

Extreme Programming Explained, Second Edition

By Kent Beck, Cynthia Andres

Chapter 12. Planning: Managing Scope

The state of a shared plan provides clues about the state of the relationship between the people affected by the plan. A plan out of touch with reality betrays an unclear, unbalanced relationship. A mutually agreed-upon plan, adjusted when necessary to reflect changing reality, suggests a respectful, mutually valuable relationship.

Planning makes goals and directions clear and explicit. Planning in XP starts with putting the current goals, assumptions, and facts on the table. With current, explicit information, you can work toward agreement about what's in scope, what's out of scope, and what to do next.

Planning in XP is like shopping for groceries. Imagine that you go into a grocery store with \$100 in your pocket. Items on the shelves each have a price attached. Some items you need; others you don't; and still others you want, but they don't fit your budget. If you get to the checkout with \$101 worth of food, you have to put something back. Your job while shopping is to spend your \$100 wisely, buying what you need and as much of what you want as possible.

In XP, the groceries are the stories. The prices are the estimates attached to the stories. The budget is the amount of time available. The desired deployment date is usually set early in a project, so you know how much you have to spend on stories. If you have two hundred pair-hours in your pocket and you have four hundred pair-hours worth of stories in your cart, pick the most valuable set of stories adding up to two hundred pair-hours. Otherwise, everyone knows you have more in the cart than you can afford.

Part of planning is deciding what to do next out of all the possibilities. Planning is complicated because the estimates of the cost and value of stories are uncertain. The information on which you base these decisions changes. We use feedback to improve our estimates and make decisions as late as possible so they are based on the best possible information. That's why planning is a daily, weekly, and quarterly activity in XP. The plan can change to fit the facts as they emerge.

Plans are not predictions of the future. At best, they express everything you know today about what might happen tomorrow. Their uncertainty doesn't negate their value. Plans help you coordinate with other teams. Plans give you a place to start. Plans help everyone on the team make choices aligned with the team's goals.

As a young software engineer, I learned three variables by which to manage projects: speed, quality, and price. The sponsor gets to fix two of these variables and the team gets to estimate the third. If the plan is unacceptable, the negotiating starts.

This model doesn't work well in practice. Time and costs are generally set outside the project. That leaves quality as the only variable you can manipulate. Lowering the quality of your work doesn't eliminate work, it just shifts it later so delays are not

clearly your responsibility. You can create the illusion of progress this way, but you pay in reduced satisfaction and damaged relationships. Satisfaction comes from doing quality work.

The variable left out of this model is scope. If we make scope explicit, then:

We have a safe way to adapt.

We have a way to negotiate.

We have a limit to ridiculous and unnecessary demands.

Plan at every timescale with these four steps:

1. List the items of work that may need to be done.
2. Estimate the items.
3. Set a budget for the planning cycle.
4. Agree on the work that needs to be done within the budget. As you negotiate, don't change the estimates or the budget.

Everyone on the team needs to be heard. Planning provides a forum in which the team acknowledges wishes, but only commits to needs.

This procedure works at the scale of a pair of programmers planning a couple of hours of test cases they want to satisfy and design they want to improve. It works at the scale of a team planning the day's activities. It works more formally at the scale of team planning a week or a quarter, where stories are the items on the list, estimation is more formal, and planning takes hours or days.

Planning is something we do together. It requires cooperation. Planning is an exercise in listening, speaking, and aligning goals for a specific time period. It is valuable input for each team member. Without planning, we are individuals with haphazard connections and effectiveness. We are a team when we plan and work in harmony.

Everyone on the team should be involved in planning. Some XP teams have the customers on the team get together privately to fight over the coming week's budget. This just shifts the zero-sum game away from the programmers so they don't have the urge to overcommit. Doing this eliminates opportunity for mutual benefit. The whole team together may be able to find a way for the seemingly divergent needs of the customers to be satisfied. If the team doesn't know all of the needs or issues involved; it can't make good choices, business or technical.

When choosing which stories to implement next, sort them several ways. The act of laying the stories out spatially provides new insight into the relationships between the stories and smooths the selection process. You could put risky stories towards the left and valuable stories towards the top. You could put all the performance tuning stories in one corner of the table and all the new functionality stories in another corner. Whenever I get lost while planning, I gather all the stories up off the table, shuffle them, and lay

them out fresh.

To estimate a story, imagine, given everything you know about similar stories, how many hours or days it will take a pair to complete the story. "Complete" means ready for deployment; including all the testing, implementation, refactoring, and discussions with the users. As your knowledge of similar stories increases, your estimates will improve. Estimates are based on a reasonable pair working on the story. Some pairs might be better and others worse, but if everyone takes a turn estimating it should all average out in the end.

At first, these estimates can be wildly wrong. Estimates based on experience are more accurate. It is important to get feedback as soon as possible to improve your estimates. If you have a month to plan a project in detail, spend it on four one-week iterations developing while you improve your estimates. If you have a week to plan a project, hold five one-day iterations. Feedback cycles give you information and the experience to make accurate estimates. Gain this experience as soon as possible so your estimates improve.

This provides the items (stories) and prices (estimates). How do you establish the budget (time to completion and size of team)? Measure how many productive programmer hours you get in the average week and divide by two to account for pairing. A team with six programmers and four hours of programming a day should plan on twelve pair-hours per week. One of the objections to pairing is that pairing cuts effective programming in half. In my experience, pairs are more than twice as effective. The actual time required for me to complete tasks solo versus paired, accounting for debugging time, is more than double; so by pairing you actually come out ahead in completed, clean code. When comparing the value of pairs to individuals, you need to include both time and productivity in deployable code. The goal is valuable software development delivered on time and in budget. The numbers in the plan matter, but only in service of this goal. In planning, you need to include all the relevant numbers in your calculations.

The first edition of Extreme Programming Explained had a more abstract estimation model, in which stories cost one, two, or three "points". Larger stories had to be broken down before they could be planned. Once you started implementing stories, you quickly discovered how many points you typically accomplished in a week. I prefer to work with real time estimates now, making all communication as clear, direct, and transparent as possible.

There is a limit to how much work can be done in a day. Paying attention to this real limit allows you to plan effectively and deliver successfully. Saying that programmers should just accomplish twice as much doesn't work. They can gain skills and effectiveness, but they cannot get more done on demand. More time at the desk does not equal increased productivity for creative work.

Whichever units you use, hours or points, you will need to deal with the situation where actual results don't match the plan. If you are estimating in real time, modify the estimates on the yet-to-be-completed stories in the light of experience. If you are estimating in points, modify the budget for subsequent cycles. A simple way to do this, dubbed "yesterday's weather" by Martin Fowler, is to plan in any given week for exactly

as much work as you actually accomplished in the previous week. Adjust the plan as soon as you are sure of new information.

Sometimes you will be in the middle of a cycle with progress slower than planned. Look for ways to realign with the plan. Have you been distracted by less important issues? Have you been lax on a practice that could help you? If no process change is going to restore the balance between plan and reality, ask the customers to choose which stories they would like to see completed first. Work on those to the exclusion of all else. The time this replanning takes will be more than repaid in increased harmony and efficiency as the team works towards the deployment date. Without the adjustment, you are working under a lie. Everyone knows it and has to hide to protect themselves. This is no way to get good software done and deployed; and it undermines team and individual confidence.

When things aren't going well is when we most need to adhere to our values and principles and modify our practices to remain as effective as possible. Inaccurate estimates are a failure of information, not of values or principles. If the numbers are wrong, fix the numbers and communicate the consequences.

Write stories on index cards and put the cards on a prominent wall. Many teams try to skip this step and go straight to a computerized version of the stories. I've never seen this work. Nobody believes stories more because they are on a computer. It is the interaction around the stories that makes them valuable. The cards are a tool. The interaction and alignment of goals, shared belief in the stories, are the valuable part. You can't automate relationships. The goal is to have a plan everyone believes in and is working to fulfill.

There is a balance of power on a project: there are people who need work done and people who do work well. They both have information necessary for believable planning. Cards on a wall is a way of practicing transparency, valuing and respecting the input of each team member.

The project manager has the task of translating the cards into whatever format is expected by the rest of the organization. He or she can also teach others to read the wall. We have nothing to hide. That's the plan, open and accessible, that reflects the kind of relationships that make for the most valuable software development.