SOFTWARE DEVELOPMENT RHYTHMS

Harmonizing Agile Practices for Synergy

Kim Man Lui and Keith C. C. Chan

The Hong Kong Polytechnic University, Hong Kong
SOFTWARE DEVELOPMENT
RHYTHMS

Harmonizing Agile Practices for Synergy

Kim Man Lui and Keith C. C. Chan
The Hong Kong Polytechnic University, Hong Kong

A JOHN WILEY & SONS, INC., PUBLICATION
To my mother and my sister  
— K.M.L

To my parents and sisters  
and to Emily, Samantha, and Jeremy  
— K.C.C.C.
# CONTENTS

**PREFACE**

<table>
<thead>
<tr>
<th>PART I: ESSENTIALS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1 NO PROGRAMMER DIES</td>
<td>3</td>
</tr>
<tr>
<td>1.1 Developing Software versus Building a Tunnel</td>
<td>4</td>
</tr>
<tr>
<td>1.1.1 The Good Old Days?</td>
<td>5</td>
</tr>
<tr>
<td>1.1.2 The More Things Change, the More They Stay the Same?</td>
<td>6</td>
</tr>
<tr>
<td>1.1.3 Behind Software Products</td>
<td>7</td>
</tr>
<tr>
<td>1.1.4 Deal or No Deal</td>
<td>10</td>
</tr>
<tr>
<td>1.2 Do-Re-Mi Do-Re-Mi</td>
<td>12</td>
</tr>
<tr>
<td>1.2.1 Iterative Models</td>
<td>14</td>
</tr>
<tr>
<td>1.2.2 Code and Fix</td>
<td>16</td>
</tr>
<tr>
<td>1.2.3 Chaos</td>
<td>17</td>
</tr>
<tr>
<td>1.2.4 Methodology that Matters</td>
<td>21</td>
</tr>
<tr>
<td>1.3 Software Development Rhythms</td>
<td>24</td>
</tr>
<tr>
<td>1.3.1 Stave Chart by Example</td>
<td>25</td>
</tr>
<tr>
<td>1.3.2 Game Theory</td>
<td>28</td>
</tr>
<tr>
<td>1.3.3 In–Out Diagram</td>
<td>30</td>
</tr>
<tr>
<td>1.3.4 Master–Coach Diagram</td>
<td>31</td>
</tr>
<tr>
<td>1.3.5 No Mathematics</td>
<td>32</td>
</tr>
<tr>
<td>1.3.6 Where to Explore Rhythms</td>
<td>33</td>
</tr>
<tr>
<td>References</td>
<td>34</td>
</tr>
</tbody>
</table>

**2 UNDERSTANDING PROGRAMMERS**

| 2.1 Personality and Intelligence | 37 |
| 2.1.1 Virtuosi | 39 |
|  | 40 |
4.1.2 Social Network Analysis 91
4.1.3 Being Plagiarized 92
4.1.4 Turn Everyone into a Programmer 96
4.1.5 Pattern Language 100
4.1.6 Software Team Capability 102
4.1.7 Rough-Cut Design 105
4.1.8 Training Is Not a Solution 107

4.2 Nothing Faster than Plagiarism 107
4.2.1 Immorality 108
4.2.2 Unprecedented Code 110
4.2.3 People Network 111
4.2.4 Rhythm for Plagiarism 112
4.2.5 Plagiarism at Work 114

4.3 Business and Rhythm for Plagiarism 117
4.3.1 15-Minute Business Presentation 118
4.3.2 Marketing Research 120
4.3.3 Chatting Robot 121
4.3.4 Old Song, New Singer 125

References 129

5 PAIR PROGRAMMING 131
5.1 Art and Science 132
5.1.1 The Right Partner 133
5.1.2 Noisy Programming 134
5.1.3 Just Training 135
5.1.4 Pay to Watch 135

5.2 Two Worlds 136
5.2.1 Moneyless World 137
5.2.2 Money-Led World 139
5.2.3 Economics 140
5.2.4 Mythical Quality–Time 140
5.2.5 Elapsed Time Accelerated 141
5.2.6 Critical Path Method 142
5.2.7 Why Two, Not Three: The Antigroup Phenomenon 145
5.2.8 Software Requirements Are Puzzles 146

5.3 Programming Task Demands 148
5.3.1 2 and 4 Is 6 148
5.3.2 2 and 4 Is 4 149
5.3.3 2 and 4 Is 3 150
5.3.4 2 and 4 ≥ 2 151
5.3.5 2 and 4 is Unknown 152
5.4 Pair Programming Is More than Programming  
  5.4.1 Design by Code  
  5.4.2 Pair Design  
  5.4.3 Rhythmic Pair Programming  
5.5 Pair Programming Team Coached  
References  

6 REPEAT PROGRAMMING  
6.1 Controversies in Pair Programming  
  6.1.1 Is Programming a Unique Work?  
  6.1.2 Are Three Minds Better than Two?  
  6.1.3 Unreplicable Experiments  
6.2 Repeat Programming  
  6.2.1 Variances  
  6.2.2 Principles  
  6.2.3 Triple Programming Unproductive  
6.3 Rhythm: Pair–Solo–Pair–Solo  
  6.3.1 Persistence  
  6.3.2 Connection  
  6.3.3 Motivation  
6.4 An Exception that Proves Brooks’ Law  
  6.4.1 Low Morale  
  6.4.2 Communication Costs  
  6.4.3 Rhythm for Late Projects  
References  

7 AGILE TEAMING  
7.1 Project Teams  
  7.1.1 Self-Organizing Teams  
  7.1.2 Teams in a Team  
  7.1.3 Project Team Composition  
  7.1.4 Team Lifecycle versus Learning Curve  
7.2 Productivity  
  7.2.1 The Illusion of Productivity  
  7.2.2 Collective Code Ownership  
  7.2.3 Accountability, Responsibility, and Transparency  
7.3 Problems and Problem Owners  
  7.3.1 Rhythm: Trouble–Restructuring  
  7.3.2 Teaming Principles  
References
7.4 Failing Projects Rescued
   7.4.1 Project Traffic Light Reporting
   7.4.2 A Business Case
   7.4.3 Steering Committee Meeting
   7.4.4 Agile Teaming in Action
7.5 Beware of Iago
References

8 INCREMENTAL DESIGN
   8.1 Modeling and Planning
      8.1.1 Agile Planning
      8.1.2 Design by Functional Modules
      8.1.3 Simple Design
      8.1.4 Total Cost Concept
   8.2 Rework or Reuse
      8.2.1 Unpreventable Rework
      8.2.2 Improvisation
      8.2.3 Up-Front Design
   8.3 Just-in-Time Software Development
      8.3.1 The CMM Rhythm
      8.3.2 A Factory Tour
      8.3.3 Walking Worker
      8.3.4 Just-in-Time Software Development
      8.3.5 Incremental Design
   8.4 Requirements Complexity
      8.4.1 Forgotten Requirements
      8.4.2 Conflicting Requirements
      8.4.3 Rapidly Changing Requirements
      8.4.4 Requirements and Design
   8.5 Refactoring
      8.5.1 Refactoring Activities
      8.5.2 Refactoring by Challenging
      8.5.3 Refactoring for Design Patterns
      8.5.4 Making Deliberate Mistakes
References

9 TEST-DRIVEN DEVELOPMENT
   9.1 Reverse Waterfall
      9.1.1 Design–Code–Test
This book helps us discover our own software methodologies in a way that respects the software development rhythms of both people and practices.

In the deep dark night, lying down on Kande beach on the shores of Lake Malawi, we looked up into the cloudless sky. Countless tiny stars were blinking at us. A little tired, or perhaps just mesmerized by those distant, mysterious lights, we closed our eyes and began to hear more, the peaceful slap of water on the little beach, and the small, almost concerted sounds of the dark night, throbbing in what seemed like a deep, rhythmic breathing. Nature is an incomprehensible concert of rhythms. Our Earth in its solar orbit spins through space composing the rhythm of day and night, endlessly recycling its four seasons. Following nature’s rhythm, we wake to learn and sleep to remember, writing and rewriting our own programs in accordance with the very best universal software practice in a flawless symphony of rhythms. From heartbeats to footsteps, rhythms are a sustaining, momentum-creating vital force. In a world where complexity appears very much like chaos, we seek the confidence of being able to assign causes and identify correlations, but sometimes it is only the discovery of rhythms that allows us to see the order that sustains all.

Like any human endeavor, software development is complex and full of generalizations and correlations, but it is devoid of rules. To help us build software, we have disciplined software models and software project management methodologies. But the ferment of software development, with constantly changing teams and requirements and new tasks, means that there is no guarantee that any past successful method will succeed on the next software project. In fact, some project leaders who appear to adopt no method or methods that are scorned as “ad hoc” are able to get their software projects
done on time. The secret of their success is the understanding of software development rhythms.

The knowledge of rhythms gives us a new perspective on some of the thorniest issues in software development. Methods that work for one team fail for another because even the most willing software teams can’t achieve success with a new method until they come to understand its rhythms. Yet in the management of complex and multifaceted software development projects, where it is vital to harmonize and synergize both team and individual practices and processes, rhythms are a largely neglected theme.

Rhythms are not another methodology. There are many methodologies, and this book does not seek to introduce a new framework for building software or managing software projects. What is needed today is not more methodologies but greater wisdom in the application of the methodologies that we choose to use. The best way to do this is to understand and work in harmony with the rhythm of whichever methodology the team adopts. To do otherwise, to fail to understand and apply the rhythms of a method, is to make the method itself more burden than benefit, and to make the journey of a project long and difficult.

This book is not for beginners. In fact, we assume that you can already fish and have caught a few in your time. It is for people who want to refine or even rediscover some of the skills and techniques that can so easily be lost when we get into the habit of seeing things from just one perspective. Then, like someone who is fishing casting the line with a supple wrist and a steady rhythm, we hope to help you catch more fish than ever before, and to feel more satisfied as you do it.

**Audience**

We have tried our best to write a technical book in plain language. Those who are interested in software development and project management (software managers, programmers, researchers, etc.) should have no trouble with these materials as we explain and provide clear examples of any terms that might be outside those areas.

Along with Kent Back’s *Extreme Programming Explained* (2nd edition), this book can serve as an advanced text on agile software development. It describes a number of project episodes and industrial cases suitable for use in case-based learning or for presentation to students as the basis of further work in group projects. This book is also a monograph as it presents many concepts that have not been adequately considered in books and scholarly papers on project management in general and software engineering in particular.
This book does address itself to a wide readership, but it is especially intended for thoughtful readers in search of creative metaphors for project management and new insights into the complex field of software development.

How to Read This Book

Generally, I would suggest that this book be read according to the chapter order. It is presented in two parts. Part I consists of three chapters. Chapter 1 introduces the idea of software development rhythms. Chapters 2 and 3 respectively discuss people and practices, clarifying some fundamental concepts in software development and asking some important questions such as “Why shouldn’t we learn from experience?,“ “what are agile values?,“ “How can it be possible to weight different intangible software practices as heavy or light?,“ and “What can we learn from open source software development?“

Part II of this book is all about development rhythms. We are used to using the familiar terms “process” and “practices”—although not everybody is confident that they know the exact differences. We compose rhythms on the basis of software practices. To effectively demonstrate how software development rhythms are a powerful metaphor that we can use to analyze when best to use a software methodology, we take a number of more controversial software practices and consider their rhythms and compare them with some other more generally accepted software practices.

Once you have learned how an analysis of software rhythms can harmonize practices, you may like, as an intermediate step, to adopt the rhythms proposed in this book or modify them in any way. Feel free. Ultimately, however, it is important to realize that rhythms are not models and that in the end, we should all compose our own rhythms.

Special Acknowledgment

The book covers some topics and ideas that are outside the normal scope of software development. Fortunately, we were able to benefit from the kind advice and guidance of a number of renowned experts in these areas. Their precious time and professional advice were much appreciated. We cannot thank them enough. In alphabetical order, they are

Paul Davies, who critiqued our description of the physicist and pair programming
Don Forsyth, who provided his insights into pair programming from a
group dynamics perspective
Michael McClellan, who reviewed the musical notation in this book
Richard Schonberger, who reviewed a number of sections connected with
lean manufacturing and engineering projects
Frank Vigneron, who commented on the angel’s gender
Joel Watson, who reviewed “Deal or no deal” and the game theory
analogy
Philip Zimbado, who advised us about his prison experiment

Many highly experienced software professionals have done us the great
honor of looking at one or two chapters of the book and providing valuable
feedback. Many thanks to (in alphabetical order): Lawrence Bernstein, Grady
Brooch, Magne Jørgensen, Pete McBreen, George Metes, Peter Middleton,

Thanks to Martin Kyle for his useful advice on writing better using plain
language. We appreciated John Nosek’s insights on collaborative program-
ming. We are delighted to have Angappa Gunasekaran’s support. We thank
Lai Shan Sit for her many funny cartoons illustrating our ideas. We would
also like to thank our friends and colleagues, Jun Wang, Zheng Li, Polly
Leung, Fei Dong, Ka Wing Wong, Whitney Lesch, and Rosalyn Farkas for
their assistance.

Finally, we cannot thank two persons enough. We thank Kent Beck for
inspiring our work on software development rhythms. Kent advised us on
which topics to choose to focus on and he reviewed the whole manuscript for
us. We also thank Paul Petralia, our editor, who told us from the very
beginning that he liked the concept of the book and thought that it would
motivate people to think about old things in their new ways. Without his
encouragement there would not be this book. It is not coincidence that all of us
believe that the idea of software development rhythms could be powerful
metaphors and effective management tools. Rhythms are not methodologies,
they are, rather, a meta-methodology – a methodology about developing new
software methodologies!

KIM MAN LUI
KEITH C. C. CHAN

Hong Kong
January 2008
Part I: Essentials
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...
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.

\[
R \rightarrow D \rightarrow C \rightarrow T \rightarrow R \rightarrow D \rightarrow C \rightarrow T
\]

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

![Diagram](image)

**FIGURE 1.1** How bill of materials (BOM) can be used for planning and costing.
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
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“All 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 Kahenman 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.” 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.