Grass Roots agile

By: Kirk Knoernschild

The only artifact that matters is quality source code!

Introduction

Software development is hard work. We have been taught that the best way to solve the tough challenges inherent to software development efforts is to treat software development as an engineering discipline. Stabilize requirements early, followed by implementation and verification. Yet we continue to fail.

We fail not because we do not try. We undergo complex process improvement efforts. We adopt iterative processes that propose attacking risk early. We create detailed artifacts that promise to increase awareness and understanding of the problem we are trying to solve. We attempt following detailed plans that will lead us toward the finish line. We establish steering committees and program managers. We hire teams of architects and designers. Yet we continue to fail.

Software development efforts fail because the traditional ceremonial approach to software development is fundamentally flawed. Worse yet, many adaptations of the most popular iterative and incremental processes are little more than reinventions of faulty practices resulting in slightly varied manifestations of the same problems that have plagued the software industry for years.

Who gets the blame for this mess? We, the humble developer. We are incompetent. Our estimates are wrong. Our code has bugs. The software doesn't satisfy requirements. For it is us who produces the only artifact upon which a software system is judged. And it is we who must push, fight, and claw for a better way. It is our responsibility to lead the charge, and initiate the transition from the prescriptive and plan-driven processes of yesterday to the adaptive and emergent practices required of today's software development effort.

agile is Good

Yes, I'm talking about agile software development. Many of us will draw comparisons with agile development and Extreme Programming, or agile development and Scrum. We might believe that if we write test cases, we are agile. Or if we refactor. Or practice continuous integration. Each of these may help increase agility, but none guarantee agility.

Agility is not defined by process or practice. Agility is defined by ability. The ability to remain nimble and responsive to change. Increased agility is always good. Why would we want to be anything other than agile? Slow, brittle, and inept are certainly not defining characteristics of success. It's easier to slow down than speed up. That's the affect of agility. As in life, remaining agile requires care, attention, and nurturing. This chapter represents a compendium of principles, guidelines. practices, and general thoughts on how to increase agility.

The list below is not exhaustive. One would be challenged to describe all such practices in it's own book, let alone a single chapter. But it serves as a starting point that can help identify painful aspects of your software development initiatives, and then begin making the transition to a better way. Often times just getting started is the most difficult.

I've also omitted some of the more obvious and already well documented practices and techniques, choosing instead to focus on those more obscure and less famous.

While upper management buy-in helps speed the transition, it's still possible to realize success with support from only peers. How? You control software development's most important artifact! Ironically, it seems that we often forget the only purpose of a software development effort is to deliver useful software. It is our professional responsibility to fulfill that promise.

The Basic Principles

You might be wondering if the lower case 'a' in this chapter's title is a typographical error. No, it is not. It is a statement. A statement against the software process and methodology wars that have led the software industry astray.

It is symbolic recognition of the agile revolution. Revolution? Revolution! A strong word indeed. But agile development is different from the traditional approach, for it is recognition that we were wrong. Only recently have we begun to understand our craft, and exposed the fundamental principles that will drive the future of software development. Moving forward, we must push for the following ideals.

Eliminate Waste

Most of us have experienced the pain of waste. On many software development teams, we are prohibited from moving forward until the requirements specification is signed, the architecture is defined, and analysis and design complete. Only then can we begin writing code. But code is not written, it is crafted.

Pretending that specifications are stable, architecture is understood, and design is correct before we craft our first line of code is wasteful. Source code is a living entity that is constantly evolving. It is evolving because when change happens, the only artifact that must be modified to support that change is the source code. No other artifact comes with this restriction. If an artifact does not directly contribute to the quality and the integrity of the source code, creation of that artifact must be questioned.

Critics will argue that even to craft code, developers must understand requirements, and those requirements must be documented. I don't question the value of well-documented requirements so long as any requirements documentation produced is a living breathing document that evolves with the source code. But that requires discipline, and discipline is not a strong trait of software development teams, especially given tight deadlines and immense pressure to meet the date.

Ideally, the most effective artifact is executable against the source. For instance, a suite of user acceptance tests that have received customer approval. Source code is an executable artifact and must be wrapped by practices and processes that verify the correctness of the source.

Sustainable Pace

It's little surprise that traditional development efforts tend to fall further behind schedule as the deadline looms. Foremost, attempting to predict completion through estimating and detailed planning does not work. Software development is a minefield of unknowns that cannot be predicted. And while planning is not bad, the value of planning is not the plan itself but instead the insight gained from the planning experience. Planning must be a continuous activity to remain effective.

To maintain a sustainable pace, however, development teams must track where they have been, and what they have accomplished. While promising delivery is received with fanfare early in the project lifecycle, broken promises have a steep price. Clients and key business stakeholders quickly lose faith. But showing sustainable progress over time earns their confidence, and that trust is a valuable asset to the software development team. A prime directive of software development is to maintain forward, sustainable progress.

Intense Collaboration

When we gather the requirements for a software system, we meet with our business clients and ask questions. The very process of eliciting requirements demands intense collaboration, traditionally between a business analyst and a subject matter expert representing the business client. Each bullet point, business rule, precondition, and postcondition in the requirements specification is based on detailed discussion.

Documenting that discussion is not a bad thing, but relying on the document as the sole form of communicating that discussion to developers cannot work. When you read a novel, you will never understand the thought process of the author. Similarly, when reading specifications, you will never experience the deep insight gained by hearing the discussion. Without that insight, attempting to design and code from a specification will lead you astray. We must avoid using documentation as the primary form of communication, and seek involvement in those discussions producing the documentation.

Frequent Delivery

All things equal, frequent delivery of functional software that provides incremental business value to a client may be the single most important activity a team can perform. It's not necessary that a product be delivered to a production environment, only that it be delivered to an environment accessible by the business client.

Frequent delivery provides significant value. Business clients interacting with the system early in the lifecycle will experience the growth of the system, and gain additional insight to how requirements are manifest. Customers will feel much more engaged throughout the development effort, and will undoubtedly identify areas they would like changed.

The development team will also experience many rewards. A functional system proves architecture and infrastructure, and allows the technical team to begin many important testing activities, including load testing, usability testing, failover testing, and even user acceptance testing. Of course, frequent delivery is one important way of seeking feedback on the quality of the application.

Continuous Feedback

Ensuring the development effort never strays too far from the intended target is vitally important. One way to achieve continuous feedback from business clients is through frequent delivery, and it is this type of feedback that ensures the development team is well aligned with business objectives.

But there are other forms of feedback that are important to the development team, as well. Software undergoes constant change, and the resiliency, malleability, and extensibility of the software's architecture and design is critical. Code quality, design, and coverage metrics offer objective feedback, that when used judiciously, can aid refactoring efforts that ensure source code maintains a high degree of quality and integrity.

Embrace Change

Throughout the software development lifecycle, many things will change from inception through final delivery, and experienced developers recognize that change is inevitable. Traditional methods have achieved very little success in stabilizing requirements early in the lifecycle. But we still try, for two reasons. First, we want stability. Second, we seek predictability. We get neither. Believing requirements are stable provides the false sense of security that leads us to believe we can predict. We cannot.

Instinctively, we may feel change impedes progress, but agile developers embrace an attitude where change is viewed as an opportunity to improve the system. Therein lies a key to embracing change - attitude. We cannot hold business clients accountable when change occurs, as this only causes friction between the development team and the business team. A key step toward building a trusted relationship with your customer is to remain adaptable. Instead of resisting change, seek for ways to accommodate the request. Prioritizing features and exploring and explaining risk helps everyone more fully understand the consequence of change.

Empowerment

The most successful teams are powered by people, not process. A key differentiator of agile teams is that process conforms to people over people conforming to process. While process may serve as a useful guide, teams must trust their intuition. It is the team with the vested interest in the outcome of the development effort. It is important that decisions be pragmatic based on the current state of the project, and not driven by the next activity dictated by the process du jour.

Team organization is also important. The most successful teams have naturally emerging leaders who are passionate about their area of expertise. Appointing architects and team leads is common, but allowing leaders to emerge naturally is more rewarding.

The Environment

Teams interested in increasing agility often emphasize process and practices, but ignore the technological aspects necessary to increase agility. Are teams agile because they use Scrum? No. Are teams agile because

they practice Continuous Integration? No. Are teams agile because they have a project room? No. Agility is not defined by process or practice. Agility is defined by ability. The ability to deliver working software. The ability to respond to change. The ability of software to accommodate change and remain working software. Agile processes and practices serve as wonderful guides to help teams develop higher degrees of agility. But processes and practices that help manage change and emphasize delivery are not enough. Teams must also craft code, develop a supporting infrastructure, and adopt tools and frameworks that encourage and embrace change. Below are some essential characteristics of technology that help maximize the effectiveness of agile practices.

Agile Code

The ability to accommodate changing requirements is directly related to the flexibility and malleability of the code base. High quality code that is easy to maintain fills the gap between a team that embraces agile practices and a team that can follow through by accommodating change quickly and easily. A brittle codebase that lacks design and architectural resiliency prevents teams from realizing the promise of agility. Below are the characteristics of agile code.

Loosely Coupled

Coupling is the degree to which one program module depends on another. Tightly coupled code with excessive dependencies is brittle. Small changes in one area of the system have a ripple affect that impact other areas of the system, often unknowingly. To reduce coupling, dependencies among all modules must be carefully managed. For instance, applying proven design and architectural patterns help modularize large software systems, leading to loosely coupled code, components, and services.

Highly Cohesive

Cohesion is the degree to which a program module performs a specific piece of functionality. Well-written code is highly cohesive. A codebase lacking cohesion is difficult to understand, maintain, test, and reuse. Cohesive code results in smaller, more focused modules that ease change and increase reusability. Cohesive classes packaged into cohesive deployable units defined at an appropriate level of granularity allow development teams to shift architecture more easily as change surfaces.

Fully Tested

All software experiences change. Fully tested software embraces change by giving developers the courage to make change. A robust suite of unit tests will help identify areas of the system that may have been unknowingly affected when refactoring. Without tests, developers have no security blanket to prove their change functions as desired. Test coverage metrics help teams identify areas of the system that are lacking robust tests.

Expressive

Expressive code is easier to understand, maintain, and test. Developers should favor crafting code that humans can easily understand over writing code that only passes compilation. Simple practices, such as using meaningful class and variable names, providing consistent formatting, avoiding complex boolean logic

and nested conditionals, and using whitespace appropriately lead to more expressive code. A robust suite of tests also aids developers by showing different ways that the code is initialized, configured, and exercised.

Rarely Duplicated

Copying code to reuse all of part of a module burdens the maintenance effort. When logic changes, it's likely that multiple areas of the application require modification. Duplication also increases the testing effort. Refactoring and nurturing the code is an important step toward maintaining a high integrity codebase with little duplication.

Agile Infrastructure

Incremental software delivery is a core tenet of agile development. Delivery does not always correspond to a production release, however. Instead, delivery emphasizes releasing functional software to a stable environment where the application can be verified through quality assurance testing, performance testing, usability testing, and more. In the most agile environments, software is delivered hourly, ensuring any problems discovered are no more than an hour old. In addition to agile practices such as continuous integration, the ability to deliver functional software frequently and incrementally is directly related to a team's infrastructure components.

Shared Environment

A shared environment that is easily accessible by the project team provides a highly visible way to track development progress. The environment should be a close approximation to the target production environment, leading to earlier and more frequent discovery of issues. Releasing early and frequently to a shared environment provides the development team the time necessary to make adjustments; automating and streamlining the deployment process encourages frequent delivery. A shared environment also allows for other important activities, such as frequent application demos. A current integrated version of the software system always exists, providing teams a morale boost early in the lifecycle and avoiding problems where the system might work only on individual developer workstations.

Source Repository

Maintaining a master copy of source, as well as a history of changes, helps the development team understand the current state of the codebase and track changes over time. Since developers know where to obtain the most current copy of source, new developers can set up a development environment quickly and easily. Building and deploying the master copy of source allows the development team to quickly respond to failed builds, failed tests, or corrupt deploys.

Build Server

Frequent delivery requires frequent builds. An automated and repeatable build process that pulls the master copy of source from the source repository, compiles and executes all tests, and deploys the application to a shared environment enables other important activities early in the lifecycle. But compiling, testing, and deploying frequently can consume a developer's workstation. Running the automated and repeatable build on a separate build server allows teams to deploy frequently, without consuming developer resources on an

hourly basis.

Local Sandbox

Each developer must have a local environment that can be quickly and easily synchronized with the source repository. Beyond simply a development environment, a local sandbox might include a database schema owned by the developer, local queues for messaging, and other technologies that allow developers to experiment and proof ideas before implementing and deploying to the shared environment.

Flexible Hardware Platform

As with software, it's not easy to predict how the hardware environment will be exercised over time. Activities such as adding new servers to a cluster, increasing capacity and storage space, and scaling to accommodate increased load are critical to ensure increased demands are met with ease. Fortunately, technology is readily available that allows us to assemble platforms and create virtual servers that abstract away operating system constraints and allocate resources as needed. Tools are also available that allow us to proactively manage server, network, and database environments.

Agile Tools and Frameworks

Many software applications and development efforts rely on third party software. When adopting tools and frameworks, teams must assess the impact a tool or framework has on their ability to work effectively and efficiently. A team's ability to remain agile is directly related to the tools they use.

Testable

Many development teams use third-party libraries, such as persistence and web frameworks, to facilitate development. Before adoption, technology should be evaluated to ensure it does not stand in the way of a team's ability to test the software. When necessary, appropriate abstractions should be defined that shield the application from aspects of the technology that inhibit testing. For instance, if an MVC framework is tightly coupled to an application server, the team should work hard to ensure that application tests do not rely on the presence of the MVC framework. Such reliance prevents testing outside the context of the application server, bringing more weight to the testing process.

Non-Invasive

Agile technologies should integrate well with the environment and a team's style of working. As much as possible, tools and frameworks should avoid dictating how a team works; rather they should conform to how the team works. Effective agile technologies integrate well into the environment and have a small footprint.

Easy to use

An important trait of an agile framework or tool is that it allows the developer to understand what the technology is doing and how it functions. For instance, if a tool is generating code, it's important that the

code is expressive. If a framework is performing persistence functions, it's imperative that developers are able to gain a clear and concise understanding of how it is accomplishing the task. When choosing a framework or tool, a good initial assessment can be made by simply evaluating the effort required to integrate the technology into an existing environment.

Flexible

Rigid and inflexible technologies prevent a tool from conforming to a developer or team's style of working. Agile technologies are configurable and extensible. When using a build tool to compile, test, and deploy an application, developers should be able to configure the tool so that they have control over how the build is performed. Developers should have the ability to extend frameworks by implementing abstractions, knowing that their extensions won't cause the framework to behave strangely. One sign of flexibility is the ease with which developers can adopt technology without being forced to conform to a specific style of working, or compromising the quality of their design.

Lightweight

Frameworks and tools with excessive dependencies on external components decrease agility. For instance, frameworks tightly coupled to an application server make testing difficult. More flexible technologies are modular, and can be introduced to an environment incrementally.

Team Guidelines

Working to establish an environment that increases agility is only a starting point. Guarding the integrity and quality of the source code is our top priority. All else must be secondary. Nothing can be proven, verified, or evaluated unless a functional system is available. While a supporting environment is critical, the team must be rigid in enforcing some simple team guidelines.

Golden Rule

A prime directive of agile development is sustainable forward progress. A failed build always represents a step back. Anytime the team experiences a build failure, it must become the priority of the team, above all else, to rectify the situation. No changes other than those correcting the problem should be released to the source repository until the application is successfully built.

Synchronization Rule

Developers should release and update their local environment at least once per day with the master source repository. Waiting any longer than this increases integration risk.

Release Rule

Avoiding build failures requires developers to exhibit more discipline while coding. Developers must write

tests and code simultaneously. When finished, a local build should be performed, followed by synchronizing changes with the repository. Any differences between the local environment and the source repository should be resolved before releasing code. Developers should give careful attention to ensure they release all modifications to avoid a partial commit that may cause a build failure.

Defect Rule

Fixing bugs must be done with rigor and discipline. Developers should first write a test case that recreates the defect. Initially, the test should fail, but after correcting the defect, the test should execute successfully. In addition to identifying and fixing defects, automating tests as part of the build ensures the defect does not resurface. When finished correcting the problem, source code changes including tests, should be released according to the Release Rule.

Team Organization

The software industry does a very poor job of learning from its history. In 1968, Melvin Conway gave us *Conway's Law*, which states that organizations that design systems are constrained to produce designs which are copies of the communication structures of these organizations. In other words, the design and architecture of software is a reflection of the team or teams that built it. Agile practices speed software delivery and increase software quality by increasing communication and sharing valuable information. The team structure and collaboration methods in place are critical aspects in ensuring the development team delivers resilient, adaptable, and high quality software.

Team Structure

It should be no surprise that large software projects have a higher failure rate than do smaller projects, and that larger teams are more difficult to manage than smaller teams. For large projects, dividing the system into smaller systems, or modules, is one technique to make development a bit more manageable. Herein lies the rub. Failure to share information across these smaller teams of developers results in each module having a separate architecture, inconsistent user experience, and disparate levels of quality depending on the practices employed by each team. This is the heart of Conway's Law. While SOA may be the architecture model du jour, the benefits of loose coupling and service transparency can work against a team if developers rely solely on the service interface as the means to communicate.

To avoid the integration and quality issues that surface when teams work in isolation, team cross-pollination is important. Experienced developers from each team must communicate on a regular basis to discuss architectural, functional, and technological issues. The structure and makeup of the teams have a significant impact on the teams' ability to communicate. I've found two common models used when organizing teams of developers.

Process Organization

With this structure, groups of developers focus on developing complete business processes. Each process may or may not be part of a larger, consolidated system. The group of developers emphasize building front-

to-back functionality, including all user interface, business logic and data access functionality. The greatest benefit is that developers gain a more intimate understanding of the business process and tend to be more connected with business objectives. For larger systems, with many smaller teams of developers focusing on developing support for business processes, Conway's law tends to surface quite readily.

Technology Organization

With this structure, groups of developers focus on working with a certain technology. For instance, one group of developers might be solely responsible for the data access layer, with others focusing on the business layer, and others focusing on integration with external systems. This allows developers to gain a significant amount of expertise with specific technology, such as Hibernate in the data access tier. But all too often the teams lack a clear and concise vision of the business goals of the project. Without this understanding, developers tend to emphasize technology excellence over business excellence. In addition, integration issues can easily arise if developers across tiers fail to communicate.

Which Way is Best

Since I've found it produces more positive results, I favor process organization over technology organization since developers gain a more intimate understanding of the business processes they must support. The challenge with process organization, however, is avoiding silos that cut off the smaller teams of developers from each other. Encouraging technology experts from within each process team to share information across teams is one way to avoid silos from developing. For instance, a usability expert that spans teams is one way to ensure the system maintains a consistent look and feel. If Hibernate is used as the data access framework, an expert well-versed in Hibernate can offer guidance to ensure access to data is done consistently, reliably, and performs well. Assigning individuals to these roles does not always work. Instead, allow technology experts to emerge from within the process teams and establish a plan for them to fulfill the requirements of this new role while allowing them to participate with their process teams.

Favoring process organization offers many advantages, but one significant disadvantage is having small teams working in silos. It's imperative to establish communication channels that span disciplines and teams. I caution against using heavy forms of documentation as the preferred means of communication, keeping in mind that the value of a document is not the document itself, but the thought that went into creating the document.

The Agile Matrix

Figure 1 illustrates the Agile Matrix, a cross-pollinating software development team. Teams of developers are responsible for fulfilling the requirements of core business processes, while technology experts working on process teams are responsible for mentoring and communicating across project teams on the most appropriate use of practices, patterns and technologies relevant to a specific tier or specific role. Open channels of communication must be present across all teams and all team members must advocate the use of techniques to ensure that intense collaboration prevails. The result is multiple levels of communication that span teams, roles, and ultimately projects.

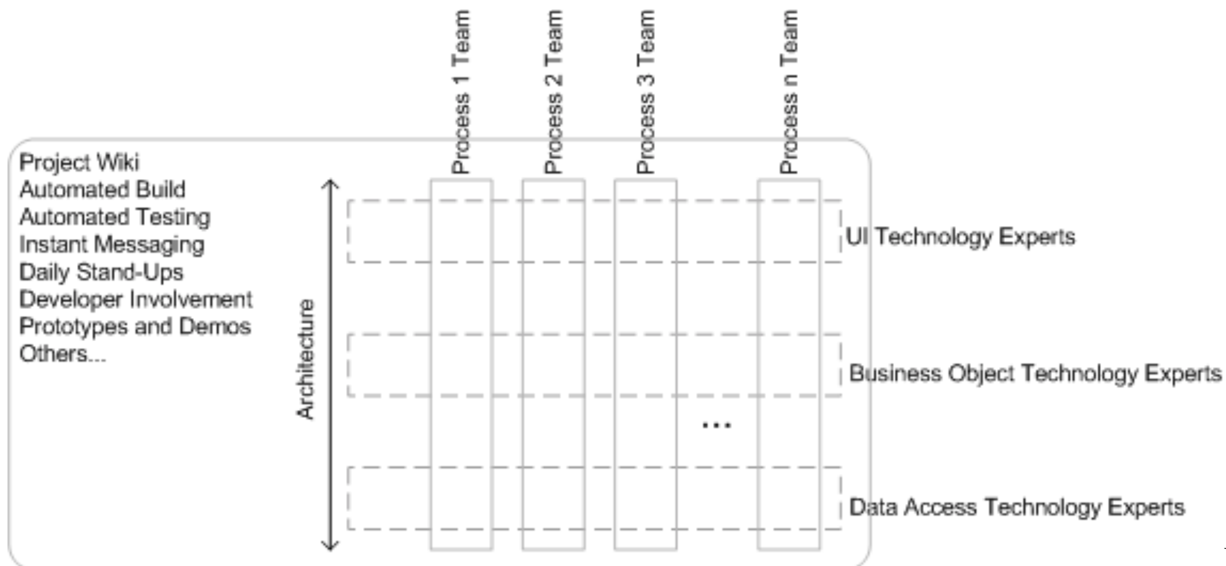

Figure 1

Fundamentally agile

In lieu of adopting in complete form an Agile software development process, such as Scrum, XP, or Crystal, injecting practices can help ease the pain. Such an approach makes an agile transition easier, less risky, and ultimately more beneficial.

These practices take a very narrow view. If you develop a codebase that is infinitely malleable, fully tested, and frequently deployed, your team possesses the ability to adapt. By emphasizing creation, growth, verification, and deployment of your code from initial project inception to final solution delivery, these practices serve as the foundation of an agile environment.

To realize the greatest benefit from these practices, they should be adapted based on the constraints, attitude, and culture of your social and technology environment. After injecting and adapting these practices, allow them to evolve in a way that helps your team continue to increase agility.

Automated and Repeatable Build

The most important artifact produced by the software development team is the source code. It is the only reliable measurement depicting the current state of the application. If the source doesn't compile, the application is not functional. Building on a regular schedule, such as hourly, helps ensure the team is on track. Incorporating e-mail notification into your build process informs all developers of the status of the build. Additionally, producing a build website allows the team to review important statistical information related to the application such as design quality, code quality, and percentage of code under test.

Continuous Integration is a strategy employed by teams to deliver functional software as frequently as possible. The cornerstone of a robust continuous integration strategy is an automated and repeatable build.

The Build

The build must be repeatable. Performing a local build in an integrated development environment (IDE) is a great way for developers to analyze the isolated impact of their changes, but it is no way to build, package and deploy an application in a team environment. Local builds are a manually intensive process performed by individual developers with no guarantee of consistency. Skipping a step, working with older copies of source or incorrectly performing a build task are common mistakes that can result in a corrupt application. A repeatable build helps solve these problems by ensuring the steps executed to build, package, and deploy the application are performed consistently each time. Fundamentally, a repeatable build consists of at least the following:

- It must be a clean compile with no syntax errors.
- It must be performed on the most current version of all source files.
- All application source files are compiled.
- The build should generate all units of deployment.
- The build should execute all relevant tests successfully.

Depending on your platform, tools are readily available that allow you to develop a repeatable build script. Ant is the most popular utility for J2EE development.

The build must be automated. Even with a repeatable build, time must be spent manually invoking the build process. Eliminating waste is a core tenet of successful agile teams, and that which is repeatable can be automated. Build automation frameworks allow you to schedule the build so that it is run at predictable intervals. Frequency of the build is a product of team dynamics. For smaller teams, it's useful to execute the build each time a change to the source code is released to the source repository. Larger teams may choose to execute the build according to a defined schedule, such as hourly, due to the heavy volume of released code. Project teams should experiment with frequency to address their specific needs.

CruiseControl is an example of a build automation framework for the J2EE platform that sits atop an Ant build script and executes the script based on configuration. Maven is a software management tool that helps manage the build, while also generating feedback aimed to provide the development team with additional project information.

The build must have a supporting infrastructure. In addition to build and automation tools, additional supporting infrastructure must be in place to support an automated and repeatable build.

Extending the Build

The build should be extensible. Once an automated and repeatable build with supporting infrastructure is in place, the team can think about other aspects of the development lifecycle that can be automated. Generating quality, dependency and coverage metrics metrics as part of a build process and subsequently publishing the results to a website or pushing the information to developers via e-mail gives team members the type of rapid feedback they need.

An automated and repeatable build is the cornerstone of any successful agile development team, enabling a plethora of important lifecycle activities. An agile environment is a living and breathing entity where

progress is tracked in minutes, not weeks and months. Here are a few of the advantages you'll realize as the result of an automated and repeatable build.

- Incremental Growth. Iterative methods advocate incremental growth of the software system. Whereas some less agile methods encourage iteration plans attempting to predict incremental growth, agile teams experience incremental growth daily.
- Frequent Deployment. Early and frequent deployment to a shared environment avoids integration risk, and offers the team the opportunity to tweak application parameters and execute tests that are best suited for a shared environment. Delaying deployment increases the likelihood of experiencing configuration problems or application packaging issues.
- Functional Demonstrations. Early in the software lifecycle, most development teams aim to understand user requirements. Typically, meetings are held that emphasize driving out these requirements and documenting them based on discussions. As the software grows, functional demonstrations can be used to help gain additional insight to requirements in a different context. Regularly scheduled prototypes and functional demos involving team members and business clients ensure that all team members remain in touch with the core business objectives surrounding the development effort. For the development team, prototypes and demos offer insight into other processes that developers may not be intimately familiar with. For business clients, it gives them a complete view of the integrated system. It's important to begin hosting prototypes and demos early in the development lifecycle to help identify inconsistencies across processes, system navigation and workflow issues, and duplication of effort. Wire frames and screen mock-ups can be used as prototypes before a functional codebase is available. As the system grows, functional demos replace the prototypes.
- Constant Feedback. Extending the build with coverage, design and quality metrics allows the team to gain feedback on the integrity of the software. Potential areas of concern can be identified and corrected immediately upon discovery. Additionally, because the build is scheduled, teams can track progress and setback trends.
- Lifecycle Automation. An automated and repeatable build automates many of the mundane tasks previously requiring manual intervention. Build, tests, packaging, and deployment are automated and therefore occur regularly and often times without manual intervention.
- Adaptability. Your build serves as a system of checks and balances for the development team. Tests provide the courage to refactor. Refactoring is validated by frequent builds. And builds are deployed to a shared environment where they can be further tested and verified. The combination of these factors contributes to an environment where change is more easily accommodated.

Minimize Dependencies

Software tends to rot over time. When you establish your initial vision for the software's design and architecture, you imagine a system that is easy to modify, extend, and maintain. Unfortunately, as time passes, changes trickle in that exercise your design in unexpected ways. Each change begins to resemble nothing more than another hack, until finally the system becomes a tangled web of code that few developers care to venture through. The most common cause of rotting software is tightly coupled code with excessive

dependencies. For large teams and large applications, managing dependencies is especially important. Excessive dependencies cause numerous problems.

- Hinder Maintenance. Dependencies hinder the maintenance effort. When you're working on a system with heavy dependencies, you typically find that changes in one area of the application trickle to many other areas of the application. In some cases, this cannot be avoided. For instance, when you add a column to a database table that must be displayed on the user interface, you'll be forced to modify at least the data access and user interface layers. Such a scenario is mostly inevitable. However, applications with a well thought dependency structure make change easier since developers have a more complete understanding of the impact of change.

- Prevent Extensibility. The goal of flexible software architecture is to remain open for extension but closed to modification. The desire is to add new functionality to the system by extending existing abstractions, and plugging these extensions into the existing system without making rampant modifications. One reason for heavy dependencies is the improper use of abstraction, and those cases where abstractions are not present are areas that are difficult to extend. Abstraction aids large teams by exposing well-defined extension points, as well as encapsulating implementation complexity.

- Inhibit Reusability. Reuse is often touted as a fundamental advantage of well-designed object oriented software. Unfortunately, few applications realize this benefit. Too often, we emphasize class level reuse. To achieve higher levels of reuse, careful consideration must also be given to the package structure and deployable unit structure. Software with complex package and physical dependencies minimize the likelihood of achieving higher degrees of reuse.

- Restrict Testability. Tight coupling between classes eliminates the ability to test classes independently. Unit testing is a fundamental principle that should be employed by all developers. Tests provide you the courage to improve your designs, knowing flaws will be caught by unit tests. They also help you design proactively and discourage undesirable dependencies. Heavy dependencies do not allow you to test software modules independently. Teams with few tests cannot respond to change easily due to the inability to access the impact of change.

- Hamper Integration. It's easy for separate teams to plow forward, usually under tremendous pressure from looming deadlines. They operate under the false assumption that if they can simply reach the final feature destination, they can quickly pull things together toward the end of a project. This Big Bang approach to integration does not work. As individual modules are pulled together, common issues that surface include degradation of overall system performance, incorrect levels of behavioral granularity provided by system modules, and transactional incompatibilities. More frequent integration brings many of these issues to the forefront earlier in the project.

- Limit Understanding. When working on a software system, it's important that you understand the system's structural architecture and design constructs. A structure with complex dependencies is inherently more difficult to understand. Clear and concise dependencies that are well-thought allow teams to more easily access the impact of change.

Checks and Balances

Managing dependencies between classes, package, and components throughout the development lifecycle is imperative when developing a software product that accommodates change and meets the business

objectives. While breaking larger teams into smaller groups of developers helps establish fewer channels of communication, these smaller teams tend to introduce other architectural and integration issues. For larger projects, it's likely these issues will affect your software before your first production deliverable. Agile practices applied throughout the lifecycle help minimize dependencies and encourage more frequent integration. An automated and repeatable build serves as a system of checks and balances that help ensure large teams maintain forward and sustainable progress.

Separate teams should be disciplined in releasing their changes to the source code repository on a frequent basis. Frequent builds help identify integration issues almost instantaneously, and help avoid the risk associated with Big Bang integration. Automated unit tests verify the integrity of code released by each team, while more complete integration and functional tests can verify behavior spanning team boundaries. Ensuring the complete system is free of compilation errors allows for frequent profiling of the application to identify serious issues related to performance, memory, and other environmental constraints. A shared environment that closely resembles the future production environment is also a critical aspect to help identify environmental issues. Additional techniques help add to this system of checks and balances to ensure development teams minimize dependencies and emphasize integration earlier in the software lifecycle.

- Dependency Metrics and Diagrams. Utilities, such as JDepend and JarAnalyzer for Java, are available that allow teams to obtain real and accurate feedback on the structural relationships between software packages and components. Understanding module and package dependencies allows developers to somewhat objectively evaluate the cost of change. As dependencies become excessive, such utilities help guide developers in prioritizing areas of the system that are higher priority candidates for refactoring.

- Levelized Build. Cyclic dependencies, recognizable as bi-directional relationships between packages or components, are especially dangerous as neither can be tested or deployed in isolation. One way to enforce acyclic relationships is building components individually, and including in the build and test execution path only those modules required to compile and test the component being built.

- Parallel Build. As systems grow, build times tend to slow. This is natural as more code must be compiled and more tests are introduced that exercise more code. When build time degradation occurs, it's important to identify ways to increase the efficiency of your build process. One technique is to build components in parallel. If dependencies are managed carefully, components with no dependencies on each other can be built simultaneously.

- Component Tests. Components built in isolation, can be tested in isolation. For each component represented by a deployable unit, creating a corresponding test component helps ensure the functional integrity of the component. Test components are typically manifest as suites of individual unit tests, and aid any refactoring to help maintain the design integrity of the component.

- Integration Tests. Test suites that verify component integration certainly help ensure that the interfaces between components remain compatible, but also help identify behavioral issues, as well. Incorporating these test suites into your build process ensures that integration tests are frequently executed, reducing the likelihood that extended periods of time pass between periods of integration.

Executing test cases frequently offers another form of very useful feedback related to the functional correctness of the application. Developers can be notified of failed tests, including an indication of an area

of the application that needs immediate attention. Notifying all developers of test status keep the team informed of the functional state of the application. When teams of developers are organized around process, automated integration testing helps the technology experts that span teams evaluate test results for the system and points to potential problem areas.

Running Tested Features

Running Tested Features is a metric used to measure the delivery of running, tested features to business clients, and collect their feedback. Suggested by Ron Jeffries as a way to increase agility and team productivity by continuously monitoring the software development effort from project inception through delivery, RTF is simple to define.

Break the software down into features that represent observable value to the business. For each feature, create one or more automated acceptance tests that validate the feature. Consolidate all features into a single, integrated product and continuously execute the automated acceptance tests. At any given moment, the project team should know the number of features that are passing all acceptance tests. This number represents the RTF metric.

Continuous RTF

RTF is a continuous measurement and, all things being equal, should grow linearly throughout the project. To explore what this means, we must first compare agile development with iterative development. Iterative development teams tend to timebox their iterations, marking each iteration with a milestone focused on delivering a planned set of features to clients every two to four weeks. Iteration planning defines the work included in future iterations based on the amount of work accomplished in previous iterations. Yet within these iterations, teams should deliver functional and valuable features daily, hourly, or even more frequently. Teams should not be constrained to delivering features at the end of each iteration. While project management defines iterations that last weeks or months, primarily for planning, the development team works in iterations that lasts days, hours, or minutes. And each feature that is released throughout an iteration planning window is a feature that can be measured and contributes to RTF.

On many software development efforts, we aim to align each process team to the same iteration schedule. This is not necessary. Each process team can work within its own iteration planning window. Agile practices, including continuous integration, ensure individual process teams remain aligned, and the processes of continuous release, continuous builds, and continuous deploys encourages the smaller iterations that often go unacknowledged on many development efforts. It is these smaller iterations where software growth occurs, and where we continuously measure Running Test Features.

Figure 2 depicts a multi-dimensional agile development effort, where iterations overlap but where RTF experiences linear growth as each iteration nears completion. Critics might argue that RTF cannot experience consistent linear growth due to unpredictable setbacks throughout the development lifecycle. While it's true that teams will experience challenges that limit RTF growth for a short period of time, analyzing RTF reports over time will show positive growth.

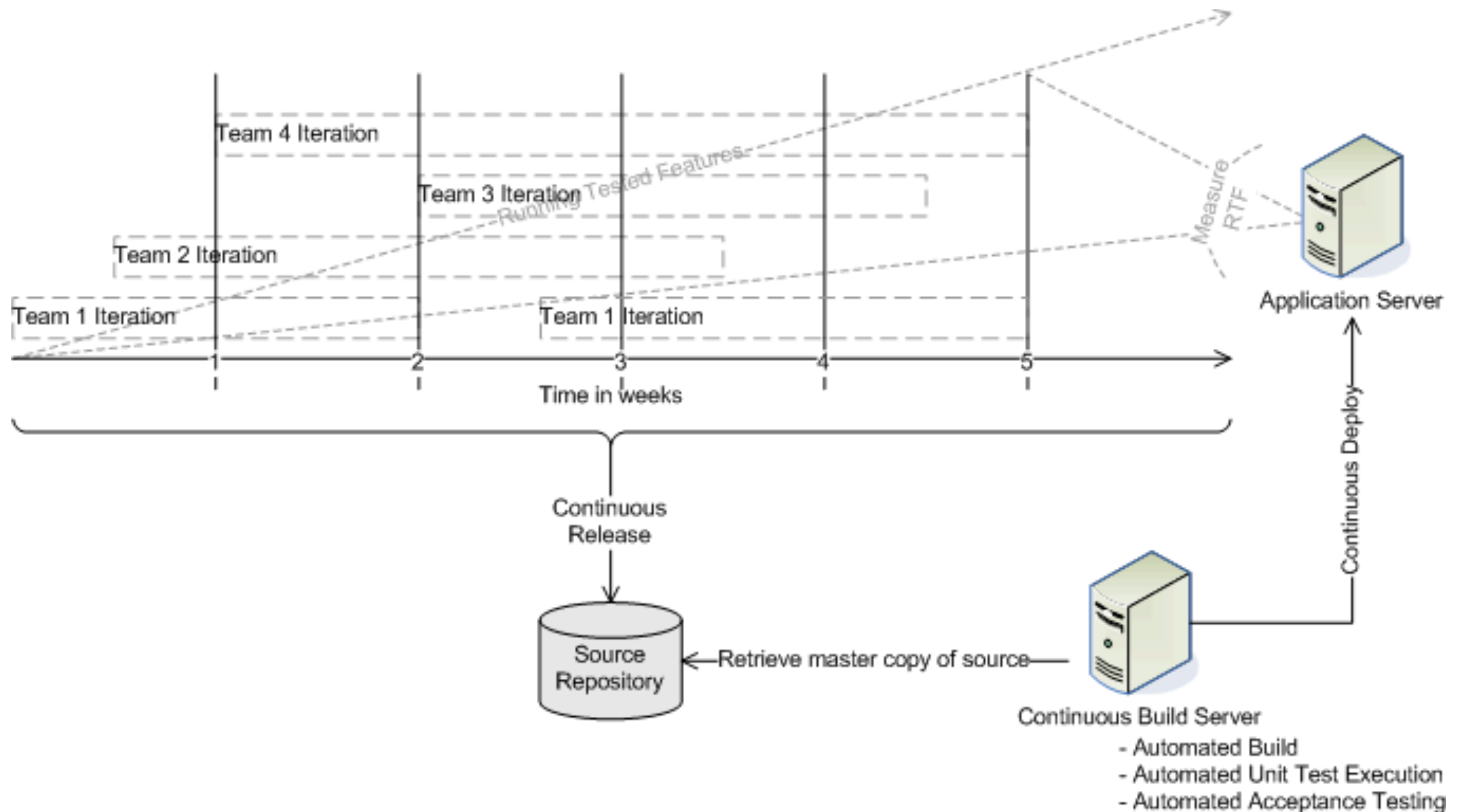

Figure 2

RTF Strategy

Choosing the correct tools has a tremendous impact on a teams ability to remain agile. Agile tools are testable, non-invasive, easy to use, flexible, and lightweight. As RTF requires continuous measurement of automated acceptance tests, measuring RTF requires a tool. A few such tools exist in the open source space.

FitNesse is a testing tool, wiki, and web server all rolled up into one. FitNesse allows project team members to collaboratively define acceptance tests by creating tables on web pages using wiki markup. FitNesse wiki pages integrate with test fixtures that invoke application code, making FitNesse a testing tool that doesn't rely on web based testing, but does require test fixtures that have the ability to directly invoke application code.

Selenium simulates the user by executing its tests directly in a browser. SeleniumIDE is a plug-in for FireFox that allows you to record test scripts. Scripts can be included on an HTML suite page using Selenium Core, or converted to Java, C#, Perl, Python, or Ruby for inclusion in an xUnit style test case using Selenium Remote Control.

Both FitNesse and Selenium are relatively agile tools. FitNesse does require running the FitNesse server, as well as test fixtures that can interact directly with the code being tested, so the test code must be deployed with the application under test. Selenium Core comes with the same restrictions, but Selenium Remote Control does not. Instead, Selenium Remote Control provides a Selenium Server that is run and communicates with the browser via AJAX, allowing you to test a web application in a very non-invasive style.

Conclusion

In 1994, Standish Group published the original CHAOS report. The top three impediments to project success were lack of user input, incomplete requirements and specifications, and changing requirements and specifications. Recent feedback indicates similar problems exist today, proving that attempts to stabilize requirements early in the lifecycle cannot and will not work.

Software requirements will always change, and it is our job to deal with it. Eventually, all change leads back to the source code. We must nurture and grow a resilient codebase. We must adopt practices that promote early and frequent discovery. We must adjust our attitude to embrace change. It is our professional responsibility to lead the effort toward a better, more satisfying way of creating software. The only guarantee in life is change. If we do not assume this professional responsibility, someone else will.