1

A Value-Up Paradigm

"A theory should be as simple as possible, but no simpler."
—Albert Einstein

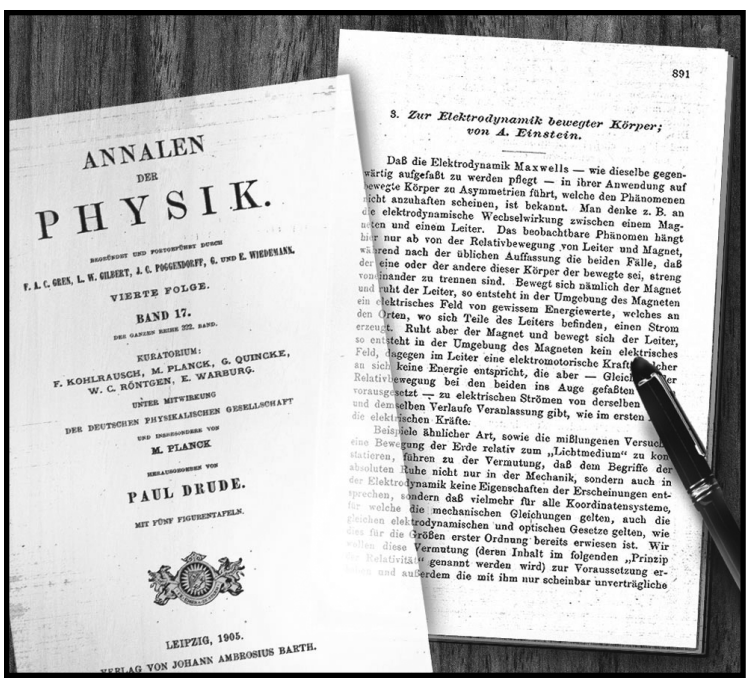


Figure 1.1 **Einstein's Theory of Special Relativity was the focal point of a paradigm shift in our understanding of physics. It capped forty years of debate on the most vexing technical challenges of his day—how to synchronize clocks and how to accurately draw maps over long distances.**

A Paradigm Shift

Paradigm shifts come in fits and starts, as old theories can no longer explain the world as observed.[1] A poster child for the scientific paradigm shift is Albert Einstein's Theory of Special Relativity, published in 1905. Einstein's work reduced Newtonian mechanics to a special case, settled forty years of debate on the nature of time and synchronicity, and set the agenda for much of science, technology, and world affairs of the twentieth century.

According to a posthumous legend many of us learned in school, Einstein was a solitary theoretician whose day job reviewing patent applications was a mere distraction from his passionate pursuit of physics. Yet this popular view of Einstein is misguided. In fact, the majority of patent applications that Einstein reviewed concerned the very physics problem that fascinated him—how to synchronize time over distance for multiple practical purposes, such as creating railroad schedules, maritime charts, and accurate territorial maps in an age of colonial expansion. Indeed, the synchronization of time was a great technological problem of the age, for which special relativity became a mathematical solution, capping decades of debate.

Einstein was not the only person to solve the mathematical problem in 1905—the far more prominent Henri Poincaré produced an alternative that has long since been forgotten.[2] Why is Einstein's solution the one taught in every physics class today? Poincaré's calculations relied on the "ether," a supposed medium of space that had pervaded nineteenth-century physics. Einstein's Special Relativity, on the other hand, used much simpler calculations that required no ether. This was the first notable example of the principle attributed to Einstein, also posthumously, that "a theory should be as simple as possible, but no simpler."

Three Forces to Reconcile

A shift similar to the contrasting views of physics 100 years ago has been occurring today in software development. On a weekend in 2001, seventeen software luminaries convened to discuss "lightweight methods." At the end of the weekend, they launched the Agile Alliance, initially charged around the *Agile Manifesto*.[3] Initially, it was a rallying cry for those who saw contemporary software processes as similar

to the "ether" of nineteenth-century physics—an unnecessary complexity and an impediment to productivity. Five years later, "agility" is mainstream. Every industry analyst advocates it, every business executive espouses it, and everyone tries to get more of it.

At the same time, two external economic factors came into play. One is global competition. The convergence of economic liberalization, increased communications bandwidth, and a highly skilled labor force in emerging markets made the outsourcing of software development to lower-wage countries (especially India) profitable.[4] The Indian consultancies, in turn, needed to guarantee their quality to American and European customers. Many latched onto Capability Maturity Model Integration (CMMI) from the Software Engineering Institute at Carnegie Mellon University.[5] CMMI epitomized the heavyweight processes against which the agilists rebelled, and it was considered too expensive to be practical outside of the defense industry. The offshorers, with their cost advantage, did not mind the expense and could turn the credential of a CMMI appraisal into a competitive advantage.

The second economic factor is increased attention to regulatory compliance after the lax business practices of the 1990s. In the United States, the Sarbanes-Oxley Act of 2002 (SOX) epitomizes this emphasis by holding business executives criminally liable for financial misrepresentations. This means that software and systems that process financial information are subject to a level of scrutiny and audit much greater than previously known.

These forces—agility, outsourcing/offshoring, and compliance—cannot be resolved without a paradigm shift in the way we approach the software lifecycle. The modern economics require agility with accountability. Closing the gap requires a new approach, both to process itself and to its tooling.

What Software Is Worth Building?

To overcome the gap, you must recognize that software engineering is not like other engineering. When you build a bridge, road, or house, for example, you can safely study hundreds of very similar examples. Indeed, most of the time, economics dictate that you build the current one almost exactly like the last to take the risk out of the project.

With software, if someone has built a system just like you need, or close to what you need, then chances are you can license it commercially (or even find it as freeware). No sane business is going to spend money on building software that it can buy more economically. With thousands of software products available for commercial license, it is almost always cheaper to buy. Because the decision to build software must be based on sound return on investment and risk analysis, the software projects that get built will almost invariably be those that are *not* available commercially.

This business context has a profound effect on the nature of software projects. It means that software projects that are easy and low risk, because they've been done before, don't get funded. The only new software development projects undertaken are those that haven't been done before or those whose predecessors are not publicly available. This business reality, more than any other factor, is what makes software development so hard and risky, which makes attention to process so important.[6]

Contrasting Paradigms

The inherent uncertainty in software projects makes it difficult to estimate tasks correctly, which creates a high variance in the accuracy of the estimates. A common misconception is that the variance is acceptable because the positive and negative variations average out. However, because software projects are long chains of dependent events, the variation itself accumulates in the form of downstream delays.[7]

Unfortunately, most accepted project management wisdom comes from the world of roads and bridges. In that world, design risks are low, design cost is small relative to build cost, and the opportunity to deliver incremental value is rare. (You can't drive across a half-finished bridge!) With this style of project management, you determine an engineering design early, carefully decompose the design into implementation tasks, schedule and resource the tasks according to their dependencies and resource availability, and monitor the project by checking off tasks as completed (or tracking percentages completed). For simplicity, I'll call this style of project management the *work-down* approach because it is easily envisioned as burning down a list of tasks.

The work-down approach succeeds for engineering projects with low risk, low variance, and well-understood design. Many IT projects, for example, are customizations of commercial-off-the-shelf software (COTS), such as enterprise resource planning systems. Often, the development is a small part of the project relative to the business analysis, project management, and testing. Typically, these projects have lower variability than new development projects, so the wisdom of roads and bridges works better for them than for new development.

Since 1992,[8] there has been a growing challenge to the work-down wisdom about software process. No single term has captured the emerging paradigm, but for simplicity, I'll call this the *value-up* approach. And as happens with new paradigms, the value-up view has appeared in fits and starts (see Figure 1.2).

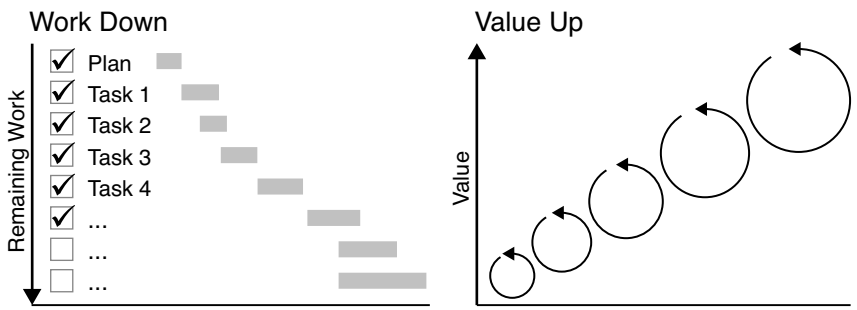


Figure 1.2 **The attitudinal difference between work-down and value-up is in the primary measurement. Work-down treats the project as a fixed stock of tasks at some cost that need completion and measures the expenditure against those tasks. Value-up measures value delivered at each point in time and treats the inputs as variable flows rather than a fixed stock.**

An example of the value-up school is the agile project management manifesto Declaration of Interdependence.[9] It states six principles that characterize value-up:

- We increase return on investment by making continuous flow of value our focus.
- We deliver reliable results by engaging customers in frequent interactions and shared ownership.
- We expect uncertainty and manage for it through iterations, anticipation, and adaptation.

- We unleash creativity and innovation by recognizing that individuals are the ultimate source of value, and creating an environment where they can make a difference.
- We boost performance through group accountability for results and shared responsibility for team effectiveness.
- We improve effectiveness and reliability through situationally specific strategies, processes, and practices.

Behind these principles is a significantly different point of view about practices between the work-down and value-up mindsets. Table 1.1 below summarizes the differences.

Table 1.1 Attitudinal Differences Between Work-Down and Value-Up Paradigms

Core Assumption	Work-Down Attitude	Value-Up Attitude
Planning and change process	Planning and design are the most important activities to get right. You need to do these initially, establish accountability to plan, monitor against the plan, and carefully prevent change from creeping in.	Change happens; embrace it. Planning and design will continue through the project. Therefore, you should invest in just enough planning and design to understand risk and to manage the next small increment.
Primary measurement	Task completion. Because we know the steps to achieve the end goal, we can measure every intermediate deliverable and compute earned value running as the percentage of hours planned to be spent by now versus the hours planned to be spent to completion.	Only deliverables that the customer values (working software, completed documentation, etc.) count. You need to measure the flow of the work streams by managing queues that deliver customer value and treat all interim measures skeptically.

Core Assumption	Work-Down Attitude	Value-Up Attitude
Definition of quality	Conformance to specification. That's why you need to get the specs right at the beginning.	Value to the customer. This perception can (and probably will) change. The customer might not be able to articulate how to deliver the value until working software is initially delivered. Therefore, keep options open, optimize for continual delivery, and don't specify too much too soon.
Acceptance of variance	Tasks can be identified and estimated in a deterministic way. You don't need to pay attention to variance.	Variance is part of all process flows, natural and man-made. To achieve predictability, you need to understand and reduce the variance.
Intermediate work products	Documents, models, and other intermediate artifacts are necessary to decompose the design and plan tasks, and they provide the necessary way to measure intermediate progress.	Intermediate documentation should minimize the uncertainty and variation in order to improve flow. Beyond that, they are unnecessary.
Troubleshooting approach	The constraints of time, resource, functionality, and quality determine what you can achieve. If you adjust one, you need to adjust the others. Control change carefully to make sure that there are no unmanaged changes to the plan.	The constraints may or may not be related to time, resource, functionality, or quality. Instead, identify the primary bottleneck in the flow of value, work it until it is no longer the primary one, and then attack the next one. Keep reducing variance to ensure smoother flow.
Approach to trust	People need to be monitored and compared to standards. Management should use incentives to reward individuals for their performance relative to the plan.	Pride of workmanship and teamwork are more effective motivators than individual incentives. Trustworthy transparency, where all team members can see the overall team's performance data, works better than management directives.

Attention to Flow

Central to the value-up paradigm is an emphasis on *flow*. There are two discrete meanings of flow, and both are significant in planning software projects.

First, flow is the human experience of performing expertly, as Mihaly Csikszentmihalyi explains in *Flow: The Psychology of Optimal Experience*:

> We have seen how people describe the common characteristics of optimal experience: a sense that one's skills are adequate to cope with the challenges at hand, in a goal-directed, rule-bound action system that provides clear clues as to how well one is performing. Concentration is so intense that there is no attention left over to think about anything irrelevant, or to worry about problems. Self-consciousness disappears, and the sense of time becomes distorted. An activity that produces such experiences is so gratifying that people are willing to do it for its own sake, with little concern for what they will get out of it, even when it is difficult, or dangerous.[10]

This meaning of flow is cited heavily by advocates of eXtreme Programming (XP) and other practices that focus on individual performance.

The second meaning of flow is the flow of customer value as the primary measure of the system of delivery. David J. Anderson summarizes this view in *Agile Management for Software Engineering*:

> Flow means that there is a steady movement of value through the system. Client-valued functionality is moving regularly through the stages of transformation—and the steady arrival of throughput—with working code being delivered.[11]

In this paradigm, you do not measure planned tasks completed as the primary indicator of progress; you count units of value delivered. Your rates of progress in throughput of delivered value and stage of completion at the units of value are the indicators that you use for planning and measurement.

Correspondingly, the flow-of-value approach forces you to understand the constraints that restrict the flow. You tune the end-to-end flow by identifying the most severe bottleneck or inefficiency your process, fixing it, and then tackling the next most severe one. As Anderson explains:

> The development manager must ensure the flow of value through the transformation processes in the system. He is responsible for the rate of production output from the system and the time it takes to process a single idea through the system. To understand how to improve the rate of production and reduce the lead time, the development manager needs to understand how the system works, be able to identify the constraints, and make appropriate decisions to protect, exploit, subordinate, and elevate the system processes.[12]

A flow-based approach to planning and project management requires keeping intermediate work-in-process to a minimum, as shown in Figure 1.3. This mitigates the risk of late discovery of problems and unexpected bubbles of required rework.

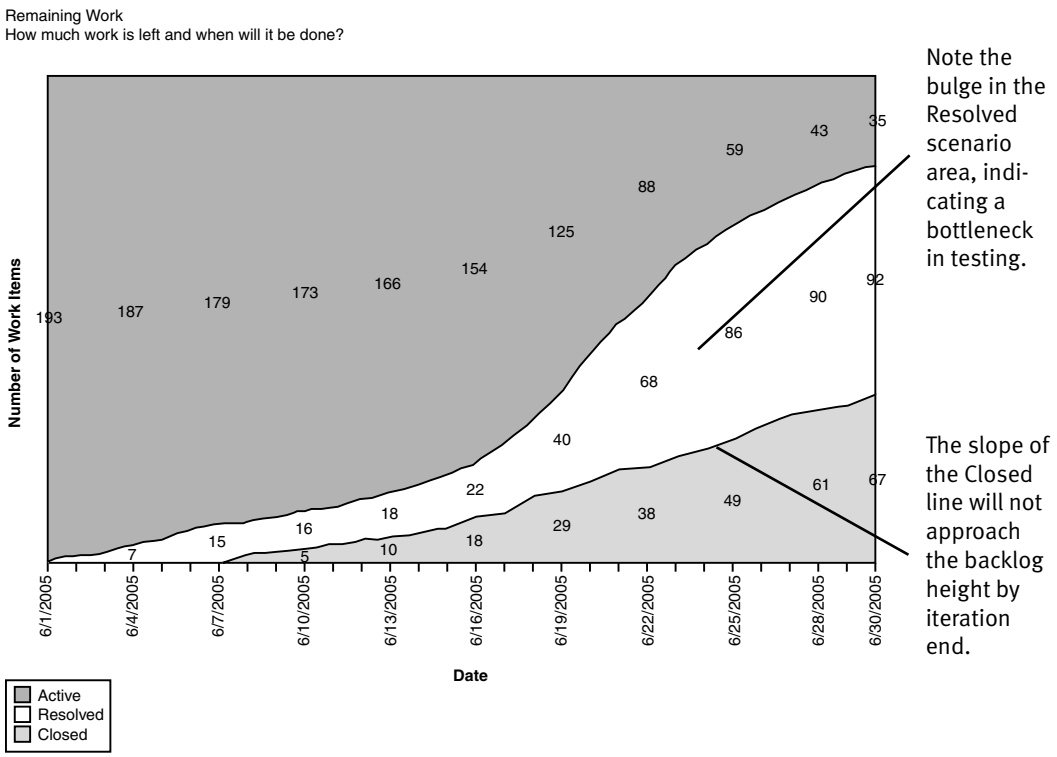


Figure 1.3 Measuring flow of scenario completion on a daily basis shows the rhythm of progress and quickly identifies bottlenecks that can be addressed as they arise.

Figure 1.3 shows how the continuous measurement of flow can illuminate bottlenecks as they are forming. Planned work for the iteration is progressing well through development (Active turning to Resolved), but is increasingly getting stuck in testing (Resolved to Closed). This accumulates as the bulge of work-in-process in the middle band. If you tracked development only (the reduction in Active work items), you would expect completion of the work by the expected end date; but because of the bottleneck, you can see that the slope of the Closed triangle is not steep enough to finish the work on time. This lets you drill into the bottleneck and determine whether the problem is inadequate testing resources or poor quality of work from development.

Contrast to Work-Down

An icon of the work-down paradigm is the widely taught "iron triangle" view of project management. This is the notion that there are only three variables that a project manager can work with: time, resources (of which people are by far the most important), and functionality. If you acknowledge quality as a fourth dimension (which most people do now), then you have a tetrahedron, as shown in Figure 1.4.

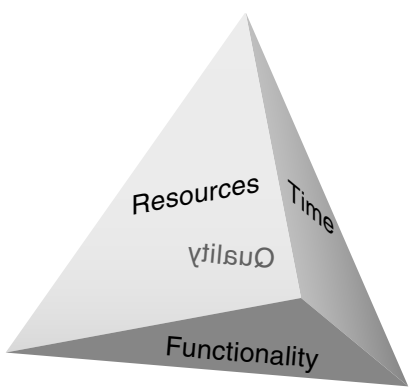


Figure 1.4 The "iron triangle" (or tetrahedron) treats a project as a fixed stock of work, in classic work-down terms. To stretch one face of the tetrahedron, you need to stretch the others.

In *Rapid Development*, Steve McConnell summarizes the iron triangle as follows:

> To keep the triangle balanced, you have to balance schedule, cost, and product. If you want to load up the product corner of the triangle, you also have to load up cost or schedule or both. The same goes for the other combinations. If you want to change one of the corners of the triangle, you have to change at least one of the others to keep it in balance.[13]

According to this view, a project manager has an initial stock of resources and time. Any change to functionality or quality requires a corresponding increase in time or resources. You cannot stretch one face without stretching the others because they are all connected.

Although widely practiced, this paradigm does not work well. Just as Newtonian physics is now known to be a special case, the iron triangle is a special case that assumes the process is flowing smoothly to begin with. In other words, it assumes that resource productivity is quite uniformly distributed, that there is little variance in the effectiveness of task completion, and that no spare capacity exists throughout the system. These conditions exist sometimes, notably on low-risk projects. Unfortunately, for the types of software projects usually undertaken, they are often untrue.

Many users of agile methods have demonstrated experiences that pleasantly contradict to this viewpoint. For example, in many cases, if you improve qualities of service, such as reliability, you can *shorten* time. Significant improvements in flow are possible within the existing resources and time.[14]

Transparency

It's no secret that most software projects are late, both in the execution and in the discovery that they are late.[15] This phenomenon has many consequences, which are discussed in almost every chapter of this book. One of the consequences is a vicious cycle of groupthink and denial that undermines effective flow. Late delivery leads to requests for replanning, which lead to pressure for ever more optimistic estimates, which lead to more late delivery, and so on. And most participants in these projects plan optimistically, replan, and replan further but with little visibility into the effects. Of course, the all-too-frequent result is a death march.

This is not because people can't plan or manage their time. The problem is more commonly the disparity among priorities and expectations of different team members. Most approaches to software engineering have lots of places to track the work—spreadsheets, Microsoft Project Plans, requirements databases, bug databases, test management systems, triage meeting notes, and so on. When the information is scattered this way, it is pretty hard to get a whole picture of the project—you need to look in too many sources, and it's hard to balance all the information into one schedule. And when there are so many sources, the information you find is often obsolete when you find it.

Things don't need to be that way. Some community projects post their development schedules on the Web, effectively making individual contributors create expectations among their community peers about their tasks. Making all the work in a project visible can create a virtuous cycle. Of course, this assumes that the project is structured iteratively, the scheduling and estimation are made at the right granularity, and triage is effective at keeping the work item priorities in line with the available resources in the iteration.

SCRUM, one of the agile processes, championed the idea of a transparently visible product backlog, as shown in Figure 1.5. Here's how the founders of SCRUM, Ken Schwaber and Mike Beedle, define the product backlog:

> Product Backlog is an evolving, prioritized queue of business and technical functionality that needs to be developed into a system. The Product Backlog represents everything that anyone interested in the product or process has thought is needed or would be a good idea in the product. It is a list of all features, functions, technologies, enhancements and bug fixes that constitute the changes that will be made to the product for future releases. Anything that represents work to be done on the product is included in Product Backlog.[16]

This transparency is enormously effective for multiple reasons. It creates a "single set of books," or in other words, a unique, maintained source of information on the work completed and remaining. Combined with flow measurement, as shown in Figure 1.3, it creates trust among the team because everyone sees the same data and plan. And finally, it creates a virtuous cycle between team responsibility and individual accountability. After all, an individual is most likely to complete a task when he or she knows exactly who is expecting it to be done.[17]

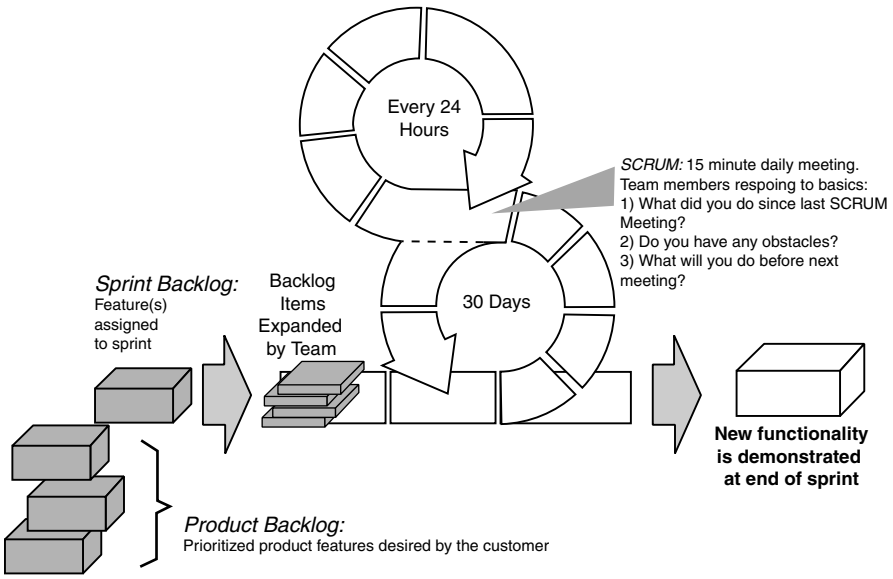


Figure 1.5 The central graphic of the SCRUM methodology is a great illustration of flow in the management sense. Not surprisingly, SCRUM pioneered the concept of a single product backlog as a management technique.

One Work Item Database

Visual Studio Team System (VSTS) takes the idea of a transparent product backlog even further (see Figure 1.6). Team System uses a common product backlog to track all planned, active, and completed work for the team and a history of the majority of actions taken and decisions made regarding that work. It calls these units "work items" and lets the user view and edit them in a database view inside Visual Studio, in Microsoft Excel, and in Microsoft Project, all the while synchronizing them to a common database.

One database behind the common, familiar tools defragments the information. Instead of cutting and pasting among randomly distributed artifacts, project managers, business analysts, developers, and testers all see the same work, whether planned in advance or scheduled on the fly, and whether from well-understood requirements or discovered while fixing a bug (see Figure 1.7). And unlike separate project tracking tools and techniques, much of the data collection in VSTS is automatic.

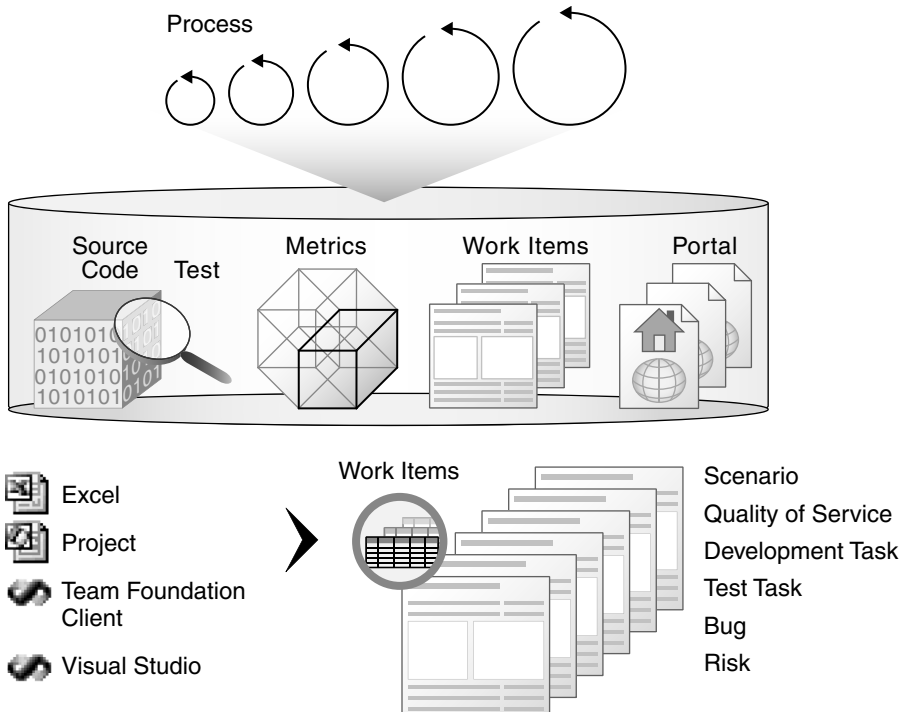


Figure 1.6 **VSTS enacts and instruments the process, tying source code, testing, work items, and metrics together. Work items include all the work that needs to be tracked on a project, such as scenarios, quality of service requirements, development tasks, test tasks, bugs, and risks. These can be viewed and edited in the Team Explorer, Visual Studio, Microsoft Excel, or Microsoft Project.**

Because VSTS uses a common database to track work items, it exposes them not just in Team Explorer but also in Microsoft Excel (see Figures 1.8 and 1.9). The use of Excel and Project is convenient but not necessary. All the functionality is available through the Team Explorer, which is the client for Team Foundation. If you're using any Visual Studio Team System client edition or Visual Studio Professional, then the Team Explorer appears as a set of windows inside the development environment.

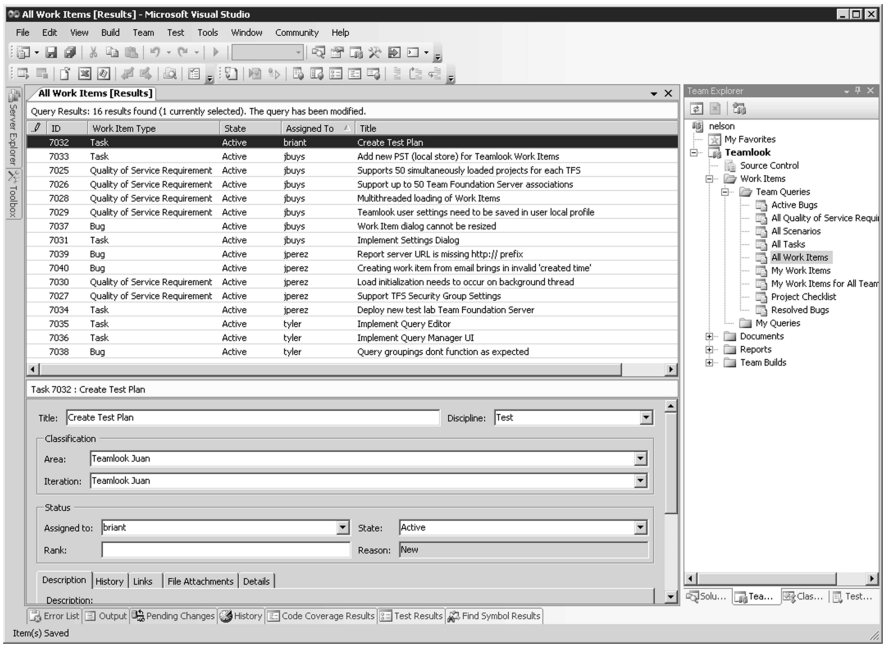


Figure 1.7 This is an example of the work items as they appear either in the Team Explorer of VSTS or in Visual Studio. Note that tasks, requirements, and bugs can all be viewed in one place.

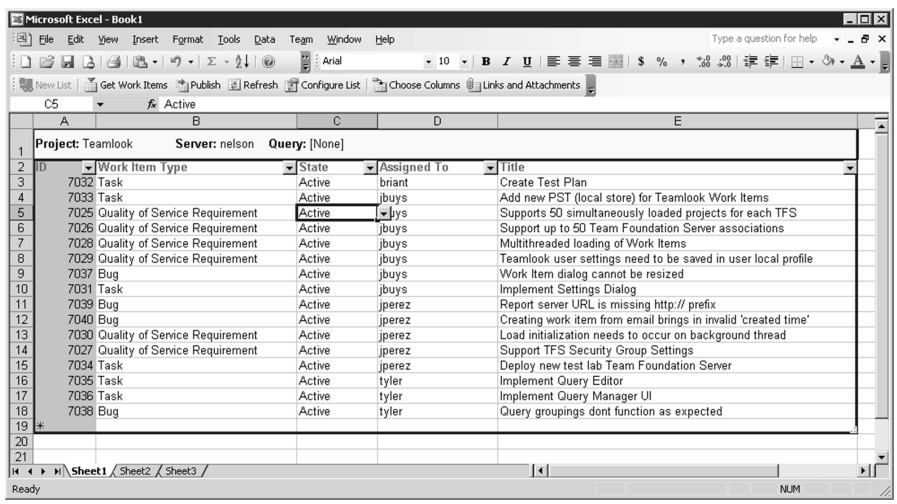
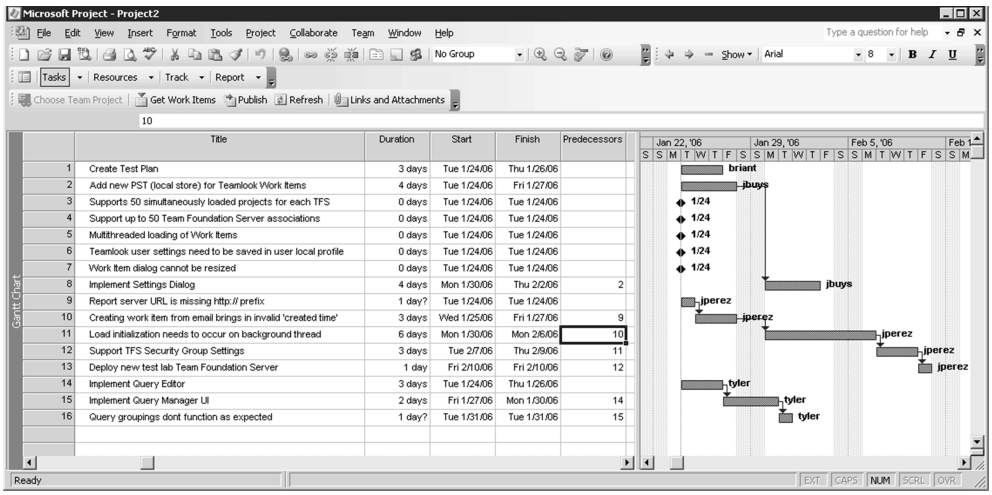


Figure 1.8 With VSTS, the same data can be viewed and edited in Microsoft Excel. The work items, regardless of type, are stored in the same Team Foundation database.



	Title	Duration	Start	Finish	Predecessors
1	Create Test Plan	3 days	Tue 1/24/06	Thu 1/26/06	
2	Add new PST (local store) for Teamlook Work Items	4 days	Tue 1/24/06	Fri 1/27/06	
3	Supports 50 simultaneously loaded projects for each TFS	0 days	Tue 1/24/06	Tue 1/24/06	
4	Support up to 50 Team Foundation Server associations	0 days	Tue 1/24/06	Tue 1/24/06	
5	Multithreaded loading of Work Items	0 days	Tue 1/24/06	Tue 1/24/06	
6	Teamlook user settings need to be saved in user local profile	0 days	Tue 1/24/06	Tue 1/24/06	
7	Work Item dialog cannot be resized	0 days	Tue 1/24/06	Tue 1/24/06	
8	Implement Settings Dialog	4 days	Mon 1/30/06	Thu 2/2/06	2
9	Report server URL is missing http:// prefix	1 day?	Tue 1/24/06	Tue 1/24/06	
10	Creating work item from email brings in invalid 'created time'	3 days	Wed 1/25/06	Fri 1/27/06	9
11	Load initialization needs to occur on background thread	6 days	Mon 1/30/06	Mon 2/6/06	10
12	Support TFS Security Group Settings	3 days	Tue 2/7/06	Thu 2/9/06	11
13	Deploy new test lab Team Foundation Server	1 day	Fri 2/10/06	Fri 2/10/06	12
14	Implement Query Editor	3 days	Tue 1/24/06	Thu 1/26/06	
15	Implement Query Manager UI	2 days	Fri 1/27/06	Mon 1/30/06	14
16	Query groupings dont function as expected	1 day?	Tue 1/31/06	Tue 1/31/06	15

Figure 1.9 Microsoft Project lets you plan and manage some or all of the work items with full round tripping to the Team Foundation database.

Offline Editing, Management, and "What-Ifs" of Work Items

For these tasks, use Excel or Project. Your files are stored locally on your client machine while you work. The changes are written back in Team Foundation when you next synchronize with the database, and any potential merge conflicts are highlighted at that time. On the other hand, when you use Team Explorer, changes are saved to the database during the session.

The extensibility of Team System makes it possible for Microsoft partners to add functionality. For example, Personify Design Teamlook[18] provides team members a view of their Team Projects on multiple Team Foundation Servers from within Microsoft Office Outlook. Team Foundation Server extensibility enables Teamlook to track work items with full accountability in the familiar communications tool, Outlook (see Figure 1.10).

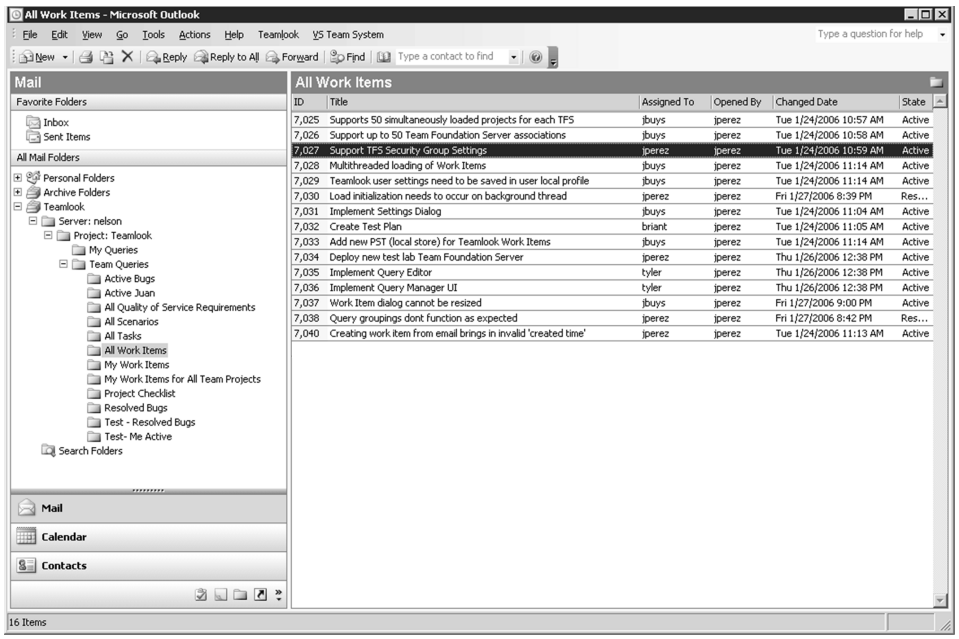


Figure 1.10 With Teamlook from Personify Design, you can also use Outlook as a client for the Team Foundation server.

Instrument Daily Activities

The transparent backlog relies on accurate data to be useful. Often, collecting the data becomes a major activity in itself that relies on willing compliance of large numbers of participants. This disciplined attention to the bookkeeping is rarely sustained in practice, especially during periods of intense activity.

The irony is that the vast majority of the data that a team needs is directly correlated to other actions that are already managed by software. Developers check in code, builds parse that code, testers write and run tests, and all their activities are tracked somewhere—in Project, Excel, the bug database, or timesheets. What if you could gather all that data automatically, correlate it, and use it to measure the process?

Team System takes that approach. It instruments the daily activities of the team members to collect process data with no overhead. For example, every time a developer checks updated code into version control, work items are updated to reflect the tasks and scenarios updated by this code. The relationships are captured in a "changeset," and when the next build runs, it identifies the change sets included and updates work items again with the build number. When tests execute, they use the same build number. Then test results, code changes, and work items are all correlated automatically by Team System (see Figure 1.11).

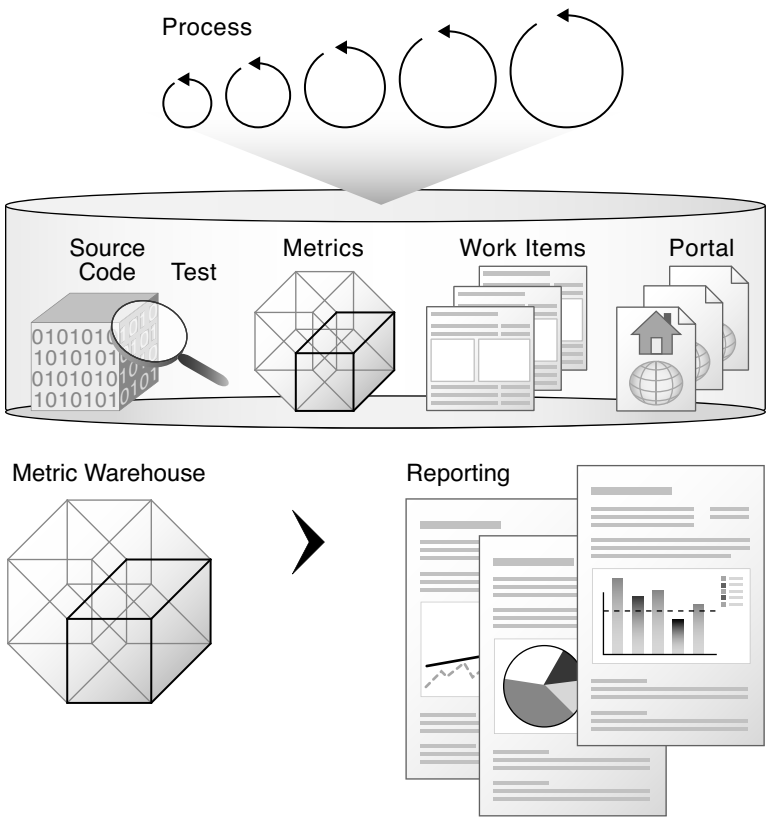


Figure 1.11 The metrics warehouse collects the data from all the actions on the project to provide reports that correlate the different sources and dimensions of data.

In addition to keeping the backlog current and visible, this automatic data collection populates a data warehouse with metrics that reveal trends and comparisons of quality from many dimensions on a daily basis. Just like a data warehouse that provides business intelligence on functions such as a sales or production process, this one provides intelligence on the software development process.

Simple Observations

With this data warehouse, basic questions become easy to answer: Is the project coming in on time, or how far off is it? How much has the plan changed? Who's over or under and needs to have work rebalanced? What rates should we use to estimate remaining work? How effective are our tests? Most project managers would love to answer these basic questions with hard data. When the data collection is automated, the answers become straightforward.

Project "Smells"

More significantly, most project managers would love to find blind spots—places where data indicates a likely problem. It is now common to talk about "smells" for suspicious areas of code.[19] Problems for the project as a whole also appear often as hard-to-pin-down smells, which are not well exposed by existing metrics. I'll cover smells in some detail in Chapter 9, "Troubleshooting the Project," but for now I'll share a common example. Imagine a graph that shows you these bug and test pass rates (see Figure 1.12).

Based on Figure 1.12, what would you conclude? Probably that the Instore Pickup Kiosk code is in great shape, so you should look for problems elsewhere.

At the same time, there's a danger of relying on too few metrics. Consider the graph in Figure 1.13, which overlays code churn (the number of lines added, modified, and deleted) and code coverage from testing (the percentage of code lines or blocks exercised during testing) on the same axes.

Suddenly the picture is reversed. There's really high code churn in Instore Pickup Kiosk, and the code is not being covered by the tests that supposedly exercise that component. This picture reveals that we may have stale tests that aren't exercising the new functionality. Could that be why they're passing and not covering the actual code in this component?

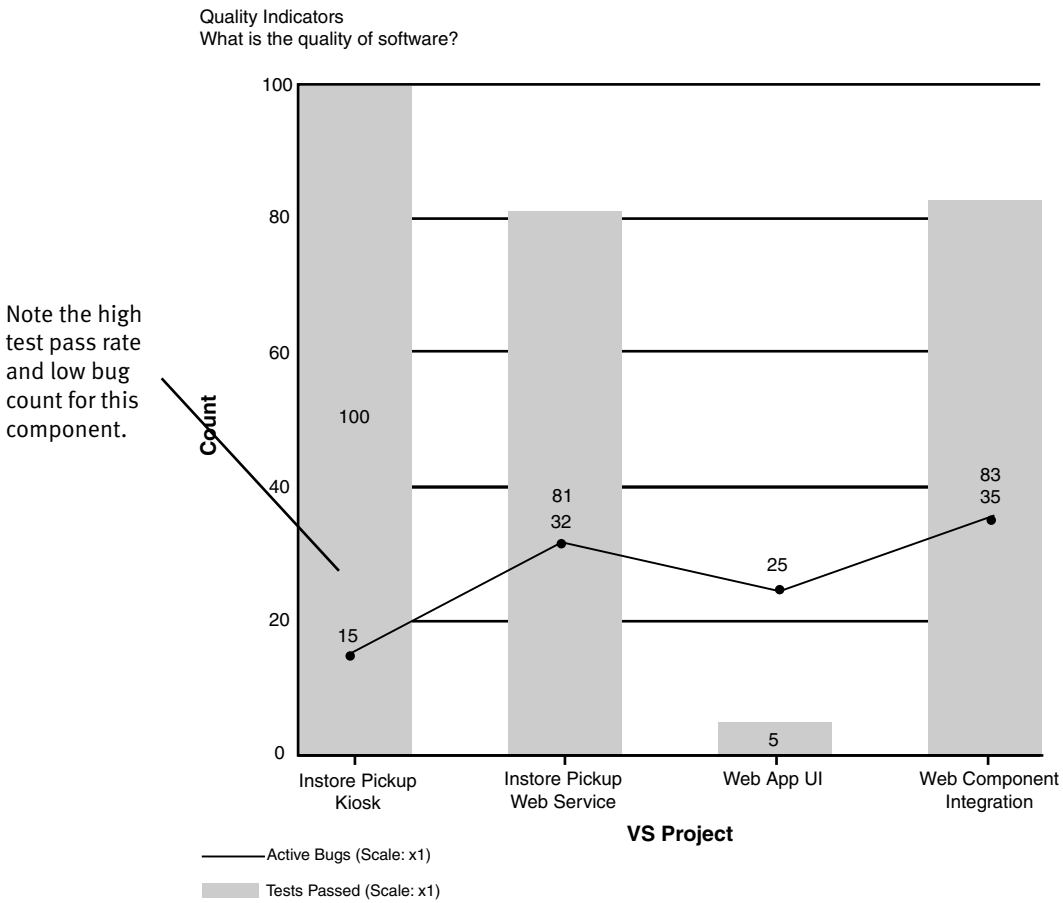


Figure 1.12 **The X-axis identifies different components of your project; the bars show you the test pass rate for each component, while the points and line show the active bug count.**

Multidimensional Metrics and Smells

The ability to see more dimensions of the project data is a direct benefit of the metrics warehouse, which collects and correlates data from daily activities. It provides a quantitative, visual tool to pursue the smells. In this way, you can achieve the visibility level needed for the strictest compliance reporting while working in an agile manner and having the same visibility into a remote project, even outsourced, that you would have in a local one.

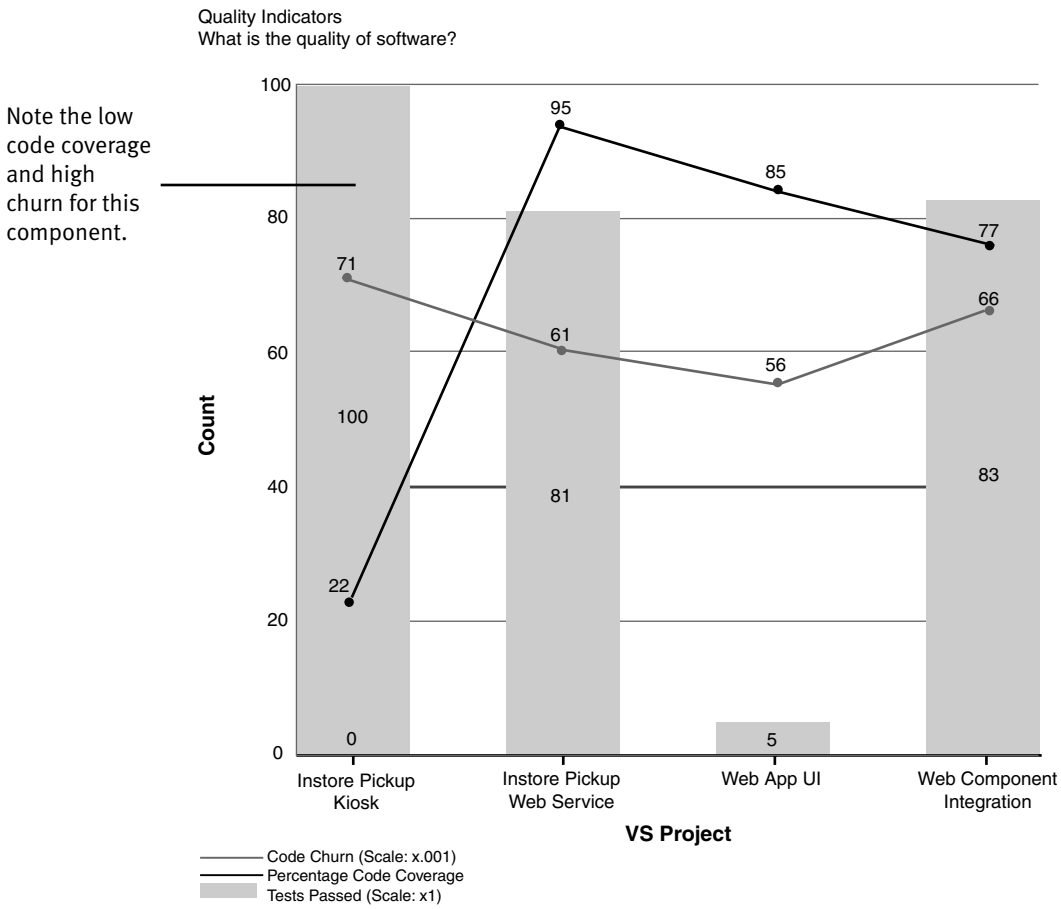


Figure 1.13 Overlaying code coverage and code churn for the components provides a very different perspective on the data.

Fit the Process to the Project

Instrumenting daily activities and automatically collecting data make it much easier to follow a consistent software process. Team System automates the process guidance and instruments the process so that most of the overhead associated with process and most of the resistance to compliance are eliminated.

However, this quickly exposes a valid concern, which is that no one process fits all software projects, even within one organization. Regulatory environment, business

risk, business upside, technology risk, team skills, geographic distribution, and project size all play a role in determining the right fit of a process to a project.

Team System takes the diversity of process into account, enabling the project team to choose or adapt its methodology for contextual realities. When you start a team project in VSTS, you pick a process template, as shown in Figure 1.14. In effect, you can choose and customize the process further for each project, determining not only guidance but also workflow, policies, templates, reports, and permissions.

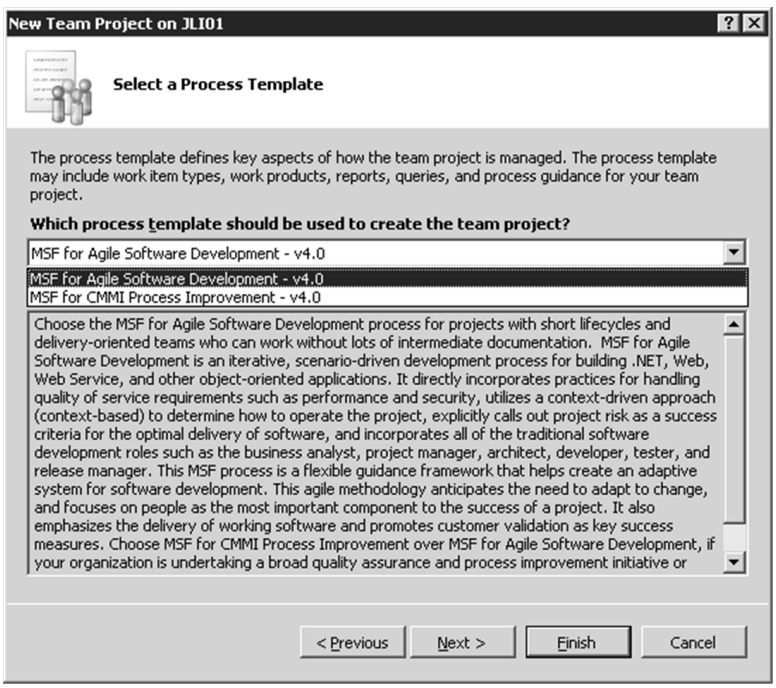


Figure 1.14 When you start a Team Project, your first choice is which "Process Template" to apply for the project. The Process Template defines the process guidance web site, the work item types and their workflow, starter work items, the security groups and permissions, and reports and templates for work products.

Summary

In practice, most software processes require manual enactment, where collecting data and tracking progress are expensive. Up front, such processes need lots of

documentation, training, and management, and they have high operating and maintenance costs. Most significantly, the process artifacts and effort do not contribute in any direct way to the delivery of customer value. Project managers can often spend 40 hours a week cutting and pasting to report status.

This constraint has left process as an exercise for managers, specialist Program Management Offices, and skilled practitioners, who sometimes define metrics and activities quite divorced from the interests of the practitioners or the tools used to implement them. The dominant paradigm in this world has been the work-down view, where software engineering is a deterministic exercise, similar to other engineering pursuits.

In contrast, the business forces driving software engineering today require a different paradigm. In keeping with the dictum "As simple as possible, but *no* simpler," a team today needs to embrace the paradigm of customer value, change, variance, and situationally specific actions as a part of everyday practice. This is equally true whether projects are in-house or outsourced and whether they are local or geographically distributed. Managing such a process usually requires a value-up approach instead.

Typically, the value-up approach requires tooling. Collecting, maintaining, and reporting the data without overhead is simply not practical otherwise. In situations where regulatory compliance and audit are required, the tooling is necessary to provide the change management and audit trails. Team System is designed from the ground up to support the value-up approach in an auditable way. The rest of this book describes the use of Team System to support this paradigm.

Endnotes

1. Thomas Kuhn, *The Structure of Scientific Revolutions* (University of Chicago Press, 1962).
2. Peter Galison, *Einstein's Clocks, Poincaré's Maps* (New York: Norton, 2003), 40.
3. www.agilemanifesto.org
4. See Thomas L. Friedman, *The World Is Flat: A Brief History of the Twenty-First Century* (New York: Farrar, Strauss & Giroux, 2005) for a discussion of the enabling trends.

5. http://www.sei.cmu.edu/cmmi/
6. There are other arguments as well, such as the design complexity of software relative to most engineering pursuits. See, for example, Boris Beizer, "Software Is Different," in *Software Quality Professional I:1* (American Society for Quality, December 1998).
7. The negative consequence of the interplay of variation and dependent events is central to the Theory of Constraints. For example, see Eliyahu M. Goldratt, *The Goal* (North River Press, 1986).
8. The first major work to highlight what I call the value-up approach is Gerald M. Weinberg, *Quality Software Management, Volume I: Systems Thinking* (New York: Dorset House, 1992).
9. The Agile Project Manifesto is available at http://www.pmdoi.org/. It is another example of the value-up approach.
10. Mihaly Csikszentmihalyi, *Flow: The Psychology of Optimal Experience* (New York: HarperCollins, 1990), 71.
11. David J. Anderson, *Agile Management for Software Engineering: Applying the Theory of Constraints for Business Results* (Upper Saddle River, NJ: Prentice Hall, 2004), 77.
12. Ibid., 77.
13. Steve McConnell, *Rapid Development* (Redmond, WA: Microsoft Press, 1996), 126.
14. For a more detailed discussion of this subject, using the nomenclature of the Theory of Constraints, see David J. Anderson and Dragos Dumitriu, "From Worst to Best in 9 Months: Implementing a Drum-Buffer-Rope Solution in Microsoft's IT Department," presented at the TOCICO Conference, November 2005, available at http://www.agilemanagement.net/Articles/Papers/From_Worst_to_Best_in_9_Months_Final_1_2.pdf.
15. The Standish Group (www.standishgroup.com) publishes a biennial survey called "The Chaos Report." According to the 2004 data, 71% of projects were late, over budget, and/or canceled.
16. Ken Schwaber and Mike Beedle, *Agile Software Development with SCRUM* (Upper Saddle River, NJ: Prentice Hall, 2001), 32–3.

17. Bellotti, V., Dalal, B., Good, N., Bobrow, D. G., and Ducheneaut, N., "What a to-do: studies of task management towards the design of a personal task list manager. " ACM Conference on Human Factors in Computing Systems (CHI2004); 2004 April 24–29; Vienna; Austria. NY: ACM; 2004; 735–742.
18. http://www.personifydesign.com/
19. Originally used for code in Martin Fowler, *Refactoring: Improving the Design of Existing Code* (Reading, MA: Addison-Wesley, 1999), 75.