Chapter 1

Sustainable Software Development

Sustainable software development is a mindset (principles) and an accompanying set of practices that enable a team to achieve and maintain an optimal development pace indefinitely. I feel that the need for sustainable development is an important but unrecognized issue facing software organizations[1] and teams today. One of the more interesting paradoxes in the high-tech sector is that while the pace of innovation is increasing, the expected lifetime of successful software applications is not decreasing, at least not in a related way. This chapter outlines the value of sustainable development, while the next chapter discusses the pitfalls of unsustainable development.

The more successful an application or tool is, the greater the demands placed on the development team to keep up the pace of innovation and feature development. Think of products like Adobe Photoshop, PowerPoint, SAP, or Oracle. These products are all successful and continue to be successful because their development teams have been able to meet user's needs over a long period of time despite persistent competitive pressures and changing technology and market conditions.

Unfortunately, there are too many projects where there is a myopic focus on the features in the next release, the next quarter, and the current issues such

1. My intended definition of *organization* is any group of people dedicated to a software project. Hence, it could be a software company, an IT organization, a software team, and an open source project. I assume that the organization is ideally multifunctional and so does not consist purely of software developers. Hence, the organization would include business people, documentation, usability and user interface designers, quality assurance, etc.

as defects and escalations reported by customers. The software is both brittle and fragile as a result of factors such as over- (or under-) design, a *code first then fix defects later* (*code-then-fix)* mentality, too many dependencies between code modules, the lack of safeguards such as automated tests, and supposedly temporary patches or workarounds that are never addressed. These are projects that are unknowingly practicing *unsustainable* development.

In unsustainable development, teams are primarily *reactive* to changes in their ecosystem. By and large, these teams are caught in a *vicious cycle* of reacting to events and working harder and longer hours akin to being on a treadmill or walking up a down escalator. The result is a project *death spiral*, where the rapidity of descent depends on the amount of complexity faced by the team and its principles and practices and discipline.

In *sustainable development*, teams are able to be *proactive* about changes in their ecosystem. Their ability to be proactive is enabled by their attention to doing the work that is of the highest value to customers with high quality and reliability and an eye toward continual improvement despite increasing complexity. These teams are in a *virtuous cycle*, where the more team is able to improve themselves and how they work together, the greater their ability to deal with increasing complexity and change.

Underlying sustainable development is a mindset that the team is in it for the long haul. The team adopts and fosters principles and practices that help them continually increase their efficiency, so that as the project gets larger and more complex and customer demands increase, the team can continue at the same pace while keeping quality high and sanity intact. They do this by continually minimizing complexity, revisiting their plans, and paying attention to the *health* of their software and its ability to support change.

Sustainable Development

Sustainable development is a mindset (principles) and an accompanying set of practices that enable a team to achieve and maintain an optimal development pace indefinitely. Note that optimal doesn't mean *fastest*—that would be pure coding, such as for a prototype.

Sustainable development is about efficiency and balancing the needs of the short and long term. It means doing just the right amount of work to

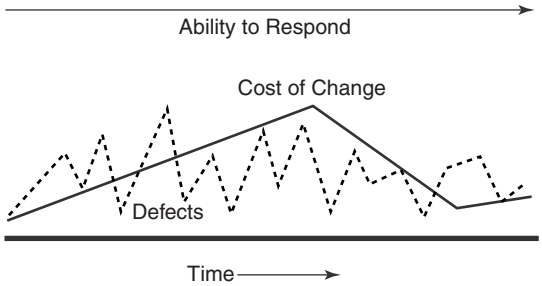

Figure 1-1
In sustainable development, the cost of change stays low over time. The team is able to respond to changing requirements and changes to the software's ecosystem. This is a pace that the team can maintain indefinitely. Key indicators of sustainable development are an ability to keep the number of defects relatively constant over time while recognizing that the software must be modified to keep the cost of change under control.

meet the needs of customers in the short term while using practices that support the needs of the long term. There are not enough software projects today where over time a team can stay the same size (or even shrink) and still deal with the increasing complexity of its software *and* its ecosystem *and* increasing customer demands. In sustainable development, the needs of the short term are met by regularly producing software that has the highest possible value to customers. This is done while keeping the cost of change as low as possible, which lays the foundation for future changes and makes it possible to quickly respond to changes in the ecosystem.

Sustainable development, as depicted in Figure 1-1, is a liberating experience for the lucky teams who can achieve it. While they have to deal with stress in the form of constant change, they have the advantage that they are *in control* of the situation and can outship their competitors because they are able to respond more rapidly and at a much lower cost. They are also able to be proactive about new technologies or new opportunities in any form.

Chemical Manufacturing and Sustainable Development

Some software companies are able to periodically reinvent themselves and their products. These companies don't need to completely rewrite their software products and in fact are able over time to add to their product line, usually with the same underlying technology and implementations. How do

these companies do it? For some of the answers, let's look at some interesting research from a seemingly unrelated industry: chemical manufacturing.

Some interesting research into productivity at chemical manufacturing plants has parallels in software development [Repenning and Sterman 2001][2]. This research focused on chemical plants that are in deep trouble. These are plants that had low productivity, low employee morale, etc. The companies who owned the plants were seriously considering, or in the process of, closing them down and moving operations to another location with higher returns on investment.

What the researchers found is that in the plants in question the first response to trouble is to ask people to work harder, usually through longer hours. However, while working harder results in a definite short-term increase in overall capability, the long-term effect is actually *declining* capability, as shown in Figure 1-2.

One of the reasons for declining capability over the long term when working harder is the resulting vicious cycle or *death spiral*. This cycle is due to unanticipated side effects of the decision to work harder: As more hours are worked, more mistakes are made, and there is a greater emphasis on quick fixes and responding to problems. These mistakes and quick fixes lead to the requirement for more work.

In a chemical plant a mechanic might incorrectly install a seal on a pump. The seal might rupture hours or days later. When it does, the entire pump would need to be replaced, which takes an entire section of the plant offline while leaking more chemicals into the environment. People will think they aren't working hard enough, so they'll put in more hours. Then, the extra costs of all the overtime and environmental cleanups kick in and costs are cut in other areas such as carrying fewer spare parts to compensate. When parts aren't available when needed, the plant is down longer. The longer the plant is down, the greater the reduction in revenue and the higher the costs. The greater the reduction in revenue, the greater the pressures to further reduce costs. Eventually, people are going to be laid off, and the

2. In this section I decided to keep the author's terminology of *working smarter* and *working harder*. However, I do recognize that these terms are clichés, and it's not wise, for example, to tell someone he or she needs to work smarter or that the person is working too hard. Hence, in the remainder of this book I use working long hours and emphasizing feature development instead of working harder and continual improvement instead of working smarter. These are rough semantic equivalents.

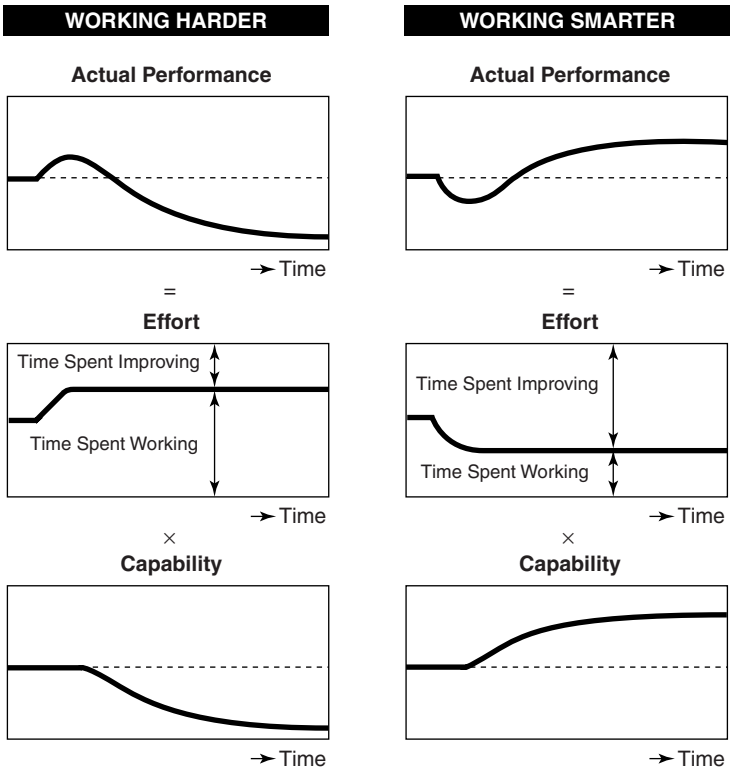

Figure 1-2
Working harder (more hours) results in declining capability over time. Working smarter, with an emphasis on continual improvement, leads to increasing capability. From [Repenning and Sterman 2001].

fewer people available, the less the ability of the plant to produce output. And so it goes.

Parking Lot Managers

I believe there are too many people in the software industry, managers especially, who judge the morale or productivity of their company by how many hours their employees work on evenings and weekends on a regular basis. I call these people *parking lot managers* because they're often *proud* of the fact that their company's parking lot is still full at midnight and on weekends. However, very few of these managers realize that full parking lot effort is not sustainable, that working harder may be valid in the short-term when a concerted effort is required, but it is definitely not in the best long-term interests of the company.

> Companies need people who treasure the contribution they make when at work and who are passionate about the success of the company. This has no correlation with the number of hours worked . . .

The largest reason for a decline in long-term capability is that working harder results in an inability to implement necessary improvements. In the plants studied, mechanics were too busy fixing problems in the pumps to do anything else. As any car owner who ignores basic regular maintenance knows, the longer mechanical parts are left untended, the greater the chance they will eventually fail, not to mention the greater the eventual cost. This leads to another vicious cycle: The harder people work and the more problems they are trying to fix (or more appropriately, the more fires they're trying to put out), the greater the chance that problems will continue to build and grow worse over time. No doubt you've been in situations like this. The problem quickly becomes one of having time stand still through continuous death march releases, or fixing things.

The employees of the chemical plants turned things around by developing a realistic simulation of their situation. The simulation was developed in such a way that it demonstrated to participants the results of various decisions. Importantly, the simulation was *not* designed to teach or test skills. They recognized that the mechanics, for example, didn't need to be taught to be better mechanics; after all, they were very adept at their craft through all the crucial problems they had to fix on the spot. The simulation, implemented as a game, realistically demonstrated the various important tradeoffs that can be made in a plant between working harder and *working smarter*. In a chemical plant, working smarter consists of activities like preventive maintenance, where a pump is taken offline, disassembled, examined, and lubricated on a regularly scheduled basis. Working smarter is also taking the time to examine the entire plant's systems and processes and continually attempting to identify problems before they occur while reducing the complexity of the overall system in order to increase efficiency.

The results of the simulation were an eye opener for the plant's employees. The results were also counterintuitive to many: They showed that working smarter (especially doing preventive maintenance) consistently produced better results over the long term. The simulation also demonstrated that with any attempt to work smarter there is an initial dip in capability caused by the need to let some problems go unfixed while initial

improvements are made as shown in Figure 1-2. This helped the employees expect the dip in capability and have the persistence to follow through with the changes (a perfectly human response would be to think that the dip was permanent and revert back to the old ways). Eventually, as the number of improvements started to make a difference, capability would climb until a point where the plant entered a *virtuous cycle*, where each additional investment in improvements led to further efficiencies and gains in output with lower environmental impact, which in turn led to more time being available to make further improvements, and so on. People were actually able to accomplish *more* work by working smarter than they had before.

As a result of the simulation, the chemical plants described experienced a complete turn-around. Not only were these plants kept open, but they also received additional work and their business grew. And some of the changes introduced by the employees had a lasting effect, some with a return on investment of 700,000 percent! The most astonishing thing, which perhaps isn't so astonishing when you consider the rut these companies were stuck in, is that virtually all of the changes that were required to make the turn-around were well known to the employees but they'd never been implemented because they were always too busy!

Continual Improvement: The Accelerator Button

The study on capability in a chemical manufacturing plant is surprisingly relevant for software companies. The main lesson is that in sustainable software development, you need to strive for continual improvement while resisting the temptation to focus on features and simply working long hours to meet demands. A software company's business is only as sound as its factory, where the factory is made up of a number of "pumps"; the *software* that the company produces. You may need to take some of your pumps offline occasionally, and every few years you will realize that your factory is completely different than it was previously because of all the changes made to the pumps. And as with real-world factories, only in extreme circumstances, such as when a disruptive technology is on the verge of widespread

adoption or a paradigm shift is about to occur, will you need to completely replace the factory.

Each of the following questions examines some of the parallels between chemical manufacturing plants and software development.

How many developers view writing test code, building testability into their code, enhancing a test framework, or cleaning up their product's build system as nonessential to their work and hence something that lower paid people should be doing?

Somehow, many developers have a mindset that non-feature work is not part of their job. This is not only preposterous but also extremely dangerous and is akin to a mechanic at a chemical plant believing the basic upkeep of his tools and workshop is not part of his job. If developers don't write automated tests, then chances are the culture of the company is focused on defect *detection*, not defect *prevention*. As shown in Chapter 5, this is the worst possible situation a software company can get itself into because it is extremely expensive and wasteful, with many defects being found by customers (which is the most expensive scenario of all). Likewise, if developers don't personally pay attention to the infrastructural details of their products (such as build and configuration management systems), it is all too easy for problems to creep in that eventually impact key performance indicators that also impact developer's productivity. Examples are build times, code bloat, source tree organization, and improper dependencies between build units creeping into the product.

How many managers refer to non-feature work as a *tax* that is unessential to product development?

If a company is in a feature war with its competition, the company's management needs to understand how the decisions and statements they make ultimately impact the long-term success of the company. Managers must recognize that they have a key role in leading the attitude of the organization. In this case, management, or even in many cases software developers themselves, need to realize that the only tax on the organization is their attitude about non-feature work. As in chemical manufacturing plants, it is counterintuitive that ongoing investment in continual improvement

will lead to greater capability than solely concentrating on features and bug fixing.

How many teams are too busy to implement any improvements, such as adding to their automated tests, improving their build environment, installing a configuration management system, rewriting a particularly buggy or hard to maintain piece of code, because they're "too busy" developing features and bug fixing?

Just as the mechanics in the chemical engineering plants are unable to perform preventive maintenance by taking a pump offline on a regularly scheduled basis because they're too busy fixing problems (i.e., fighting fires), many software teams are also unable to pay attention to the "health" of the underlying software or the development infrastructure. As described earlier in this chapter, this is a key part of the software development death spiral. It will eventually lead to a virtual crisis with a team spending an increasing amount of time during each release fixing defects at the expense of other work. Either the company will eventually give up on the product or be forced into a likely rewrite of major portions of the software (or even the complete software) at great business expense.

When developers are fixing defects or adding new features, how often do they use shortcuts that compromise the underlying architecture because they're "under a tight deadline"?

The main danger with these shortcuts is that they're almost always commented (e.g., ugly hack introduced before shipping version 2) but rarely fixed. These shortcuts are *broken windows* (see chapter 4 for more details) in the software and are another part of the software death spiral.

Are the people who develop the products in your company proud of the products, but not proud of how the products were created?

If your products are absolutely brilliant but your people are burned out by the experience of individual releases, maybe it's time for a step back. Maybe there is too much heroism required, or too little planning takes place, or there is an ongoing requirement for long hours because the

schedule is slipping. There are many possible factors, but the underlying cause can usually be traced back to a lack of focus on continual improvement and technical excellence.

The Genius of the AND versus the Tyranny of the OR

One way to think about continual improvement is through the genius of the AND and the tyranny of the OR [Collins and Porras 2002]. Software organizations need to stop thinking about features and bug fixing as being exclusive from underlying software health and infrastructure improvements. This is the tyranny of the OR; thinking you get features OR software health. By focusing on continual improvement through paying constant attention to sustainable practices, software companies will be able to achieve the genius of the AND: features and bug fixing AND underlying software health. Greater capability leads to an *increased* ability to introduce *more features* with *fewer defects*, and hence have *more* time to innovate, not less. Hence, focusing on continual improvement is not a tax it's an ***accelerator button***[3]!

A Sustainable Development Experience

I have been fortunate to work on at least one team that achieved sustainable development. In one case, I worked on a software project that was released to customers over the course of several years. Over this time, the capabilities, complexity, and size of our product increased but the number of defects in the product stayed constant and our ability to innovate *increased*. And our team did not increase in size. How did we do it?

- Our software worked every day.
- We relied heavily on automated testing to catch problems while we were working on new features.
- We had high standards of testing and code quality, and we held each other to those standards.

3. Jim Highsmith deserves the credit for this observation.

- We didn't overdesign our work and built only what our customers needed.
- We were uncompromising with defects, and made sure that all the known defects that were important to our customers were fixed in a feature before we moved on to the next one. Hence, we never had a defect backlog.
- We replanned our work as often as we could.
- We were always looking for ways to improve how we worked and our software.

Other teams in our company were incredulous that we could produce the amount of work we did *and* keep quality so high. For example, the company had an advanced practice of frequent integration, which was vital because the product was developed across many time zones. Because of our stability and quality we were able to pick up and fix integration problems extremely early in a development cycle. This was vital to the success of the company's products.

Think of Cobol when you think of sustainable development: The original developers of the Cobol language could not conceive of programs written in Cobol that would still be in use in 1999, and yet when the year 2000 came along, all of a sudden there was a huge demand to fix all the Cobol applications. Here's a related joke:

It is the year 2000 and a Cobol programmer has just finished verifying that the Y2K fixes he has made to a computer system critical to the U.S. government are correct. The president is so grateful that he tells the programmer that he can have anything he wants. After thinking about it for a while, the programmer replies that he would like to be frozen and reawakened at some point in the future so he can experience the future of mankind. His wish is granted.

Many years pass. When the programmer is woken up, people shake his hand and slap him on the back. He is led on a long parade, with people cheering him as he goes to a huge mansion. The master of the universe greets him enthusiastically. Pleased but puzzled, the programmer asks the master of the universe why he has received such a warm reception. The master of the universe replies "Well, it's the year 2999 and you have Cobol on your resume!"

Summary

Developing software is a complex undertaking that is performed in an environment of constant change and uncertainty. In the Introduction, I likened this to a coral reef, not only because of the complexity of the software ecosystem and the need for constant change, but also because of the fragility of existence. It is very hard to build or inhabit a software ecosystem that thrives over the long term.

Very little software is written once, installed, and then never changed over the course of its lifetime. And yet, the most prevalent development practices used in the industry treat change as an afterthought. Competition, the changing ecosystem, and the fact that users (and society in general) are becoming increasingly reliant on software, ensure that the software must change and evolve over time. The resulting combination of increasing complexity, need for change, and desire to control costs is a volatile one because very few software organizations and teams are equipped with the mindset, discipline, and practices to both manage and respond to complexity and change.

The answer to all the stresses placed on software organizations and teams lies in sustainable development, which is the ability to maintain an optimal pace of development indefinitely. In sustainable development, teams are able to be proactive about changes in their ecosystem. Their ability to be proactive is enabled by their attention to doing the work that is of the highest value to customers with high quality and reliability and an eye toward continual improvement *despite* increasing complexity. These teams are in a virtuous cycle, where the more the team is able to improve themselves and how they work together, the greater their ability to deal with increasing complexity and change.

The next chapter describes unsustainable development and its causes. This is important to understand before considering how to achieve sustainability.