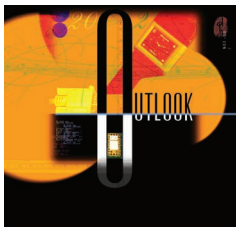
Get Ready for Agile Methods, with Care



Although many of their advocates consider the agile and plan-driven software development methods polar opposites, synthesizing the two can provide developers with a comprehensive spectrum of tools and options.

Barry Boehm
University of
Southern California

Faced with the conflicting pressures of accelerated product development and users who demand that increasingly vital systems be made ever more dependable, software development has been thrown into turmoil. Traditionalists advocate using extensive planning, codified processes, and rigorous reuse to make development an efficient and predictable activity that gradually matures toward perfection. Meanwhile, a new generation of developers cites the crushing weight of corporate bureaucracy, the rapid pace of information technology change, and the dehumanizing effects of detailed plan-driven development as cause for revolution. In their rallying cry, the Manifesto for Agile Software Development (http://www.agileAlliance.org), and in columns[1,2] such as those the "Ongoing Debate" sidebar describes, these developers call for a revitalized approach to development that dispenses with all but the essentials. Unsurprisingly, many developers who favor plan-driven methods have reacted to the manifesto with scathing criticism.

Real-world examples argue for and against agile methods. Responding to change has been cited as the critical technical success factor in the Internet browser battle between Microsoft and Netscape. But overresponding to change has been cited as the source of many software disasters, such as the $3 billion overrun of the US Federal Aviation Administration's Advanced Automation System for national air traffic control.

I believe that both agile and plan-driven approaches have a responsible center and overinterpreting radical fringes. Although each approach has a home ground of project characteristics within which it performs very well, and much better than the other, outside each approach's home ground, a combined approach is feasible and preferable.

THE PLANNING SPECTRUM

We can place various agile and plan-driven development approaches along a spectrum of increasing emphasis on plans, as Figure 1 shows. In this context, the term "plan" includes documented process procedures that involve tasks and *milestone plans*, and product development strategies that involve requirements, designs, and *architectural plans*.

Compared to unplanned and undisciplined hacking, agile methods emphasize a fair amount of planning. Their general view, though, places more value on the planning process than the resulting documentation,[1] so these methods often appear less plan oriented than they really are. Although hard-core hackers can use agile principles to claim that the work they do is agile, in general these methods have criteria, such as Extreme Programming's 12 practices, that help determine whether an organization is using an agile method or not.

Another encouraging trend is that the buzz of agile methods such as XP is drawing many young programmers away from the cowboy role model and toward the more responsible agile methods.

0018-9162/02/$17.00 © 2002 IEEE

The price that XP pays for this benefit, however, is a reluctance by more conservative managers to sanction a method called "extreme."[3]

Plan-driven methods also have a responsible center focused on major milestones and their overall content, rather than on micromilestones (or inch pebbles) locked into an ironbound contract. Excessively prespecified plans overconstrain the development team even at minor levels of change in personnel, technology, or commercial off-the-shelf upgrades. Such plans also provide a source of major contention, rework, and delay at high-change levels. On the other hand, when reviewing projects using the risk-driven spiral model, I find that to keep from losing their way, such projects, particularly larger ones, need to have at least three major anchor-point milestones to serve as project progress indicators and stakeholder commitment points.[4]

Figure 1 also shows the location of the current software Capability Maturity Model and its successor, the Capability Maturity Model Integrated, in the planning spectrum. Its proponents often use the software CMM in an overly restrictive and bureaucratic way, but it can be interpreted to include some forms of milestone risk-driven models. Software CMM leaders are also showing how liberal versus literal interpretations can include major portions of XP as being software CMM-compliant.[5] We can also interpret the CMMI, which adds process areas for risk management, integrated teaming, and an organizational environment for integration, to include agile methods.

COMPARING THE METHODS

The agile and plan-driven approaches each have strengths and weaknesses, as a direct comparison of several key areas shows.

Developers

Alistair Cockburn and Jim Highsmith emphasize several critical people factors for agile methods: amicability, talent, skill, and communication.[2] Larry Constantine's independent assessment identifies a potential problem for agile methods: "There are only so many Kent Becks in the world to lead the team. All of the agile methods put a premium on having premium people…."[6]

When you work with premium people, Highsmith and Cockburn's statement that "A few designers sitting together can produce a better design than each could produce alone"[1] rings true. Without premium people, however, you're more likely to get a design-by-committee mess. A significant consideration here is the unavoidable statistic that 49.9999 percent of the world's software developers are below average.

This is not to say that agile methods require uniformly high-capability people. Many agile projects have succeeded with mixes of experienced and junior people, as have plan-driven projects. The

Ongoing Debate

In the Software Management column in *Computer*'s September and November 2001 issues, Jim Highsmith and Alistair Cockburn summarized and rationalized the shared value propositions embodied in the Manifesto for Agile Software Development. The developers of several emerging software development methods, such as Adaptive Software Development (ASD), Agile Modeling, Crystal Methods, Dynamic System Development Methodology (DSDM), Extreme Programming (XP), Feature Driven Development, Lean Development, and Scrum emphasize the following values:

- *individuals and interactions* over processes and tools,
- *working software* over comprehensive documentation,
- *customer collaboration* over contract negotiation,
- *responding to change* over following a plan.

That is, while there is value in the items on the right, agile method proponents value the items on the left more.

Although their advocates have used each of these agile methods successfully in practice, critics who prefer process-based methods remain skeptical. In a letter[1] to *Computer*, Steven Rakitin indicates that in his experience, the items on the right *are* essential, while those on the left serve only as easy excuses for hackers to keep on irresponsibly throwing code together with no regard for engineering discipline. He provides "hacker interpretations" that turn agile value propositions such as "responding to change over following a plan" into chaos generators. Rakitin's hacker interpretation of "responding to change over following a plan" is roughly "Great! Now I have a reason to avoid planning and to just code up whatever comes next."

Reference

1. S. Rakitin, "Manifesto Elicits Cynicism," *Computer*, Dec. 2001, p. 4.

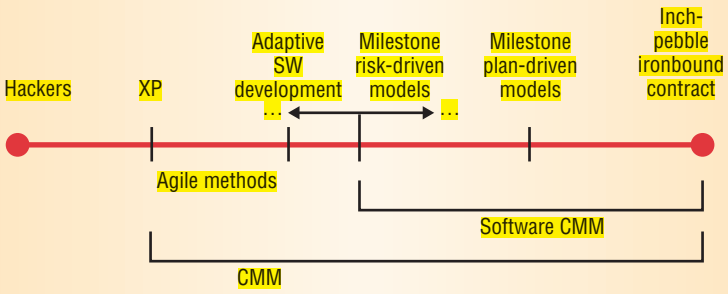


Figure 1. The planning spectrum. Unplanned and undisciplined hacking occupies the extreme left, while micromanaged milestone planning, also known as inch-pebble planning, occupies the extreme right.

Agile methods derive much of their agility by relying on the tacit knowledge embodied in the team, rather than writing the knowledge down in plans.

main difference is that agile methods derive much of their agility by relying on the tacit knowledge embodied in the team, rather than writing the knowledge down in plans.[7]

When the team's tacit knowledge is sufficient for the application's life-cycle needs, things work fine. But there is also the risk that the team will make irrecoverable architectural mistakes because of unrecognized shortfalls in its tacit knowledge. Plan-driven methods reduce this risk by investing in life-cycle architectures and plans, and using these to facilitate external-expert reviews. In doing so, however, they accept a risk that rapid change will make the plans obsolete or very expensive to keep up to date.

Customers

In USC's rapid-development electronic services projects, we have found that unless customer participants are committed, knowledgeable, collaborative, representative, and empowered, the developed products generally do not transition into use successfully, even though they may satisfy the customer. Participants in the XP 2001 "Customer Involvement in XP" workshop reached similar conclusions.[8] Agile methods work best when such customers operate in dedicated mode with the development team, and when their tacit knowledge is sufficient for the full span of the application. Again, though, these methods risk tacit knowledge shortfalls, which the plan-driven methods reduce via documentation and use of architecture review boards and independent expert project reviews to compensate for onsite customer shortfalls.

Requirements

Highsmith and Cockburn emphasize that agile approaches "are most applicable to turbulent, high-change environments." According to their world view, organizations are complex adaptive systems in which requirements are emergent rather than prespecifiable.[1] However, while agile manifesto tenets such as the second principle—welcome changing requirements, even late in development—offer great potential for success, developers can misapply them, with disastrous results.

Plan-driven methods work best when developers can determine the requirements in advance—including via prototyping—and when the requirements remain relatively stable, with change rates on the order of one percent per month. In the increasingly frequent situations in which the requirements change at a much higher rate than this, the traditional emphasis on having complete, consistent, precise, testable, and traceable requirements will encounter difficult to insurmountable requirements-update problems. Yet this emphasis is vital for stable, safety-critical embedded software.

Architecture

The agile manifesto values working software over comprehensive documentation, and emphasizes simplicity: maximizing the amount of work not done. This principle can be interpreted in many ways. Most are quite good, but some interpretations can cause problems. For example, based on the YAGNI precept: "You Aren't Going to Need It," XP advocates doing extra work to get rid of architectural features that do not support the system's current version. This approach works fine when future requirements are largely unpredictable. However, in situations where future requirements *are* predictable, this practice not only throws away valuable architectural support for them, it also creates problems with customers who want developers to believe that their priorities and evolution requirements are worth accommodating. Some agile methods such as Crystal and DSDM do more architectural preplanning.

As with requirements, plan-driven methods that emphasize heavyweight architecture and design documentation will encounter difficult to insurmountable problems in keeping up with rapidly changing requirements. On the other hand, if the architecture anticipates and accommodates requirements changes, plan-driven methods can keep even million-line applications within budget and schedule. Walker Royce provided a good example of such an effort in his description of the CCPDS-R project.[9]

Refactoring

With great developers and small systems, the assumption that refactoring is essentially free is valid. If so, the YAGNI approach is low-risk. However, empirical evidence indicates that with less-than-great developers, refactoring effort increases with the number of requirements or stories. For very large systems, our Pareto analysis of rework costs at TRW indicated that the 20 percent of the problems causing 80 percent of the rework came largely from "architecture-breakers," such as architecture discontinuities to accommodate performance, fault-tolerance, or security problems, in which no amount of refactoring could put Humpty Dumpty back together again.

Size

Cockburn and Highsmith conclude that "Agile development is more difficult for larger teams," but they cite occasional successful larger agile projects with up to 250 people.[2] Larry Constantine finds agile methods highly attractive for small projects, but he concludes that "The tightly coordinated teamwork needed for these methods to succeed becomes increasingly difficult beyond 15 or 20 developers."[6]

Plan-driven methods scale better to large projects like the million-line CCPDS-R project. But a bureaucratic, plan-driven organization that requires an average of a person-month just to get a project authorized and started won't be very efficient on small projects.

Primary objective

The first principle of the agile manifesto states that "Our highest priority is to satisfy the customer through early and continuous delivery of valuable software." "Early" and "continuous" are reasonably compatible goals for small systems developed by great people. But overfocus on early results in large systems can lead to major rework when the architecture doesn't scale up. In such cases, a good deal of planning will be necessary, in the spirit of the eighth principle of Lean Programming: "Ban Local Optimization."[10] Some agile methods are experimenting with concepts similar to the "software architecture skeleton" used in the CCPDS-R project.

Plan-driven methods are most needed in high-assurance software. Scott Ambler, the originator of agile modeling, says, "I would be leery of applying agile modeling to life-critical systems."[11] Another major set of objectives for the more traditional plan-driven methods, and for the software CMM, has been predictability, repeatability, and optimization. But in a world involving frequent, radical changes, having processes as repeatable and optimized as a dinosaur's may not be a good objective. Fortunately, the CMMI and associated risk-driven methods provide a way to migrate toward more adaptability.

BALANCING AGILITY AND DISCIPLINE

Particularly in the e-services sector, companies with a large customer base don't need just rapid value or high assurance—they need both. Pure agility or pure plan-driven discipline alone can't meet these needs. A mix of each is needed.

Martin Fowler's "Is Design Dead?"[12] essay and Jim Highsmith's *Adaptive Software Development*[13] book describe ways for combining agile and plan-driven methods. Risk management offers another approach that can help balance agility and discipline and answer questions such as "How much is enough?"[4] I apply risk management here to address the question "How much planning is enough?"

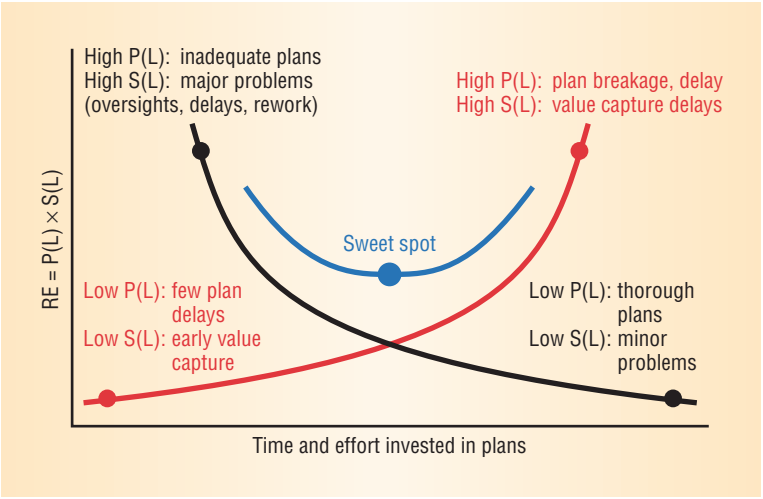


Figure 2. Risk exposure (RE) profile. This planning detail for a sample e-services company shows the probability of loss P(L) and size of loss S(L) for several significant factors.

A central concept in risk management involves determining the risk exposure for a given course of action. To determine RE, you assess the probability of loss, P(L), involved in a course of action, the corresponding size of loss, S(L), and then compute the risk exposure as the expected loss: RE = P(L) × S(L). Loss can include profits, reputation, quality of life, or other value-related attributes.

Figure 2 shows risk exposure profiles for a sample e-services company with

- a sizable installed base and thus the desire for high assurance,
- a rapidly changing marketplace and thus the desire for agility and rapid value, and
- an internationally distributed development team with a mix of skill levels and thus the need for some level of documented plans.

The black curve in Figure 2 shows the variation in risk exposure from inadequate plans as a function of the company's level of investment in its projects' process and product plans. At the left, a minimal investment corresponds to a high P(L) that the plans will have loss-causing gaps, ambiguities, and inconsistencies. It also corresponds to a high S(L) that these deficiencies will cause major project oversights, delays, and rework costs. At the right, we see that the more thorough the plans, the less probability that plan inadequacies will cause problems, and the smaller the size of the associated losses.

The red curve in Figure 2 shows the variation in RE from market share erosion caused by delays in

Table 1. Home ground for agile and plan-driven methods.

Home-ground area	Agile methods	Plan-driven methods
Developers	Agile, knowledgeable, collocated, and collaborative	Plan-oriented; adequate skills; access to external knowledge
Customers	Dedicated, knowledgeable, collocated, collaborative, representative, and empowered	Access to knowledgeable, collaborative, representative, and empowered customers
Requirements	Largely emergent; rapid change	Knowable early; largely stable
Architecture	Designed for current requirements	Designed for current and foreseeable requirements
Refactoring	Inexpensive	Expensive
Size	Smaller teams and products	Larger teams and products
Primary objective	Rapid value	High assurance

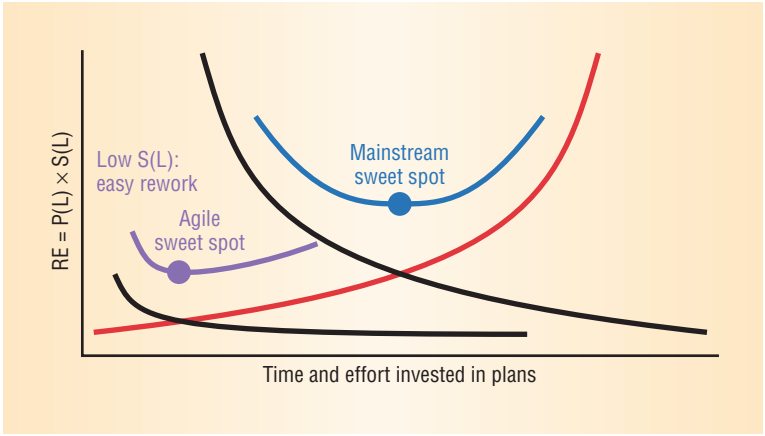


Figure 3. Comparative RE profile for an agile home-ground company with a small installed base and less need for high assurance.

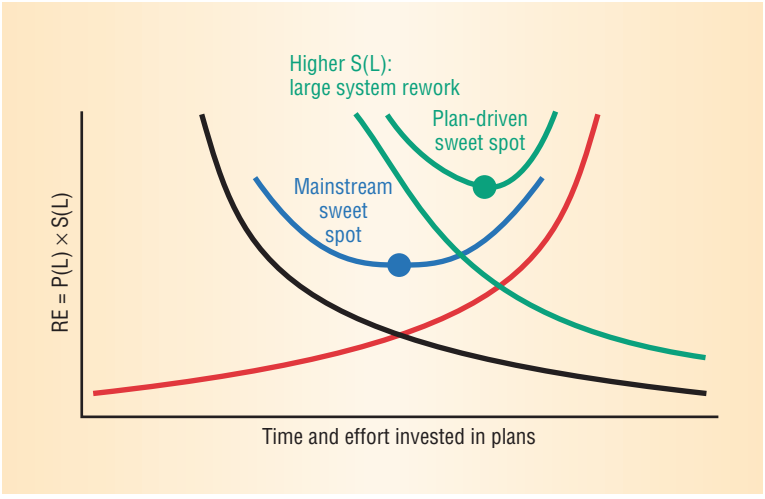


Figure 4. Comparative RE profile for a plan-driven home-ground company that produces large, safety-critical systems.

product introduction. Spending minimal time in planning will get at least a demo product into the marketplace early, enabling early value capture. Spending too much time in planning carries a high loss probability because of the time lost to planning and because rapid changes will cause delays via plan breakage. Longer planning times will also cause higher-sized losses because the delays will let stronger competitors capture most of the market share.

The blue curve in Figure 2 shows the sum of the risk exposures from inadequate plans and market share erosion. The very low and very high investments in plans have high overall risk exposures, and the sweet spot when overall risk exposure is minimized occurs in the middle, indicating "how much planning is enough" for this company's operating profile.

With the sample company situation as a reference point, we can run comparative risk exposure profiles for companies occupying the home ground of either agile methods or plan-driven methods, as summarized in Table 1.

For example, Figure 3 shows the comparative RE profile for an e-services company with a small installed base, a rapidly changing marketplace, a collocated team of highly capable and collaborative developers and customers, and less need for high assurance. With this profile, the major change in risk exposure from Figure 2 involves less rework loss from minimal plans because the team can rapidly replan and refactor, thus the company's sweet spot moves to the left, toward agile methods.

Figure 4 shows the corresponding RE profile for a company in the plan-driven home ground, with a more stable product line of larger, more safety-critical systems. Here, the major difference from Figure 2 involves more rework loss from minimal plans, with a resulting shift of the company's sweet spot toward higher investments in plans.

ASSESSING RISK EXPOSURE

In practice, quantifying estimates of P(L) and S(L) is difficult. But if you're combining agile and plan-driven methods, you can use each method's home-ground attributes to determine relative risks. So, for example, if your company or project fits an agile home-ground profile except for having a mix of equally important customers with less-than-ideal development representatives, you can reduce your risk exposure by conducting stakeholder requirements negotiations among the developers and customers. Then you can document the results to minimize the risk of future customer misunderstandings.

Or, if your project fits a plan-driven home-ground profile except for highly volatile user inter-

face requirements, you can use risk-driven specifications[4] to document those requirements that pose a high risk if they remain unspecified, such as critical message formats, but to avoid documenting volatile user interface requirements whose specification would be high risk. Risk-driven spiral methods and frameworks such as the Rational Unified Process (RUP),[9] Model-Based Architecting and Software Engineering (MBASE),[14] and the CMMI provide guidelines for finding a good balance of discipline and flexibility, although the current CMMI approach needs more explicit support for agility.

Agile and plan-driven methods both form part of the planning spectrum. Despite certain extreme terminology, each is part of the responsible center rather than the radical fringe. Indeed, agile methods perform a valuable service by drawing erstwhile cowboy programmers toward a more responsible center.

Both agile and plan-driven methods have a home ground of project characteristics in which each clearly works best, and where the other will have difficulties. Hybrid approaches that combine both methods are feasible and necessary for projects that combine a mix of agile and plan-driven home-ground characteristics. Risk analysis of your project's characteristics versus a given method's home-ground characteristics can help determine the best balance of agile and plan-driven disciplines.

Although information technology trends are moving us closer to agile methods' emergent requirements and rapid change home-ground characteristics, increasing dependability concerns call for measures best implemented with plan-based solutions. To meet these disparate needs, organizations must carefully evolve toward the best balance of agile and plan-driven methods that fits their situation. I expect we will see agile methods used increasingly for projects such as financial services; electronic commerce; air traffic control; and distributed, mobile, semiautomated, network-centric military or medical systems. ■

Acknowledgments

This work was supported by the National Science Foundation, the Department of Defense Software Intensive Systems Directorate, and the Affiliates of the USC Center for Software Engineering, http://sunset.usc.edu.

References

1. J. Highsmith and A. Cockburn, "Agile Software Development: The Business of Innovation," *Computer*, Sept. 2001, pp. 120-122.
2. A. Cockburn and J. Highsmith, "Agile Software Development: The People Factor," *Computer*, Nov. 2001, pp. 131-133.
3. L. Copeland, "Developers Approach Extreme Programming with Caution," *Computerworld*, 22 Oct. 2001, p. 7.
4. B. Boehm and W. Hansen, "The Spiral Model as a Tool for Evolutionary Acquisition," *CrossTalk*, May 2001, pp. 2-11; http://www.stsc.hill.af.mil/crosstalk/crostalk.html (current Dec. 2001).
5. M. Paulk, "Extreme Programming from a CMM Perspective," *IEEE Software*, Nov.-Dec. 2001, pp. 19-26.
6. L. Constantine, "Methodological Agility," *Software Development*, June 2001, pp. 67-69.
7. A. Cockburn, "Agile Software Development Joins the 'Would-Be Crowd,'" *Cutter IT J.*, vol. 15, no. 1, 2002, pp. 6-12.
8. A. van Deursen, "Customer Involvement in Extreme Programming: XP2001 Workshop Report," *ACM Software Eng. Notes*, Nov. 2001, pp. 70-73.
9. W.E. Royce, *Software Project Management: A Unified Framework*, Addison Wesley Longman, Reading, Mass., 1998.
10. M. Poppendieck, "Lean Programming: Part 2," *Software Development*, June 2001, pp. 71-75.
11. S. Ambler, "When Does(n't) Agile Modeling Make Sense?"; http://www.agilemodeling.com/essays/whenDoesAMWork.htm (current Dec. 2001).
12. M. Fowler, "Is Design Dead?" *Extreme Programming Explained*, G. Succi and M. Marchesi, eds., Addison Wesley Longman, Reading, Mass., 2001.
13. J. Highsmith, *Adaptive Software Development*, Dorset House, New York, 2000.
14. B. Boehm and D. Port, "Balancing Discipline and Flexibility with the Spiral Model and MBASE," *CrossTalk*, Dec. 2001, pp. 23-28; http://www.stsc.hill.af.mil/crosstalk/crostalk.html (current Dec. 2001).

Barry Boehm *is director of the University of Southern California Center for Software Engineering. Contact him at boehm@sunset.usc.edu.*