**Agile Practices for eChronos**

I recently finished reading a book by Bertrand Meyer, “Agile! The Good, the Hype, and the Ugly”, Springer 2014 (thanks Tim). Meyer is a heavyweight computer scientist who can be ranked with Edsger Dijkstra, Nicklaus Wirth, Donald Knuth, and Alan Turing—all providers of significant advances in computer science and programming practices. He presented a description and assessment of current agile practices. Not everything he had to say was complimentary.

I have listed the ideas below that Meyer points out, and how they affect the eChronos project. I have also attached copies of the assessment chapter from his book, in case you want to see his reasoning.

**The Bad and the Ugly--**Certain practices that agile has adopted that are either an over-reaction to old practices, or driven by personal of ideological forces, and not empirical data.

1. Deprecating Upfront Tasks. Requirements analysis and architecture must be part of the pre-iteration work. I have already defined the software architecture as the MVP model, not much more is needed. I will continue defining high-level requirements before the iterations in which they are actually implemented.
2. User Stories as a Basis for Requirements. User stories are not sufficient for complete requirements or broad-enough scope, but they are necessary for unit tests and coding. I will continue writing use cases and validating them with XUML. The programmers can convert them to user stories as they wish.
3. Feature-based development and ignorance of dependencies. We don’t treat eChronos as a collection if independent features so not change is needed for this.
4. Rejection of dependency tracking tools. Don’t see a need for them for eChronos.
5. Rejection of traditional manager tasks. This doesn’t apply to eChronos because we are not reporting to an upper management cadre looking for ROI. Burnup charts and working software are enough.
6. Rejection of upfront generalization. No change here if I continue to build the domain model (class diagram for the PDC) when I validate the requirements.
7. Embedded customer. I am the customer at this point in our project, so although Meyer doesn’t think this is practical, it is a side-benefit for us.
8. Coach as a separate role. Sometimes I mentor and sometimes I coach and sometimes we pair. All good stuff. There are no talk only-write nothing people on our team.
9. Test-driven development (TDD). This means writing unit test code before writing the code method itself so that the first test fails, contrasted with having tests drive the coding at a higher level (test first development, or TFD). We have been trying to do both TDD and TFD, and finding we do not follow TDD very consistently. Neither does the programming industry. We have been doing TFD well, and have a repository of test cases and regression tests to show for it. We can lay off on the detailed test-code-refactor cycle if you wish, but I will continue to try to do it. We must not stop building unit and integration tests for each of our requirements (user stories and use cases).
10. Depreciation of documents. We need some documents, and although we don’t have the development context of an organization, we still need the requirements docs (user cases), design docs (domain model and UX artifacts), coding docs (Javadoc comments for our API), and support docs (Java standards, architecture description, Dave’s procedures, etc.). I think we are documenting at the appropriate level. If we reference at least once after it is written, the effort was probably worthwhile.

**The Hyped--**Certain practices that agile has adopted but have exaggerated benefits. They are good ideas, but not required-for-success ideas.

1. Pair programming. Good for preventing defects in the making, and allowing thoughts to be considered by a second head. We already do this, although we can do this more and better.
2. Open-space working environments. Not applicable for our distributed team, but Skype and Team Viewer works pretty well for us to share screens and ideas.
3. Self-organizing teams. I try not to assign tasks, but let people pick them as they wish, either because the task feeds into their strength and experience, or because they want to learn about that task.
4. Working at a sustainable pace. We are a volunteer team, so there is no choice but for people to work as they wish. Sometimes I would like to see certain things done by the next weekly meeting, but there is nothing I can do if it doesn’t happen. It all falls on personal integrity, and even then, we are subject to the events of life that stop us from meeting a low-priority commitment such as eChronos.
5. Producing minimal functionality. We have narrowed our weekly scope to the tasks of implementing a user story, and ensure that it works completely before. This is working ok. We should look at removing more support and housekeeping tasks that slow us down.
6. Planning game, planning poker. We decide as a team the size of use cases and stories. We don’t play with the cards, which requires a physical presence anyway.
7. Members and observers. We the distinction and roles are not applicable to our team—we have no observers.
8. Collective code ownership. We use GiT for this and I think, despite the difficulty of working with commands in the terminal window, this is working for us fine. It is a tool issue and not a repository issue. We can improve by having more code reviews before committing our code.
9. Cross-functional teams. We have what we have and all the skills we need for eChronos. I think our situation is appropriate.

**The Good—**Ideas that should be continued, whether the project is agile or not.

1. Refactoring. Revise the code to improve its design and readability without changing its behavior whenever such an improvement can be found. This is particular common as part of unit testing.
2. Short daily meetings. We have only met as a team weekly, and sometimes twice a week for pairing. I would like to meet more frequently, but time is an issue for all of us.
3. Team communication, i.e., osmotic communication. Again, due to virtual meetings, we do not have a large agile room to share, and must continue to work across the Internet.
4. Removing impediments, as a task on the Kanban board. We so this as often as we can.
5. Focus on waste removal. We can do better to identify waste activities, such as useless meetings or housekeeping tasks that slow us down.

**The Brilliant—**New ideas from agile that has moved software development forward in the industry.

1. Short iterations. With Kanban, and the nature of our team and project, there are no iterations. We select a card to work, as we need it.
2. Continuous integration and regression testing. We need to run the regression test suite at least once per day, which means after a few hours of work, after a logical break point, then move the code to the code repository. The more frequent the code commits, the easier the merges will be. Long merger result in more painful file merge issues. We need to improve our regression test suite (too many commented-out entries in the suite class) and the frequency by which we commit the code.
3. Closed-window rule. No changes are allowed to the iteration after it starts. With Kanban and no iterations, this rule does not apply.
4. Time boxing, applies to iterations again.
5. Product owner rule. I work as the product owner, and primary requirements source. In this regard, we work closer to the XP approach in that we discuss requirements as a team after the initial requirements are defined.
6. Delivering working software. We need to improve here. We go way to long before we can demonstrate working software. I would like to have something demonstrated each week.
7. Team velocity and task boards. Our Kanban board and story point estimates seems to be working fine. I am not sure of the value for eChronos of burnup charts. If there is one, I will need to come up with a way of showing a burnup chart for our progress since working software demos are infrequent.
8. Associating a test with every piece of functionality (TFD). We do this fairly well, but fall down when refactoring—we do not write tests for all additions or changes. We need to keep tighter rein on when to add, remove, or change test cases. The regression test could help a lot with this if we kept up the discipline we should.