Patterns for Agile Development Practice
Part 2

Joseph Bergin
Pace University
jbergin@pace.edu
(Version 5)

This set of ten patterns is intended to complement the standard wisdom that can be gleaned from the Agile Development literature such as Kent Beck's *Extreme Programming Explained*. It is directed primarily at those who are starting out with Extreme Programming or another agile methodology and might miss some subtle ideas. Once a team gains experience these patterns will become obvious, but initially some of them are counter intuitive. While this study began in Extreme Programming practice, most of the advice applies to agile development in general. The ten patterns here extend the work of 2004-2005 on the same topic.

We consider XP to be a pattern language in which the practices are the basis of the patterns. They have the characteristics of a true Pattern Language in that they are synergistic and generative. The dozen or so practices detailed in Beck and elsewhere, such as "Do the simplest thing that could possibly work" and "Yesterday's Weather", form a subset of this language.

As this "language" is in its early stages of development, there is no significance to the current ordering of the patterns here. This paper presents ten of the fifty or so patterns developed so far. The full set of patterns is listed briefly in the Thumbnail section at the end.

The pattern form used here is as follows

Name
Context Sentence: Who the pattern is addressed to and when in the cycle it can be applied.
Problem paragraph. The key sentence is in italics.
Forces paragraphs. What do you need to consider in order to apply this pattern? In this version we will put the forces in bulleted lists.
Therefore, solution. Key (usually first) sentence is in italics.
Commentary and consequences paragraphs

These are written in the "you" form as if the author is speaking to the person named in the pattern's context sentence. "You" could be a customer, a developer, or even a manager, depending on the pattern.

Thumbnails and acronyms appear at the end of the paper.

An Agile Development Story (a.k.a. Fairy Tale):

A team has been formed by a **Sheltering Manager**, consisting of an **Onsite Customer**, seven developers, and a tech gofer with knowledge of infrastructure and tools issues that are customary in the organization. The **Whole Team** has found a workspace and set it up, both physically with tables for the workstations and virtually with the development platform. The latter includes testing, code repository, and integration tools. The **Whole Team** along with the **Sheltering Manager** has gotten together for 3 days to **Train Everyone** under the direction of the **Effective Coach**. The **Onsite Customer** and the developers have gone through the initial development of the **Guiding Metaphor** and started on the **Planning Game**. The metaphor gives them an initial vocabulary to ease the communication between the customer and the developers.

The manager agrees to **Be Human**, and the **Whole Team** commits to the practices. While they know this will be difficult, they also trust the Manager to maintain a **Sustainable Pace** for the development. They have had some fun in the training sessions and those exercises have also helped go gel the team.

A day is spent with the **Whole Team** beginning the just-in-time requirements gathering. Thirty stories have been written, covering an **End To End** first cut at the desired functionality and these stories have been estimated by the team. The team has (**Flexibly**) set its initial **Velocity** quite low as they are new to this project. A few of the stories were larger than can be accomplished in a single two-week iteration by an individual, so these have been broken into tasks and the **Implementer Estimates** these **Tasks**.

The customer chooses a few of the most important stories, looking at their value and their cost as reflected in the estimates. Development begins on just these stories, using **Test First** development with **Executable Tests.** Many questions are asked of the **Onsite Customer** and the answers to the **Questions Generate Acceptance Tests**. One of the developers works with the **Onsite Customer** to begin to develop an **Acceptance Test** suite for the application. Meanwhile, the developers **Program Promiscuously**, frequently changing partners so all are familiar with the code being developed. As each task is completed it is (**Continuously**) **Integrated** into the build, so that all unit tests pass. Of course all programmers **DTSTTCPW** in all coding and design, thereby **Delivering Value** to the customer. The **Onsite Customer Checks Off** the **Tasks** when they are done, reviewing unit tests as appropriate and noting the changes in the **Acceptance Tests** written and passing. Halts are called when **Acceptance Tests** that were passing suddenly fail. This is obvious since we use only **Executable Tests** and the suite is executed frequently; especially at each integration: several times each day.

The team holds daily **Standup Meetings** to flag difficulties and assure that each person knows what progress they will make that day. The **Coach** and **Social Tracker** keep the meeting going using advice from the **Coach** (**ScrumMaster**) on process, and the **Tracker** on progress. They hold the **Schedule Sacred** and will end the iteration at noon on Friday. The afternoon is used for **Retrospectives**, games, and the **Planning Game** for the next iteration.

The **Coach** works with the **Onsite Customer** to make sure that the first release (after one month) is both **End to End** and delivers **High Value First**. The Customer works with her own business groups to determine value and current required direction. She then uses this

knowledge to guide the team via the **Planning Game** for each iteration. She continues to write stories and feeds them in to the process at iteration points to keep the direction consistent with her needs.

It takes the team a while to learn how to estimate effectively, and this becomes an issue at the **Retrospective** held at the end of each iteration. The **Coach** helps with suggestions and training games to make estimation more effective. The notion of personal Velocity is initially foreign, but the Coach and Tracker get the developers to record their work to give a baseline for their personal practice. After a couple of months the Coach makes only periodic visits to the team, but is on site for **Retrospectives** and is available to be called whenever the tracker or manager thinks it would be helpful.

The developers **Spike** when they must to learn how to estimate and build things. They **Refactor** the code whenever new stories can be built more easily by changing the existing code (improving its design). This is made easier as they own the code collectively and have been pairing promiscuously throughout. Thus they take **Collective Responsibility** for the project and its code. The **Executable Tests** give them the confidence that they don't break anything when they refactor.

They give **Full Communication** to the customer on opportunities, costs, and options available. As time permits they **Re-Estimate** a few of the older stories to keep the **Planning Game** short and sweet. They learn to give **Best Effort** at all times and to **Ask For More** when they complete tasks early. Conversely, the customer doesn't beat them up when they must beg to drop work from an iteration when difficulties occur or unfortunately low estimates are occasionally made. The Customer learns that **Easy Does It** is a better policy than pushing (too) hard, though everyone demonstrates commitment and the nature of the work room makes everyone's effort obvious to all.

The Coach and Tracker keep **Everything Visible** in the development so that all stakeholders can see the current direction and measure the rate of progress toward the (ever changing) goal. **Cards and Whiteboards** are in evidence in the team space containing most of this information.

Alas, user level documentation was neglected for the early iterations, so a documentation specialist joins the (**Whole) Team** and starts to build the documentation structure that is henceforth kept up with the software development. Thereafter, **Documentation** becomes **Just Another Task**.

After several iterations and a few releases, the customer realizes that she now anticipates more work than the team can produce in the desired time available. She works with her business partners to choose between (a) later delivery of greater functionality, (b) earlier delivery of a smaller application, or (c) higher cost per week by growing the team methodically. She chooses the last option (after cost-benefit analysis) and the team decides to add two developers each iteration for the next two iterations. Pairing gives them the confidence that this won't slow them down by much and that the newbies will meld quickly into the team.

The project completes at the desired date. Some functionality that was thought desirable was not built, but the customer gives this low value. The overall cost is about 80% of the back-of-the-envelop estimate initially given for the project by executive management. The

customer gets a promotion from her team. The developers hold a party to celebrate their raises, swearing fealty to the manager. They all live happily ever after.

Implementer Estimates Task

You are a developer on an agile team and are in the **Planning Game** phase. Stories have been received from the customer for the iteration and have estimates. You are tasking out the stories and estimating the individual tasks.

Individuals take responsibility for individual tasks. Tasks are carried out by a pair (**Pair Programming**), but one person has responsibility. Commitment to the task is required for successful completion. You need commitment by the team's individuals to the work so that they are excited about what they do. *More important, The team needs accurate estimates of the time needed to perform the task.*

- People are individuals. Some are optimistic estimators, some pessimistic, though not everyone understands her own abilities.
- On the other hand, if someone is interested in a task it is likely that she knows something about it and is therefore the best person to provide an estimate, both because she knows the work and because she knows herself.
- Fine-grained estimation like this can seem chaotic. You want a **Social Tracker**.
- It will be hard to gain appropriate commitment if task estimates are assigned by others. That practice can also lead to the hardest tasks not finding any takers.

***Therefore**, when you pick up a task card to implement it, re-estimate the task. If it doesn't agree with any existing estimate, inform the Tracker.*

- Straightforward tasks are estimated quickly and accurately. More difficult situations take time and effort. Sometimes you must **Spike** to learn or split stories into tasks.
- But note that estimates are just that: estimates. They aren't perfect. They aren't perfectly accurate. Don't spend time estimating that you could use to **Deliver Customer Value**.
- The story estimate done earlier for the **Planning Game** might be rather different from the aggregate of its task estimates. This is because the story was estimated by the team and is something of a compromise. When a task is about to be performed, the person who will do it is in a position to give the best estimate of the required effort. If the total task estimate is less than the total story estimate the team can ask for more work then or later. However, if the sum of task estimates is higher than that for stories you must go back to the customer and have some stories dropped from the iteration. Not doing this is asking for trouble. On the other hand, if things go well you can always **Ask for More**.

Stories, unlike tasks, are estimated by the team as a whole, and so the team owns those estimates. If the granularity of the estimates is small the stories may not need to be further broken into tasks. Often, however, this is not the case and the team "tasks out" the stories. The resulting tasks are then estimated by the individuals who chose to perform them. As a rule of thumb, if an iteration is 10 days, then a story estimate of more than 3 ideal (person) days may be too big. Sometimes the customer can break this story into parts, which will aid her in scheduling (as the parts may not have equal value). If the customer cannot do this, then the team should task it out so that as much parallel effort as possible is achieved. In general, a story whose estimate is a significant fraction of the average individual velocity is at risk of not actually fitting into the iteration. The "rule of three" stated above

is a bit conservative, but you would rather be safe. Experience will give you a better number at which you start to worry.

(Set Aside) Enough Time

You are a developer in the **Planning Game** of an agile project. You are estimating a story or task So, you are the **Implementer** who **Estimates** the **Task**.

When you build something, to do it successfully, you must test it properly. You must understand it properly and document it. In agile development, all this is critical due to the highly iterative nature of the process and the changing requirements. *But talking to the customer, testing, and infrastructure work take time.*

- Many tasks have necessary prerequisites that haven't been explicitly stated in any story. If they must be done, they must be estimated and the customer involved.
- Testing, for example, will take you a significant fraction of the time it takes to build a task. You may write more code for the tests than is needed for the task itself.
- Testing in XP is not an extra task. It is part of every task.
- **Documentation is** also **Just Another Task** that must be done for every story.
- You need to talk to the customer to refine the few sentences on the story card into a real understanding of the underlying requirement. This takes time.
- You may need to **Spike** to understand the requirement.
- Refactoring to improve the code and make a story easier to build takes time.
- However, you will be tempted to estimate only the obvious things in the press of time.
- Slight overestimates are easier to live with in time-boxed development than underestimates.
- And you want to plan so that keeping the **Sacred Schedule** will be possible.

Therefore*, include in your estimates of stories and tasks the time needed to for* all *the work that supports the story. This includes, for example, time to test the feature and document it.*

- Some things, like testing, recur in every story. Some things are unique.
- It can be hard to think of everything, of course. When you err, you may need to re-estimate. This may cause you to go back to the customer to drop work from an iteration. Customers don't like it when this occurs.
- If you find your team underestimating stories, be sure to address this specifically in the **Retrospectives**.

Testing stories may require you to write some test fixtures for a tool like FIT or FitNesse so that the customer can write the actual tests. Unit testing in JUnit requires writing the actual tests. Include all of this in your estimates. Keep records on how effective you are at this estimation by keeping a record of velocity in your **Project Diary**. These records can help you improve if you refer to them when you next have to make an estimate. Flexible Velocity sometimes works as a stopgap when you fail to set aside Enough Time.

The estimate includes the time that will be required for any activity that must be completed to successfully build the story. This might include time to set up some required infrastructure, though it might be best to separate that out as a separate story. Certainly time required to refactor existing code to aid implementation of a new story must be included in that story's estimate. Note, however, this implies that the story estimate must change over time, and that estimates done long ago have less certainty than they did when

made. For this reason we **Re-estimate Periodically**.

Testing tools: http://fitnesse.org, http://junit.org, http://fit.c2.com

Team Owns Individual Velocities

You are a member of an agile team in the development phase, or you are a manager overseeing an agile team. Your **Implementers Estimate Tasks**, and you set aside **Enough Time** for the tasks.

When a developer takes responsibility for a task she estimates the (ideal) time to complete the task. Enough tasks are taken and estimated to equal the developer's individual velocity for the iteration. *People estimate differently.*

- Unless a task is very small and well defined, estimates will be inaccurate.
- Managers like to manage. They think of that as their job. They also like to use available numbers to help them. Some numbers, however, may mislead.
- Estimates affect velocity and velocity affects estimates. Both are largely independent of how much actual work can be accomplished. Individual velocity is just a measure of how much work a person can do in their own personal point scale.
- The tracker needs to know the individual velocities to project the likelihood of successful completion of the iteration and to plan the next.
- An optimistic estimator will set low estimates, ignoring that things can go wrong because they usually don't. A pessimistic estimator will be aware of everything that could possibly go wrong and will give long estimates. Both will likely be wrong. This inaccuracy is corrected for in the velocities. The optimistic estimator won't get as much done as she thinks. The pessimistic one will get more done than his worst-case scenario. So the optimistic estimator will tend to have lower velocities than a pessimistic one even though they may do the same work.

***Therefore**, let the team own the individual velocities.* As manager, you must ignore individual velocities as they don't measure quantity or quality of work done. Individuals keep their own velocity in their personal project book. These are shared with the tracker to plan an iteration.

To emphasize this consider the situation in which there is a task card on the table that could be done by either John or Sue. John picks it up and estimates 3, and his velocity is 3. If he gets the task done this cycle all is well. Suppose, instead, that Sue picks up the card and estimates 7. If her velocity is 7 and she does the work in the iteration we are equally pleased. But John's velocity in the next cycle is still 3 and Sue's is still 7, though they do equal work.

- Note that keeping individual velocity is a difficult practice to maintain, as it requires discipline. The individual velocities are needed, however, so that each developer knows how much work they can reasonably commit to in an iteration. (record of velocity in the **Project Diary**)
- Like the team velocity, the individual uses **Yesterday's Weather** to determine their velocity for the next iteration.
- If an individual picks up too much work, it leaves the team at risk of not completing the iteration.
- From the individual's perspective, knowing your personal velocity and being able to accurately and consistently estimate how long it will take you to perform tasks is a skill that will pay you dividends. It is difficult for someone to coerce you into an impossible schedule if you have a reputation for accurate estimation.

- The team's velocity on the other hand is worth tracking. It is an important planning and projection tool.

Spike

You are a developer on an agile team, perhaps the **Implementer** who **Estimates Tasks**. During the **Planning Game** you don't understand how to estimate a story or task or you don't know how to implement it.

You will often get a story or task that you don't know how to build or even how to estimate accurately. Often the best solution here is to have the customer break the story into smaller parts since it is easier to estimate small things than large. But you can also take a few minutes to a few hours to do exploratory development to see what the problems are. *Wild guessing can be dangerous.*

- Before you can estimate something you have to know how to do it. You need to estimate **Enough Time** for this story.
- Usually you can learn something by building a simple prototype or by drawing out a design or playing with CRC cards or the like.
- Sometimes you need to choose between somewhat nonequivalent alternatives. The tradeoffs between them may be critical to success.
- Experimentation takes time and effort.
- Building the wrong thing is expensive.
- Experimental programming is fun. You can waste a lot of time doing it. But you want to **DTSTTCPW** and **Deliver Value**.

Therefore*, when you don't know how to do something build some throwaway code or create a throwaway design to investigate the issue.* This is called a "spike." You can also spike to help choose between alternatives. Sometimes the customer gets involved in choosing the solution as it affects cost and quality. Sometimes what you learn in a spike can give the customer ideas that will cause her to re-steer the project, in fact.

- Spikes are thrown away at the end and not incorporated into the build. For this reason it is sometimes appropriate for individuals (rather than pairs) to spike. It is also sometimes acceptable for a larger group to spike.
- Spikes are small and fast and probably dirty. They are not prototypes.
- If you have two or more approximately equal possible solutions, don't waste a spike on choosing one. Just pick one and get on with it. You aren't optimizing here.
- Build *just enough* in the spike to learn the needed lesson. Don't try to build a complete thing. That would be wasting resources.
- Time to spike is included in the estimates, though to some extent, it is also included in the difference between ideal time and real time (the velocity). But when you recognize a spike is needed, include it in the estimate.
- Spikes cost you time and money, of course, but the team needs to learn somehow.
- The daily **Stand Up Meeting** can often be used to send someone off on a spike. Make sure you bring the knowledge back to the team.

Promiscuous Programming

You are a developer on an agile team. You have responsibility for one or more tasks in this iteration. You are about to begin a task or subtask and are looking for a partner for **Pair Programming**.

The practices of XP are synergistic. They cover the goals of any development project, but often in a different way. Common code ownership is an important practice. It helps you avoid a lot of documentation, as the team generally knows everything about the code base. *No one person should become a knowledge bottleneck with respect to any aspect of the project or its artifacts*. This implies that what one person knows, others know as well.

- Sometimes you feel more comfortable with some team members than others and you tend to choose those you favor when you pair up.
- Some team members may be considered “newbies” or “outsiders” for some reason and might not be chosen as often or might be left to pair with each other. This can cause an “us” vs. “them schism within the team.
- Sometimes a guru is the one every team members wants to work with to ensure that deadlines are met. Or you are overawed by the guru and avoid him/her.
-
- You want to build trust and knowledge in the team.
- You want to be effective and have the team also be effective.
- You don't want to be left behind. But others have skills you don't.
- You want to improve your own skill and you want those with less to get up to speed.
- You don't want to be bored.
- The work environment can improve if it has some social aspects that don't interfere with the work.

***Therefore**, switch partners for every task at least.* Spread the knowledge of the code and of the programmers throughout the team. Pair with everyone repeatedly, even if it takes **High Discipline**. Track who you pair with in your **Project Diary**. Note the times as well.

- Knowledge will spread through the team.
- You become a generalist, rather than a specialist. This is a trade-off, of course, but agility depends on it.
- Often not everyone is especially compatible with everyone else. Work to overcome this. If management encourages everyone to **Be Human** it can help.
- If your team tends to form permanent cliques, then XP may not be your best methodology.
- You want to encourage practices by which the least skilled among you can increase their skills rapidly and become more productive. Pairing with experts aids in this. Pairing with a wide variety of people with different skills aids in this.
- Oddly enough there is some evidence that pairing with people unlike yourself (life history, race, sex, etc) will teach you things you couldn't easily otherwise learn. The work of diverse teams has been shown to be higher quality than that of homogeneous teams. See Surowiecki [13].
- There is some evidence that switching every hour is beneficial to the project. See Belshee [3]

Cards and Whiteboards

You are any member of an agile team that needs to create some artifact other than code. Perhaps this is a planning or tracking artifact. You need to keep track of where you are now and where you are going in the short term.

Agile projects are very fluid. *Things change all the time. People need to see what has changed*.

- Sophisticated planning tools can produce nice reports and can be good for archiving information. But you need to learn them and agree on their use.
- If you use sophisticated tools to keep track of things you will see two unfortunate things occur. First it is natural to be reluctant to throw away things that you have committed effort to creating, but this creational effort is required by sophisticated tracking tools. Second it is harder to get a team around a screen than to get them around a table picking up and rearranging cards.
- Your planning horizon is the iteration and the release. Things will change between iterations and so the current view of the release will change as you approach its completion.
- You want to encourage gesturing with the planning artifacts to make a point, scribbling quick notes on them, etc.
- If you don't keep **Everything Visible**, you wont know if you are on track without expending additional efforts needlessly.

***Therefore**, make the main planning and tracking tools simple, tactile, visible, and non-technological.* Paper is good. Stories are written on small cards, not put into some tool. You want to be able to throw away mistakes without guilt. Cards are tacked to the wall. Whiteboards are equally good, as they can be made to easily change as the situation changes. Make the workspace itself informative [1] so that anyone in the room can immediately see progress and problems. Make sure that housekeeping knows not to erase your whiteboards, of course.

- If your team is distributed then it may be necessary to use electronic means, but you will pay a cost for this. Keep such electronic documents in a common electronic workspace and make sure everyone can edit these: common ownership of all artifacts.
- The customer owns the story cards, by the way. They must be tracked; by name or number.
- Story cards and other important artifacts should not leave the workspace.
- Yes, things can get lost. This is a risk. You may want a backup electronic copy. Make sure it is clear which is the original and which is just backup.
- Note that Scrum tends to use spreadsheets, but is careful to make them visible, accessible, and modifiable.
- As your team grows, this will be harder to do. Do what you must, but keep **Everything Visible**. The technological solution will cost you something, however, so make sure you watch for entropy in the **Retrospectives**.

The author was recently coaching a team in which a very helpful tracking tool was a hand-drawn thermometer on a whiteboard. It measured the "temperature of the build" and represented the probability that the iteration wouldn't finish successfully. This also kept

Everything Visible.

Documentation is Just Another Task

You are a member of an agile team. You recognize that internal or external documentation is required. You also recognize that most traditional documentation isn't needed here. You will need documentation, both user level and system level. You have not gathered all the requirements up front, but you need to document requirements as well as decisions made.

Some people believe that documentation isn't done on agile projects. This is not true, but the documentation is different. You must do user-level documentation, of course, on any project. It may be necessary to have a separate view of the project readable by non-coders. In agile development these are not prepared in advance, as the project direction will change. They are not prepared after the fact as people forget what was done. *You must prepare all* necessary *documentation and you must keep it in sync with the evolving project even as it changes*.

- The project requirements are changing constantly.
- If you leave documentation to the end, it won't get done, or will get done poorly.
- If documentation is not consistent with what you have built it is worse than worthless.
- Your build unit is the story. Your horizon is the iteration.
- You need **Enough Time** to build the story completely. Otherwise documentation will become the orphan child of the project.
- Agile teams themselves need less documentation than other teams.

Therefore*, treat documentation creation and updating as just another task.* Estimate it and refactor it as you would a programming task. A small project can keep most of its documentation on cards and on the whiteboards. A large project can use documentation specialists on the team who work with other developers (pairing) in the usual way. A smaller team can treat documentation as a role, just like testing is. The documentation tasks may follow slightly in time the other development tasks, but not by more than a few days. User level documentation must be brought to sync at the release points.

- When a project changes direction, the documentation, like the code, must change or be discarded. This means you spent money that you might not have had to spend if you knew better earlier. It is a sunk cost, however. Move on in the best direction you can with the knowledge you now have.
- Your project isn't driven by the documentation prepared in advance. You have the customer to steer and the code to tell you what you have done.
- There is implied coordination here, and that requires communication. Those writing documentation work with the customer, as does anyone on the team.
- Code is still the best documentation for the team, assuming it is clean and well-factored. Others need a different and/or higher level view.

In an agile project the code itself is considered to be the key piece of documentation. Make sure it fits that role by always writing the code in the clearest possible way. While this is important in any project, it is essential in an agile project.

In general, remember that the team must include *all* necessary skills to build a quality product. This includes documentation skills. Note that acceptance tests are a form of executable documentation.

For more on Agile Documentation, see Rueping [11].

Question Implies Acceptance Test

You are customer on an agile team. Someone has just asked you a question about the meaning of a story. You want your answer to be faithfully captured by the development process.

If the answers to questions aren't captured reliably and accurately, the answer could get lost or misinterpreted. But the target is moving and the project is continuously re-aimed via customer steering.

- Developers ask questions of the customer continuously throughout the development process. There are a lot of questions and answers.
- Answers to some questions invalidate answers to questions previously asked.
- But it takes time and effort to capture everything. Sometimes the simplest things require the most discipline, though more complicated things require the most effort.
- If requirements become inconsistent you need the inconsistencies to show up dramatically and early.
- The project may never develop sufficient traditional requirements documentation to drive a traditional black box test.

***Therefore**, whenever you, the customer, answer a question on a story or task, immediately write an acceptance test that will verify that the answer is codified in the resulting application.* The best way is to make this an **Executable Test**, but in some cases the customer will need to resort to a special card, a **Test Card**. Then a programmer can generate a test from the card. The **Test Card** should stay with the story that generated the question.

An absolute requirement for successful agile development is an adequate set of acceptance tests so that both customers and developers can agree on the target and when it is reached.

If your organization has a Quality Assurance department they can be helpful in showing the customer how to build the acceptance tests as you go. A testing expert on the team is a real asset.

- Note that answers to questions can also be captured in unit tests in many cases. But unit tests get refactored along with the code and are owned by the developers, not the customer. There is some danger of losing an answer if they are only captured in unit tests.
- This practices takes time and discipline. It may require that someone make an **Executable Test** to capture this answer. If your acceptance testing framework is really solid, the customer may be able to do it himself.

Acceptance tests are under the control of the customer or product owner. She may or may not be comfortable writing the tests directly and may need constant assistance of a team member to formulate these in an executable way. Tools such as FIT and FitNesse can help if the team can work out a way to express the tests in tabular form. Most people with business skills are comfortable thinking in spreadsheets and such tools can then be used to directly capture the test requirements.

For more on acceptance testing see Mugridge and Cunningham [10].

Re-Estimate Periodically

You are a developer on an agile team. You have a bit of time available in the development phase.

Normally you estimate stories long before they are built. But you do so making certain assumptions. *These assumptions change and so estimates become obsolete as the application gets built and the code base changes.*

- What you originally thought to be easy may now be hard to integrate into the code base.
- What you originally thought to be hard may now be easy as you have support for it already in the code.
- You may know more than you did when you first estimated the story. It may have a different meaning now than it did earlier.
- The customer needs good estimates of stories to balance the cost against the current value. These are used for both long (project) and short (iteration, release) term planning.
- As always, you need **Enough Time**, but this changes as you **Refactor**.

Therefore, *in every iteration, take some time to re-estimate a few of the stories that the customer believes to be high value*. Do this only for stories with estimates that may be somewhat dated. It will be easier to know that you need to do this if you keep **Everything Visible**.

- Sometimes people have a bit of time because their forward progress is stopped while waiting for another task to complete, of they have finished their tasks and there are none left that will fit in the remaining time. This time can be productively spent spiking and re-estimating.
- The estimates, combined with the team velocity, are the basis for long-term projection for project management. The average story estimate is an important projection tool as long as it is reasonably accurate.
- Failing to have estimates that accurately reflect the current understanding of the project will greatly complicate the **Planning Game** and make planning a long and arduous task, rather than the half-day or so per month that it should be. Do just enough of this to smooth the next **Planning Game**. The customer can guide you here, by indicating the likely stories in the near future.
- But don't agonize over the estimates. If you aren't sure, you can **Spike** now or later and you can always make a consciously high estimate to cover uncertainty. Another strategy when estimation is hard is to split stories.
- If you find you seldom do this and the Planning Game often bogs down for poor/outdated estimates, you need to deal with the issue in the next **Retrospective** and set a **Flexible Velocity**.

For more on agile estimation, see Cohn [4].

Flexible Velocity

You are a **Social Tracker** for an agile team. The team recognizes that there is work to do that is not covered by the stories. You are in the **Planning Game** working on the next iteration.

Sometimes the stories to be included in an iteration have implications that are not obvious to the customer. They may require **Refactoring** or infrastructure development. *Work may need to be done that is not covered in any of the stories*.

- Setting an appropriate velocity for the iteration is the key to success. If it is too high, the developers will have to drop work in the iteration, making everyone unhappy.
- We want everything stated in the stories. All necessary work is estimated in ideal developer days ("sunny days").
- In practice this won't always happen. You aren't perfect.
- You need to do some things to support current stories (not future possibilities). These extra tasks were not captured in the estimates, though they should have been.
- The velocity needs to accurately estimate what the team can reasonably do in the iteration.
- As always, it is easier and more satisfying to go back to the customer for more work than to have to go and ask what should be dropped because not all can be completed.

Therefore, *adjust the velocity downwards for an iteration in which you know that there is work to be done that isn't included in any of the stories*.

This extra work can be refactoring the existing code since the stories might have been estimated at an earlier time and the current state of the code base implies refactoring. Alternatively (preferably) the story estimate can be updated to include this refactoring.

Some teams use *developer stories* to cover this situation. They are estimated and scheduled in the normal way, but they are inserted into the mix *just in time* by the developers to support a customer story. Don't use them to speculate what might be coming in the future and build code speculatively. **DTSTTCPW**, of course. But there will likely be arguments between the customer and the developers over the priority, and even the need for, developer stories as they aren't well understood by most customers.

- Note that in XP we do the simplest thing that could possibly work. This does not mean we hack. It does mean that we do not build the most general solution to every problem when we first encounter it. It is the second or third occurrence of a problem that pays for the general solution. Since you don’t know the order in which the stories will be built when you first estimate them, it is hard to write estimates that include this time to refactor. This is one reason why velocity stays at a fraction of available time, of course, but it also sometimes requires adjustments to the velocity of an iteration.
- If you set aside **Enough Time** for each story then the need for this should not occur. This is a temporary stopgap. Discuss the problem in your next **Retrospective**.
- Be aware that the customer won't like it when this happens. The long-term solution

is to try to get more complete estimates for the stories and their implications.
- Management will need to merge infrastructure requirements smoothly into the development. Make sure the required equipment does arrive in time and that you know who will set it up and check it out.

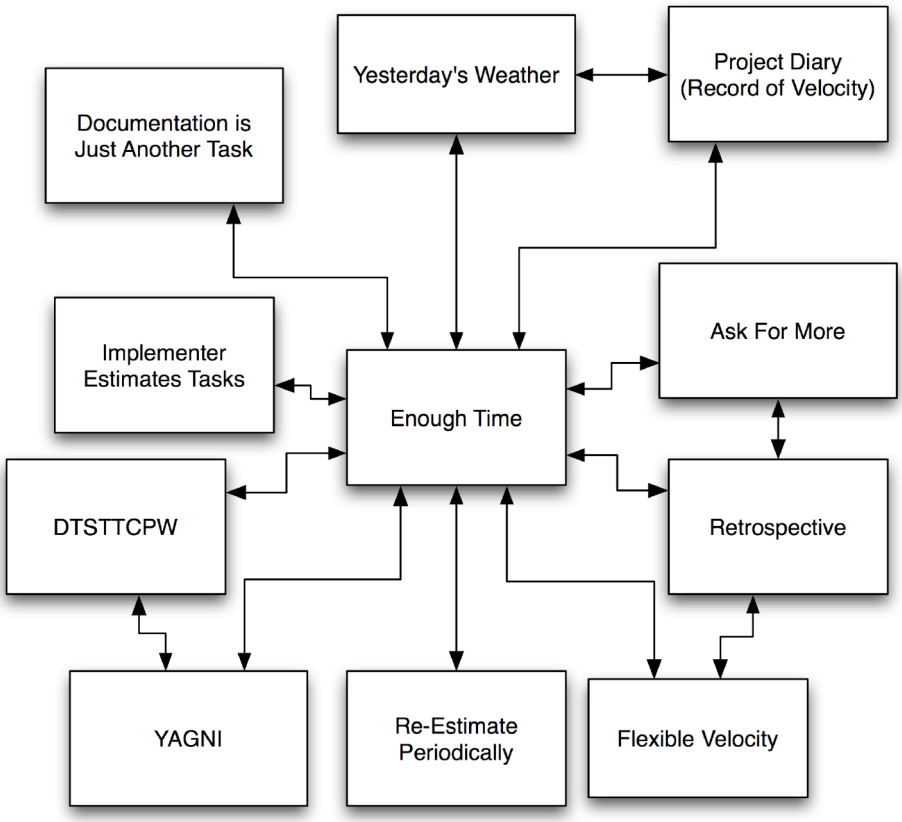
This extra work can include necessary infrastructure work needed to support a story. On a recent project, the velocity stayed at about one-third of available time (the organization has frequent required meetings), but hardware infrastructure was poorly planned and didn't get accounted for in story estimates. Velocity had to be reduced to get the machines up and running.

Clusters of Patterns of Agile Practice

In any pattern language there are clusters of patterns that work together and reinforce one another. In particular, no single pattern is likely to completely resolve all of its forces. Therefore, other, smaller scale, patterns are applied to help resolve the remaining forces.

We have identified a number of clusters that may help explain the synergy between the practices. Here is one that contains many of the patterns in this collection and is centered on a key pattern.

Enough Time Cluster (Figure 1)



The central pattern in this group both reinforces and is reinforced by the other patterns here. For example, we need **Enough Time** for **Documentation**. On the other hand, the **Implementer Estimates Tasks** precisely to give us **Enough Time**. Likewise, in finding **Enough Time** to do rich and complex things we need to **DTSTTCPW** to achieve it or we will not be able to end the iteration successfully. **Retrospectives** help us retarget our practices when we find difficulties generally, and especially with time. And when we have set aside too much time, we can and do **Ask for More**.

Note that there are some complex interconnections here, and also note that the patterns seemingly at the periphery are themselves richly connected to other patterns in this language.

Acknowledgement.

Linda Rising was the most able shepherdess of these patterns for EuroPLoP 2006. As always her advice is helpful, drawing as it does on her deep knowledge of both patterns and topic. I thank her profusely.

Thumbnails and Acronyms: This section includes short descriptions of all the patterns we have identified to date, including the ones detailed in this paper. .

Acceptance Tests. Create a suite of Executable Tests that will be sufficient for the customer to accept the work. They are under control of the customer.

Ask for More. When you know you will have extra time within an iteration, ask the customer for more work**.**

Be Human. Provide a humane workspace to maximize productivity.

Best Effort. The contract is not for features delivered on a given date. You want best effort and full communication.

Bug Generates Test. When a bug appears in code, write a set of tests that will only pass when it is corrected.

Cards and Whiteboards. Things change too frequently to depend on elaborate documentation mechanisms.

Coding Standards. Everyone shares the same coding look and feel.

Collective Responsibility. The team shares responsibility and rewards for all tasks.

Collective Ownership. The team as a whole owns all of the created artifacts, especially the code.

Constant Refactoring. The structure of the code is continuously improved to take account of all stories built to date.

Continuous Integration. Every task is integrated at completion and all unit tests are made to pass.

Customer Checks-Off Tasks. Only the customer knows when something is done.

Customer Obtains Consensus. The customer role is responsible for obtaining consensus among the stakeholders.

Deliver (Customer) Value. Building things may be fun or not, but don't lose track of the real reason we are doing this.

Documentation is Just Another Task. Every story requires some kind of documentation. If it must be extensive, include it in estimates.

DTSTTCPW. Do the Simplest Thing that Could Possibly Work. Build the code to implement the story and nothing more. Pay for generality only when you know you need it.

Easy Does It. As a customer, don't push too hard. It frustrates everyone. If you push too hard and "win," you lose if the iteration doesn’t complete successfully.

Effective Coach. A novice team depends fundamentally on a coach (ScrumMaster) to keep you to the discipline and help you see opportunities and problems.

End To End. The first release is an end to end version of the product.

Enough Time. Estimates must include everything necessary for a story.

Everything Visible. Whiteboards, note boards, etc., in the team space need to have enough graphically displayed information that anyone can immediately see the progress of the current iteration as well as any bottlenecks. When you get in trouble the **Retrospective** needs to see what happened and why.

Executable Tests. Tests are run so frequently they must be executable.

Flexible Velocity. Use velocity to allow for needed work that is not in the stories. But learn to get it into the stories.

Full Communication. The developers keep the customer apprised always of opportunities, costs, difficulties, etc. The customer keeps the developers in the loop on the business needs and thinking that may affect future directions.

Grow Up. Start with a small team and grow it to the required size by adding a few developers at iteration points. The other practices enable this: **Promiscuous Programming**…

Guiding Metaphor (Topos). Develop a guiding metaphor or story for the project that guides people as to the general direction.

High Discipline. No methodology will succeed if you don't actually do its practices faithfully. On the other hand, make sure they are the right practices or deal with the issue in a **Retrospective**.

High Value First. Customer selects highest value features at every point.

Implementer Estimates Tasks. Tasks are best estimated by the person who will do the work.

Individual Customer Budgets. When customer representatives can't come to a common understanding of priorities, they may need individual budgets of team resources.

Infrastructure. Before the project begins make sure the basic build, test, integrate, deploy infrastructure is in place.

Once and Only Once. [2] Refactor code so that everything is said only once. But pay for generality only when you must.

Onsite Customer. The customer works in the team's room along with the rest of the **Whole Team**. Communication distance is very expensive.

Our Space. The **Whole Team** works together in an open workspace to optimize

communication.

Pair Programming. No code is committed to the code base unless it is written by a pair.

Promiscuous Programming. Spread the knowledge of the project amongst the team members.

Planning Game. Once each iteration (every two weeks, say) the team spends time planning the iteration, including what stories will be immediately built. See the literature as this is a highly disciplined planning exercise.

Project Diary. Each developer keeps a bound book for the project. It is private to the individual and contains things like estimates vs. actuals on stories built, who you paired with, ideas for the next Retrospective, etc.

Question Implies Acceptance Test. When the customer answers a question from the developers, she captures the answer in an acceptance test.

Re-estimate Periodically. Things change and estimates become obsolete.

Retrospective. Periodically hold a retrospective [8] of the team's practices.

Sacred Schedule. Time never slips in agile development. Features are the dependent variable.

Sheltering Manager. A new team will depend on some shelter from those in the organization who don't readily accept change.

Simple Design. Design only for the current stories. Simple logic, minimal generality, pass the tests.

Small Releases. Software is released on short cycles, say monthly.

Social Tracker. The tracker must know how everyone is doing.

Spike. Do quick prototypes to learn how to build or estimate something.

Stand Up Meeting. (Daily Scrum) Fifteen minutes every day, to keep everyone on the same page.

Sustainable Pace (40 hour week). Pace the team for the long haul, not a sprint. You want everyone working in top form all the time.

Team Continuity. Management commits to keeping the team together throughout the project. Team members make a similar commitment.

Team Owns Individual Velocities. Individual estimates are too variable to be a management tool.

Test First. [1] No code without a failing test.

Test Card. If the customer cannot write executable tests herself, then she creates Test Cards in answer to each question. The card specifies an acceptance test that will then be written by the implementer of the story.

Train Everyone. Initial training includes everyone, including customers and management.

Whole Team. The team includes everyone with an essential skill. In particular, it includes the customer as a full team member.

YAGNI. You Ain't Gonna Need it. Don't anticipate what might not occur. Don’t scaffold speculatively.

Yesterday's Weather. The velocity of the next iteration is exactly the work successfully completed in the previous one. Of course this assumes that the time and personnel are fixed.

References

[1] Beck, *Extreme Programming Explained: 2ed*, Addison-Wesley, 2004
[2] Beck, *Smalltalk Best Practice Patterns,* Prentice Hall, 1996
[3] Belshee, Promiscuous Pairing and Beginner's Mind: Embrace Inexperience, http://www.agile2005.org/XR4.pdf
[4] Mike Cohn, Agile Estimating and Planning (Robert C. Martin Series) Prentice Hall, 2005
[5] Coplien, Harrison, *Patterns for Agile Software Development*, Prentice Hall, 2004
[6] Jackson, Michael A. *Principles of Program Design*. Academic Press, London and New York, 1975
[7] Jeffries, Anderson, Hendrickson, Extreme Programming Installed, Addison-Wesley, 2001
[8] Kerth, Norm, *Project Retrospectives: A Handbook for Team Reviews*, Dorset House, 2001
[9] Manns, Rising, *Fearless Change*, Addison-Wesley, 2004
[10] Mugridge and Cunningham, Fit for Developing Software : Framework for Integrated Tests, Prentice Hall, 2005
[11] Rueping, Agile Documentation : A Pattern Guide to Producing Lightweight Documents for Software Projects, Wiley, 2003
[12] Schwaber, Beedle, Agile Software Development with Scrum, Prentice Hall, 2002
[13] Surowiecki, The Wisdom of Crowds, Anchor, 2005