UNIT IV

Planning

Planning

- Vision reveals where the project is going and why it's going there.
- Release Planning provides a roadmap for reaching your destination.
- The Planning Game combines the expertise of the whole team to create achievable plans.
- Risk Management allows the team to make and meet long-term commitments.
- Iteration Planning provides structure to the team's daily activities.
- Slack allows the team to reliably deliver results every iteration.
- Stories form the line items in the team's plan.
- Estimating enables the team to predict how long its work will take.

Vision

- We know why our work is important and how we'll be successful.
- Our vision is to serve customers while maximizing stakeholder value

Product Vision

- projects start out as ideas focused on results.
- The idea is so compelling that it gets funding, and the project begins.
- transition from idea to project, the compelling part—the vision of a better future—often gets lost.
- create stories, schedule iterations, and report on progress

Vision

Where Visions Come From

- compelling idea
- There are multiple visionaries, each with their own unique idea of what the project should deliver.
- promote
- the vision to the team and to stakeholders.

Vision

- **Identifying the Vision**
- the best approach is to have that visionary act as product manager.
- Prioritize the vision
- **Documenting the Vision**
- Disagreement and confusion. Without a vision statement, it’s all too easy to gloss over disagreements and end up with an unsatisfactory product.
- maintain and promote the vision
- Better understanding

Vision

- **How to Create a Vision Statement**
- The vision statement documents three things: what the project should accomplish, why it is valuable, and the project's success criteria.
- Based on requirements statements

Promoting the Vision

- After creating the vision statement, post it prominently as part of the team's informative workspace.
- Use the vision to evangelize the project to stakeholders and to explain the priority of specific stories.
- help in improving the plan, and give them opportunities to write stories.

Vision

Results

- maximize value while minimizing development cost.
- When the visionary promotes the vision well, everyone understands why the project is important to the company.

Contraindications

- If the project isn't worth doing, it's better to cancel it now than after spending a lot of money.

Alternatives

- The only alternative to a clear vision is a confusing one—or no vision at all.

Release Planning

- Release Planning provides a roadmap for reaching your destination.
- We plan for success.
- Maximize our return on investment

Release Planning

One Project at a Time

- work on only one project at a time.
- Many teams work on several projects simultaneously, which is a mistake.

Release Early, Release Often

- Releasing early is an even better idea when you're working on a single project.
- most valuable features together and release them first

Release Planning

How to Release Frequently

- Releasing frequently doesn’t mean setting aggressive deadlines.
- Instead, release more often by including less in each release.
- Minimum marketable features(small set of functionality in a product that must be provided for a customer to recognize any value)
- provides value to your market
- MMFs provide value in many ways, such as
- competitive differentiation, revenue generation, and cost savings.
- The most difficult part of this exercise is figuring out how to make small releases.

Release Planning

Adapt Your Plans

Keep Your Options Open

- At any time, you should be able to release a product that has value proportional to the investment you've made.

How to Create a Release Plan

- There are two basic types of plans: scopeboxed plans and timeboxed plans.
- A scopeboxed plan defines the features the team will build in advance, but the release date is uncertain.
- A timeboxed plan defines the release date in advance, but the specific features that release will include are uncertain.
- Timeboxed plans are almost always better.

Release Planning

Planning at the Last Responsible Moment

- To plan at the last responsible moment, use a tiered set of planning horizons. Use long planning horizons for general plans and short planning horizons for specific, detailed plans.
- depend on your situation and comfort level.

Adaptive Planning and Organizational Culture

- Work within your organization's culture.

Release Planning

Results

- When you create, maintain, and communicate a good release plan, the team and stakeholders all know where the project is heading.
- If you are adapting your plan well, you consistently seek out opportunities to learn new things about your plan, your product, and your stakeholders.

Release Planning

Contraindications

- Working on one project at a time is an easy, smart way to increase your return on investment.
- Releasing frequently requires that your customers and users be able to accept more frequent releases.
- Adaptive planning requires that your organization define project success in terms of value rather than “delivered on time, on budget, and as specified.

Release Planning

- **Alternatives**
- The classic alternative to adaptive release planning is predictive release planning, in which the entire plan is created in advance.
- This can work in stable environments, but it tends to have trouble reacting to changes.

Risk Management

- Risk Management allows the team to make and meet long-term commitments.
- Team members get sick and take vacations; hard drives crash, and although the backups worked, the restore doesn’t; stakeholders suddenly realize that the software you’ve been showing them for the last two months needs some major tweaks before it’s ready to use.

Risk Management

A Generic Risk-Management Plan

- Every project faces a set of common risks: turnover, new requirements, work disruption, and so forth.
- These risks act as a multiplier on your estimates, doubling or tripling the amount of time it takes to finish your work.

Risk Management

Project-Specific Risks

- List of catastrophes, brainstorm scenarios that could lead to those catastrophes. From those scenarios, imagine possible root causes.
- These root causes are your risks: the causes of scenarios that will lead to catastrophic results.
- Estimated probability
- Specific impact to project if it occurs

Risk Management

For the risks you decide to handle, determine transition indicators, mitigation and contingency activities, and your risk exposure:

- Transition indicators
- Mitigation activities (thecontrol, tools, other mechanisms employed to manage each risk)
- Risk exposure
- Contingency activities

(course of action designed to help an organization)

Risk Management

How to Make a Release Commitment

- With your risk exposure and risk multipliers, you can predict how many story points you can finish before your release date. Start with your time boxed release date from your release plan.
- Figure out how many iterations remain until your release date and subtract your risk exposure.

Risk Management

Success over Schedule

- The majority of this discussion of risk management has focused on managing the risk to your schedule commitments.
- Success is more than a delivery date.

Results

- With good risk management, you deliver on your commitments even in the face of disruptions.

Risk Management

Contraindications

- failure is not an option.
- You may face criticism for looking for risks. You may still be able to present your risk derived schedule as a commitment or stretch goal, but publicizing your risk census may be a risk itself.

Alternatives

- Risk management is primarily an organizational decision rather than an individual team decision.
- add a risk buffer to individual estimates rather than the overall project schedule.

Iteration Planning

The Iteration Timebox

- In XP, the iteration demo marks the end of the iteration. Schedule the demo at the same time every week. Most teams schedule the demo first thing inthe morning, which gives them a bit of extra slack the evening before for dealing with minor problems.

The Iteration Schedule

- Iterations follow a consistent, unchanging schedule:
- Demonstrate previous iteration (up to half an hour)
- Hold retrospective on previous iteration (one hour)
- Plan iteration (half an hour to four hours)
- Commit to delivering stories (five minutes)
- Develop stories (remainder of iteration)
- Prepare release (less than 10 minutes)

Iteration Planning

The Commitment Ceremony

- Openly discuss problems without pressuring anybody to commit.

After the Planning Session

- After you finish planning the iteration, work begins. Decide how you'll deliver on your commitment.
- Every team member is responsible for the successful delivery of the iteration's stories.

Dealing with Long Planning Sessions

- Iteration planning should take anywhere from half an hour to four hours. Most of that time should be for discussion of engineering tasks.

Iteration Planning

Tracking the Iteration

- Like your release plan, your iteration plan should be a prominent part of your informative workspace.

When Things Go Wrong

- When you discover a problem that threatens your iteration commitment

Partially Done Work

- You may delete code, but you won't delete what you've learned.

Emergency Requests

- responsiveness to business needs is a core agile value, so suppress thathomicidal urge, smile, and provide your customer with options.
- change the iteration schedule under the condition
- replace stories that you haven't started yet.

The Batman

- Dealing with an emergency request every now and then is fine

Iteration Planning

Results

- When you use iterations well, your team has a consistent, predictable velocity.
- The team discovers mistakes quickly and deals with them while still meeting its commitments.

Contraindications

- In order to achieve a consistent velocity and deliver on commitments, your iteration must include slack.
- Energized work is also important. Without it, the team will have trouble maintaining equilibrium and a stable velocity.
- **Alternatives**
- Use independent phases instead of simultaneous phases and iterations.

Stories

- Stories may be the most misunderstood entity in all of XP.
- They're not requirements.
- They're not use cases.
- They're not even narratives. They're much simpler than that.
- Stories are for planning.
- They're simple one- or two-line descriptions of work the team should produce.
- Alistair Cockburn calls them "promissory notes for future conversation

 Everything that stakeholders want the team to produce should have a story, for example:

- "Warehouse inventory report"
- "Full-screen demo option for job fair"
- "TPS report for upcoming investor dog-and-pony show"
- "Customizable corporate branding on user login screen"

- This isn't enough detail for the team to implement and release working software, nor is that the intent of stories.
- A story is a placeholder for a detailed discussion about requirements.
- Customers are responsible for having the requirements details available when the rest of the team needs them.
- Although stories are short, they still have two important characteristics:
- 1. Stories represent customer value and are written in the customers' terminology. (The best stories are actually written by customers.) They describe an end-result that the customer values, not implementation details.
- 2. Stories have clear completion criteria. Customers can describe an objective test that would allow programmers to tell when they've successfully implemented the story.

The following examples are not stories:

- "Automate integration build" does not represent customer value.
- "Deploy to staging server outside the firewall" describes implementation details rather than an end-result, and it doesn't use customer terminology. "Provide demo that customer review board can use" would be better.
- "Improve performance" has no clear completion criteria. Similarly, "Make it fast enough for my boss to be happy" lacks objective completion criteria. "Searches complete within one second" is better.

Story Cards

Write stories on index cards.

- During release planning, customers and stakeholders gather around a big table to select stories for the next release.
- It's a difficult process of balancing competing desires.
- Index cards help prevent these disputes by visually showing priorities, making the scope of the work more clear, and directing conflicts toward the plan rather than toward personalities

- Story cards also form an essential part of an informative workspace.
- After the planning meeting, move the cards to the release planning board—a big, six foot whiteboard, placed prominently in the team's open workspace

- You can post hundreds of cards and still see them all clearly.
- For each iteration, place the story cards to finish during the iteration on the iteration planning board (another big whiteboard) and move them around to indicate your status and progress.
- Both of these boards are clearly visible throughout the team room and constantly broadcast information to the team

Customer-Centricity

- Stories need to be customer-centric. Write them from the on-site customers' point of view, and make sure they provide something that customers care about.
- On-site customers are in charge of priorities in the planning game, so if a story has no customer value, your customers won't—and shouldn't— include it in the plan.

- Customer-centric stories aren't necessarily always valuable to the end-user, but they should always be valuable to the on-site customers.
- For example, a story to produce a tradeshow demo doesn't help end-users, but it helps the customers sell the product.

Splitting and Combining Stories

- Although stories can start at any size, it is difficult to estimate stories that are too large or too small.
- Split large stories; combine small ones. The right size for a story depends on your velocity.
- You should be able to complete 4 to 10 stories in each iteration.
- Split and combine stories to reach this goal. For example, a team with a velocity of 10 days per iteration might split stories with estimates of more than 2 days and combine stories that are less than half a day.
- Combining stories is easy. Take several similar stories, staple their cards together, and write your new estimate on the front.

Special Stories

- Most stories will add new capabilities to your software, but any action that requires the team's time and is not a part of their normal work needs a story.

Documentation stories

- XP teams need very little documentation to do their work, but you may need the team to produce documentation for other reasons.
- Create documentation stories just like any other: make them customer-centric and make sure you can identify specific completion criteria.
- An example of a documentation story is "Produce user manual."

“Nonfunctional” stories

- Performance, scalability, and stability—so-called nonfunctional requirements —should be scheduled with stories, too.
- Be sure that these stories have precise completion criteria.

Bug stories

- Ideally, your team will fix bugs as soon as they find them, before declaring a story “done done.” Nobody’s perfect, though, and you will miss some bugs.
- Schedule these bugs with story cards, such as “Fix multiple-user editing bug.”
- Schedule them as soon as possible to keep your code clean and reduce your need for bug-tracking software. Bug stories can be difficult to estimate.
- Often, the biggest times in debugging is figuring out what’s wrong, and you usually can’t estimate how long that will take.
- Instead, provide a time boxed estimate: “We’ll spend up to a day investigating this bug. If we haven’t fixed it by then, we’ll schedule another story.”

Spike stories

- Sometimes programmers won't be able to estimate a story because they don't know enough about the technology required to implement the story.
- In this case, create a story to research that technology.
- An example of a research story is "Figure out how to estimate 'Send HTML' story."
- Programmers will often use a spike solution to research the technology, so these sorts of stories are typically called spike stories.

Results

- When you use stories well, the on-site customers understand all the work they approve and schedule.
- You work on small, manageable, and independent pieces and can deliver complete features frequently.
- The project always represents the best possible value to the customer at any point in time.

Contraindications

- Stories are no replacement for requirements. You need another way of getting details, whether through expert customers on-site (the XP way) or a requirements document (the traditional way).
- Be very cautious of using customer-centric stories without also using most of the XP development practices.
- Customer-centric stories depend on the ability to implement infrastructure incrementally with incremental design and architecture.
- Without this ability, you're likely to incur greater technical debt.

- Physical index cards are only appropriate if the team sits together, or at least has a common area in which they congregate.
- Experienced distributed teams often keep physical index cards at the main site and copy the cards into the electronic system.
- This is an administrative headache, but for these teams, the benefits of physical cards make the added work worthwhile. Some organizations are skittish about using informal planning tools.
- If important members of your organization require a formal Gantt chart, you may need to provide it. Your project manager can help you make this decision.
- As with a distributed team, you may find it valuable to use physical cards as your primary source, then duplicate the information into the tool.

Alternatives

- For teams that don't use stories, the main distinction between stories and the line items in most plans is that stories are customer-centric.
- If you can't use customer-centric stories for some reason, customers cannot participate effectively in the planning game.
- This will eliminate one of its primary benefits: the ability to create better plans by blending information from both customers and programmers.
- Unfortunately, no alternative practice will help.

- Another distinctive feature of stories is the use of index cards.
- Physical cards offer many benefits over electronic tools, but you can still use electronic tools if necessary.
- Some teams track their stories using spreadsheets, and others use dedicated agile planning tools.
- None of these approaches, however, provide the benefits of physical index cards.

Estimating

- We provide reliable estimates. Programmers often consider estimating to be a black art—one of the most difficult things they must do.
- Many programmers find that they consistently estimate too low.
- To counter this problem, they pad their estimates (multiplying by three is a common approach), but sometimes even these rough guesses are too low.
- Are good estimates possible? Of course! You just need to focus on your strengths.

What Works (and Doesn't) in Estimating

- One reason estimating is so difficult is that programmers can rarely predict how they will spend their time.
- A task that requires eight hours of uninterrupted concentration can take two or three days if the programmer must deal with constant interruptions.
- It can take even longer if the programmer works on another task at the same time.
- Part of the secret to making good estimates is to predict the effort, not the calendar time, that a project will take.

- Make your estimates in terms of ideal engineering days (often called story points), which are the number of days a task would take if you focused on it entirely and experienced no interruptions
- Using ideal time alone won't lead to accurate estimates. I've asked some teams I've worked with to measure exactly how long each task takes them (one team gave me 18 months of data), and even though we estimated in ideal time, the estimates were never accurate.
- Still, they were consistent. For example, one team always estimated their stories at about 60 percent of the time they actually needed.

- This may not sound very promising. How useful can inaccurate estimates be, especially if they don't correlate to calendar time? Velocity holds the key.

Velocity

- Although estimates are almost never accurate, they are consistently inaccurate.
- While the estimate accuracy of individual estimates is all over the map—one estimate might be half the actual time, another might be 20 percent more than the actual time—the estimates are consistent in aggregate.
- Additionally, although each iteration experiences a different set of interruptions, the amount of time required for the interruptions also tends to be consistent from iteration to iteration.

- As a result, you can reliably convert estimates to calendar time if you aggregate all the stories in an iteration. A single scaling factor is all you need.
- This is where velocity comes in. Your velocity is the number of story points you can complete in an iteration.
- It's a simple yet surprisingly sophisticated tool. It uses a feedback loop, which means every iteration's velocity reflects what the team actually achieved in the previous iteration.

Velocity and the Iteration Timebox

- Velocity relies upon a strict iteration timebox. To make velocity work, never count stories that aren't "done done" at the end of the iteration.
- Never allow the iteration deadline to slip, not even by a few hours.
- You may be tempted to cheat a bit and work longer hours, or to slip the iteration deadline, in order to finish your stories and make your velocity a little higher. Don't do that! Artificially raising velocity sabotages the equilibrium of the feedback cycle.
- If you continue to do it, your velocity will gyrate out of control, which will likely reduce your capacity for energized work in the process. This will further damage your equilibrium and your ability to meet your commitments

- Velocity tends to be unstable at the beginning of a project. Give it three or four iterations to stabilize.
- After that point, you should achieve the same velocity every iteration, unless there's a holiday during the iteration.
- Use your iteration slack to ensure that you consistently meet your commitments every iteration.
- I look for deeper problems if the team's velocity changes more than one or twice per quarter.

TIPS FOR ACCURATE ESTIMATES

You can have accurate estimates if you:

1. Estimate in terms of ideal engineering days (story points), not calendar time
2. Use velocity to determine how many story points the team can finish in an iteration
3. Use iteration slack to smooth over surprises and deliver on time every iteration
4. Use risk management to adjust for risks to the overall release plan

How to Make Consistent Estimates

- There's a secret to estimating: experts automatically make consistent estimates.
- All you have to do is use a consistent estimating technique. When you estimate, pick a single, optimistic value.
- How long will the story take if you experience no interruptions, can pair with anyone else on the team, and everything goes well? There's no need to pad your estimates or provide a probabilistic range with this approach.
- Velocity automatically applies the appropriate amount of padding for short-term estimates and risk management adds padding for long-term estimates.

How to Estimate Stories

Estimate stories in story points. To begin, think of a story point as an ideal day.

- When you start estimating a story, imagine the engineering tasks you will need to implement it.
- Ask your on-site customers about their expectations, and focus on those things that would affect your estimate.
- If you encounter something that seems expensive, provide the customers with less costly alternatives.
- As you gain experience with the project, you will begin to make estimates intuitively rather than mentally breaking them down into engineering tasks.
- Often, you'll think in terms of similar stories rather than ideal days.

For example,

- you might think that a particular story is a typical report, and typical reports are one point.
- Or you might think that a story is a complicated report, but not twice as difficult as a regular report, and estimate it at one and a half points.
- Sometimes you will need more information to make an accurate estimate.
- In this case, make a note on the card. If you need more information from your customers, write “??” (for “unanswered questions”) in place of the estimate.
- If you need to research a piece of technology further, write “Spike” in place of the estimate and create a spike solution story.

How to Estimate Iteration Tasks

- During iteration planning, programmers create engineering tasks that allow them to deliver the stories planned for the iteration.
- Each engineering task is a concrete, technical task such as “update build script” or “implement domain logic.”
- Estimate engineering tasks in ideal hours rather than ideal days.
- When you think about the estimate, be sure to include everything necessary for the task to be “done done”—testing, customer review, and so forth.

Explaining Estimates

- It's almost a law of physics: customers and stakeholders are invariably disappointed with the amount of features their teams can provide.
- Sometimes they express that disappointment out loud.
- The best way to deal with this is to ignore the tone and treat the comments as straightforward requests for information.

- One common customer response is, “Why does that cost so much?” Resist the urge to defend yourself and your sacred honor, pause to collect your thoughts, then list the issues you considered when coming up with the estimate.
- Suggest options for reducing the cost of the story by reducing scope.

- Your explanation will usually satisfy your customers. In some cases, they’ll ask for more information.
- Again, treat these questions as simple requests.
- If there’s something you don’t know, admit it, and explain why you made your estimate anyway.
- If a question reveals something that you haven’t considered, change your estimate

How to Improve Your Velocity

- Your velocity can suffer for many reasons. The following options might allow you to improve your velocity.
- Pay down technical debt The most common technical problem is excessive technical debt (see “Technical Debt””). This has a greater impact on team productivity than any other factor does.
- Make code quality a priority and your velocity will improve dramatically. However, this isn’t a quick fix. Teams with excessive technical debt often have months—or even years—of cleanup ahead of them.
- Rather than stopping work to pay down technical debt, fix it incrementally. Iteration slack is the best way to do so, although you may not see a noticeable improvement for several months.

Improve customer involvement

- If your customers aren't available to answer questions when programmers need them, programmers have to either wait or make guesses about the answers.
- Both of these hurt velocity. To improve your velocity, make sure that a customer is always available to answer programmer questions.

Support energized work

- Tired, burned-out programmers make costly mistakes and don't put forth their full effort.
- If your organization has been putting pressure on the team, or if programmers have worked a lot of extra hours, shield the programmers from organizational pressure and consider instituting a no-overtime policy.

Offload programmer duties

- If programmers are the constraint for your team—as this book assumes—then hand any work that other people can do to other people.
- Find ways to excuse programmers from unnecessary meetings, shield them from interruptions, and have somebody else take care of organizational bureaucracy such as time sheets and expense reports.
- You could even hire an administrative assistant for the team to handle all non-project-related matters.

Provide needed resources

- Most programming teams have all the resources they need. However, if your programmers complain about slow computers, insufficient RAM, or unavailability of key materials, get those resources for them.
- It's always surprising when a company nickle-and-dimes its software teams.
- Does it make sense to save $5,000 in equipment costs if it costs your team half an hour per programmer every day? A team of six programmers will recoup that cost within a month.
- And what about the opportunity costs of delivering fewer features?

Add programmers (carefully)

- Velocity is related to the number of programmers on your team, but unless your project is woefully understaffed and experienced personnel are readily available, adding people won't make an immediate difference.
- As [Brooks] famously said, "Adding people to a late project only makes it later."
- Expect new employees to take a month or two to be productive.
- Pair programming, collective code ownership, and a shared workspace will help reduce that time, though adding junior programmers to the team can actually decrease productivity

- Likewise, adding to large teams can cause communication challenges that decrease productivity.
- Six programmers is my preferred size for an XP team, and I readily add good programmers to reach that number.
- Past 6, I am very cautious about adding programmers, and I avoid team sizes greater than 10 programmers.

Results

- When you estimate well, your velocity is consistent and predictable with each iteration.
- You make commitments and meet them reliably.
- Estimation is fast and easy, and you can estimate most stories in a minute or two.

Contraindications

- This approach to estimating assumes that programmers are the constraint .
- It also depends on fixed length iterations , small stories, and tasks. If these conditions aren't present, you need to use a different approach to estimating.
- This approach also requires trust: developers need to believe they can give accurate estimates without being attacked, and customers and stakeholders need to believe the developers are providing honest estimates.
- That trust may not be present at first, but if it doesn't develop, you'll run into trouble.
- Regardless of your approach to estimating, never use missed estimates to attack developers.
- This is a quick and easy way to destroy trust.

Alternatives

- There are many approaches to estimating. The one I've described has the benefit of being both accurate and simple.
- However, its dependency on release planning for long-term estimates makes it labor-intensive for initial project estimates.