6

Agile practices: managerial

The agile principles imply, for a software development project, not only specific roles as studied in the previous chapter, but a set of concrete practices, such as the daily meeting, pair programming and test-driven development.

What, by the way, qualifies as a practice in software development? A practice has to be an activity or a mode of working, but with a special twist: repeated application. In the absence of repetition, we may have an interesting technique, but it is not a practice unless it is performed regularly (in the case of an activity) or enforced systematically (in the case of a mode of working).

Scrum also uses, for practices, the more picturesque name *ceremonies*.

We start in this chapter with practices affecting project organization and management. The following chapter will cover technical, software-specific practices.

6.1 SPRINT

One of the core principles of agile development is to work iteratively, producing frequent deliveries. All agile methods apply this idea, with various prescriptions for the duration of the individual iterations. To denote these iterations, the Scrum term “sprint” has come into wide use.

← *“Develop iteratively”, page 70.*

The purpose of a sprint is to advance the project by a significant increment, working from a task list, known in Scrum as the sprint backlog. In most agile approaches each task on the list is defined as the implementation of a “user story”.

6.1.1 Sprint basics

A Scrum sprint usually lasts one month. Many teams use other durations, and non-Scrum agile authors recommend iterations of varying lengths, although never more than a few weeks in line with the fundamental agile idea of short-cycled iterative development.

This idea of cutting up development into individual iterations lasting a month or so defines the notion of sprint, but a second property, particularly emphasized in Scrum, is just as important. It is the rule that *during a sprint, the task list does not grow.* The rule has to be absolute: no one, laborer, duke or emperor — or project manager — is permitted to add anything while the sprint is in progress.

This rule is made realistic by the short duration of sprints. Clearly, if iterations lasted six months, it would be impossible to repress the customers' and managers' natural urge to add functionality. With a one-month period, once everyone has signed on to the policy, the project may enforce the strict ban on extensions. No exceptions are allowed, whatever the rank of the supplicant. If there is a really pressing need, it gets parked until the end of the current sprint, and will be examined for possible inclusion in the next sprint. If not having the envisioned feature is a real show-stopper, then the only solution is the extreme one (akin, in the execution of a program, to raising an exception): terminating the sprint early — a decision that, as we have seen, is the privilege of the product owner. It is a pretty drastic decision; unless the product owner feels things are so critical as to justify it, he will just wait, like everyone else, until the next sprint.

6.1.2 The closed-window rule

The rule barring additions of functionality during a sprint follows from one of the principles we saw in an earlier chapter. It does not seem to have received a specific name in the agile literature but it is so important that it deserves one. Let us call it the **closed-window rule**: the window for changes is closed whenever a sprint is in progress.

← *"Freeze requirements during iterations", page 71.*

The closed-window rule addresses one of the biggest practical obstacles to successful software development: *disruptive feature creep*, more precisely disruptive customer- or management-induced feature creep. Customers and managers teem with ideas, and keep dreaming up new features. Giving them demos of early versions (in general a good practice, and strongly advocated in agile approaches) can make the phenomenon even worse by bringing to light what functionality is still missing. By itself the feature creep phenomenon is inevitable and in many respects healthy; a successful system will serve the business best if the key stakeholders have had their say. The problem is the disruptive nature of feature requests coming from a person carrying enough authority to change priorities. He or she comes up with a superb idea, so superb indeed that it has to be implemented right this minute at the expense of the currently scheduled tasks. Such interruptions can quickly derail a project: priorities get messed up, important work is delayed, and developers lose morale. But without a clear process such requests can be politically difficult to refuse.

The genius of the closed-window rule is that it neither ignores the risk of feature creep nor fights it head-on, but channels it into the limited framework of sprint planning exercises. A practical consequence is that a kind of natural selection takes place between feature ideas. Many a brilliant suggestion loses its luster when you look at it again after a few days, and when the time does come to select features for the next sprint it may no longer seem so urgent. Disruptions are avoided and noise takes care of itself. The ideas that were truly worthy of consideration get prioritized against all other tasks.

6.1.3 Sprint: an assessment

Two aspects are interesting to discuss: sprint duration, and the closed-window rule.

The one-month standard duration of sprints appears just right. In this book we often note that strict agile rules are too rigid, and sometimes see that the spirit is more important than the exact details; but in this particular case it appears that following the exact Scrum one-month prescription (including sprint planning and sprint review) works well. More precisely, Scrum specifies "thirty days"; I have found, as noted earlier, that it is more effective to use a calendar month. Simplicity breeds focus.

← *"Iteration length", page 71.*

The closed-window rule is an outstanding idea. While it contradicts the Agile Manifesto's principle A2, "*Welcome changing requirements, even late in development*", by conceding that not all change is welcome at all times, it provides a framework for handling change (or "harnessing" change, as the principle puts it).

← *Page 50.*

6.2 DAILY MEETING

A core agile practice is the daily meeting, also known as the "stand-up meeting" and as the "daily scrum". *Stand-up* because one of the original ideas was to make sure the meeting does not last long — fifteen minutes is the standard — by requiring everyone to stand; this requirement is impractical and usually not applied. *Scrum* because many groups use some approximation of the version fine-tuned by the Scrum method.

The rationale for meeting at the beginning of every workday is the general agile principle that direct contact is critical to project success. It meets here with the just as general agile distrust of heavy processes and such waste-inducing practices (think "lean") as long meetings. Hence the emphasis on both frequency and strict time limits. The method insists in particular on what a daily scrum is *not*: it is not intended to solve problems or engage in deep technical discussions. Its focus is precisely defined: answering the "**three questions**". What did you do on the previous working day? What will you do today? Any impediments?

The first two questions give the team the opportunity to catch up with each other on the progress of the project and its immediate future. They also help ensure that team members make realistic commitments and fulfill them, since today's answer to the second question, the promise, will meet tomorrow's answer to the first, the reckoning. As Cohn writes, the exercise is not a status update where a boss finds out who is behind schedule, but an opportunity for team members to make commitments to each other.

In the third question, an impediment is any obstacle that stands between a team member and the realization of his stated goals. There are technical impediments, such as problems with hardware or software products, and organizational impediments, such as the absence of a team member whose input is needed. The meeting should remove the

→ *"Impediment", 8.12, page 129.*

impediments when possible in the short time imparted, and otherwise assign responsibility for removing them. In Scrum, more specifically, removing impediments is one of the key responsibilities of the Scrum Master.

As emphasized by agile authors, one should be on the alert for practices that distort the purpose of the daily meeting and threaten its effectiveness. The two main threats are project members who go off into digressions, and the temptation to engage in deep technical discussions. Once you are aware of these risks, it is relatively easy to fend them off: the person in charge of enforcing good practices — in a traditional approach the project manager, and in Scrum the Scrum Master — can:

- Remind the ramblers to be concise; a more indirect technique is to enforce the time limit even (or *especially*) if this means that some people do not get to speak. It should not take more than one or two experiences of that kind for those who spoke too long to understand that they are the ones at fault. If it does, the team truly has a problem.
- If a technical discussion takes off on its own, intervene and suggest holding a separate meeting.

The idea of the daily meeting, with its focus on the three questions and the strict limitation of scope and duration, is brilliant. As with other agile ideas, you can stop listening to the advice when it becomes dogmatic. Some circumstances, such as geographically distributed projects, naturally lead to variations over the basic scheme:

- Setup time. A 15-minute meeting is fine for a resident team but generally not effective for a distributed team. Even with good technology and an experienced group of people, it can take a few minutes (“*Can you hear me?*”, “*Let’s switch from Skype to WebEx*”, “*The video conference room is still occupied*”) to get down to business.
- Flexible working schedules. In many organizations, some employees come in at different times or occasionally work at home. Such practices contradict the agile insistence on direct personal communication, but they have other justifications, such as the desirability of a “sustainable pace”, and companies may legally be required to allow them.

← *“Maintain a sustainable pace”, 4.4.3, page 56.*

- Time zones. Consider a group with some members in California and others in Shanghai. 7 AM for the former means (in the winter) 11 PM for the latter. You can ask people to be up late once in a while, but not every day.
- Meeting inflation. While there are good reasons for moving deeper technical discussions to a separate meeting, they should be balanced with the overhead of organizing separate meetings (“*Let’s discuss this on Tuesday afternoon — Tuesday I am not here, can you make it Wednesday at 10? — Yes, but I think the meeting room is not available*” and so on), plus the context-switching time (the time for everyone to remember what it was all about). Sometimes when an issue can be solved by a 20-minute discussion it is just as simple to have that discussion then and there.
- Length variability. There is no reason to use the same limits regardless of team size. 15 minutes may be fine for a group of five people and too short for ten.

A distributed team I know, which works across three continents and has honed its process over several years, has two weekly meetings, Monday and Thursday, at a time that is acceptable in all time zones affected. Both last one hour for the reasons just mentioned. They have complementary goals:

- The Monday meeting is developer- and deadline-based. Its purpose is to check progress towards the next deadline. It is run in the spirit of a Scrum daily meeting: each member of the team presents his or her current status based on the "three questions". Since it uses a full hour, technical discussions are not prohibited as long as they remain short; anything that requires deeper analysis is moved to the Thursday meeting or some other medium (such as an email discussion, or an extraordinary meeting). The team long ago learned to make good use of the available time and never overruns the one-hour limit. There is no agenda for those meetings; they are organized around the task list, a shared document that everyone can consult (through screen sharing) during the meeting.
- In contrast, the Thursday meeting is agenda-based; it is devoted to the discussion of a list of issues collected in advance by the meeting secretary (a task that rotates between members of the group). Its decisions are recorded as "action items" in the minutes (produced in real time during the meeting), and copy-pasted to the agenda of the next meeting so that the first matter of the day is to check what has been promised, just as in a daily meeting.

This particular formula, obtained by trial and error (as well as reading agile and other software books) works well for that particular group. A team subject to different constraints will fine-tune its own variant of the daily meeting idea. Freed of dogmatism — adapted in particular to the multi-site, flexible-personal-schedule working style of modern companies — that idea, particularly its focus on the "three questions", is one of the principal contributions of the agile school. Some day, the whole industry will be practicing it and not even conceive that anyone could ever have been working otherwise.

6.3 PLANNING GAME

The next two practices to be reviewed (in this section and the next) address one of the toughest challenges of software management and development: estimating the cost of a system to be developed, or part of that system. The planning game comes from Extreme Programming, the planning poker from Scrum. Cost estimation, the goal in both cases, is only a subset of what "planning" normally covers; but this limited scope of the term is consistent with the rest of the agile creed, which does not like the idea of upfront tasks.

The unit of estimation has traditionally been a unit of work: person-month or, at a finer level of granularity, developer-day (one programmer working for one day). More sophisticated metrics have been developed recently, in particular the *story point*, which we will study in the discussion of artifacts. The discussion in this section and the next does not depend on the particular metric used.

→ *"Story points", 8.4, page 121.*

The XP planning game is a "game" not in the sense of a competition, with winners and losers, but in the game theory sense of a cooperative game, where two actors try to maximize different criteria and seek an optimal compromise between them. The two actors are "business" and "development" in Beck's term, or more simply the customer and developer groups. The customers seek to maximize functionality and minimize the time to obtain it. The developers understand the difficulty level associated with every element of functionality, and the incompressible time that it requires. In the game:

- Customers define the respective priority of a set of functionality elements — defined in agile style as user stories — for a project, or a particular iteration.
- Developers estimate the cost (person-days) of implementing each story.

In playing the game, the two groups perform these tasks repeatedly, engaging into negotiation over the estimates. Customers sort the stories on the basis of priority. The game terminates when the two sides have agreed to select the highest-priority tasks with a total cost that fits within the time allotted for the release and the number of developers. In a variant of the game, the result is not so strictly tied to a release cycle but simply consists of a prioritized list of user stories.

6.4 PLANNING POKER

Scrum's planning poker is another approach to the same problem as XP's planning game, how to estimate the cost of user stories in advance. Again the discussion does not depend on the choice of measurement unit, such as developer-day or story point.

The two ideas of planning poker are to:

- Rely on the collective judgement of a panel of estimators, iterating until they agree.
- Avoid pointless haggling over small differences by forcing the values to be taken from a sequence of clearly distinct values.

A sequence of values satisfying this criterion is the Fibonacci sequence: 0, 1 (and 1 again), 2, 3, 5, 8, 13, 21, 35, …

> I hear you: that is not the Fibonacci sequence! Indeed. The last number cited should be 34. Congratulations on your mathematical sophistication! But one agile consultant has had the brilliant business idea of producing and selling a deck of planning-poker playing cards. Trouble is, copyrighting the Fibonacci sequence is kind of hard, since it has been around since something like 1202 in Italy (and a couple of millennia earlier in India). Not to worry: just change one of the values. Not exactly as I did above — I am far too scared of a copyright infringement suit! — but you get the idea.

If estimates are done in person-days, the second value is sometimes replaced by 0.5 since some simple user stories may be implementable in less than a day. What matters is that the values are sufficiently different to avoid the estimators getting into a fight over insignificant differences, such as whether a particular task will take 11 or 12 days; the aim is rough-cut estimation rather than exactness.

Some variants of planning poker rely on an even smaller set of choices, in particular "T-shirt sizing" which offers five values from X-small to X-large. Most variants also include the value "?" for the benefit of an estimator who feels there is not enough information yet to propose an answer.

The panel of estimators is the development team, including the product owner and other customer representatives as appropriate. It applies a form of the "Delphi" expert-consensus decision method, which originated with the US military and has been in use for decades. It is also influenced by the more recent concept of "wisdom of the crowds", according to which a group can collectively reach a better decision than even the best individual experts in its midst. The goal is to arrive at a consensus, but to avoid reaching it through the intimidation of outlying thinkers by the initial majority.

[Surowiecki 2004]

The process for estimating the cost of a functionality element involves the following steps:

1 Someone, typically the product owner, describes the feature.

2 The participants discuss it and ask questions as needed.

3 Every participant privately picks an estimate, from the preset sequence of values.

4 The choices are revealed. This where the process gets its name: as in a game of cards, you show your hand only when asked.

5 If the values agree, the process stops for this item and the common estimate is retained. (This is where it is important to have widely separated values in the sequence.)

6 If the values are not identical, a discussion takes place, with each member arguing for his or her choice. Then the process is repeated from step 3, on the basis of information gained in the discussion.

7 If the process does not converge to a common value, the participants will have to abandon it and discuss what else to do, such as getting more information and postponing the estimation to a later date.

Cohn states that

> *Teams estimating with Planning Poker consistently report that they arrive at more accurate estimates than with any technique they'd used before*

[Cohn site].

without, however, citing actual studies. My own experience, also individual and also not backed by studies, is less thrilling. The problem I have seen is the power of majority pressure. If you are truly an expert and you come up with an estimate that is widely different from those of the rest of the group, it is difficult to argue for long without appearing arrogant. To preserve group harmony you are naturally led to give up — at least if you know you are not yourself going to get the task of implementing the item. This outcome can be damaging to the project, especially when the expert knows how hard some task really is but is unable to convince the rest of the group, which has not performed such work before and thinks it will be a breeze.

6.5 ONSITE CUSTOMER

All agile methods, as we have seen, recommend involving customers or their representatives in the project. XP in particular has the notion of an "active customer", also known as an embedded customer. This practice is mentioned here as a reminder since an earlier chapter discussed the corresponding roles: "customer" and "product owner".

← *Chapter 5, particularly 5.2 and 5.5.*

6.6 OPEN SPACE

Agile methods put considerable emphasis on the physical organization of the workspace.

Many development teams traditionally use, at least in the US, private offices for the lead people and cubicles for everyone else. (Cubicles are less common in Europe, and the more extreme formats are incompatible with local labor laws; some countries, for example, require providing every office worker with access to daylight.)

Closed offices and cubicles are anathema to agile development. Because of the core role of communication, it is a tenet that developers should work in an open space. Here is a typical exhortation:

> *Use open working environments. Such environments allow people to communicate more easily* [*and*] *get together, and facilitate self-organization. When I walk into open areas, I can immediately tell how the team is doing. Silence is always a bad sign. I know that people are collaborating if I can hear conversations. When I enter a cubicle environment, there is often silence indicating an absence of interaction. Cubicles are truly the bane of the modern workplace. They quite literally keep people apart and break teams up.*

[Schwaber 2002], page 39.

In the recommended agile layout:

- The development area is a large room.
- Developers are seated at desks not too far from each other. If the team practices pair programming, there will be two developers at each desk, but in any case people should be able to hear conversations at neighboring desks and spontaneously join them.
- The walls are largely covered with whiteboards to support technical discussions.
- A quiet meeting room is available for technical meetings.

Many developers, in my experience, like this kind of arrangement, contradicting the stereotype of programmers as inward-looking nerds. Many does not mean all; witness the frequent practice of wearing noise-reduction headphones. Some agile authors recognize the need for occasional isolation, "cones of silence" in Cockburn's terms.

[Cockburn 2003].

Indeed, while the basic idea is sound, and cubicles deserve all the scorn they get from agile critics, it would be nice if everyone would follow Cockburn's example and refrain from sweeping absolutes. Open spaces are not the solution for all people and all times. It is impossible to take Schwaber's "*Silence is always a bad sign*" as a serious statement.

Software development is a challenging intellectual activity. There is the engineering part, which often requires "*communication*", "*collaboration*", "*interaction*" and "*conversation*", and the research part, which is in many respects akin to doing mathematics. There is a time for talking, and a time for concentrating. Some people think best by explaining their thinking to someone else, pair-programming style; some people think best while walking (like Napoleon); and some people think best by shutting themselves off from the world for a while. Most people think best by alternating between various models.

We have all met instances of the shy, introverted programmer who stays silent during meetings and one morning comes in with an impeccably designed and implemented subsystem, which all the "conversations" in the world would never have produced. It is part of the respect due to programmers (as advocated forcefully in the Crystal method) to accept that people are different and not to force a single scheme on them. Sure, you can gently nudge the silent genius, once in a while, to communicate a bit more. But if you start harassing him by enforcing a communicate-at-all-costs policy, all you will get is that he will soon take his talents to a more accommodating environment.

The gentle nudging, by the way, may have to apply to both sides. An incessant chatterer may fulfill the agile ideal of "*valuing interaction*", as the Agile Manifesto has it, but may become a serious obstacle to the project's progress, and deserve an encouragement to stop talking and produce something for a change.

If silence is "*always a bad thing*", what of the reverse situation: a workplace where everyone is babbling all the time? It is just as alarming. A healthy environment, in my experience, is one in which sometimes people talk and sometimes they silently read, or write, or just think. When "*walking into*" a development space and seeing a programmer who is just staring at the ceiling, only a naïve (and mean, and incompetent) manager jumps to the conclusion that the programmer is wasting the company's money.

The need for flexibility comes not only from developers' personality traits but from the nature of the tasks at hand. Requirements definition calls for lots of interaction (although even here quiet thinking, to classify and abstract information, is essential); design and implementation call for lots of thinking (although even here communication, of the kind advocated by agile methods, is essential).

These reservations do not affect the essential soundness of the agile view: open spaces often work well. Just do not turn the idea into a dogma. Different people, different circumstances and different times during projects call for different solutions.

6.7 PROCESS MINIATURE

Agile training frequently uses a technique that Cockburn calls "process miniature": get familiar with a proposed software process by applying it to some non-software tasks over a short period, such as a day, an hour or even less. Scrum tutorial sessions, for example, are notorious for asking participants to design paper planes by applying the Scrum roles, principles and practices. Throwing the planes around is great fun.

[Cockburn 2005], page 91.

Process miniature can be a good way to visualize techniques that might otherwise appear abstract, and understand the dynamics of group interaction in a self-organizing team. One should not forget, however, that it is just a simulation, and that the most serious issues, technical and personal, will only materialize in the thick of a real project. Building paper planes is not quite the same as building planes.

6.8 ITERATION PLANNING

A number of agile practices take the form of regular meetings. We have already seen the "daily meeting", but there are others, codified in particular by Scrum.

At the start of an iteration (a sprint in Scrum) there should be a meeting to plan that iteration. The meeting should produce three main outcomes:

1 An **iteration goal**, describing what the team plans to achieve in the iterations, concisely — a sentence or two — and in terms understandable by ordinary stakeholders. A typical example (assuming a compiler project) is: implement the new functional-language extensions.

2 An **iteration backlog**: the list of tasks to be implemented. This outcome is primarily for the internal benefit of the team.

3 The **list of acceptance criteria** for each task.

Conspicuously absent from these goals are: the *assignment of tasks* to individual team members, which will be done at the "last possible moment" according to the rule of cross-functionality; and a list of *testing tasks*, since testing is done continuously as part of the implementation of user stories, not as a separate activity.

The meeting is primarily reserved for the team and the product owner. As the team will be responsible for implementing the backlog in the allotted time, the result represents a commitment on its part, normally ruling out the participation of observers.

← *"Members and observers", 5.4, page 82.*

The definition of tasks (outcome 2 above) is a two-step process: select user stories from the backlog for the entire product; then, decompose each of them into tasks.

The process also requires estimating the cost of each task. This is where techniques such as the planning game and planning poker, discussed earlier in this chapter, come into play. Because the team is in the best position to size up tasks that it will have to implement, the product owner may at times be asked to leave the meeting while this estimation is in progress. Disagreements may imply repetitive application of the process.

To avoid endless discussions, the meeting has a time limit, generally a single day (eight hours), sometimes split into two parts, one for selecting user stories and the other for decomposing them into tasks.

6.9 REVIEW MEETING

Scrum

The review meeting mirrors, at the end of a sprint, the planning meeting performed at the beginning. Its purpose is to assess what has actually been done.

In the meeting, the development team presents to outside stakeholders, and in particular to the product owner in Scrum, the results of the sprint. It discusses what has been achieved, and not, against the original goals, cost estimates and acceptance criteria.

Such a review meeting is focused on results, not process. An end of sprint is also a good opportunity to reflect, beyond what has been done, on how it was done. In Scrum a separate meeting is reserved for that purpose: the retrospective.

6.10 RETROSPECTIVE

Scrum

A sprint retrospective reviews what went well and less well during the latest sprint, with a view to identifying what can be improved for the next one. The purpose is similar to what we find at level 5, "Optimizing", of the CMMI: integrating into the process (even if this word is not welcome in agile circles) a feedback loop so that it can improve itself.

← *"CMMI in plain English", 3.6.1, page 44.*

Whereas a review meeting requires the presence of the product owner (or, outside of Scrum, other stakeholders representing the viewpoint of the customer), a retrospective meeting is inward-looking and hence should primarily include the team and coach (Scrum Master), although the product owner may attend.

6.11 SCRUM OF SCRUMS

Scrum

Basic agile techniques are intended for small teams, up to about 10 people. The question arises of how to scale up to larger projects. The Scrum answer is worth studying here. It is known as a "scrum of scrums", defined as

> *a daily scrum consisting of one member from each team in a multi-team project.*

[Schwaber 2004], page 44.

except that "daily" is according to Larman too high a frequency; two or three times a week is enough.

[Larman 2010], page 200.

The challenge confronting scrums of scrums is coordination. It manifests itself in two ways:

- Interface changes.
- Dependencies between sub-projects.

Regular meetings are an effective way to address the first problem; if you make sure that API changes that can break client code are clearly publicized (and, if possible, discussed in advance), you avoid a serious source of trouble.

On the second problem, the best agile answer that I have seen is that dependencies should be avoided. According to Schwaber:

> *Before a project officially begins, the planners parse the work among teams to minimize dependencies. Teams then work on parts of the project architecture that are orthogonal to each other. However, this coordination mechanism is effective only when there are minor couplings or dependencies that require resolution.*

[Schwaber 2004], page 44.

Quite true; dividing the project into "orthogonal" parts works only if the complexity is of the additive kind. But of course a large project is usually large because it is truly — that is, multiplicatively — complex, and then the dependencies will be tricky. Although the agile literature claims that Scrum, XP and other methods can scale up, and gives examples of successful large projects, it provides little guidance on how to tackle the issues. As described in its own texts, the agile approach mostly targets projects involving a small group of developers.

← *"Additive and multiplicative complexity: the lasagne and the linguine", page 63.*

6.12 COLLECTIVE CODE OWNERSHIP

We end this review of management-related agile practices with an agile prescription that could also be classified as a principle, although it enjoys neither the same importance nor the same general application as the principles of the previous chapter.

In many projects every software module or subsystem is under the responsibility of a specific person. A typical comment in dealing with teams at Microsoft is "*If you want to change something to that API, you will have to convince Liz, she owns that piece*". She does not "own" it in the sense of intellectual property but in the sense of technical authority: who decides whether to accept a request for change. Code ownership in that sense is not restricted to commercial software: many open-source projects, such as Mozilla, also enforce a similar model, where:

> *A module owner's OK is required to check code into that module. In exchange, we expect the module owner to care about what goes in, respond to patches submitted by others, and be able to appreciate code developed by other people.*

[Mozilla modules].

6.12.1 The code ownership debate

Individual code ownership has clear benefits: someone is in charge, and will feel responsible for ensuring the consistency of the software and its integrity. One of the worst risks in the evolution of a software system is a general degradation due to inconsiderate extensions ("creeping featurism"); having a clear point of responsibility helps avoid it.

Individual code ownership can have negative consequences as well, emphasized by agilists and in particular by proponents of Extreme Programming: balkanization of the system, where each part of the code becomes a little fiefdom; concentration in one person of the expertise about each part of the system, raising a serious risk if that person leaves; and barriers to change, as the owner of a particular element (even if still a member of the team) may not be available or willing when others need a change, or they may simply not dare to ask.

XP promotes collective code ownership:

> *Anyone on the team can improve any part of the system at any time. If something is wrong with the system and fixing it is not out of scope for what I'm doing right now, I should go ahead and fix it.*

[Beck 2005], page 66.

This statement is in fact more nuanced than its predecessor in the first edition of the same book, which stated that "***anybody** who sees an opportunity to add value to **any** portion of **any** code is required to do so at **any** time*".

[Beck 2000], page 59, emphasis added.

Both versions surprisingly ignore the role of another core XP practice, pair programming, studied in the next chapter. In the actual application of XP as described by Cockburn, pair programming does temper the free-for-all:

> *XP has a strong ownership model:* ***Any two people sitting together and agreeing on it*** [the change] ***can change any line of code in the system***.

[Cockburn 2005], page 216. Emphasis in original.

This restriction seems to be the minimum needed for making collective code ownership reasonable. Even in a competent and self-organized team, it would be dangerous to allow arbitrary changes without involving at least a second pair of eyes. The free-for-all policy may have made the success of Wikipedia, but only with safeguards such as a vigilant community of millions of editors and thousands of administrators, and with generally less momentous consequences. (A mistyped digit in the population figure for the Duluth entry, even if it takes a few hours before someone detects it, should cause no tragedies. Program bugs are a serious matter.)

The Crystal method takes a more moderate attitude:

> *Most of the Crystal projects I have visited adopt the policy "change it, but let me know".*

Same reference as above.

In assessing the possible policies — personal ownership, collective ownership, and solutions in-between — it is important to note that preserving correctness is not the only issue. Agile methods require running the regression test suite regularly; so if as a result of a change-by-all policy someone messes up code that he does not completely understand, there is a good chance that the problem will be caught right away. A potentially more serious problem is degradation of the code, as described by Cockburn:

> [*If*] *everyone is allowed to add code to any class,* [*then*] *no one feels comfortable deleting someone else's code from the increasingly messy class. The result is* […] *like a refrigerator shared by several roommates: full of increasingly smelly things that almost everyone knows should be thrown out, but nobody actually throws out.*

Again from the same place.

Indeed a question more important than code ownership is change control. With modern configuration management tools it is possible to enforce specific rules automatically; for example you may prohibit committing a change unless at least one other person approves it. Google has such a rule. A more formal version requires a *review* of the code before it is committed; it is known as RTC, "Review Then Commit" and was Apache's initial policy. After complaints in 1998 that it was too constraining, Apache introduced the CTR option, "Commit Then Review", tempered by the possibility — seldom used but keeping programmers on their guard — of veto by any approved committer.

Every project should define its policy on this fundamental issue of change control, somewhere between the extremes of too much freedom, leading to code rot and bugs, and too much restriction, leading to an ossified process. The decision on code ownership should follow from this more fundamental policy, and also depends on other aspects of the company's or open-source project's culture. Once again a one-policy-fits-all rule, as prescribed here by Extreme Programming, does not survive objective analysis.

6.12.2 Collective ownership and cross-functionality

The extreme suggestion of letting anyone change anything becomes less surprising when viewed in light of another common agile practice: assigning the next task to the next available developer. Such an approach can only work if the developers are interchangeable; anyone can work on anything. This is the agile assumption of cross-functional teams: developers should remain generalists about the project, and not specialize in a narrow area.

← *"Cross-functional", 5.3.2, page 81.*

Arguments for and against cross-functionality are pretty much the same as those for and against individual code ownership. The risks of specialization are the emergence of jealously defended fiefdoms, and the dependency on individuals who may leave or be unavailable when the project needs them. On the other hand, a complex project will require highly focused competence in specific areas; it is inefficient to ask non-specialists to handle tasks in such an area, for which they will either botch the job or repeatedly disturb the expert. It is usually more productive to wait until that expert becomes available to do the job himself.

The application domain has a considerable influence on this discussion. When reading agile discussions, such as the recommendation of cross-functional teams, I sometimes have the impression that they are all based on consultants' experience with run-of-the-mill commercial developments for customers. In areas of advanced technical development, specialization is inevitable. If you are building an operating system and the next task involves updating the memory management scheme, you do not ask just anyone on the team. You ask the person who has devoted the last five years of his life to crafting the memory manager.