

Community-Agile Software Guidance

Ian Clatworthy, Canonical

Abstract

This paper presents the values, principles and practices of Community-Agile Software Guidance, a process framework that refactors Agile to incorporate the lessons learnt from the Open Source world.

Contents

- 1 Introduction
 - 2 What is Software Guidance?
 - 3 Values
 - 3.1 Knowledge over co-location
 - 3.2 Community over in-house
 - 3.3 Integrating over planning
 - 3.4 Quality every build
 - 4 Principles
 - 4.1 Eliminate waste
 - 4.2 Amplify learning
 - 4.3 Decide as late as possible
 - 4.4 Deliver as fast as possible
 - 4.5 Empower the community
 - 4.6 Build integrity in
 - 4.7 See the whole
 - 5 Technical Practices
 - 5.1 Release Early, Release Often
 - 5.2 Product Backlog
 - 5.3 Feature Branching
 - 5.4 Open Peer Reviews
 - 5.5 Automated Regression Testing
 - 5.6 Pre-Commit Continuous Integration
 - 5.7 Product as Platform
 - 5.8 One-click Upgrade
 - 6 Social Practices
 - 6.1 Open Collaboration
 - 6.2 Community Sprints
 - 6.3 Self Selection
 - 7 Getting Involved
 - 8 Tool Support
 - 9 Problems with Community-Agile
 - 10 Conclusion
 - 11 Acknowledgments
-

1 Introduction

For moderately sized, co-located teams developing software, Agile processes like [SCRUM](#) and [XP](#) have now established a solid track record of success and are crossing the chasm into mainstream usage. In many cases though, the parameters differ from those [preferred when using Agile development](#): large numbers of people may be involved, these people may be scattered across the globe and development activities may be tightly interwoven with deployment and support. In these cases, much of the magic of traditional Agile - putting people together in one place and leaving them alone while they develop the next iteration - simply doesn't apply.

Taking a high level viewpoint, Agile is just one of several ways of developing complex adaptive software. [Lean software development](#) and [Open Source](#) processes also encourage emergent design and arguably scale better. I believe the next wave of software process innovation will combine ideas from all of these communities - Agile, Lean and Open Source - into a holistic framework that will be more broadly applicable than any one of the methods popular today. Community-Agile is just one example of how this can be done.

2 What is Software Guidance?

When new software is being released every few weeks, days or hours, partitioning teams and processes along the lines of development, deployment and support is a mistake: these things should not be considered in isolation. In much the same way that *design a little, code a little, test a little* replaced the waterfall model common before the [Unified Process](#) became popular, development needs to be treated as just one activity in the broader process of "guiding" software as it grows up.

Specialization is good but it's the wrong thing to partition teams on. As the Open Source movement has shown, self organizing communities certainly contain experts in all of the required roles from idea to executing software including visionaries, designers, coders, writers, artists, testers and users. There *are* barriers but these are largely based on willingness and ability to contribute, not role specialization. As Eric Raymond has eloquently pointed out, there is [no master plan for the cathedral](#), just a bazaar of highly motivated people with a core team co-ordinating and guiding the collective effort.

3 Values

While SCRUM and other agile methods were established long before the [Manifesto for Agile Software Development](#) was written in 2001, having a document that clearly states what things are most valued has helped many people better understand what Agile is ultimately about. With a similar objective, I present the *Manifesto for Community-Agile Software Guidance ...*

We are uncovering better ways of developing, deploying and supporting software by doing it and helping others do it. Through this work we have come to value:

Knowledge and motivation over sitting together

Community collaboration over in-house development

Integrating contributions over iteration planning

Quality every build over quality every release

That is, while there is value in the items on the right, we value the items on the left more.

This manifesto is not meant to replace the Agile one but to extend it, highlighting where the emphasis is different. An explanation of the values is given below.

3.1 Knowledge over co-location

Some regions truly are centers of expertise in certain fields. For example, the financial capitals of the world are a rich source of IT professional who understand banking. Likewise the national capitals *ought* to be full of people who understand how government works. Generally speaking though, employing people to work on software based on where they live is valuing the wrong thing.

Why? Software engineering is ultimately a communications challenge. Sitting people together greatly increases the bandwidth at which people can communicate and it ought to be done whenever possible. However, open source communities have clearly demonstrated that highly knowledgeable people, using the right processes and tools, can work together effectively regardless of their location. To stress the critical importance of having the right people, I contend the following to be true:

Two experts on opposite sides of the world can normally solve a tough problem faster together than any one of them can with a non-expert right next to them.

Of course, knowledge alone is not enough and motivation plays a key part in successful communication. If you want to build great software, start with the very best people you can for that problem domain, ensure they are motivated and can work together. Be sure to get them together a few times a year to build trust and relationships, but don't worry about everyone being co-located.

3.2 Community over in-house

It has long been understood that it takes a village to raise a child. In modern terms, you might prefer to call it the [Wisdom of Crowds](#) or the benefits of [Wikinomics](#). Regardless of how you convince yourself it's a good idea, seed and grow a healthy community around your software. In a great deal of cases, the community is just as important as the software itself, if not more so. It is understood that for commercial reasons, completely open communities aren't always possible. Even in this case, look to make your closed community as large and open as you sensibly can.

A key tenet of Agile development is having constant communication between those who know the business and developers. While Agile is far more effective than the document-centric development processes that preceded it, Community-Agile goes further than "in-house customer" by enabling those who know the business to **be** developers.

This doesn't necessarily mean that you must open source your software, though that is often an excellent choice. It does mean that your software needs to be **open for extension** via mechanisms such as plugins, reporting toolkits, scripting APIs, hooks and event triggers. It also implies open communication with your community of stakeholders about road-maps, designs of proposed new features and known issues.

Good collaboration with a community of stakeholders also leads to better testing, quality assurance, end-user support and localization. A community can bring far more environments, resources and skills to the table than even the largest software companies can afford to set-up and employ. However, it must be stressed that open software with a healthy community around it does not eliminate the need for

in-house QA and support personnel. Instead, the nature of these roles changes just as it does for product managers and developers: listening and co-ordinating become more important, and take more time, than they do in traditional closed development teams.

3.3 Integrating over planning

One of the most important aspects of Agile development is its facilitation of [adaptive over predictive planning](#) in the medium term. Community-Agile takes this approach further by favoring adaptive planning even within an iteration.

When a community is collaborating openly to raise software, you simply don't know who will propose what and when that will happen. Even when stakeholders have volunteered to work on changes important to them, it can be difficult to predict with any certainty when changes are going to be put forward and what the quality of any particular change will be. Ignoring external interruptions - like community contributions - and focusing on commitments made at the start of the current iteration can be sub-optimal.

Why? To maximize velocity and hence the value of the product to the community, changes meeting quality requirements ought to be included into the product as early as possible. Even when changes are below the quality required, it is important that the core team provide timely feedback to those submitting proposed changes in order to keep the thinking behind the change fresh in the submitter's mind. Being responsive in this way enhances community involvement over time, further accelerating product development. In other words, the velocity of the core team is definitely important but growing the community and increasing their effective involvement may well get you there faster over time.

3.4 Quality every build

Even when everyone is co-located, breaking the build by accidentally committing bad code is frustrating and expensive. When people are scattered across multiple locations, it's absolutely critical that the quality of the evolving code base (trunk) be maintained each and every commit. Bad quality of the trunk is a cancer:

- developers lose large amount of time chasing down problems that they shouldn't need to
- users on the leading edge delay testing new code because the cost to them outweighs the benefit.

On the other hand, high quality each and every build is addictive. Insisting on a release-quality build every time saves time and energy, the team is well-rehearsed at what it means to do a release build, and can react quickly if a new release is needed at short notice (to fix a security issue say).

A large number of the required practices given later in this paper - including Feature Branching and Pre-Commit Continuous Integration - are designed to ensure quality every build. Even if the rest of Community-Agile doesn't make sense for your project, your life - and that of your team/community - will improve for the better if you adopt this value.

4 Principles

Lean thinking is the common bridge between Agile and Open Source. The principles of Community-Agile are therefore the same as the principles as Lean Software Development. Re-use is good, particularly when the source material is as soundly based as Lean is.

Some brief comments on each principle in the context of Community-Agile are given below. For a more detailed look at Lean as it applies to software development, see Mary and Tom Poppendieck's outstanding book [Implementing Lean Software Development: From Concept to Cash](#).

4.1 Eliminate waste

Writing software than solves the wrong problem is waste. The Self Selection practice reduces the odds of that happening.

Partially completed work is waste. It is therefore important than changes proposed by the community be reviewed and integrated in a timely manner.

Defects are waste. Many of the practices including Open Peer Reviews, Automated Regression Testing and Pre-Commit Continuous Integration work together to reduce the chances of defects being introduced into the trunk.

There is undoubtedly efficiency in specialization and having the right people as business analysts, architects, coders, writers, testers, deployment and support engineers is important. Hand-offs from one group to another are a great source of waste so minimize the barriers by organizing teams around product areas, not role specialization.

4.2 Amplify learning

The rate of which a community can grow is largely determined by how successfully knowledge can be disseminated. Many of the practices including Open Collaboration, Community Sprints and Open Peer Reviews are designed to amplify learning.

4.3 Decide as late as possible

This principle fits closely with the *Integrate contributions over iteration planning* value. The Feature Branching practice makes this possible without incurring unwanted and unpredictable integration costs.

4.4 Deliver as fast as possible

Short iterations are good because they reduce the time required from great idea to running code in the hands of end users. When software is delivered as a service, the time from one released version to another may even be less than a day.

The key to speed is the *Quality every build* value. The One-click Upgrade practice also directly supports this principle.

4.5 Empower the community

Maximize the ability for the community to contribute. Be open about plans, proposed designs and known issues by using Open Collaboration. Make your software easy to extend in documented and supported ways using the Product as Platform practice.

4.6 Build integrity in

Deliver an automated regression test suite in lockstep with the software.

4.7 See the whole

Great things can happen when a diverse group of people work together as one towards a shared goal. Encourage communication so that developers better appreciate what users want to do, and so users better appreciate why the software has been designed like it has. Encourage excellence in specialized fields but encourage people to put themselves in the shoes of others.

Embracing community-centric processes leads to a reduction in direct control and increased uncertainty. On the other hand, it strengthens your relationship with your partners and customers and often leads to a better end result. Focus on outcomes, not deviations from plans, when measuring success.

5 Technical Practices

5.1 Release Early, Release Often

A large number of technical practices are shared by the Agile and Open Source worlds and this one is [arguably the most important](#). Vision is a good thing but releasing something quickly gives early adopters something to sample and build on. Releasing often after that sets a heartbeat that the community can synchronize their efforts around.

There is no right answer as to how often you should release - it is highly dependent on the nature of your community and project. Different parts of your community can have different needs as well. To note the broader trend in the IT industry though, the time between new product releases has been getting shorter year after year.

5.2 Product Backlog

Ideas are the life-blood of innovation. In the words of Linus Pauling ...

The best way to get a good idea is to get lots of them.

As explained in [The Elegant Solution](#), Toyota implements *one million ideas a year*. You can only innovate like that if your culture encourages creating and recording ideas and gives people the freedom to continuously improve things that impact them. Open Source does exactly that.

The list of desired features is known as the Product Backlog in SCRUM. In Community-Agile, the Product Backlog can take many forms including:

- an Ideas database like [Ubuntu Brainstorm](#)
- a list of Blueprints for new features
- an Issue Tracker with feature requests, suggested improvements and bugs.

There are advantages and disadvantages to each and using all of them might be fine for large projects. For smaller projects, starting with an Issue Tracker is often enough. The most important thing is that people wanting to make an improvement can find similar or related ideas and the discussions held to date about them.

5.3 Feature Branching

The widespread adoption of Continuous Integration has been one of the most important practical software engineering advances in recent years. It wasn't that long ago that "Integration" was viewed as a project phase, one that always took weeks longer than 99% of project schedules had allowed for it. By encouraging developers to break work into short tasks and to integrate and commit every day, integration has gone from being a major problem to a non-event for teams using this practice. Central version control tools like CVS, Subversion and Perforce have been instrumental in helping teams keep evolving code synchronized.

There are however problems caused by encouraging developers to commit their changes several times a day. Practically, quality dips during the course of each development iteration and there is still a mad scramble towards the end of each iteration to restore quality back to that achieved when the software was last released. Developers should certainly be making small changes and *snapshotting* those changes frequently. However, trunk quality can be maintained at a much higher level if those changes are only *integrated* when they are fully cooked. This practice and the technology enabling it - distributed version control - is now being heavily adopted by open source communities and is beginning to see uptake by commercial development teams.

There are many advantages to this practice:

- Developers are continuously working from and integrating with a more stable base, so much less time is spent chasing down why their sandbox broke after they last synchronised.
- Release managers can delay making decisions as to whether to include a given change or not. For example, if quality is below par for a new feature, it is no longer necessary to hack in some changes at the last moment to hide it. Instead, the code simply isn't merged until it does meet quality standards.
- All the commits related to a logical change are merged into the trunk at a single point in time instead of being intermingled with other changes over several days/weeks. As a consequence, it is easier to find what change introduced a bug and easier to back out a logical change if required.
- The trunk is more reliable so more users use and test it every day.
- The trunk can be shipped at short notice as the next production version.
- The practice can be scaled to hundreds of developers by using a hierarchy of gatekeepers that pull changes from feature branches into integration branches.

This practice also has several implications:

- Code, tests and documentation need to be managed and version controlled as one. That doesn't mean that one person must be an expert at all of these things. It does mean developers, quality engineers and technical writers need to be able to collaborate on a branch using the one version control system. It also means that code, tests and documentation need to be stored in formats that can be easily merged.
- The automated build needs to build both code and documentation.

For a deeper look at the technology that makes feature branching practical, see my paper on [Distributed Version Control Systems - Why and How](#).

5.4 Open Peer Reviews

Pair Programming can be a useful way of developing software for two main reasons:

- It helps to share knowledge and enable collective code ownership.
- It provides instantaneous code review.

In a distributed team however, Pair Programming is often not an option. There is also substantial benefit in reviewing an *overall* change as opposed to only reviewing lots of small changes. For example, each of the small changes can be fine but the overall change may still be lacking some integration tests or documentation changes.

When a Feature Branch is ready to be merged, an Open Peer Review needs to approve it before it can be integrated. By making the review and ongoing discussion open to the community, knowledge is shared and learning is amplified. If the reviewer asks for changes, additional commits are made to the Feature Branch and it is resubmitted.

The rules around peer review need to be simple and transparent. We have found the following guidelines useful:

- Submitted changes need to be accompanied by a "cover letter" email explaining why the change is needed and how it works at a high level.
- New code should alleviate bugs, improve features, improve speed or simplify the code base.
- New code must come with test cases that proves that the modification makes the promised changes and associated user documentation changes.
- Changes proposed by a core developer need review by another core developer. Changes proposed by community members less familiar with the code base need review by two core developers.
- Community members should be encouraged to perform reviews of changes that they are interested in. While core developers votes are still required before those changes can be accepted, reviewing is a good way for community members to apply their skills and learn more about the code. It also assists core developers by raising issues they may not have considered or by confirming that the design meets the broader community needs for a feature or otherwise.
- Trivial changes (like a spelling mistake in a comment) can be integrated by a core developer without review. As a matter of courtesy though, the developer should still inform the community that the change was made.

5.5 Automated Regression Testing

Many developers have found [Test Driven Development](#) and [Behavior Driven Development](#) to be a productive ways of developing software. Community-Agile does not require tests to be written first but it does require executable tests to be submitted at the same time as code changes.

It is also important that integration tests are provided in addition to unit tests. Ideally, resource utilization tests that check performance, memory usage, network usage and disk usage are included as well, though frameworks for helping with this challenge have some way to go before this will be practical for most projects. Utilization testing is also more complex because the results are numbers along a scale, not simple pass/fail checks like functional testing allows.

As important as automated testing is, keep in mind that ad-hoc testing plays an important role in any quality assurance program. Usability testing in particular will always require people sitting down and using software, perhaps in ways it was never intended. One way to maximize the amount of ad-hoc testing done on new features is to keep trunk quality as high as possible so more users feel safe running the latest build.

5.6 Pre-Commit Continuous Integration

Traditional CI tools run regression tests after each commit and notify the team when the build breaks. Pre-Commit Continuous Integration turns this around so that a change is never merged *unless* all tests still pass. This practice is an important part of keeping trunk quality sacred.

For any non-trivial project, running all regression tests takes many minutes, if not hours. It is therefore important to have tool support that enables trunk commits to be submitted asynchronously and for developers to be able to continue working without interruption while changes are being tested and merged into the trunk. Distributed version control and Feature Branching complement this practice in that developers can continue working with minimal interruption despite the tight control over trunk commits.

5.7 Product as Platform

Agile is all very well and good but rapid evolution only works as systems grow if the architecture is clean and well considered. Good architecture is the key to sanity. Having an architecture does not imply [Big Design Up Front](#) but it does imply putting together some [high level conceptual views](#) of the software so developers can find their way around more easily. To manage complexity over time, [well established design principles](#) must also be followed.

In a distributed development community, architecture is even more important than it is for in-house teams. The product needs to be a platform that others can independently build on and feel safe about investing time to do that. As a consequence, there needs to be documented and supported ways of extending and enhancing the product.

Community-Agile succeeds in environments where you win when your customers win. The right architecture makes these win-win outcomes more likely. As Tim O'Reilly explains in [The Architecture of Participation](#), participatory architectures like the World Wide Web result in users pursuing their own "selfish" interests building collective value as an automatic by-product.

Of the numerous conceptual views making up the architecture, one of the most critical is the domain model. [Domain Driven Design](#) is the recommended technique to use to create this. Pay particular attention to communicating and maintaining the [Ubiquitous Language](#) in a distributed community.

5.8 One-click Upgrade

Just as building software needs to be a painless process in Agile development, upgrading software needs to be a painless process for your users for Community-Agile to succeed. There are several parts to this:

- Core software and dependencies.
- Configuration settings.
- Databases.

With Product as Platform, users need to easily upgrade any add-ons they have as well. Some add-ons might be generally available to everyone in the community while others may only exist locally at a customer site and be in varying stages of maturity.

A range of technologies are needed to support this practice including smart package management and database refactoring/migration tools.

6 Social Practices

In addition to the technical practices, Community-Agile recommends some selected social practices as outlined below.

6.1 Open Collaboration

In Agile development, a [good team room](#) can play a key role in keeping everyone informed about plans, priorities and progress. In a distributed environment, open and frequent communication about these things is equally important. The most commonly used tools to achieve this include issue trackers, wikis, mailing lists, Internet Relay Chat (IRC), forums, calendaring and code hosting services.

There is an additional overarching challenge when working in a distributed community: transparency. When everyone in a team works in the one location, an amazing amount of information is communicated informally: in the elevator, at the water-cooler, over lunch, etc. If you want to keep your community engaged, the important parts of those conversations need to be made public so that those outside the office can still contribute to discussions that interest them. Be aware that making this cultural adjustment is no small feat and allow plenty of time for it when trying Community-Agile for the first time. For a detailed treatment of good policies to use in distributed collaboration, see Karl Fogel's landmark book [Producing Open Source Software](#).

6.2 Community Sprints

Note: In Community-Agile (and Open Source in general), a "sprint" refers to the practice of getting people together: it is not a synonym for an iteration (as the term is used in SCRUM).

While it isn't necessary for everyone to work in the one place, it is important to get people together at least annually for face-to-face time. The main reason for doing this is a deeply personal one: it is much easier to trust people you've met in person, and trust is the foundation on which all good relationships are based. Hiring a new person into the core team is a particularly good time to have a community sprint.

Of course, there are also a large number of activities that are best done around a table or around a white-board. These include:

- high level planning
- discussing and debating designs
- retrospectives on how processes can be improved
- getting new community members trained up.

On the softer side, getting people together is a great way of building community spirit. Most attendees come away from these events energized with new ideas, or with a clearer insight into what will be required to implement existing ideas.

Many technical conferences like [PyCon](#) are now explicitly including community sprints as part of their program. In other cases, the community get together may be large enough to be its own event, e.g. the six-monthly [Ubuntu Developer Summits](#).

6.3 Self Selection

Just as self-managed teams are an important Agile practice, self selection of tasks to work on -

scratching your own itches - is an important practice in all community-centric development models. To put it another way, there is no central manager in charge of resourcing and deciding who works on what when. Feature Branching, Open Peer Reviews and Product as Platform complement this practice. The overall effect is to change the software management game, trading control for performance.

Of course, this makes road-maps and planning less predictable. On the other hand, a great deal of development gets done that simply wouldn't be done anywhere near as quickly if there was only a small team working behind closed doors. The other big upside is that the odds of new features missing the mark in terms of requirements are greatly reduced, given people are mostly likely to be working on and contributing to features that directly impact them - stuff that they feel passionate about.

Self-selection does not imply working in isolation. Contributors are encouraged to discuss ideas with the broader community before starting large changes. In some cases, several people or groups may be interested in a feature and decide to work together in bringing it to fruition. It is also important to break larger development blocks into small tasks, deliver those frequently, and keep other developers and users informed about the overall plan and progress.

7 Getting Involved

My vision is for the Community-Agile methodology to be guided forward using itself! If you wish to join the community developing, deploying and supporting it, visit the Community-Agile web-site, <https://launchpad.net/community-agile>. The framework is fully open source so you can download it, tweak it and roll it out on your project. If you want to extend it, make a Feature Branch!

There are a massive number of really useful practices (in [IXP](#) for example) that I could have included in Community-Agile. With a view towards keeping it simple and broadly applicable to most projects, I decided against this. I also support the approach recommended by Boehm and Turner in [Balancing Agility and Discipline](#): it is better to start with a small methodology and extend it to suit your needs than starting with a complex methodology and cutting it down. Community-Agile is designed to be that base methodology that communities and teams can build on. In time, my hope is that teams will develop Add-Ons that can be shared. For example, Add-Ons might extend the base methodology with coding standards for various languages, or add any number of useful practices that make sense for many (but not all) projects.

If you share the values promoted by the *Manifesto for Community-Agile Software Guidance*, please get involved and help us make it better!

8 Tool Support

To explain things using Lean terminology, the software guidance game is all about maximizing the rate at which value flows to the community. Good tools can make all the difference in terms of effectiveness and encouraging people to think the right way. Always keep in mind though that tools are a means to an end. Furthermore, this area changes rapidly so the information in this section may well be out of date in a year or two. Be sure to see the [Community-Agile web-site](#) for updated information about tool support.

The critical piece of technology needed to minimize the overall cycle time is smart package management. Some applications (e.g. Acrobat) now check with a central web site for updates and tell the user when new versions are available. Other applications (e.g. Firefox) do this for both the core application and for add-ons built on top of that application. This functionality is also part of all modern

operating systems including Linux, Windows and OS X. An excellent example of this technology - [APT](#) - is freely available in Debian, Ubuntu and any Linux distribution based on these. By leveraging that technology, your application, required libraries and dependent add-ons can all be safely upgraded as frequently as needed. Security updates can also be managed using the same infrastructure. If shipping on Linux is not an option, be sure to find a smart package management solution that works for you and makes One-click Upgrade practical. If required, build the technology directly into your application using frameworks like [Sparkle Updater](#).

A huge and growing range of tools are available to support Open Collaboration. Wikipedia provides links to and comparisons of [Issue Trackers](#), [Wikis](#), and [Forum](#) software.

Good version control software is the key to practical Feature Branching. Distributed version control systems including [Bazaar](#), [Git](#) and [Mercurial](#) are the best option here, though central version control tools with good branching and merge tracking can be used as well with some success. [PQM](#) provides Pre-Commit Continuous Integration for Bazaar projects. [TeamCity](#) provides a similar capability for Java developers.

The least mature and in many ways the most interesting tool space is the one between version control tools and smart package managers. How well can the workflow be managed from Feature Branch to One-click Upgrade? Helping communities grow and work well is core business for Canonical, so this issue is an important one for us. Many of the features recently added to [Launchpad](#) including [merge proposals](#) and [Personal Package Archives](#) assist in bridging the gap between software being developed using Bazaar and being deployed on Ubuntu. [Further streamlining](#) is also under development.

9 Problems with Community-Agile

Every methodology has its limitations and problems. Here is a brief introduction to some of the potential issues you may face when adopting Community-Agile.

While open and transparent collaboration are critical, the down-side is that the amount of communication and the time taken to digest it all can be overwhelming as more and more people contribute. At an individual level, developers need good triaging, prioritization and time management skills or their productivity suffers. See [Getting Things Done](#) (TM) and [Inbox Zero](#) as examples of general methodologies that can make a big difference in this area.

Increasing the number of people involved can also lead to delays in making decisions. At times, it is necessary for community members to agree to disagree, for the project leader to choose the direction forward and for everyone to get on with it. There is always a risk that some community members may leave after a heated debate. On the other hand, going over and over the same arguments time and time again - not going forward - will also cause people to leave. It simply isn't possible for everyone to agree about everything. When dictatorship becomes necessary though, it is crucial that the leadership is completely transparent about why a particular direction is being selected over the alternatives.

Community development introduces a range of legal issues. To simplify license management, some projects request that contributors assign Copyright to a single person, company or organization. Most projects also ban anonymous contributions so that the source of each change can be more easily established in the event of (for example) patent disputes.

Feature branching has a considerable risk of waste through work that either goes out of date or is never integrated, At other times, it indirectly encourages people to take on too many different things at once. Keeping track of in-progress work using tools can help but most communities minimize these problems

via social means, e.g. a core developer helping to "sponsor" a change through the process, or gently inquiring about where an expected item of work is up to.

Utilization testing produces "numbers along a scale" that require subjective interpretation by a human. That means it can't be done every time and therefore some changes decrease overall quality. It's good to look for creative ways to make the results *red/green* deterministic so these tests can be enforced each and every build. This is very much an unsolved problem to my knowledge.

Finally, be aware that Community-Agile is not a full lifecycle methodology: projects still need to be initiated using some sort of process and some projects eventually reach retirement. In a corporate setting, an umbrella project/program management framework like [PRINCE2](#) might be used to decide which projects ought to be initiated and when. For IT projects, Scott Ambler's [Enterprise Unified Process](#) nicely describes many *enterprise disciplines* that need consideration and are well outside the scope of Community-Agile. At the other extreme, many Open Source projects start because someone has a problem to solve and some ideas on how to do it. Necessity remains the mother of invention!

10 Conclusion

Useful software of any notable complexity is continuously being integrated into new environments in much the same way that living organisms adapt to their surroundings. Software that does not adapt and evolve becomes extinct. Guiding complex software as it evolves requires more than an empowered team in one location with an in-house customer representative: it requires a community of the very best minds you can muster, and it requires a process framework for that community to work together effectively.

While it has evolved largely independently of Agile, the Open Source movement has pioneered many best practices for developing, deploying and supporting software. To those who mistakenly view discipline and agility as being on opposite ends of a spectrum, Open Source projects may appear unmanaged and chaotic. In reality, it takes a clear vision, strong leadership and a great deal of discipline to deliver high quality software over and over again as many open source projects do.

There's nothing revolutionary in Community-Agile - it's simply a combination of selected Agile practices with selected Open Source practices using Lean thinking as a common bridge. Community-Agile values are an extension of Agile values. In many ways, Community-Agile is more conservative than popular Agile methods in that it places a higher premium on architecture, peer reviews and quality. In other ways, it is more radical by being even more people centric and encouraging adaptive planning even within iterations. Above all though, Community-Agile changes the scope of the game from software development to software guidance. Stakeholders and end users become partners that share the responsibilities, risks and rewards of not just development but also deployment, support and ongoing planning.

The good news about Community-Agile is that the practices are known to scale and work well. The tools vary from good to great though integration between the tools is currently immature. The bad news is that there is currently (mid 2008) limited experience using the practices in a commercial context. While the social practices are arguably the more important, I expect the technical practices to gain acceptance first in many companies. It takes a great leap of faith to open up and embrace communities: some companies will simply never make the necessary cultural adjustment to do it. For those that do and do it successfully, the rewards are high. These include gaining a deeper understanding of, and integration into, your customers' businesses. You might just produce some amazing software while you're at it.

The common ground between Agile and Open Source is undoubtedly a fertile place for the next wave of software process innovation to spring from. There is no right way that applies to every project. We can however mash up process frameworks that adapt and scale better than today's popular methods do. I hope Community-Agile proves to be a step in that direction.

11 Acknowledgments

I'd like to thank the many people who have inspired me over the years by writing passionately about software processes. These include Fred Brooks, Tom Gilb, Humphrey Watts, Steve McConnell, Grady Booch, Peter Coad, James Rumbaugh, Jeff Sutherland, Robert C. Martin, Martin Fowler, Scott Ambler, Joel Spolsky, Eric Raymond, Andy Hunt, Dave Thomas, Mike Cohn, Mary Poppendieck, Tom Poppendieck, Robert Glass and Karl Fogel.

I'd like to thank the many people at Canonical who have contributed to this paper. Mark Shuttleworth started the ball rolling by encouraging me to write it. Martin Pool, Elliot Murphy, Barry Warsaw, Steve Alexander, Maris Fogels, Dustin Kirkland and Joey Stanford reviewed early drafts and provided valuable feedback. Most of all though, I'd like to thank the fantastic Bazaar community. Each and every day, they prove that Community-Agile works awesomely well, and continuously improve the processes and tools needed to make it work for others.