Patterns for Agile Development Practice
Part 3

Joseph Bergin
Pace University
jbergin@pace.edu
(Version 3)

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*[1]. 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 2004-2006 on the same topic ([3] and [4]). This paper contains some of the standard practices of Extreme Programming as detailed in [1].

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 sixty or so patterns developed so far. Most of the known patterns are 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.

Stand Up Meeting

You are an agile team. It is 8am on any day of the development phase.

Extreme Programming works best when everyone knows what is going on. New information comes to the customer and to the developers constantly. Some of this knowledge affects what others must do, but *the person receiving the information may not know the consequences that it implies for others on the team*.

- Short meetings can bring everyone up to speed and let everyone know the current level of risk.
- Long meetings waste everyone's time and are boring for most participants.
- You need to know where you are and where you are headed in the short term in an agile team.
- Changes in the business may make some things especially important or unimportant in the short term.
- What one person on an agile team knows, others should know as well.
- You don't need to solve every problem as a team.
- A potential danger of frequent meetings (even short ones) is that the customer will try to over-steer the project.

***Therefore**, hold a 15 minute stand-up meeting every day.* No one sits unless they are pregnant or otherwise have a physical need. If you don't stand up, the meeting will last too long.

The tracker is an important part of the meeting he or she gives a quick report. If there are problems, say where.

- Issues are not solved at this meeting, though individuals may be assigned to solve them. The coach can keep the meeting moving. Everyone says what they are working on at the moment and if there are any problems on the horizon that they see. In SCRUM the rule is: say what you worked on yesterday, say what you will work on today, say what obstacles you see to your success.
- One possible danger of the stand up meetings is that some customers will try to use them to micro-steer the build. The **Effective Coach** can help guard against this. The customer changes direction only at the planning game points. The customer may, of course, drop work from an iteration as soon as it is known that changing conditions have made it obsolete. New work may then, perhaps, be able to be added when the team finishes the rest of the work in the iteration and has more time available. The customer doesn't add or change work in an iteration and cannot look on the stand-up as an opportunity to do so.
- Don't let the short meeting get long.
- Don't let the nit pickers pick their nits. Coaches take notice.
- If it becomes just a ritual, then try to have an on-demand stand-up as needed. Evaluate the effectiveness of this in your periodic **Retrospectives**.

One useful practice at the meeting is to "take the temperature" of the iteration. The coach or tracker can ask if anyone thinks the iteration won't be completed successfully. A graphic

(Information Radiator) can be drawn on the white board to represent the degree of risk of not completing all the stories successfully.

This practice was adopted from SCRUM [14], and has become a standard XP practice.

Test First

You are a developer in the development phase of a project. You are beginning the development of a task or subtask.

You think differently when writing tests and when writing code. Testing requires that you take a broader view. Coding requires a microscopic view. *Tests written after a task is coded too often test what was done, not what is wanted.*

- Testing takes time.
- You are pretty confident you know what to do for a task.
- But things will change. Changes in future may require changes here also.
- You want to build what the story says to build, but ONLY that. You want to **DTSTTCPW** (Do the Simplest Thing that Could Possibly Work).
- You feel some pressure, of course, to "just get on with it."
- Programming is creative work. It requires thinking as well as coding.
- You can design while testing. You can test while designing.

Therefore*, write your tests for a feature before you build the feature.* If the tests pass you are done. Write no code without a failing test.

- If your organization has a testing group that will develop its own tests from some requirements documents (here just story cards) then this can seem like wasted effort. If you can get a member of the test group on your team, you will probably be able to write better tests and you might be able to feed your tests into their process. In other words, your tests might help them. But their tests, coming late, won't help you.
- The purpose of unit tests is not the same as that of acceptance tests.
- It will take the team a lot of time and effort to get comfortable with this practice. It also requires initial setup before the project begins. The coach can help with the former.

There are three ways to fail when you write tests after the code.

(a) The press of time will push you away from writing the tests.
(b) You will be tempted to get too creative when programming, thinking that some extension could be easily added – so why not. If what you do isn't needed, it is a waste of the customer's resources, complicates the code, and needs to be maintained in future.
(c) You will spend time in the coding process doing design without capturing your decisions in tests. You aren't saving time, as the thinking is the bulk of the time in any case.

Unit tests have all of the benefits of any regression test system. **Executable Tests** give you confidence that you got it right the first time, but more importantly, they let you refactor with confidence. They also tell you immediately that a new requirement is inconsistent with an old one.

Anecdote: The author did once sit down to build a feature in a project, wrote a test for the feature, and it passed. In fact the feature was implemented as an unintended side effect of other features. Don't count on this happening often, though.

Special note on testing: This author believes (with only anecdotal evidence) that when correctly done, testing doesn't cost you time, but speeds you up. This implies that you are using pair programming and test driven development AND you are doing your thinking and planning (designing) for a story while strapped into the test harness. You must think about the structure of your solution. You must design classes and methods to solve the problem. If you do this planning and thinking with JUnit (or equivalent) running you will capture all of your decisions immediately as tests. The planning needs to be done anyway. The tests serve then as notes about your decisions as well as the tests that will eventually prove your code.

Once and Only Once

You are a developer on an agile team. Development is proceeding. You are coding a task.

Often when you are coding in an iterative environment, you notice that you are writing a piece of code that you have written before. If your methods are small and simple this will be mostly at the level of method bodies, not switch cases. *Redundancy costs you in maintainability*, however. When the system changes in the future, all redundant copies must be brought into sync and the tools are not very helpful in finding the places that need update.

- You want simple code that is easy to write.
- You want good code that is easy to maintain.
- You want elegant code that is easy to modify.
- The above are often in conflict.
- Building code more elegant than necessary wastes money.
- You have tests to tell you it is correct. Your skill should tell you if it is ugly.

***Therefore**, when you refactor, bring the redundancies together using an appropriate object/functional design.*

- However, don't anticipate this need. Remember that it is the second (or third) use that pays for generality. You *may* need to revisit code that you are writing now for the first time, but you may not. Building in unnecessary generality at every chance is expensive and wasteful. Pay the price when you must, but only when you must.
- Recognizing this situation takes some practice, as does solving it. If you find it difficult to do this in some case, examine your overall coding practice. If your redundancy is at the level of switches it is harder to handle than if it is at the level of method bodies.
- **DTSTTCPW** is not an excuse to hack. Don't let your code go out of coherence. But don't pay for generality that may not be needed. In most cases **YAGNI**.

From Beck, Smalltalk Best Practice Patterns[2].

Like the "first rule of optimization", the first rule of generalization in an agile project is "Don't do it". The second rule, for experts, is "Don't do it, yet." (I learned this from Ledgard, but it is really due to Michael Jackson.)

There is a lot more to **Constant Refactoring** than Once and Only Once, but it is a good place to start.

Executable Tests

You are a member of an agile team. You must develop effective unit and acceptance tests.

Tests must be run often so that the project doesn't veer off into the woods. Manual testing is expensive and tedious.

- Both the developers and the customers depend on tests to know where they are.
- In an iterative project that is using test driven development, the tests must be exercised frequently. This is your proof you haven't broken something.
- At every change or addition to the code base, the tests need to be re-checked.
- Manual testing is tedious. So tedious that most people will ignore it and hope for the best. Manual tests are hard to execute and therefore are not run often. This can lead to too much development between runs of the test suite and then many problems making the tests run and fixing the bugs.
- You need tools and frameworks for executable tests, especially for user interface tests. You need machine cycles for their execution.
- There are a lot of tests.
- Some things the customer wants to specify are very difficult to specify in an executable way. But when done, the customer has confidence in the result.

***Therefore**, capture tests in a way in which they can be directly executed.* Unit tests can use something like JUnit. Acceptance tests can use something like FIT/FitNesse. The tests should be collected into suites that can be run all together.

- You need to be able to test at different levels of granularity. You should be able to test a single class, a single story, or the entire state of the application. This requires good tools. You need to be able to run *all* tests at every code commit point. You need to plan what happens when the test fail at a commit point. In some projects everything else halts until the tests pass. This practice is considered normal, not extreme.
- Get your test framework in place prior to your first iteration. Investigate specialized testing tools you may need at the very beginning and again whenever it becomes necessary. Don't underestimate the difficulty of finding and learning the right test framework for your particular tools and technology. Many of them are free, but have little support. Many can be found on Sourceforge.
- While it is possible to use robot driven test tools to test GUI parts of the application, these often take a long time to execute and may require user intervention. This gives additional incentive to make clean design breaks between model and view, so that the underlying business logic is all in the model and can be effectively tested independent of its interface. Then the GUI tests test only the look of the interface, not is behavior.
- The customer will need help in making acceptance tests automatic. Make this a task. Helping the customer formulate the acceptance tests is a role.

The coach and tracker can assure that the tests are written. The tracker can track and plot the number of tests and the number of passing acceptance tests using Information Radiators. The shape of these curves can tell you when you are starting to slow down development. This can

imply that costs are rising, that stories are getting harder to build, and especially that it is time for serious refactoring.

Continuous Integration

You are a developer on an agile team. You have just finished the development of a story or task and all of the unit tests pass.

Stories are broken into tasks. It is the tasks that are assumed and built by the developers. Unit tests are written at the task (and sub task) level. *If you don't integrate the task into the build you won't know early enough that you have a problem*.

- Many projects fail because they are built from parts that won't integrate at the end.
- Small assumptions made frequently by many people accumulate into large problems.
- Integration takes time. It often invalidates your assumptions.
- Small tasks are easier to integrate than large ones.
- But with many pairs working, integration can be a bottleneck.
- Task integration requires good CVS tools unless the team is very small.

***Therefore**, integrate every task into the build when all its unit tests pass.*

A code repository that helps you do this is essential. CVS has such capabilities. You can run (and must pass) all the tests against the code base before you commit, so that when you do, you know that nothing is broken. Since tasks are small, integration is more likely to be successful and when not, the problems more likely to be manageable than when you do "big bang" integration. You also learn earlier when you have a problem if you integrate frequently. But if you can't make all the tests pass, you can't commit the code.

- This is another job that requires discipline. The coach will be vigilant.
- Each pair of programmers may be integrating a task every day or so (more frequently in some teams). This implies discipline with respect to maintenance of the code base. An integration machine on which all integration tasks are carried out can help here.
- To emphasize: when you integrate, all the unit tests must pass. Not just the tests for this feature, *all* the tests. This implies, of course, that the tests are in the repository along with everything else and thus are accessible to everyone.
- You also need to run the acceptance tests. Those for the story just built must pass, of course, but you need a plan for what happens when a new story being integrated makes an acceptance test that was passing now fail. In many teams this is a show-stopper and all work halts until the situation is resolved.
- You may actually need two repositories for code. One contains the committed code against which all tests pass. The other contains work in progress and is put in a centralized place only to assure that backups get done and nothing gets lost if a machine crashes or is stolen.
- When you think you have finished a task, but nothing you do makes it integrate correctly, warn the tracker that there may be delays.

Social Tracker

You are tracker on an agile team. Development progresses.

The tracker needs fine-grained knowledge of the state of the project, but the knowledge is in individual heads. He or she is responsible for important guidance at the **Stand Up Meeting** every day. Most team members are mired in the detail of their own tasks to see the bigger picture.

- If the tracker doesn't talk to everyone frequently she won't know what is going on.
- The tracker has other tasks as well. Tracker is seldom a full time role.
- The tracker integrates information held by the developers, but needs to visit with them to gather it.

***Therefore**, the tracker spends five minutes or so with everyone at least twice a week.* She needs to know if the tasks being undertaken are likely to be completed and will compile individual and team velocities as well as the number of tests written and passing.

- The tracker also reports progress at the **Stand Up Meeting** and through visuals (Information Radiators) on the walls.
- As with many of the agile roles, it is best if this one rotates every iteration or so. This spreads knowledge and skill.
- Note, however, that each developer keeps track of her own velocity and progress. Tracking is not an external, or management task, but an individual information task.

Note: I hesitated to call this one Promiscuous Tracker.

Project Diary (Record of Velocity)

You are a developer on an agile team. You are building code to implement stories and tasks.

Individual velocity figures must be known and accurate. If you don't know how fast you can go, you won't know how much work you can pick up.

- The only way to learn to estimate well is to do it and to record a history of your estimates and the resulting actual times.
- The team estimates are based ultimately on individual estimates.
- Agile development can't function if estimates can't be depended on.
- Recording takes discipline. The coach can be a nag here. Every task should be recorded.
- With self-reflection a person can learn a lot about how they estimate: optimistic, pessimistic, or generally accurate. Being optimistic or pessimistic isn't necessarily bad, but it is useful to know how you compare with others.

***Therefore**, each developer keeps a bound book for the project in which estimates and resulting actual times are recorded for each task.* The book should say something about the nature of each task and the times need to be in the same (ideal) terms. To do this you must record the "ideal" time you spend on each task, not just the elapsed time.

- Your record book will help you become a better estimator over time if it is accurate, up to date, and consulted while you are estimating the next task or story.
- Estimation, like swimming, is not a skill you learn by thinking about it. You have to practice it. Initially your estimates will be terrible, but make them, just to give yourself a baseline. You won't improve your estimates unless you record them so that you can look back over how you estimated similar things in the past. The record can also tell you if you are a generally optimistic or a pessimistic estimator.
- The estimation book of a developer should be considered her property. The numbers in it are not useful for planning or evaluation of the employee. The tracker keeps a book for the team. The estimates there are the ones management needs for planning.

Note (This may be a pattern): Velocity is Relative
Velocity is the relationship between ideal and real time. Project and individual velocity is never the same between individuals or between projects. They differ widely depending on the individual, the organization, and the work. In some organizations, people normally spend half their time in meetings and on tasks other than the development tasks that are nominally their main job description. Therefore, don’t try to compare your velocity to that of another project, team, organization, or individual.

Customer Checks Off Tasks

You are customer on an agile development team. Developers have come to you with a "completed task" for your approval.

Tasks must be checked off only *when done.* Otherwise you will miss things.

- When a task is completed it is checked off (and a small celebration ensues.) However, if the developer checks of the task when she thinks she is done, there is room for disagreement.
- A task isn't done until the customer is satisfied.
- If you think it is done and it isn't, then you will find integration difficult.
- If you think it is done, but the customer is out of the loop you may drift toward assuming what the customer wants.
- It needs to be clear to all when a task is done.
- Done doesn't mean coded. It means coded, tested, integrated, documented, etc. **Whole Task**.

Therefore, when the developer is "done" with a task, she takes it to the customer for a demo. The customer will run any acceptance tests on the task (or write and run them if this has been neglected). *When the customer is satisfied, the customer checks off the task as done.*

- Note that if the developer thinks a task is done and the customer disagrees there may be a problem with understanding of the task. The customer can accept the task, but write new stories to correct it and schedule these in the usual way. The customer can also reject the task. If there is time in the current iteration to satisfy the customer, do so, otherwise you will need to discard the work and let the customer reschedule. This affects current velocity, of course. Don't count work not accepted in the computation of **Yesterday's Weather**.
- Avoid blame when tasks are not accepted. Blame buys you nothing. It matters not a bit if the customer or the developer is responsible for the "failure." What is important is that you move the project forward. Stories do that. Write stories and move forward.
- But work on communication, especially when you find that misunderstandings are frequent. This is more likely to occur when the customer is not physically present when questions arise, of course. The coach should help you with this. Work toward everyone taking responsibility for miscommunication. Customers and developers usually speak different languages, have different world views and constraints. All of this can lead to miscommunication that is no-one's fault, but everyone's responsibility to correct.
- Some claim that XP humanizes the work environment, especially with such practices as **Sustainable Pace**. The customer, however, may be overworked. She has such a central role in steering.

Customer Obtains Consensus

You are customer on an agile development team. Your project has lots of stakeholders. You want to be the main guide of the project team.

In many projects the "customer" is a complex beast. There may be many customers or constituencies outside the team that have a business interest in the project.

- The customer representative must speak as one for all of these individuals and groups.
- She must be empowered by the organization to make (or at least communicate) all of the business decisions that relate to the project.
- If the developers get conflicting answers to questions, chaos can ensue.
- The customer is responsible for keeping a coherent picture of the target. It must be shared (as you go along) with the developers.
- If the picture is incoherent, so will be the product.

***Therefore**, one of the primary tasks of the customer is to obtain consensus among all of these stakeholder groups on priorities and business value.*

- It is essential to the team that there is only one "mind" setting the direction. Otherwise, if questions are answered inconsistently then the code will become incoherent.
- There will be many questions and the team will need answers quickly if they are to continue their forward momentum. The customer with many stakeholders needs good lines of communication to all of them.
- Customer has a lot to do and must do it quickly but accurately.
- Occasionally this consensus is impossible. The customer's just can't agree. If this becomes a problem, see **Individual Customer Budgets**.
- Another option when customers cannot agree is to "just do something." Satisfy one of the stakeholders this iteration and then try again for consensus. This is risky, of course.

Note. This is a specialized pattern that may apply or not, depending on the number and alignment of the stakeholders.

Individual Customer Budgets

You are managing an agile development team and have learned that it is not possible for the **Customer** to **Obtain Consensus** among the stakeholders.

*Sometimes it is impossible for a **Customer** to* **Obtain Consensus**. There may be stakeholders with conflicting goals in the process.

- The customer may not be able to gain consensus from among the stakeholders.
- There may be too many or too diverse a set of stakeholders.
- You still have to make progress and please the "customers" in any case.
- Not moving forward while people fight isn't very productive.

***Therefore**, in each iteration, break up the velocity into individual budgets for each stakeholder that must be satisfied.* Let each segment choose stories of most value to itself. Sometimes the stakeholders themselves can agree on an equitable distribution. Otherwise, allocation among the customers must be done by a Big Boss, or the Customer representative, not by the developers.

- It is hoped that the application of this is rare. It complicates planning greatly. It can also complicate integration of the various pieces that weren't integrated as you go along, having been under the control of different minds.
- Note: This pattern is used "when all else fails."
- If in your planning you recognize this as likely to occur, agile development may not be your best methodology.

Acknowledgement. This paper has not yet had a formal shepherd.

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. [3]

Best Effort. The contract is not for features delivered on a given date. You want best effort and full communication. [3]

Bug Generates Test. When a bug appears in code, write a set of tests that will only pass when it is corrected. [3]

Cards and Whiteboards. Things change too frequently to depend on elaborate documentation mechanisms. [4]

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

Collective Responsibility. The team shares responsibility and rewards for all tasks. [3]

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. [4]

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. [3]

Effective Coach. A novice team depends fundamentally on a coach (ScrumMaster) to keep you to the discipline and help you see opportunities and problems. [3]

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

Estimate Whole Task. Estimates must include everything necessary for a story[4].

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. [4]

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. [3]

Implementer Estimates Tasks. Tasks are best estimated by the person who will do the work. [4]

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. [4]

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. [4]

Re-estimate Periodically. Things change and estimates become obsolete. [4]

Retrospective. Periodically hold a retrospective [10] of the team's practices.

Sacred Schedule. Time never slips in agile development. Features are the dependent variable. [3]

Sheltering Manager. A new team will depend on some shelter from those in the organization who don't readily accept change. [3]

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. [4]

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. [4]

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. [3]

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, Andres, *Extreme Programming Explained: 2ed*, Addison-Wesley, 2004
[2] Beck, *Smalltalk Best Practice Patterns,* Prentice Hall, 1996
[3] Bergin, Patterns for Agile Development Practice, Part 1, EuroPLoP 2005, available at http://csis.pace.edu/~bergin/patterns/xpPatternsEuroV7.html
[4] Bergin, Patterns for Agile Development Practice, Part 2, EuroPLoP 2006, to appear.
[5] Belshee, Promiscuous Pairing and Beginner's Mind: Embrace Inexperience, http://www.agile2005.org/XR4.pdf
[6] Mike Cohn, Agile Estimating and Planning (Robert C. Martin Series) Prentice Hall, 2005
[7] Coplien, Harrison, *Patterns for Agile Software Development*, Prentice Hall, 2004
[8] Jackson, Michael A. *Principles of Program Design*. Academic Press, London and New York, 1975
[9] Jeffries, Anderson, Hendrickson, Extreme Programming Installed, Addison-Wesley, 2001
[10] Kerth, Norm, *Project Retrospectives: A Handbook for Team Reviews*, Dorset House, 2001
[11] Manns, Rising, *Fearless Change*, Addison-Wesley, 2004
[12] Mugridge and Cunningham, Fit for Developing Software : Framework for Integrated Tests, Prentice Hall, 2005
[13] Rueping, Agile Documentation : A Pattern Guide to Producing Lightweight Documents for Software Projects, Wiley, 2003
[14] Schwaber, Beedle, Agile Software Development with Scrum, Prentice Hall, 2002
[15] Surowiecki, The Wisdom of Crowds, Anchor, 2005