An overview of extreme programming

One popular approach to product development, specific to software, is extreme programming (XP). Extreme programming takes the best practices of software development to an extreme level. Created in 1996 by Kent Beck, with the help of Ward Cunningham and Ron Jeffries, the principles of XP were originally described in Beck's 1999 book, *Extreme Programming Explained* (Addison-Wesley Professional), which has since been updated.

Discovering extreme programming principles

Basic approaches in extreme programming are based on Agile principles. These approaches are as follows:

- **Coding is the core activity.** Software code not only delivers the solution but can also be used to explore problems. For example, a programmer can explain a problem using code.
- **XP teams do lots of testing.** If doing just a little testing helps you identify some defects, a lot of testing will help you find more. In fact, developers don't start coding until they've worked out the success criteria for the requirement and designed the unit tests. A defect is not a failure of code; it's a failure to define the right test.
- **Communication between customer and programmer is direct.** The programmer must understand the business requirement to design a technical solution.
- **For complex systems, some level of overall design, beyond any specific function, is necessary.** In XP projects, the overall design is considered during regular *refactoring* — namely, using the process of systematically improving the code to enhance readability, reduce complexity, improve maintainability, and ensure extensibility across the entire code base.

TABLE 4-1 Key Practices of Extreme Programming

XP Practice	Underpinning Assumption
Planning game	All members of the team should participate in planning. No disconnect exists between business and technical people.
Whole team	The customer needs to be collocated (physically located together) with the development team and be available. This accessibility enables the team to ask more minor questions, quickly get answers, and ultimately deliver a product more aligned with customer expectations.
Coding standards	Use coding standards to empower developers to make decisions and to maintain consistency throughout the product; don't constantly reinvent the basics of how to develop products in your organization. Standard code identifiers and naming conventions are two examples of coding standards.
System metaphor	When describing how the system works, use an implied comparison, a simple story that is easily understood (for instance, "the system is like cooking a meal"). This provides additional context that the team can fall back on in all product discovery activities and discussions.
Collective code ownership	The entire team is responsible for the quality of code. Shared ownership and accountability bring about the best designs and highest quality. Any engineer can modify another engineer's code to enable progress to continue.
Sustainable pace	Overworked people are not effective. Too much work leads to mistakes, which leads to more work, which leads to more mistakes. Avoid working more than 40 hours per week for an extended period of time.

	activities and discussions.
Collective code ownership	The entire team is responsible for the quality of code. Shared ownership and accountability bring about the best designs and highest quality. Any engineer can modify another engineer's code to enable progress to continue.
Sustainable pace	Overworked people are not effective. Too much work leads to mistakes, which leads to more work, which leads to more mistakes. Avoid working more than 40 hours per week for an extended period of time.
Pair programming	Two people work together on a programming task. One person is strategic (the driver), and one person is tactical (the navigator). They explain their approach to each other. No piece of code is understood by only one person. Defects can more easily be found and fixed before merging and integrating code with the system.
Design improvement	Continuously improve design by refactoring code — removing duplications and inefficiencies within the code. A lean code base is simpler to maintain and operates more efficiently.
Simple design	The simpler the design, the lower the cost to change the software code.
Test-driven development (TDD)	Write automated customer acceptance and unit tests before you code anything. Write a test, run it, and watch it fail. Then write just enough code to make the test pass, refactoring until it does (red-green-clean). Test your success before you claim progress.
Continuous integration	Team members should be working from the latest code. Integrate code components across the development team as often as possible to identify issues and take corrective action before problems build on each other.
Small releases	Release value to the customer as often as possible. Some organizations release daily. Avoid building up large stores of unreleased code requiring extensive risky regression and integration efforts. Get feedback from your customer as early as possible, as often as possible.

Putting It All Together

All three agile approaches — lean, scrum, and extreme programming (XP) — have common threads. The biggest thing these approaches have in common is adherence to the Agile Manifesto and the 12 Agile Principles. Table 4-2 shows a few more of the similarities among the three approaches.

TABLE 4-2 Similarities between Lean, Scrum, and Extreme Programming

Lean	Scrum	Extreme Programming
Engaging everyone	Cross-functional development team	Entire team
		Collective ownership
Optimizing the whole	Product increment	Test-driven development
		Continuous integration
Delivering fast	Sprints of four weeks or less	Small release

- *Product vision statement* (elevator pitch, clear statement of direction for reaching the outer boundary of the project)
- *Product roadmap* (a representation of the features required to achieve the product vision)
- *Velocity* (a tool for scrum teams to plan the workload for each sprint and empirically predict the delivery of functionality long-term)
- *Release planning* (establishing a specific mid-range goal, the trigger for releasing functionality to the market)
- *User stories* (structuring requirements from an end-user's point of view to clarify business value)
- Relative estimation (using self-correcting relative complexity and effort rather than inaccurate absolute measures, which give a false sense of precision)
- Swarming (cross-functional teams working together on one requirement at a time until completion to get the job done faster)

Agile Environments in Action

- Creating your agile workspace
- Rediscovering low-tech communication and using the right high-tech communication
- Finding and using the tools you need

Creating the Physical Environment

success. You can even hire people who specialize in [illegible] work environments.

Collocating the team

If at all possible, the scrum team needs to be *collocated* — that is, physically located together. When a scrum team is collocated, the following practices are possible and significantly increase efficiency and effectiveness:

- Communicating face to face
- Physically standing up — rather than sitting — as a group for the daily scrum meeting (this keeps meetings brief and on topic)
- Using simple, low-tech tools for communication
- Getting real-time clarifications from scrum team members
- Being aware of what others are working on
- Asking for help with a task
- Supporting others with their tasks

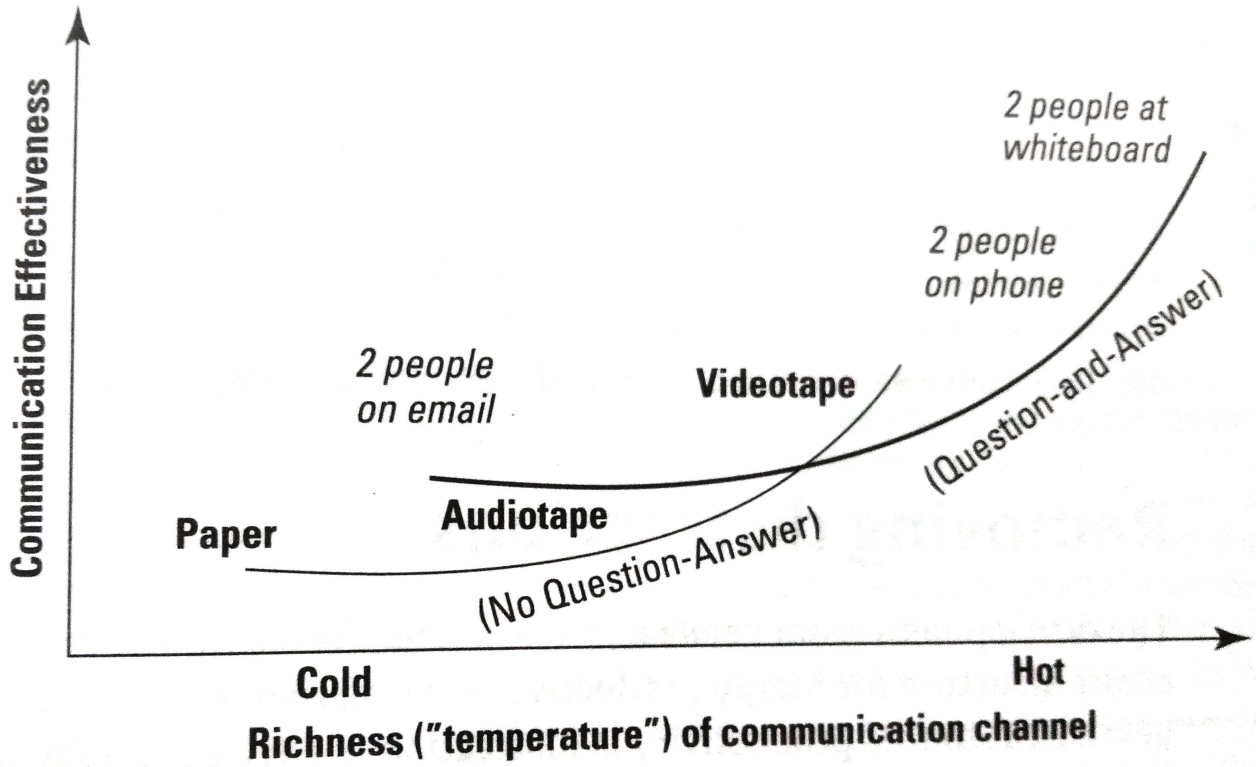

FIGURE 5-1: Better communication through collocation.

Setting up a dedicated area

If the scrum team members are in the same physical place, you want to create as ideal a working environment for them as you can. The first step is to create a dedicated area.

REMEMBER

The right space allows the scrum team to be fully immersed in solving problems and crafting solutions.

The situation you have may be far from

Removing distractions

The development team needs to focus, focus, focus. Agile methods are designed to create structure for highly productive work carried out in a specific way. The biggest threat to this productivity is distraction, such as . . . hold on a minute, I need to take a call.

TABLE 5-1 **Common Distractions**

Distraction	Do	Don't
Multiple projects	Do make sure that the development team is dedicated 100 percent to a single project at a time.	Don't fragment the development team between multiple projects, operations support, and special duties.
Multitasking	Do keep the development team focused on a single task, ideally developing one piece of functionality at a time. A task board can help keep track of the tasks in progress and quickly identify whether someone is working on multiple tasks at once.	Don't let the development team switch between requirements. Switching tasks creates a huge overhead (a minimum of 30 percent) in lost productivity.
Over-supervising	Do leave development team members alone after you collaborate on iteration goals; they can organize themselves. Watch their productivity skyrocket.	Don't interfere with the development team or allow others to do so. The daily scrum meeting provides ample opportunity to assess progress.
Outside influences	Do redirect any distracters. If a new task outside the sprint goal surfaces, ask the product owner to decide whether the task's priority is worth sacrificing sprint functionality.	Don't mess with the development team members and their work. They're pursuing the sprint goal, which is the top priority during an active sprint. Even a seemingly quick task can throw off work for an entire day.
Management	Do shield the development team from direct requests from management (unless management wants to give team members a bonus for their excellent performance).	Don't allow management to negatively affect the productivity of the development team. Make interrupting the development team the path of greatest resistance.

Going mobile

Judging by the "Going mobile" heading, you might have thought this section was about smartphone teleconferencing, but it isn't. Agile project teams take a responsive approach, and scrum team members require an environment that helps them respond to the project needs of the day. An agile team environment should be mobile — literally:

- Use movable desks and chairs so that people can move about and reconfigure the space.
- Get wirelessly connected laptops so that scrum team members can pick them up and move them about easily.
- Have a large mobile whiteboard. Also see the next section on low-tech communication.

Low-Tech Communicating

The primary tool for communication should be face-to-face conversation. Tackling problems in person is the best way to accelerate production:

- **Have short daily scrum meetings in person.** Some scrum teams stand throughout a meeting to discourage it from running longer than 15 minutes.
- **Ask the product owner questions.** Also, make sure he or she is involved in discussions about product features to provide clarity when necessary. The conversation shouldn't end when planning ends.
- **Communicate with your co-workers.** If you have questions about features, the project's progress, or integrating, communicate with co-workers. The entire development team is responsible for creating the product, and team members need to talk throughout the day.

Only a few tools are needed to support this low-tech communication:

- A whiteboard or two (ideally, mobile — on wheels or lightweight). Nothing beats a whiteboard for collaboration. The scrum team can use one for brainstorming solutions or sharing ideas.
- A huge supply of sticky notes in different colors (including poster-sized ones for communicating critical information you want readily visible — such as architecture, coding standards, and the project's definition of done).

  A personal favorite is giving each developer at least one tabletop dry erase/sticky note easel pad combination, with a lightweight easel. These low-cost tools facilitate communication fantastically.
- Lots of colorful pens.
- A sprint-specific task or kanban board (described in Chapters 4 and 9) for tracking progress tactility.

TECHNICAL STUFF

An *information radiator* is a tool that physically displays information to the scrum team and anyone else in the scrum team's work area. Information radiators include kanban boards, whiteboards, bulletin boards, *burndown charts*, which show the iteration's status, and any other sign with details about the project, the product, or the scrum team.

Basically, you move sticky notes or cards around the board to show the status (see Figure 5-2). Everyone knows how to read the board and how to act on what it shows. In Chapter 9, you find out the details of what to put on the boards.

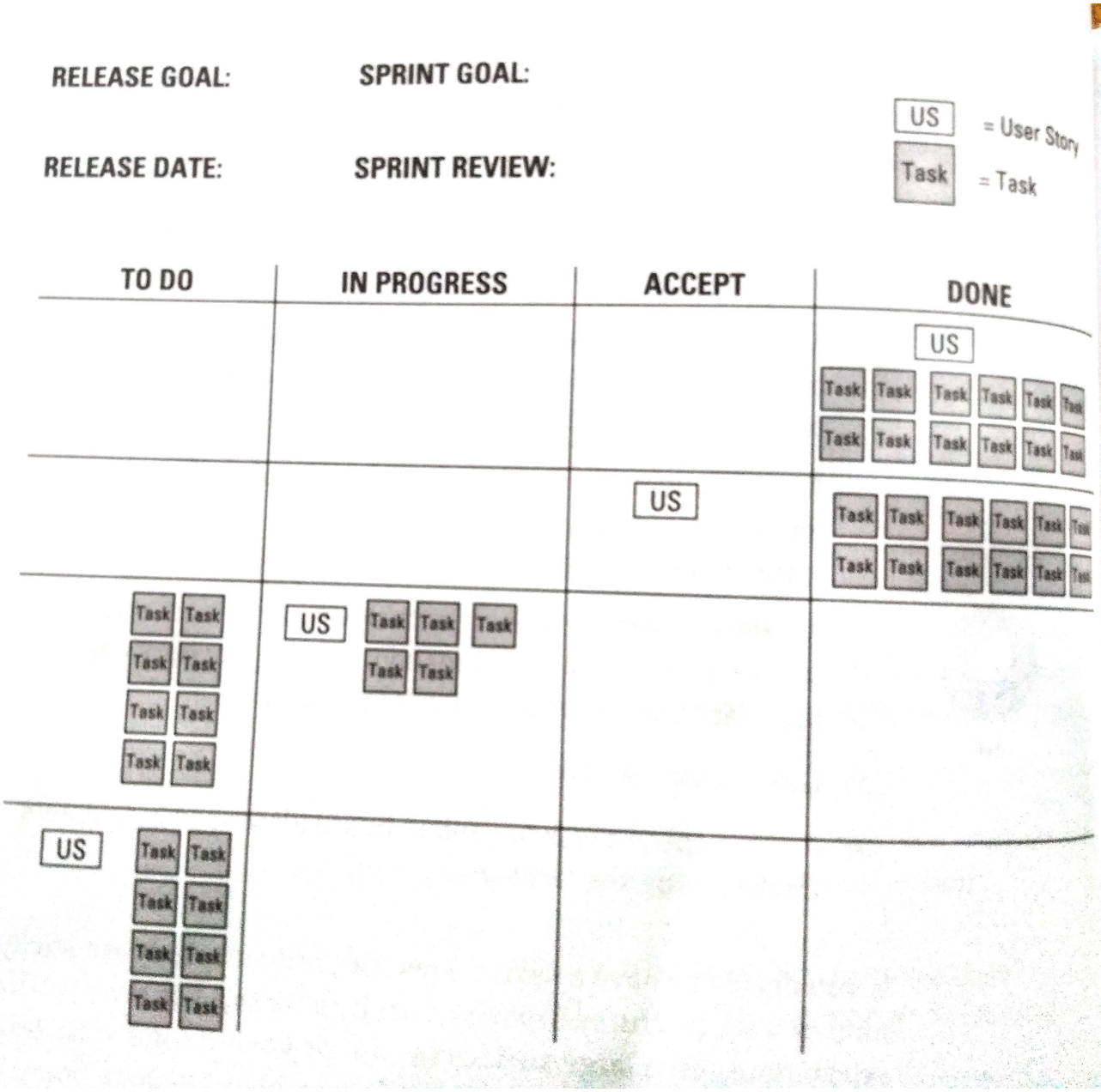

FIGURE 5-2: A scrum task board on a wall or whiteboard.

High-Tech Communicating

When determining which types of high-tech communication tools to support, first consider the loss of face-to-face discussions. Some tools you can use follow:

- **Videoconferencing and webcams:** These tools can create a sense of being together. If you have to communicate remotely, at the very least make sure you can see and hear each other clearly. Body language provides the majority of the message.
- **Instant messaging:** Although instant messaging doesn't convey nonverbal communication, it is real time, accessible, and easy to use. Several people can also share a session and share files.
- **Web-based desktop sharing:** Especially for the development team, sharing your desktop allows you to highlight issues and updates visually in real time. Seeing the problem is always better than just talking it out over the phone.
- **Collaboration websites:** These sites allow you to do everything from sharing simple documentation so that everyone has the latest information to using a virtual whiteboard for brainstorming.

TIP

Using a collaboration site (such as SharePoint, Confluence, and Google Drive) allows you to post documents that show the status of the sprint. When managers request status updates, you can simply direct them to the collaboration site to pull the information they need, on demand. By updating these documents daily, you provide managers with better information than they would have with formalized status reporting procedures under a traditional project management cycle. Avoid creating separate status reports for management; these reports duplicate information in the sprint burndowns and don't support production.