# Software Development on Adrenalin

Kenneth N. McKay

北京理工大学出版社
BEIJING INSTITUTE OF TECHNOLOGY PRESS

# Software Development on Adrenalin

Kenneth N. McKay

北京理工大学出版社
BEIJING INSTITUTE OF TECHNOLOGY PRESS

**图书在版编目（CIP）数据**

激情式软件开发＝Software Development on Adrenalin：英文/（加）麦凯（McKay，K. N.）著．—北京：北京理工大学出版社，2011.3
ISBN 978－7－5640－4228－8

Ⅰ．①激…　Ⅱ．①麦…　Ⅲ．①软件开发－英文　Ⅳ．①TP311.52

中国版本图书馆 CIP 数据核字（2011）第 012869 号

北京市版权局著作权合同登记号　图字：01－2011－0382 号

出版发行／北京理工大学出版社
社　　址／北京市海淀区中关村南大街 5 号
邮　　编／100081
电　　话／（010）68914775（办公室）　68944990（批销中心）　68911084（读者服务部）
网　　址／http：// www. bitpress. com. cn
经　　销／全国各地新华书店
印　　刷／保定市中画美凯印刷有限公司
开　　本／787 毫米×1092 毫米　1/16
印　　张／22. 25
字　　数／439 千字
版　　次／2011 年 3 月第 1 版　　2011 年 3 月第 1 次印刷
印　　数／1～1500 册　　　　　　责任校对／周瑞红
定　　价／68. 00 元　　　　　　责任印制／边心超

# Acknowledgements

There are a number of people who have inspired or provided feedback on this book. I will likely have forgotten some and wish to apologize in advance.

Some of the writing in the text has its origin in other pieces I have written or co-written with co-authors such as Gary Black and Vincent Wiers. Some of the writing is extracted or inspired from research work I have done with a number of undergraduate and graduate students. I worked with Jennifer Jewer on risk management ideas for software project management. And, then there was Louise Liu, David Tse, Sylvia Ng, and Hao Xin. They worked with me as I initially formalized my value framework for a course I taught and by working with them, the framework benefited.

While I have re-interpreted and re-crafted previous writings, it is possible that some bits will bear some resemblance to scattered text I have crafted or co-crafted before. Some of the Agile Overview and Ethnographic Methods sections come to mind.

The writing of this text, at this time, can be attributed to Dr. Alan George who gave me an opportunity to do a reasonably large project that fits the methodology described in this book. This project in turn led to an opportunity to try to teach students and others associated with the project, the principles and concepts behind the project itself; hence the book. Alan also provided input on some of the content, such as the ideal characteristics of software.

Several individuals are named within the text — people who radically altered my thinking and affected my IT skill sets: Brian Coleman, Romney White, and David Pryke.

Others who have contributed to the quality and content of the text, directly or indirectly include: Will Gough (who has had an ongoing software engineering dialog with me lasting well over a decade), Patrick Matlock, Jesse Rodgers, Doug Suerich, and Trevor Grove. The first set of students exposed to the text have also contributed in a number of ways: Nick Guenther, Ivan Surya, Ivan Salgado Patarroyo, Chris Carignan, Shanti Mailvaganam, Yatin Manerkar, Scott MacLellan, Rajesh Swaminathan, and Jarek Piorkowski.

Thanks also go to the Department of Cultural and Educational Experts, the State Administration of Foreign Experts Affairs, and the International Office of Beijing Institute of Technology for giving me the financial support to publish this book.

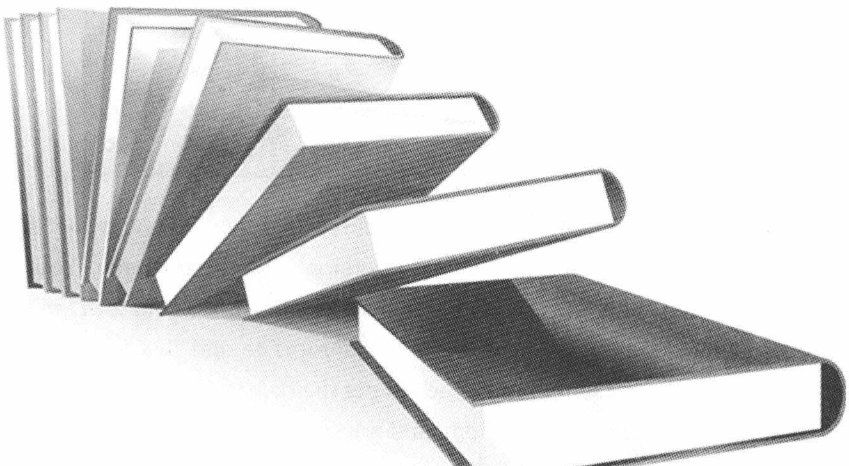
# Table of Contents

## Part Ⅲ Architecture & Design

## Part Ⅳ Level Ⅵ Rapids & Mushing

# Part Ⅰ

## S/W Development: a Personal View

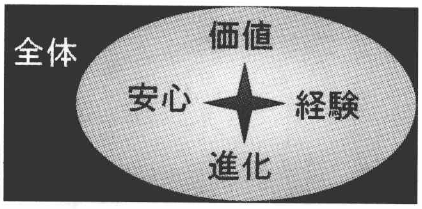


## Overview

This text is about software development that is potentially risky, causing the release of adrenalin and the rush that is associated. The heart beat that drives the development is speeded up, breathing rate increases, and the blood going to the muscles has more oxygen. This means that more can be done faster and at higher levels of achievement. It is symbolic adrenalin of course. Instead of chemicals, it is processes and ideas that speed things up and improve the muscles being used to develop the software. It is also about the feeling that comes from delivering software that the user values and wants to use. Software that the user will fight to keep using! It is a great feeling when the user actually values your code. When this has happened to me, it feels like an adrenalin rush. When both adrenalin rushes occur, it is a really good feeling and I have been fortunate throughout my career to experience both rushes repeatedly; the rush while creating and the later rush that comes from usage.

I have named this process of concurrently developing high value software at breakneck speed ***ZenTai Mushing***:

*ZENTAI MUSHING*



*ZenTai Mushing* refers to a holistic, unified way of viewing software functionality and usability combined with a high velocity version of Agile/Extreme. You can use the *ZenTai* design method with and without *Mushing*, and you can mush without *ZenTai*. The one is a *what* and the other is a *how*. Sometimes they are both appropriate and this book describes this situation: what they are, and how to use them together to get the dual rushes of adrenalin. When all of the variables align, the software form, function, and journey are unique and special. This is not a magical incantation though and there are many risks and possible rough spots, as not all people or processes can fit or operate in this fashion. Not all projects are suitable candidates either. It will push people's comfort zones and challenge assumptions. Good for some. Not so good for others.

Part Ⅰ introduces you to the basics of the above and the philosophy behind the text.

# Chapter 1 Introduction

This particular book is the result of approximately forty years of programming and development that has involved a wide variety of systems. It will not tell you everything you need to know about software development. Other software engineering books I recommend are Hunt and Thomas (2000), Glass (1997, 2003), Brooks (1995), McConnell (1996), McCarthy (1995), Jackson (1975), and Orlicky (1969). I suggest that you check out each of these and reflect upon what the authors are trying to get across. They are full of good suggestions and commonsense ideas. My own objective is to complement these other books and provide additional insights.

Who am I to write such a book in the first place? What are my credentials? I do not write witty, sarcastic blogs, issue on-line pronouncements or write about best practice, nor do I have a vast community of followers who hang onto every word I utter. I do not do self-promotion on the software engineering topic, and I try to avoid extrapolating off limited experiences. All I have done is design and code systems for close to four decades. Over the years I have quietly designed and programmed dozens of systems and software solutions ranging from operating systems and relational databases to accounting and veterinarian systems, and probably 150'ish end user applications based on custom toolkits I have created; sometimes as a team member, sometimes as a single developer. I have programmed an average of approximately 20,000 lines of code per year for more than thirty of those years, and more than once as much as 60–70,000 lines of code over a six month period. I am a geek, a code freak and have written lots of code. In over three decades I have failed once to deliver a project on time, on budget. No one is perfect. I like to build systems people fight to keep using and who find value from my designs and code. The proof is in the delivered systems and not in speculation and wild claims. I have had extended relationships with most of the systems I have been involved with, and I have been able to see how users have used the systems and to also see how the designs and code fared over time. I prefer to have demonstrations of skill, not could have, should have, would have, or will do.

I am not good at too many things. You never want to hear my attempts at music of any form. My kindergarten teacher thought my skills were so poor that she noted "difficulty in tone matching and in rhythms." How bad must you be to have this explicitly noted on your report card? In kindergarten? Nor would you like to see me dance, play sports, or attempt many other feats. And, my backyard shed went together in a scene from a comedy skit. However, I do seem to be good at software and software related activities. In my late-forties, a company executive with a firm I was interacting with nicknamed me *Code-Boy*.

During these many years I have evolved a specific style and approach. There are better programmers and there are better designers. There is always someone better. It is also good to work with better, smarter people, and I have been lucky to have worked with a number who have provided many lessons. I do not know if different is better, but it seems that I think and do things differently. I have been told this many times. Perhaps. Since I am not someone else, it is hard for me to judge others' thought processes. *ZenTai Mushing* is my attempt at describing the method behind my madness. *ZenTai Mushing* appears to be a way to consistently understand what is needed, and then craft the code that provides a unique, high value user experience that is obtained in very short order. The resulting software has demonstrated high quality, has been used for long periods of time with evolutionary changes, and has surpassed almost all costs and time expectations.

I am now reaching the end of my programming career and I finally feel ready to put words and thoughts on paper. I had actually planned to retire from coding when I turned fifty, but I have continued for various reasons. In the last two years I have done quite a bit of sustained coding, over 150k lines of code and it has allowed me to reflect once again on what I do and how I do it. The first draft of this book was completed just as I started coding again. Instead of a book written by someone with a few years of experience thinking that they are an expert and are capable of providing guidance on all matters concerning software, this is written by an old guy who has written a lot of code and who has specialized in making mistakes and learning from them.

Back in 1977 I actually had hair... really... thirty years later, I have ears... Beware, young geeks turn into old geeks...

Although I have touched a bit on some of the ideas in my academic papers, I have not directly approached software development as an author. I have always doubted my skill and ability, but in reflecting back over my career, it is hard for me to say that my repeated successes were accidental. They were not Herculean efforts with each being done via all-nighters and my face buried in the keyboard. They were done time and time again using a specific style and rhythm. I had a life most of the time. I hate egos and I hate people who go about talking about what they have done blah-blah. Especially those people who take one or two projects and extrapolate wildly to all kinds of software and projects. However, I also hate people who do not share with others any potential nuggets of wisdom that will help people following along behind. So, damned if I do, damned if I don't. I have felt uncomfortable writing most of the sections of this book.

Perhaps a few of my ideas have value and can be leveraged by others who can deliver better software products to the users with even more efficiency and effectiveness. I also do not expect anyone to pick up the whole lot and be able to do what I do. I am me and you are you. And, if you are older and have been immersed in one way for many years, it may be hard to adopt the ideas in this book. The ideas here are strictly another set of ideas to consider and you should have many in your arsenal. There is no single thing to do for achieving success, and you need many tools. I also do not know all of the causal relationships between the ideas. Sorry. I have also learned that most of the ideas in this book are not likely to be appreciated or understood by junior or inexperienced developers, or by software developers who are more technicians or assemblers than they are developers. Nothing I can do about that either.

I hope you are not the type that will read this book and say "that just can't work" or "I cannot do that." This type of attitude is self-defeating and sad. Over the years I have worked with positive, open minded individuals, and others who are closed minded and who insist in a narrow view; open to any way as long as it is their way. For effective software development in a number of areas, you need to be open and willing to adapt and change. I often give people a chance to show me their way first and see how it goes. Give them the benefit of doubt. If the results are close enough, we will both be happy and I will have learned something new along the way. If there is a large gap in results that cannot be dealt with, I will not be happy and if I am accountable for the project I will have to do an intervention and re-anchor the project and process: my way. You start off working with people, then go around them, and finally you might have to remove them from the project.

There is a risk with reading any book like this. You cannot be a perfectionist or be 100% literal in interpreting the methods and ideas you read. I have never done two projects exactly the same way. For high velocity development you need to be open and willing to experiment, lead with your chin, and develop fast responses. You cannot be literal, pedantic, rigid, or a perfectionist if you are going to apply the ideas you will read here. Over the years I have had supervisors, peers, and subordinates say that these ideas do not work, cannot work, and could not have worked. They do, can, and have. But, they have to be interpreted in the context of the project you are doing and the team you have.

If there is one key to my whole approach, it is one underlying assumption. I try to always remember that:

*I never know the right or one-and-only way to do something.*

I always doubt. I always question. What I am more confident about are wrong ways. I know **many** wrong ways, some of which are less wrong than others. I am an expert on *wrong*. I have learned the wrong ways by making many mistakes in my career. I seem to learn best by making mistakes, by willing to admit that I can and do make mistakes, and then by trying my hardest to learn from the mistakes and not repeat them. Luckily, most of my mistakes do not get seen or experienced by the users.

In a software project this means that I am willing to make mistakes with code and then re-write the code as necessary, when necessary; you need to know when something is at the end of its life and when it should be buried. As an undergraduate student in 1974, I wrote the worst code I have ever seen, fragile and really ugly. It was terrible.

That piece of bad code provided me one of the best lessons in my software career. The functionality was great and the users were very happy, but the code was horrendous in terms of robustness, and maintainability. It was like a plate of cooked pasta. I learned a lot from that first, major programming experience. It was my first assembler program, about 5k lines of code, and I have tried to avoid the same mistakes ever since. I had to maintain that piece of code for over two and a half years, and every day I checked with the operations group: "Did it run last night?" If not, I would skip algebra, calculus or statistics to fix the software. I was not asked by my supervisor to take such accountability. I just thought it was the professional thing to do. I built the fragile system and I was responsible. In hindsight, I should have built better software and skipped fewer classes. I buried the code in 1976 by replacing the code with a much better software program, better user functionality and better robustness. Code was designed to be robust and reliable. I learned many things because of that initial program and I used the lessons in subsequent software. How good were the lessons? I have been told that the replacement code was still being used in 2001. Several of the other programs I worked on or created in the mid 1970s also had long deployments. Some were running a decade or two later. I used the lessons again when working with a team in the early to mid 1980s. I have been told that the basic ideas and architecture developed in 1981 are still being used. I have used the same basic concepts throughout my career. I learned how to make good code the old fashioned way, by writing lots of code, making mistakes and learning from them.

If you only take three things away from this book, here are my three most important points to share. They are my humility principles:

- **Assume that you really do not know the requirements and what you think you know about the problem is partial, and possibly wrong.**
- **Assume that your design is faulty and that pieces will have to be ditched in a hurry and replaced.**
- **Assume that your code is buggy and that you are NOT a code ninja.**

Notice how these assumptions are aligned with my key underlying assumption of not knowing what is right! If you apply these three humility assumptions, I believe that you will then do requirements analysis in a certain way, that you will then design and build architectures in a certain way, and that you will then code in a certain way. The end result will be resilient, flexible, and sympathetic to the user's changing requirements, and be very robust. If you assume and act like you are an expert, your projects will likely stink and will possibly have a short shelf life. If you assume and act like you are **NOT** a hotshot, the projects are probably going to be far better than you imagined they could be.

And, be proud of your mistakes if you have learned from them and have controlled the damage the mistakes caused. Here is a phrase from a fortune cookie:

*How can you have a beautiful ending without making beautiful mistakes?*

I think that this is very true for software. You can indeed have beautiful mistakes and that they can contribute positively to a system. But not all mistakes are created equal. There are good mistakes that help get you to the beautiful endings, and there are mistakes with zero value. I often describe the task of management as constantly solving problems, some big, some small. This is what an analyst also does; constantly solving problems and just like a manager, he/she must develop good problem solving skills. A good manager will solve a problem once. If the manager keeps solving the same problem, being a manager might not be the best career for that in*duh*vidual. A repeated mistake is not a good mistake. A good architect and designer should also solve a problem once, or at least remember how to solve the problem when encountering it again, perhaps in a different context, going by a different name.

This book is not really about the methods and ideas for software engineering. There is probably not a new or unique idea to be found in this book. Good programming that is full of commonsense has been done since the beginning of automation and most ideas are built on other existing ideas. Some of the suggestions I will make about how to look at mission critical problems, or identify what characteristics to manage via interfaces are inspired by Babbage's 1832 masterpiece on manufacturing. Nothing is really new in terms of the individual ideas. What I am describing is how all of the ideas in the book can be used together.

At the end of the day, it is very much about what the final software provides the users! It does not matter to me how good the software is with respect to technical savvy and exotic features, or whether the programmer would have fun building the software if the user cannot or will not use it. The user's value comes first. That is the most important part of software development. This is a book about creating software that people want to use. I think it is about creating good software. But, what is good software?

Here is a brief summary of what could be called good or ideal software characteristics:

1. Software should be reliable and available when a user wants to use it.
2. Software should always focus on the user's goals and objectives.
3. Software should respect the user's time and effort, avoiding unnecessary data entry, unnecessary navigation, and unnecessary re-entry of data.
4. Software should recognize the user's knowledge and experience, and adjust the level of guidance and help accordingly.
5. Software should match the semantics of the task and problem, using the language of the user community.
6. Software should be generally self-supportive, without the need for "outside the system"

spreadsheets, documents, and databases.

7. Software should be intuitive and require minimum documentation, training, and instruction.
8. Software should naturally fit the user and not force the user to unnaturally fit the software.

## 1.1 ZenTai

*ZENTAI* (全体) basically means the essence of the unified whole and when I consider software, I include the users and interacting systems as part of the whole. I also consider the whole life cycle. The *ZenTai* way of design has four pillars and I will introduce them with Japanese words and concepts since Japanese (and Chinese) do a better job than English at describing the intent and spirit.

The first pillar is **Kachi** — 価値 — meaning value. To me, a good piece of software provides real value to the user and each step or activity must contribute to the generation of that value. No non-value added GUI, no waste, no clutter. Features and functions exist when and where you need them. Every screen and every feature on a screen needs to be considered in terms of the value chain and what it contributes.

The second pillar is **Anshin** — 安心 — comfort. This covers fear, stress, anxiety, and the general way a user feels when using your software, either because of the software, or because of the situation in which the software is used. The software should be designed understanding when and where the software will be used, and what can cause the user feelings of discomfort as these feelings can result in errors, frustration, and other user issues. A comfort analysis should be done for each bit of the interface and functionality. How does the system improve the comfort level and how does the system decrease the comfort level?

The third pillar is **Keiken** — 経験 — experience. Systems must recognize that people bring different initial experience (as in "I have experience doing that") to the system in the first instance, that they will accumulate additional experience through the use of the system, and that they will simultaneously accumulate other experience external to the system. Repeated experience with suitable feedback leads to expertise and different skill levels — which can affect how they interact with the system. What experience does the system recognize? What experience does the system develop? The notion of experience means something to me when designing the user interface. What does it mean to you?

The fourth and final pillar is **Shinka** — 進化 — evolution. Nothing remains as it is. Constants are not and variables will not. Assumptions are momentary things that should not be clung to. The software entity in its design and interface must support and accept evolution in the intended purpose, form, and function, as well as evolution in the user and the environment around the software. It is not likely that the users and the world will be standing still. Why do you think your software will not change? What are the assumptions behind each major function or software bit? What if those assumptions change?

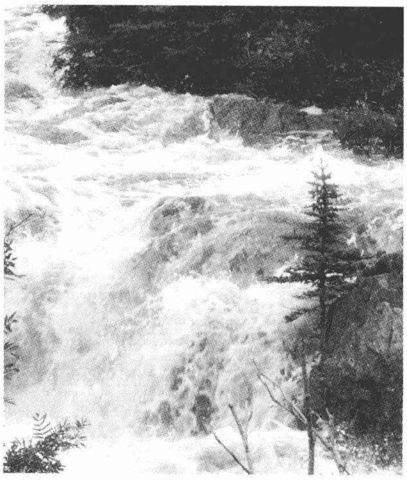
All of these pillars will be explained in later chapters.

## 1.2 Mushing

The term *Mushing* is used for several reasons. I use it because it best describes what the process looks like from a few meters away. It is like ceramic art being formed. The software process I call *Mushing* looks ill-formed, chaotic and sometimes looks full of hand waving, and it seems like nothing is firm. This is not a bad interpretation. When *mushing* you are working with partial information solving partial problems arriving at partial solutions. In another sense, the high end projects I am describing in this book feel like dog sled mushing when you are in a situation going through the wilderness, careening around curves and over bumps without knowing exactly what is around the curve or over the hill. You are relying on the team and the lead dog. And, when in the wilderness, you have to be able to deal with whiteouts, wind, cold, isolation, and rely on your own knowledge and skill. You have to be able to fix the sled and make new paths when needed. Not only do you need to know how to careen, you need to know how to solve problems from first principles. For the type of software addressed in this book, you cannot rely on the web, surfing for the answer and then assembling the solution via copy and paste. You need to know how to program, really program and not rely on assembly skills. So, I like *Mushing* for a few reasons. I could have also used white water experiences for the hurtling and skills needed to survive. In fact, since there are nice rating schemes for rapids, I will use that analogy throughout the book.

The *Mushing* I describe in this text could be considered an agile and extreme version of Agile/Extreme; when you cannot find solutions online, find best practices, or just assemble solutions. These types of developments are not as common as they once were, but if you are pushing the limits, you might find yourself with one of these. I do not know of any software rating scheme that can be used for categorizing software projects with respect to agility requirements. The types of projects I typically do could be described using the international white water classification scheme (American Whitewater — www.awa.org):

- **Class Ⅵ: Extreme and Exploratory.** These runs have almost never been attempted and often exemplify the extremes of difficulty, unpredictability and danger. The consequences of errors are very severe and rescue may be impossible. For teams of experts only, at favorable water levels, after close personal inspection and taking all precautions. After a Class Ⅵ rapids has been run many times, its rating may be changed to an appropriate Class 5.*x* rating.



Although there are exceptions, most of the cases and stories I have heard about Agile/Extreme being used in read more like Class Ⅰ or Ⅱ:

- **Class Ⅰ: Easy.** Fast moving water with riffles and