Innovative Software

Innovative Software:

Running the Rapids

By

Kenneth N. McKay

Cambridge
Scholars
Publishing

Innovative Software: Running the Rapids

By Kenneth N. McKay

This book first published 2019

Cambridge Scholars Publishing

Lady Stephenson Library, Newcastle upon Tyne, NE6 2PA, UK

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

ISBN (10): 1-5275-2781-6
ISBN (13): 978-1-5275-2781-2

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

There are many people who have inspired or provided feedback on material used in 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 material is from courses I have taught. Some of the writing is extracted or inspired from research work I have done with undergraduate and graduate students. I worked with Jennifer Jewer on risk management ideas for software project management, and Cathy May on ideas for innovative software project management. And, then there were 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. I have also enjoyed many system design and architecture discussions with Quinn Turner. I have learned many things from many people.

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. My previous publication from ten years ago and now out of print called *Software Development On Adrenalin* forms the backbone of this text.

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 this text, directly or indirectly include: Will Gough, Carol Hulls and the Hulls family (Mike, Carey, Maggie), Chelsea Marr, Patrick Matlock, and Alan George. Special thanks actually to Will and the Hulls. Will patiently read and read, gave valuable feedback, and helped with the mechanics of making it pretty. The Hulls family provided good challenges and pointers on what to fix!

PART I

DEVELOPING INNOVATIVE SOFTWARE: A PERSONAL VIEW

OVERVIEW

Innovative software? Why a book dedicated to innovative software since all software is innovative to some degree? Answer: because the degree matters. Innovativeness relates to changes in the status quo. There is always something new afoot in the who, what, where, when, why, and how. I have never seen a software project where there was not something 'different'.

If there are few changes in the status quo, the degree of innovativeness is relatively low. Think of kayaking or rafting in your backyard. Hard to get hurt. You have to try hard to mess it up. So, if your software project is reasonably safe with few risks, this book is not for you. There are known practices, design patterns, and the path forward can be tackled by the less experienced. You can find many other books and resources on the web that will help you. You can stop reading.

However, if there are many changes in the status quo that impact almost all of the who, what, where, when, why, and how dimensions, the degree of innovativeness can be high and is akin to running the extreme white-water rapids; the ones where all assumptions are challenged. You need a different game plan if you want to come out of the other end in one piece. You will need a different level of training and a different skill set. Compared to calm water, you will need a different type of kayak or raft and your shore crew will not be the same as your backyard adventure. This book is for the software projects which start to get risky and which require a different approach and strategy.

This text is about **rapid** software development that is highly innovative and potentially risky; causing the release of adrenalin and the rush that is associated with risk and the unexpected. Truly innovative software is like running extreme white-water rapids in a kayak or raft, not knowing what lies ahead. If you are looking at this overview, either you are in the middle of a run or thinking about doing one. Running the rapids, going into the unknown and surviving, possibly thriving, is possible if you go prepared and understand unknowns and what they imply.

Developing software that has not been created before is inherently different from re-implementing software, enhancing software, or copying software. The more ***new*** the software is to both developer and end user, the higher the risk and more different the journey will be. Most software projects that involve innovative software end up over budget, behind schedule, and with dissatisfied stakeholders on the first go around. This book is about improving the odds! The concepts in this book have been used repeatedly over the last five decades on dozens of projects and they have resulted in most (not all) projects being done on time, on budget, and with very happy users. About 1/2 of the book is about design and user requirements, 1/2 is about project management and execution.

Isn't innovative enough? Why do I include **rapid** in the game plan? Familiar with the one-week build plans associated with Agile? That is too slow in my opinion. I prefer to 5x this – daily build sequences for 90% of the build. This is really rapid development. Doing something rapidly implies a kind of velocity. And, velocity implies speed and direction. To keep people's interest (on both sides – development and user), it is important to have quick build cycles where there is something demo-able at the end of each cycle. You build in smaller pieces so that the cost of refactoring and code correction is controlled. You build in smaller pieces to also avoid the personal attachment some programmers develop with their code – it is easier to change code that you do not have much invested in. Smaller delivered chunks imply everything is speeded up.

One day build cycles – final design, code, debug, integrate, system test – tag and bag it. Daily. As someone commented, instead of *milestones*, you will be thinking *inchpebbles* – lots of small deliverables. There are many benefits. You never know when you have to show someone something and I believe it is best to be prepared. I also have a short attention span and like to fix or change things as soon as possible after they are coded. I have also found that users have a short attention span and like the daily build cycles. The direction aspect of velocity reflects the choice of activities and deliverables per build cycle. The toothpick strategy described in this book sets the initial direction and guiding light. As you build out the tree from a mere toothpick, combining the toothpicks to make a trunk, then adding the limbs and leaves, you have to do it in a balanced way. So, the direction is say 10-20% on each subsystem, perhaps a bit more, but you grow the systems together, discovering how the tree should be shaped in a natural, organic fashion influenced by internal and external forces.

Large build cycles, such as a week, invest way too much effort in something only to find out that changes should have been made along the way. I believe that highly innovative projects are best developed using a high-velocity philosophy – high speed with strategic direction. The developed code must also be nimble and adaptable, but high-velocity addresses a different characteristic of the build experience. The daily build strategy creates a nice balance of stress and tension with daily accomplishments that helps focus the development team's energy.

If you think you are doing innovative software, and it is going along fantastically well on all fronts, but you are not experiencing adrenalin rushes occasionally, it might not be as innovative as you think it is! I do not think that it is possible with *truly* innovative software to anticipate everything and manage everything in a predictive, calm fashion. This book will help avoid some of the bad rushes and provide the occasional good rush, but there will still be lots of excitement when living on the edge.

This book 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 also feels like an adrenalin rush. When both adrenalin rushes occur – doing something successful that is innovative and that users value – 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, innovative software at breakneck speed ***Zentai Development***. *Zentai Development* refers to a holistic, unified way of viewing software functionality and usability combined with a high-velocity version of Agile/Extreme. In Japanese, *Zentai* (全体) means the whole, the entirety. One aspect of the *Zentai Development* method is a *what* and the other aspect is a *how*. It is possible to use the *what* ideas with or without the *how* ideas and vice versa. 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 innovative, 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. This is somewhat short and can actually be skipped if you want when you first dive into the book.

Part II is perhaps the most important part of the book: because if you do not get this bit right, you might as well close up shop now. It is the part of the process where experts tell you that you 'need to know enough' – you need to know enough about the problem and what the big, high value deliverables need to be to address the problem. This is the one part of the book you should not skip!

Part III has some potentially useful ideas for design and architecture when you are doing something new. This is like icing on the cake and can be totally ignored if you want. I think it has useful stuff, but that is my opinion. It is for people interested in system design and how the big pieces fit together.

Part IV is probably useful to the majority of readers. It is about project management and execution – when you are running the rapids. I would read this after Part II.

CHAPTER ONE

INTRODUCTION

1.1 My Journey

You can skip this section if you wish. It is about my general attitude and approach to developing software.

This particular book is the result of approximately fifty 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. They are oldies and goldies. There are also many other good blogs and books if you surf a bit. My own objective is to complement these other sources 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 five decades, starting in 1968. Over the years I have 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 forty of those years, and multiple times as much as 60-70,000 lines of code over a six-month period. Probably close to 2m lines of code in my career using a wide variety of languages and systems from drivers to applications. I have kept myself busy creating code, not pontificating

about it. I still code at 65. I am a geek, a code freak and have written lots of code. In this time, I have failed twice 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 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 tone matching and in rhythms”. Seriously! 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 above average at software and software related activities.

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 Development is my attempt at describing the method behind my madness. *Zentai Development* appears to be a way to consistently understand what is needed in an innovative setting, and then quickly 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 cost and time expectations. More than one system has grown from a 10k proof of concept to 300-500k first release. The trick is understanding the problem space, and then designing a system that respects and reflects the target situation – and not forcing a solution onto a problem.

I am now probably reaching the end of my programming career, perhaps another five years at the keyboard. In the last decade I have done quite a bit of sustained coding on three projects, over 500k lines of

finished code and it has allowed me to reflect once again on what I do and how I do it. 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, who does not think of himself as an expert, and who has specialized in making mistakes and learning from them.

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 extrapolates 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. Perhaps you should write a book too? The more wisdom and experiences we can share, the better off things might be.

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, 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. Nor by people who have never been challenged by trying to create truly innovative software. 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", "that cannot be done", 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 on 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 go through them, possibly removing 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. Remember, this is a book about doing software that you have not been done before. The implication is that there will be new requirements, new solutions, new methods, new technology, and new problems to solve.

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. Not possible! Crazy ideas. No so crazy and not so impossible. They do, can, and have worked; repeatedly. However, 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. Luckily, most of my mistakes do not get seen or experienced by the users. 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. In some sad, sick way, I think I actually like to experience mistakes and fail. I embrace failures. You learn so much by failing and when you fail and think about the failure and dissect it, you learn more about whatever you are trying to do, and more of the subtle bits or hidden causes of failure. This brings me happiness. This is the discovery element! Creating highly innovative software is a voyage of discovery and it feels great when you finally get to the destination.

In the software project context, 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. Real spaghetti code. I learned a lot from that first, major programming experience. It was my first assembler program, about 5k lines of code, and 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 designed to be robust and reliable. I learned many things because of that experience 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 leading a team in the early to mid-1980s. A couple of years ago, I was told that the basic ideas, architecture, and even some of the code developed in 1981-4 were still being used – thirty years later. 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 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, 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 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 if the programmer had fun building the software if the user cannot or will not use it. I do not care about what can be tweeted about or shared on social media. I do not care what others think about my code or my designs. I do not worry if others will make fun of my design or code. They do not matter. The user's value comes first. The users matter. Not other geeks or my reputation within the geek world. I write simple code because I think simple code is best. I can write fancy code and use obtuse functions and exotic libraries, but I find that these esoteric approaches rarely actually help the users. The users are the most important part of software development. They need software that delivers value, is robust, is maintainable, and can evolve over time.

This is a book about creating software that people want to use. I think it is about creating good software. 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, 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.

1.2 Your Journey

This section is intentionally left blank – for you to write about your own experiences and your own learning experiences. Use crayon, pen or pencil.

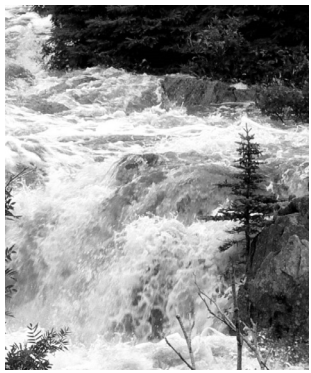
CHAPTER TWO

ZENTAI DEVELOPMENT

2.1 The Rapids

The terms *high-velocity* and the white-water analogy of *running-the-rapids* are used in this text for several reasons. I use them because they best capture what the process feels like and looks like from a few meters away when I am creating really innovative software at high speed. It does not look like a Waterfall process and it also does not look like Agile. The software process at times looks ill-formed, chaotic, sometimes looks like it is full of hand waving, and it seems like nothing is firm. This is not a bad interpretation. And there is no scrum-master in the traditional sense – best to think of someone careening down the rapids in a raft, sitting at the rear, guiding the raft's journey.

When working in a very rapid way at *high-velocity* you are inevitably working with partial information solving partial problems arriving at partial solutions. This is what is going to happen in the extremely innovative projects. The more innovative – the more partials! The high-speed projects I am describing in this book feel like hurtling down the rapids, twisting and turning, hanging on at times for survival, careening around boulders, surviving whirlpools, and without knowing exactly what is around the bend. If you do not understand the analogy, surf: "running class vi rapids". We are talking white water and lots of it.



Lots! See the water. Feel the force of the water? At the extreme level of software development, your job is to run the rapids and survive. *Zentai Development* gives you the tools and insights necessary to survive.

In these situations, you are relying on the team and the team's ability to react quickly to whatever situation is thrown in front of you. At times, 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 googling skills. Truly innovative software does not get done by cutting and pasting. So, I like *running-the-rapids* for a few reasons. Since there are nice rating schemes for white-water rapids, I will use that analogy throughout the book.

The *running-the-rapids* 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 successfully used for read more like Class I or II:

- **Class I: Easy.** Fast moving water with riffles and small waves. Few obstructions, all obvious and easily missed with little training. Risk to swimmers is slight; self-rescue is easy.

- **Class II: Novice.** Straightforward rapids with wide, clear channels which are evident without scouting. Occasional maneuvering may be required, but rocks and medium-sized waves are easily missed by trained paddlers. Swimmers are seldom injured and group

assistance, while helpful, is seldom needed. Rapids that are at the upper end of this difficulty range are designated "Class II+".

Agile/Extreme concepts can be used at all levels, but it is my view that the processes and concepts need to be adapted to the type of rapids being run. This will be talked more about in the project management chapter. Since I will occasionally refer to the white-water analogy, here are the remaining classes:

- **Class III: Intermediate.** Rapids with moderate, irregular waves which may be difficult to avoid and which can swamp an open canoe. Complex maneuvers in fast current and good boat control in tight passages or around ledges are often required; large waves or strainers may be present but are easily avoided. Strong eddies and powerful current effects can be found, particularly on large-volume rivers. Scouting is advisable for inexperienced parties. Injuries while swimming are rare; self-rescue is usually easy but group assistance may be required to avoid long swims. Rapids that are at the lower or upper end of this difficulty range are designated "Class III-" or "Class III+" respectively.

- **Class IV: Advanced**. Intense, powerful but predictable rapids requiring precise boat handling in turbulent water. Depending on the character of the river, it may feature large, unavoidable waves and holes or constricted passages demanding fast maneuvers under pressure. A fast, reliable eddy turn may be needed to initiate maneuvers, scout rapids, or rest. Rapids may require "must" moves above dangerous hazards. Scouting may be necessary the first time down. Risk of injury to swimmers is moderate to high, and water conditions may make self-rescue difficult. Group assistance for rescue is often essential but requires practiced skills. A strong Eskimo roll is highly recommended. Rapids that are at the lower or upper end of this difficulty range are designated "Class IV-" or "Class IV+" respectively.

- **Class V: Expert**. Extremely long, obstructed, or very violent rapids which expose a paddler to added risk. Drops may contain large, unavoidable waves and holes or steep, congested chutes with complex, demanding routes. Rapids may continue for long distances between pools, demanding a high level of fitness. What eddies exist may be small, turbulent, or difficult to reach. At the high end of the scale, several of these factors may be combined.

> Scouting is recommended but may be difficult. Swims are dangerous, and rescue is often difficult even for experts. A very reliable Eskimo roll, proper equipment, extensive experience, and practiced rescue skills are essential. Because of the large range of difficulty that exists beyond Class IV, Class 5 is an open-ended, multiple-level scale designated by class 5.0, 5.1, 5.2, etc... each of these levels is an order of magnitude more difficult than the last. Example: increasing difficulty from Class 5.0 to Class 5.1 is a similar order of magnitude as increasing from Class IV to Class 5.0.

The basic forms of Agile/Extreme have certain benefits in some situations when compared to the traditional Waterfall methods and are agile, flexible, and adaptive in ways that the formal Waterfall methods are not. However, it is possible to make Agile/Extreme too formal, too reliant on artifacts, buzzwords, and prescribed how-to-do-it methodologies. The prescribed methods might be necessary, but they are not sufficient. In addition, when a method becomes too standard and too formal, it has the possibility of losing any agility and flexibility it once had. There are no standards or 'only way' or normative prescriptions with *Zentai Development*. None. Not possible for innovative development. There are consistent principles and concepts, but the realization and instantiation of the process is likely to be different each and every time! If you go into a Class V or VI rapids with a firm plan that you are fanatically attached to, you will discover the hard way what extreme and exploratory really means.

I have worked with flexible developers and I have worked with developers who were very pedantic (i.e., rigid, black and white thinkers who are excessively rule or text book driven). People who take things literally and are pedantic will have lots of problems with this book and the way I develop software. It is like diesel fuel and fire. Key lesson I have learned: do not mix the two. I have had to occasionally, on some jobs you just have to accept who and what you are given, and the result has not been pretty and I have had to rely on Plan B. Plan A was just not going to work. The mismatched individuals might be fine on some types of developments but are not well suited for the extreme variety.

To recap, I use the phrase *running-the-rapids* to capture the feeling of kayaking or rafting down white water. As you are running the rapids, you are moving towards something that cannot be initially seen. Structure and key elements take form, features appear, final tweaking is performed, and the software takes shape. There is a flourish of movement and it looks chaotic and that it will never work, but something different is happening.

In the hands of an experienced craftsman, beauty in form and function will result. The project does not look like anything for a while and suddenly the form can be recognized, but the path is not straight and linear, the piece is reworked, and re-crafted. You cannot fight the software, you have to flow with it, feel it.

You hear the same types of analogies with white water kayaking; not fighting the river, feeling the river, understanding the river and going with the river's flow. Software can be created the same way, with the same benefits and the same risks. Sometimes you will hit a rock while kayaking or capsize because of the current. Sometimes the code works and sometimes it does not and you have to refactor a small bit or do major surgery. Nothing is perfect, no one is right 100% of the time. Almost everything is a compromise as well, with various trade-offs between effort, costs, value, comfort, experience, and evolution.

What is all of this discussion about kayaking and white water leading up to? Well, I have been wondering if something similar to the six white water classes could be described for Agile/Extreme software development:

- **Class I: Easy.** Obvious requirements, many examples, and existing toolkits. Almost all ideas and features are commonly known, with simple customization being done. Few problems, all obvious and easily missed with a little training. Risk to programmers and users is slight; self-rescue is easy.

- **Class II: Novice.** Straightforward development with occasional backtracking and recovery. Majority of functionality is well known and understood. Almost all problems can be missed by trained programmers, and individual programmer recovery is most common. Developers and users usually come out of the process unscathed.

- **Class III: Intermediate.** Development with moderate, irregular patterns of challenges which may be difficult to avoid and which can delay or cause an over budget situation. Complex maneuvers are often required to get the functionality right and to complete the build. Increased requirements effort is required in advance for inexperienced developers. Delays and problems can be recovered, but often require a group effort involving additional developers.

- **Class IV: Advanced**. Intense, powerful development effort as the requirements and process is turbulent and unstable. Many requirements become revealed as the project proceeds. Precise reactive refactoring and redevelopment needed during the development to keep daily builds going. There may be extended periods without daily builds as the refactoring and redeveloping may require additional requirements analysis and prototype development to test ideas. Risk is moderate to budgets and time expectations. Additional personnel might be needed at various points. Some tool work might be needed to adjust the toolkit's suitability to the tasks at hand. The work might be done in two stages; a fuller toothpick before full development is authorized.

- **Class V: Expert**. Extremely long, challenging developments with many unstable requirements that will not be revealed till the last minute during the development. Extended periods of tool crafting during development and re-jigging the system as major functions expose themselves. There will be periods where both developers and users doubt the wisdom of the endeavor. Developers not only need to know how to deliver the functions, use the tools, and do minor repairs on the tools, they will likely need the ability to build tools from scratch. Risk is high and project failure is a strong possibility. Initial toothpicks and fuller prototypes might be attempted to address some of the risks, but this will not be possible for all of the risks.

- **Class VI: Extreme and Exploratory.** These developments are relatively unique and few exemplars if any exist. The requirements are extremely fluid and appear to constantly change. The use of technology is also pushed to the limit. Progress is extremely unpredictable. There are many risks and the consequence of error is severe. This is for teams of experts and all precautions must be taken.

This book is focused on the last three classes – Class IV Advanced, Class V Expert, and Class VI Extreme and Exploratory. These are the categories with increasing degrees of innovativeness.

Two of the checklists we will introduce next in Section 2.2 can help you size your development to one of the categories. You are definitely in Class VI if the user situation is going to be extremely complex and full of end user innovation, and you are also pushing the envelope on the build

side with newness. That is, high degrees of innovativeness in both checklists get you to the top of the chart! Extreme New-New on a large, critical project will make it VI. The first checklist in the next section, on the team, will give you a read for how ready you are for such a ride. Your team best be prepared for the task at hand.

You can potentially back down to a Class IV or V depending how extreme the New is. A medium/high combo will likely be a V and a medium/medium will be a IV. A low/high will also get you into the V or VI – depending.

Plan and execute accordingly. As the degree of difficulty increases, you will need to adapt your processes and definitely review your assumptions and planning estimates. As the difficulty or newness increases, the crew is different, the tools are different, the prepping is different, the journey is different. If you ignore this, you will likely join the list of those who have gone before and who have failed. There have been the occasional, rare project that blissfully ploughs ahead ignoring all commonsense and was successful not because of what they did but because of what others did or did not do. The old phrase – easy to be perceived to be an eagle when surrounded by turkeys – comes to mind. Successful, long term enterprises rarely have long streaks of blind, random luck.

For a Class VI rapids, the kayak is likely to be different as the conditions will be more severe and the kayak might have to be fixed in the middle of the river, the kayaker will need different skills, experience and training (might have to fix the kayak), the prepping before the run will be different, supporting crew different, precautions different, and so on. Kayaking on grass or in a backyard pool is not the same as experiencing a Class VI rapids, and creating routine, familiar software is not the same as creating innovative software.

2.2 Rating the Rapids

We want to be clear. In the truest sense of the word, ***innovation*** implies any change in the status quo. It does not have to be a physical gizmo – it can be policies, processes, methods, software technology, anything. The more changes in the status quo, the more innovative the venture! A software project can be innovative in many possible ways – it can be how it is actually done, the project management, what tools are used, what the users see or do, and what the users use. Always think of the who, what, where, when, why, and how for both the *doing* and the *what*.