Dancing with Pigs[1]

There are many ways to choreograph the activities within a sprint, but all of them involve intricate, dance like, interactions among testers[2], product owners, other team members and stakeholders.

Becoming good at agile development is a little like learning to dance. Some of us are naturally born dancers and learn quickly and seemingly effortlessly. Others of us struggle for quite a while to get our two left feet coordinating effectively with the rest of the team. But natural or not, every team member needs to become good at the agile dance. This article hopes to help you become a better dancer by exploring some of the principles of agile interactions that I have seen play out on a number of scrum teams. More specifically, we will focus on those interactions necessary to discovering and elaborating requirements within the context of the Scrum framework[3]. For clarity of discussion, I assume the simple context of a single scrum team building a product with a typical set of diverse stakeholders. The principles scale, but the details add unnecessary complexity to this paper.

Principle 1. The buck stops with the product owner, but the interactions must not - sometimes the team members and other stakeholders should dance together.

Having a clearly defined, effective product owner is an important part of what makes Scrum work so well (see principle 2). The product owner **is** the "single wringable neck" with the final responsibility for determining the rank order of the product backlog and delivering an enabling specification[4] for each backlog item, but the team has a responsibility to deeply understand the business context and provide important input into the backlog negotiations so as to:

- help the product owner create backlog items that maximize long term ROI, and
- make the myriad of detailed decisions necessary to turn the sprint backlog items into maximum business value that truly delights users.

To gain this deep understanding of the business context, and thus knowledge of what will best provide value to the organization, the team[5] must interact with a variety of individuals and multiple sources of information. I like the statement by Kaner, Bach and Pettichord[6]: "You discover requirements by conference, inference, and reference." In keeping with this good advice, team members should:

---

[1] For the Pig/Chicken analogy, see http://www.implementingscrum.com/2006/09/11/the-classic-story-of-the-pig-and-chicken/ Because the analogy tends to pit the "chickens" against the "pigs" we have not used the analogy beyond the title.

[2] Whenever I say tester, I mean any team member assuming a testing role

[3] For the interactions necessary to achieving good architecture on a Scrum Team, I recommend the new book by James O. Coplien and Gertrud Bjørnvig: Lean Architecture for Agile Software Development.

[4] http://scrum.jeffsutherland.com/2009/11/enabling-specifications-key-to-building.html

[5] Note that the team includes the product owner

[6] For a good discussion on non-traditional sources of requirements see chapter two in "Lessons Learned in Software Testing: A Context Driven Approach" by Kaner, Bach and Pettichord.

Copyright Timothy D. Korson, QualSys Solutions 2011

- observe users in the workplace,
- listen to the vision of management,
- hear, firsthand, the concerns of marketing,
- be familiar with the functionality of competing products,
- understand the impact of regulatory agencies,
- know what is being said about the product in the regular media, the social media, and product evaluation forums,
- dialog with mid-level managers affected by the system, and
- talk with support and deployment staff.

Team members are, themselves, an important source of requirements. They can envision technical possibilities that the product owner may not be able to imagine. In addition many team members bring domain expertise from their past experiences.

One standard Scrum forum for fostering these interactions is the sprint review. Another natural occasion is any pre-sprint planning meeting where the product backlog is being groomed in preparation for the sprint planning meeting. Usability testing is another appropriate activity in which the teams get direct interaction with users. Domain experts should be consulted during architectural design. All of these interactions should be seen as a crucial.

An appropriate scrum dance is like a square dance, with everyone, all together, interacting on the dance floor.

All team members need to participate in these interactions for reasons cited above:

1. **To help the product owner create backlog items that maximize long term ROI**. As indicated earlier, the team has a responsibility to deeply understand the business context and goals. This understanding is necessary when grooming and elaborating the backlog so that the final accepted version of the story benefits from the team's business knowledge. I have seen countless backlog items substantially revised, consolidated, or expanded by the product owner based on insightful input from the team. If the team neglects their responsibility to engage with the product owner and instead passively looks to the product owner as their requirements oracle, much value is lost.
2. **So that team has the necessary knowledge to make the myriad of detailed decisions necessary to turn the backlog items into maximum value that truly delights users**. No matter how detailed the written requirements or how rich the verbal interaction with the product owner, the team will make thousands of detailed decisions on their own. The value-adding quality of these detailed decisions is directly dependent on the richness of the team's understanding of the business context and goals. Please note that we are not talking about the team overstepping their responsibilities. Decisions at the requirements level are always the responsibility of the product owner – and decisions about how to build the product are always the team's responsibilities, but there are many decisions that blend how and what. These decisions are often left to the team's discretion. For example, a major tradeoff between performance and scalability should be decided by the PO, a minor one by the team. Major UI issues should be

Copyright Timothy D. Korson, QualSys Solutions 2011

clear from the enabling specification, but minor UI decisions are often made by the team. However in both of these examples it is easy to envision specifics that are not clearly major or minor and it is not always clear whether to involve the PO or not. Whenever the decision is made to not involve the PO, the quality of the decision will be affected by the level of the team's understanding of the business domain, context, and goals.

A truly exciting product is one where the mind of the user is embedded in the software and users experience the exhilaration of being able to do what they only dreamed of doing[7]. This can only occur when the developers deeply understand the mind of the user.

If the product owner is overly jealous and won't ever let the other stakeholders onto the dance floor, the team will not be able to deliver optimal value.

Principle 2. The product owner can never stop dancing.

Coordinating and synthesizing the insights and needs of all stakeholders is often more than a fulltime job[8]. To properly understand long term ROI and thus be able to optimally rank order the product backlog, the PO needs insight from business partners on value and insight from the technical team on cost. The product owner's dance with the business stakeholders must be as continuous and intricate as the team interactions. As Iain McKenna points out[9], it is as much a mistake for the business partners to abdicate their product responsibilities as it is for the team to abdicate theirs. The PO must be the catalyst to drive the stakeholders' creative input into the backlog.

The PO will often interact with business partners and users without the rest of the team being present, but during those times the team is dancing with non-team members, the PO should always be present. Because the PO cannot be in two places at once, this is not always possible, but it should be a goal, and compensated for when not achieved. For example, if the PO is meeting with a hard-to-schedule important stakeholder while the tester is running a usability test with selected users, the interactions of the usability test need somehow to be communicated to the PO. Sometimes this is a verbal discussion, sometimes it is through written or video-taped documentation, but the PO must be kept in the loop.

Interactions between the PO and the team, especially mid-sprint, are a little trickier to choreograph. Several interacting principles come into play here. The PO must not interfere with the team mid-sprint by giving them new requirements, but the PO must be available to the team to answer questions about any of those pesky emergent requirements, that are ideally not there, but in practice always seem to pop up. For example, a scrum team I am working with was implementing a back-up and restore feature. It turns out that the team decided to use a cloud-based file syncing utility that enabled a more feature rich, version enabled, restore for nominal extra team effort. The team, properly realizing the potential value of this feature, turned mid-sprint to the product owner to enquire if the additional restore functionality was desired given that it could be implemented without disturbing the sprint commitment.

---

[7] http://agileconsortium.pbworks.com/f/AgileArchitectureRedPillBluePill.pdf by Jeff Sutherland

[8] Fully exploring the role and responsibilities of the PO is a book length task.

[9] Private email correspondence, February 18, 2001

Copyright Timothy D. Korson, QualSys Solutions 2011

This type of common sense, mid-sprint, emergent requirements interaction is reasonable as long as the new functionality is an opportunity that does not disturb the team's focus or sprint commitment.

A little more difficult to choreograph is the pre-sprint planning interaction required between the team and the PO so that so that product backlog is appropriately groomed and the PO can have enabling specifications ready for the next sprint. To minimize context switching caused by unscheduled PO interruptions, some teams limit such interactions to a scheduled pre-sprint planning meeting. But this can be an impediment to the PO, leaving the PO blocked until the scheduled pre-sprint planning meeting; so most teams I have worked with allow the PO a limited amount of unscheduled interaction for the purpose of grooming the backlog. Use common sense. Inspect and adapt.

Principle 3. In a mature scrum dance, partners will not only talk to each other, they will continuously pass notes back and forth.

It is a mistake to relegate written documentation to the beginning of a sprint. Requirements get elaborated as the sprint unfolds. As pointed out in the "Enabling Specifications" pattern[10] "testers are notoriously good at sniffing out requirements lapses" and since scrum is test intensive, requirements lapses will be found.

Many of these lapses and emergent requirements need written documentation. Prototypes, code, and executable tests are often preferred by scrum teams as ways to "write down" the teams understanding of these requirements. These are powerful mechanisms, but I find that in addition to these artifacts, written test scenarios, elaborated in concert with the code, are a simple, low overhead, effective and powerful way for the development team to communicate with each other and with the PO. These test scenarios are not "the specification." The PO owns and writes the enabling specification complete with acceptance criteria. But because these test scenarios do elaborate and detail the acceptance criteria, they thus end up, as a practical matter, being treated as an extension of the requirements. Because of this, the tester needs to stay in close contact with the PO. When I have been the PO on a project, I require the tester to get my sign-off on the completeness of the test scenarios. I find that no matter how many acceptance criteria I create, the tester always finds more. For example if the specification calls for accepting cash, charge card, or corporate account, the tester ends up clarifying which charge cards are accepted and creating a scenario testing for correct behavior when a non-accepted card in attempted. If the specification details that only Visa and MasterCard are accepted for car rentals, then the tester ends up clarifying the difference between a standard and a Platinum Visa, which is important because the Platinum card includes car rental insurance when used. If the specification details how to handle Platinum cards, then the tester creates scenarios for dealing with stolen credit cards… As a PO, I find that my relation with a tester is like that of an author to reviewers and editors. When I write a paper for publication, it must be peer reviewed and then edited before it can be published. Peer reviewing always finds lack of clarity in the article and ends up with numerous suggestions for improvement. In addition, there is always the brutal editorial process. Editors always find grammar and structural improvements. As an author I am never careless in my writing, just because I know the review process will uncover imperfections. As a PO I am never careless in providing my team with an enabling specification just

[10] https://sites.google.com/a/scrumplop.org/published-patterns/product-backlog/enabling-specification

Copyright Timothy D. Korson, QualSys Solutions 2011

because I know the team will uncover imperfections, inconsistencies and lapses in it. But I do need to realize it will happen and I need to plan for it. One of the ways I plan for it is by a continuous dance with the tester. I require the tester to create short test scenarios that document all the details I forgot and all the inconsistences the team is finding – and share the written scenarios with me and the rest of the team as they are being discovered and documented. Different teams have different ways of documenting these scenarios. Some use short natural language phrases stored in a spreadsheet or written on sticky notes. Some capture them more formally in a testing tool like Fitness.

Another reason for these "written" test scenarios is that part of the specification for the sprint is discovered and delivered verbally during the sprint planning meeting. To verify common understanding of the specification, and to ensure that all required aspects of the feature are tested adequately, these verbal understandings need to be documented as light weight test scenarios that detail and augment the acceptance criteria.

I am not an advocate of heavy written documentation. Agile teams tend to focus on the multisensory interpersonal interaction that is required to achieve understanding of the business goals and system requirements. This is as it should be. Historically too much reliance was put on written documentation to communicate understanding. However, experience with Scrum teams indicates that if you rely too much on verbal communication:

- understanding is not verified, so misunderstanding, or incomplete understanding, still occurs and the team ends up doing unnecessary re-work,
- because of the cost of re-work, the product owner ends up accepting functionality that is not quite what was wanted,
- it is very difficult to scale beyond one team,
- maintenance, or sustained development of any form, is difficult,
- testing is never as comprehensive as needed or expected by the stakeholders.

Face to face communication is optimal for communicating understanding, but written requirements are necessary for communicating over time and space. Time matters to projects because memories fade and stakeholders change. Distance matters because even within a single co-located team, not everyone is present all the time at every discussion. Distributed teams obviously increase the need for written communication even more.

Who writes the documentation, and how the document is formatted is not nearly as important as that emerging understanding does get documented at the right level.

In this paper, I am focusing primarily on interaction relative to requirements, but there are many other things that can benefit from lightweight written documentation. For example, when a tester finds a bug that is symptomatic of a process issue that the tester wants to discuss during the next retrospective; write the issue down. In fact I am an advocate of the team owning a written retrospective backlog. Anyone can add process improvement items to the backlog at any time. During the retrospective, the backlog is ranked according to ROI, and the team commits to the top process improvement items for the next sprint.

Copyright Timothy D. Korson, QualSys Solutions 2011

I have also tried the idea of a written impediment backlog, that the team ranks, and the PO commits to… Use common sense. Inspect and adapt.

Thus in most mature agile dances, partners will not only talk to each other, they will continuously pass notes back and forth.

Principle 4. When dancing with a tester, let the tester lead.

When training scrum teams, I often have them discuss the following idea: "finish detailing the acceptance criteria with test scenarios before you finish the coding." Notice that the phrase does not say, "Finish the test scenarios before you start coding." To achieve maximum speed, agile teams employ parallelism, but this requires careful choreography.

Just like there are many different dances, there are many ways for requirement elaboration to unfold on a scrum team. I am going to describe one dance that I have seen work successfully on a number of projects. The following steps are simplified for ease of discussion, but you get the idea.

- Step zero is a group dance: everyone, all together. The top of the product backlog is groomed in preparation for the sprint planning meeting. This is one forum in which the team might dance with other stakeholders and gain a broader stakeholder perspective. But the primary purpose is for the product owner to clarify and uncover issues that need to be resolved before the sprint planning meeting so that the PO comes to step one with a clear, detailed understanding of the candidate stories[11] for that sprint.
- Step one, the sprint planning meeting, is another group dance, this time limited to the team. The goal is that when the dance is over the entire team understands the requirements and derived tasks in enough detail to **start** implementation. The level of written requirements detail at this point will vary depending on the project size, type, and numerous external factors such as the existence of regulatory agencies, but the team understanding should be at the level of "enabling specification."
- Steps two and three are multi-threaded and involve multiple intricate dances, each involving a developer[12], a tester and the product owner. Based upon the understanding gained during step two, a developer takes a task and starts to work on it. At the same time (or before) a tester starts elaborating the test scenarios for the related requirement. As described earlier, no matter how detailed the requirements were written out in step one, a tester will need to create more detailed test scenarios to cover boundary conditions and variations not conceived of during sprint planning. During the time when the tester is elaborating the test scenarios, the quality of the set of test scenarios will often be enhanced by the tester interacting with the developer to gain additional insight from the developer's understanding of the feature and its envisioned technical implementation. At the same time, the quality of the code being developed will be enhanced by the developer reviewing the existing set of test scenarios created by the tester. Whenever these interactions highlight a difference in product vision or

[11] In keeping with common practice, I use the term stories to denote the backlog items. If fact, scrum does not dictate the format for documenting backlog items.
[12] Or pair of developers, if pair programming.

Copyright Timothy D. Korson, QualSys Solutions 2011

understanding of the requirements, the product owner is brought into the dance for clarification and arbitration[13]. Letting the tester lead, means that before the developer declares that they have finished coding, the developer has reviewed the completed and approved set of test scenarios and is confident the code will pass tests based on those scenarios.

- In step 4, the developer has finished coding, and the tester has finished elaborating the test scenarios and test cases. The test cases are now executed against the code. Discrepancies are noted and resolved. If the test fails due to a bug[14] (type a issue), the developer debugs the code and life goes on. However, in some of the failing test cases, the code is doing exactly what the developers intends, but the developer's intention does not match the tester's expectation (type b issue).

Type b issues can, in fact, be healthy, because the process of resolving them can result in a higher value feature. The trick is to resolve them at the right time. If you resolve them too late (step 4) you end up with wasted work and re-work that does not add value. If you try to resolve all of them too early (step 1), you end up with too much group think and not a rich enough analysis and exploration of the feature.

Time boxing the sprint planning meeting helps avoid resolving issues too early. The top bi-directional arrow of Figure 1 (labeled 2&3) is key to avoiding resolving these issues too late. This arrow depicts the early and continuous dance between the tester and developer.

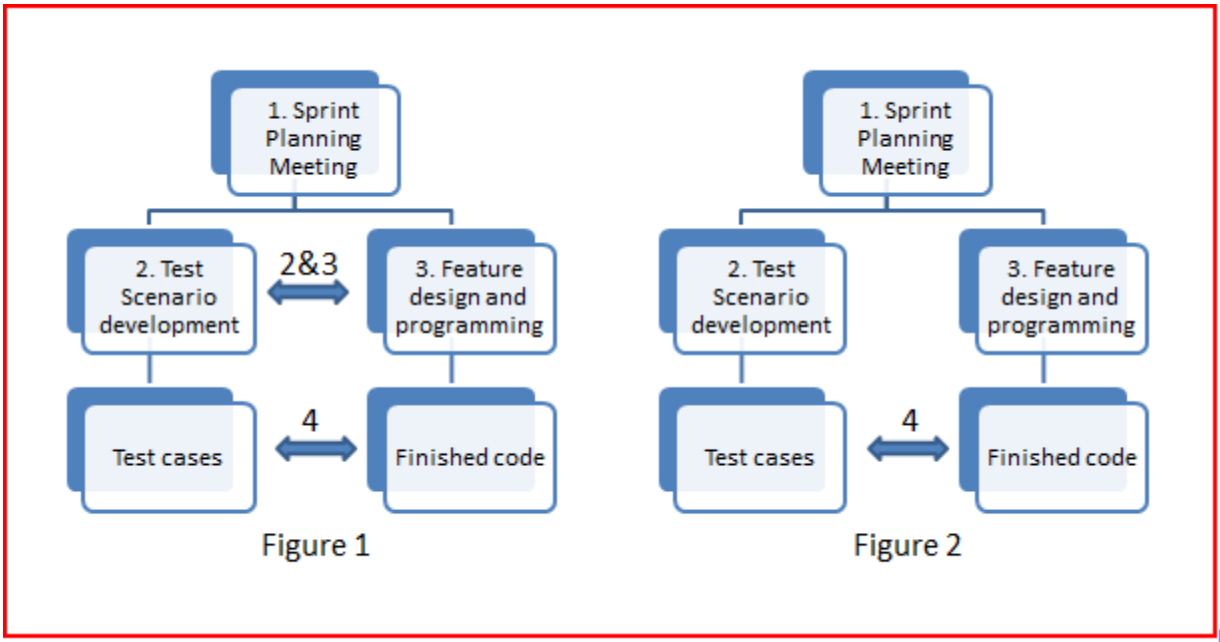
If these interactions are missing, as in Figure 2, then the team does not find out until after the code is finished, and the test cases fail, that either

1. the developers' understanding of the details was sub-optimal, or in conflict with the product owners vision. In this case
    a. time is wasted rewriting code that could have been written correctly the first time, or
    b. due to the cost of re-work, the product owner ends up accepting sub-optimal code.
2. the tester's understanding of the details was sub-optimal, or in conflict with the product owners vision. In this case time is wasted rewriting test scenarios and test cases that could have been written correctly the first time.

[13] See *Agile Testing: A Practical Guide for Testers and Agile Teams* by Lisa Crispin and Janet Gregory on the "power of three"

[14] I define a bug as when the code is not doing what the developer intended it to do. According to this definition, many failing test cases do not highlight bug, but instead highlight other issues.

Copyright Timothy D. Korson, QualSys Solutions 2011



We do not expect to find all type b issues before step 4. Some issues only become visible when faced with an actual executing implementation. Also some type b issues will be due to difference in opinion of the proper way to handle the details of a particular test case. However, issues that could be easily discovered before the code is finalized should be.

We talk about letting testers lead the sprint dance, because in many respects, in scrum, a tester's role gets blended with that of a business analysis. The product owner owns the product backlog and is responsible for delivering an enabling specification at the start of the sprint, but as a part of the development process, the specification has to be structured and elaborated. This job of structuring and elaborating belongs to the team, but lacking a business analyst to lead and document this effort, the documentation task naturally falls to the team members taking on a testing role. After all, if a test case fails, it has to be addressed. So in a very real operational sense, the test cases are an extension of the acceptance criteria. I encourage teams to treat test scenarios quite formally and to have the product owner sign off on them. In this setting, the Product Owner and Team come to an agreement on acceptance tests during sprint planning, but the dance continues throughout the sprint, with the product owner signing off on more detailed test scenarios are they are created.

Principle 5. Nano Bugs. While dancing, make sure to squash all the bugs without missing a beat.

Managers love metrics. So here is a metric that will help you know if you've mastered a properly choreographed fast-paced Scrum dance: the amount of time between when a bug is introduced and when it is killed. I prefer seconds; minutes are ok and hours are tolerable, but days are not.

I'm not suggesting that teams actually take the time to measure this metric, but it is a number every team should keep small.

In the waterfall world, projects love their bugs. They build and maintain a home for them. They nurture them and give then a name, rank and serial number. High ranking business people read and triage

Copyright Timothy D. Korson, QualSys Solutions 2011

them. In the Scrum world, we're ruthless. As soon as a bug is identified we kill it. We don't even wait for sunrise.

The test first philosophy is what keeps our bug's lives down to the nano timeframe. A pair programming dance can help with this at the unit level. The goal at the unit level is that no bug lives longer than 10-30 minutes.

The interactions of Figure 1 are designed to ensure:

1. that conceived bugs are aborted before being given life by the programmers, and
2. that bugs that make it to life and squeeze through unit testing are almost immediately caught by feature acceptance testing. If a test is right for automation, then the tester automates it before the code is finished, so that the **developer** can execute the test the moment the feature is ready. If the test is to be run manually, the tester stays in sync with the developer so that as soon[15] as the feature is ready for testing, the tester picks it up and tests it.

Once the team is confident that a feature is done, there is still the unfortunate possibility of undiscovered bugs. The sprint review, along with the early and frequent deployment of new code, is designed to uncover as quickly as possible any undiscovered bugs. An important goal of a scrum team is to minimize the number of issues discovered after the team has designated the feature as "done." The longer a bug lives, the more it costs the organization.

Principle 6. When the band is playing a waltz, it's not the right time to tango.

Is your project a desktop application? Real-time? Embedded? Mission critical? Distributed? Regulated?

Is this your organization's pilot Scrum project?

Is your team experienced? New? Large? Multi-site?

No matter how well you learned to dance on your last scrum project, your next project will require new steps. Like learning a new dance, you will find, however, that learning the optimal interactions for a new project gets easier the more scrum experience that you have.

Specifics of how to document the items in the backlog, which test automation tools to use, how much technical documentation is needed, the ratio of automated to exploratory testing, etc. will change from project to project. Experienced scrum professionals realize that the specifics of how a scrum principal is implemented are not as important as the principle itself.

Recently I've been following an animated discussion on a scrum forum about the use of the terms done and done-done. The majority of the posts caution against having a variety of flavors of "done." Multiple persons have posted quite forcefully about the potential negative consequences of using done and done-done to mean different things. I must admit, however, that I have worked with a team that quite successfully used both terms. Their success was grounded in the teams' understanding of Scrum principles. Once the team is committed to transparency, limiting work in progress, cross-functional self-

[15] The daily scrum helps with coordinating testing and development

Copyright Timothy D. Korson, QualSys Solutions 2011

organizing teams, a Keizen mindset, empirical and adaptive, frequently shipping value-adding new features, etc., the exact vocabulary, documentation style, and number of columns on the scrum board become less important. Sharp disagreements over the specifics, to the extent that face to face communication becomes tense, is far more damaging to a team than having one too many columns on the scrum board.

You may personally prefer the Tango, but if the band is playing a waltz: chill. Relax and enjoy the Waltz. Trying a tango move with your waltz partner may just get your face slapped.

Summary.

I like Cope and Gertrud's[16] focus on the secret for success: Everyone, all together, from early on.

Everyone, all together, all the time, can make scrum teams productive and fun, but a group effort without coordination can become chaotic and unproductive. The scrum framework gives teams guidelines to help them choreograph their interactions and turn a roomful of disorganized persons into a disciplined dance troupe. The underlying scrum principles are even more helpful as a team finds its own rhythm and explores new steps.

I don't claim to have introduced any fundamental new principles in this article, but I hope to have explored and discussed a few basic principles in a way that will help you remember them and encourage you to reflect on their application to your context.

1. The buck stops with the product owner, but the interactions must not - sometimes the Scrum team members and other stakeholders have to dance together.

   > *To deliver features that delight the customer and maximize value to the business, all team members must have a rich understanding of the business context and goals as well as the mental models of the end users.*

2. The product owner can never stop dancing.

   > *The Product Owner role is more than a full time job. The PO must be continuously interacting with the team as their requirement's oracle, and must be continuously interacting with all the other stakeholders as their official representative.*

3. In a mature scrum dance, partners will not only talk to each other, they will continuously pass notes back and forth.

   > *As the requirements unfold during the sprint, the details of the acceptance criteria need to be continuously elaborated, shared and reviewed. Automated tests in a business readable format, such as Fitness, are preferred for this shared and reviewable knowledge, but sometimes natural language test scenarios stored in a spreadsheet are appropriate.*

[16] James O. Coplien and Gertrud Bjørnvig: Lean Architecture for Agile Software Development

Copyright Timothy D. Korson, QualSys Solutions 2011

4. When dancing with a tester, let the tester lead.

   Don't let coding get ahead of understanding the details of the acceptance criteria

5. Nano Bugs. While dancing, make sure to squash all the bugs without missing a beat.

   Maximum velocity depends on rapid bug removal

6. When the band is playing a waltz, it's not the right time to tango.

   Inspect and adapt

There are many other principles that have been[17], and will continue to be, articulated. Ponder them. Learn from them. Debate them. Share them.

Enjoy the dance.

[17] This is the thrust of the patterns movement in the software community

Copyright Timothy D. Korson, QualSys Solutions 2011