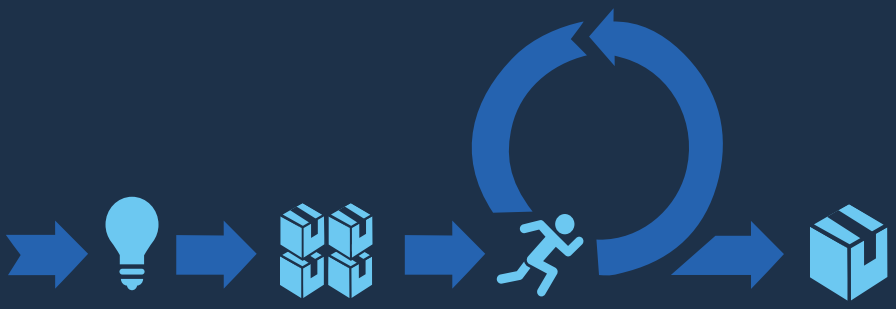
VERSION
1.0

*Segue Technologies, Inc.*

# ADOPTING AGILE DEVELOPMENT



www.seguetech.com
Version 1.0

Published 2016 by Segue Technologies, Inc.

Cover Design, Layout, and Illustrations created by
Segue Technologies.

All links featured in this book can be found at

www.seguetech.com/blog

***About Segue Technologies***

*Segue has developed innovative, dependable, and user-friendly applications since our founding in 1997. We provide a wide range of Information Technology services, focusing on Software Engineering, Information Management, Quality Assurance, and Systems Integration. We are a growing small-business, supporting Federal, Commercial, and Non-Profit clients.*

www.seguetech.com

# TABLE OF CONTENTS

# FOREWORD

I've been working in software development for over 20 years professionally, and for another decade before that. When I was in the first grade, my father brought home a TI-99/4A – an old home computer which plugged into our TV via an RF modulator. We didn't have any program cartridges or peripherals, other than an audio cassette deck which we could use to store things on – you guessed it – audio cassettes. When we turned it on, it booted in ROM BASIC – an extremely rudimentary interface for writing programs using the BASIC programming language. If you're familiar with Visual Basic or VB.Net, those are to ROM BASIC as a Lamborghini is to a wheelbarrow. I remember spending many hours as a child transcribing small computer programs (mostly games) from *Compute!* magazine into the computer, then running them to see what happened. Eventually, I figured out how to make changes and get the programs to behave in different ways. Believe it or not, as a seven year old, I stumbled upon a rudimentary form of Agile Development – start with something, make changes, see what happens, rinse, repeat.

As I matured as a software developer, my methodologies also matured. I started working with other people to figure out what needed to be done; kept a log of the changes I was going to make; and reviewed the changes with others once I was done with the work – all very quickly and efficiently. I was still doing everything in a very ad-hoc manner though. Once I started

working in the field as a professional, that all changed. As a member of a software development team, I was thrust into an environment of formal process, meetings, reviews, documentation and, worst of all, red tape. Changes took longer to get approved, work took longer to complete, and at the end of the day I felt like I had spent more time following processes that added little to no value to my work, and less time actually being productive.

When I came to Segue, things went back to a more relaxed, informal way of doing software development. We tracked change requests on notepads full of bullet points; had weekly meetings to review the prior week's work and plan the next, and were quickly able to deliver software that met our end users' needs – all without the process. As we grew, though, and as our customers demanded more rigorous processes from their vendors, we had to start using a more formally defined and documented software development methodology. However, this time instead of being thrust into an existing environment, I was able to help craft one. The Segue team collectively drew upon our combined decades of experience to craft the earliest version of the Segue Process Framework, which focused on what we considered to be the best, most useful principles of various software development methodologies including Waterfall, Rapid Application Development, and others. Over time, we evolved our framework to include many practices of Agile Development, including most aspects of the Scrum methodology.

In this eBook, we will introduce you to software development methodologies (if you are not already familiar with them); dive into Agile development; discuss what characteristics key players in an Agile team should possess; and talk about how you can adopt Agile principles within your own organizations or work with Agile development vendors with whom you contract to build your custom applications.

Join us, then, on our Agile Journey!

*Mark Shapiro*
*Senior Architect- Application Development*

# ALIGNING PROJECT TRAITS *with* DEVELOPMENT METHODOLOGIES

| PROJECT TRAIT/FACTOR | AGILE | PLAN - DRIVEN (WATERFALL) | COMMENTS |
|---|---|---|---|
| CUSTOMER AVAILABILITY | Prefers customer available throughout project. | Requires customer involvement only at milestones. | Customer involvement reduces risk in either model. |
| SCOPE/ FEATURES | Welcomes changes, but changes come at the expensive of Cost, Schedule, or other Features. Works well when scope is not known in advance. | Works well when scope is known in advance, or when contract terms limit changes. | Change is a reality so we should prefer adaptability where possible. Contract terms sometimes restrict it. |
| FEATURE PRIORITIZATION | Prioritization by value ensures the most valuable features are implemented first, thus reducing risk of having an unusable product once funding runs out. Funding efficiency is maximized. Decreases risk of complete failure by allowing "partial" success. | "Do everything we agreed on" approach ensures the customer gets everything they asked for; "all or nothing" approach increases risk of failure. | Contract terms may not permit partial success and may require "do everything". |
| TEAM | Prefers smaller, dedicated teams with a high degree of coordination and synchronization. | Team coordionation/ synchronization is limited to handoff points | Teams that work together work better, but when contracts are issued to different vendors for different aspects of the project, high degrees of synchronization may not work. |
| FUNDING | Works extremely well with Time & Materials or other non-fixed funding, may increase stress in fixed-price scenarios. | Reduces risk in Firm Fixed Price contracts by getting agreement up-front. | Fixed price is tough when scope is not known in advance, but many government contracts require it. |
| SUMMARY | Agile is better, where it is feasible. | Plan-Driven may reduce risk in the face of certain constraints in a contract between a vendor and external customer such as the government. | Through educating our customers about the strengths and weaknesses of each model, we hope to steer them towards a more Agile approach. This may require changes to how our customers, particualarly the government, approach software development projects. |

Chapter 1

# Introduction

## An Introduction to Software Development Methodologies

In software engineering, one will often hear the term “Software Development Methodology” (SDM). A Software Development Methodology is a framework used to structure, plan, and control the process of developing an information system. Whether you choose Waterfall, Iterative, Agile, or some other methodology, how well you adhere to the SDM can effectively determine the success or failure of a project and/or company.

In this day and age, the number of projects and/or companies that fail to consistently adhere to a process is baffling. Software development initiatives are conceived and executed using the

"fly-by-night" approach. This is typically because customers may struggle to see or understand the immediate value of the numerous concept analyses, business requirements, use-case analyses and/or design specification meetings that are required to produce a quality product. Instead, time after time, they feel they can say *"I want..."* and the developers can figure it out and give them what they want with no questions asked. Developing a product without the guidance of a SDM often leads to systems that are delivered late, over budget, and in many cases, that fail to meet customer or end-user expectations. It could even lead to a complete project failure.

Adhering to a properly-defined methodology enables a project to provide better estimates, deliver stable systems, keep the customer informed, create a clear understanding of the task ahead, and identify pitfalls earlier, allowing for ample time to make adjustments. At Segue, we've found that as our SDM has grown and matured, we have become more effective at identifying potential problems before they occur, which has improved both our ability to proactively manage against their occurrence, and develop more effective workarounds for when they do happen.

When a SDM is not properly implemented, a variety of problems become more and more prevalent as development continues. For instance, a lack of proper communication between the customer and development teams often leads to systems that don't meet the intended needs of the customer. Mistrust from the customer management staff can have effects on the development contractor successfully maintaining or being awarded a follow-on contract. Furthermore, a lack of basic methodology concepts

or processes, such as an internal peer review process, often leads to software deployments with numerous defects. Delivering an unstable system is a bad reflection upon a company as well as the developers.

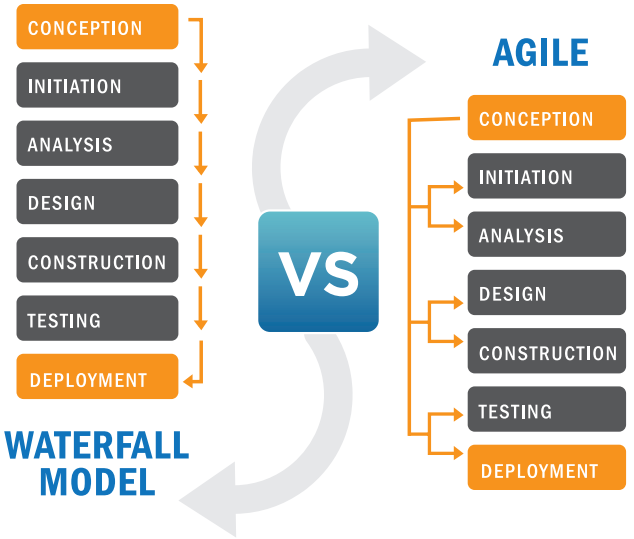
On a less tangible front, many developers often suffer from side effect issues such as poor morale or lack of motivation due to changing scope or the need for constant "bandage" fixes due to poorly-defined or poorly implemented processes around their development effort. The utilization of an SDM can greatly reduce this issue. With properly-defined processes in place, a roadmap is followed that allows for better management of scope creep and avoids common development problems. Of course, there will always be situations that may arise during the course of a project that may not have been planned, but adhering to a process will definitely minimize these occurrences.

Beginning a new SDM process definition initiative by simply conducting a "Lessons Learned" meeting and documenting the pitfalls and shortcomings of a completed project can do wonders for future endeavors with a similar scope. For instance, the review may result in the elimination of steps within the development process that ultimately didn't provide any value. Performing steps within a process just for the sake of doing it can be a waste of valuable time and effort, and the exclusion of this redundant work may deliver improvements, including the potential of an early system release date or the flexibility to address unplanned issues along the way.

First and foremost, the key to success is first to START using an SDM. There is no such thing as a "cookie cutter" approach or a specific SDM solution for all projects. The SDM acts as the starting point for the effort and should be tailored to meet the specific needs of a project. Find what works for you and stick to it. More importantly, as you move through the process, be sure to adjust your methodology to better suit your particular scenario and needs.

# Waterfall vs. Agile: Which is the Right Development Methodology for Your Project?

One of the first decisions we face for each of our project implementations at Segue is *"Which development methodology should we use?"* This is a topic that gets a lot of discussion (and often heated debate). This is NOT about a style of project management or a specific technical approach, although you will often hear these terms all thrown together or used interchangeably.



The two basic, most popular methodologies are:

1. **Waterfall:** (ugh, terrible name!), which might be more properly called the "traditional" approach.
2. **Agile:** a specific type of Rapid Application Development and newer than Waterfall, but not that new, which is often implemented using Scrum.

Both of these are usable, mature methodologies. Having been involved in software development projects for a long time, here are our thoughts on the strengths and weaknesses of each.

## THE WATERFALL METHODOLOGY

***Waterfall*** is a linear approach to software development. In this methodology, the sequence of events is something like:

1. Gather and document requirements
2. Design
3. Code and unit test
4. Perform system testing
5. Perform user acceptance testing (UAT)
6. Fix any issues
7. Deliver the finished product

In a **true** Waterfall development project, each of these represents a distinct stage of software development, and each stage generally finishes before the next one can begin. There is also typically a stage gate between each; for example, requirements must be reviewed and approved by the customer before design can begin.

There are good things and bad about the Waterfall approach. On the **positive** side:

- Developers and customers agree on what will be delivered early in the development lifecycle. This makes planning and designing more straightforward.
- Progress is more easily measured, as the full scope of the work is known in advance.
- Throughout the development effort, it's possible for various members of the team to be involved or to continue with other work, depending on the active phase of the project. For example, business analysts can learn about and document what needs to be done, while the developers are working on other projects. Testers can prepare test scripts from requirements documentation while coding is underway.
- Except for reviews, approvals, status meetings, etc., a customer presence is not strictly required after the requirements phase.
- Because design is completed early in the development lifecycle, this approach lends itself to projects where multiple software components must be designed (sometimes in parallel) for integration with external systems.
- Finally, the software can be designed completely and more carefully, based upon a more complete understanding of all software deliverables. This provides a better software design with less likelihood of the "piecemeal effect," a development phenomenon that can occur as pieces of code are defined and subsequently added to an application where they may or may not fit well.

Here are some **issues** we have encountered using a pure Waterfall approach:

- One area which almost always falls short is the effectiveness of requirements. Gathering and documenting requirements in a way that is meaningful to a customer is often the most difficult part of software development. Customers are sometimes intimidated by details, and specific details, provided early in the project, are required with this approach. In addition, customers are not always able to visualize an application from a requirements document. Wireframes and mockups can help, but there's no question that most end users have some difficulty putting these elements together with written requirements to arrive at a good picture of what they will be getting.
- Another potential drawback of pure Waterfall development is the possibility that the customer will be dissatisfied with their delivered software product. As all deliverables are based upon documented requirements, a customer may not see what will be delivered until it's almost finished. By that time, changes can be difficult (and costly) to implement.

## THE AGILE METHODOLOGY

***Agile*** is an iterative, team-based approach to development. This approach emphasizes the rapid delivery of an application in complete functional components. Rather than creating tasks and schedules, all time is "time-boxed" into phases called "sprints." Each sprint has a defined duration (usually in weeks) with a running list of deliverables, planned at the start of the sprint. Deliverables are prioritized by business value as determined by the customer. If all planned work for the sprint cannot be completed, work is reprioritized and the information is used for future sprint planning. The goal of each sprint is to deliver a working, functional product.

As work is completed, it can be reviewed and evaluated by the project team and customer, through the use of daily builds and end-of-sprint demos. Agile relies on a very high level of customer involvement throughout the project, but especially during these reviews.
Some **advantages** of the Agile approach are easy to see:

- The customer has frequent and early opportunities to see the work being delivered, and to make decisions and changes throughout the development project.
- The customer gains a strong sense of ownership by working extensively and directly with the project team throughout the project.
- If time to market for a specific application is a greater concern than releasing a full feature set at initial launch, Agile can

more quickly produce a basic version of working software which can be built upon in successive iterations.

- Features are prioritized by business value and development is focused on delivering maximum business value.
- Development is often more user-focused, likely a result of more and frequent direction from the customer.

And, of course, there are some **disadvantages**:

- The very high degree of customer involvement, while great for the project, may present problems for some customers who simply may not have the time or interest for this type of participation.
- Agile works best when members of the development team are dedicated full time to the project, rather than being shared between multiple projects.
- Because Agile focuses on time-boxed delivery and frequent reprioritization, it's possible that some items set for delivery will not be completed within the allotted timeframe. Additional sprints (beyond those initially planned) may be needed, adding to the project cost. In addition, customer involvement often leads to additional features requested throughout the project. Again, this can add to the overall time and cost of the implementation. This is balanced by the focus on delivering business value, however, so the extended schedule and additional cost will be delivering more value, rather than just delaying the project.
- The close working relationships in an Agile project are easiest

to manage when the team members are located in the same physical space, which is not always possible. However, there are a variety of ways to handle this issue, such as webcams, collaboration tools, etc.

- The iterative nature of Agile development may lead to frequent refactoring if the full scope of the system is not considered in the initial architecture and design. Without this refactoring, the system can suffer from a reduction in overall quality. This becomes more pronounced in larger-scale implementations, or with systems that include a high level of integration.

## MAKING THE CHOICE BETWEEN AGILE & WATERFALL

We consider the following factors when deciding what methodology to use on a project:

- Customer preference or stated methodology requirement
- Customer tolerance for scope and cost changes
- Project size and complexity
- Customer availability
- Level of integration with external systems
- Time to market

The first two points tend to direct the decision towards a particular methodology more than the others. As with any IT decision, you should make your choice based on what is best for your project, rather than what is the trendiest. The remainder of this e-book should help you understand when Agile will work well for you.

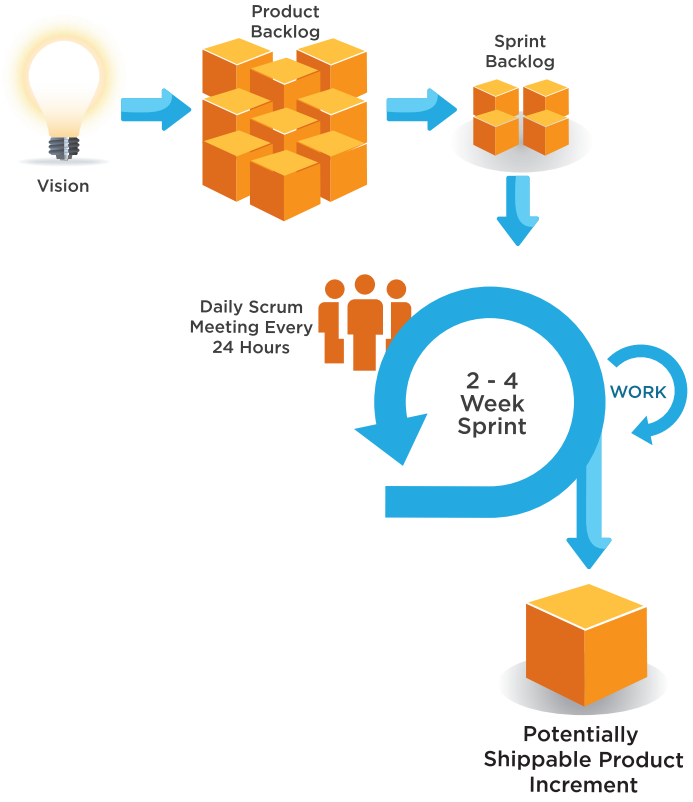
Chapter 2

# The Agile Methodology

## What is Agile Software Development?

There's been a lot of talk about Agile in the IT world lately. So what's all the buzz about? As you now know, ***Agile*** is a method of developing software solutions, including websites, web applications, and mobile applications, that focuses on delivering high-quality working software frequently and consistently, while minimizing project overhead and increasing business value. Stick with us to learn how an Agile software development approach can improve the effectiveness and quality of your next software development project.

During our 20+ years as IT Professionals, we have all too often seen projects that are over budget and over schedule, while not always providing value to clients and their users. At the same time, we recognize that IT projects are often complex, requiring a deep understanding of our customer's goals, challenges, industry, and customer needs and expectations. To address these challenges and opportunities, more and more software development projects are turning to Agile. In our experience as Agile Teams, we have seen the ability of Agile methodologies to dramatically improve results, while also increasing client and team collaboration, engagement, and satisfaction.



## THE AGILE MANIFESTO

The Agile Manifesto, originally written in February 2001 by 17 software developers, states:

> *"We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:*
>
> - *Individuals and interactions over Processes and tools*
> - *Working software over Comprehensive documentation*
> - *Customer collaboration over Contract negotiation*
> - *Responding to change over Following a plan*
>
> *That is, while there is value in the items on the right, we value the items on the left more."*
>
> *Kent Beck | James Grenning | Robert C. Martin | Mike Beedle | Jim Highsmith | Steve Mellor*
> *Arie van Bennekum | Andrew Hunt | Ken Schwaber | Ron Jeffries | Martin Fowler | Brian Marick*
> *Alistair Cockburn | Jeff Sutherland | Ward Cunningham | Jon Kern | Dave Thomas*
> 

While there have been criticisms that the Agile Manifesto is outdated and does not reflect the emergent realities of software development; its core values still influence modern Agile practices.

## THE AGILE APPROACH

While there are many implementations of Agile methodologies, the most common steps in an Agile software development approach are:

1. **Discovery**
   It's important to understand a client's vision and background when starting any new project. Agile software development projects start with a series of Discovery Sessions and research to understand a client's goals, challenges, business climate, and customers and users. These sessions include key members of the project team including the client, project manager, designer, developer, and product owner to ensure a shared understanding across the entire team.

2. **The Product Backlog**
   During Discovery, the team works together to create a high-level Product Backlog, a wish list of all the features that would be useful to the client and their users. The product owner works with the client to prioritize these features, determining the order in which the features are elaborated, developed, tested, and delivered. By allowing the client to determine priority, the team stays focused on delivering the highest value features before moving on to lower value features.

3. **Iterations**
   After ensuring the team understands the client's vision and has created a high level backlog of features, the team delivers features through a series of time-boxed iterations called

Sprints. These are fixed durations of 1-4 weeks (depending on project size and duration), each delivering a subset of the overall product backlog.

4. **Continuing the Cycle**
   Additional Sprints are conducted as needed to deliver additional features and incorporate feedback from previous iterations, reviews, and user beta testing. Each successive Sprint is both Iterative, providing improvements to work completed in previous sprints; and Incremental, adding new features to the system.

## AGILE ACTIVITIES

At the beginning of each iteration, the team conducts a **Planning Meeting** to review and elaborate the highest priority product backlog items, refining the details of each with the product owner and client to ensure their expectations are understood. After this initial planning meeting, the project team of designers, developers, testers, and other roles as needed work together with the product owner to develop, test, and accept each feature, working in priority order.

The team meets daily to stay in sync at a **Standup Meeting** or **Daily Scrum**, reviewing completed work, planned work, and any issues. This meeting should be no more than fifteen minutes, hence the timer that's sitting on the desk. At this time, team members share (1) what they did the previous day, (2) what they plan to accomplish today, and (3) what possible blockers they see to getting their job done. Sometimes these meetings are standups,

as in everyone literally remains standing throughout the meeting. Standing helps remind the more verbose teammates to keep their input short, sweet, and to the point.

In addition to this daily meeting, daily **builds** of the software allow the team to quickly inspect progress and ensure the software is working as expected. Testing is also conducted frequently throughout the iteration, with the goal of identifying and resolving defects quickly to reduce their impact later in the project.

At the end of each iteration, the team and client meet at a **Review Meeting** to review what was accomplished during the iteration, including a **Demo** of the newly delivered features. This review provides a valuable checkpoint to identify feedback and a deeper understanding of the solution. At the end of each iteration, the working software is deployed to a demo environment for additional client review and potentially beta testing with real users, and can even be deployed into a Production environment if the Product Owner determines it is ready.

After the Review Meeting, the project team (usually excluding the client) gets together for a **Retrospective**, during which the team discusses the manner in which work was conducted (the 'how', rather than the 'what') to determine if the process needs improvement. Team members discuss what practices are working well by answering (1) what they should continue doing, (2) what they should stop doing, and (3) what they should start doing in future Sprints.

## AGILE ROLES

The **Scrum Master** acts as a coach to the team. The Scrum Master participates in all the activities, guides the team, deals with any impediments (such as blockers brought up during the Daily Scrum), and aids the team in understanding Agile practices.

The **Product Owner** serves a unique role, either as a client representative dedicated to the project, or as a member of the project team partnering with the client to gain a deep understanding of their vision and advocate for that vision throughout the project. The Product Owner represents the interests of the customer and users, and is responsible for maintaining the Product Backlog, prioritizing features, and guiding the team toward building the right product. This grooming and prioritization ensures that the project team is focused on maximizing business value.

A **Scrum Team** has three to nine members and will include software developers, analysts, testers, designers, and any other roles pertinent to the project. Larger projects may have multiple independent Scrum Teams, to keep individual team sizes within the recommended guideline and help break the large project up into smaller sub-projects. The Team is expected to collaborate and work together to deliver the features to deploy whatever functionality has been committed to in the Sprint.

## AGILE ARTIFACTS

The **Product Backlog** (sometimes called a parking lot, wish list, or to-do list in non-Agile projects) is a collection of requirements typically in User Story form. In Agile fashion, these User Stories start out short and simple, but are fine-tuned with additional details as part of managing the Backlog. The Product Backlog will grow and evolve throughout the project as more is learned about the product and client's requirements change.

**User Stories** are a non-technical format that describes a feature in context of a given type of user and specifies the acceptance criteria for determining when the feature has been successfully delivered. Focusing on individual user needs helps the team to understand why a feature is important to a given user, and also ensures that each feature provides value to a user upon delivery. The team also provides high-level estimates for each backlog feature, allowing the product owner and client to re-prioritize based on the level of effort.

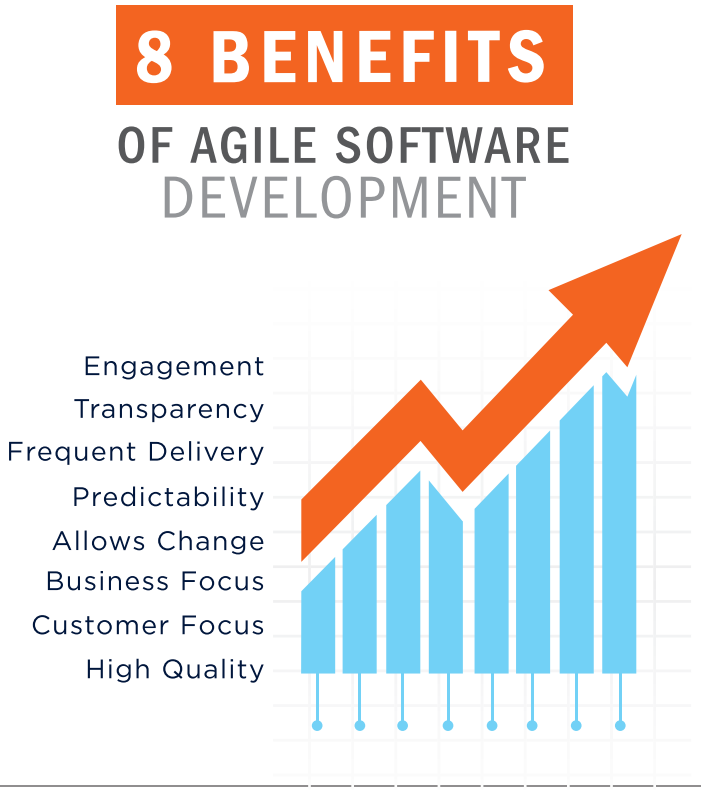
A **Sprint Backlog** is similar to a Product Backlog, except that it contains only the features that are being tackled in a particular Sprint. Features not completed in one Sprint may rollover into the next Sprint, or can be pushed back into the Product Backlog to deal with in a later Sprint.

A **Sprint Burndown** graph is a graphic representation of the amount of remaining work in the Sprint. It gives the team a quick visual check on whether or not they are on schedule to complete the planned work within the Sprint timeframe.

Incorporating Agile methodologies into your software development process can have a big impact on the overall success of your program, as well as the interim usefulness of your development and investment. Feedback and correction can occur quickly to fix small glitches before they become big problems. Communication throughout the process is also improved by the Agile approach to project management. Overall, Agile provides a lean and effective model for the successful development of software.

# 8 Benefits of Agile Software Development

In the previous chapter, we discussed a number of benefits to using an agile process to manage software development projects. In this chapter we will expand upon these benefits and illustrate why they are compelling reasons to consider Agile. These benefits of agile software development include:



**1. Stakeholder Engagement**

Agile provides multiple opportunities for stakeholder and team engagement – before, during, and after each Sprint. By involving the client in every step of the project, there is a high degree of collaboration between the client and project team, providing more opportunities for the team to truly understand the client's vision. Delivering working software early and frequently increases stakeholders' trust in the team's ability to deliver high-quality working software and encourages them to be more deeply engaged in the project.

**2. Transparency**

An  Agile approach provides a unique opportunity for clients to be involved throughout the project, from prioritizing features to iteration planning and review sessions to frequent software builds containing new features. However, this also requires clients to understand that they are seeing a work in progress in exchange for this added benefit of transparency.

**3. Early and Predictable Delivery**

By using time-boxed, fixed schedule Sprints of 1-4 weeks, new features are delivered quickly and frequently, with a high level of predictability. This also provides the opportunity to release or beta test the software earlier than planned if there is sufficient business value.

**4. Predictable Costs and Schedule**

Because each Sprint is a fixed duration, cost is predictable and limited to the amount of work that can be performed by the team in the fixed-schedule time box. Combined with the estimates provided to the client prior to each Sprint, the client can more readily understand the approximate cost of each feature, which improves decision making about the priority of features and the need for additional iterations.

**5. Allows for Change**

While the team needs to stay focused on delivering an agreed-to subset of the product's features during each iteration, there is an opportunity to constantly refine and reprioritize the overall product backlog. New or changed backlog items can be planned for the next iteration, providing the opportunity to introduce changes within a few weeks.

**6. Focuses on Business Value**

By allowing the client to determine the priority of features, the team understands what's most important to the client's business, and can deliver the features that provide the most business value.

**7. Focuses on Users**

Agile commonly uses user stories with business-focused acceptance criteria to define product features. By focusing features on the needs of real users, each feature incrementally delivers value, not just an IT component. This also provides the opportunity to beta test software after each Sprint, gaining valuable feedback early in the project and providing the ability to make changes as needed.

**8. Improves Quality**
By breaking down the project into manageable units, the project team can focus on high-quality development, testing, and collaboration. Also, by producing frequent builds and conducting testing and reviews during each iteration, quality is improved by finding and fixing defects quickly and identifying expectation mismatches early.

During Segue's own experience of adopting Agile software development practices, we have seen solutions delivered on time and with a higher degree of client and customer satisfaction. By incorporating the ability to change, we have been able to better incorporate feedback from demos, usability testing, and client and customer feedback.

Agile is a powerful tool for software development, not only providing benefits to the development team, but also providing a number of important business benefits to the client. Agile helps project teams deal with many of the most common project pitfalls (such as cost, schedule predictability and scope creep) in a more controlled manner. By reorganizing and re-envisioning the activities involved in custom software development, Agile achieves those same objectives in a leaner and more business-focused way

# Common Problems Encountered When Adopting Agile

There are a number of challenges non-Agile organizations face when attempting to adopt Agile development practices and the Scrum methodology. In this article we will discuss three of the more common ones.

### PROBLEM 1:

**Scrum fails to get traction or is a distraction from the real work of the project.**

In order to be effective, a Scrum Master and as many team members as possible must have start-to-finish experience with team projects of enough duration to have had scheduling delays,

non-project distractions, and requirements drift, among other things. Six months is usually long enough to have experienced at least some of these issues, but a longer duration has a way of compounding them and challenging the team even more. This experience has a way of making the value and purpose of agile practices vividly clear. It helps to have worked on a number of waterfall life-cycle projects and to have been frustrated by the impedance mismatch between how they were managed and how developers actually work.

Without that experience, at least for the Scrum Master, Certified Scrum Master (CSM) training will not have sufficient relevance, and the Scrum Master will probably not be able to guide the team through the many day-to-day decisions that will have to be made. Scrum and agile are practice frameworks, and the unique details of each project must be considered carefully. Experience is keenly important here.

This may well be the source of complaints from experienced developers that Scrum and agile are ineffective. If your project involves a team consisting of subject matter experts, a product owner, developers, and a project manager, then Scrum provides the avenues of collaboration sorely needed for project success, and should be seriously considered for the benefit of the whole team and overall project goals.

## PROBLEM 2:

**Developers accustomed to working autonomously may find that Scrum is unnecessary and slows them down.**

There is no question that Scrum adds some overhead to the development process, as compared to a development process with no formal methodology. By design, Scrum is a management tool for agile projects; intended to give management a meaningful view of the health of the project, and the ability to make management decisions about how to proceed. This, unfortunately, entails some amount of overhead. But, arguably, Scrum, done right, provides more realistic information about the project than traditional tools, helps the team self-manage, and incurs less overhead than traditional tools would incur.

Some projects are better suited for a smaller number of developers working autonomously. Personal Kanban might be a more useful project management tool for these projects. However, when you need to scale up to a team of developers and product owner(s), you need to emphasize collaboration among the team members, and Scrum is an excellent solution for that situation.

Whether to use a collaborative approach like Scrum, or a more individual-based approach, should be based on the nature of the project.

On one hand, if you are building a new product in a new product space with no user base against which to measure it, you might consider a less collaborative approach to development, at least early in the project, to give the inventors a broad range of

freedom. If you have a few visionaries, they may be all you need to steer the project (you might also consider a Lean marketing approach to learn what might win the market).

On the other hand, if you are developing a project based at least in part on any existing solution, especially if subject matter experts already exist, then a collaborative approach like Scrum is the right way to go. You may also want to consider Scrum or another agile approach if the number of people needing to communicate regularly exceeds 3 or 4.

### PROBLEM 3:

**Some development efforts don't easily fit into a time-boxed sprint. Therefore, Scrum doesn't work for me.**

This is a real problem. Several kinds of development resist being meaningfully squeezed into standard size sprints.

Here's a partial list:

- New system architecture
- New complex user interface design
- Database ETL requiring extract, cleanse, transform, stage, and present data

Some of these may take several tries to get something that even works, let alone the best solution. They all have trouble conforming to a sprint-sized effort.

If time-boxed sprints are one of the best ideas of agile, why does it appear that I am suggesting that we need to make exceptions

sometimes? The goal of a sprint is to ensure that all of the sprint's backlog items are completed, tested, and working, and that the sprint delivers its designed functionality, however small, to the end user within the fixed duration allocated to each sprint. Delivering a sprint on time requires good planning, good discipline to stay focused, and good teamwork. One of the problems is with the phrase "to the end user". If the end user is defined as one of the consumers of the application, there are some development tasks that normally take longer than a single sprint.

**But wait!** There are things you can do to make them work within an agile framework. Let us start with 3 problem areas:

**New System Architecture** often involves many different hardware components, many existing software applications, and different layers of an organization's IT and administrative staff. Hardware must be purchased, installed, and made to work. Third-party and in-house applications must be made available for access by the to-be-built application. Security must be implemented according to the organization's current infrastructure. Permissions must be planned and granted for the development team and at least a small SME test team. All of this crosses organizational org-chart boundaries and requires administrative approval. Delays can be caused by hardware not being ordered or not arriving when expected, IT and administrative delays, low support staffing, and many other things not directly under the control of project planners. While virtualization and cloud resources help lessen the burden of setting up the initial operating and development environments, it is easy to see how implementing new system architecture is hard

to predict and is dependent on many external factors which do not lend to fitting it into sprints.

Another example is **Complex UI Design**, which can take many tries to get right, involving both development team and SMEs, and often requires many trials and many errors, involving the creation of a large number of style sheets, mockups, wireframes, graphics, and other design assets.  All of this effort does not lend itself to right-sizing for sprints.
The third common problem area is **Database ETL**, which often requires many layers of work, including extract, cleanse, one or many data transformations, staging, and finally presentation of the data according to requirements. Presentation is the first time that the end user actually sees the output of this work. Again, a big bite for a single sprint.

Each of these scenarios presents a challenge to fitting into a standard sprint. However, there are several ways that even the largest of tasks can be broken down into sprint-sized chunks:

1. **Loosening the definition of "end user":** Open the definition of end user to mean something other than a person using the application. The end user could be the next hardware layer (even if no human sees the interface). It could be the business layer interface, which is one step away from the presentation layer that the end user sees. It could be the cleaned extracted data ready for transformation, something a real end user never sees. By breaking large tasks down into layers, you can split those layers between sprints for more manageable delivery.

2. **Narrowing the "river" of the sprint:** Instead of delivering the entire environment right away, focus on delivering the pieces of the environment that will be needed earliest in the development cycle: if a server (virtual or otherwise) will be supporting capabilities you will not be developing for several months, provisioning that server can be tackled in a later sprint. Instead of tackling ETL for multiple data sources from disparate organizations, focus on one data source at a time. If a data source is sufficiently large, one sprint might focus on extraction & cleansing, the next sprint on transformation, and a third sprint on loading. For additional database-relevant ideas, see *Agile Data Warehousing Project Management* by Ralph Hughes (Morgan Kaufmann, 2013).

3. **Using the idea of a special "Sprint 0":** Sprint 0, which may be shorter or longer than a standard sprint, focuses on all of the tasks necessary to launch a project before any "productive" work can be done. Carefully allowing a sprint to take longer than the normal time-boxed iteration is acceptable as long as it does not become a habit. Allowing some slack when there is inherent uncertainty is a good thing.

Allow us to underscore the value of using the time-boxed sprint:

Parkinson's Law describes the phenomenon that *"Work expands so as to fill the time available for its completion."* Often working on a very specific solution, developers will suddenly come up with a much more flexible generalized solution that would take a bit longer but would be so much more capable.

Sometimes, however, the more generalized solution is just not needed. When you are working under a finite time-boxed constraint, you tend to avoid the not strictly necessary embellishments. The discipline of a time constraint can be an amazingly effective productivity tool. Consider also that breaking larger efforts into smaller time chunks has been proven to be much more likely to succeed. You are less likely to get waylaid, you are more likely to stay focused, and the frequent sanity check keeps you closer to your chosen track. These are among the reasons that agile works.

If you find yourself resisting the adoption of an agile approach, consider whether the issue is primarily a problem with agile itself, or the execution of agile that you are experiencing. Here, I have addressed the top three complaints that we hear. Like anything new, agile methods and Scrum will cause their share of growing pains. But the track records of Agile and Scrum are good, and these approaches will almost always reward the teams that embrace them.

Chapter 3

# The Agile Team

## The Qualities of Highly Effective Scrum Masters

A Scrum Master is a coach and facilitator for a team using Scrum, helping the team to stay focused on the project's goals and removing impediments along the way. A Scrum project uses only 3 roles: Product Owner, Scrum Master, and the Team. While the Product Owner brings the product vision, manages the return on investment (ROI), and guides the product development by determining what to build and in what sequence; the Team actually builds the product using agile practices; and the Scrum Master's job is to facilitate communication and problem resolution, and deliver maximum

value to the customer. A highly effective Scrum Master does this by making sure that all involved have the resources they need, are communicating well, and are shielded from distractions and interruptions.


## A HIGHLY EFFECTIVE SCRUM MASTER:

**1. Focuses the team on the goals of the current iteration, keeping them on track.**

Having a short window in which to deliver working software helps

keep the team focused, but should that not be sufficient, the Scrum Master is there to help by keeping the goals visible on a prominent score board, keeping the daily standup meeting focused on goals, and removing distractions that would otherwise interfere with reaching goals.

**2. Removes barriers that block the team so they can deliver high quality working software.**
The Scrum Master monitors such distractions as too many meetings, unneeded procedural complexity, resource-based delays, or work environment or human factor challenges, and protects the team from them all.

**3. Works with the Product Owner, providing a check and balance between getting more done and maintaining high quality and efficiency.**
The Product Owner is rightfully concerned with receiving the most visible value. The Scrum Master makes sure that this is done, but not at the expense of software quality.

**4. Coaches the team in Scrum project management practices through:**

- Building Organic, self-organizing agile teams and integrating them into the enterprise
- Creating a guiding, shared team vision, project vision, and product vision
- Implementing simple, adaptable methodology rules to deliver business value rapidly and reliably
- Creating open flow and exchange of information among

project team members and external groups

- Maintaining a light touch by supporting team autonomy, flexibility & customer value focus without sacrificing control
- Tracking and monitoring the project for timely and relevant feedback while instituting systemic learning and adaptation

**5. Introduces selected engineering practices and tools to help ensure that each iteration is potentially shippable, including:**

- Automated Builds and Continuous Integration: Reduce time and effort associated with manual builds and the risk of big-bang integrations
- Simple Design and Refactoring: Keep incremental development from leading to poor architectures
- Multi-Level/Automated Testing and Test-Driven Development: Reduce testing time and effort and allow developers to make changes with confidence
- Pair Programming: Increase software quality without impacting time to deliver
- Other industry best practices, standard corporate practices, and procedures which may benefit the project

**6. Encourages collaboration and facilitates Scrum-prescribed collaborations through:**

- Release Planning Sessions: To determine what a Release should include and when it should be delivered
- Iteration Planning Sessions: Elaborate, estimate and prioritize highest-value product deliverables for the next iteration

- Daily Standup Meetings: Very brief meetings to rapidly take the pulse of the project, address challenges, and coordinate activities of the team and with the Product Owner
- Iteration Review / Demo Sessions: Demonstrate completed functionality to interested stakeholders and users to show progress and gain important feedback
- Iteration Retrospective Sessions: Reflect on project and process issues and take action as appropriate. Continuous collaboration and process improvement

As you can see, a Scrum Master has many hats to wear, some in addition to those usually associated with project management. Agile emphasizes people over process, and that is certainly evident in the team-directed focus of a Scrum Master. Agile emphasizes delivering customer value over extensive documentation and other non-value added artifacts and processes, and that is reflected in the Scrum Master's emphasis on engineering practices and focus on delivering working value-driven software. Agile and Scrum promote open communication and active contributions from team members and the Product Owner throughout the project, and a highly effective Scrum Master accomplishes this by encouraging and facilitating on-going verbal collaboration, both formally and informally, and by promoting the use of prominent visual displays of project status. A Scrum Master who executes on the strategies outlined here will indeed be a highly effective Scrum Master, and will truly meet or exceed the customer's expectation of value.

# 7 Habits of Highly Effective Product Owners

The Product Owner is generally regarded as the single most important role on the project. It is the Product Owner who ultimately makes decisions about what features will be included in the system, what order they will be worked on, and when they will be accepted as "Done". Agile projects without an effective Product Owner are doomed to failure. For product-based organizations or on internal projects, having a Product Owner who is an integral part of the project team is fairly standard. For service-based organizations like Segue, where most of the work we do is based on a contract with a customer, it is often that customer who must serve as the Product Owner. Cultivating these seven habits should help any Product Owner, whether an integral part of the team or an external Customer, be effective and help the project succeed.

### 1. Be Involved

This is the single most important habit of an effective Product Owner. The Product Owner must be involved in the project, from outset to completion (and often beyond.) The PO should understand the full scope of system features, should be aware of which features are being developed now and which are down the road, and should actively monitor the daily builds and demos to ensure that user stories are implemented the way the Product Owner intended. Regular and continuous involvement in the project helps ensure that the project team is always working towards delivering business value.

### 2. Be Knowledgeable

The Product Owner must be knowledgeable about the subject matter involved in the system being developed. The project team will often look to the Product Owner for explanations about the customer's business practices, as well as any business rules or customer industry practices which may apply. If the Product Owner is not a Subject Matter Expert (SME), he or she should be able to reach out to other resources to get the necessary information in a timely manner.

### 3. Make Decisions

Ultimately, it is the Product Owner who will be responsible for communicating to the project team any decisions that need to be made by the customer – in particular, about prioritization of work, business process, UI design, and other non-technical matters. While the Product Owner may not always be empowered to make those decisions, he or she must be able to reach back to whoever in the customer's organization has that authority, and is responsible for ensuring that decisions get made and get communicated to the project team.

### 4. Be Responsive

Timely responses to questions, requests for information, acceptance of user story details, and acceptance testing of feature implementations are necessary to ensure the project is completed on time and within budget. Agile teams try to stay fully engaged at all times, so any delay in responding can result in work being done incorrectly, or in extreme cases, can result in the project stalling. A stalled project is more expensive, because the project team still

needs to get paid even though they are not able to do any work. In one extreme example, our project team completed work on a project and was disbanded so they could work on other projects, only to find out that the customer had an issue with Feature X which had been implemented over a month prior. In that month, we had implemented other features which depended on Feature X, all of which had to be redone. Pulling the project team back together negatively impacted other projects, caused stress, and increased the cost for the customer as we had to extend the project duration to account for the rework. A timely review of the user stories and implementation could have prevented all of that.

### 5. Be Willing to Learn Something New

In addition to being knowledgeable about a new system, the Product Owner will be asked to fully participate as an integral part of the software development project. This means a willingness to learn new methods and techniques is required. You might be asked to use a specific requirements tool or work management system that the project team will be using; or work in a way you're not used to working in. You may also learn more about your business processes as you turn a watchful eye on them to make sure the system you're helping to develop meets your needs.

### 6. Ask Questions

The project team will be listening to you as they try to understand the requirements for an application. Please listen to the team when they propose alternate ways to implement what you're asking for; identify technical, schedule, or cost challenges of a particular approach or requirement; or suggest additional features you might

not have previously considered. The members of the project team aren't always going to be subject matter experts in your business area, but they are software development experts and can give a fresh perspective on your processes.

### 7. Don't Assume

One of the biggest risks to any project is assumptions. We all make them, even if we don't realize it. Don't assume the project team knows something about your business that you haven't told them. Don't assume that certain features will be included because "everyone does it that way" (rarely, if ever, does 'everyone' really do it that way!) If you want the system to have a particular feature, be certain to ask for it; if there is a business constraint, tell the team; and if you don't see something you want written down in the user stories, bring it up again so the team can make sure to include it if appropriate.

### Bonus Habit: Be Understanding

Sometimes there will be factors which limit the team's ability to deliver exactly your vision of what you want the system to be. These factors could include cost, schedule impact, technical constraints, legal or industry standards, user interface design principles, or others. A customer once asked us to allow users to upload any number of files, of any type and in any file format, and then have the system automatically merge those files into a single PDF – all within seconds. Implementing this feature would require us to purchase several very expensive third party modules (which were outside of the customer's budget), spend weeks integrating them, and still would only support the most common file formats.

Additionally, in order to meet the response time requirement, the hosting environment would have needed to be scaled up to a size that would cost the customer significantly more than they had budgeted for operational expenses. Even though the customer identified this as a "critical" need, there was simply no way we could deliver that functionality within the scope of the project. After explaining the issue to the Product Owner, he was able to convince his superiors in the customer organization that the feature should be shelved until a later date.

By following these habits, you will be a more effective Product Owner, and the project team will be better enabled to deliver the software you want, on schedule and within the agreed-upon budget.

Chapter 4

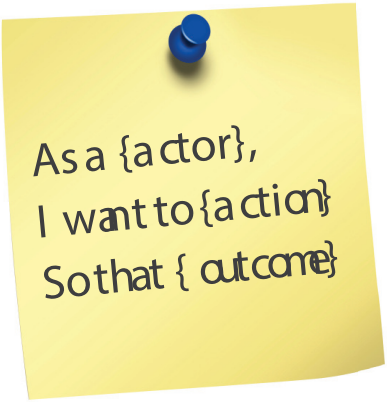
# User Stories

## Creating Effective User Stories

There are many ways to document requirements. In today's Agile-centric world, User Stories have become the popular and preferred method. While not exclusive to an Agile methodology, User Stories can help to express requirements in terms of the business value they provide. We employ Agile practices, and thus utilize User Stories in many of our mobile and website projects. Remember, requirements are meant to guide developers to create an application that fits a real-world business purpose. The software system must ultimately serve a user community, so User Stories are a way to define these requirements in their intended context.

## ANATOMY OF A USER STORY

User Stories are simple, concise ways of detailing requirements from the perspective of the system user. They are composed of three parts: the Card, the Conversation, and the Confirmation.




### THE CARD

The Card is the scenario of the User Story in its simplest form. It gives the Who, What, and Why of the scenario and is usually written out in the following format:

*As a [actor], I want to [action], so that [outcome].*

The **Actor** is the type of user who is interacting with the system. Examples might include “administrator”, “student”, “parent”, “end-user”, or others. It may be helpful to identify user types or roles (sometimes called “personas”) before you start creating user stories.

The **Action** describes what the user wants to perform in the system. Examples might include "log in", "pay my bill", "search for classes", or others. Actions, as their name suggests, should contain active verbs; and should be relatively short and limited in scope. Actions such as "View and pay my bill" should probably be broken up into two User Stories.

The **Outcome** explains why the action is important to the Actor. Examples might include "access my private account information", "stay in contact with my friend", or "pay down my debt." Outcomes could be positive or negative ("so my account doesn't get cancelled" is a perfectly legitimate outcome) but should always describe the value of the user story, from the perspective of the Actor.

### THE CONVERSATION

The conversation fills in the details; this is where the team can elaborate on the User Story. Because the Card is limited to a single sentence following a specific format, the Conversation is critical to understanding the scope and details of the story and provides guidance on how it should be implemented.

### THE CONFIRMATION

Confirmations are the steps or parameters for the User Story. These steps form the Acceptance Criteria that can be used to perform User Acceptance Testing (UAT). We will go into more detail about Acceptance Criteria in the next chapter.

## A HEALTHY USER STORY

Creating User Stories is a seemingly simple process, but what makes a good User Story? For that, we refer to Bill Wake's **INVEST** acronym:

**I is for Independent:**
*The user story should be self-contained, in a way that there is no inherent dependency on another user story.*

User Stories should be independent entities. While completely eliminating dependencies is impossible (after all, they are describing features of the same system - you can hardly describe a hand without somehow addressing the fact that it attaches to the arm) you can minimize dependencies, and by doing so afford for more flexibility in their implementation.

**N is for Negotiable:**
*User stories, up until they are part of an iteration, can always be changed and rewritten.*

A User Story should be negotiable, particularly in the Agile environment. Aspects of a User Story are defined, but the details should be open to continual adjustments. Specifications in a User Story are not fixed upon initial definition; they can and often do change until they are implemented. Once again, it goes back to the flexible and collaborative nature of Agile. When a User Story has been implemented, however, it is closed and if any further changes are needed a new story is created.

**V is for Valuable:**
*A user story must deliver value to the end user.*

All User Stories should be of some value to the actors in the process, be it the system owner, users, financial backers or other effected parties. All parties involved should be able to see the value of why each User Story needs to be implemented. Going back to the User Story format, *"As a [actor], I want to [action], so that [outcome],"* if no discernible value can be seen in the User Story outcome, it should not be implemented.

**E is for Estimable:**
*You must always be able to estimate the size of a user story.*

A team should be able to get together and estimate what it would take to implement a User Story. Estimation is valuable in determining the prioritization of User Stories and determining how to split the level of effort between sprints. If the team cannot, estimate a User Story, it probably needs to be reworked because it is too large, too vague, or there is not enough information available.

**S is for Size-Appropriate:**
*User stories should not be so big as to become impossible to plan/task/prioritize with a certain level of certainty.*

User Stories should be small enough to implement in chunks. Exactly how small is dependent upon the team, but they should be small enough that X number can be implemented within a sprint, so a meaningful amount of functionality can be delivered to the client.

**T is for Testable:**
*The user story or its related description must provide the necessary information to make test development possible.*

If a User Story cannot be tested, how do you know it is Done? This goes back to the Acceptance Criteria listed in the Confirmation. The Team needs to collaborate up front to ensure that testable Acceptance Criteria are developed for the User Stories. This lets the Developers know what their code will be tested against, gives the testers guidelines on how to test, and gives the Team a measure for when a User Story truly is done.

## DIAGNOSIS

Healthy requirements form the foundation of development. Be sure to make your User Stories independent, negotiable, valuable, estimable, size appropriate and testable. Investing upfront in the health of your User Stories will make for a more effective development process.

## A CASE IN POINT

On a project one of our Analysts had previously worked on, a team member wrote the word “Breadcrumbs” on a Post-it® and subsequently added it to the Project Backlog. This single word was the sum total content of what was supposed to be a User Story.

So how does one turn “Breadcrumbs” into something actionable, implementable, and testable? The Book of Agile says that good

User Stories should focus on the user, facilitate a conversation, be simple and concise, have actionable acceptance criteria, and be testable. To focus on the user, "Breadcrumbs" should tell a story about the person wanting to use them. The story should be written from the user's perspective and employ personas or actors such as "user", "manager", or "buyer"; and should describe what action the actor needs to perform and why. Applying this user story template, "Breadcrumbs" became:

*"As a new user, I want "breadcrumbs" on each screen so that I know where I am and how I got there."*

Now that we have expressed the, "what and why" of our user story, how does the project team know when they have achieved their goal? How can we demonstrate that the user story has been successfully achieved? To accomplish this, we still need actionable, testable Acceptance Criteria.

Applying these guidelines to "Breadcrumbs," we might arrive at the following two **Acceptance Criteria**:

**AC 1:** I can see a breadcrumbs trail on the current screen page which shows me the navigation path from the Home page to where I am.

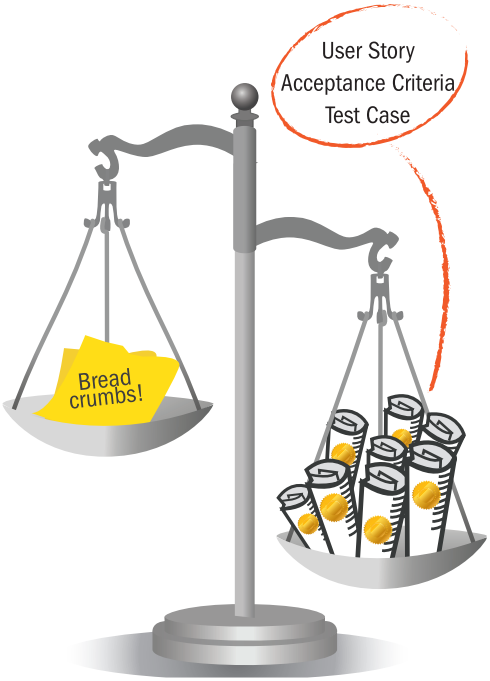
**AC 2:** I can click on a segment of the breadcrumbs trail and I will be taken to the screen page that corresponds to that breadcrumb trail segment.

These Acceptance Criteria are readily actionable and testable, with **Test Cases** easily written to verify each of these conditions of satisfaction:

**TC 1:** From the Home page, click on the Reports menu option, then the Dashboard entry. Expected Results: the Dashboard page displays with breadcrumb trail: Home>Reports>Dashboard.

**TC 2:** From the Dashboard page, click on the 'Reports' segment of the Home>Reports>Dashboard breadcrumb trail. Expected Results: the Reports menu page displays, with its associated breadcrumb Home>Reports."

We have gone from "Breadcrumbs" on a Post-it® to a more mature, **INVEST**ed User Story.



# Characteristics of Good Agile Acceptance Criteria

Good Acceptance Criteria will help get your Agile project from *"It Works as Coded"* to *"It Works as Intended."*


A User Story is a description of an objective a person should be able to achieve, or a feature that a person should be able to utilize, when using a software application. A User Story cannot stand alone, however. It must be accompanied by "good" Acceptance Criteria to provide a way to clearly demonstrate if the Project Team has indeed made the User Story come true.

## WHAT ARE THESE ACCEPTANCE CRITERIA AND WHAT MAKES A "GOOD" ONE?

Microsoft Press defines Acceptance Criteria as *"Conditions that a software product must satisfy to be accepted by a user, customer or other stakeholder."* Google defines them as *"Pre-established standards or requirements a product or project must meet."*

Acceptance Criteria are a set of statements, each with a clear pass/fail result, that specify both functional (e.g., minimal marketable functionality) and non-functional (e.g., minimal quality) requirements applicable at the current stage of project integration. These requirements represent "conditions of satisfaction." There is no partial acceptance: either a criterion is met or it is not.

These criteria define the boundaries and parameters of a User Story/feature and determine when a story is completed and working as expected. They add certainty to what the team is building.

Acceptance Criteria must be expressed clearly, in simple language the customer would use, just like the User Story, without ambiguity as to what the expected outcome is: what is acceptable and what is not acceptable. They must be testable: easily translated into one or more manual/automated test cases.

Acceptance Criteria may reference what is in the project's other User Stories or design documents to provide details, but should not be a re-hash of them. They should be relatively high-level while still providing enough detail to be useful.

They should include:

- **Functional Criteria:** Identify specific user tasks, functions or business processes that must be in place. A functional criterion might be "A user is able to access a list of available reports."
- **Non-functional Criteria:** Identify specific non-functional conditions the implementation must meet, such as design elements. A non-functional criterion might be "Edit buttons and Workflow buttons comply with the Site Button Design."
- **Performance Criteria:** If specific performance is critical to the acceptance of a user story, it should be included. This is often measured as a response time, and should be spelled out as a threshold such as "<2 seconds for a query response."

Acceptance Criteria should state intent, but not a solution (e.g., "A manager can approve or disapprove an audit form" rather than "A manager can click an 'Approve/Disapprove' radio button to approve an audit form"). The criteria should be independent of the implementation; ideally the phrasing should be the same regardless of target platform.

**An Example**

As a Company Administrator, I want to create User Accounts so that I can grant my employees access to the system. Acceptance Criteria for this User Story might look like the following:

1. Only Company Administrator users can access this feature.

2. Company Administrators can only create User Accounts within their Company hierarchy.
3. I can create a User Account by entering the following information about the User:
    a. Name
    b. Email address
    c. Phone Number
    d. Access Level
    e. Reports to (from a list of other User Accounts within my Company hierarchy)
4. I cannot assign a new User to report to a User in another Company hierarchy
5. I cannot assign a new User to report to a User if it creates a cyclical relationship at any level (e.g., User 1 --> User 2 --> User 1; or User 1 --> User 2 --> User 3 --> User 1.)
6. The system sends an email to the new User's email address, containing a system-generated initial password and instructions for the person to log in and change their password.

Apply these ideas to your Agile project and you will quickly transform it from *"It Works as Coded"* to *"It Works as Intended."*

# User Stories vs. Use Cases: The Pros and Cons

It's a debate that has been raging in the world of Agile development for years: User Stories vs. Use Cases. Are they the same thing? If not, which is better? Which is necessary? Should you use one or both?

As outlined in a 2012 post from Boost New Media, the general consensus is that User Stories and Use Cases are not the same thing. They may achieve the same outcome, but they do so in very different ways.

## USER STORIES

According to the Boost post, a User Story is a short description of what your user will do when they come to your website or use your software. Dice goes a little more in-depth with that description, stating that it's a high level or conceptual scenario. The example used is that a user needs to be able to save a report in two different formats. While the formats are different, Dice explains, the scenario is the same.

The Use Case Blog states that User Stories often start out the same way as Use Cases, in that each describes one way to use the system, is centered around a goal, is written from the perspective of a user, uses the natural language of the business, and -- on its own -- does not tell the whole story.

**The pros:** according to Future of CIO, it's an informal process that should start with a simple sentence. As a (blank), I want to be able to (function of the system), so that (goal) can be achieved. By stating this sentence, the User Story then becomes the starting point by which Use Cases can be derived. User Stories are especially helpful for those who want the agility of adding value sooner and in smaller increments, explains The Use Case Blog. A User Story doesn't have to be simple, however. According to Future of CIO, by using high level User Stories, User Stories can make for more productive planning sessions and a versatile way to add last-minute functions to the project.

**The cons:** All About Agile states that one of the cons to using User Stories is that they often leave out a lot of details, relying instead on their conversational method of relaying details and time of development to the customer. This can be time consuming as this documentation is not complete upfront, as it is with Use Cases, but rather relies on collaboration that may or may not be present.

## USE CASES

So what is a Use Case, then? It's a set of interactions, Boost explains, between a system and one or more actors, with actors being people, other systems, or both. It's a complete specification of all possible scenarios, writes Future of CIO. According to Dice, it's a way of capturing the process flow or the steps involved in generating a report and the expected outcomes or alternatives. It's a functional requirement that describes not only a behavior, but how that behavior can be achieved.

**The pros:** The Agile Machine writes that Use Cases have gone out of favor to the detriment of formalized concepts. Some of the concepts that the Use Case provides include the identification of actors and the ability to break the problem down into subdomains. In addition, Boost notes, there are times when the upfront research that is required for Use Cases is beneficial to the project and should be undertaken.

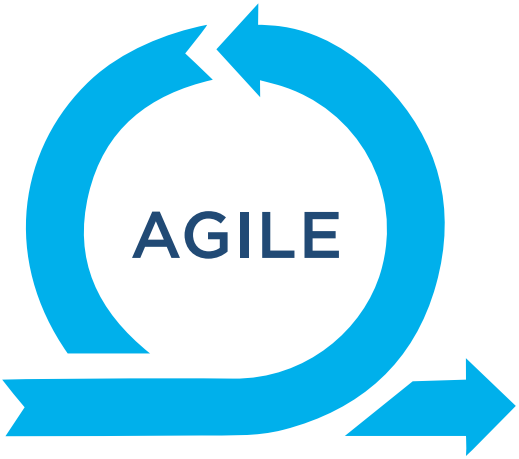
**The cons:** Use Cases are meant to provide such a formalized blueprint of the project, Future of CIO explains, that they often leave little room for negotiation or project additions. Additionally, states All About Agile, the Use Case can get a bit complicated and isn't a format that is generally palatable for end users or business people.

## ONE, THE OTHER, OR BOTH?

Whether your project requires a User Story, Use Case, or both, depends on the project, the collaboration available, the level of formality, and the upfront research required. Some have found success in a hybrid, such as a highly detailed User Story, while others find the User Story as an important launching point for the more detailed Use Case.

At Segue, we tend to employ Use Cases predominantly with government projects that have more stringent documentation requirements. For smaller, short duration projects we tend to lean towards User Stories. The many-years debate shows not so much that one is better than the other but that each can be applicable in varying degrees from project to project.

# Reconciling Agile User Stories with Formal Requirements Documents




Another common challenge encountered by organizations attempting to adopt Agile is Requirements Management. Different development methodologies have their own approaches to tracking project requirements and ensuring the development team produces a desired product for the customer. The path from concept to completion may take different routes per the type of project or the chosen methodology of the team, but regardless, it critical to have some form of requirements as a point of shared understanding and agreement between the team and the customer. This allows the project to not only proceed as desired, but serve as a point of reference for disagreements over the software as it comes to life, a standard by which to test against, and a starting point for any changes in direction that may occur during the project.

## TRADITIONAL REQUIREMENTS APPROACH

In traditional BRUF (Big Requirements Up Front) or Waterfall development, the project team creates a monolithic requirements document, often called a Software Requirements Specification (SRS) or Software Requirements Definition (SRD), before development work begins; and will update that document as change occurs during the rest of the development lifecycle. The requirements document always represents *"the Truth, the Whole Truth, and Nothing But the Truth"* for the software system. If the requirements document says something, you can count on it to be true – and if the software doesn't work as described in the SRS, it's a bug. If there is a change in what the software needs to do, it's a change request – which must then be reflected by updating the requirements document by creating or modifying any number of individual requirements.

## HOW DOES AGILE ACCOUNT FOR REQUIREMENTS?

In Agile development, we use User Stories instead of a monolithic requirements document. User stories are typically retained only until they are "Done", and are then discarded. A user story is considered "Done" when it meets all the criteria the development team has set in their Definition of Done. Typically, "Done" includes some variant of *"the user story has been implemented, tested, and accepted by the Product Owner."* If the software doesn't work as described... well, once a User Story is Done it's Done, and there's no such thing as *"doesn't work as described."* Instead, everything is *"doesn't work as desired,"* and results in a new User Story reflecting the desired change.

The Agile User Story approach to requirements is all well and good for product-based teams with full ownership of the project, but for service-based teams building software on contract for a customer, things can get much trickier. Often, when a customer is looking at a software release, either for Acceptance testing or in Production, seeing something that doesn't match their expectations may send them searching for some sort of requirements document or specification to see *"is this what we agreed on."* This is particularly important when it comes to determining whether the customer will have to pay for the "correction" or not; and can even be critical in systems such as software operating medical devices, financial systems, or other similarly important systems. With User Stories, which the Scrum Alliance describes as *"not a highly documented series of requirements but rather a reminder to collaborate about the topic of the user story"* it can be very difficult to get a full picture of *the Truth, the Whole Truth, and Nothing But the Truth* – you have to gather up all the User Stories affecting a particular feature and rebuild that picture of Truth.

## COMMON GROUND – USER STORIES FOR SERVICE BASED PROJECTS

Getting customers on board with the Agile way can be difficult, particularly when those customers are not tech-savvy or have no experience with software development projects. For those customers, it's important to be able to look back to some kind of requirements document and know *"this is the agreed-upon Truth."* To further complicate matters, when the customer is the Government there may be laws, regulations, policies, or contractual terms which require creation of a formal requirements document.

In these scenarios, it can be helpful to have a way to transform User Stories into a requirements document, and ensure the requirements document stays up to date as the project evolves. When starting a new project, the User Stories and the Requirements Document will be similar in terms of content, although the structure may differ. As user stories are refined before implementation, the requirements document should be updated to reflect that refinement. Once a user story is "Done" it can be closed and forgotten about (or even destroyed, if you are keeping User Stories on index cards or post-it notes). When a new User Story is created, whether to add a new feature or change an existing feature, the Requirements Document should be updated to reflect the change. In this manner, at any time during the development lifecycle, you can look at something which reflects *the Truth, the Whole Truth, and Nothing But the Truth*, while still having the flexibility and collaborative aspects of User Stories.

Keeping User Stories and a Requirements Document in sync is extra work, but that should pay off in the long run as you can ensure that everyone, including the customer, is working from the same definition of Truth.

## ABOUT THE AUTHORS

***John Freeman*** *has extensive software development experience, encompassing the full life-cycle. His recent technical focus has been on databases and associated technologies, with some forays into user interface design where needed. He has successfully taken applications to market, and was application architect on a series of projects for State and Federal clients. He has served as chief application architect and collaborated with a team, and has had good success working alone on special projects. He is an agile evangelist and Certified Scrum Master, and has both published and presented on these topics. With Segue, he is making use of his database skills in support of Air Force financial applications, and continues to promote good agile practices.*

***Mark Shapiro*** *is Segue's Senior Software Architect and has been with the company since 1998. He has a Master's Degree in Software Engineering and over 20 years of work experience in a wide array of languages including Perl, PHP, Java, Delphi (Object Pascal), and C#, and once helped build the network for a major telephone company's dialup ISP. In his spare time he enjoys scuba diving and playing with his Alaskan Malamute.*

***Mary Lotz*** *is Segue's Director of Engineering. She is a certified project manager (PMP) and scrum master (CSM), and has been directing application development teams and projects for a variety of customers and industries for over 15 years. A former application developer, Mary holds both B.S. and M.S. degrees in Applied Computer Science.*

***Nicole Pearson*** *is an analyst at Segue focusing on requirements and testing for predominantly government applications. A graduate from University of Maryland, she enjoys reading science fiction and designing LEGO sets in her spare time.*

*This eBook contains content provided by former employees of Segue Technologies including* ***Adam Zolyak***, ***James Peterson***, *and* ***Walter Jackson***.

Thank you for reading our eBook. We hope you found it informative and interesting. If you are interested in working with Segue, please contact us so we can learn about your needs and plan a development path that works for you.

**Segue Technologies provides:**

Web

- Custom Application Development
- User Interface Design
- Content Management Systems

Data

- Business Intelligence
- Data Quality and Profiling
- Enterprise Data Management

Mobile

- iOS Development
- Android Development
- Mobile Web