Practices of an Agile Developer

Jan Pool

NioCAD – University of Stellenbosch

17 October 2007

Introduction

- This presentation is a summary of:
  - Practices of an Agile Developer – Working in the Real World
    - Venkat Subramaniam and Andy Hunt
    - http://www.pragmaticprogrammer.com/titles/pad/
    - ISBN: 0-9745140-8-X
    - Cost about $20 at Amazon
- I assume you are familiar with agile processes, such as Scrum.
- Disclaimer:
  - I do not have personal experience with all the practices presented here.
  - Not all practices are applicable to all organisations. Be pramatic.
  - Agile processes and practices will not solve all your problems, but it can improve your software products and increase your team's effectiveness.
    - Remember "No Silver Bullets" – Frederick P. Brooks.

My Agile Background

- Introduced to agile processes and practices via:
  - CT-SPIN meetings.
  - Courses by Jaco van der Merve (SunSpace) at Stone Three.
- Worked as an electronic/software engineer and project manager at Stone Three.
  - We were always agile.
  - Started incorporating some agile practices first.
  - Implemented Scrum later.
- Project manager at NioCAD.
  - Used Scrum from project inception.
  - We incrementally improve our processes over time.

Purpose & Target Audience

- Purpose:
  - Provide practical advise that you can apply in everyday software development projects, big or small.
  - Generate interest in agile development principles and practices.
- Target Audience:
  - Developers.
  - Project Managers.

Structure

- Book:
  - Practice number and title.
  - Devil's advice.
  - Discussions about practice.
  - Angel's advice.
  - What it feels like.
  - Keeping your balance.
- Presentation:
  - Practice number and title.
  - Angel's advice.
  - Some notes about practice.

Beginning Agility

- Practice #1: Work for outcome.
  - **Blame doesn't fix bugs.**
    - Instead of pointing fingers, point to possible solutions.
    - It is the positive outcome that counts.
  - Teams who have a culture of blaming member have lower productivity.
  - The team should work together and everyone should feel that they can admit that they have a problem.
  - People should learn from mistakes.
  - The highest priority is to fix the problem.
  - Offer to help.

Beginning Agility

- Practice #2: Quick fixes become quicksand.
  - **Don’t fall for the quick hack.**
    - Invest the energy to keep code clean and out in the open.
  - Don’t be an impatient developer who tries to fix problems without understanding them.
  - Quick fixes reduces the quality and the clarity of the code.
  - Unclear code is hard to maintain.
  - You need to know enough about a problem to fix it properly, you do not need to be an expert.
  - Do not develop code in isolation.
    - Collective ownership. (Practice 40)
    - Code reviews. (Practice 44)
  - Unit tests can help to increase code design and clarity.

Beginning Agility

- Practice #3: Criticize ideas, not people.
  - **Criticize ideas, not people.**
    - Take pride in arriving at a solution rather than proving whose idea is better.
  - Negative critique does not increase understanding and suppresses innovation.
  - Guide to people to reach solutions by asking probing questions.
  - If the team cannot reach consensus:
    - Set a deadline to force a decisions to be made.
    - Explore opposites.
    - Get a mediator.
    - Everyone should support the decision.

Beginning Agility

- Practice #4: Damn the torpedoes, go ahead.
  - **Do what's right.**
    - Be honest, and have the courage to communicate the truth.
    - It may be difficult at times; that's why it takes courage.
  - The best of plans can fail without courage.
  - Don't be afraid to ask for help, even when you headed in the wrong direction.
    - It is never to late to turn back.
  - Carefully decide if it is worth rewriting a peace of code.

Feeding Agility

- Practice #5: Keep up with change.
  - **Keep up with changing technology.**
    - You don't have to become an expert at everything, but stay aware of where the industry is headed, and plan your career and projects accordingly.
  - The software field is continuously changing.
    - And so is the job market!
  - Decide what problems a technology addresses, before pursuing it.
    - Beware of industry hype!
  - You should never stop to learn:
    - Learn iteratively and incrementally.
    - Follow the buzz in the industry, know what is out there.
    - Attend user groups, workshops or conferences.
    - Eagerly read industry books.

Feeding Agility

- Practice #6: Invest in your team.
  - **Raise the bar for you and your team.**
    - Use brown-bag sessions to increase everyone's knowledge and skills and help bring people together.
    - Get the team excited about technologies or techniques that will benefit your project.
  - The team should be willing to learn from each other and to teach each other.
  - Discuss books and articles, both classic and recent.
    - Cover technical and non-technical topics.

Feeding Agility

- Practice #7: Know when to unlearn.
  - **Learn the new; unlearn the old.**
    - When learning a new technology, unlearn any old habits that might hold you back.
    - After all, there's much more to a car than just a horseless carriage.
  - You need to realise that you are using outdated techniques or technologies before you can improve yourself.
  - We are progressing towards higher levels of abstractions.
    - Learn to handle abstraction.

Feeding Agility

- Practice #8: Question until you understand.
  - **Keep asking Why.**
    - Don’t just accept what you’re told at face value.
    - Keep questioning until you understand the root of the issue.
  - You often don’t go into problems deeply if you are under time pressure.
  - Computer systems and software are complex and many factors can influence a problem.

Feeding Agility

- Practice #9: Feel the rhythm.
  - Tackle task before they bunch up.
    - It's easier to tackle common recurring tasks when you maintain steady, repeatable intervals between events.
  - Irregular events break the team's rhythm.
  - Agile processes aim to establish a sustainable rhythm.
  - Establish a rhythm of daily accomplishment and resolution.

Deliver What Users Want

- Practice #10: Let customers make decisions.
  - **Let your customers decide.**
    - Developers, managers, or business analysts shouldn’t make business-critical decisions.
    - Present details to business owners in language they can understand, and let them make the decision.
  - Implementing a specification that the user doesn't want is useless.
    - Ignoring customers can results in costly rework later.
    - Provide flexible options when neither you or the customer is sure what the requirement is of if it may change.
  - Keep records of important decisions made by the team or the customer.
  - Do what is best for the marketability of the product, not just what is the quickest to implement or what is your preference.

Deliver What Users Want

- Practice #11: Let design guide, not dictate.
  - **A good design is a map; let it evolve.**
    - Design points you in the right direction.
    - It's not the territory itself; it shouldn't dictate the specific route.
    - Don't let the design (or the designer) hold you hostage.
  - Agile projects do not shun design, a good design is still required.
    - Upfront design by experienced team members is strategic.
    - Tactical design is done by the developers while implementing.
      - Everyone must be able to design.
  - Good design is:
    - Accurate but not precise (prescriptive).
    - Only detailed enough to allow implementation.

Deliver What Users Want

- Practice #12: Justify technology use.
  - **Choose technology based on need.**
    - Determine your needs first, and then evaluate the use of technologies for those specific problems.
    - Ask critical questions about the use of any technology, and answer them genuinely.
  - Don’t use technology just because it is new, cool or fashionable.
  - Download or buy is mostly better than to build from scratch.
  - Don’t forget maintenance costs of technologies.

Deliver What Users Want

- Practice #13: Keep it releasable.
  - **Keep your project releasable at all time.**
    - Ensure that the project is always compilable, runnable, tested, and ready to deploy at a moment's notice.
  - As part of a team, you need to keep other people in mind.
    - Don't commit broken code or forget to commit new code.
      - Compile and test!
  - Some changes are difficult to make without breaking the system.
    - Create a branch.
    - Create a sandbox.
    - Version important API.
    - Always make sure you can rollback.
  - With this practice, you can feel confident to informally demonstrate the latest build.
  - For really big changes, rather go for the short downtime.

Deliver What Users Want

- Practice #14: Integrate early, integrate often.
  - **Integrate early, integrate often.**
    - Code integration is a major source of risk.
    - To mitigate that risk, start integration early and continue to do it regularly.
  - With integration, you determine how parts interact and how information flows.
  - The longer you wait, the more time it takes and the riskier it becomes.
  - Sometimes working in isolation is good:
    - Isolation can be more productive.
    - Stay isolated when working on prototype or experimental code.
    - Use mock objects to integrate and test in isolation.
  - Integrate at least two times a day, and never less that once every three days.
  - Use continuous integration as part of your normal development cycle.

Deliver What Users Want

- Practice #15: Automate deployment early.
  - **Deploy your application automatically from the start.**
    - Use that deployment to install the application on arbitrary machines with different configurations to test dependencies.
    - QA should test the deployment as well as your application.
  - Automate deployment from the start.
    - Catching problems early, means fixing them early and hopefully at a lower cost.
  - Installers should:
    - Never destroy user data without their permission.
    - Easy to uninstall.
  - When it is difficult and time consuming to maintain deployment scripts, the design might not be optimal.

Deliver What Users Want

- Practice #16: Get frequent feedback using demos.
  - **Deploy in plain sight**.
    - Keep your application in sight (and in customers' mind) during development.
    - Bring customers together and proactively seek their feedback using demos every week or two.
  - Requirements are fluid and will most likely change during a project. Embrace it.
    - Stable requirements imply stable markets, competition, no learning, evolution or growth.
  - Getting feedback from customers means that you can correct course more often.
    - Track all change requests, features, bugs and critical decisions.
    - Do not demonstrate broken functionality, it is annoying to the customer and portrays a bad image.
    - Try to get feedback at least once a month.

Deliver What Users Want

- Practice #17: Use short iterations, release in increments.
  - **Develop in increments.**
    - Release your product with minimal, yet useable, chunks of functionality.
    - Within the development of each increment, use an iterative cycle of one to four weeks or so.
  - Identify core or risky functionality and implement first.
    - Don’t be distracted by nice to have features.
  - Hard deadlines help to focus the team’s effort.
  - Increment and iteration lengths should be set by the team/project rhythm.

Deliver What Users Want

- Practice #18: Fixed prices are broken promises.
  - **Estimate based on real work.**
    - Let the team actually work on the current project, with the current client, to get realistic estimates.
    - Give the client control over their features and budget.
  - You cannot fix time, budget and quality at the same time.
    - By fixing price and features, you compromise quality.
    - What if requirements change?
  - Estimation and planning techniques can improved, especially over the life of a project.

Agile Feedback

- Practice #19: Put angels on your shoulders.
  - **Use automated unit tests.**
    - Good unit tests warn you about problems immediately.
    - Don’t make any design or code changes without solid unit tests in place.
  - Agility is about managing change and code changes the most.
  - Testing takes time, invest your time wisely.
    - Make tests more effective, do not just make more tests.
  - Use xUnit tools. Also look at code coverage tools.
  - Run the unit test each time you compile or build your code, especially before check-ins.
  - Get a dedicated build machine.
    - You can run the tests regularly.
    - This also ensures that tests work in isolation.

Agile Feedback

- Practice #20: Use it before you build it.
  - **Use it before you build it.**
    - Use Test Driven Development as a design tool.
    - It will lead you to a more pragmatic and simpler design.
  - Test Driven Development:
    - Write tests before the code.
    - Help you think how you will use the code.
    - Help to reduce overly complicated design.
  - Only implement features that you need.

Agile Feedback

- Practice #21: Different makes a difference.
  - **Different makes a difference.**
    - Run unit tests on each supported platform and environment combination, using continuous integration tools.
    - Actively find problems before they find you.
  - Differences in platforms and environments can cause problems.
  - Continuous integration tool should report problems automatically.
  - Use virtualisation tools, such as VMWare and Virtual PC, to save on hardware costs.

Agile Feedback

- Practice #22: Automate acceptance testing.
  - **Create tests for core business logic.**
    - Have your customers verify these tests in isolation, and exercise them automatically as part of your general test runs.
  - You will need the customer's input for these tests.
    - You might also need test data, ask for it early.
  - Use a tool such as Framework for Integration Testing (FIT) to automate.

Agile Feedback

- Practice #23: Measure real progress.
  - **Measure how much work is left.**
    - Don’t kid yourself - or you team - with irrelevant metrics.
    - Measure the backlog of work to do.
  - Measure how log tasks actually took and learn from previous estimations.
    - Over time you estimation skills will improve.
  - Keep a backlog of tasks that still needs to be done.
  - Focus on functionality that needs to be implemented, not just calendar time.

Agile Feedback

- Practice #24: Listen to users.
  - **Every complaint holds a truth.**
    - Find the truth, and fix the real problem.
  - A misunderstanding by users is still the teams problem.
    - Fix the program, documentation, training material, etc. to help users.
    - Look especially at cases where many users have the same misunderstanding.
  - Provide useful information to users when something goes wrong in your program.

Agile Coding

- Practice #25: Program intently and expressively.
  - **Write the code to be clear, not clever.**
    - Express your intentions clearly to the reader of the code.
    - Unreadable code isn't clever.
  - We have all seen code that is confusing and unclear. After some time, even for the original developer.
  - Choose readability over convenience.
  - Should we choose readability over performance?
    - In most cases yes.
    - Typically only a small portion of an application require clever tricks.
  - Work systematically:
    - Understand what the code does. (Hard)
    - Figure out what to change.
    - Change code and test.

Agile Coding

- Practice #26: Communicate in code.
  - **Comment to communicate.**
    - Document code using well-chosen, meaningful names.
    - Use comments to describe its purpose and constraints.
    - Don’t use commenting as a substitute for good code.
  - A good name relays a lot of information, a bad one nothing and a terrible one incorrect information.
  - Commenting:
    - Purpose. Why does an entity exist?
    - Requirements (pre-conditions): inputs, object states, etc.
    - Promises (post-conditions): object state, returned values, etc.
    - Exceptions: what can go wrong, what are thrown?
  - Do not comment obvious and irrelevant information.

Agile Coding

- Practice #27: Actively evaluate trade-offs.
  - **Actively evaluate trade-offs.**
    - Consider performance, convenience, productivity, cost, and time to market.
    - If performance is adequate, then focus on improving the other factors.
    - Don’t complicate the design for the sake of perceived performance or elegance.
  - Be careful of focusing on any one aspect of your product exclusively.
  - No one best solution fits all circumstances.
  - Donald Knuth: Premature optimisation is the root of all evil!

Agile Coding

- Practice #28: Code in increments.
  - **Write code in short edit/build/test cycles.**
    - It's better than coding for an extended period of time.
    - You'll create code that's clearer, simpler, and easier to maintain.
  - Pause periodically to reflect on your work.
  - Take breaks away from the computer.
  - Constantly look for small ways to improve (refactor) your code.

Agile Coding

- Practice #29: Keep it simple.
  - **Develop the simplest solution that works.**
    - Incorporate patterns, principles, and technology only if you have a compelling reason to use them.
  - We are often very proud of creating complex systems (it was difficult!), but we should be proud when we create functional simple systems that work well (it can be even more difficult!).

Agile Coding

- Practice #30: Write cohesive code.
  - **Keep classes focused and components small.**
    - Avoid the temptation to build large classes or components or miscellaneous catchall classes.
  - Cohesion is a measure of how functionally related the members of a component are.
    - A high level of cohesion indicates that the members work toward one feature or set of features.
  - The way you organise a component makes a big difference in your productivity and overall code maintenance.
  - Follow the *Single Responsibility Principle*: a module should have only on reason to change.
  - Use the *Model-View-Controller* pattern to separate the presentation logic, the control and the model.

Agile Coding

- Practice #31: Tell don't ask.
  - **Tell, don’t ask.**
    - Don’t take on another object or component’s job.
    - Tell it what to do and stick to your own job.
  - An object's logic should be the object's responsibility.
  - Use the C*ommand-Query Separation Principle*: categorise each of your methods as either a command or a query.
    - A command change the state of an object, but can return information as a convenience.
    - A query return information about an object's state.

Agile Coding

- Practice #32: Substitute by contract.
  - **Extend systems by substitute code.**
    - Add and enhance features by substituting classes that honour the interface contracts.
    - Delegation/Composition is almost preferable to inheritance.
- *Liskov's Substitution Principle*:
  - "Any derived class object must be substitutable wherever a base class object is used, without the need for the user to know the difference."
- Use inheritance for *is-a* relationships.
- Use delegation/composition for *has-a* or *used-a* relationships.

Agile Debugging

- Practice #33: Keep a solution log.
  - **Maintain a log of problems and their solutions.**
    - Part of fixing a problem is retaining details of the solution so you can find and apply it later.
  - The log should be searchable.
  - Searching online helps, but not for project specific problems.
  - Share your log with team members and with the world.
  - Bug tracking software can also serve as a log.
  - Some items that you can log:
    - Date, short description, detail of the solution, references (articles, URLs, etc.), code, settings, screenshots, platform and version.

Agile Debugging

- Practice #34: Warnings are really errors.
  - **Treat warnings as errors.**
    - Checking in code with warnings is just as bad as checking in code with errors or code that fails its tests.
    - No checked-in code should produce any warnings from the build tools.
  - Not all warnings are benign.
    - Learn what the warnings mean.
  - Check how your compiler can set warning reporting levels and learn how it works.
    - Consider setting your compiler to treat all warnings as errors.
  - Set your continuous integration build to the same levels.

Agile Debugging

- Practice #35: Attack problems in isolation.
  - **Attack problems in isolation.**
    - Separate a problem area from its surroundings when working on it, especially in large applications.
  - Use unit tests to isolate problems.
    - Unit testing forces you to layer your code using good abstractions.
    - Use mock objects to stand in for other required modules.
  - Use prototype and experimental code to help isolate problems.

Agile Debugging

- Practice #36: Report all exceptions.
  - **Handle or propagate all exceptions.**
    - Don’t suppress them, even temporarily.
    - Write your code with the expectation that things will fail.
  - Determining who is responsible for handling an exception is part of design.
  - Report an exception that has meaning in the context of the code.

Agile Debugging

- Practice #37: Provide useful error messages.
  - **Present useful error messages.**
    - Provide an easy way to find the details of errors.
    - Present as much supporting detail as you can about a problem when it occurs, but don't bury the user with it.
  - It is not always possible to display all information required to debug a program to the user.
    - Use logging to provide this information for developers
  - During development, look for error messages that are hard to understand, even for developers.
  - Error reporting can have a big impact on support costs and developer productivity.

Agile Collaboration

- Practice #38: Schedule regular face time.
  - **Use stand-up meetings. (Daily Scrum)**
    - Stand-up meetings keep the team on the same page.
    - Keep the meeting short, focused, and intense.
  - Each member answers these questions:
    - What did I do yesterday?
    - What am I planning to do today?
    - What is my impediments?
      - Talk about specific issues after the stand-up meeting.
  - Some advantages:
    - Start the day focused (when holding meeting in the morning).
    - Everyone knows what the rest of the team does.
    - Make issues visible and get help.
    - Identify where more man power is required.
    - Identify redundant work where solutions exists.
    - Speed up development by sharing.

Agile Collaboration

- Practice #39: Architects must write code.
  - **Good design evolves from active programmers.**
    - Real insight comes from active coding.
    - Don’t use architects who don’t code – they can’t design without knowing the realities of your system.
  - Designers need to know the small details of the system to be effective.
  - Encourage, and allow, your developers to design.
    - Start with small responsibilities.

Agile Collaboration

- Practice #40: Practice collective ownership.
  - **Emphasise collective ownership of code.**
    - Rotate developers across different modules and tasks in different areas of the system.
  - Any person who understands a piece of code should be allowed to work on it.
  - Project risk increases if code is kept exclusively to one person.
  - When multiple people work with code, the code is constantly checked, refactored and maintained.
  - The overall knowledge of team members increase.

Agile Collaboration

- Practice #41: Be a mentor.
  - **Be a mentor.**
    - There's fun in sharing what you know – you gain as you give.
    - You motivate others to achieve better results.
    - You improve the overall competence of your team.
  - Feel eager to share your knowledge and make those around you better.
    - You can even start a blog and share with the world.
  - By explaining to others, you also gain insight.
    - Areas where you struggle to explain, indicates areas where you need to learn more yourself.
    - When explaining the same thing over and over, consider writing an article or book about the topic.
  - Limit the time how long people can be stuck on a problem.

Agile Collaboration

- Practice #42: Allow people to figure it out.
  - **Give others a chance to solve problems.**
    - Point them in the right direction instead of handing them solutions.
    - Everyone can learn something in the process.
  - A mentor must teach people how to help themselves.
    - Guide by answering questions with questions.
    - If someone is stuck, show them the answer.
  - Advantages:
    - Learn how to attack problems.
    - Learn more than just the answer.
    - Make sure the same questions are not asked over and over.
    - Help your team to function when you are not around.
    - You can also learn when your students discover new solutions or ideas.

Agile Collaboration

- Practice #43: Share code only when ready.
  - **Share code only when ready.**
    - Never check in code that's not ready for others.
    - Deliberately checking in code that doesn't compile or pass its unit tests should be considered an act of criminal project negligence.
  - Get in the habit of checking in code as soon as you are done with a task.
    - Work in increments.

Agile Collaboration

- Practice #44: Review code.
  - **Review code.**
    - Code reviews are invaluable in improving the quality of the code and keeping the error rate low.
    - If done correctly, review can be practical and effective.
    - Review code after each task, using different developers.
  - It is the most cost effective to fix problems soon after they occurred.
  - Everyone should be reviewed (junior and senior).
  - Styles:
    - The all-nighter: Massive period reviews.
    - The pickup game: Review as soon as code is written and tested.
    - Pair programming: Program in pairs (Famous XP practice).
  - Follow up on review recommendations, otherwise reviews are useless.

Agile Collaboration

- Practice #45: Keep others informed.
  - **Keep others informed.**
    - Publish your status, your ideas and the neat things you're looking at.
    - Don't wait for others to ask you the status of your work.
  - Don't wait until the deadline to deliver bad news. No one likes that kind of surprise.

Final Comments

- You can use some/all of these practices even if you do not use an agile process.
- Don’t try to implement all these, and other, practices at once.
- Focus on one practices at a time.
- Start with the practices that you think will improve your effectiveness the most.
- It is better to apply some practices than none.

Questions?

- Questions?

- Many more Pragmatic titles at:
  - http://www.pragmaticprogrammer.com/
- Contact Information:
  - jpool-at-pshymorphic.com
  - http://www.pshymorphic.com/
  - http://research.ee.sun.ac.za/niocad/