BEYOND AGILE

DELIVERING SUCCESSFUL SOFTWARE IN A POST-AGILE WORLD

JESSE TAYLER WITH ALEX CONE

For information, contact:
Beyond Agile
222 E. 34th Street, #1826
New York, NY 10016
http://beyondagilethebook.com/

Book design by Alan Barnett

Printed in the United States of America

CONTENTS

PREFACE

This book is for anyone with a stake or interest in the outcome of software production. It is meant to assist those tasked with the management and scheduling of software teams with accuracy and without requiring expertise in any one specific software discipline.

Beyond Agile begins where contemporary software process leaves off. It is a modern playbook or a guide to understanding how, and why, certain groups of software engineers are so highly effective while others are not. This difference is not due to the collective IQ of the engineering staff. This difference is merely a reflection of process execution.

> **"Design is a funny word. Some people think design means how it looks. But of course, if you dig deeper, it's really how it works."**
>
> *— Steve Jobs*

We'll examine how, in software, there is no relationship of complexity between the visible portion (interface) and the underlying design structure (implementation). In other words, if you were to show an engineer a picture of an interface and ask how long it would take to implement, there is no sensible answer; the question ignores the underlying nature of software. The relationship is moot, and this fact makes software particularly challenging for most people, technical and nontechnical alike.

Beyond Agile is derived from and focused upon projects that rely on object-oriented design and use structured, compiled source. However, these same practices can be readily applied to any software, even the outermost layers of scripted web interfaces. Beyond Agile is equally applicable regardless of the project management methodologies each software team adopts. It does not dictate how often teams should meet, communicate,

or even deploy finished work. Beyond Agile is agnostic of specialized technique, respecting the fact that each possesses its own rhythm and practicality.

Beyond Agile makes this expert knowledge visible and available to everyone because what is at stake is everything we thought we knew about the world of software.

Let us go, beyond agile.

INTRODUCTION

Software has become a part of life for nearly everybody on earth. From the logistics of shipping to the calculation of finance or simply the daily communication that makes our world go, the computer has become the center of it all.

Software is what makes a computer do all those things that we want it to do. We see it is the lifeblood of our modern world —part of nearly every business. Many of the world's most valuable enterprises make some of the world's most visible products and services entirely out of software.

Software construction however, is entirely invisible. The question arises, how does one determine the progress of software construction? How does one reliably estimate schedule, or guide software development toward greatest efficiency. And how does one measure something if it can't even be seen in the first place?

This book provides a way of observing the invisible. It is possible to detect the changes in behavior patterns happening across software construction, and to use these insights to assess schedule and measure quality of work with a significant degree of accuracy.

This process is informed by the ways software construction itself evolved, so let us start our story at the beginning.

IN THE BEGINNING THINGS GOT EXTREME

The term software was coined in the 1950's and the first University Degree in Computer Science was offered in the 1960's. At that time, pools of programmers would produce what was called a "cascade release" of completed software strictly held within a rigorously controlled process. Interface consisted of type-written text. There were no graphic designers, or artists of any sort, involved in software construction. Progress was not speedy, but reliability was the only critical concern. Mission-critical software

systems of sufficient complexity (e.g. NASA) continue to use this rigorously controlled process today.

As software became increasingly ingrained in the everyday world, software construction techniques evolved as well. Cascade gave way to the far more rapid *"Extreme-Programming"* era.

Extreme programming emphasized the value of pair-programming and fast moving release strategies. Extreme programming extolled the potential of small teams working collaboratively.

THE AGE OF AGILE

The emergence of the web and scripted programming necessitated that the software construction industry change once again. Scripted languages were brisk but also brittle and ignited a whole new era, known today as 'Agile'.

In the era of Agile, a team's ability to quickly adjust on the fly was considered the hallmark of effective management and an indication of achieving reliable foundations while avoiding unnecessary weight in process.

However, the reality is that Agile had limits in its applicability in the modern world. Software construction has become vastly more complex with the advent of mobile computers. Today, we combine mobile apps, reactive web and traditional server systems and expect them all to work together as a single, unified product or service. This multi-disciplinary environment generates a measure of complexity greater than the entire prior evolution of software construction combined. The range of languages and related disciplines used across teams today creates a new era of challenge for project management.

Those who do not see and adjust to the complexities inherent in today's software universe will continue to be increasingly off the mark.

BEYOND AGILE

Which brings us to beyond Agile, a methodology that is product of studying software engineering experts over decades. This methodology is shared among software projects of all shapes and sizes, spanning from high-energy startups to validated financial, clinical and regulatory controlled software systems. Teams that adopt this methodology are highly effective.

Every technology has experts who derive great value from specialized techniques. Upon close inspection, these experts are not only effective in a software language but are, perhaps unknowingly, arbiters of a sensible approach, because they are successful in ways that deliver advantageous results. This personal success however, does not necessarily translate across different types of teams and technologies. This success does not answer how to organize group initiatives or interdepartmental efforts into one coherent software methodology. This success, in of itself, does not help schedule or predict outcomes of software construction at all.

There is an inherent nature to software construction, one we can take advantage of to great effect. Beyond Agile breaks down this successful methodology into a set of properties that harness the power of software's true nature.

In Beyond Agile, we'll show how software construction generates cyclical waves of potential energy we can either ride or fight. This ebb and flow greatly effects every software project, whether it is recognized and understood or not. This "cycle-based" philosophy, is not about day-to-day tactics but rather a software construction process that is fundamentally successful.

BOWLING HAS A SEASON

Did you know that bowling has a season? It is an indoor sport! Now, why would they need a season?

Seasons usher changes in behavior and activity which naturally occur as we move through repeating cycles. This observation is the basis behind the extraordinary efficiency and organization of all professional sports, even those without regard to the elements.

In other words, those who are responsible for the effectiveness of The National Bowling League recognize what many software project managers do not. It is the schedule itself, the cycle of seasons that provides the scaffolding for order, and the assurance of predictability.

Every cycle has phases you can tap into and work in harmony with. Harnessing the energy generated during change of phase, can move you forward ever faster. Project managers can ignore this, but the flow of energy is a fact of all cycles, even ones without foliage to show for it.

Consider for a moment what experts in software engineering do with their time: they intuitively work within a repeating cycle that is responsible for predictable changes in habits and activities, just like the seasons of

the year. Let's take a closer look at what those phases are, and how their repetition is the recurring heartbeat of software construction.

Each software construction cycle can be divided into four distinct phases: **design, develop, debug** and **deploy.** You could call it spring, summer, winter and autumn if you prefer, the point is the same: changes in behaviors, habits and activities, both great and small, are dictated by the change of phase within cycles.

Let's break this down in further detail.

At the start of every software project, every release, or any segment of software work, regardless of its size—you'll find there are a lot of freeform ideas about how to approach problems. This is only natural. We call this *most collaborative* phase **Design.**

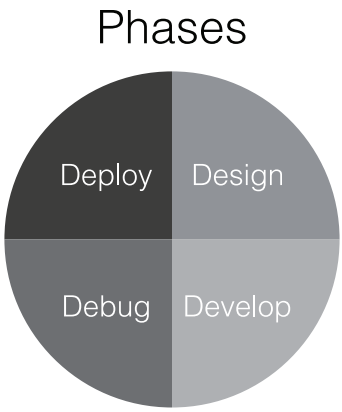


Figure 1. Phases. The four phases of the software development lifecycle.

There comes a time when these freeform ideas give way to stretches devoted to the bulk of the intended development. The so called **Develop** phase is often the longest and *least directed* in the software Cycle.

As timelines and milestones approach, unfinished work must be completed or deferred to prepare for a stable release. During this structured **Debug** phase, hit-lists of issues (even formally naming bugs if they are tenacious) get identified and attacked. This phase is often the *most directed* of any phase because of this rigorous triage approach and formality of process.

Finally, we enter a phase we simply refer to as **Deploy**. Controls and procedures take precedent. We follow the book. We use countdowns, checkoff-lists and procedural guides which have been predetermined for every step. This phase is typically the *least collaborative* precisely because of the required attention to studied and predicted situation handling.

It is through taking advantage of this change of phase, tapping into the flow of each Cycle and channeling the energy into action that makes development groups take off and fly.

THE DISCIPLINE OF PHASE

Cycles repeat.

Endless cyclical repetition provides for a tremendous efficiency in what we might refer to as *the discipline of phase.* Just like an army marching predictably in lock-step and turning in unison requires practice and study, working within software Cycles can help continuously improve every team's results and efficiency.

Each release cycle is like a winemaker's season. The outside world provides the rain and sun of that year. The best possible wine is made using a process of minimal mistakes. In software and in winemaking both, techniques exist to optimize natural potential and to minimize accidental degradation.

Emphasizing architecture early, during the Design phase of development pays dividends throughout each and every cycle. Adopting a cyclic methodology allows engineers and architects to reassess design integrity, with a relentless regularity.

The *discipline of phase* is essentially a way to help everyone get in sync. Visibility as to why we offer new requirements only at the start of the Cycle, and why we avoid distraction during other periods. This visibility can immediately provide a noticeable reduction in frustration, and offers improvement toward greater reliability in any construction.

We must learn to think of software construction not as a sprint to a goalpost, but rather as a train rolling down the track. Software has motion, it has momentum and direction.

Software in motion is a ceaseless act of balance.

F.I.D.E. LAWS OF CHESS 11.5

Cyclic disruption is like an athlete falling down in the 100 m dash. Quoting the F.I.D.E. Laws of Chess 11.5: *"It is forbidden to distract or annoy"*—this rule of can equally be applied to software engineers engaged in the Develop phase of the software Cycle.

Much of software construction requires deep analytical thinking. Like holding the pieces of a chessboard in your head, any distraction can disrupt.

In software there are times (typically during the development phase) when even a momentary interruption of a developer's work is like distracting a chess player during Tournament.

What is certain is that at times during any Cycle simply asking a software engineer to stop, stand up and answer a question can lead to a slow-down like falling down in the heat of software construction. By taking note of this fact as a guiding principle, we can reduce frustration and improve pace simply by knowing when to get out of the way.

NEW YORK CITY TAXI CAB DRIVERS

We can see the nature of software Cycles, but nature doesn't dictate software's power and influence.

New York City Taxi Cab Drivers are some of the most professional on earth, and if you've been to New York you know you'll find some of the fastest.

THERE ARE SPEED LIMITS

Imagine two people take a cab ride from the same point in the city. One person might get the best driver in all New York, while the other, the worst. The best might deliver sooner and maybe save some fare. Maybe the worst driver got stuck in traffic and you'd end up paying twice as much.

But it is not going to be one hundred times difference no matter which driver you get. Taxi cabs just can't go that fast, they are limited by the laws of physics.

Professional software engineers are not gated by the physical universe while professional Taxi drivers are. Computer scientists and their

effectiveness are left unencumbered by the laws that govern the world in which we live.

Consider that changes in context for software engineers can greatly affect results. When a professional taxi driver switches passengers and route, there is no delay or disorientation associated with that change. In software, this is not the case.

In our post-industrial worldview, we've been trained to think: adding more staff results in getting more done. We can readily see, each hour an employee is working is an hour's worth of production. We can see this if you were doing just about anything from writing blog articles to assembling widgets, even paving the sidewalk—more man-hours equate to more work done, and why wouldn't it?

WHAT IF WE COULD SEE SOFTWARE DESIGN?

In the real word, where construction is visible, there are clearly sensible ways of making things. In our everyday experience we don't see people building staircases into walls, or doors leading to nowhere. In our everyday world, if you think that on aggregate an hour worked is an hour of useful production, you'd be right. But in the unseen world of software, you'd be wrong.

The fact that software's design is invisible can serve to hide nonsensical structures. In software, you cannot see the walls nor the stairs. On the other hand, within the unseen universe of software there are no production speed limits as there are for professional Taxi drivers. It is this vast range of possibilities that makes software a challenge and a potentate.

The problem here just might be how to see what direction you are going in the first place. Hours worked, even tasks completed are not indicators of underlying software design or construction integrity, and therefore they are not reliable standards of progress.

This implies that some measures that are plainly sensible from the real-world perspective, are simply not valid for software construction.

This fact may seem counterintuitive, confusing or even frustrating.

The important thing to consider is that compared to real-world professions such as Taxi drivers, the effectiveness of professional software engineers has a far greater spectrum of performance.

Can this software phenomenon be harnessed? Can it be measured? Can it even be understood?

The answer is Yes.

AN ICEBERG LAYS 90% UNSEEN

We see invisible software construction is unlike any real-world product. Software is not limited by the laws of physics and this fact carries great consequence for project managers.

In the real world, we know that 90% of an iceberg remains unseen and underwater. This relationship is certain, it has been proven by science.

In software there is no relationship between the visible artifacts and the true underlying complexity of that system. This fact may sound obvious but it has profound implications for the ways we measure, manage and predict the outcome of software construction.

Examine industry-wide results of software teams as it relates to schedule and budget, and you'll see dishearteningly inaccurate predictions.

But why? What is it that project managers are not seeing?

THE OTHER 90%

This unseen "other 90%" of software is nearly always far greater in complexity compared to that which can be detected by the untrained eye.

This underwater portion of software goes far deeper down than the predictably submerged part of an iceberg. If you ignore this submerged part of software, then you are managing a tiny fraction of the overall construction.

Consider common software measurement tools and you'll see how easy it is to forget about this "other 90%". Software management tools and associated methodologies are after the parts one could identify; the visible artifacts, screens and wireframes—the features and functions that people perceive and interact with.

Project management tools track what gets reported. This is only natural. Project management tools are not so good at helping identify great software design initiatives. These tools do not accredit design, lifecycle or architecture and those are the utmost insights and contributions engineers make.

Unscientific reliance on these tools confuses activity with progress. Construction based on resulting artifacts alone will cut across the grains of software's true, layered structure and avoid the far larger, unseen critical imperative.

Design schemes are everywhere, inherent in the underlying system. Visible or not, these designs are the true levers effecting software construction. In the short and the long-term both, it is the design that is the most authoritative element of progress and order.

Design is the most potently influential aspect of software there is.

THE SOFTWARE EQUILIBRIUM

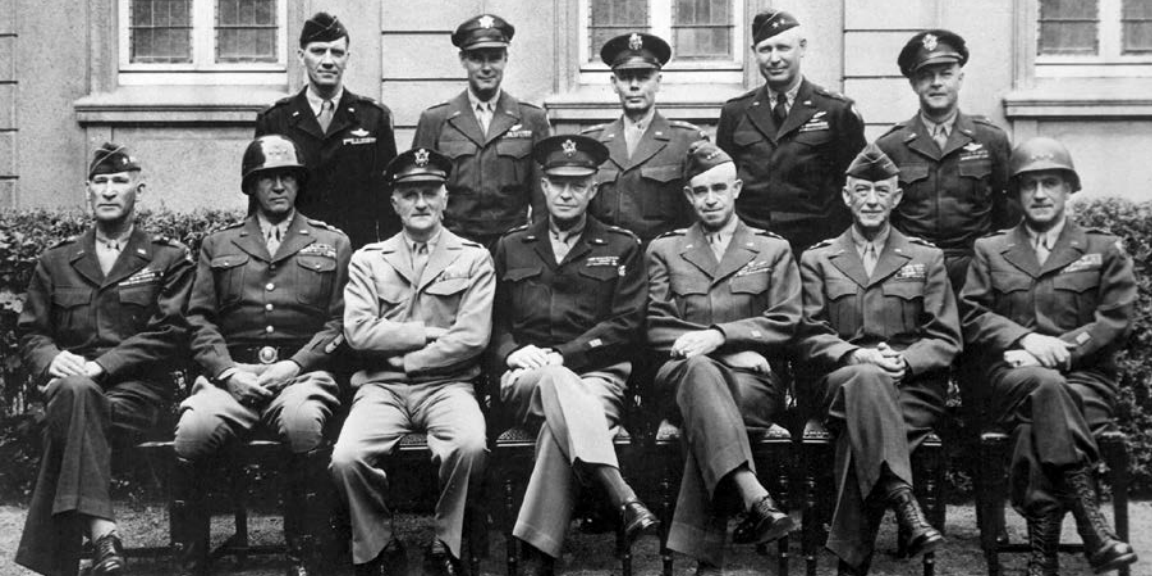
Software is always in motion, it remains in a balanced equilibrium—building up with new features and functions until it begins to collapse under its own weight. We fight back against this collapse using design. Design helps us to build to ever greater heights. This continuous balancing-act only ends when a software product is end-of-life and no longer supported.

Observing this continuous balance, this equilibrium between rise and collapse—we see clearly that architecture and design are what keep the building rising. Like a geodesic dome, design drives the reliability, agility and overall quality of software beyond simple function. Again, this is true in both the short and long term.

Unfortunately, design is often ignored in the quest to measure and pace progress altogether. Design is paramount, and yet project management tools have little or no facility to model and track what cannot be readily seen or reported.

Software is art. Does this mean that some elements of software are such artistic endeavors they are beyond the reach of reason or measure?

It certainly does not.



URGENT VS. IMPORTANT

We see that software design is critical, and visible artifacts of software do not relate to complexity.

Software can be accurately assessed and managed even at the largest of scales, but only if we divide between that which is most easily seen and that which is truly significant in terms of design integrity.

Visible defects, artifacts of logical flaws boiling down to bug reports are just an historic record of previous software management failure.

> **"Program testing can be used to show the presence of bugs, but never to show their absence!"**
>
> *— Edsger Dijkstra, Dutch Computer Scientist*

Bug reports sometimes represent omissions or logical error, even misunderstandings; but in the software development process, the most important bug reports represent failure of design. Design flaws are what

should have been predicted and tested against, but somehow got ignored until they were seen or sensed enough to report.

From this standpoint, the tracking of issue reporting and bug fixing is actually a distraction from the true underlying mechanism that is responsible for creating those flaws. Thus, the most important task during software production is to continually address matters of design.

FIXING BUGS IS NOT PROGRESS

We can see that fixing bugs is not a valid measure of progress, it can be more like a dog chasing its own tail. Even completing features is not a reliable indicator, as we still ignore the unseen design complexity underneath. So how do we evaluate that which cannot be seen?

Eisenhower Decision Matrix—Consider what must be done in quadrant form.

In software, urgent is what people see, while what is important is good design. Imagine all that goes into a project and consider what is visible and what is important—we can easily see where the popular choices are to be made.

Put in practical terms, to get a sense of how to evaluate what is most Urgent vs. most Important we could examine all that went into a previously completed release.

Consider all that each engineer and each team accomplished. All the designs and features, all of the bug reports—all that was done. These tasks and goals each can be divided into that which was most important vs. that which was really just urgent by using a quadrant matrix sometimes referred to as the "Eisenhower Decision Matrix".

When we perform this activity for software construction, we readily see why people avoid so much of what is important and focus on that which is most urgent. Urgent matters are the things that outside stakeholders can see, and take interest in.

Sadly, this also reveals why so many groups perform so very poorly. The invisible nature of software means nobody ever gets fired for chasing bugs, even if failure to emphasize design is the cause of those very same bugs.

By definition, bug chasing is reactive. In effect, it is the failure to emphasize design or testing that perpetuates a habit responsible for creating the very same bugs we reward engineers for fixing.

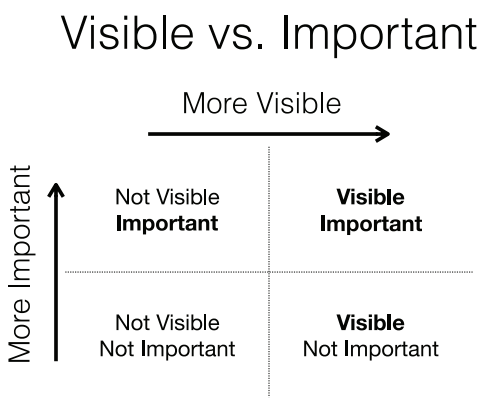


Figure 2. Visible vs. Important. The Eisenhower Decision Matrix.

This happens. It is like making a hero of a firefighter who carries a book of matches.

Can this trap be avoided? How can we emphasize that prevention is always better than cure?

The solution is to divide and conquer.

REFACTORING IS PART OF EVERYDAY LIFE

Refactoring is the process of restructuring existing code into a new implementation, that performs the same result. By definition, refactoring looks like a real waste of time. When done properly however, there is nothing that could be further from the truth.

Recognizing that software is in a constant state of flux and that design is the critical partner in creating a reliable structure, we have to accept that refactoring code and correctly evaluating the lifecycle of code is a major part of this everyday balancing act.

On the chart of Urgent vs. Important refactoring is clearly important, but it also carries risk. One cannot replace the retaining wall of a reservoir's dam and leave this construction partway complete. Once you embark on refactoring there is no turning back until the finish line has been crossed.

When does maintaining software outweigh its value, or when is progress better served by accepting the risk of refactoring outdated or ineffective designs?

How does one know when software needs refactoring if it can't even be seen?

WHAT IS THAT SMELL?

It has been said that code can have a bit of a smell to it. This effect may have first been noted by computer scientist Martin Fowler, a leading voice on object-oriented software design and software construction.

> **"a code smell is a surface indication that usually corresponds to a deeper problem in the system"**
>
> *— Martin Fowler, British Computer Scientist*

Code smell is just shorthand for the visible artifacts of hasty and disordered construction. This "smell" manifests itself in the structure of the code's writing, the seeming clarity, uniformity and overall organization of files and resources.

Code smell refers to the cracks that appear in the archways of the Cathedral.

These cracks are indications of underlying stress on design. Cracks tend to propagate and if left unchecked they result in catastrophic structural failure.

Refactoring takes many forms in many stages of the software lifecycle. Once we accept the influential value of refactoring, it is not hard to identify what must be refactored or what value that will provide.

If code has a bad smell or if the root cause of too many defect reports relate to unwanted, degraded and smelly designs then engineers will already be aware. Many times, these very elements will have been christened by the team with some snarky nickname simply due to their notoriety.

You won't have any troubling finding it, so why doesn't it get done?

Refactoring is never Urgent, you could say it is a thankless task. Furthermore, refactoring always carries some element of risk. It is not figuring out what should be refactored but rather having the courage to understand, evaluate and accept the risk involved.

Endless short term gains can be warmly received, but engaging in a policy of deferred maintenance has consequences. It can result in avoiding refactoring, which is plotting the very demise and collapse of your software Cathedral.

Stay vigilant, and don't bother second-guessing your expert designers and engineers, they already know what's causing the trouble.

NO BABY IN A MONTH

It has been rightly said one could lock nine highly collaborative women in a room, but they cannot produce a baby in a month. Babies simply do not divide into more manageable pieces.

Software cannot be arbitrarily split in any direction or be cut into any size or shape. Software has a grain to it, it has layers—each reliant upon the ones below.

From the Greek "atomos" or indivisible, an atom is the smallest particle with sufficient structure to provide material meaning. Similarly, there is a minimum viability that elements of software can be broken down into before they no longer provide any material structure.

This indivisible, atomic nature of software is real. This fact means we cannot neatly fit all software efforts into discreet segments of arbitrary size. It is not possible to divide software indefinitely, or cleave in unreasoned order.

This has profound effect on the approaches used to accelerate schedule.

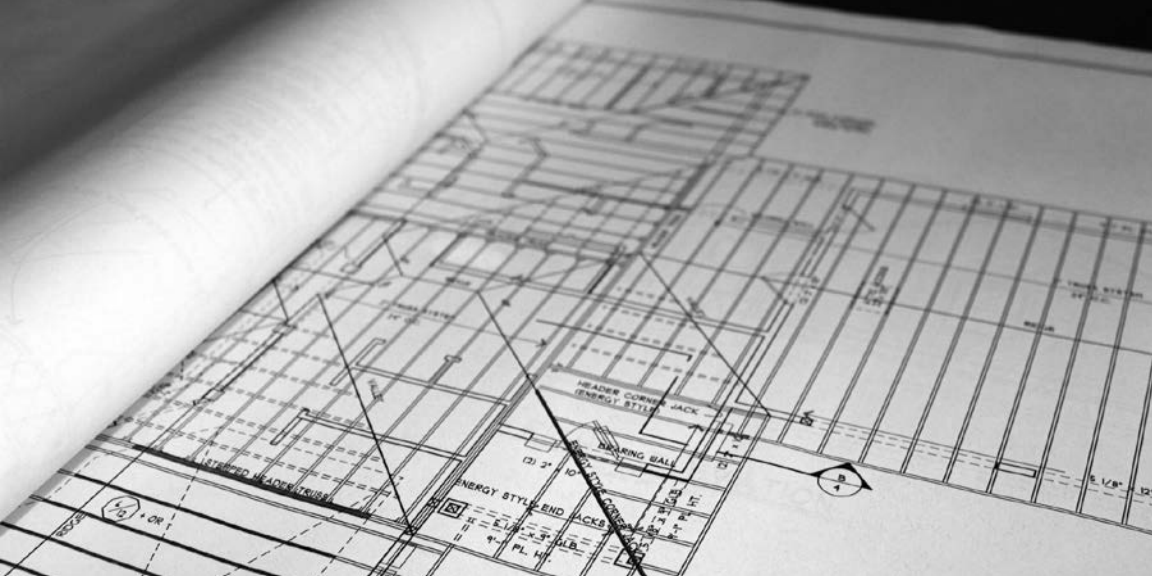
Some software work simply cannot be accelerated by division. Software cannot be divided such that one engineer works on the "save" button while another works on "cancel".

If one believes that any software segment can be broken down into a week-long checkpoint or divided between engineers for faster results—they would unknowingly be decapitating critical underlying structures in so doing.

In the real world, if we increase the number of workers on a construction site, we should rightly expect improvement of progress. This simple division of labor is a reality we see everywhere in our regular experience.

Since software cannot always be divided, it is implied that simply adding engineers to a mature project can sometimes, if not often times, actually impede progress.

Make no mistake about it. The implications of this simple fact of atomic division is the single most commonly and tragically confused reality of the software world.



DESIGN MATTERS

We know software design is not visible. This makes some of the most important work to be done, far-less attractive for us to attend. Even so, how do we separate out that which is truly important?

For the purposes of this text, the word **"Architectural"** refers to bugs or features where stability of design is critical, and complexity of function merits something of a blueprint. This word refers to complicated things—as opposed to simple software features or logical flaws and trivial misinterpretations.

Every project has what we might call **"Architectural Bug-Makers"** or flawed designs that are identified as the root cause behind different groups of defects. Architectural Bug-Makers are a drag on maintenance and an impediment to progress in ways that could have been avoided. These are software structures that just seem to fail their purpose, or fall short of the quality of design that is required of them.

Every project has what we might call **"Architectural Features"** or requirements of the system that represent new structures and functionality

necessitating significant design. These features are complex enough to not only require extended development, but their implementation governs potential reliability and agility of the resulting construct.

BUG-FIX APPROACHES CAUSE SOFTWARE ENROPY

By virtue of its nature, ad-hoc bug fixing results in a patchwork effect that degrades design into a more disordered state. Patchwork-style repair punches through the layers of software foundations, causing cracks and leaving behind some amount of distortion in their wake.

In this way, a culture rewarding unordered bug-fixing tactics actually results in more defects being produced. Furthermore, this culture contributes to software entropy which is the major force and key factor behind shortness of lifecycle. This duration of lifecycle is the currency value of your software assets and most critically, all of this lifecycle damage remains completely unseen to the untrained eye.

> **"Most software today is very much like an Egyptian pyramid with millions of bricks piled on top of each other, with no structural integrity, but just done by brute force and thousands of slaves."**
>
> *—Alan Kay, American Computer Scientist*

Simply by identifying the Architectural features which require design, and separating out the Architectural bugs that are produced by flaws of design, you have immediately helped every engineer on each and every team.

Architecture has even greater meaning in the invisible world of software than it does in the everyday world in which we live.

Consider that the "Arch" in Architecture is the difference between a Pyramid and a Cathedral. The Cathedral is made using a million times less stone and the difference is not materials, it is knowledge.

Unlike the real world that is limited by the laws of physics, Architecture and software invention can change the entire landscape of construction and profoundly alter outcome.

If your process only measures the end resulting bug report, you cannot see the architectural causes behind flaws, and thus you cannot accurately measure the quality of underlying design.

EVALUATING QUALITY DOES NOT REQUIRE READING CODE

Project managers should be experts in computer programming no doubt.

In the world of Beyond Agile, project managers can no longer be experts in all languages, technologies and techniques used to build a single product. Project managers are additionally burdened with the broader responsibility of schedules and communications and this makes specific personal engineering expertise less and less applicable.

The seemingly subjective art of software construction can be measured and evaluated for purposes of quality using science. We succeed at this evaluation by organizing and examining properly categorized bug reports, while keeping a finger on the pulse of the software development Cycle.

We must always begin with the design in mind.

BALANCE & SERIALIZE

Schedules are critical to software planning at any size or stage.

As teams grow and multiple disciplines emerge—schedules become the key to inter-departmental organization and reliably joining technical efforts into the broader release plan.

There is one overarching strategy behind accurate schedules and release projections in Beyond Agile. We refer to this process and strategy as "balance and serialize".

Although software is invisible, construction schedule is governed by a visibly identifiable phase.

In other words, phases can be identified by taking note of the types of activities in which each team is currently engaged, and we can use this information to know where in the development Cycle any team or project truly is.

Because Cycles repeat phases in order, we can determine the boundaries of our schedule as surely as predicting spring follows winter.

BALANCE FIRST

A well-designed release plan focuses on only a handful of major architectural issues for any given team or group. These are either previously identified bug-makers or creations of new features that require architectural consideration.

It is not practical to tackle too much design for any given release, and thus there is always a balance between quality and progress which is itself, bound by schedule and resource.

This is balance.

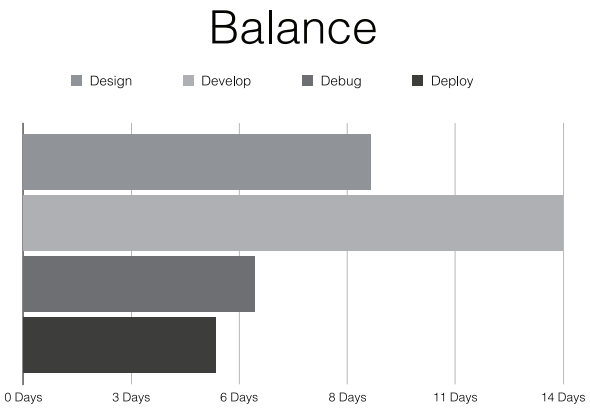


Figure 3. Balance. Compare and estimate quickly and accurately.

Any product already in use, will have a list of issues reported from the production system. These bugs must also be separated into major Architectural elements or simple logical flaws and misunderstandings. This task is best performed by the leading software architect of that system.

Within the methodology of Beyond Agile, even a minor release can pile on many simple logical fixes, however major Architectural elements are separated at birth and given a different life.

TICKET TO DEFER

Logical flaws and bugs being tracked must be ordered with an eye to that which we might call deferment.

The choice to defer bug fixes, is critical to efficiency.

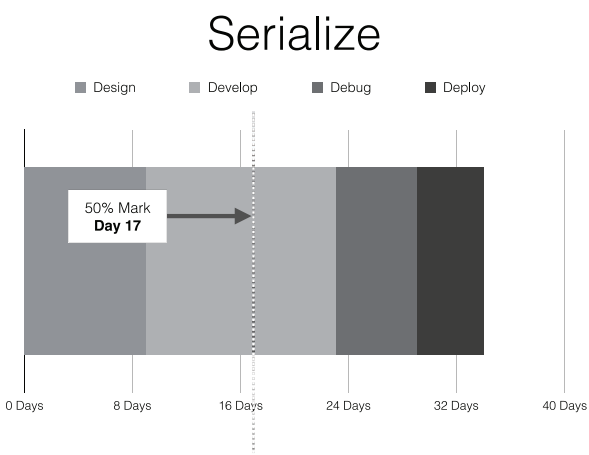


Figure 4. Serialize. Visualize the balanced values as a serial schedule that can contract and expand without loss of integrity of phase.

Consider that we've noted the impact of design and that we can only address a few Architectural issues in any given Cycle. This knowledge greatly impacts the order in which bugs are most effectively scheduled.

If a future release is intended to address underlying design problems in a "bug-maker" element of the software, we may well fix those issues all at once.

Bugs having a root cause relating to Architectural design should be part of that designer's scope of work when that scheduled time arrives.

Handily, each group of related Architectural bug reports becomes a set of proofs that provide a useful unit-test for that one software element.

It is only sensible to line up bug reports in ways that are most efficient. By quickly identifying Architectural elements, we can easily see the most effective ordering of fixes. This practice is at the very foundation of highly effective scheduling promoted in Beyond Agile.

Balancing any release or milestone is as simple as asking *"how different is this phase from the previous build? Is there a lot more to design? Is this release going to need more debugging or less?"*—If we divide and compare by phase, we can estimate and serialize quickly and accurately.

Relying upon this practice of "balance and serialize", the schedule can contract and expand with internal or external pressures. We can accordion the length of all phases at once and without disrupting the integrity of the Cycle as a whole.

Balance is by design.

ARE WE ON TIME YET?

Software design has unlimited potential to affect results. We recognize that software design is not visible, and further that software does not adhere to real world physical limits and measures.

Are we on time? How could anybody say firsthand, if software can't even be seen?

The measures and methodology revealing this answer are not as complicated as they are misunderstood.

There are ways to abstract the essence of expert software process. Exploiting this does not require a project manager to be an expert in any specific computer language, nor does it require them to be the architect and designer.

In other words, it is possible to put a finger on the pulse of software construction from outside of the box.

It is possible to adopt a mechanical methodology, and by mechanical we mean an *unthinking* process—one that axiomatically reveals the invisible nature of software construction regardless of computer languages or specific team tactics.

THE CURRENT PHASE IS BELLWETHER

Each phase of the software Cycle is identifiable by taking note of the activities in which each team is engaged. In this way, it becomes possible to determine which phase any given project truly is. This fact is what indicates the boundaries of the schedule's accuracy.

There is a rule of thumb in software scheduling implying that when software feels about 80% complete then you are likely to be at 50% of time, or halfway through your schedule.

Using this rule of thumb as a checkpoint, along with a diligent watch on critical design-flaw related bugs we can reliably assess on-time status at a distance.

Let's walk through this process in detail.

A SIMPLE RULE OF THUMB

One interested party or another may inquire as to whether a release remains on-schedule, just about every day. However, there are only a few select moments in the schedule for which this response ever truly changes. There are even fewer leading-indicators used to identify the responsible software signposts that dictate any change in said response.

> **"If debugging is the process of removing software bugs, then programming must be the process of putting them in."**
>
> *— Edsger Dijkstra, Dutch Computer Scientist*

Architectural flaws are impactful. Architectural flaws often form the single point of failure behind any number of bug reports. These bugs are the critical issues affecting the integrity of release.

By tackling Architectural design issues early in the Cycle, we benefit from a Debug phase focused on logical flaws and not the moving targets of incomplete or unstable design foundations.

Consider this rule of thumb we've cited, when trying to ascertain schedule. It is common to feel about "80% complete" at or around the halfway point of the Cycle. Now compare this sense of progress, with the detection of critical defects relating to design, still failing test.

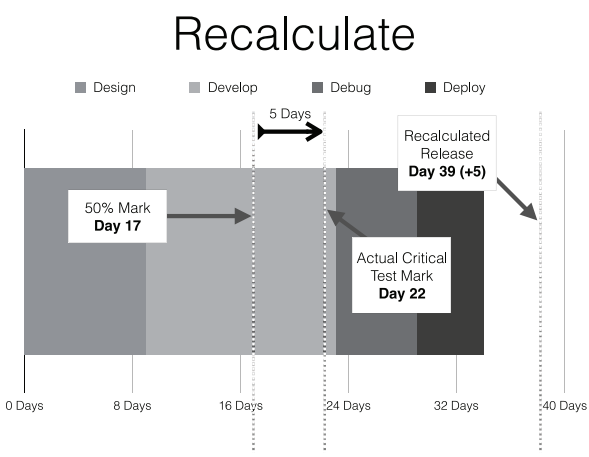


Figure 5. Recalculate. Schedules can be estimated as reliably as predicting sunrise.

The debug phase is not truly the current phase in earnest, at least not until architectural level design flaws are proven reliable.

If a team is attempting to repair something already defined as being complex enough to require design, something Architectural —then you have not truly phased into Debug activities. You are behind in schedule and this is predictable at the halfway point in schedule.

As we've noted, one cannot leave a reservoir's retaining wall incomplete. This remaining critical flaw must pass test before we are able to change phase and move on. Although, some amount of time can be made up of course, the schedule will begin to push out with each passing day.

Importantly, while architectural design flaws persist, one cannot be certain in any reassessment or recalculation of schedule. This fact is due to there being no way to accurately anticipate when architectural flaws will be re-designed such that the result will be proven reliable in testing. Passing these last critical tests is now the goal engineers are most focused upon, and that goal being achieved, will be the next time any schedule response can be accurately recalculated.

Conversely, once Architectural issues are proven reliable the release is considered "downhill" after that 50% halfway schedule mark. This is because even a large order of logical flaws could be either repaired or deferred without disruption of phase or loss of release integrity. This would be the first time a project manager could predict with confidence that the schedule will be met.

To sum up:

1. there are pre-defined milestones set into each Cycle
2. these milestones dictate when estimates can and should be made
3. there are times when we cannot accurately update schedule
4. there are times we must update schedule because milestones were met

Materially, one does not have to read code to accurately schedule and predict software construction. Having a finger on the pulse of construction while adopting a process that identifies Architectural elements, is the key to highly reliable schedule management.

There are only a handful of moments this response is ever even in question.

Adopting this method makes accuracy in responding to schedule as calmly dependable as predicting the rising sun.

TESTING

Testing is a technical vocation, performed by professional computer scientists throughout the software lifecycle.

Testing is assurance. It is the National Transportation Safety Board (NTSB) of software construction. It is there to prevent accidents, and when they happen to determine probable causes and formulate carefully crafted safety recommendations that are based on science.

Testing, by using objective measures and assessments, assures that software successfully conforms to its requirements. Testing provides a bedrock of assurance and reliability which are the underpinnings of responsive and agile construction.

Testing can also grind software construction to a halt.

The potential to shut down progress is real. Testing has an extremely broad set of professional techniques, each evolving within dramatically different environments and sharing only distantly related evolutionary pressures and influences. This breadth of technique is testament to the incredible spectrum of software application, and the maturity of modern software construction practice.

Put simply, there are many more ways to apply the wrong technique than there are ways to correctly assemble a viable Testing strategy.

HOW ABOUT SENSIBLE TESTING?

In the view of Beyond Agile, Testing is a profession, a vocation of computer science.

We acknowledge the modern multi-disciplinary world of today requires a set of tools whose details lay outside the scope of this text. This material attempts to offer from the project manager's perspective—a set of ways and means to evaluate and optimize testing with economic precision.

If your plan is to reduce bugs by increasing the number of testers, you may want to think again.

There are ways in which science can evaluate the efficacy of Testing. Visibility can be given to the results and merits of Testing, but this is only useful if we educate ourselves on the deeper purpose.

Many popular project management philosophies revolve around suggesting dedicated testing staff or perhaps a certain ratio of testers to engineers. Some emphasize continuous proofs to be coded alongside business logic and still others will propose regression, integration or unit-test approaches.

These philosophies are each in response to a changing landscape of problems arising with new software construction strategies and evolving computer languages for more than fifty years.

In the mobile-centric world of today, full of many different stakeholders, it is not surprising that there are hordes of divergent testing interests. Some have reliability concerns and may want unit, stress or performance testing, while interface related responsibilities focus on results from A/B release reports or usability and acceptance.

How could applying any particular technique always be sensible?

TESTING IS OCCUPATIONAL SAFETY

Consider broadly the object of Testing. Suitable testing is coming from engineers, not at them. Testing tools and techniques exist to keep engineers performing software construction at the height of efficiency and with the greatest effect.

This may sound obvious, but even a cursory evaluation of most Testing processes reveals over or under testing, and perhaps even more often an ineffective or misconstrued Testing process that slows production without improving quality at all.

Bugs in production systems represent failures of Testing, not to be confused with the need for more testing. Testing is part of any practice intended to avoid bugs, it is a powerful weapon every engineer uses, but this does not make it causally connected to defect rates.

Performance testing, A/B testing, exploratory and scripted, regression or human interface testing—quite literally all testing techniques are fundamentally occupational safety.

Testing is what allows software architects to confront complexity with a clarity and composure that comes only from a sense of security. Testing is the NTSB of software construction, providing the guidance and support to operate free from anxiety—but safety is not without cost.

DON'T GROUND ALL AIRPLANES

Think of Testing as the underwriter's insurance of your project. Underwriting is the business of determining risk potential and quantifying the results of safety procedures.

If the one-and-only concern of the NTSB were safety, the investigative agency would simply ground all airplanes and close all roads.

Testing must consider the economy in which it operates.

We see that a proper safety net enables software construction to be faster, safer and more reliable. But this safety net can impede progress. Uneducated and unscientific testing is a recipe for creating friction and economic imbalance.

Proper testing is achieved by appropriately allocating resources to ensure engineers perform their mission in the most cost-effective and friction-free manner possible.

In the view of Beyond Agile, it is certainly not the quantity of the testing you engage in that provides safety, it is the process of learning how determine probable cause and finding its most fitting solution.

Testing is safety, sensibly.

ANATOMY OF A RELEASE PLAN

Software is planning, and with Beyond Agile we attempt to reveal the true value and nature of expert planning.

If you have a release, you have a plan.

Communication regarding this plan can be condensed into its most rudimentary elements. These elements, if made visible and accountable in any form or fashion, can help the project manager optimize individual, team and inter-departmental efficiencies.

Communication is paramount. It is communication that gives others a framework to engage effectively. For instance, visibility of schedule makes it clear to see that requirements are best provided early in the Cycle, during design. Those needing to review a near-final release can see when along the way this will happen. Each individual engineer, tester or stakeholder can distinguish their own part within the process, and plan accordingly.

THE SEVEN RUDIMENTARY ELEMENTS OF COMMUNICATION

Beyond Agile highlights the essential activities performed by the most effective teams. These groups all share certain practices of communication and documentation; a continuous process of providing visibility and dissemination. This communication is the very essence of professional software methodology.

Beyond Agile has condensed essential communication to seven rudimentary elements. These seven elements are common to any software construction playbook.

These seven elements are as follows:

1. a central authority to govern approach, a **style guide**
2. each Cycle has a **release plan,** give it a memorable name
3. **balance and serialize**—a strategy of schedule ownership and visibility
4. **release notes** discuss what changed from the original plan
5. candidate **release docket** lists when and why release candidates were rejected
6. **post-mortem** outlines what went well, and what can be improved
7. a diary of dated log entries or **lineage-report** contains historic events of merit

Let us review each of these elements in greater detail.

Style guide: even the smallest project group needs to agree upon, and set forth a style guide to establish a shared project-level uniformity. This is almost like a mission statement for coding practice. This guide is always work in progress and typically begins the very first time there are multiple disciplines and engineers working on the same project.

Release plan: is predicated on a memorable and suggestive title or release name, along with a simplified set of instructions describing the release. A well-balanced release has only one or two major Architectural features or fixes, plus a reasonable amount of logical flaws for repair, and of course some amount of time to accomplish it. A release plan lists major visible features for those outside the group and provides some "rules of engagement" directing team members to work within specified bounds (typically limits on technologies and approaches or guidance regarding architecture and design).

Balance and serialize: is a process that serves as a framework and guide to help estimate and visualize time needed for each phase in the software development Cycle. This is a collaborative effort within the team and requires input, advice and shared consent among members. Balance and serialize helps visualize the structure of each Cycle and get everyone to better understand their place within the goals of that schedule. This can take the formality of a Gantt chart or a casual whiteboard drawing of lines and dates. The purpose is to "see" the shared agreement of values, and the estimates uniting all efforts.

Release notes: created and catalogued for each release. This document outlines major feature elements and provides a detailed list of items differing from their stated or implied intent. These technical details help communicate and catalog how projections in the plan were later realized.

Release docket: when the time comes to produce a final production release, a release candidate makes its way through various stakeholders of the organization who typically offer a "go, no go" countdown-style response. We record the dates and reasons for any rejections in a **release docket.** This docket provides invaluable historic data that inform future cycles, and it provides critical visibility. It shows patterns of rejection along with any flaws in design or process that are responsible for those patterns.

Post-mortem: after a release has hit the shelves and is in real world use, it becomes time to regroup and consider what could have been done better. We refer to this as a post-mortem of each release or Cycle, and it is a major chapter of the playbook of continuous improvement for software construction.

Lineage-report: logging significant internal and external events in a software diary or **lineage-report** can give an overview of your entire software lifecycle. This reference increases in value the longer and more visibly it is maintained.

For each of these seven elements of communication, it is the visibility and authenticity that make them invaluable. Visibility is achieved by regularly updating a standard set of documents people can see and use. Authenticity is achieved with the shared ownership and input from each team and each individual, and by precisely informing stakeholders at each step along the process.

Software is a plan, make it visible.

"SOFTWARE CAPITAL" THE PURSUIT OF QUALITY

We see how cyclic scheduling fosters team spirit and shared responsibility. We appreciate the importance of design, documentation and visibility. We understand how sensible Testing is a bargain when economically balanced. This leaves one remaining matter: Software Quality.

But what exactly *is* Software Quality?

"Software Capital" is a term first coined by American Computer Scientist, Dean Zarras who cited software's astronomical capacity to provide broadly useful foundations. "The bottom line with Software Capital is better business solutions, delivered faster"—states Zarras, in his foresightful 1996 paper, republished by Hacker Noon, at https://hackernoon.com/software-capital-achievement-and-leverage-2c30f6f01ed9.

Quality can be viewed as available capital, an absence of bugs, or lack of technical debt perhaps, but this is not the whole story.

Quality is often seen, incorrectly, as a luxury. To some, Quality is what you sacrifice for the benefit of greater progress at lower cost.

The reality is far more subtle.

The most successful software development groups passionately rely on an organizational commitment to Quality. It is Quality which these groups use to precisely and accurately accelerate development, mature feature production and improve overall project performance. These groups take an interest in "Software Capital" and they expend resources in so doing.

How do these groups expend greater resource, while also expecting to outperform?

Let's take a step back and consider the very nature of Software Quality. At its core, *Quality is a pursuit*. Quality is both an activity and an investment.

Quality is the *will* behind extending ourselves and our effort.

Quality is the mortar holding the bricks of our Cathedral, the fabric that binds it all together.

It is this pursuit of Quality that provides the very energy we sense within Cycles. It is this flow of energy we tap into for effective turn of phase. Without this pursuit, your software construction engine runs dry.

Quality is a software concept which is embodied within the physical world. The building up of software assets, tackling of software deficit even agility itself —these are all results of Quality.

Excellent software developers must work to be architecturally mindful and efficiently detect, isolate, and resolve structural deficiencies before they manifest as visible artifacts.

It is this forward moving effort of design and enquiry, that requires fuel to persist.

The reasons behind this effect are largely self-evident. Like salt in soup, problems become more intractable as the lifecycle of the codebase matures. This results in ever greater effort being required for progress to be maintained.

> **"If we want to be serious about quality, it is time to get tired of finding bugs and start preventing their happening in the first place."**
>
> *— Alan Page, American Computer Scientist*

Unlike software itself, Quality is part of the natural energy system we, as human beings take part in. Quality is the force we sense pushing us through the change in phase.

In terms of Software Capital, it turns out this relentless pursuit of Quality is what really provides the inertia we're looking for. In a sense the best software is attainable only alongside the pursuit of Quality, and thus it is Quality in which we are investing our effort.

FORCES IN MOTION

External pressures from real-world requirements are what drive software construction. The call of urgency and the pressure of schedule are held in place by the power of Quality. This is the balanced equilibrium that we sense when Cycles are at their height of efficiency.

During software construction these three critical forces: *schedule, urgency* and *quality* mix together and combine. It is the balance between these forces which sets speed and trajectory. This mixture provides the fuel we use to advance our position and offers us the control we exert to navigate.

These laws regarding the software universe and its relationship to the real world, are the very heart and soul of software construction. Quality is both the lever, and the valve we use to direct construction with precision, symmetry and stability.

Lastly, Quality is something that everyday users can truly sense and appreciate. Engineers realize the production benefits of Quality, and ultimately your commercial stakeholders will appreciate the benefits as well.

"Quality. Up yours."

WHAT GOES AROUND, COMES AROUND

The iterative process is circular in nature, each phase receives input from, and provides output to, the following phase and ultimately carries into the next cycle. At the highest level, this circularity ensures relentless reassessment of structure. Teams act to advance Quality at each phase because every engineer has stake in the target goal, and the schedule they must work within to achieve it.

Above all else, because Cycles repeat and phases begin anew—all that was deferred, all that could be improved and all that was learned, can now be used to inform the next game, the next cycle, the next release.

When you think of cyclic software construction, you don't envision a sprint but rather see an opportunity for tapping into a natural flow of continuous energy. It is a way to recognize and appreciate the seasons happening all around you.

Take advantage of Cycles to foster deep acceptance of schedule and shared purpose. Promote the values that provide a guiding self-governance by means of Quality and sensible safety. It is this commitment to Quality that

inspires the best effort from even the most diverse set of computer science professionals.

Just as in professional sports, software excellence is achieved through focused discipline applied across seasons, practicing maneuvers over and over again. After each software construction Cycle, we can review that release in post-mortem. With each turn of phase, we can improve our collective marching pace.

Professionals refer to their playbook. Coaches adopt a repeating process of training and a discipline to continuously compare and improve. Beyond Agile defines the professional playbook and let's everyone benefit from this expert knowledge.

Like professional sports leaders, software project managers must improve continuously and plan relentlessly in order to remain at the top of their game.

OBSERVANCE OF CYCLES HAS ADDITIONAL BENEFIT

Balancing phases fosters a deep shared sense of responsibility within each team. The process of balancing is a collaborative act, requiring unanimous support and participation.

In this way, Cycle based goals and schedules are mutually understood as being valid and this authentic worthiness produces a natural self-governance in behavior. This sense of responsibility helps individuals recognize when they must be the one to step up, and without the need to directly manage or manipulate. Furthermore, this process creates an inspiring sense of ownership and a culture of sharing credit for accomplishment within every workgroup.

Cycles provide visibility into inter-team dependencies, providing greater accuracy in schedule overall. Cycles help engineers know when to consider design, when to halt development and emphasize testing. Cycles offer a framework for all individuals to know where they are in the bigger picture, and how to responsibly consider their own time and schedule within that framework.

Perhaps most importantly: identifying software Cycles helps everyone recognize and reward engineers for ***great design.*** Simply by adopting a practice of cyclical observance, teams can better highlight the design goals that allow the most effective teams to perform at their greatest heights.

We must go, Beyond Agile.

GLOSSARY

Agile. A commonly cited process that provides a popular structure for team activities. For the purposes of this material, the term Agile represents the status quo of modern tools and trends associated with software management as a whole.

App Store. The App Store drives the modern software economy. The web freely transfers data across the internet, while the App Store protects artists's digital rights online. These two software inventions are the pillars upon which the modern world of mobile computing stands.

Architectural. A term encapsulating software design and the intentions of the designer. In Beyond Agile, we submit that an Architectural element is defined as any new feature or any existing structure where simple logic gives way to a broader construction blueprint.

Beyond Agile. In the complicated world of mobile devices, scripted web programming and traditional server systems all working together—being 'Agile' is no longer enough. Beyond Agile is expert software knowledge refined into useful traffic signals and bellwether indicators revealing the true underlying nature of software construction.

Bit. A binary digit of information. The binary digit has only two states and the contraction of binary digit is Bit. A Bit is the smallest readable state with sufficient discrimination to convey anything at all, and is the basis of computer information.

Byte. A word or phrase of Bits ordered to make up a distinct segment of meaningful information. Typically 8-Bits make a single Binary-Term, or Byte of information.

Cascade. A word referring to the earliest software management practice, evolving from large pools of human computers. This rigorous process was characterized by mission-critical, and largely static system requirements to be crafted using only low-level computer languages. Cascade predates object-oriented theory in the enterprise.

Code. The instructions in a computer program. It is code that makes computer hardware perform arbitrary tasks without requirement of change in the material structure of the computing device.

Computer. Computer was a job title. People who calculate financial payments, or missile trajectories or any other form of mathematical computation. Since the 1950's or so, the term Computer refers to a non-thinking mechanical process that orders calculations on binary information stored in digital memory.

Cycle. A Cycle is any series of events that are regularly repeated in order. The Software Cycle is the heartbeat of software construction and is a natural, endemic part of that construction.

Extreme Programming. Extreme is a software development methodology intended to improve quality and responsiveness. Extreme evolved as a way to react to a world where requirements were no longer static, they were rapidly changing.

Internet. Different than the web, the internet is the underlying information protocol shared by early computer networks, thus unifying the world of communication between computers.

Phase. A distinct period or stage in any process. Software construction is cyclical in nature and the phases of development repeat predictably in ordered fashion.

Project Manager. A misnomer. We don't manage software but rather manage to take note of expert behaviors and leverage the energy and education afforded by repetition.

Software. The programatic instructions performed by a computing device upon information stored in memory. The first software was separated from the underlying hardware in research conducted by Ada Lovelace in the 1840's but the term "Software" was coined by John Tukey in the 1950s.

World Wide Web. Different than the internet, the web is built on top of internet protocol and provides the leading mechanism for freely transporting computer information.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the support and contributions to the Beyond Agile movement and all its independent contributors, mentors and thought leaders including the following:

Aaron Hillegass has spent his career refining the art of object oriented design and software theory. Aaron runs the Big Nerd Ranch providing enterprise level software training to the world's most advanced engineering groups. Established in 2001, Big Nerd Ranch specializes in expert instruction from the brightest developers and designers in the industry. Aaron got his start mastering NeXT computers in the early 1990's.

Andrew Stone "Jesse and Alex were there at the beginning of object oriented programming on the NeXT, which enabled collaborative software efforts to succeed. Learn from those who have managed teams and delivered software, baptized by fire!"—Andrew Stone. Andy famously created the Twittelator iOS app selling millions in the early iPhone AppStore. Andy has the distinction of being the first to distribute his wares on the first ever AppStore, The Electronic AppWrapper in the early 1990's.

In conclusion, the authors would like to humbly acknowledge the existence of:

Mom
The Flying Spaghetti Monster
Toilet Paper
$C_8H_{10}N_4O_2$

ABOUT THE AUTHORS

Jesse Tayler is a recognized expert in software systems. Jesse is uniquely suited for the complexities of today's mobile-oriented software construction. Jesse's hands-on leadership has kept him at the heart of software invention for over 30 years. Jesse creates responsible engineering cultures from the top. His personal faculties as a software engineer allowed him to excel as lead designer and central architect of validated financial systems, clinical and regulatory affairs systems requiring the highest degree of proficiency. Today, Jesse technology venture Object Enterprises Incorporated, builds mobile and server-side applications while providing training and leadership skills-enhancement for some of the largest advertising and publishing brands. Jesse still creates novel software inventions and advises young startups on business and technology.

Alex Cone is an entrepreneur computer geek with a penchant for starting software companies. Before he graduated from college he was writing software for NASA that helped build the Space Shuttle, then spent more than a decade building trading and risk management systems at top Wall Street firms. His latest enterprise consulting firm CodeFab was founded in 1997 with the intention of using better process to improve the success of mission critical software development projects. As CodeFab's Alpha Geek he was an early advocate for & adaptor of "Extreme Programming" (an early version of Agile Development) and continued to use agile techniques to deliver success in large scale mission critical software development projects. A lead architect for mobile development at B&N, HBO and IBM he has built an impressive track record for delivering cutting edge mobile applications and large interactive systems.