

Patterns for Extreme Programming Practice

Joseph Bergin
Pace University
berginf@pace.edu
(Version 6)

Address:

45 North Oakwood Ter
New Paltz, NY 12561 USA
1 845 255 4369
1 845 255 2883 (fax)

This set of patterns is intended to complement the standard wisdom that can be gleaned from the Extreme Programming literature such as Kent Beck's *Extreme Programming Explained*. It is directed primarily at those who are starting out with Extreme Programming and might miss some subtle ideas. Once a team gains experience these patterns will become obvious, but initially some of them are counter intuitive.

The 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

We omit here most of the well-known advice, such as "Do the simplest thing that could possibly work" and "Yesterday's Weather", though we sometimes refer to them. They are well explained in the literature. We do consider them to be patterns, however, and a more complete compendium would include them as well. In fact, 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.

As this "language" is in its early stages of development, there is no significance to the current ordering of the patterns here other than a general arrangement by phases. This paper presents ten of the thirty-two patterns developed so far. The remaining are mentioned briefly in the Thumbnail section at the end.

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.

Permission is granted to EuroPLoP and to Hillside Europe to make copies of this work for purposes of the EuroPLoP 2005 conference.

Sheltering Manager

You are a manager who is responsible for a trial XP project. The organization of which you are part has little experience so far with this methodology.

Many things can disrupt any project, but when trying a new methodology one of the most difficult for the team is when the rest of the organization is not on board. *When unnecessary disruptive influences impinge on the team they won't be able to concentrate on the task at hand.*

- Any new methodology is revolutionary in an organization. It will have supporters, skeptics, and detractors.
- Before you can judge the worth of a new methodology you need to give it a fair trial.
- To shelter a team requires either institutional power and prestige or great bravery.

Therefore, *the manager's chief task is to shelter the team from disruptive influences by the rest of the organization.* Provide them with sufficient resources and keep the wolves at bay.

The sheltering manager can be the immediate supervisor or someone more remote. If the team comes from diverse parts of the organization it may need to be a high level supervisor or an especially cooperative team of lower level supervisors. Your job as the sheltering manager is to take the heat, but provide the light.

- You may become the focus of some sniping by detractors. You need to be prepared for this. Try not to let it reach the team, however.
- The team will largely manage itself, however. This will be a good thing if it works. Keep your eyes open for running off the rails, of course, but let that happen.
- If you are not, yourself, knowledgeable about XP include yourself when you **Train Everyone**. You need to know what to expect.

See **Patron Role** in [3]. This pattern details an additional task for the Patron.

Train Everyone

You are someone who is initiating an XP project in an organization and you have a pretty good idea who the personnel will be. You might be a manager, or a change agent.

There are a lot of stakeholders in any project. There are a lot of skills needed to develop it. The **Whole Team** consists of everyone with an essential skill, including the customer who represents the business stakeholders. *Everyone has different ideas about how the project will proceed. If the methodology is new to the company, much of this information is faulty.*

- You are engaging in a project with a new methodology. No one is yet skilled in its use.
- Everyone will have a place in the development, but everyone needs to understand the role of everyone else.
- The team needs to work cooperatively and meld into a whole.
- XP requires a high level of discipline. Think of it as a highly choreographed dance.
- In XP, most participants play a variety of roles.

Therefore, *provide training that both shows everyone all the essential roles, but also brings the participants together as a team.*

Don't underestimate the importance of this. Select the team, including the customer. Bring in a trainer, skilled in XP training and practice. Spend two or three days working together to introduce the required practices. The training should have an intellectual (reading/ listening/ discussing) component, but also an active component. Let people play roles in the training that are not their normal roles. Your goal in designing the training is not only to let people understand XP at a deeper level, but also to begin to build an effective team.

- Training will cost you both time and money.
- You may need to return to training after the project runs for a while. Having a **Retrospective** at a release point can let you know if this is necessary.
- The trainer(s) can, perhaps, later stay on to coach. This is not essential, but you will likely need an external coach if this is a first XP project. The coach needs to know what training was done, in any case.

There are several available training programs. Some of them are quite fun. Some focus on a few practices of XP and some on the process as a whole. The team can be asked to build something other than software. This lets non-programmers act as developers and permits programmers to take the customer role. The goal of the training is twofold: (a) demonstrate the effect of not carrying out the practices, and (b) bring the team together as a cohesive unit.

Hint: Include management in the training, especially the Sheltering Manager.

Collective Responsibility

You are the overall manager of an XP team. You need to set expectations of rights and responsibilities for those who work on the team.

XP is a lot about rights and responsibilities. It is also a lot about sharing information and skill building. *You need a high performance team in which every individual is deeply committed to every aspect of success of the project.*

- Individual responsibility is best for assigning blame when things go wrong, but not for reinforcing cooperation.
- When responsibility is individual it is often possible to know why something failed and to assign blame: Some individual didn't fulfill some task.
- However, this doesn't aid cooperation in a team. In extreme cases people believe that their only security is what they know that others on the team don't know, making development into a zero-sum game.
- If responsibility is shared and things go wrong you either need to blame everyone or no one when things go wrong. But blame doesn't build software.

Therefore, *make the responsibility for completion and for the artifacts collective by the team, not by individuals.* Let the team itself manage the fine-grained aspects of this.

- A consequence of this is that the reward structure must also be collective. It should not be possible for an individual to succeed while the team fails, nor for an individual to fail while the team succeeds, except in extraordinary circumstances.
- XP tries hard to avoid blame. When problems arise, give them to the team to solve. Experienced coaches can help with finding ways out of problem areas, particularly if the problem is due to lack of discipline in following the agreed practices. The short iteration cycles here help you, as problems are likely to be noticed earlier and, as a consequence likely to be of smaller scale than otherwise. Solving problems can help build the team, in fact.
- On the other hand, an XP team exerts quite a lot of pressure on its own members. The pressure can be positive or negative. In the best situations, the pressure to conform to team norms will tend to bring an errant member back into the fold. A coach needs to watch that the pressure stays positive. It is fine to not want to let the team down. It isn't good if someone is the goat. Some teams assign all blame to an imaginary team member or even to a member who agrees to always take the blame. Blame Chet [5].
- If you have a completely introverted hero hacker, don't put her on an XP team unless she demands it.

Note: There are certain situations in which **Collective Responsibility** is not possible. For example, in safety critical software you may have certain arcane skills known to only a small number of the developers. While it is preferable, in general, to spread these skills, it may not be possible. These may not be good candidates for XP practice, of course, or may require partitioning of the project.

Note that **Constant Refactoring**, in particular, won't work without **Collective Responsibility** as that practice implies that any code that needs to be changed to improve structure can be changed by anyone who recognizes the need. **Pair Programming** and **Collective Code Ownership** reinforce it.

Best Effort

You are a member of an XP team and are carrying out the planning phase (**Planning Game**) of a release. You are using XP in a project for which it is appropriate; the requirements are either not known in advance or it is known that they will change during the project.

Management and people paying bills like to have contracts for delivery of software. Since software is a creative human endeavor it is rarely possible to be successful with this approach, especially if the requirements can't be nailed down at the start. *XP does not capture all requirements prior to beginning development, so the target of the project is not clearly defined at the beginning.* The customer remains involved to steer the project to a fairly fine level of detail and the team will give feedback continuously on costs and risks. The essence of XP is that the team can react to changing conditions at reasonable cost.

- People often take estimates as a contract. They sometimes insist on it. However, if estimation is inherently inaccurate, then no one wins by insisting on delivery.
- The thing doesn't get built just because someone insists that it must.
- Something about the project is such that it is not possible to gather accurate and stable requirements at the beginning.
- Underestimates and overestimates can both waste resources unless there is a self-correcting mechanism.
- In some organizations it is common practice to hide information from one another.
- What you (as a manager) really need is a system of estimates that are useful for planning whether they are accurate or not.
- XP tries hard to avoid blame. Blame doesn't move the project forward.
- XP has other, more effective, ways to decide when to stop paying for product.

Therefore, in XP the contract is for best effort and complete communication. The customer will have complete information on which to base business decisions and can terminate the project at any time there is insufficient value delivered. Ideally this is when good business value has been given and only low priority/high cost features remain.

- If you can know the requirements in advance and they will not change, XP is probably not your best methodology.
- In XP you pay something for this flexibility to change requirements that you need not pay if they won't change. XP saves money by delaying expensive decisions and avoiding building low value features.
- To achieve this, however, you must build **High Value Stories First**.
- The tradeoff is that you get what you can for a reasonable effort and you don't need to guess what you may need in the future. But you can't insist on a certain level of cost/performance.

Many projects try to over-specify a project. As Beck explains in [1] there are four variables: cost, features, schedule, and quality. You can only specify three of these and the fourth is dependent. Many projects try to specify all of them in a contract, but in reality the schedule slips and so becomes the dependent variable. In XP the *features* variable is explicitly made the dependent variable. Features may slip (in an iteration), but the others are held fixed.

High Value First

You are a customer on an XP project and you are involved in the **Planning Game**. You are trying to decide which stories to schedule in the next few iterations.

Stories have different business value. They have different development costs. *The effort to develop a product should be commensurate with its business value.*

- In top-down (structured, waterfall) development the developers often don't get good guidance as to the relative value of the features required. They may spend a lot of effort (time and money) on low value features. This should be avoided.
- From the customer's standpoint, some features are essential and some "would be nice to have."
- The customer needs to be able to assign value to each story she writes. The developers will assign it a cost (the estimate).
- The value will change over time as business conditions and strategies change.

Therefore, *build the high value, most essential, stories first*. When the value curve and the cost curve cross, cancel the project. And note that no scaffolding was put in place to support these low value features since you have always DTSTTCPW. At any point, schedule the highest value remaining stories in the next available iterations. If the cost of a story is higher than its value you can often split the story into its essential and inessential parts. Once these parts are re-estimated you may be in a better position to proceed.

- This assumes, of course, that your estimate of value is measurable and accurate. This is often difficult to achieve. Short iterations force you to think in terms of small granularity features. This makes estimation easier, but partitioning harder. If something important is more complex than can be done in an iteration it must be split. When this is fundamentally impossible XP will not be an ideal methodology.
- It also assumes that the value is more than the cost implied by the estimate provided by the developers.
- There is an additional cost here, since the developers don't see every story from the beginning, they cannot make certain optimizations that would lower cost.
- But these optimizations will waste money if requirements change and make them obsolete. XP developers don't anticipate future needs.
- Complicating this is the fact that the first iteration needs to create an **End To End** version of the product, though many features will be omitted and others in skeleton form only.

The value of stories will then generally decrease. The cost will generally increase, as it gets harder to incorporate new stories as you go along. **Refactoring**, tries to keep this rising cost curve as flat as possible by keeping the design coherent even as new things are added. At some point, however, the remaining stories are probably not worth building. Note that this is one of the major ways that agile processes can save money over planned development: the low value requirements are just dropped. It can also deliver you a product sooner.

In extremely volatile situations the crossing curves effect may not occur. You might only learn of a high value requirement late in the process so the value curve might take a sharp upward turn. But this ability to quickly retarget the project toward a different goal delivers value in a different way: you get a more suitable product.

Easy Does It (was Don't Push Too Hard)

You are the customer on an XP team. You are deep into the **Planning Game**. You are reviewing estimates and using the velocity given by the team to schedule stories.

You, the customer may be too anxious to see a lot of progress. You may not entirely trust the developers to give you best effort. *If you push too hard, you are at risk of not completing anything.*

- You like low story estimates, thinking that things get done quickly.
- You like a high velocity for an iteration, thinking that a lot will get done.
- Nearly everyone wants to do a good job.
- Coercion doesn't build software. People respond poorly to coercion.
- The environment is important. A high-pressure environment seldom improves quality or motivation, especially when the pressure is external.
- There are more effective ways to keep the project moving forward smoothly than pure pressure.
- You want the team to work at its maximum sustainable pace.

Therefore: *Don't try to push the developers for low estimates or for high velocity.* Only so much work will get done in an iteration no matter what the estimates and velocities are. You are better with accurate numbers than with optimistic ones. If they cannot be accurate, due to lack of knowledge, you are better off with conservative rather than aggressive estimates.

If you try to put too much into an iteration, you run the risk of *nothing* being completed. Partly this is due to the fact that wise developers will both optimize over the stories in an iteration and will refactor to make it easier to build them. This takes time. If there is too much to do generally then it won't be done well and you can end the iteration with many stories partially built, but few or none completed. This will drive the velocity down for the next iteration, complicate your own planning, and generally dishearten everyone.

- If you really don't trust the developers, then work with them to develop trust. If you really *can't* trust the developers, then replace them. But their responsibility is the technical requirements that go into the stories, while yours is the business value.
- You will be present continually as the team works, so should be able to judge if people are slacking or giving best efforts.
- Let the pressure come from the developers themselves. If progress is slow hold a **Retrospective** to learn what can be changed to increase it. Coach, take notice.

Estimates that are too high by a modest amount are easier to work with than underestimates, similarly with velocities that are too low. Near the end of an iteration it is much easier and more satisfying if the team comes to you for more work than if they need to come to you asking what should be dropped since not everything can be completed. If you push too hard you put yourself in a poor position as well as leading the team to burnout, which will not improve your product or future development efforts. But even if you don't complete your tasks this iteration, you will correct in the next cycle using **Yesterday's Weather**.

Hints for success: Make a big deal of it when the developers first come to you for more work in an iteration. Find some way to reward them. Make it possible for this to occur early in the project.

Be Human (was Do Food)

You are managing an XP team. You are setting up the environment, or you are trying to keep the process running smoothly with lots of happy and productive developers.

People work best when they can behave like people, not machines. People are social. People need a variety of experience so that they don't get bored or stuck.

- To keep people happy, balance work and life.
- Food is an important part of our social interactions.
- If people work too hard for too long they often become less effective.
- The mind works when "idle". Sometimes it is more effective when not being pushed.
- Don't underestimate the power of serendipity. Sometimes just chatting can bring out important information, opportunities, and solutions.

Therefore, provide a work environment that lets people interact naturally as if it were a social setting. For example, provide food in the workspace as well as a social corner in which people can talk. Have occasional time-outs that are purely social or purely fun.

A wise manager will bring snacks into the workplace. Provide a small refrigerator for the team's use. Have a party occasionally at the end of a successful iteration. Tension breaking activities are also useful whether they involve food or games or whatever the team likes to do. Games and toys can be useful in keeping the team moving forward and happy about the work. Have ceremonies that are meaningful to the team members and reward good behavior. Let the team members find ways to reward each other.

- There is a cost in money and time, of course, but happy, rested, people think and work more effectively than those overworked and tired.
- Sugar, fat, and caffeine are not especially healthy, though these are the typical choices.
- This may be culturally dependent, of course. If your coach is knowledgeable about the local culture, she can probably help you do this appropriately.
- If you absolutely must have a high pressure environment, perhaps XP is not your best methodology. It is based on the principle that development is a creative activity.

Note that the goal here is more than just morale. People work best when they are happy and rested. The **Sustainable Pace** (40 hour week) practice of XP is designed not only to create a humane workplace, but also to get the best work out of everyone every minute they are engaged. Development work is creative. You, the manager, need to foster the creative environment.

If you want to see the effect of this on creative work, visit an art studio. Disney Studios was known for its pranks and tension breakers, but was both creative and productive.

Notes: *Do Food* also appears in [7, pg 132] for much the same reason, though in a different context.

Sacred Schedule

You are a developer on an XP project. It is during the development phase of an iteration.

If you don't end the iteration on time, you have no basis of knowing where you are. Recall that most projects are estimated as 90% done after the first third of the project and remain so until long after the due date. If you slip schedule you lose control rapidly. It becomes easy to do it again. If you fail to complete a task in the current iteration, someone is unhappy, of course, but you know where you are.

- There is pressure to finish everything, but the clock gets in the way.
- One more (hour, day, week) would let you finish all this work. But you might not finish *then*, either.
- If you don't know how fast you can really go, you can't plan for the future.
- With accurate velocities (story points per iteration) you can plan future iterations and releases with confidence. This lets you control cost.
- Nothing in development is really sacred, however.

Therefore, don't put off the scheduled end of an iteration, even for just a day. Some tasks may not get completed, but this is self-correcting in your future velocities. The only work that gets counted in an iteration for the computation of the next velocity (**Yesterday's Weather**) are the tasks completed and accepted by the customer. If you didn't finish it, don't count it. This will give you a lower, but more realistic, velocity for the next iteration.

- If you don't finish a task or a story in the current iteration, you can save (but not commit) the work done and write a story for its completion with a new time estimate. The customer can schedule these or not in future iterations.
- Customers may have a hard time with this initially. They like to think that the pile of stories in an iteration is like a contract. They are unhappy when you don't complete everything. If you have kept everyone informed along the way you can lessen the blow for the customer at the end. Write stories and move on. Coach, take notice.
- Plan to have the final integration test at noon of the last day of the iteration. If everything comes together nicely you can refactor for half a day. Or have a party. If not, you have half a day to make it right.
- Over a few iterations you will know how much work, in story points, the team can do in (say) ten days. This gives you the basis of long-term estimation by extrapolation, as long as the average difficulty of the stories remains the same.
- However, there are situations in which the customer has a special need. There may be long-scheduled meetings with real users who have traveled a distance to try certain features. You might need to accommodate this, even if it means (gasp) putting in overtime for a short period. Management should expect lower productivity after a push, however. Sometimes it helps to reward people for the effort required in the push. Above all, however, avoid the death-march syndrome that is all too common today.
- The work completed forms the basis of long term planning by management. Once you know how many story points you can complete in an iteration you know how much work can get done by some future date and/or how long it will take you to complete an estimated amount of work. If you slip schedule, you have no basis for this planning.

Bug Generates Test

You are a developer on an XP team, or perhaps the customer. You have just found a bug. Oh my.

Bugs will occur. They are an indication of inadequate testing. *When you fix a bug you want to know both that it is fixed, and that it stays fixed in the future.*

- Good programmers write few bugs, but not NO bugs. Bugs happen to good programmers. You tried, but you didn't catch everything that could possibly go wrong when you originally built the feature and its tests.
- Requirements are evolving. Sometimes what appears to be a bug is an inconsistency in requirements.
- You need to know how the bug is behaving and why.
- You need confidence that you have removed it.
- You don't want a solution to introduce other bugs.

Therefore, *whenever you find a bug, create a test* so that you know when you have fixed it and that it stays fixed.

It may require several tests, actually, but the tests you write will guide your debugging efforts.

- Like **Test First**, this is a practice that takes discipline and it may seem awkward at first.
- If you do this a lot, you probably need to improve your **Test First** discipline.
- If you do this a lot *recently*, it is probably time for a big **Refactoring** push.
- If bugs are discovered by the customer, it is a sign that communication needs to be improved. Developers may need a better understanding of the stories.
- A good set of tests written when the bug is first discovered will also help you avoid the common practice of applying a cosmetic patch, and will lead you to explore the problem more deeply.

Some bugs occur because what you built was inadequately specified. You need to learn what should have been built, of course, and capture that knowledge in tests. Some bugs occur because you built the wrong thing. But if you had adequate tests beforehand, this would not have occurred.

Note that changing requirements does not imply an error has occurred, though tests will fail as they match the old requirements, not the new. This implies that when tests fail you need to understand whether it is the code or the test that is failing. Update accordingly.

Effective Coach

You are managing an XP team, setting up the environment for success. Or you have been designated coach of an XP team.

No process can work if you don't perform its practices. *XP requires more discipline than you might expect.* This is especially true as some of the practices are counter-intuitive to many programmers. Many practices that are counter-productive in practice are ingrained in many work environments. People do these things "naturally" even though they know that they don't work.

- You want to give XP a fair trial to see if it will work for you, but your habits are not aligned with the practices.
- The practices require discipline.
- You really don't want to consistently do all the practices.
- It can be difficult to know when something goes wrong how it might have gone better if the practices had been different.
- Not every practice is "right" in every circumstance. It takes experience to judge this.
- Everyone has a lot to do.
- Coaching costs money whether the coach is a regular employee or a hired consultant.

Therefore, someone on the team will have a full time role as Coach. The coach keeps the team faithful to the practices. The coach role becomes part time after you have experience. Initially it will be helpful to have an external coach.

- An external coach is expensive. A dedicated internal person is also expensive.
- Coaching is a role, not a job description. It can be shared and rotated in an experienced team. An external coach can train her successors. But budget enough so that the coaching is pervasive initially.
- The trainer can segue into a coach if you have someone with the right skills. The best coach is someone who has worked with several XP teams.
- Guard against the coach taking on a management function. A manager should not coach. Management should not ask the coach for member evaluations. The coach has a professional counseling role with the team.

A first XP project is strongly encouraged to hire an external coach who is experienced with the practices. Gradually, the coaching function can be taken over by a team member. The coach is not a manager. It is her job to point out the consequences of not performing the practices, seeing when the practices are not working, and adapting the practices to the local situation. The coach needs to be present for at least the planning game sessions and the daily meetings. And of course, the coach needs to be experienced with XP.

Acknowledgement. James Noble kindly and ably shepherded this paper for EuroPLoP 2005. I have learned from his insights. Till Schuemmer represented the committee in the process.

I hope the readers will forgive the fact that we are, all three, academics. The patterns here come from real practice, however, not academic exercises.

Thumbnails and Acronyms (Incomplete). This includes patterns that are properly part of this work, but are not presented in detail in this paper.

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.

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.

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

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

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.

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

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

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.

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

Retrospective. Periodically hold a retrospective [6] of the team's practices.

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

Estimate Everything. Estimates must include everything necessary for a story.

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

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

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

Switch Partners. Spread the knowledge of the project amongst the team members.

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

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

Paper is the Best Tool. Things change too frequently to depend on elaborate documentation mechanisms.

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

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

Continuous Integration. Every task is integrated immediately into the codebase until all unit tests pass.

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

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

Record of Velocity. You need to know how fast you can go to estimate effectively, but it is an acquired skill that requires documentation over time.

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.

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

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

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

References

[1] Beck, *Extreme Programming Explained: 2ed*, Addison-Wesley, 2004

[2] Beck, *Smalltalk Best Practice Patterns*, Prentice Hall, 1996

[3] Coplien, Harrison, *Patterns for Agile Software Development*, Prentice Hall, 2004

[4] Jackson, Michael A. *Principles of Program Design*. Academic Press, London and New York, 1975

[5] Jeffries, Anderson, Hendrickson, *Extreme Programming Installed*, Addison-Wesley, 2001

[6] Kerth, Norm, *Project Retrospectives: A Handbook for Team Reviews*, Dorset House, 2001

[7] Manns, Rising, *Fearless Change*, Addison-Wesley, 2004