Blue-White-Red, an example Agile process:

Blue-White-Red is a simple Agile system originated by Liz Sedley and myself for a London company transitioning to Agile development. There wasn't a great deal of up front thought that went into this system, we just started trying to approximate XP (Beck 2000) and at first it was a poor approximation. We modified our process as we went along. Looking back there is a heavy Scrum (Schwaber and Beedle 2002) influence but mainly this is what worked for us.

This process originated at one company and eventually Liz and myself left the company and went our separate ways. We both took the same basic process and implemented it elsewhere with modifications. This description also draws on our experience, since so although I've described it as one company it is more of a composite image.

I have come to the conclusion that you can't just take an Agile, or any other, process off the shelf and use it. You have to create a process that works for you. If you do want to take an existing process and implement it then you are going to need help. In fact I would go as far as saying I doubt you can actually implement XP unless you actually have Kent Beck, Ward Cunningham or one of the other process authors actually on your team. The same goes for any other process you care to choose, DSDM, Scrum, Crystal, etc.

Basics

The whole system revolved around a large magnetic white board upon which index cards were placed to represent work. The cards themselves were blue, white or red and held on the board with magnets. The board was marked with important information like iteration deadline, a record of how much work was done in previous iterations and was divided into four columns: *work to do, in progress, waiting for test* and *completed work.*

Product Managers produced *Product Requirements Documents*. Pieces of work from these documents, usually features, would be written on blue index cards. The information on these feature cards only needed to be brief, perhaps a title and a

document section. These could be prioritised and the highest priority cards looked at in more depth.

According to most Agile methods each card should represent a complete piece of deliverable work. However the nature of our product, the existing code base and perhaps our own inexperience meant that one 'feature' was more work than could be done in a short space of time. So we put the features on blue cards and developers would break the work down into a set of tasks written on white index cards. For every blue feature card there would be multiple white task cards. Each white card could be achieved in a day or two. If it couldn't we tried to break it down further.

The break down from blue to white was usually done during the bi-weekly planning meeting. If the feature was very complicated or poorly understood a special meetings might be held to discuss the work and break it down. Developers could also add white task cards to the work pile if they felt some piece of remedial work was needed, e.g. larger refactorings.

Iterations were two weeks long. They would finish in the morning - a Tuesday, Wednesday or Thursday, never a Monday or Friday. The online systems would complete with a release to live. Then in the afternoon we would convene a planning meeting. (More recently I have been running one week iterations with monthly releases to a live server.)

Planning

In the planning meeting the Product Manager would select the features to be implemented during the next iteration. During the iteration the team would focus on only these features and their associated tasks. Other blue cards would be held offline in an index card box. Since each feature could be quite large there would normally only be a few (one, two or three) feature cards in play at any one time.

Work estimation was done in abstract points. At first this caused some confusion but teams quickly converged on a shared understanding of how much work could be accomplished in a single point. Estimates usually ranged between half a point and two points. Occasionally zero point cards would be written to remind ourselves of things or for trivial tasks.

Although each team placed a slightly different value on a single point we normally found that a card with a point value of more than three needed to be broken down further. We also found that the more words used on a white card to describe the task the more accurate the estimate. Cards with brief descriptions were usually poorly understood and poorly estimated.

The first task in the planning meeting was to clear the board and count the point value of the cards completed in the previous iteration. This was recorded and used as a guide for the coming iteration's capacity. The team could accept slightly more points into the iteration than had been completed the previous week on the understanding that some might not get done.

With blue cards and white cards prepared and our estimate of work that could be done the Product Manager would prioritise all the white cards. Developers would advise of any dependencies between cards, risks, opportunities and such but the final say on prioritisation was the Product Manager's. All cards were prioritised in absolute order - 1, 2, 3, and so on. No two cards were allowed the same priority. This made it clear what was the top priority and which the last. When priorities are set as "must have", "should have" and "nice to have" teams typically end up with too many *must have's* to tackle in one go so the actual work order gets decided by the team. If a business abdicates its responsibility to articulate its needs and priorities to developers then it should not be surprised by the results. (Although it might be difficult to communicate this message to the business.)

Once work was prioritised the cards could be placed on the board. This formed the *work* queue. Wherever possible we tried to avoid associating individuals with pieces of work. When someone is named as the individual responsible for a given piece of work the queue does not get worked from top to bottom. Other individuals skip a card which is associated with someone else even if it has a high priority. Of course sometimes you need a particular individual to work on a particular piece of work, but on the whole we tried to avoid this.

Knowing how many points of work the team had completed in previous weeks meant we could be quite confident of what would and would not get done during the coming iteration. Say a team had done 10 points in the previous iteration, we would put about

13 or 14 point on the board for the coming iteration. We could be fairly sure 7 would get done, hopefully another 4 would be cleared and if we were lucky then we might get more.

Developers were not supposed to work on more than one card at a time, this was intended to keep focus. However some developers would cherry pick cards they felt they could do as 'back ground tasks' or during spare moments. While well intentioned this tended to be disruptive. Developers usually picked refactoring cards and because it was a secondary tasks it became difficult to track the status. On one occasion I found several cards on a developers desk, he had intended to do them in 'spare time' but the fact that they were on his desk meant the work was hidden.

Each day the team would hold a short stand up meeting and select the cards they would work on from the work queue. Some developers would choose to pair on some work but we did not pair all the time. If work was completed without pairing it would be subject to a short desk based code review before checking into source code control.

Testing

Developers tried to write unit tests for the new features. However due to the existing legacy code base this was not always possible. There were no unit tests for code that already existed so refactoring existing code was difficult. All unit tests were run each night after the nightly build was completed. Should the build or any tests fail the whole team received mail and the first person in started to investigate the failure.

Each team had a software tester who was responsible for accepting a completed white card. When the developer felt a card was complete it would be moved to the *waiting for test* column on the whiteboard and the developer would take another card. Only when the tester was satisfied the work was completed would it be marked and moved into the *completed* column.

Testers had a variety of ways of testing cards: they could perform a manual test, they might ask to verify the unit tests were working and they might ask for proof the code had been reviewed. If they were not satisfied, or a defect was found, then the card would move back to the *in progress* queue with a high priority.

Finally, if a fault did slip into the system, or was reported by another team and it was added to the team's work load it would be written up on a red card. Red cards automatically took priority as the next piece of work to be started. Unfortunately the nature of the system meant it was difficult to eliminate such tasks but over time the number did fall.

Each team that has used this process has modified it in different ways. No team was able to eliminate all manual testing because it was not possible to retrofit unit tests to parts of the legacy code base. Over time unit test coverage increased but never covered the whole application.

One project that used extensive COM components had particular problems with Test Driven Development (TDD). Testing at the component level tends to be too general, one COM call tends to do too much stuff to test properly, neither is it possible to test the state of the object satisfactory after the COM call. Testing inside the component, below the COM interface tends to be difficult because COM gets into code in all sorts of odd ways. Whether it is memory management, COM pointers, lifetime, startup, or shutdown issues, COM makes it difficult to isolate code for testing.

Micro project variation

I term projects with very little developer resource micro-projects. Typically a micro project has less than two full time developers. Even if there are more than two developers attached to the project, when they are split between multiple projects the net effort may be less than two full time people. For example, developer Fred is full time on the project but Pete is split between three different projects

One variant of the Blue-White-Red process was used on a one developer micro-project where I played the role of Product Manager with additional responsibility for project management. Here we made several modifications. Firstly the code base was smaller and leant itself more to one feature, one task, one card so we were able to dispense with blue cards altogether and just work with white and the occasional red.

Second there was no tester on the project, neither had the developer attended the TDD training course our other developers had. So in the last day or two of the iteration development would stop and the developer would test. If necessary myself and others

would join in too. Since this was an online system we could put interim versions of the software onto a staging server during the iteration for preview by customers and feedback.

Finally, there was a pre-planning meeting where the developer and myself (as Product Manager) would get together a few days before the end of the iteration and assess what had been done, what work was coming up, and how long it might take. After this meeting I would be able to decide my priorities prior to the planning meeting

Good luck

What I didn't recognise at the time was how lucky we were to start with. For example, we had source code control and a regular build in place. We couldn't do intra-day builds, it just took too long, but we could know within a day if things were broken.

We were also fortunate that a well-established product owner system was already in place in the form of Product Managers. Although Agile development is good at handling vague and changing requirements you still need someone to articulate what the requirements are and answer questions about how the system would work. In too many organizations developers are left without this guidance and have only their own resources to fall back on. This can work when a product is mature or when developers are close to the final customers. At other times the business requesting the software seems to rely on direct thought-transference.

Some things we weren't so lucky with. When you start with a large legacy code base getting unit tests in place is hard. It is not impossible but it is hard. I think there are two main problems here. Firstly there is a mental block: too many developers have a kind of automatic dislike of the word "test". "Testing" is associated with "software testers" who are obviously paid less and therefore better suited to testing while developers, well, develop.

We had managed to persuade our management that TDD was *a good thing* and they had paid to bring a trainer in house to deliver a series of training courses. Although most of our developers were trained in TDD some did not feel it was worth doing or believed it took too much time. Unfortunately the management who had paid for the

courses declined to make the use of TDD mandatory so usage was patchy in the company apart from the blue-white-red teams.

Still most developers do not have experience of adding tests to legacy code so they simply don't know how to do it. There is one book on the subject (Feathers 2004) but as good as this is it is no substitute for experience. In retrospect I recognise the need to employ a part-time TDD coach to work with the team in addition to the training.

Retrospectives

The other thing I would do differently is to do more retrospectives. This is hard because we did hold some. Our problem was getting the rest of the company to help implement the recommendations that came out of the retrospective.

Any retrospective will suggest a number of changes in the way people work and the processes followed. When these changes are entirely within the team they are relatively easy to change. But inevitably, over time, these changes are made leaving the more difficult ones to make.

The more difficult changes typically need the involvement of people outside the team and the support of more senior managers. Unless these managers take part in the retrospective it can be hard for them to see the need or opportunity for the change. However trying to persuade a manager to spend several hours in a retrospective is hard.

Like our process our retrospectives were a simplified form and again we used blue, white and red cards. Normally I would start by constructing a timeline (Kerth 2001). This is useful because it helps the team remember the early days of the project, without a timeline people tend to concentrate on the most recent events. The time line also helps put the whole project in perspective.

I would put a generous supply of cards on the table and at any time in the retrospective people could write on the card and throw them in a pile in the middle of the table. The rule was:

- Blue cards for thing that worked and we should do again.
- White cards for suggestions for change: these were specific suggestions to do something different.
- Red cards for puzzles, things we don't understand and would like to discuss some more. (Recently I have experimented using Red cards to capture things 'To avoid' doing.)

	Iteration	**Retrospective**
Blue cards	Feature under development	Things we did right (should do again)
White cards	Development task	Suggestions for improvement
Red cards	Fault (to be fixed as priority)	Puzzles (variation: things to avoid)

Table 1 - Summary of card use

As we created the timeline (usually on the wall with Post-It notes) people would suggest ideas , write them on cards and throw them into the pile. Once the timeline was built we would walk through it and discuss the events and their sequence. As we did so more cards would be written.

Eventually we would reach a point where we understood the project better. Then it was time to start to wrap up. I would take the cards and sort them into three piles, one for each colour. There was no strict rule but I usually worked through the red cards first. By this stage we had normally answered a lot of the puzzles already. Some of the puzzles would be beyond our understanding and others we could resolve and produce blue and white cards

Next we would start on the blue cards. I would read them out and we would agree (or not) to keep the activity on the card.

Finally the white cards: things to do differently. Quite often people would have suggested the same things. Here it depended on the team and the items in the pile. Some items everyone would agree on and they were within our power to change.

Other items we might not agree on, maybe some people would want to change and others would not. Sometimes I would have the team vote on the top three things to change. By limiting the items to change effort can be focused.

Since coming up with this formula Agile Retrospectives (Derby and Larsen 2006) has been published. There are more exercises in this book which I will try to incorporate in future retrospectives.

Conclusion

In writing this up I hope to convey a sense of how an Agile process can work, and how you can start with something quite simple and build up. Inevitably I've hidden some details, knocked off some rough edges and highlighted our successes. Simply deciding to follow this, or any other, process doesn't remove all your problems overnight. You still have to work at them.

What this process does do is increase focus and expose problems. Once problems are exposed you can go about fixing them. This is where great improvements can be made. This process provided us with a framework which allowed us to start adopting Agile ideas and to start improving.

Unfortunately exposing problems does not make you popular. Showing the slow pace of development does not look good even if it is true. Exposing problems means someone needs to fix them rather than not ignore them.

The technique I now call Blue-White-Red has worked, with modifications, at three different companies. I don't think this makes it universally applicable but it does show that you can roll-your-own Agile process and I encourage more people to give it a try.

A short version of this piece appears in *Changing Software Development: Learning to Become Agile* by Allan Kelly, published by John Wiley & Sons, 2008.

References

Beck, K. (2000). Extreme programming explained, Addison-Wesley.

Derby, E. and D. Larsen (2006). Agile Retrospectives, Pragmatic Programmers.

Feathers, M. (2004). Working Effectively with Legacy Code, Prentice Hall.

Kelly, A. (2007). Changing Software Development: Learning to Become Agile, John Wiley & Sons.

Kerth, N. L. (2001). Project Retrospectives. New York, Dorset House.

Schwaber, K. and M. Beedle (2002). Agile Software Development with SCRUM.