises when I will deliver it.

Typically, the requirements document and project plan are seen as "contractual obligations" to which all parties "must be held accountable." But in reality, they are simply predictions about what needs to be built and how long it will take to build it.

If both sides made excellent predictions on what should be built and how long it will take, there's no problem. But too often the predictions are faulty and we move into the "finger-pointing" phase of the project:

> **Business Partner:** "It's taking longer than you said!"
>
> **Dev Team:** "Yeah, well, your requirements were not complete. You didn't tell me about everything I needed to build!"

The key difference with an Agile process is that there is no expectation that anybody will make excellent predictions. Rather, Agile processes embrace the *biological fact* that the human mind truly sucks when it comes to predicting how long it will take to complete a challenging and complex project, like a software engineering project (Kahneman, 2011).

I say "biological fact," because in the last decade an emerging scientific understanding of the human mind has shown that our "intuitive thinking" very naturally leads us to underestimate complexity.

Psychologist Daniel Kahneman's excellent book *Thinking, Slow and Fast* is one of the best compendiums of the scientific research on this subject. It's not a management book, but describes the latest scientific consensus on how the mind works—or in some cases doesn't work the way we think it should.

The waterfall process is often brought down by a mental glitch Kahneman calls the "planning fallacy." The planning fallacy results from a "cognitive bias" to forecast "based only on the information in front of us—what you see is all there is." (Kahneman 2011, 247).

In other words, we assume "a best-case scenario [when] there are too many ways for the plan to fail, and [we] cannot foresee them all." (Kahneman 2011, 254).

The science suggests we are naturally bad at planning complex tasks. And we are naturally blind to the "unknown unknowns." On top of this is our tendency to bolster ourselves with feelings of confidence at the beginning of projects when the devil in the details is still unknown. Kahneman says:

> "Neither the quantity nor the quality of the evidence counts for much in subjective confidence. The confidence that individuals have in their beliefs depends mostly on the quality of the story they can tell about what they see, even if they see little. We often fail to account for the possibility that evidence that should be critical to our judgment is missing—what we see is all there is. Furthermore, our associative system tends to settle on a coherent pattern of activation and suppresses doubt and ambiguity." (Kahneman 2011, 87).

The difference with Agile is that we are frank about our cognitive weakness in predicting what is needed or how long it will take. The Agile mindset focuses on providing the highest value features as early as possible, and assumes that any predictions about the future are likely incorrect. Let's look what you might call "The Agile Conversation" and see how it avoids certainty and over-confidence.

> **Business Partner**: (*After brainstorming requirements on whiteboard)* "This is the kind of thing I need you to build. Features A, B and C are the most important. D, E and F are still important but they can come later."
>
> **Dev Team**: "That's cool. We'll spend the next iteration building A. After that we can see whether you want to change your mind.
>
> **Business Partner**: "Cool."

In the Agile conversation folks don't speak with any certainty about requirements or schedules. More importantly, they don't demand certainty from the other side. However, there is great confidence that they can *deal* with uncertainty, and it really has worked this way for our team. We no longer waste time preparing for the "finger-pointing" phase with the business. We don't pretend we know exactly how things will go, but we do know we are agile enough to deal with it. And when problems come up, we don't confront each other like opposing lawyers. Instead, we keep on collaborating.

# Advocate for Myself

I work from home, so I rarely see people face-to-face. Nobody reports to me, so I have no management authority. Somehow, I need to guide and influence my colleagues toward pleasant collaboration around staggering complexity for multiple projects, all running at the same time.

The only way to pull this off is to maintain a certain kind of reputation. I need my colleagues to see me as the guy who:

- Is fluent in the details of the required complexity
- Raises, tracks and brings issues to resolution
- Updates documents on a timely basis
- Is methodical and comprehensive while remaining *relaxed*

When coordinating lots of people, requirements and tasks, there's a tendency to go a little manic, a tendency to impress others with the amount of effort needed to stay on top of things. I try to avoid this. A BA can best serve as the calm face of the Dev team. As spokesman for the Dev team, the message I

want to send is: "We can do this. But we are going to do it right. We are not going to be hasty and frantic. We are going to be methodical and comprehensive."

The key to giving this impression is to demonstrate that I can stay on top of all the changing complexity without getting stressed about it. A book that helped me pull this off is David Allen's *Getting Things Done: The Art of Stress-Free Productivity.*

Allen offers a fairly simple system for tracking and prioritizing all the things you have to do in life, be it software engineering or anything else. His system is platform agnostic. It can be implemented on paper or electronically, using any software that makes sense. The goal is to capture and process details in an external system, so you don't have to think so much about it.

Allen calls it a "logical and trusted system outside your head and off your mind" (Allen 2001, 3). He goes on to explain the impact of such a system on your reputation: "When people with whom you interact notice that without fail you receive, process and organize in an airtight manner the exchanges and agreements they have with you, they begin to trust you in a unique way." (Allen 2001, 225)

I have followed Allen's advice to implement my own version of his tracking system. As a result, I'm confident I will follow through on what I have to do, and because I don't have to *consciously* remember the details, I do not worry about what I might be forgetting, which helps, especially when tracking multiple projects.

An even greater benefit (one that Allen doesn't mention in his book) is that while those details are off my *conscious* mind, they are sorting themselves in my *unconscious* mind. This may sounds like new-age hooey but it is backed up by recent research in neuroscience.

David Eagleman's book, *Incognito: The Secret Lives Of The Brain* explains what this research has shown: "The first thing we learn from studying our own circuitry is a simple lesson: most of what we do and think and feel is not under our conscious control. The vast jungles of neurons operate their own programs. The conscious you — the *I* that flickers to life when you wake up in the morning — is the smallest bit of what's transpiring in your brain" (Eagleman 2011, 4).

While the conscious mind plays the smaller role, it is an ego-maniac, taking credit for the work done by the subconscious. Eagleman explains that when you suddenly see a solution to a problem, "your brain performed an enormous amount of work before your moment of genius struck. When an idea is served up from behind the scenes, your neural circuitry has been working on it for hours or days or years, consolidating information and trying out new combinations. But you take credit without further wonderment at the vast, hidden machinery behind the scenes" (Eagleman 2011, 7).

This underscores the benefit of "keeping the details off my mind" in a new way. I track all the details in my external system, so my conscious mind doesn't worry so much, but I also do it to prime my unconscious so it can carry on with these details while I go on to the next thing.

The key to the system is to express details in the clearest, most precise language I can muster. These are, in a sense, notes to my unconscious. They may be disconnected. I may not see how they should be organized, and I may not see the solution yet. I don't worry about that. I just want to explain to myself the issue that needs to be addressed.

It is important to do this in real time, as I discover the details. I do not let them pile up, because that puts too much pressure on the conscious mind to remember them. When a new task, requirement or issue comes up, I find the appropriate place to document it as soon as I can. I may spend time re-reading the old details, and muse about a new organizing principle. I toy with it a little to seed the problem in my unconscious, but then I try to forget about it and move on to the next thing.

This method works for major deliverables, like requirements documents, and it also works for smaller project tasks, like writing emails or meeting agendas. I take the opportunity to lay out the problem or action as clearly as possible when I'm thinking about it, which is when I'm drafting the email or scheduling

the meeting. This allows me to forget about the action so I can move on to the next thing, and gives my unconscious something to chew on.

People may not read these emails or agendas carefully, but that doesn't matter, because I am doing this partly for myself. When the issue or meeting comes up, the details are clearly documented, so I quickly remember what we need to address. And sometimes, I find an insight or solution I didn't see before.

One way I remind myself to do this is to pretend I am leaving for vacation the next day. When people go on vacation, they usually think: "I know I won't remember all this stuff after I get back from vacation, so I better write it down before I go."

I try to do this every step of the way. Whenever a pertinent detail comes up I think to myself: "I don't want to have to remember this, so I better capture it as clearly as possible."

Then, at the end of the day, I go on vacation until the next morning. But I keep a note pad by my bed, because I often get some pretty good ideas when I'm not thinking about work.

# Conclusion

If you are a BA, think about actively managing your reputation with the Biz and Dev teams. You want to have a strong reputation as a good advocate for both sides.

If you are a developer, let your BAs know what level of detail you want from them *before* they go to work. Understand that they *must* advocate for the Business team, and use them to keep collaboration as effective and pleasant as possible.

If you work on software engineering projects, take a moment now and then to appreciate your team's ability to collectively manage complexity. Marvel at the complexity of the social interactions that somehow come together to define and build the complexity of IT systems. It may seem like a mess much of the time, but it is a wonder we can do it at all.

# References

Allen, David. 2001. *Getting Things Done: The Art of Stress-Free Productivity*. New York: Penguin Books.

Child, Julia. 1995. *In Julia's Kitchen with Master Chefs*. New York: Alfred A. Knopf.

Eagleman, David. 2011. *Incognito: The Secret Lives of the Brain*. New York: Pantheon Books.

Kahneman, Daniel. 2011. *Thinking, Fast and Slow*. New York: Farrar, Straus and Giroux.