2

Contrasts and Home Grounds

Comparing agile and plan-driven approaches requires study and judgment

Comparison is difficult and imprecise

The complex nature of software development and the wide variety of methods make comparison of agile and plan-driven approaches difficult and imprecise. Nevertheless, we have found several important software project characteristics for which there are clear differences between agile and plan-driven methods. These are

- *Application characteristics,* including primary project goals, project size, and application environment.
- *Management characteristics,* including customer relations, planning and control, and project communications.
- *Technical characteristics,* including approaches to requirements definition, development, and test.
- *Personnel characteristics,* including customer characteristics, developer characteristics, and organizational culture.

In this chapter we discuss how agile and plan-driven methods approach each of these characteristics, giving examples to illustrate the differences. From that discussion, we consolidate our observations into specific home grounds for agile and plan-driven methods, and identify five critical factors that can be used to determine how a project or organization relates to those home grounds.

For those interested, Appendix E provides a summary of the small but growing body of empirical data available on agile and new plan-driven methods.

Application Characteristics

We have found a number of differences in the type of projects where each of the approaches has been successful. One area of difference is the appropriateness of the goals of each approach to those of the project. Other areas include the size of the project in terms of people, complexity and volume of software, and the type of business environment within which the project is developed.

Primary Goals

Agile goals are rapid value and responsiveness

The primary goals of agile methods are *rapid value* and *responsiveness to change.* The first of the 12 Agile Manifesto principles states, "Our highest priority is to satisfy the customer through early and continuous delivery of valuable software." Agile projects generally do not perform return-on-investment analyses to determine an optimal allocation of resources to deliver specific value. They prefer to build things quickly and find out through experience what activity or feature will add the most value next. This laissez-faire approach generally avoids the loss of large investments pursued on faulty assumptions, but can lead to local or short-term optimization problems that may impact the project adversely later on.

The fourth Agile Manifesto value proposition prefers "responding to change over following a plan." This is a reactive posture rather than a proactive strategy. In a world characterized by rapid changes in the marketplace, technology, or environment, such a reactive posture has considerable advantages over being locked into an obsolete plan. One downside is that reactive management with a flighty customer can result in an unstable, chaotic project. Another potential downside is an overemphasis on tactical over strategic objectives.

Reactive posture has advantages where change is rapid, but some risk

The primary goals of plan-driven methods are *predictability, stability,* and *high assurance.* The plans, work products, and verification and validation strategies of plan-driven methods support these goals. Process improvement, as represented by the SW-CMM and CMMI, focuses on predictability and stability by increasing process capability through standardization, measurement, and control. Prediction is based on the measurements of prior standard activities. Control is asserted when the current progress is outside of expected tolerances.

Plan-driven goals are predictability, stability, and high assurance

For relatively stable projects, the proactive investments in process and up-front plans by organizations implementing the CMM or CMMI can achieve predictability, stability, and high assurance. However, when confronted with unprecedented projects and high rates of unforeseeable change, these organizations find that predictability and stability degrade, and the project incurs significant expenditures in keeping processes relevant and plans up to date.

Proactive posture is effective with stability

For high-assurance, safety-critical projects, following a thorough, documented set of plans and specifications is the only way to meet existing certification standards such as RTCA DO-178B. These standards require strict adherence to process and specific types of documentation to achieve safety or security.

Plans and specifications are required for certification

Size

Agile works best on smaller projects

Currently, agile processes seem to work best with *small to medium* teams of people working on relatively small applications. In his landmark XP book, Kent Beck says, "Size clearly matters. You probably couldn't run an XP project with a hundred programmers. Not fifty. Nor twenty, probably. Ten is definitely doable."[1] The general consensus is that the tight coordination and shared knowledge generally prevents agile methods with teams over forty.[2, 3]

Scaling up has proven difficult

See Appendix E3

There have been occasional successful larger agile projects with up to 250 people. The highly successful 50-person Singapore lending application, and another successful 250-person banking application,[4] are good examples. However, the manager of both confessed, "I would never do [a 250-person project] again. It was way too big." The largest project to date used Scrum to develop a corporate portfolio of related applications involving around 800 developers at IDX, a medical information services company.[5] As shown in the 50-person XP case study in Chapter 4, a larger agile project needs to adopt traditional plans and specifications in order to deal with the increasingly complex, multidimensional interactions among the project's elements.

Traditional rigor is more effective on large projects

Traditional plan-driven methods scale better to *large* projects. The plans, documentation, and processes provide for better communication and coordination across large groups. However, a bureaucratic, plan-driven organization that requires an average of one person-month just to get a project authorized and started is not going to be very efficient on small projects.

Plan-driven is a necessity on large complex projects

For the extremely large, path-breaking U.S. Army/DARPA Future Combat Systems program, our Software Steering Committee recently participated in a 150-person, week-long review of the completeness and consistency of thousands of pages of specifications. The specifications dealt with 34 highly complex system elements such as robotic combat vehicles and integrated command-control vehicles. They were produced by multiple integrated product teams and were about to be released for subcontractor bids. The process produced around 2,000 problem reports, many of which, if issued in their current state, would have caused person-years of rework. We see no way to avoid an activity of this nature on extremely large systems of systems, and absolutely no way to handle the problem with agile standup meetings and tacit knowledge propagation.

Environment

Agile approaches are comfortable in high-change environments—with some risks

Agile approaches "are most applicable to *turbulent, high-change* environments," and have a world view that organizations are complex adaptive systems, in which requirements are emergent rather than prespecifiable.[6] However, "welcome changing requirements, even late in development," can be misapplied with disastrous results. One of the authors was involved in a review of software-caused rocket vehicle failures. The main cause by far was the inadequate testing, verification, and configuration management of last-minute changes—i.e., responding to change over following a plan.

Agile focuses on the product at hand

Agile methods concentrate on delivering a specific software product, on time, that completely satisfies a customer. As such, the scope of concern is focused on the *product at hand* and generally ignores problems that may occur later. There is little, if any, concern with the organization beyond the project except as supporting or interfering with the development. This works well at the project level, but as some users have found, ". . . early experience with elements of XP on departmental applications produced code that didn't integrate well with [the] company's overall infrastructure or scale in production."[7]

Agile successes have largely been in in-house environments

Agile methods have been almost entirely performed within in-house or dedicated development environments.[8] This makes it easier to have a close relationship to local users, but harder to perform various forms of distributed development, evolution, and usage.

Agile assumes flexible user system to accommodate evolution

Problems in applying an agile approach can manifest because it is assumed that the user's operational system will be flexible enough to accommodate unplanned evolution paths.[9, 10] This assumption fails in several circumstances:

- The need to overcome stovepipes, where several independently evolved applications must subsequently be closely integrated.
- "Information-sclerosis" cases, where temporary procedural workarounds caused by software deficiencies solidify into unchangeable constraints on system evolution and can cause unnecessary rework. The following comment is a typical example: "It's nice that you reprogrammed the software and changed those equipment codes to make them more intelligible for us, but the Codes Committee just met and established the current codes as company standards."
- Bridging situations, where the new software is incrementally replacing a large existing system. If the existing system is poorly modularized, it is difficult to undo the old software in ways that fit the expanding increments of new software.
- Monolithic requirements, such as the need for critical-mass core capabilities. For example, delivering 20 percent of an aircraft flight control module might not be practical.
- Continuity requirements, where you must maintain familiarity with the system across a large, mission-critical user base. Consider the safety risks in making significant monthly changes to air traffic controllers' operational procedures.

Plan-driven methods work best when the *requirements are largely determinable in advance* (including via prototyping) and remain *relatively stable.* Change rates on the order of 1 percent of the requirements per month are acceptable. Unfortunately, in the increasingly frequent situations where the rate of change is much higher than this, traditional methods designed for stable software begin to unravel. Time-consuming processes to ensure complete, consistent, precise, testable, and traceable requirements will face possibly insurmountable problems keeping up with the changes.

Plan-driven methods need stability

Plan-driven methods also cover a broader spectrum of activities than agile methods. Often used in contracted software development, they can address product line, organizational, and enterprise concerns that span multiple projects. In order to better handle this broader spectrum, plan-driven methods anticipate (plan for) future needs through architectures and extensible designs. They develop capabilities in related disciplines (e.g., systems engineering, human factors) and expect to impact a large number of people at various levels within the organizational hierarchy. Plan-driven organizations also usually exhibit a strong, quantitative process improvement focus.

Plan-driven scope includes system engineering, organization, outsourcing

See Appendix E3

Management Characteristics

There are a number of differences in the approaches with respect to how they are managed and to the expectations each has for the customer and other stakeholders. Planning, control, and communication are crucial to success, but are approached differently by the agile and plan-driven camps.

Customer Relations

Agile encourages a dedicated collocated customer

Agile methods strongly depend on *dedicated, collocated customer* representatives to keep the project focused on adding rapid value to the organization. This generally works very well at the project level.

The main agile stress point is the interface between the customer representative and the users

To succeed, agile customer representatives must be in synch with both the system users they represent and the development team. As will be seen in the lease management case study in Chapter 4, problems can arise when the representative does not accurately reflect the needs and desires of the users through lack of understanding or failure to keep abreast of the user concerns. Thus the customer representative becomes the primary stress point for agile methods.

Plan-driven methods depend on contracts and specifications

Plan-driven methods generally depend on some form of *contract* between the developers and customers as the basis for customer relations. They try to cope with foreseeable problems by working through them in advance and formalizing the solutions in a documented agreement. This has some advantages, particularly in stable situations. The developers know what they have to do. The customers know what they are getting and are better able to continue with their operational responsibilities. This maintains their operational knowledge and increases their value when they exercise prototypes or review progress and plans.

The main plan-driven stress point is the interface between the developer and the customer

However, the contract makes the developer-customer interface the major stress point for plan-driven methods. A precise contract causes startup delays and is harder to adapt to needed changes. An imprecise contract can create incompatible expectations, leading to adversarial relations and lack of trust. The worst case is when a tight, fixed-price contract results in the lawyers negotiating changes rather than the problem-solvers. Creative award-fee or profit-sharing contracts can help, but the presence of the contract remains a potential stress point on developer-customer trust.

For a project to succeed, the stakeholders must trust that the developing organization will perform the needed work for the available, agreed-to resources. Agile developers use *working software and customer participation* to instill trust in their track record, the systems they've developed, and the expertise of their people. This is one of the reasons agile projects have been primarily used for in-house development. It takes time to build up the level of trust required for the customer and developer to minimize contractual safeguards and detailed specifications. In many software development sectors, that length of time is simply not available.

Agile developers use working software to build customer trust

Plan-driven people count on their *process maturity* to provide confidence in their work. CMM appraisals are often used in source selection for large system implementation or for sourcing decisions, but one shouldn't assume documented plans guarantee the project will follow them. Although customers often feel secure when contracting with a SW-CMM Level 5 organization for software development, there have been a number of cases where the trust was misplaced. In several instances, particularly with offshore software organizations, customers have discovered that the "Level 5" software teams they hired were in fact raw, untrained new hires with little knowledge of CMM practices. Trust, in both plans and people, can have its limits.

Plan-driven developers use established process maturity to build customer trust

Planning and Control

In the agile world, planning is seen as a *means to an end* rather than a means of recording in text. Agilists estimate that their projects spend about 20 percent of their time planning or replanning. The agile projects' speed and agility come largely from deliberate group planning efforts that enable operation on the basis of tacit interpersonal knowledge rather than explicit documented knowledge as represented in plans and specifications. Many of the agile practices—pair programming, daily standup

Agilists see planning as a means to an end all-hands meetings, shared code ownership, collocated developers and customers, team planning—are as much about developing the team's shared tacit knowledge base as they are about getting work done. When unforeseen changes come, the team members can call upon their shared vision of the project's goals and their shared understanding of the software content to quickly develop and implement a revised solution. As the project scales up, however, this becomes increasingly difficult.

Plan-driven methods use plans to communicate and coordinate

Plan-driven methods use plans to *anchor their processes* and *provide broad-spectrum communication.* Plans make up a large portion of the required documentation in most plan-driven approaches. Plan-driven methods rely heavily on documented process plans (schedules, milestones, procedures) and product plans (requirements, architecture, standards) to keep everyone coordinated. Individual plans are often produced for specific activities and then integrated into "master plans."

Both use past performance to inform planning

Considerable effort is spent on maintaining historical data so that planning projections can be more accurate. The fundamental measure of progress is tracking of progress against plans. Planning is done and adjusted constantly. Plans make explicit the expectations and relationships between various project efforts, and between the project and other independently evolving systems. Remember, however, that for rapidly evolving systems, the more detailed the plans, the more expensive and time consuming the rework.

Agile is "planning driven," rather than "plan-driven"

With respect to distinctions in how agile and traditional methods process plans, Kent Beck, cocreator of XP, agreed with the distinctions in the following e-mail to us:

> I think the phrase "plan driven" is the key. I would characterize XP as "planning driven" in contrast. What XP teams find valuable is the collaboration, elicitation,

and balancing of priorities in the planning act itself. The plans that result have a short half-life, not because they are bad plans, but because their underlying assumptions have a short half-life.

Project Communication

Agile methods depend on tacit knowledge

Agile methods rely heavily on *tacit, interpersonal knowledge* for their success. They cultivate the development and use of tacit knowledge, depending on the understanding and experience of the people doing the work and their willingness to share it. Knowledge is specifically gathered through team planning and project reviews (an activity agilists refer to as "retrospection"). It is shared across the organization as experienced people work on more tasks with different people.

Communication is person-to-person and frequent

Agile methods generally rely on more frequent, person-to-person communication. As stated in the Agile Manifesto, emphasis is given to "individuals and interactions." Few of the agile communication channels are one-way, showing a preference for collaboration. Standup meetings, pair programming, and the planning game are all examples of the agile communication style and its investments in developing shared tacit knowledge.

Relying on tacit knowledge can be risky

Relying completely on tacit knowledge is like performing without a safety net. While things go well, you avoid the extra baggage and setup effort, but there may be situations that will make you wish for that net. Assuming that everyone's tacit knowledge is consistent across a large team is risky, and as people start rotating off the team, the risk gets higher.

Tacit knowledge is difficult to scale

At some point, a group's ability to function exclusively on tacit knowledge will run up against well-known scalability laws for group communication. For a team with N members, there are N(N–1)/2 different interpersonal communication paths to keep up to date. Even broadcast techniques, such as standup group meetings and hierarchical team-of-teams techniques, run into serious scalability problems. Consider the previously discussed Future Combat Systems subcontractor specification reviews as an example of a limiting case.

Plan-driven approaches use explicit, documented knowledge

Plan-driven methods rely heavily on *explicit documented knowledge.* With plan-driven methods, communication tends to be one-way. Communication is generally from one entity to another rather than between two entities. Process descriptions, progress reports, and the like are nearly always communicated as unidirectional flow.

Agile and plan-driven methods use both kinds of knowledge

We should note that this distinction between "agile-tacit" and "plan-driven-explicit" is not absolute. Agile methods' source code and test cases certainly qualify as explicit documented knowledge, and even the most rigorous plan-driven method does not try to get along without some interpersonal communication to ensure consistent, shared understanding of documentation intent and semantics.

Agile adds documentation when needed while plan-driven methods generally subtract what is not needed

When agile methods employ documentation, they emphasize doing the minimum essential amount. Unfortunately, most plan-driven methods suffer from a "tailoring-down" syndrome, which is sadly reinforced by most government procurement regulations. These plan-driven methods are developed by experts, who want them to provide users with guidance for most or all foreseeable situations. The experts therefore make them very comprehensive, but "tailorable-down" for less critical or less complex situations. The experts understand tailoring the methods and often provide guidelines and examples for others to use.

Once specified, removing documentation is difficult

Unfortunately, less expert and less self-confident developers, customers, and managers tend to see the full-up set of plans, specifications, and standards as a security blanket. At this point a sort of Gresham's Law

("Bad money drives out good money") takes over, and the least-expert participant generally drives the project to use the full-up set of documents rather than an appropriate subset. While the nonexperts rarely read the ever-growing stack of documents, they will maintain a false sense of security in the knowledge they have followed best practice to ensure project predictability and control. Needless to say, the expert methodologists are then frustrated with how their tailorable methods are used—and usually verbally abused—by developers and acquirers alike.

Shortfalls of tailor-down methods have been known for decades

This process has been going on for decades, in the United States from MIL-STD-1679 through -2167, -2167A, and -498, and through IEEE/EIA-016 and 12207. It is currently seen in nongovernment methods such as RUP, TSP, and one of the authors' methods, MBASE. Both authors thank the agilists for making it clear that a better approach to plan-driven methods is needed.

Technical Characteristics

Technical characteristics have been the focus of much of the debate as to the effectiveness of agile and plan-driven methods. This section looks at how each of the approaches handles requirements elicitation and management, development activities, and testing.

Requirements

Agile uses informal, user-prioritized stories as requirements

Most agile methods express requirements in terms of *adjustable, informal stories.* Agile methods count on their rapid iteration cycles to determine needed changes in the desired capability and to fix them in the next iteration. Determining the highest-priority set of requirements to be included in the next iteration is done collaboratively by the customers and developers. The customers express their strongest needs and the developers assess what combinations of capabilities are feasible for inclusion in the next development iteration (typically on the order of a month). Negotiations establish the contents of the next iteration.

Plan-driven methods prefer specific, formalized requirements

Plan-driven methods generally prefer *formally baselined, complete, consistent, traceable, and testable specifications.* For some time, the plan-driven world has been aware of collaborative requirements determination[11] but has been slow to respond. Because it was developed when the accepted practice was for the systems engineers to identify, define, and hand off the software requirements, the SW-CMM states, "Analysis and allocation of the system requirements is not the responsibility of the software engineering group but is a prerequisite for their work."[12]

Prioritization has not been widely used in plan-driven approaches

The plan-driven world has also been much slower than the agile world to assimilate new concepts such as prioritized requirements and evolutionary requirements. Some progress is being made with the introduction of the CMMI product suite because it extends beyond that of the SW-CMM. CMMI includes integrated teaming, requirements development, and risk management, all of which support the use of risk-driven, evolving requirements specifications over the traditional "complete" requirements specifications and their limitations.

Risk-driven approaches are similar to agile

Risk-driven approaches assert it is better not to specify elements where the risks of specifying them are larger than the risks of *not* specifying them.[13] For example, prematurely specifying a detailed graphical user interface runs the risk of breakage due to changing requirements, systems that are not responsive to the users, and solutions that can't evolve as the understanding of the system and its operational concept matures. The only risk in *not* specifying it is that it may take a few more passes with an automated interface creation tool. So, risk-driven approaches would rather you not specify the user interface early on.

The opposite case might apply to a mission-critical function or security requirement.

Plan-driven methods handle nonfunctional requirements better

On the other hand, the plan-driven world is considerably ahead of the agile world in dealing with quality or nonfunctional requirements such as reliability, throughput, real-time deadline satisfaction, or scalability. These become increasingly important for large, mission-critical systems and are a source of expensive architecture breakers when an initial simple design doesn't scale up. Most agilists participating in a recent Center for Empirically Based Software Engineering (CeBASE)* eWorkshop on this topic tended to consider a quality attribute as just another feature to be easily dealt with. Most plan-driven developers consider quality attributes as a range of system-level properties affecting many features and extremely difficult to "add on." We'll discuss this more in the next section.

Development

Agile advocates simple design

The primary difference between agile and plan-driven development practices deal with the design and architecture of the software. Agile methods advocate *simple design,* one that emerges as functionality is implemented. Simple design can be characterized by XP's goal to "always have the simplest design that runs the current test suite."[14] Agilists encourage the developer to make the design simpler at every opportunity. Taken to its logical conclusion, this means if your design has capabilities that are beyond the current user stories or that anticipate new features, you should expend extra effort to remove them. Of course, the idea is not to have them there in the first place.

*CeBASE is an NSF-sponsored collaborative research institute led by the University of Maryland's Fraunhofer Center for Experimental Software Engineering and the University of Southern California's Center for Software Engineering. Its mission is to strengthen and propagate the results of empirical research in software engineering (http://www.cebase.org/).

Simple design depends on low-cost rework and rapid change

The basis for advocating simple design rests on two fundamental assertions. The first is that the cost of rework to change the software ("refactoring" in agile language) to support new, possibly unanticipated, capabilities will remain low over time. The second fundamental assumption is that the application situation will change so rapidly that any code added to support future capabilities will never be used.

Low-cost rework is not guaranteed with agile methods

See Appendix E1

The assertion concerning low-cost rework is based on the hypothesis that constant refactoring yields constant improvement, so there will be a constant, low rate of change rather than a few large, expensive redesigns. Unfortunately, this seems to be a fragile proposition. Experiences where the cost to change has remained low over time tend to be anecdotal, associated with smaller applications, and usually involve expert programmers who are able to quickly refactor the design and correct defects. But the only sources of empirical data we have encountered have come from less-expert early adopters who found that even for small applications, the percentage of effort spent on refactoring and correcting defects increases with the number of requirement stories.[15, 16]

Low-cost rework doesn't scale

Experience to date also indicates that low-cost refactoring cannot be depended upon as projects scale up. The most serious problems that arise with simple design are problems known as "architecture breakers." These highly expensive problems can occur when early, simple design decisions result in foreseeable changes that cause breakage in the design beyond the ability of refactoring to handle. Here are some examples of architecture breakers.

- An early commitment to a fourth-generation language and its infrastructure that works beautifully when the application and user base are small, but is impossible to scale up as the application and user base become large.

- Escalating reliability, throughput, response time, or other quality attributes during development, or the addition of functionality within a tightly specified, real-time control loop that cannot be accommodated within the design. Frequently, the best performing architecture has only a limited range with respect to the level of quality specified.[17, 18]
- Deferred implementation of special conditions or functions, such as multinational and multilingual operations, fault tolerance through processor failover, or the need to handle extra-long messages, that significantly impact many parts of the software.

Simple design implies YAGNI—You Aren't Going to Need It

The second assumption concerning rapid change addresses programming efficiency. The thought here is that adding hooks for future functionality unnecessarily complicates the design and increases the effort to develop subsequent increments. Within the XP and other agile communities, this concept is known as You Aren't Going to Need It (YAGNI). YAGNI works fine when future requirements are largely unpredictable, but can be highly inefficient where there is a reasonable understanding of future needs. In situations where future requirements are predictable, YAGNI both throws away valuable architectural support for foreseeable requirements and frustrates customers who want developers to believe their priorities and evolution requirements are worth accommodating.

Plan-driven methods advocate architecture to anticipate changes

Plan-driven methods use *planning* and *architecture-based design* to accommodate foreseeable change. This effort allows the designers to organize the system to take advantage of software reuse across product lines and can have a major impact on rapid development.

HP had excellent results with product line architecture

In one division, Hewlett-Packard was able to reduce its software development cycle time from 48 months to 12 months over 5 years, by developing plug-and-play reusable software modules.[19] Significantly, Hewlett-Packard also found that its reuse economic model needed to add costs for adapting product line assets to unforeseeable change.[20] These included cost factors for architectural update, component obsolescence, and adaptive maintenance of components to stay consistent with changing external interfaces. Even with these added costs, software product lines have been highly successful for HP and many other organizations.[21, 22, 23]

Architecture can waste resources in rapidly changing environments

See Appendix E2

Thus, for the levels of predictability and dependability that are primary objectives of plan-driven methods, a significant amount of effort goes into analyzing and defining a robust architecture that will accommodate the system's envisioned life cycle usage envelope. For small and rapidly changing applications, this level of architecture investment, sometimes referred to as Big Design Up Front (BDUF), will be overkill. Several agile methods use some level of architecting: Crystal, DSDM, FDD, and Lean Development, for example. Scrum's consideration of its requirement backlog helps avoid misinformed simple design.

Other agile practices can support plan-driven approaches

While simple design and architectural design are definitely conflicting approaches, there are some agile development practices that can readily and productively be adopted for plan-driven projects. Examples are evolutionary and incremental development, continuous integration, and pair programming.

Testing

Agile methods develop tests before code, and test incrementally

Testing is one way of validating that the customers have specified the right product and verifying that the developers have built the product right. It requires that the code be developed and executed, which means that for long developments, problems will not be discovered until late in the development cycle, when they are expensive to fix. Agile methods address this problem by organizing the development into short increments, and by applying pair programming or other review techniques to remove more code defects as they are being generated. They also develop executable tests to serve in place of requirements and to enable earlier and continuous regression testing. Automated testing support is recommended by most agile methods. This approach has a number of significant advantages.

See Appendix E1

- It ensures that the requirements are testable.
- It avoids a great deal of documentation for requirements, requirement/test matrices, and test case definitions.
- It enables incremental build-and-test, with earlier identification of defects and misinterpreted stories.
- It helps modularize the applications structure and provides a safety net for refactoring.
- It helps form an explicit working knowledge of the application.[24]

However, there are some risks inherent to the test-first approach.[25]

- Rapid change causes expensive breakage in the tests.
- Rapid change causes mismatches and race problems between the code and the tests.
- Lack of applications or testing expertise may produce inadequate test coverage.

Plan-driven methods test to specifications

Plan-driven methods address the expensive late-fix problem by developing and consistency-checking requirements and architecture specifications early in the development process. They also invest in automated test suites to support the considerable planning and preparation before running tests. This creates a good deal of documentation that may undergo breakage due to changing requirements, but the documentation rework effort will usually be less than the test rework effort, particularly with automated test suites. On the other hand, late testing misses much of the agile early-testing advantages cited above. Plan-driven methods also frequently manifest an independent (and often adversarial) testing bureaucracy that can be entirely divorced from developer and customer, and so may spend a good proportion of scarce project resources determining if the product matches the letter of the specifications rather than operational intent and customer need.

Personnel Characteristics

While there has been significant discussion of the technical differences between agile and plan-driven approaches, we believe that some of the most fundamental differences lie in the people issues. The customers, developers, and organizational cultures have a significant influence on the success of most projects. In this section we discuss how the approaches are reliant on specific characteristics in each of these areas.

Customers

There is significant risk in unsuitable customer representatives

In the “Customer Relations” section above, we concluded that the major difference between agile and plan-driven methods was that agile methods strongly emphasize having dedicated and collocated customer representatives, while plan-driven methods count on a good deal of up-front, customer-developer work on contractual plans and specifications. For agile methods, the greatest risk is that insistence on a dedicated, collocated customer representative will cause the customer organization to supply the person that is most expendable. This risk establishes the need for criteria to determine the adequacy of customer representatives.

In our critical success factor analysis of over 100 e-services projects at USC, we have found that success depends on having customer representatives who are Collaborative, Representative, Authorized, Committed, and Knowledgeable (CRACK) performers. If the customer representatives are not collaborative, they will sow discord and frustration, resulting in the loss of team morale. If they are not representative, they will lead the developers to deliver unacceptable products. If they are not authorized, they will incur delays seeking authorization or, even worse, lead the project astray by making unauthorized commitments. If they are not committed, they won't do the necessary homework and won't be there when the developers need them most. Finally, if they are not knowledgeable, they will cause delays, unacceptable products, or both.

USC found customer representatives need to be Collaborative, Representative, Authorized, Committed, and Knowledgeable (CRACK)

This summary of customer impact on the landmark C3 project, considered to be the first XP project, is a good example of the need for CRACK customer representatives.

Chrysler provides an example

> The on-site customer in this project had a vision of the perfect system she wanted to develop. She was able to provide user stories that were easy to estimate. Moreover, she was with the development team every day, answering any business questions the developer had.
>
> Half-way [through] the project, several things changed, which eventually led to the project being cancelled. One of the changes was the replacement of the on-site customer, showing that the actual way in which the customer is involved is one of the key success factors in an XP project. The new on-site customer was present most of the time, just like the previous on-site customer, and available to the development team for questions. Unfortunately, the requirements and user stories were not as crisp as they were before.[26]

Plan-driven methods also need CRACK customer representatives and benefit from full-time, on-site participation. Good planning artifacts, however, enable them to settle for part-time CRACK representatives who provide further benefits by keeping active in customer operations. The greatest customer challenge for plan-driven methods is to keep project control from falling into the hands of overly bureaucratic

Plan-driven methods also need CRACK customers, but not full-time contract managers who prioritize contract compliance above getting project results.

FAA example of bureaucratic plan-driven approach

A classic example of customer bureaucracy is provided in Robert Britcher's book, *The Limits of Software,*[27] describing his experience on perhaps the world's biggest failed software project: the FAA/IBM Advanced Automation System for U.S. national air traffic control. Due to many bureaucratic and other problems, including responding to change over following a plan, the project was overrunning by years and billions of dollars. One of the software development groups came up with a way of reducing the project's commitment to a heavyweight brand of software inspections that were slowing the project down by consuming too much staff effort in paperwork and redundant tasks. The group came up with a lightweight version of the inspection process. It was comparably successful in finding defects, but with much less time and effort. Was the group rewarded for doing this? No, the contracting bureaucracy sent them a cease-and-desist letter faulting them for contract noncompliance and ordering them to go back to the heavyweight inspections. This is the kind of plan-driven bureaucracy that agilists justifiably deride.

Developers

Agile developers need more than technical skills

Critical people-factors for agile methods include amicability, talent, skill, and communication.[28] An independent assessment identifies this as 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 . . ."[29] Figure 2-1 distinguishes the most effective operating points of agile and plan-driven projects.[30, 31] Both operate best with a mix of developer skills and understanding, but agile methods tend to need a richer mix of higher-skilled people.

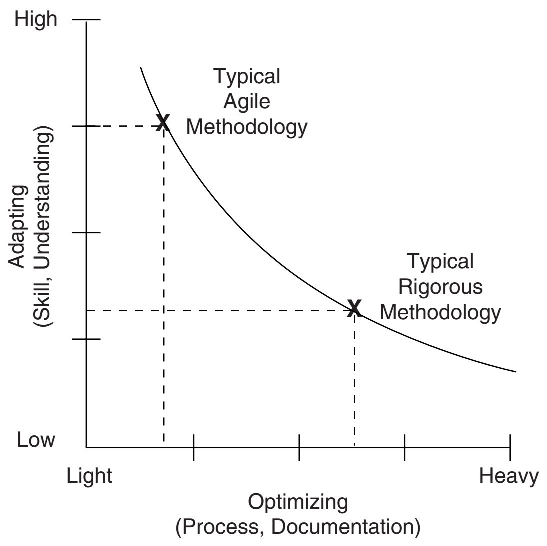

Figure 2-1 Balancing Optimizing and Adapting Dimensions (from Cockburn and Highsmith)

When you have such people available on your project, statements like "A few designers sitting together can produce a better design than each could produce alone" are valid. If not, you're more likely to get design-by-committee, with the opposite effect. The plan-driven methods of course do better with great people, but are generally more able to plan the project and architect the software so that less-capable people can contribute with low risk. A significant consideration here is the unavoidable statistic that 49.999 percent of the world's software developers are below average (slightly more precisely, below median).

Plan-driven methods need fewer highly talented people than agile

It is important to be able to classify the type of personnel required for success in the various methods. Alistair Cockburn has addressed levels of skill and understanding required for performing various method-related functions, such as using, tailoring, adapting, or revising a method.

We modify Cockburn's levels to meet our needs

Table 2-1 Levels of Software Method Understanding and Use (after Cockburn)

Level	Characteristics
3	Able to revise a method (break its rules) to fit an unprecedented new situation
2	Able to tailor a method to fit a precedented new situation
1A	With training, able to perform discretionary method steps (e.g., sizing stories to fit increments, composing patterns, compound refactoring, complex COTS integration). With experience, can become Level 2.
1B	With training, able to perform procedural method steps (e.g., coding a simple method, simple refactoring, following coding standards and CM procedures, running tests). With experience, can master some Level 1A skills.
–1	May have technical skills, but unable or unwilling to collaborate or follow shared methods.

Drawing on the three levels of understanding in Aikido (Shu-Ha-Ri), he has identified three levels of software method understanding that help sort out what various levels of people can be expected to do within a given method framework.[32] Modifying his work to meet our needs, we have split his Level 1 to address some distinctions between agile and plan-driven methods, and added an additional level to address the problem of method-disrupters. Our version is provided in Table 2-1.

Reassign Level -1s

Level –1 people should be rapidly identified and reassigned to work other than performing on either agile or plan-driven teams.

Level 1B people are average-and-below, less-experienced, hard-working developers. They can function well in performing straightforward software development in a stable situation. But they are likely to slow down an agile team trying to cope with rapid change, particularly if they form a majority of the team. They can form a well-performing majority of a stable, well-structured plan-driven team.

Level 1Bs need considerable guidance, work well in plan-driven environment

Level 1A people can function well on agile or plan-driven teams if there are enough Level 2 people to guide them. When agilists refer to being able to succeed on agile teams with ratios of five Level 1 people per Level 2 person, they are generally referring to Level 1A people.

Level 1As need guidance but can work well on agile teams

Level 2 people can function well in managing a small, precedented agile or plan-driven project but need the guidance of Level 3 people on a large or unprecedented project. Some Level 2s have the capability to become Level 3s with experience. Some do not.

Level 2s can manage precedented projects but need Level 3 guidance on unprecedented projects

Culture

In an agile culture, the people feel comfortable and empowered when there are *many degrees of freedom* available for them to define and work problems. This is the classic craftsman environment, where each person is expected and trusted to do whatever work is necessary to the success of the project. This includes looking for common or unnoticed tasks and completing them.

Agilists like many degrees of freedom

In a plan-driven culture, the people feel comfortable and empowered when there are *clear policies and procedures* that define their role in the enterprise. This is more of a production-line environment where each person's tasks are well-defined. The expectation is that they will accomplish the tasks to specification so that their work products will easily

Plan-driven people need clear process and roles integrate into others' work products with limited knowledge of what others are actually doing.

Cultural inertia is a significant challenge

These cultures get reinforced as people tend to self-select for their preferred culture, and as people within the culture get promoted to higher levels of management. Once a culture is well established, it is difficult and time consuming to change. This cultural inertia may be the most significant challenge to the integration of agile and plan-driven approaches.

Agile culture change has a revolutionary flavor

To date, agile culture change has had a bottom-up, revolutionary flavor. Failing projects with no hope of success have been the usual pilots, supported by an "it can't hurt" attitude from management and a "no challenge is too hard" adrenalin-charged response from practitioners. Successes have been extraordinary in many cases and have been used to defend migration to less troubled projects.

CMM faced culture change issues early

Early CMM adopters faced similar challenges, although there was early involvement of middle management. The concept of culture change evolved rapidly and is now well understood by the managers and Software Engineering Process Groups (SEPGs). These have been the main change agents in evolving their organizations from following improvised, ad hoc processes toward following plan-driven, CMM-compliant processes.

CMMI improves CMM, but is a culture change in itself

The new CMMI upgrades the SW-CMM in more agile directions, with new process areas for integrated teaming, risk management, and overall integrated systems and software engineering. A number of organizations are welcoming this opportunity to add more agility to their organizational culture. But others that retain a more bureaucratic interpretation of the SW-CMM are facing the challenge of "change-averse change agents" who have become quite comfortable in their bureaucratic culture.

Summary

This chapter has provided a lot of information on a number of characteristics. In the following section we summarize the material and provide a graphic way to use the characteristics to describe the agility/plan-driven profile of a project or organization in terms of five factors.

Home Grounds

Home grounds can also characterize projects

Table 2-2 summarizes the comparisons in the four areas of Chapter 2 by showing the "home grounds" for agile and plan-driven methods—the sets of conditions under which they are most likely to succeed. The more a particular project's conditions differ from the home ground conditions,

Table 2-2 Agile and Plan-Driven Method Home Grounds

Characteristics	Agile	Plan-Driven
Application		
Primary Goals	Rapid value; responding to change	Predictability, stability, high assurance
Size	Smaller teams and projects	Larger teams and projects
Environment	Turbulent; high change; project-focused	Stable; low-change; project/organization focused
Management		
Customer Relations	Dedicated on-site customers; focused on prioritized increments	As-needed customer interactions; focused on contract provisions
Planning and Control	Internalized plans; qualitative control	Documented plans, quantitative control
Communication	Tacit interpersonal knowledge	Explicit documented knowledge

(*continued*)

Table 2-2 Continued

Characteristics	Agile	Plan-Driven
Technical		
Requirements	Prioritized informal stories and test cases; undergoing unforeseeable change	Formalized project, capability, interface, quality, foreseeable evolution requirements
Development	Simple design; short increments; refactoring assumed inexpensive	Extensive design; longer increments; refactoring assumed expensive
Testing	Executable test cases define requirements	Documented test plans and procedures
Personnel		
Customers	Dedicated, collocated CRACK* performers	CRACK* performers, not always collocated
Developers	At least 30% full-time Cockburn Level 2 and 3 experts; no Level 1B or -1 personnel**	50% Cockburn Level 3s early; 10% throughout; 30% Level 1Bs workable; no Level -1s**
Culture	Comfort and empowerment via many degrees of freedom (thriving on chaos)	Comfort and empowerment via framework of policies and procedures (thriving on order)

* Collaborative, Representative, Authorized, Committed, Knowledgeable
** These numbers will particularly vary with the complexity of the application

the more risk there is in using one approach in its pure form and the more valuable it is to blend in some of the complementary practices from the opposite method. In Chapter 4 we will provide case studies showing how an agile and a plan-driven project were able to succeed outside their home grounds. Chapter 5 illustrates a risk-driven approach for tailoring balanced agile/plan-driven strategies for non–home ground projects.

Misconceptions

People misrepresent both approaches

Table 2-3 is an attempt to counter several misconceptions about agile and plan-driven methods. Many of these are caused by people misrepresenting their use of agile and plan-driven methods. A good example was presented in Chapter 1 where a team claimed to follow XP, but on further investigation had simply stopped documenting their software.[33]

Misconceptions hurt everyone

Such misconceptions propel agile and plan-driven advocates into a lot of unnecessary, polarized arguments and add to the perplexity that we are trying to help our readers sort out.

Table 2-3 Misconceptions and Realities about Agile and Plan-Driven Methods

Misconceptions	Realities
Plan-Driven Methods	
Plan-driven methods are uniformly bureaucratic	Overly bureaucratic cultures and methods can stultify software development
Having documented plans guarantees compliance with plans	Not necessarily
Plan-driven methods can succeed with a lack of talented people	Plan-driven methods can succeed with a smaller percentage of talented people
High maturity guarantees success	Explicit, documented plans provide more of a safety net than tacit plans
There are no penalties in applying plan-driven methods when change is unforeseeable	Plan-driven methods work best in accommodating foreseeable change

(*continued*)

Table 2-3 Continued

Misconceptions	Realities
Agile Methods	
Agile methods don't plan	Agile methods get much of their speed and agility through creating and exploiting tacit knowledge
Agile methods require uniformly talented people	Agile methods work best when there is a critical mass of highly talented people involved
Agile methods can make the slope of the cost-to-change vs. time curve uniformly flat	Agile methods can reduce the slope of the cost-to-change vs. time curve
YAGNI is a universally safe assumption, and won't alienate your customers	YAGNI helps handle unforeseeable change, but is risky when change is foreseeable

Five Critical Factors

Decision factors are size, criticality, dynamism, personnel, and culture

As a "summary of summaries," we have concluded that there are five critical factors involved in determining the relative suitability of agile or plan-driven methods in a particular project situation. These factors, described in Table 2-4, are the project's size, criticality, dynamism, personnel, and culture factors. As we shall see in Chapters 4 and 5, a project which is a good fit to agile or plan-driven for four of the factors, but not the fifth, is a project in need of risk assessment and likely some mix of agile and plan-driven methods.

Size, criticality, and culture map easily to home grounds

The five critical factors associated with the agile and plan-driven home grounds from Table 2-4 are summarized graphically in Figure 2-2. Of the five axes in the polar graph, *Size* and *Criticality* are similar to the

Table 2-4 The Five Critical Agility/Plan-Driven Factors

Factor	Agility Discriminators	Plan-Driven Discriminators
Size	Well-matched to small products and teams. Reliance on tacit knowledge limits scalability.	Methods evolved to handle large products and teams. Hard to tailor down to small projects.
Criticality	Untested on safety-critical products. Potential difficulties with simple design and lack of documentation.	Methods evolved to handle highly critical products. Hard to tailor down to low-criticality products.
Dynamism	Simple design and continuous refactoring are excellent for highly dynamic environments, but a source of potentially expensive rework for highly stable environments.	Detailed plans and Big Design Up Front excellent for highly stable environment, but a source of expensive rework for highly dynamic environments.
Personnel	Requires continuous presence of a critical mass of scarce Cockburn Level 2 or 3 experts. Risky to use non-agile Level 1B people.	Needs a critical mass of scarce Cockburn Level 2 and 3 experts during project definition, but can work with fewer later in the project —unless the environment is highly dynamic. Can usually accommodate some Level 1B people.
Culture	Thrives in a culture where people feel comfortable and empowered by having many degrees of freedom. (Thriving on chaos)	Thrives in a culture where people feel comfortable and empowered by having their roles defined by clear policies and procedures. (Thriving on order)

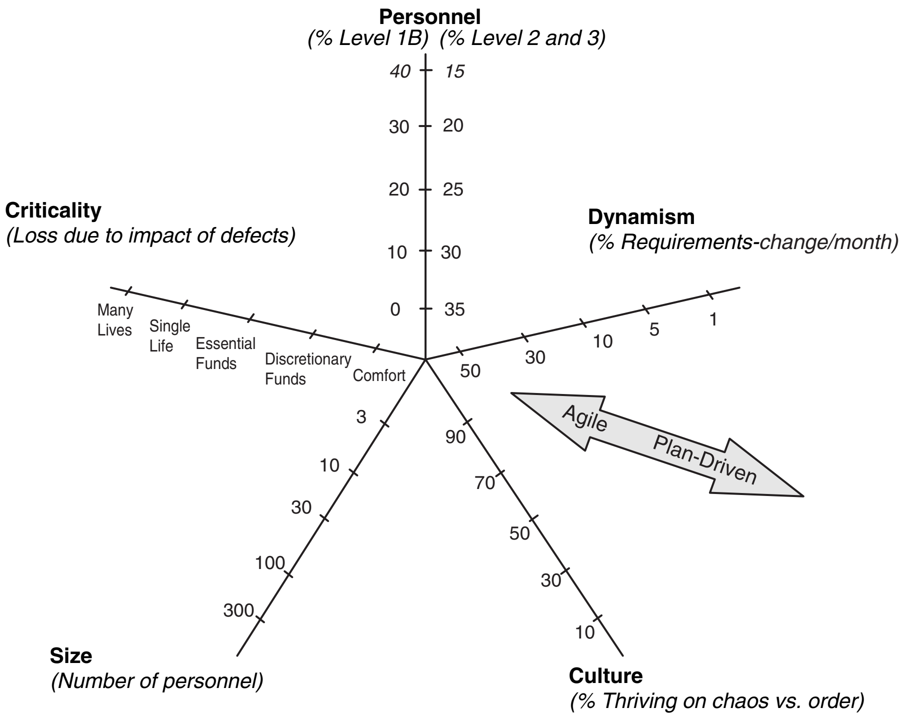

Figure 2-2 Dimensions Affecting Method Selection

factors used by Alistair Cockburn to distinguish between the lighter-weight Crystal methods (toward the center of the graph) and heavier-weight Crystal methods (toward the periphery). The *Culture* axis reflects the reality that agile methods will succeed better in a culture that “thrives on chaos”[34] than one that “thrives on order,” and vice versa.

Dynamism reflects the rate of change, primarily a plan-driven issue

The other two axes are asymmetrical in that both agile and plan-driven methods are likely to succeed at one end, and only one of them is likely to succeed at the other. For *Dynamism,* agile methods are at home with both high and low rates of change, but plan-driven methods prefer low rates of change.

The *Personnel* scale refers to the extended Cockburn method skill rating scale discussed earlier in the chapter. Here the asymmetry is that while plan-driven methods can work well with both high and low skill levels, agile methods require a richer mix of higher-level skills (see Figure 2-1).

Personnel addresses mix of Level 2 and 3 and Level 1B developers

For example, a plan-driven project with 15 percent Level 2 and 3 people and 40 percent Level 1B people would initially use more than 15 percent Level 2 and 3 people to plan the project, but reduce the number thereafter. An agile project would have everybody working full-time, and the 15 percent Level 2s and 3s would be swamped trying to mentor the 40 percent Level 1Bs and the remaining Level 1As while trying to get their own work done as well.

A typical example of personnel mix

By rating a project along each of the five axes, you can visibly evaluate its home ground relationships. If all the ratings are near the center, you are in agile method territory. If they are at the periphery, you will best succeed with a plan-driven approach. If you are mostly in one or the other, you need to treat the exceptions as sources of risk and devise risk management approaches to address them.

Rating shows home ground relationship graphically