Doug Rosenberg · Barry Boehm
Matt Stephens · Charles Suscheck
Shobha Rani Dhalipathi · Bo Wang

Parallel Agile – faster delivery, fewer defects, lower cost

Parallel Agile – faster delivery, fewer defects, lower cost

Doug Rosenberg • Barry Boehm • Matt Stephens
Charles Suscheck • Shobha Rani Dhalipathi
Bo Wang

Parallel Agile – faster delivery, fewer defects, lower cost

Springer

Doug Rosenberg
Parallel Agile, Inc.
Santa Monica, CA, USA

Matt Stephens
SoftwareReality.com
London, UK

Shobha Rani Dhalipathi
University of Southern California
Fremont, CA, USA

Barry Boehm
Center for Systems and Software
Engineering (CSSE)
University of Southern California
Santa Monica, CA, USA

Charles Suscheck
Juniper Hill Associates
Liberty Township, OH, USA

Bo Wang
University of Southern California
Alhambra, CA, USA

Parallel Agile and Parallel Agile CodeBot are both registered trademarks of Parallel Agile, Inc.

ISBN 978-3-030-30700-4 ISBN 978-3-030-30701-1 (eBook)
https://doi.org/10.1007/978-3-030-30701-1

© Springer Nature Switzerland AG 2020
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors, and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Foreword

Software development has proven to be a highly problematic procedure; the data indicates that most development projects for systems or devices that contain a lot of software (and these days, that is almost everything) are significantly delayed, run over budget, and deliver far less capability than originally promised.

A number of factors—both technical and social—contribute to this depressing state of affairs:

- Most projects that contain software are awarded through a process of competitive bidding, and the desire to win the competition likely induces organizations to bid an amount that they consider the lowest credible price, with the shortest development schedule that they consider credible, too.
- Such projects are of quite amazing size and complexity; literally, in terms of the number of "pieces" involved, they are by far the largest and most complex endeavors that human beings have ever undertaken. It is routine for a system or device to have millions of lines of software code these days—BMW, for example, says that its newest cars have 200,000,000 lines of software code in them. I have seen estimates that Microsoft Windows and Microsoft Office are each about that size, too. No other human creation comes close to this level of scale and complexity.
- This is a difficult and specialized work, and unlike other human endeavors (e.g., building construction), it has proven difficult to separate the work by the various skills required, which places an additional burden on each software developer. In the building trades, no one is expected to be a master electrician, a master plumber, and a master mason, but in the software business, we often make designs that require each developer to have mastered quite a number of complex and diverse skills. This, naturally, leads to errors in the implementation.
- Such projects, due to their large size, now take very long periods of time to complete. Software development periods measured in years is a common phenomenon. These long schedules inevitably mean that particular individuals will come and go during the course of the project, and such turnover in a difficult and specialized work is an additional source of errors, delays, and cost increases.

I could go on and on, but you get the idea.

In my experience, the long development time periods are the most insidious aspect of this problem. Customers simply do not want to wait years for their new system or product, and long development time periods also increase cost—developers need to be paid every day.

What to do?

Naturally, many people have worked to solve this problem, myself included.

The collection of concepts and tools called "Agile software development" is one approach that has been offered to solve this problem. Unfortunately, Agile methods seem to work well only in a narrow set of circumstances and conditions.[1] These conditions do not seem to apply to very large systems—the ones that contain tens of millions of lines of software code. Yet these very large systems are often those that are the most important to society: automation systems for oil refineries and chemical plants, for healthcare diagnosis, for air-traffic control, for managing processing and payment of government benefits, etc.

One obvious way to shorten software development schedules is to do more of the work in parallel, that is, break the work into lots of small pieces to allow these small pieces to be built simultaneously by lots of separate teams. Often, unfortunately, the problem of selecting the set of small pieces so that they actually work the way you want when you try to put them all together after they have been built has proven to be quite difficult. Some of my own methodological improvements to the software industry are aimed at exactly this aspect of the problem (and have worked well within the industries and types of software in which I was interested). But there remain many other industries and types of software for which these problems remain unsolved.

In this book, Doug Rosenberg and my longtime friend and mentor[2] Professor Barry Boehm, together with a set of their colleagues, propound their own offering to address portions of this important—and still unsolved—problem. I believe that you will find what they have to say worthwhile.

University of Southern California, Neil Siegel
The IBM Professor of Engineering Management;
formerly Sector Vice-President & Chief Technology Officer,
The Northrop Grumman Corporation
Rolling Hills Estates, CA, USA
July 2019

---

[1] I talk about this in my textbook *Engineering Project Management*, published by Wiley.

[2] I am fortunate also to be able to claim Professor Boehm as my own PhD advisor.

Preface: Why Can't We Develop Software in Parallel?

From the beginning of software time, people have wondered why it isn't possible to speed up software projects by simply adding staff. This is sometimes known as the "nine women can't make a baby in 1 month" problem. The most famous treatise declaring this to be impossible is Frederick Brooks' 1975 book *The Mythical Man-Month*, in which he declares that "adding more programmers to a late software project makes it later," and indeed this has proven largely true over the decades.

Two of the authors of this book, Barry and Doug, have wondered for some time about how absolute Brooks' Law might be. When he was chief scientist at TRW in the 1980s, Barry had a team that did large-scale parallel development work on Ada projects, and Doug has spent a couple of decades teaching Unified Modeling Language (UML) modeling to classes in industry organized into parallel teams for lab work. It seemed to both that, at a minimum, there must be a loophole or two somewhere.

There's Gotta Be a Loophole

This book details our attempts over the last 4 years to find those loopholes. It started innocently enough when Barry and his co-instructor Sue invited Doug to be guest lecturer in their graduate class in the University of Southern California (USC) on software engineering, CS577, in 2014. This had been a once-a-semester invitation for a couple of years, but this time was different, because Doug had a project in mind that he wanted to get developed: a mobile app that used geofencing to deliver a coupon to a user's phone when he or she gets near a business. Thinking that it might be interesting to get the students to put together a UML model of this system, he offered to grade a couple of homework assignments. When this offer was accepted, he split his problem into 47 use cases and assigned a different use case to each of the students to model.

At this point, neither Doug nor Barry knew of their mutual interest in parallel development. Barry's reaction upon learning that Doug was assigning a use case to

each of his 47 students was simply a tilt of the head, a brief locking of eyes, and the comment, "That's *really* interesting."

Doug was a little unsure of what he was getting himself into, trying to critique 47 different use case designs in parallel, but he decided that if chess masters could play simultaneous games by quickly evaluating the position of pieces on the chessboard, he should be able to read class diagrams, sequence diagrams, and model-view-controller (MVC) decompositions quickly enough to make the attempt, and that however mentally taxing the effort might be, it would be worth it to get the location-based advertising system designed quickly, thus began a 4-year learning experience that resulted in this book being written.

We Learned a Few Things in Graduate School

The first lesson learned was that USC graduate students tend to do their homework between midnight and 3:00 a.m., the night before the assignment is due, and the second lesson was that most of these graduate students are really smart. The two homework assignments were called "Build the Right System" and "Build the System Right," with the first assignment covering storyboards, use cases, and MVC (robustness) diagrams and the second covering class, sequence, and data model diagrams. While grading the first homework assignment, it began to look like we were going to get a better-than-expected result, and we decided to offer an optional extra-credit assignment where the students could implement a prototype of their use case. We also decided to start tracking student time and effort expended. Twenty-nine out of the 47 students decided to try the extra-credit assignment, and that's when things got interesting.

But This Will Never Integrate, Right?

The original expectations for this exercise were that we would wind up with a fairly detailed UML model (which we did) and not much in the way of working code. The expectation of a decent UML model came from a couple of decades of ICONIX JumpStart training workshops, in which it is standard practice to work on a real industry project with multiple lab teams, with each team working on a different subsystem of the project. In those classes, we typically limit each instructor to three lab teams, so whether this approach could be stretched to 47 parallel threads of development was unknown. There was no expectation that any of the independently developed prototype code would integrate together, and the fact that it did became the first surprising result of the project.

The unavoidable fact that 29 independently developed use cases had somehow integrated into a system that hung together with a reasonable amount of cohesion seemed significant, because difficulty in integrating independently developed code

has long been one of the underlying reasons why Brooks' Law has remained in effect for all of the decades since he wrote it. It also defied explanation for a while—we knew something had happened, and we knew it had something to do with NoSQL databases and REST APIs, but the underlying mechanism wasn't immediately obvious to us.

A few years later, a clear explanation seemed to have emerged: we had applied microservice architecture (the same strategy commonly used for business-to-business integration), but at a more fine-grained level, where each domain object had its own API for common database access functions, and doing this had enabled developer-to-developer integration. This design pattern was named *executable domain models* and subsequently developed into a code generator that creates a functional microservice architecture from a domain model during the inception phase of a Parallel Agile project. Executable domain models mitigate two of the underlying factors behind Brooks' Law: they improve communication across a large team, and they enable independently developed code to integrate more easily. We'll be talking a lot more about executable domain models and how they are a key enabler of the Parallel Agile process in the chapters ahead.

4 Days per Use Case × 47 Parallel Use Cases … Is 4 Days?

The other surprising result was that we had taken a system from inception through analysis, design, and prototype implementation in about 28 hours per student total, with all of the students working in parallel. Since the students weren't working full time—this was just homework from one of several courses—the calendar time was around 5–6 weeks total. The detailed breakdown was around 8 hours for analysis, 8 hours for design, and 13 hours for prototype coding (see Fig. 1).

We thought this was a pretty fascinating result and worthy of further study. So we took the location-based advertising system through to completion and an initial commercial deployment over several semesters. We considered this first system to be a proof of concept. We subsequently took the system through a careful design

Actual time expended by 577B students was tracked and totaled 350–370 hours per assignment.

1 day on Build the Right System (requirements), average across 47 students
1 day on Build the System Right (design), average across 47 students
2 days on implementation, average across 29 students (80% of student HW worked the first time when delivered)

4 days per use case (total requirements, design, and implementation)

Actual cumulative LOE reported by students (hours)	361	358	369.5	
Hours per student (1 use case per student)	7.7	7.6	12.7	26.0
	Requirements	Design	Code/test	Total

Fig. 1 In 2014, we built a proof of concept system with a large team of developers, each working on a single use case in parallel

pass to build a minimum viable product (MVP) and then spent another semester producing a more refined version suitable for commercial release. Students for the MVP and optimization stages of the project came from a directed research program that Barry was running (CS590), where students typically worked 5–15 hours per week for course credit.

Proof of Concept, MVP, Then Release 1 in 3 Months

We followed up the original location-based advertising project with several additional projects over the next 3 years: a photo-sharing mobile app, a system for crowdsourced video reporting of traffic incidents, and a game featuring virtual reality, and augmented reality capability. We found the three-sprint pattern of proof of concept, MVP, and optimization to be a useful strategy that fits the USC semester schedule and that with part-time students and a 3-month semester, the full-time equivalent for each of these "semester sprints" was a little under 1 calendar month (see Fig. 2).

After 4 years of experimentation, data collection, and analysis, the results seemed clear and repeatable. Larger projects didn't have to take dramatically longer than smaller projects if we were able to scale the number of developers in each of our "sprints." The pattern we adopted was compatible with Barry's work on the Incremental Commitment Spiral Model (ICSM), a general-purpose roadmap for

Note
One of the noteworthy returning student exceptions was a brilliant woman named Shobha from the traffic incident reporting project (now called CarmaCam), who is also a co-author of this book.

In addition, a student from the first CS577 class, Bo, is now in the PhD program at USC and is a co-author of this book. Bo developed the REST API on the original location-based advertising (LBA) project, and subsequently, he developed the code generator for executable domain models.

Shobha wrote the chapter on our example project (Chap. 4), CarmaCam, and Bo co-wrote the chapter on executable domain models (Chap. 3).

reducing risk and uncertainty with a phased development effort, as will be discussed later in this book.

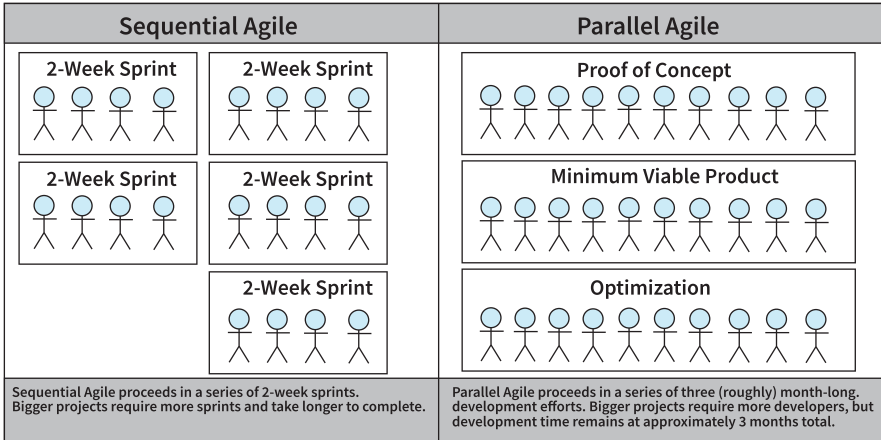

Fig. 2 Compared with a sequential Agile approach, Parallel Agile compresses schedules by leveraging the effort of large numbers of developers working in parallel

Surviving 100% Staff Turnover

There was one more noteworthy surprise: with only a couple of exceptions, we got a brand-new set of CS590 students every semester, and our projects were succeeding despite *nearly 100% staff turnover*. We were getting fairly sophisticated systems built with part-time students over three semesters, which each had a full-time equivalent of about 1 calendar month—so about 3 calendar months from inception to optimization.

So, have we repealed Brooks' Law? Probably not. But based on our experience, it does appear as though if you have properly partitioned the problem for parallel development, and if you have a good strategy for integrating the work that's been developed in parallel, you can in fact accelerate a schedule dramatically by increasing the number of developers.

Who Needs Parallel Agile?

You don't need Parallel Agile if your software development process is working perfectly, and you have no need to bring your software systems to market faster while simultaneously improving quality.

However, if you're like the rest of us mere mortals and you're developing software in an Agile environment, we hope you'll find some of our work interesting. If your feedback-driven development process is devolving into hotfix-driven development and you're not happy about it, then this book is definitely for you.

What's in the Rest of the Book?

Of course we'll reveal all of the important secrets of the universe in the remaining pages of this epic, but more specifically, you can expect to learn the following:

- Why parallel processing can speed up software development like it speeds up hardware computation
- How to be feedback driven and plan driven at the same time
- Why making domain models executable is an awesome boost to productivity and quality
- How to manage sprint plans using visual modeling
- How to do top-down scenario-based estimation and bottom-up task-based estimation
- The fundamental importance of the model-view-controller (MVC) pattern to use case-driven development
- Why acceptance testing offers greater "bang for the buck" than unit testing
- How to adapt Parallel Agile within your current scrum/Kanban management paradigm
- How all of the above topics have been put to use on the CarmaCam project
- The ways in which Parallel Agile is compatible with the Incremental Commitment Spiral Model (ICSM)
- How to scale Parallel Agile techniques to very large systems
- How to scale your projects horizontally by adding developers rather than vertically by stretching the calendar

Ready to get started? Continue to Chap. 1 for a big-picture overview of Parallel Agile concepts.

Santa Monica, CA, USA

Doug Rosenberg
Barry Boehm

Acknowledgments

Doug would like to thank the following people for their contributions to this book:

The Greatest Copyeditor of All Time, Nicole LeClerc, for taking on this project on top of her full-time job when she really didn't have the time and for helping us create a book.

Michael Kusters who allowed us to use The Scream Guide as Appendix A.

The USC CS590 administrators: Julie Sanchez, Anandi Hira, and Elaine Venson.

And most especially Barry Boehm, for making all of this possible, and all of his USC Viterbi students (see the following list) who built CarmaCam and helped us understand parallel development by doing it.

Fall 2016 (proof of concept):

Rajat Verma, Preksha Gupta, Tapashi Talukdar, Zhongpeng Zhang, Chirayu Samarth, Ankita Aggarwal, Ankur Khemani, Parth Thakar, Longjie Li, Asmita Datar, Qingfeng Du, Maithri Purohit, Shobha Rani Dhalipathi, Seema Raman, and Sharath Mahendrath.

Spring 2017 (minimum viable product):

Shobha Rani Dhalipathi, Sharath Mahendranath, Ting Gong, Soumya Ravi, Namratha Lakshminaryan, Yuanfan Peng, Asmita Datar, Ragapriya Sivakumar, Yudan Lu, Ishwarya Iyer, Chuyuan Wang, and Jingyi Sun.

Fall 2017 (minimum viable product/optimization):

Shreyas Shankar, Akansha Aggarwal, Zhuang Tian, Yanbin Jiang, Jiayuan Shi, and Guannan Lu.

Spring 2018 (optimization):

Yue Dai, Yingzhe Zhang, Pengyu Chen, Haimeng Song, Jingwen Yin, Qifan Chen, Khushali Shah, Ying Chen, Shih-Chi Lin, Xiyan Hu, Yenju Lee, Basir Navab, Lingfei Fan, and Raksha Bysani.

Summer 2018 (optimization):

Yenju Lee, Lingfei Fan, and Haimeng Song.

Fall 2018 (optimization and machine learning proof of concept):

Akanksha Priya, Bowei Chen, Chetan Katiganare Siddaramappa, Chi-Syuan Lu, Chun-Ting Liu, Divya Jhurani, Hankun Yi, Hsuan-Hau Liu, Jienan Tang, Karan

Maheshwari, Nitika Tanwani, Pavleen Kaur, Ran He, Runyou Wang, Vaishnavi Patil, Vipin Rajan Varatharajan, Xiao Guo, Yanran Zhou, and Zilu Li.

Spring 2019 (optimization and machine learning minimum viable product):

Asmita Mitra, Chi Lin, Julius Yee, Kahlil Dozier, Kritika Patel, Luwei Ge, Nitika Tanwani, Pramod Samurdala, Shi-Chi Lin, Tiffany Kyu, Vaibhav Sarma, Zhao Yang, Zhengxi Xiao, Zilu Li, Chi-Syuan Lu, Bowei Chen, and Yanran Zhou.

Summer 2019 (optimization and machine learning minimum viable product):

Luwei Ge, Shi-Chi Lin, Zilu Li, Bowei Chen, Yanran Zhou, and Khushali Shah.

Contents

1 Parallel Agile Concepts 1
1.1 Partitioning a System for Parallel Development 1
1.2 Just Enough Planning 3
1.3 Feedback-Driven Management, Model-Driven Design 4
1.4 Risk Management 5
1.5 Project Management 6
1.6 Executable Domain Models 7
1.7 Parallel Agile Process 9
1.8 Scalability and Evolution of Parallel Agile from ICONIX Process 12
1.9 Summary 13
References 14

2 Inside Parallel Agile 15
2.1 Code First, Design Later 15
2.2 Prototyping as Requirements Exploration 16
2.3 Overview of the Parallel Agile Process 16
2.4 Inception 17
2.4.1 Evolving Database Schemas 17
2.4.2 Enabling Integration Between Developers 18
2.5 Parallel Development Proceeds in Three Phases After Inception 19
2.6 Proof of Concept (Building the Right System) 19
2.7 Minimum Viable Product (Building the System Right) 20
2.7.1 Using MVC Decomposition to Make Your Use Cases Less Ambiguous 21
2.7.2 Using Parts of Parallel Agile with Scrum 21
2.7.3 Adding Controllers to the Scrum Backlog 22
2.7.4 Tracking Agile Projects by Epic, User Story, and Task 23
2.8 Optimization and Acceptance Testing 23

2.9 Balancing Agility and Discipline 24
2.10 Summary 25
References 26

3 CodeBots: From Domain Model to Executable Architecture 27
3.1 Solving Problems to Enable Parallelism 28
3.1.1 Parallel Agile Versus Agile/ICONIX 29
3.1.2 Cost Benefits 30
3.1.3 Origin of Executable Architectures 30
3.1.4 Developer to Developer Integration 31
3.1.5 Resilience to Staff Turnover 31
3.2 Domain-Driven Prototyping 32
3.2.1 Introducing CodeBot 34
3.2.2 What Is Prototyping? 35
3.2.3 CarmaCam Domain Modeling Workshop 36
3.2.4 The Prototyping Is Done—What's Next? 45
3.3 Using CodeBot During the MVP Phase 46
3.3.1 What to Expect from Your UML Relationships 47
3.4 Deployment Architecture Blueprints (Preview) 49
3.5 Summary 49

4 Parallel Agile by Example: CarmaCam 53
4.1 CarmaCam Architecture 55
4.2 Sprint 1: Proof of Concept 56
4.2.1 Executable Domain Model 59
4.2.2 Use Cases and Storyboards 60
4.2.3 Prototypes 62
4.3 Sprint 2: Minimum Viable Product 68
4.3.1 MVC Decomposition 68
4.3.2 State Transition Diagram 70
4.3.3 Testing 70
4.3.4 Server Migration to AWS 74
4.4 Sprint 3: Optimization 74
4.4.1 Web App 75
4.4.2 Mobile App 76
4.4.3 Emergency Alert Receiver App 77
4.4.4 Server App 79
4.5 What's Next? 79
4.6 Summary 80

5 Taking the Scream Out of Scrum 81
5.1 Agile Mindset 82
5.1.1 General Agile Mindset Misconceptions 85
5.1.2 The Sweet Spot of Parallel Agile in the Agile Mindset 85
5.1.3 Scaled Agile Framework and Parallel Agile 86

5.2 Scrum as It Should Be: A Quick Overview . . . 87
5.2.1 Misconceptions About Scrum Roles . . . 89
5.2.2 Misconceptions About Scrum Events . . . 90
5.2.3 Misconceptions About Scrum Artifacts . . . 93
5.3 Example: Parallel Agile with Backlogs and a Small Team . . . 95
5.3.1 From Product Vision to Product Delivery . . . 96
5.3.2 Preparing the Product Backlog. . . . 96
5.3.3 Sprint Planning. . . . 98
5.3.4 Sprint . . . 102
5.3.5 Sprint Review. . . . 104
5.4 Summary . . . 105
References. . . . 106

6 Test Early, Test Often . . . 107
6.1 A Note About Software Quality. . . . 109
6.2 Errors of Commission vs. Errors of Omission. . . . 110
6.3 Acceptance Testing Fails When Edge Cases Are Missed . . . 112
6.4 CarmaCam Example . . . 113
6.5 Testing at Different Levels. . . . 114
6.6 A Capsule Summary of Domain Oriented Testing. . . . 115
6.7 Drive Unit Tests from the Code . . . 117
6.8 Drive Component Tests from the Use Case Scenarios. . . . 118
6.8.1 When Should Component Tests Be Written?. . . . 119
6.9 Drive Acceptance Tests from the User Stories and Use Case Scenarios . . . 120
6.9.1 Who Is Responsible for the Acceptance Tests? . . . 121
6.9.2 Manual vs. Automated Acceptance Tests. . . . 122
6.9.3 Acceptance Test Early and Often. . . . 123
6.9.4 Creating a Manual Acceptance Test Script . . . 124
6.9.5 Creating an Automated Acceptance Test with BDD and Cucumber . . . 125
6.9.6 Creating an Automated Acceptance Test with DDT . . . 126
6.9.7 Prototype to Discover Requirements, Review Them Carefully, and Test Each One. . . . 127
6.10 Summary . . . 129
References. . . . 130

7 Managing Parallelism: Faster Delivery, Fewer Defects, Lower Cost. . . . 131
7.1 Believe in Parallelism: Resistance Is Futile. . . . 133
7.1.1 Multicore CPUs . . . 135
7.1.2 Elastic Cloud of Developers. . . . 135
7.1.3 You Want It WHEN? . . . 137
7.2 Managing Parallelism: Instant Tactical Assessment . . . 138
7.2.1 Task Estimation from Sprint Plans. . . . 140
7.2.2 Rapid and Adaptive Planning. . . . 140

7.2.3 Seizing the Initiative. 141
7.2.4 Analyzing Use Case Complexity 143
7.2.5 Redeploying Resources as the Situation Evolves 145
7.2.6 Accelerating a Late Project by Adding Developers 145
7.3 Improving Quality While Going Faster 147
7.3.1 Generating Database Access Code. 147
7.3.2 Generating Acceptance Tests from Use Cases 147
7.3.3 Testing in Parallel with Developing 148
7.3.4 Hurry, but Don't Rush 149
7.4 Lowering Development Cost 150
7.4.1 Doing "Just Enough" Design Reduces Costs. 150
7.4.2 Combining Planning with Feedback 151
7.5 Summary 151
References. 152

8 Large-Scale Parallel Development. 153
8.1 Parallel Agile and the Incremental Commitment Spiral Model 155
8.1.1 ICSM Phases Map to Proof of Concept, Minimum Viable Product, and Optimization 156
8.1.2 ICSM Spiral Includes Prototyping, Design, and Acceptance Testing 156
8.1.3 ICSM Principles Were Developed on Very Large Projects. 158
8.2 Parallel Agile Critical Success Factors. 158
8.3 TRW-USAF/ESC CCPDS-R Parallel Development 160
8.3.1 CCPDS-R Evidence-Based Decision Milestones 161
8.3.2 CCPDS-R Parallel Development 162
8.4 Parallel Development of Even Larger TRW Defense Software Systems 162
8.5 Comparing the Parallel Agile Approach and CSFs and Other Successful Scalable Agile Approaches 164
8.5.1 The Speed, Data, and Ecosystems Approach. 165
8.5.2 The Scaled Agile Framework (SAFe) Approach 167
8.6 Conclusions and Latest Parallel Agile Scalability Experience 167
8.7 Summary 168
References. 168

9 Parallel Agile for Machine Learning. 169
9.1 Phase 1: Proof of Concept and Initial Sprint Plan 170
9.1.1 CarmaCam Incident Reports 172
9.1.2 CarmaCam Emergency Alert Videos 173
9.1.3 Identifying Multiple Lane Changes at High Speed 175
9.1.4 Training Machine Learning Models from CarmaCam Videos. 177

9.1.5 Training Machine Learning Models Using Video Games 179
9.1.6 Phase 1 Results and Summary 180
9.2 Phase 2: Minimum Viable Product – Detecting a Likely DUI from Video 182
9.3 Summary 184
Reference 184

Appendix A: The Scream Guide 185

Appendix B: Architecture Blueprints 205

References 215

Index 217

Chapter 1
Parallel Agile Concepts

Parallel Agile (PA) is a process that allows software schedules to be radically compressed by scaling the number of developers who work on a system rather than stretching the project schedule. In this chapter, we'll discuss the main concepts involved in PA, including partitioning for parallel development, planning, management and design models, risk mitigation, project management, executable domain models, process, and finally the evolution of PA from ICONIX.

1.1 Partitioning a System for Parallel Development

Parallel development requires careful partitioning of a problem into units that can be developed independently. In PA, you decompose systems along use case boundaries for this purpose. Parallelism in development is achieved by partitioning a project into its use cases. Each developer is assigned a use case and is responsible for everything from operational concept through working code for his or her assignment.

Figures 1.1 and 1.2 show the essence of how to develop software in parallel. In Fig. 1.1, the system is decomposed along scenario boundaries. Figure 1.2 shows the spiral diagram for each use case: build the right system, then build the system right, and then test what you built. To the extent that you can do these three activities in parallel, you can compress the schedule of a project.

The spirals shown in the preceding figures are basically "uncertainty reducing" or "disambiguation" spirals. To make a software system work, you have to move from a state of complete uncertainty (speculation) to executable code. PA attacks uncertainty one use case at a time using storyboards, UML models, and prototypes, as appropriate for each use case. After you set up your project for parallelism, each use case progresses along the spiral in parallel at its own pace.

Within each use case, PA follows a standard set of steps, including a complete sunny-day/rainy-day description for all use cases, as confronting rainy-day scenarios early adds resilience to software designs. Each use case is then "disambiguated"

© Springer Nature Switzerland AG 2020

D. Rosenberg et al., *Parallel Agile – faster delivery, fewer defects, lower cost*,
https://doi.org/10.1007/978-3-030-30701-1_1

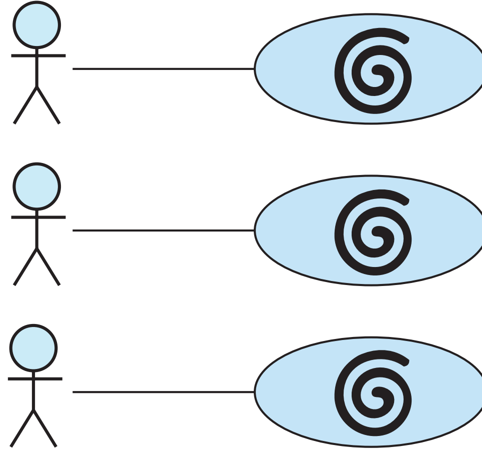

Fig. 1.1 Decomposing along use case boundaries enables parallel development

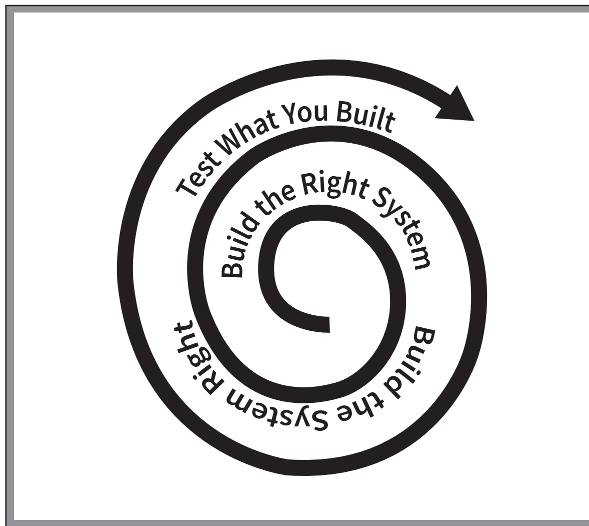

Fig. 1.2 Disambiguation spiral

using a conceptual model-view-controller (MVC) description, and then carefully designed. Typically, designs are shown on sequence diagrams; however, you could use test driven development (TDD) as an alternate detailed design process with some sacrifice of productivity. Requirements are modeled and allocated to use cases, and traceability between requirements and design is verified during design reviews, further increasing resilience of the software.

Having a small, standardized set of design steps and artifacts facilitates communication among team members who are working in parallel and also makes developers more interchangeable. One of the characteristics of student projects that run across multiple semesters is nearly 100% staff turnover every 3 months, since students typically don't take the same class over multiple semesters. Significantly, this turnover rate has not caused a problem on our student projects. Use of the UML model is a key reason that our projects have succeeded despite this turnover rate.

1.2 Just Enough Planning

Given the dramatic schedule acceleration that's possible by leveraging parallelism, and the never-ending quest to deliver software rapidly (if you're not publishing every 11 seconds, you're so last millennium), you might wonder why there haven't been more attempts to "go parallel" in software engineering. A large part of the answer relates to planning and management. Simply put, a parallel processing approach to software engineering requires careful planning and good management. But planning has been out of fashion in software engineering since the release of the Agile Manifesto, which explicitly values *responding to change over following a plan*. Since management's role often involves making sure a plan is being followed, devaluing planning simultaneously devalues management. Software processes can be rated on a formality scale ranging from *feedback-driven* (less formal) to *plan-driven* (more formal). In his book *Balancing Agility and Discipline* (Boehm and Turner 2003), Barry makes the case that for most systems the extremes on either side of this scale are expensive, and there is a cost-minimum somewhere in the middle (see Fig. 1.3).

The net effect of the Agile Manifesto has been that most Agile projects have a tendency to operate on the underplanned side of this feedback vs. planning continuum, with the classic example being the eXtreme Programming mantra of "Do the simplest thing that can possibly work" (DTSTTCPW). DTSTTCPW represents underplanning in its most eXtreme form.

In *Agile Development with ICONIX Process* (Rosenberg et al. 2005), Doug makes the case that "just enough planning" in the form of a minimalist, use case–driven approach gets close to this cost minimum and, in fact, PA has its roots in Agile/ICONIX, as we discuss later in this chapter. PA strikes a balance between plan-driven and feedback-driven development, with UML modeling used for planning and prototyping used for feedback.

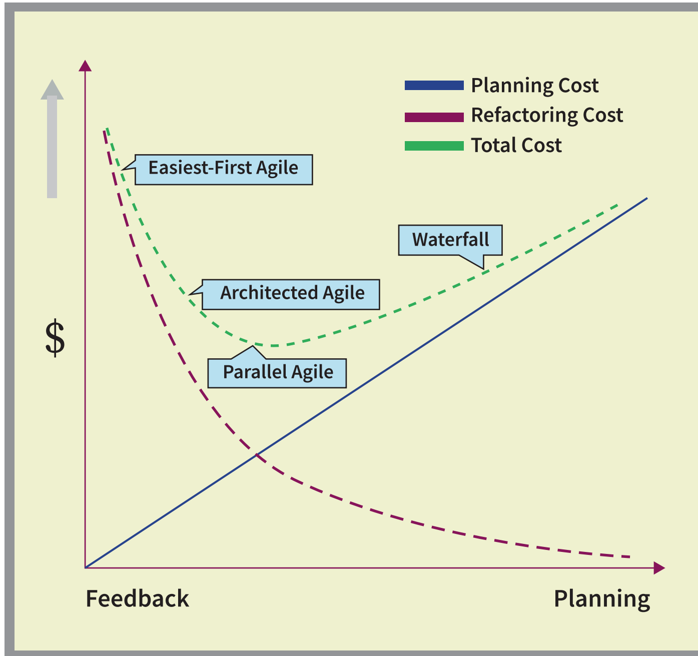

Fig. 1.3 How much planning is "just enough"?

1.3 Feedback-Driven Management, Model-Driven Design

Like most Agile methods, PA gets to code early and uses feedback from executable software to drive requirements and design. PA is feedback-driven on the management side (heavy use of prototyping) but model-driven (UML) on the design side.

PA uses technical prototyping as a risk-mitigation strategy, for user interface refinement, to help sanity-check requirements for feasibility, and to evaluate different technical architectures and technologies. PA prototypes help to discover requirements and are used to evolve the database schema, with developers prototyping various use cases in parallel against a live database.

Unlike many Agile methods, PA does not focus on design by refactoring, nor does it drive designs from unit tests. Instead, PA uses a minimalist UML-based design approach that starts out with a domain model to facilitate communication across the development team, and partitions the system along use case boundaries, which enables parallel development. PA emphasizes continuous acceptance testing (CAT) to a greater extent than unit testing.

Combining a model-driven design approach with extensive prototyping allows PA to be feedback-driven and plan-driven at the same time. Since both underplanning and overplanning are expensive, this blend of planning and feedback gets us near the cost minimum, as shown in Fig. 1.3.

The clear distinction between prototype code and production code is one of the big differentiators between PA and other agile approaches. In PA, production code is modeled, and acceptance test case scripts are generated from use case sunny/rainy day paths. This "build the right system" exercise takes a small upfront investment for each use case but results in dramatically less time spent refactoring code that was "the wrong system." By differentiating prototype code from production code, you're able to free the prototyping effort of some time-consuming tasks; notably, prototype code does not require extensive unit and regression testing.

Having a UML model allows PA to leverage automation to further accelerate development. Uniquely, PA uses UML modeling to assist with prototyping, enabling prototype code to interact with a live database by making domain models executable, using automatic code generation very early in the inception phase of the project. To be more specific, domain models are made executable by code generation of database collections, database access functions (create, read, update, delete, or CRUD, functions), and REST APIs.

Executable domain models allow developers to write prototype code against a live database in parallel sandboxes, which enables evolutionary feedback-driven database schema development. Executable domain models also assist with integration by using code generation to rapidly produce microservice architectures in the form of NoSQL databases and REST APIs at the inception of the project.

1.4 Risk Management

The Incremental Commitment Spiral Model (ICSM) provides a general-purpose risk-management strategy for software projects, where development proceeds in phases with a commitment to the next phase only after evidence has been evaluated from the previous phase. PA maps nicely to the ICSM model, where the phases are proof of concept, minimum viable product (MVP), and optimization.

Our student projects have generally applied this strategy over several semesters, where the first semester involved building a proof of concept system, the second semester involved building an MVP version of the system, and the third semester involved optimization and performance tuning, leading to an optimization.

Each of the three phases emphasizes different development techniques, as shown in Fig. 1.4. For the proof of concept sprint, we used a mix of storyboards and prototypes that connected to a live database via executable domain models. For the MVP sprint, we did rigorous use case modeling covering sunny- and rainy-day scenarios and elaborated using MVC decomposition. For the optimization sprint, we had a heavy focus on acceptance testing.

We'll talk more about the ICSM and its risk management strategies in Chap. 8.