



1

NO PROGRAMMER DIES

The Bible shows the way to go to heaven, not the way the heavens go.

—GALILEO GALILEI

There is one question that is so frequently asked in software engineering that it may seem tedious to ask it yet again, but here it is, anyway: “What are the basic differences between software development projects and engineering projects (or manufacturing production), that is, say, between producing enterprise information system and building tunnels or manufacturing cars?”

The usual, dry, academic answer is that software is a conceptual, intangible, invisible artifact. This definition may be useful, but there is another attribute of software projects that distinguish them even more starkly from traditional engineering projects. The distinction, which is rarely mentioned, is that — while engineers may always be in danger — *software developers are never killed or injured while working on their projects.*

No matter how lousily or messily planned or implemented a software development project may be, nobody in a software team is ever seriously physically hurt at the office computer. There is a clear difference between developing Adobe and digging the Panama Canal, and this may be one reason why so many software development projects are hastily, carelessly, and sloppily managed.

In 2005 the death toll from a tunnel project in western China broke a new record indicating project mismanagement and poor supervision of safety procedures. An investigation revealed that many fatal accidents

Software Development Rhythms: Harmonizing Agile Practices for Synergy

By Kim Man Lui and Keith C. C. Chan

Copyright © 2008 John Wiley & Sons, Inc.

could have been avoided. This brought the issue of the rushed productivity for the project rather than workers' and engineers' safety, environmental concerns, and social needs under even closer scrutiny. The public severely blamed the chief engineer for the tunnel accidents.

—LOCAL NEWS IN CHINA, 2005

In real-world engineering projects, the prospects and costs of death are always looking over our shoulders, holding individuals personally responsible for the consequences of their decisions and actions. Thus, project managers must adhere to strict procedures and industrial standards: agreed-on plans, signed confirmations, written workflows, and timing. When an error occurs, a project management model enables us to track the work process, conduct a postmortem review, and identify errors; in addition, this may also involve financial issues of insurance and litigation.

Because life matters¹ and mistakes incur heavy costs, real-world engineering demands discipline, consistency, consideration, commitment to detail, and a strong sense of teamwork. The result is not just greater safety; it's also better products. You have to wonder whether software development can afford to continue in its current (often) irresponsible way. Are there any factors in society or the marketplace that will ever make it change? If so, what are they?

1.1 DEVELOPING SOFTWARE VERSUS BUILDING A TUNNEL

Many types of cancer are treated with radiotherapy, in which high-energy rays are used to damage malignant tumors. Given the danger of overdosage, the amount of radiation energy is supposed to be precise, and safely controlled by a computer system. The Therac-25 was developed for this purpose by the Atomic Energy of Canada Limited from a prototype in 1976 to a safety analysis in 1983 (Leveson 1993). Between 1985 and 1987, the Therac-25 overdosed a number of patients, killing five. Subsequent investigations blamed the software, but there's something strange in this. The programming code for the Therac-25 was built by only one person, who also did most of the testing. This is not even conceivable in a real-world engineering project, but in some software development the programmers are often responsible for their own testing. How did the software development process ever get itself into this mess?

¹The ISO 9000, which have often been compared with CMM and CMMI, came from BS5750, which was adopted to control quality problems of bombs going off in munitions factories and bombs not blasting when they should have.

1.1.1 The Good Old Days?

In 2005, a Helios Airways Boeing 737 crashed in Greece, with the loss of all 121 on board. The suspected cause was a series of design defects in the 737 where the plane's pressurization warning alarm made the same sound as the improper takeoff and landing alert. Confusion over the reason for the warning may have contributed to the fatal crash. When things start to go wrong, it sometimes doesn't take much to spin them right out of control. Factors that may seem trivial in normal circumstances, may contribute to tragic outcomes when things aren't going according to plan .

Regardless of whether engineering product defects may be unavoidable, we are taught that rigorous development processes do remove as many as possible. A "rigorous" process normally means the separation between planning and execution. During construction, planned tasks should be designed to be strict to follow and easy to control. Ideally, constructive peer pressure should positively shape workplace behavior to ensure that a development process will be "as rigorous as possible."

Adopting that philosophy in engineering management, software development activities can normally be divided into two types of process—(1) *analysis and design as planning* and (2) *programming as execution*, with (2) following (1). This intuitive model, generally referred to as the "waterfall model" by Winston Royce (1970), is normally adopted when managing large software projects. These two processes are often chopped into smaller but still ordered processes. Dividing and conquering allows us to better allocate limited resources and control and track project progresses through a number of checkpoints and milestones. The analysis–design process is made up of such activities as software requirements gathering, system analysis and logical design, while the programming process is made up of coding, unit testing, system integration, and user acceptance testing, all of which are basically linked serially, one after the other. For the purpose of discussion, we consider here what is called a *four-stage waterfall model* as below:

Requirement→design→coding→testing (R→D→C→T)

The nature of the waterfall model makes it easy for a project plan to be executed the same way engineers manage their projects. Focusing on breaking down larger tasks into smaller tasks and putting them in the right order for execution better allows project resources to be allocated and conflicting problems to be resolved. With this idea of the separation between planning and execution behind a waterfall model, a project plan can be reviewed to optimize against

time and resources. With this, we can then identify and weight various risk factors to draw up a contingency plan for a project.

Such a project management paradigm to develop software may sound intuitive, but one could easily discover that it does not encourage the exploration of interrelationships between people, programming tasks, and software practices. It can be difficult for some project managers to comprehend development synergies between these three elements, particularly in a situation where something can change unexpectedly during execution.

1.1.2 The More Things Change, the More They Stay the Same?

When project requirements are constantly changing, sometimes more rapidly than what we had imagined, and when developers know that what they are building is not what their customers need, they will start to realize that their software can be developed only by progressing in small steps so as to obtain constant feedback all the time. This is, however, easier said than done.

Our thinking is often limited by our past experience. For many software managers, their formative software experience is with the waterfall. Seeking to improve on it, we come up with an enhanced waterfall. As single-phase analysis for user requirements may rarely provide a full solution, more than one phase is often considered necessary, and for this, straightforwardly, we link two or smaller waterfall cycles together in a chain.

$$R \rightarrow D \rightarrow C \rightarrow T \rightarrow R \rightarrow D \rightarrow C \rightarrow T \rightarrow R \rightarrow D \rightarrow C \rightarrow T$$

There is really nothing new here. The same principles behind the waterfall model apply except that, in each cycle, one can plan according to the feedback obtained from what has previously been done. The current cycle will therefore be less stringent and more flexible than previous cycles.

The waterfall model, if strictly implemented as “one cycle” or some “bureaucratic procedures for turning back,” may not be too popular in the commercial world. Many software teams take the concept of the waterfall model but implement their software projects more flexibly. Some teams adopt the enhanced waterfall model while still others may go even further to adopt an adaptive model so that the length and activities in each iteration can be dynamically adjusted. All these models can be considered as belonging to a waterfall family of models.

In some extreme cases in such a family, to deal with unexpected changes, some software managers would substantially revise their project plans on a weekly basis. Since they know that none of their team members could die or be injured, they are free to revise their plan to cope with any

change when it occurs. Compared to software projects, in engineering projects this would be considered very unusual. It would be more normal to delay the project rather than risk changing what and how we have already planned and managed.

When project variables keep changing, a revision of a project plan is the way out of potential crisis. Many project managers do not care how often the project plans are revised as long as it is necessary. But, what really matters is our way of thinking being limited to the style of waterfall management, which always involves breaking down tasks into many sequential tasks, and resources, responsibilities, and any understanding of any bottleneck are planned along this line. Whenever there is any change, replanning is needed and it is hoped that the revised plans can reflect the situation as quickly as if such changes were already anticipated. This is undesirable as a software team does not manage change in this case; they are, instead, managed by change.

1.1.3 Behind Software Products

Let us look at the design and planning of manufacturing products and then come back to software products. If a product is supposedly made up of a number of components, subcomponents, and sub-subcomponents, and so on, then one can draw up a hierarchical architecture that consists of the complete product at the top with a hierarchy of subcomponents, which, in turn, are made up of sub-subcomponents, and so forth. This structure is called a *bill of materials* (BOM) and it is at the heart of operations in many assembly plants. It supports assembly task planning in manufacturing resource planning (MRP), as shown in Figure 1.1, where one plans when, what types, and what

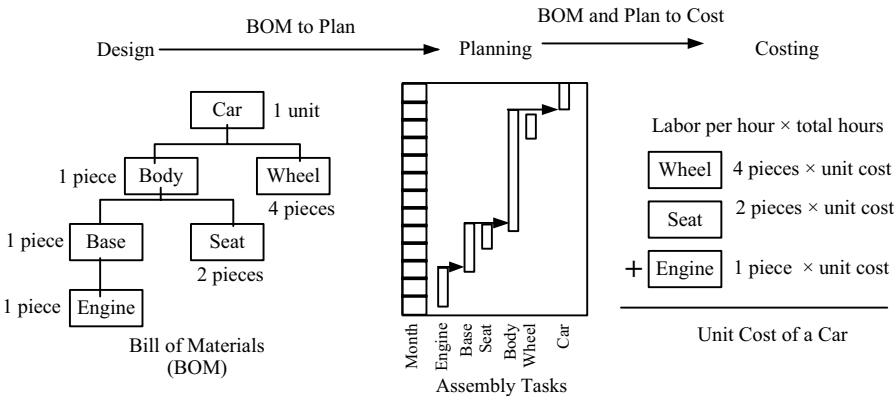


FIGURE 1.1 How bill of materials (BOM) can be used for planning and costing.

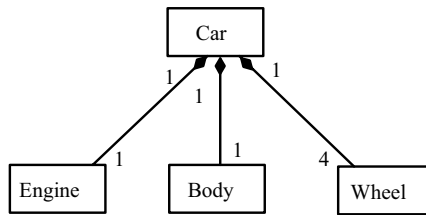


FIGURE 1.2 Class diagram for a car (simplified) that resembles bill of materials (BOM) but serves a different purpose.

quantities of materials or subcomponents are needed for production (Chapman 2006). The assembly task planning will allow costing to be determined (Figure 1.1). Subcomponent information can be used to do cost rollup calculation for customer quotations and for effective internal control over production costs.

Similar to engineering projects, software is often designed using a class diagram (see Figure 1.2), which resembles a bill of materials. Class diagrams help us understand attributes, methods, and class component relationships. Unfortunately, we could *rarely* use a class diagram to tell us how to do assembly task planning and costing. It would be good to have an integrated approach to tighten up or clarify what needs to be written and how a project should be planned. Only recently has it become possible to do this to some extent through the concept of a “user story” in eXtreme programming (XP), which can be used both for requirements management and project planning.

Compared to software tasks, other engineering tasks are often more tangible. Components built in a typical engineering project can be combined in the order suggested by a bill of materials (BOM) so that work progress can be objectively measured and quality can easily be monitored. This, when compared with software, is more tangible. For instance, as part of an engineering project, one can assemble an engine to the gearshaft and then form the base before installing the wheels and finally carrying out wheel tests. The sequence in which these tasks are performed could be designed in accordance with both physical constraints and economic efficiency, and this sequence somehow solidifies the idea of the separation of planning and execution into two stages.

In software projects, products cannot be assembled with this kind of job sequence as defined with class diagrams in the same way BOMs are, no matter how these products are designed. Programmers can work out login interfaces and main menu interfaces in an order that corresponds to how the users

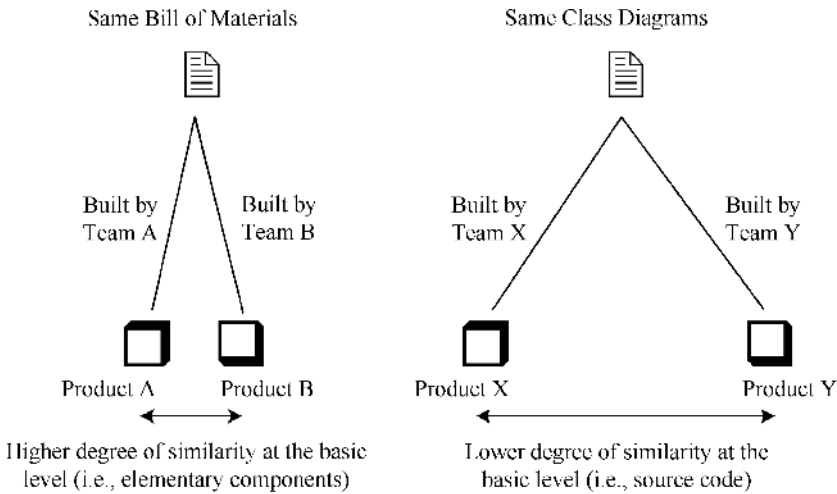


FIGURE 1.3 *Degree of difference* is a conceptual term measuring how two products can be built differently using the same design.

operate the system. But they can also do these later on.² There are virtually no restrictions on the ordering when we build with software components. Walker Royce (2005) of IBM suggests that software managers should manage software in the same way as managing a movie production rather than as a typical engineering production. To make a film, we have to effectively assess how all the elements of scenes of the film work together (actors, timing, lines, sets, props, lighting, sound, etc.) so that scenes of the film will be shot in a way that will minimize production costs and production time so that the film can be completed with the least amount of time and money.

In manufacturing, when two products, designed by two groups of engineers, eventually appear on the same BOM, we can almost speculate that these products should be built in similar ways. Furthermore, since products are built to follow the design as given by a BOM, if there are defects, either they are design problems or the products have not been made to plan.

Unfortunately, this same logic that is applicable to manufacturing does not apply to software development (refer to Figure 1.3). Unlike BOMs, class diagrams do not fully address code implementation. Given the same diagrams, implementation could be done in a variety of ways by different programmers. The programmer will not have to write the same software twice for a second installation, but may have to redo it for a second version,

²“It may make some kind of logical sense that you have to finish writing the login servlet before you start the logout servlet; but in reality you could write and test them in any order,” said Robert Martin (2003).

and this can be done even without modifying the class diagrams! For instance, programmers may tune structured query language (SQL) algorithms for better performance when they know the characteristics of real data. Some software teams will adopt the practice of revisiting each other's code to detect defects and improve readability. Again, none of this necessarily implies redesigning the class diagrams.

In the case of software projects, not all that is well designed ends well! Worse yet, many software problems cannot be classified as problems even when the class diagrams or code are not written in compliance with the design. Bad code but good design is not that rare! In short, having qualified experienced system analysts do design using data models, unified modeling language (UML) diagrams, and so on, is not the only necessary condition for producing good software; we also need qualified experienced programmers to write code to build the system. Furthermore, with the right design and good-quality code, we need skilled testers to discover bugs in products. Managing these people effectively in a team, whether each member has just one role (e.g., system analyst, programmer, tester) or multiple roles requires a methodology for coordination, collaboration, and communication! Left to themselves, things may go wrong, and once they do, they will go from bad to worse. One cannot expect a bunch of the right technical people sitting together (without proper management or coordination) to produce software on time, within budget, and according to requirements if there is no development framework.

1.1.4 Deal or No Deal

Traditionally, software management emphasizes mainly relatively formal, rigorous, software development processes. Recently, agile development approaches have grown quite popular. There is now an agile or eXtreme version for formal methods, testing, database, tools, or project management. Although this new trend has attracted great attention in the software community, it has not taken over the waterfall model as the dominant approach. In fact, agile practices are often adopted within a waterfall framework. It appears that the waterfall model is either so intuitively better than the others or that software developers have been so used to it that they cannot think of any other ways better.

The popular TV game show *Deal or No Deal* displays a number of briefcases, each of which contains a different cash prize ranging from just one dollar to millions of dollars. A contestant who wins a game on the show is allowed to pick any of these briefcases as a prize. The contestant, however, is not allowed to open the briefcase until the end of the game. As the game

progresses, a “Banker” offers the contestant a deal to buy the chosen briefcase. If the contestant rejects the deal, other cases can be chosen and opened while the banker continues to make offers to the contestant regarding the suitcase the contestant chose at the beginning. The contestant can either accept the banker’s offer or take the cash prize inside the briefcase selected at the beginning of the game. It is interesting to note that many contestants who had chosen the right briefcase often accepted a lower-value offer from the bankers. They would have, say, accepted \$250,000 dollars, rather than resisting temptations to hold onto the end to win millions. Even in the presence of favorable odds, it is interesting that many people are actually highly risk-averse (Post et al. 2006).

In a study involving 150 volunteers (Tversky and Kahneman 1981), who were asked to choose between a guaranteed \$250 or a 75% chance to win \$1000 dollars, the overwhelming majority (84%) of the participants took the \$250 cash. Interestingly, when the choice was between winning or losing \$750 dollars with a 75% chance, 87% preferred to try their luck. Mathematically, the odds were the same but not the subjects’ perception of winning and losing.

Daring to take risk for a higher reward is an entrepreneurial attitude. For entrepreneurs to be successful, they need to be risk-takers. They need to understand the odds on success and failure, so that they can spot markets and seize opportunities before others do. If not, they need to have the gamblers’ attitude. Compared to an entrepreneur or a gambler, how much risk is a software project manager willing to take when adopting a new development methodology? On the surface, this seems to be a matter of personal preference. However, it may be a bit more complicated than this. There is a chance that the members of a software team may not be so cooperative. They may try to stick to their usual way of thinking and work consciously or subconsciously toward it. If things do not seem to go as originally expected, these members may well place the blame on the manager. They may say that the manager should have been more prudent and should not have replaced the usual practice with something unproven. Is this prudence? Does fear overwhelm ambition? Or is it politics that has raised its ugly head?

Typically, user requirements continue to change and our competitors act and react much more quickly than we do. Even with all these arguments and hesitation, there is a chance that members of a software team will eventually be willing to adopt a new development methodology. But as software projects rarely go wrong at the beginning, it can take a significant investment of time and money before we realize that the old way isn’t working.

Meeting deadlines is often a pressure to make us change our old way of working. Let us look at a real case here. In 1995, TechTrans, a Hong Kong

software house with a technical staff of around 20 that specialized in the development of retail-chain points-of-sales (POS) systems written in C and Clipper, won a software outsourcing contract to redevelop an AS/400 application on a truly client/server platform. The system had to be written in PowerBuilder and Informix. At that time, no TechTrans programmer knew these tools. TechTrans could have used its existing Clipper database model for the Informix relational database. However, PowerBuilder is an event-driven programming tool under Windows 3.0, while Clipper is a programming language used to create business applications under the disk operating system (DOS). The project leader asked two developers to pair up to explore how to start their programming. The pair was expected to develop a set of code patterns that the other developers would try to follow. The project was managed using the waterfall model, and both the leader and the team firmly believed that this would be an effective, efficient, and less risky way forward.

1.2 DO-RE-MI DO-RE-MI

Experience keeps people growing professionally. The customers today are different from yesterday's customers, and so are members of a software team. For this, one can only expect software projects, and how they should be managed, to also keep changing. When projects cannot be effectively managed using the simple and familiar waterfall model, an iterative approach is used. This can revolutionize the way a software team develops software, but even though resistance to new ways of doing things can be expected, the resistance may be small as there is a familiar simplicity here.

"When you read, you begin with A-B-C" and "when you sing, you begin with do-re-mi."³ A good place to begin iterative software development is with the waterfall model's requirements analysis (R) – design (D) – coding (C) – testing (T). The simplest way to perform iteration is to simply join two smaller waterfalls together as

$$R \rightarrow D \rightarrow C \rightarrow T \rightarrow R \rightarrow D \rightarrow C \rightarrow T$$

One benefit of iterative software development is that it can be adopted flexibly when coping with the inevitable changes that arise from customer feedback, communications, and working software. Because of changes and the issues discovered earlier, we have more realistic views to control projects to ensure that they are within scope, budget, and timeframe. Another benefit is that it breaks a protracted system analysis into more phases, and thereby actual

³From Rodgers and Hammerstein's *The Sound of Music* in 1965.

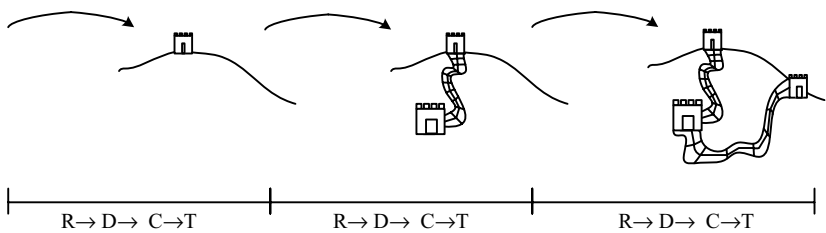


FIGURE 1.4 Phase-by-phase development.

programming can start earlier. This is real progress for software delivery as design diagrams do not cover the details of how to code.

There are at least three ways to implement a simple iterative waterfall model. Most straightforwardly, a system is logically broken down into two or three modules, each of which could be consecutively released for production. It is also possible to implement one or two modules and withdraw the rest. The development approach is referenced as step-by-step or phase-by-phase. A metaphoric example of this approach is given in Figure 1.4.

The second way to implement iterative waterfall model is to review the system nature and functions and to define a kernel and its interface at the beginning (Figure 1.5). The goal of such an iterative cycle is to build new components that could be integrated with a kernel. Different software applications are assembled using the components, which are blackbox to the outside world, but are accessible via their defined interface. Components themselves can be written in several different programming languages as long as they are in full compliance with the interface specifications. This approach to implementing the iterative waterfall model is particularly useful when a number of different applications, each sharing the same reusable login components, are to be developed. Although this can appear to be ambitious, it is a very traditional computing approach. An example is for one to think of an operating system (OS) as a kernel and each application running on the OS, developed with the use of application programming interfaces (APIs), as components so that the computer running the application can serve as a dedicated point-of-sales (POS) or an application server.

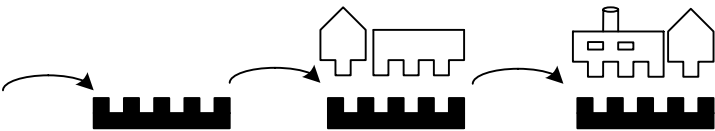


FIGURE 1.5 Component-based incremental development.

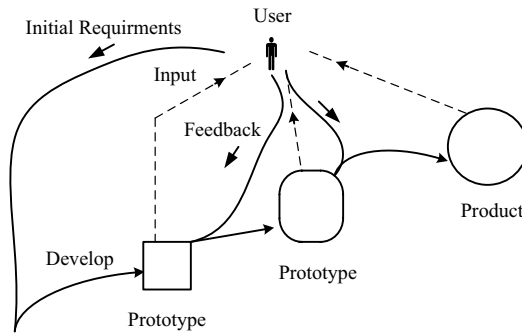


FIGURE 1.6 Evolutionary software development.

Brooks (1995) captures the spirit of evolutionary software development very well by saying “Grow, don’t build software.” This third way of implementing iterative software development is iterative, generative, and incremental (see Figure 1.6). With this approach, a small, immature prototype evolves through constant or regular customer reviews until the software has all the functionalities required. Customers are encouraged to reengineer their requirements so that the final product fits their business needs. With this approach, an early prototype may not even be software. It could be a paper prototype including a set of screen layouts showing the required functionality. However, the prototype must be sufficiently complete for customers to provide solid feedback.

All these different ways of implementing the iterative waterfall model can be adopted in the same project. They can be integrated to different extents into a hybrid iterative model. One way to do this is to have an outer loop taking a step-by-step approach so that each outer loop has several inner loops that can take, say, the evolutionary approach. Such a double-loop iterative model has been proposed and used with some successes as part of some agile methods such as the scrum (i.e., scrummage meeting, as in rugby) .

1.2.1 Iterative Models

As early as the 1950s, Deming popularized Shewhart’s closed-loop model in statistical process control for business continual process improvement, to measure and identify sources of variations so that one can identify and manage the areas where improvement is needed. The feedback loop included in so many project management texts has been generally known as the “plan–do–check–act” (PDCA) cycle. The PDCA cycle is shown in Figure 1.7, which is self-explanatory. The PDCA cycle involves a solid grounding in identifying

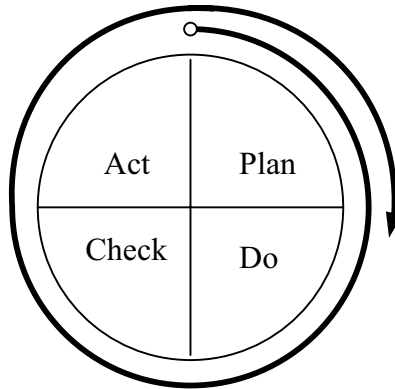


FIGURE 1.7 The PDCA cycle.

performance metrics and measuring them for analysis. The underlying principles have become the foundation of software project management.

Returning to a basic iteration like $R \rightarrow D \rightarrow C \rightarrow T \rightarrow R \rightarrow D \rightarrow C \rightarrow T$, we can see that the iteration does not tell us how to sustain actions! For this reason, a review session is normally needed after each cycle to determine whether we have done as planned so that we can realistically plan what the next cycle should be. In addition to this, we also need to reevaluate the different risk factors that may affect a project so as to ensure that we can better control budgets, resources, and schedules against the original project plan. Clearly, some supporting process areas should be considered to sustain the iteration of $R \rightarrow D \rightarrow C \rightarrow T \rightarrow R \rightarrow D \rightarrow C \rightarrow T$ so that each iteration delivers solid progress toward the final product until an application is released for production.

The PDCA and waterfall model activities, can be combined to establish a complete iterative model — the spiral model — as proposed by Barry Boehm (1988). This spiral model can be modified as shown in Figure 1.8. The sequences of $R \rightarrow D \rightarrow C \rightarrow T \rightarrow R \rightarrow D \rightarrow C \rightarrow T$ can therefore become, say, $R \rightarrow R \rightarrow D \rightarrow R \rightarrow D \rightarrow C \rightarrow T$ (see Figure 1.8). As expected, processes and practices are required to sustain such a model. It should be noted that this modified spiral model does not contradict the iterative waterfall model of $R \rightarrow D \rightarrow C \rightarrow T \rightarrow R \rightarrow D \rightarrow C \rightarrow T$ and software teams can choose to substitute it with the modified spiral model.

The spiral model was originally proposed to develop different prototypes at various stages of a project until the final product is completed. The use of such model is both generative and evolutionary. In practice, software teams may adopt a spiral model according to project requirements. The implementation of the iterative waterfall model can be flexible, and the three different approaches to implementing such a model can be integrated and hybridized.

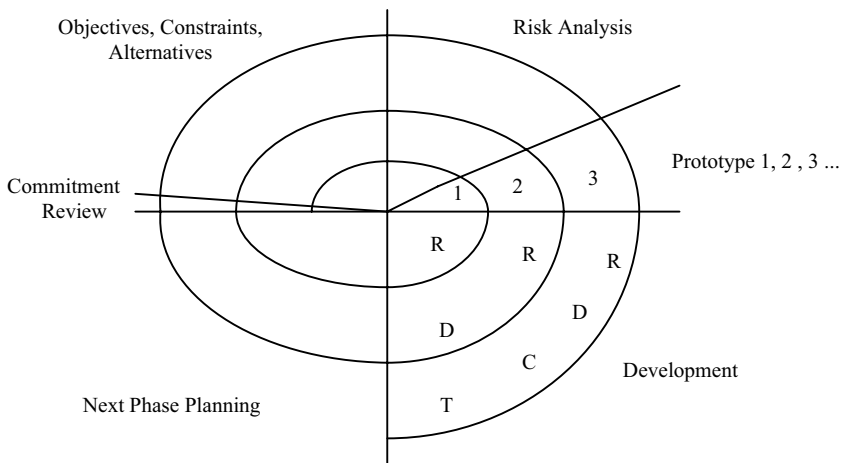


FIGURE 1.8 The spiral model (simplified version).

With these characteristics, the spiral model, which applies the ideas of the PDCA and a combination of these three implementation approaches, can be used rather flexibly with different software projects and thus has been generally accepted as much better than the waterfall model.

1.2.2 Code and Fix

Even though iterative software development approaches have their advantages, not all iterative approaches are desirable. The code-and-fix approach, for example, is repetitive. It involves writing code to clarify requirements for better design later. It is a time-buying strategy where the target is to release the software on schedule and to release patches afterward. It is common in software development that project pressure quashes discipline and that when software developers are under time constraints, they will naturally handle this pressure by jumping into coding immediately. Another situation in which developers would adopt a code-and-fix approach is when the application being developed is so popular that it attracts new, additional, originally unintended users who demand additional expansions in performance and functionalities.

Let us consider a real case as follows. In a retail chain of 45 outlets in a metropolitan area, operational staff might need to split their time between staying in office and visiting other stores. The human resources (HR) department therefore decides to have their information technology (IT)

department write a system for them to allow staff to submit trip records electronically. Their goal was to replace manual forms with a database so that the HR department could quickly retrieve information relating to these business trips. The written requirements provided by HR were a brief sample copy of their current form!

The HR system, called *TripLog*, was written in under a fortnight in Microsoft Access using Delphi 6.0. Since the functionalities of TripLog were rather simple, so the HR staff were quick with their user acceptance testing. As expected, HR occasionally reported minor bugs, but these were quickly fixed.

After 2 months, the HR staff asked the IT team to distribute TripLog to user departments so that they could directly enter data into the system. After an additional 2-month period, the HR department decided to add vacation leave as a type of a “trip” in the TripLog so as to automate leave applications. Now that the number of users had unexpectedly increased, the system became extremely slow. Naturally, users start to request that the IT department to improve system performance and to have TripLog display leave balances.

The IT department decided to rewrite the system in MS SQL Server using Delphi 6.0. This took a month, but this was not the end of the story yet as TripLog continued to be the subject of modifications and eventually its user base included all staff of 150 back offices.

The development of TripLog was not disciplined, but the system was not complicated and the software team managed to do a good job. However, the software team actually redeveloped the system completely. The code-and-fix cycle in this case resembles the following sequence:

$$\text{Code} \rightarrow \text{use} \rightarrow \text{fix}(\rightarrow \text{code}) \rightarrow \text{use} \rightarrow \text{fix}(\rightarrow \text{code}) \rightarrow \text{use} \rightarrow \text{fix} \dots$$

The activity shown in parentheses may occasionally be required.

The code-and-fix approach is different from the iterative model in that we could not tell when a development activity would occur and when one activity would switch to another. Although there was no sense of rhythm and events appeared to occur randomly, the pattern was iterative. This approach can be considered by some developers to be ad hoc.

1.2.3 Chaos

Timing and patterns are important in any kind of iterative model. There can be huge differences of days and even months in completion dates when the same iterative software activities are followed. In this section, we will see what an iterative model may look like when a cycle is as short as a day.

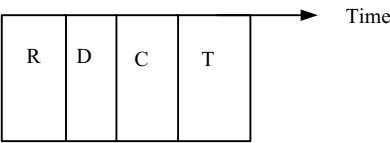


FIGURE 1.9 Waterfall topology.

Figure 1.9 shows a four-stage waterfall model over time. If we assume that there is a deadline to meet for each stage, the project can be tracked with four separate milestones. According to Parkinson’s law, work expands to fill the time available for its completion. Therefore, it is rare for a software team to complete its tasks on time. Assuming the probability of delay in project schedule be $\frac{1}{2}$, for four stages, we can be very pessimistic about the chance of completing the project on time as $\frac{1}{2} \times \frac{1}{2} \times \frac{1}{2} \times \frac{1}{2} = \frac{1}{16}$. In other words, there is very little chance that a project is able to finish on time. Of course, many project managers would squeeze time from later stages to compensate for earlier delays, but this effectively shortens the time available for the tasks that are to be achieved in later stages. This may lead to sacrificing either quality, functionalities, or both.

To cope with this problem, we can implement an incentive scheme. When developers are able to complete jobs on time (see the “time box” in Figure 1.10), they receive a bonus pay. To implement this effectively, we need to assign different roles to different members of a software team at each stage. It is possible for us to assign different roles to the same developer. For instance, we can assign requirements engineers the role of software testing to test the final

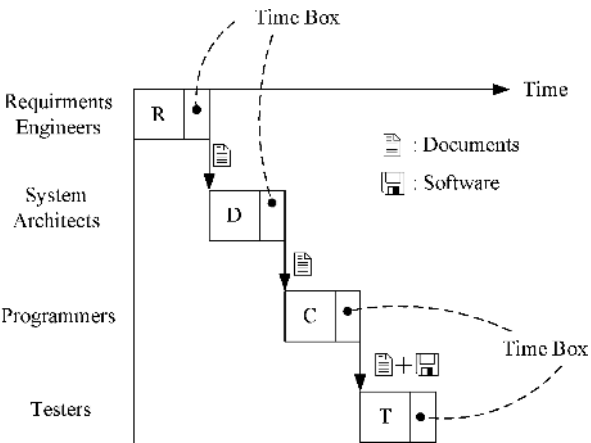


FIGURE 1.10 Waterfall in action.

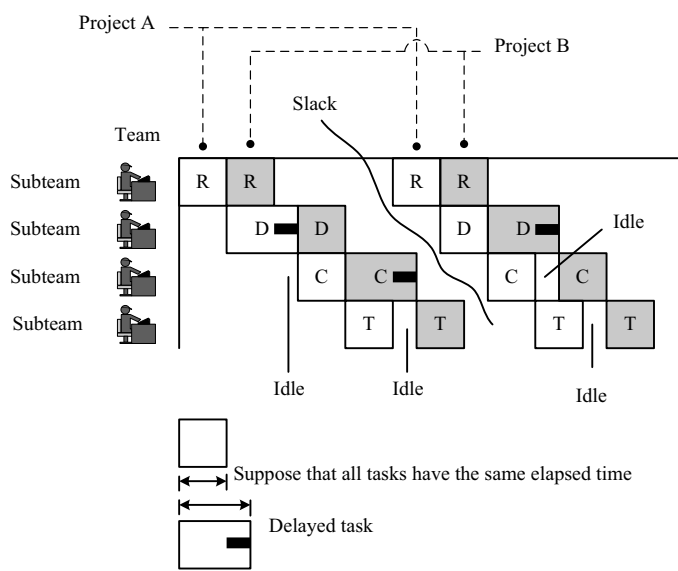


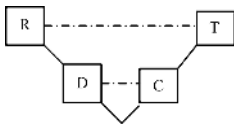
FIGURE 1.11 Delayed tasks and idle developers.

product and the role of coder to system analysts to enable them to write programs for verification.⁴

In addition to an incentive scheme and the assignment of different roles, another question arises as to how people in a team communicate and whether such communication is effective. For example, there is a need for well-written documents to be used as a communication tool between two sequential stages. Figure 1.10 illustrates the waterfall in action.

A software team may be involved in several projects at the same time with team members organized as divisions. Figure 1.11 illustrates how two projects can be run by the same team in parallel. Basically, each subteam either handles just one project at a time or the members deal with one project's tasks at a time. However, when documents passed down from a previous

⁴The idea of how quality control checkpoint can be integrated into each phase throughout the development as implemented in the V model. As in our simple example, the look resembles a “V” as shown here:



stage are difficult to understand, incorrect, or incomplete, the responsible subteam has to follow up. Thus, some subteam members will find themselves handling the tasks of two projects at the same time. Although this is quite typical in the real world, this kind of task switching adds no value at all to software development (Poppendieck and Poppendieck 2003). Human concentration is easy to break and hard to get back. Switching tasks between two projects eats up time (say, 10–15 minutes) as people reenter the flow of thought for a new task. Frequent interruptions are time-wasting.

Figure 1.11 illustrates another issue that is perhaps even more disturbing than task switching. Most staged models require the completion of one stage before it is possible to enter the next. This makes it difficult to plan two projects to avoid any slack-time between them! Compounding this is the fact that delaying some activities in one project will *tremendously* affect another project. Figure 1.11 illustrates how “delayed time” and “idle time” intertwine even though there is no real idle developer as the project plan can be revised as often as necessary.

To tackle these problems with the simultaneous management of multiple projects, we are brought to the arena of concurrent engineering. We do not wait for the completion of one task before the other starts (see Figure 1.12), and we allow different development processes to run in parallel. To allow a process to evolve more flexibly, we should not be confined only to documents. Instead, we hold more face-to-face meetings to facilitate proper communications. To manage single projects, we can also adopt concurrent software engineering (Figure 1.13).

Concurrent software engineering can be adopted by applying a model for managing single as well as multiple projects (Figure 1.13). The greatest benefit of such a model, called the *Sashimi model* (Raccoon 1995), is that it shortens the iteration cycle.

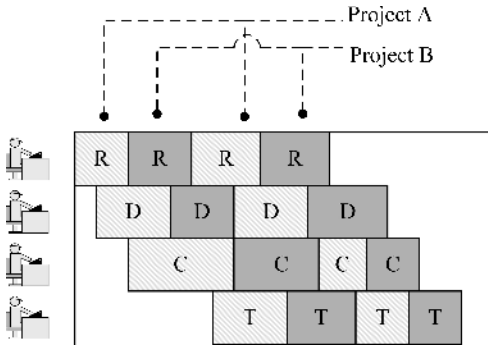


FIGURE 1.12 Concurrent engineering.

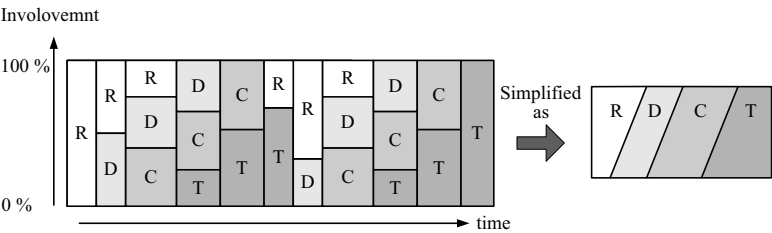


FIGURE 1.13 Sashimi model.

One way to expedite tasks is to shorten iteration cycles. Iteration cycles can be shortened by allowing many things to happen simultaneously. For example, the chaos model (Figure 1.14) looks at a team’s activities as a whole, fractally. The model uses a short, small problem-solving loop, but unlike the case with code-and-fix, the chaos model can be very rhythmic as far as we anticipate when things work, when things can be used (i.e., how one loop turns to another), how to sustain the rhythm, and so on.

1.2.4 Methodology that Matters

The following statement was made by a finance director in charge of accounting, administration, personnel, IT, and purchasing departments: “My daughter, 15, was already building her home page at school! I just don’t understand what our IT team is busy with.”

Customers can be users within an organization, or they can be the external client of a software house. They may not see our service the way we do. Building and managing customer relationships are as important as developing quality software. Projects with teaming relations with customers could be twice productive (Bernstein and Yuhas 2005). When it comes to what

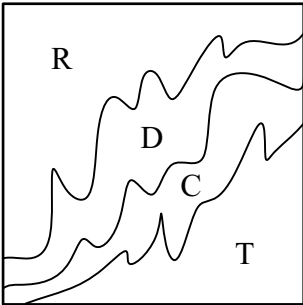


FIGURE 1.14 Chaos model developed by Raccoon (2006).

		Developer	
		Ugly	Attractive
Customer	Ugly	☺	☹
	Attractive	☹	☺

FIGURE 1.15 Customer–developer relationship.

customer-relations management (CRM) is, Ed Thompson of the Gartner Group presents the following matrix (see Figure 1.15). CRM, according to him, is all about how customers see the development team and how the development team sees them. When customers and developers regard each other as mutually ugly or mutually attractive, nothing can or needs to be done (shown as “☺” in Figure 1.15). But when the mutual perception is ugly–attractive, there is room to improve the customer–developer relationship.

From a developer’s perspective, ugly customers are reluctant to participate in the development process [i.e., requirements management process and user acceptance testing (UAT)]. Such customers think that what and how software is built is not their concern. Of course, software that is not targeted to specific business processes or domain knowledge demands less user participation. Other than these kinds of customers, there are also customers who are not concerned with the details of software functionalities. For example, some people use Microsoft Word every day but have no interest in knowing anything more than what they already know even though there are better ways of doing things. Customers with that kind of attitude are not helpful. In fact, such attitudes can be harmful to a software team. From the customer’s perspective, besides return on investment, satisfaction with the product or service, schedules and scheduling, speed of response, ongoing service support, and product quality are major concerns. Much of this is directly related to how a software team builds software, namely, software development methodologies.

Generally, customers do not like jargon or complicated flowchart diagrams. For this reason, the ideas of metaphoric communications or writing short descriptive requirements (see user story in Chapter 3) have been

proposed. In addition, since many users, not trained as programmers, have problems with too many if-then-type requirements, keep them fewer than five.

From time to time, a software team has to give customers progress reports. Some software team will send their customers an enormous, perplexing updated project plan but customers like a product demo (demonstration model) or prototype. Instead of a report, it is therefore easier and more effective if an update of the latest progress status is reported by releasing a demonstration of working software. Owing to their organizational culture and operational processes, some customers accept only big-bang implementation, but it is still a good idea to release working software for a demo, or for early training.

In the commercial world, customer requirements are often dynamic. There is often much need for effective exceptional handling ability in an organization. Customers appreciate quick response from developers. How fast a software team can change a software to meet new business requirements depends on a number of factors. If one is to ignore technical issues and look only at development procedures, one may conclude that the implementation of bureaucratic document control and awkward team structure may make us change our work more slowly than we should.

Modifications can be made with these concerns in place and, after modification, it is necessary to retest the software. Moreover, after modification, retesting what software has worked is necessary. Testing and retesting are two basic concepts in software testing. Generally speaking, the purpose of testing is to detect faults in executed code that causes a failure, while retesting is done to confirm that the changes do not introduce error to other parts of the code (this is also known as *regression testing*). Retesting is often boring and tedious. Automating all or some part of the testing could be an answer to some of the problems mentioned here. Coding and testing need to be better integrated here for a more complete solution. Developers need to be able to write automated testing cases while coding.

When some developers leave a team, others will have to take over, and developers who take over will have to spend some time to understand and make changes to a piece of code. To do so, they may have to refer to technical documents or to read the code directly. This takes time as documents may not be written in such a way as to make them easy for other developers to follow. Ideally, developers should rotate jobs among themselves so that each piece of code can be maintained by more than one person. This, however, may not always be feasible.

All these development problems can arise at the same time, making it difficult to respond quickly to changing requirements. Changing our software

development practices overnight will not lead to successes. We have to take small steps first. The iterative model discussed so far may meet this requirement as it allows us to manage processes, schedules, budgets, and risks. However, it is not complete. There is still something missing. We need to harmonize *practices*, *people*, and *software*, and this leads us to so-called software development rhythms, inspired by Kent Beck who says, in his XP book (Beck and Andres 2005), that rhythms operate at all different scales. A principle such as do-check can be applied to the process of doing before the process of checking or practice for doing before practice for checking. They are still do-check!

1.3 SOFTWARE DEVELOPMENT RHYTHMS

If you drop a frog into a pan of hot water, the frog will leap straight out. But if you put the frog into a pan of cold water and slowly heat it, the frog will sit there until it is cooked, unaware of the gradual changes in temperature. Well, maybe frogs are that dumb and maybe not, but there is an interesting point here. No one likes sudden, unexpected changes and, ironically enough, that includes techies such as software developers.

Suppose that a consultant is hired to coach a software team in a development methodology that has a number of new software practices. He asks the team to start with two or three practices and to gradually exercise others. Alternatively, he suggests that the team take a maturity approach in which the team advances toward software practices suggested by the methodology. For instance, an onsite customer requires at least one customer representative to be available onsite all the time. We begin with the representatives visiting the developers frequently enough to sustain personal contacts, then being available not less than 2 hours per day and eventually being an onsite customer (Nawrocki et al. 2003). Both approaches are widely used. The secret of making this successful lies in whether we have successfully complemented either way with the right rhythms of a development methodology while it is introduced!

For instance, Beck, in his book on eXtreme programming, suggests a “standup meeting” to start a day and software integration before calling it a day. Participation of all team members in these meetings is necessary. Developers have to be punctual; otherwise, time is lost in waiting. Not all people in every software team can get it done as easily as we thought. In some extreme cases, people who are not used to the time rhythm dislike the idea of morning meetings. A better way is to organize an informal morning coffee meeting for the whole team and to have a day-end gathering to orally confirm who could not join the coffee meeting the following day. We can see the

rhythm as morning gathering–work–day-end gathering. Once the team gets used to that rhythm, we may easily change to “standup meeting–work–code integration before go home.”

Often a development framework can have many such rhythms playing simultaneously. In this case a software team should better get used to a thematic rhythm that actually drives the success of the framework. For example, in the iterative waterfall, the thematic rhythm can be design–programming–design–programming.⁵ This thematic rhythm must be sustained; otherwise, the paradigm could be more harmful than helpful.

To sustain any rhythm, such as A–B–A–B, requires both strategy and execution. Determining what practices between A and B are selected and how they can be harmoniously combined to establish effective development strategies is the same as exploring when software practices work and when they can be used. Adopting the right strategy is only half of the story; we need execution, and we especially need to be able to sustain a rhythm. This issue will be revisited in Section 1.3.2.

1.3.1 Stave Chart by Example⁶

“Most people live within a wall of rationality that is defined by the real and the apparent limits of the world they inhabit” (Anderson 1993). Software professionals, by occupation, have been trained for so many years that they are mentally fixed to think in certain patterns. For instance, they often carry a preference of conditional logic when seeing diagrams looking like flowcharts. This is sort of reflection, and hence we are often being limited by what we see (Figure 1.16).

Development rhythm can be expressed in flowcharts as illustrated in Figure 1.16. However, the use of flowcharts may cause us to lose the ability to sustain, harmonize, and, most importantly, synergize. For simplicity and readability, we use stave charts to represent software development rhythms. We believe that the stave chart gives us a stronger sense of exploring deep harmony by putting two or more software practices in harmony. (For instance, in the sequence A–B–A–B depicted in the four different scenarios in Figure 1.16, when practices A and B are harmonized to produce synergies,

⁵For those who have known eXtreme programming (XP), another example is that the thematic rhythm of XP is test–code–refactor; see Chapter 9.

⁶The authors would like to thank Dr. Michael E. McClellan from the Department of Music at The Chinese University of Hong Kong for technical comments on the “stave chart by example.”

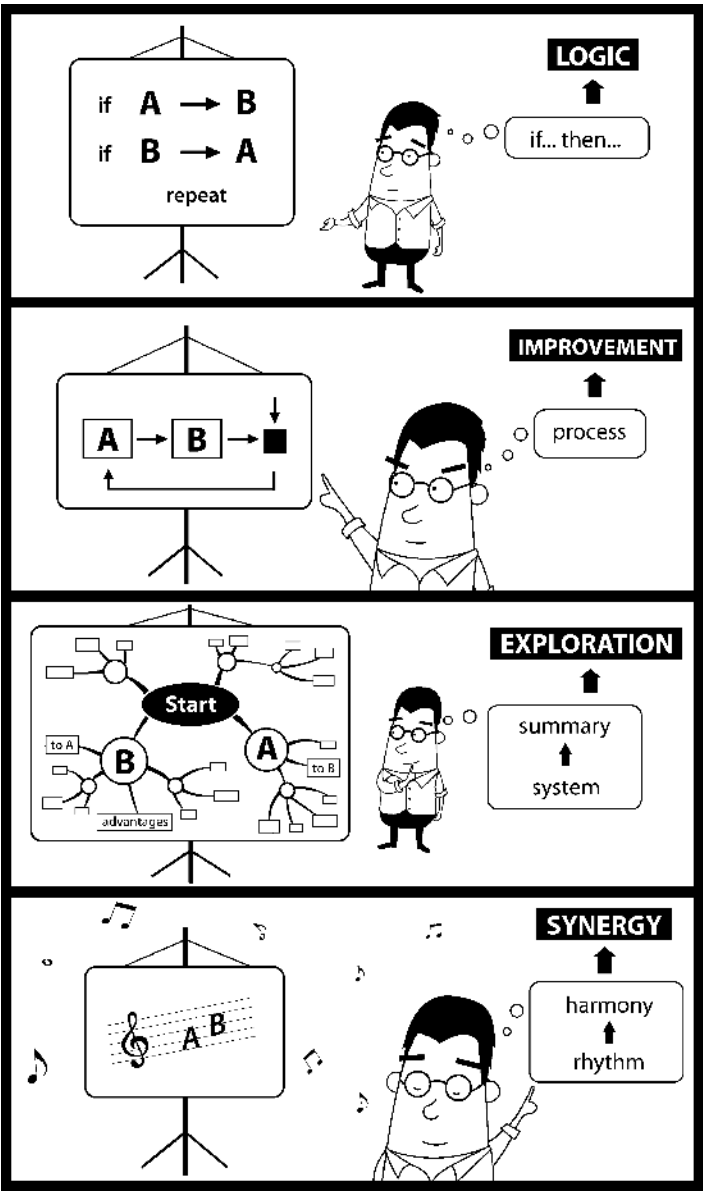


FIGURE 1.16 Different visual representations of the same thing affects our thinking.

the stave chart is a good choice.) The main purpose, however, is to help us think of software practices in rhythms.

Let us look at a simple rhythm of software practices such as A–B–A–B–A–B. The rhythm starting with A is denoted as $\text{♩} \text{ } \text{♩}$. Then we have $\text{♩} \text{ } \text{♩} \text{ } \text{♩} \text{ } \text{♩} \text{ } \text{♩} \text{ } \text{♩}$.



FIGURE 1.17 ☹ Explained by code–fix.

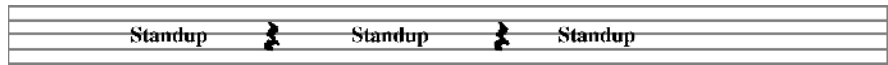


FIGURE 1.18 ☹ Explained by standup meetings.

Since it is repeatedly moving between A and B, we use the symbol $\| \cdot \|$ to show repetition. As with many rhythms, we are often concerned about which practice should come at the end. The rhythm will be $\| \cdot \|$. Where we don’t care about the order of starting and ending, a practice can be expressed as $\| \cdot \|$.⁷

Now let us look at another example of code–use–fix–(code)–use–fix–(code)–use–fix. Here “(code)” has an unplanned duration. It can even be skipped. The notation ☹ is placed over it to mean unplanned or uncertain practices. The rhythm tells less about when that practice happens and how long it may last. Figure 1.17 illustrates the rhythm of code–fix using a stave chart.

In some cases, we would emphasize pause and interruption. Sometimes, doing and holding onto something for a bit longer will unavoidably incur a stop or interruption. For example, when trying to have a standup meeting for an hour, team members will naturally ask for a regular break after 15 minutes. The symbol ☹ indicates an interruption.

If we do not place ☹ in Figure 1.18, then we would just write one “standup” instead of three. In this case, we deemphasize any interruption during a standup meeting.

The pause or interruption of an unknown duration can be really problematic. It is not wise to have a long standup meeting in the morning to discuss every project and technical issue that arose yesterday until all issues are resolved. Long meetings fragment, as shown in Figure 1.19. People may ask for breaks to return calls and do not return to the meeting on time. In addition,

⁷This is a minor point, but technically, if something is enclosed in repeat signs, it will be repeated only once and everything within it should be repeated. So, for this example, the result would be A–B–A–B, nothing more and nothing less. There should be an indication that the section enclosed in the repeat signs should be repeated more times or an indefinite number of times.

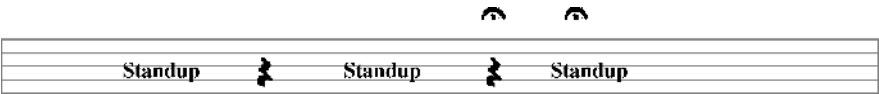
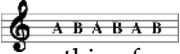







FIGURE 1.19 Long standup meetings.

urgent matters that should be handled soon may come up in the meeting and the meeting may have to be suspended.

Now we talk about the structure of a rhythm. Consider a rhythm such as . We can interpret this to mean that activity A is done to deliver something for activity B. Another meaning is that A and B are linked by time.

Another scenario is a bit more complicated. Some objectives of activities within A are to turn B. Normally, outputs of A are the input for B. But it is possible that A itself is much more important than its outputs and contributes much for B. The slur mark here  indicates a special condition in which A itself triggers B.

Some rhythms have a faster tempo (e.g., hourly or daily basis). We use “Vivace” to represent that tempo as . The five lines on which practices or processes (e.g., A and B) appear are written such that, by comparing A with B, the higher it is on the stave chart, the more difficult it is or the more human dynamism is required for a team so that people will pay attention. However, in the real world,  can be  for some, but becomes  for others.

The stave chart is self-explanatory. It is not a detailed workflow diagram. Basically, we have not invented any notation, although one of them has been slightly altered. The musical notation here parallels those between the compositional processes that Bach and Mozart used and the processes that programmers employ. We draw it on the whiteboard to coach software teams in development rhythms.

1.3.2 Game Theory

Companies from one country venturing into another are faced with a thicket of unfamiliar and easily misinterpreted regulations to which they must make their business operations conform.

Dave is a software leader who is going to take over a project to build an insurance application in a developing country. His software team has around 10 experienced colleagues, and they have successfully used the waterfall model to develop similar systems for almost a decade. What Dave needs to do is to lead the team and repeat the earlier success. However, unprecedented challenges are ahead of him.

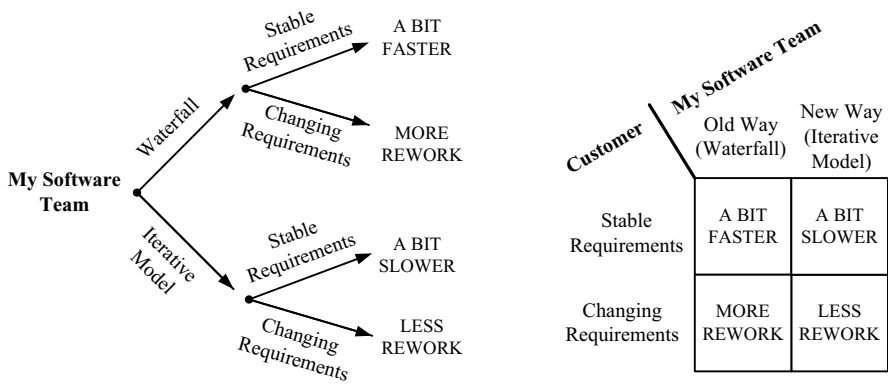


FIGURE 1.20 Software team playing their development game with customer.

The situation is more or less like exploring a game to determine the best strategy by understanding how the customer and the software team interact. When requirements are correctly formulated and relatively stable, Dave’s team can do as well as usual. They feel confident. This way should therefore be considered as the faster or most expedient approach (see Figure 1.20). Unfortunately, the customers could ask for any change in their business requirements during the construction stage. More risks could result from the frequent change requests. In such a case, rework seems unavoidable.

Dave knows that the requirements could be unstable in that environment. A big waterfall model is not desirable. On the basis of his past experience, he works out a simple iterative model to build the system through evolution. There will be three or four releases, and review sessions will be held immediately afterward. During the review, the customers are allowed to raise any questions or comments for modification. Once satisfied with the progress, the customers have to pay the development fee. Rework can be minimized.

Although this sounds great, this could cause problems in Dave’s software team, who may not be familiar with such an iterative approach. Adopting a new software paradigm is a team-level change! As the size of the development team is small and Dave has established a good relationship with the team, he can manage this situation. Nevertheless, the whole development process is definitely longer.

According to the maximum principle in game theory, players prefer to minimize the maximum possible loss. Thus, the project leader will plan for the iterative model because the loss is less.

This game theory analysis is satisfactory only if we can have data to understand the implications in detail. Moreover, change in software development is more than a yes/no issue. To fully analyze the game, the matrix

shown in Figure 1.20 would have to be much more complicated. Still this type of tool is helpful for our strategic thinking about playing rhythms.

1.3.3 In-Out Diagram

When praised for brilliant, fantastic piano playing, Johana Sebastian Bach humbly said, "There's really nothing remarkable about it. All you have to do is to hit the right key at the right time and the instrument plays itself." To play the piano well, we know how good a start is when attempting to tackle a piece of work that can have tremendous psychological impact on us. A good start motivates players to keep their focus and continue to strive for better results. A good beginning is work half-done. How a rhythm can be sustained is another key factor to be considered for a music player. One wrong key could break the melody immediately and could ruin all previous efforts!

Every rhythm can be represented as A-B-A-B regardless of what A and B are. They could be $R \rightarrow D \rightarrow C \rightarrow T \rightarrow R \rightarrow D \rightarrow C \rightarrow T$ or $code \rightarrow use \rightarrow fix \rightarrow code \rightarrow use \rightarrow fix$ or anything at all. Each rhythm is uniquely different from the others. Some rhythms are easy to start but require a lot of effort to sustain, and external factors can also affect them negatively. Some rhythms, however, once we are used to them, are easy to sustain.

Sustainability is a key issue in software development rhythms. So often, a development rhythm no longer delivers expected values to both the team and the software but continues to be used. We use the in-out diagram shown in Figure 1.21 to represent rhythms as easy or difficult to start and to sustain. The in-out diagram provides a tool for strategic thinking. It is crucial to software development rhythms and is used throughout this book.

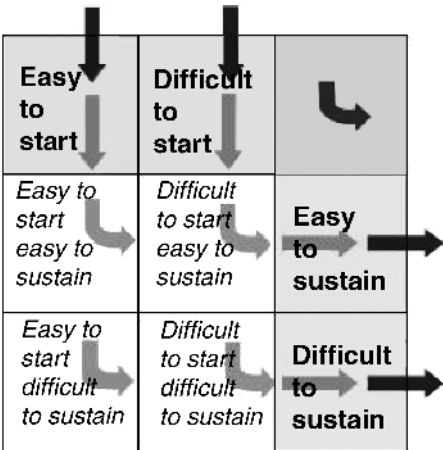


FIGURE 1.21 In-out diagram.

Easy-to-start	Difficult-to-start	
Interactive Waterfall		Easy-to-sustain
Traditional Waterfall		Difficult-to-sustain

FIGURE 1.22 Dave’s decision.

Both the conventional and the iterative waterfall models are easy to start with but difficult to sustain. Any changes can require substantial rework on software and its documentation and can also break the rhythm, at least temporarily. Changing requirements are less vulnerable to the iterative waterfall than to the conventional waterfall. Thus Dave’s decision can be depicted in the in–out diagram in Figure 1.22.

In this book, we discuss the in–out development rhythm diagram on the basis of our experience. When putting this concept into action, it should be noted that all teams are different and all participants should reevaluate the diagram for their own team.

1.3.4 Master–Coach Diagram

The in–out diagram alone does not tell us what will happen to a software team when developers change with old hands leaving and new blood coming in. For better planning, there is a need to know that a worker has gone and taken his or her project knowledge and development experience.

Knowledge itself is an evergreen topic in philosophy. Ontologically, knowledge can be explicit and/or tacit. Explicit knowledge can be simply recorded in text, symbols, and/or diagrams. It can be articulated. Tacit knowledge is individual’s actions, experience, values, enjoyment, rapport, or passion, or the emotions that they embrace. It is human knowledge. A software team putting rhythms into action has to work with tacit knowledge.

It takes time to learn and master new rhythms of practices in a unique development environment. Even a single practice such as two people collaborating in programming appears so simple, yet both people require hours or days to learn how to communicate well with each other. One dimension to

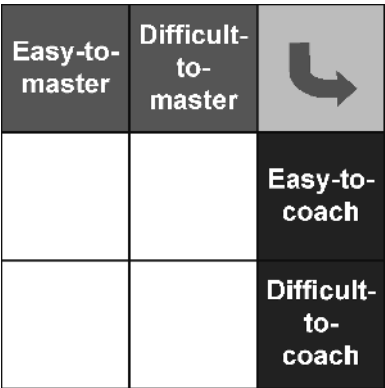


FIGURE 1.23 Master-coach diagram.

consider when adopting development rhythms is whether they are easy or difficult to master.

Newly hired developers may join a team during the development stage. Newcomers may find some rhythms easier to learn through on-the-job training alongside those who have already had experience with them or through previous project documents or even by absorbing the development atmosphere and culture. Newcomers start as apprentices to master craftsmen. However a skill is acquired, the ease or difficulty of acquiring a rhythm adds another dimension. Bringing team learning and newly hired programmer training together, we have the master-coach diagram shown in Figure 1.23. In formal terms, the diagram reflects knowledge transfers between those who have mastered the rhythm and those who have not.

1.3.5 No Mathematics

We do not need to perceive things through the use of mathematics. For instance, we can turn a burner to high and heat up a water-filled pot. The pot warms up and large bubbles rise to the surface. Eventually the pot boils dry. We have learned the principles of this phenomenon from our own experience. We do not need equations.

There is no mathematics in rhythms. Depending on how software practices are played as a rhythm, their synergy cannot be clear through understanding each of them individually. An example is shown in Figure 1.24. *Pair programming*, which is done by a team of two programmers who always collaborate on the same program together, is easy to start but difficult to sustain because the team has only two programmers, there is no partner exchange with

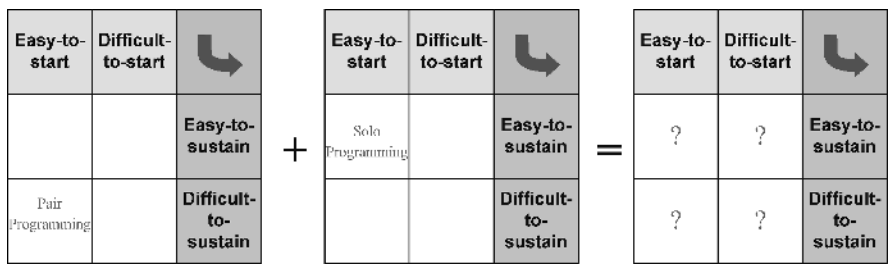


FIGURE 1.24 Rhythms have no mathematics.

other pairs. If they work on the same task for long, it may not be easy for them to always maintain their concentration. *Solo programming* is easy to start and easy to sustain. It is hard to conceive of the in-out diagram of two rhythms as one just by understanding these two individual rhythms. We have to look into how a rhythm is established.

1.3.6 Where to Explore Rhythms

Iterations, patterns, and rhythms are interrelated. *Rhythm* refers to harmonized processes and practices in the sense that each element should be used at appropriate times so as to deliver synergistic values to people and software. Software development rhythms are also relevant when it comes to the use of different development strategies and how and when they should be executed. Both the in-out and master-coach diagrams can guide such analysis.

It is possible for one to identify many rhythms in good software development. Some are easy to start but difficult to sustain, while others are difficult to start but easy to sustain. In this book, we are interested only in those that are both easy to start and easy to sustain. There is no single one rhythm that applies to all kinds of software development. Identifying rhythms is a matter of observation and experience, and it may even involve many trials and errors. We have to try different rhythms out in our teams in practical situations. Agile practices are generally amenable to this kind of approach, and for this reason, rhythms of agile practices one of the main themes in this book.

Good software rhythms are required to ensure that a software team is productive and the software projects are completed successfully. For this, we need to know how to meet our new software teams, and how to recruit new software developers for our team. A software team has its own norms, and it is difficult for one to talk about a general template that can be adopted to achieve the same results with a different team. For this reason, instead of discussing some standard practices, we present some case studies. We will discuss software teams in developing countries to emphasize the importance of

cultural elements in exploring the right rhythms. To make software team productive, a team must be aware that not all the knowledge gained from their software project experience may be helpful or useful. We return to this issue in Chapter 2.

It is tremendously challenging to tell a software team that they need to change their usual practice and adopt something better. As there are so many ways to build a piece of software, it is possible for some people to prefer one method and others to prefer a completely different method. To address such conflict in typical modern-day software development in Chapter 3 we discuss open-source software development that is almost diametrically opposite to the methods used to develop software in the commercial world. Almost every programmer believes that there is something to be learned from open-source software development, and in Chapter 3 we describe some of our experience with using agile software development processes.

Chapters 2 and 3 establish some basics and cover a very broad spectrum of topics in contemporary software engineering, touching on the essentials of programmers, social culture, project experience, team communications, software processes, and practices. The second part of the book makes use of proven techniques and applications in engineering management, sociology, industrial psychology, and group dynamics.

We explain software development rhythms in varying depths throughout the other chapters in Part II and discuss several software development rhythms. Many software rhythms are closely related to eXtreme programming (XP), and this is not just a coincidence. While many software teams have successfully adopted those XP practices, some teams are crying out loud to get out of it. I have often heard complaints such as “Kim and Keith, we already tried the agile practices before, but they did not work here!” We think that they get the software development rhythms wrong or they only get their old development rhythms right. Trust me! To succeed with any software paradigms, the mindsets and ways of working have to catch one critical element right: software development rhythms.

We hope that this book will help you become more aware of the rhythms of software development and see how they can contribute to the quality of both processes and products in your own firsthand experience of writing software.

REFERENCES

- Anderson JV. Mind mapping: A tool for creative thinking. *Business Horizon* 1993; **36** (1):41–46.
- Beck K and Andres C. *Extreme Programming Explained*. 2nd ed. Boston: Addison-Wesley; 2005.

- Bernstein L and Yuhas CM. *Trustworthy Systems through Quantitative Software Engineering*. Hoboken, NJ: Wiley; 2005.
- Boehm B. A spiral model of software development and enhancement. *IEEE Computer* 1988; **21** (5):61–72.
- Brooks FP. *The Mythical Man-Month: Essays on Software Engineering*. Reading, MA: Addison-Wesley; 1995.
- Chapman SN. *The Fundamentals of Production Planning and Control*. Upper Saddle River, NJ: Pearson/Prentice-Hall; 2006.
- Leveson N. An investigation of the Therac-25 accidents. *IEEE Computer* 1993; **26** (7): 18–41.
- Martin RC. *UML for Java Programmers*. Upper Saddle River, NJ: Prentice-Hall; 2003.
- Nawrocki J, Walter B, and Wojciechowski A. Toward maturity model for extreme programming. *Proceedings of 27th Euromicro Conference*, 2001, p. 233–239.
- Poppendieck M and Poppendieck T. *Lean Software Development: An Agile Toolkit*. Boston: Addison-Wesley; 2003.
- Post TJ, Baltussen G, and Van den Assem M. Deal or no deal? Decision making under risk in a large-payoff game show. *EFA 2006 Zurich Meetings* 2006; available at SSRN: <http://ssrn.com/abstract=636508>.
- Raccoon LBS. The chaos model and the chaos life cycle. *ACM Software Engineering Notes* 1995; **20** (1):55–66.
- Royce W. Successful software management style: Steering and balance. *IEEE Software* 2005; **22** (5):40–47.
- Royce W. Managing the development of large software systems. *Proceedings of IEEE WESCON*, Aug (1970) p.1–9.
- Tversky A and Kahneman D. The framing of decisions and the psychology of choice. *Science* 1981; **221** (4481):453–458.

