"Agile Development" is an umbrella term for several iterative and incremental software development methodologies.
Contents

Agile Development .................................................................................................................. 6
   What is Agile Development? ............................................................................................... 6
   The Evolution of Agile Development .................................................................................... 6
Agile Methodologies ............................................................................................................. 7
   Scrum ...................................................................................................................................... 7
   Extreme Programming (XP) ................................................................................................. 7
   Crystal ................................................................................................................................... 8
   Dynamic Systems Development Method (DSDM) ............................................................. 9
   Feature-Driven Development (FDD) ..................................................................................... 9
   Lean Software Development ............................................................................................... 10
Benefits of Agile Development ............................................................................................... 11
   Problems with Traditional Development ............................................................................ 12
Agile Hallmarks ...................................................................................................................... 13
   1. Releases and Fixed-Length Iterations ............................................................................. 13
   2. Running, Tested Software ............................................................................................. 14
   3. Value-Driven Development ........................................................................................... 15
   4. Continuous (Adaptive) Planning .................................................................................... 15
   5. Multi-Level Planning ..................................................................................................... 16
   6. Relative Estimation ......................................................................................................... 16
   7. Emergent Feature Discovery .......................................................................................... 17
   8. Continuous Testing ......................................................................................................... 17
   9. Continuous Improvement ............................................................................................... 18
  10. Small, Cross-functional Teams ....................................................................................... 18
Customer and Management Practices .................................................................................. 19
   Planning and Managing the Project .................................................................................... 19
Feature Estimation .................................................................................................................. 19
   What is a Feature? .............................................................................................................. 19
   Feature Breakdown Structure (FBS) ................................................................................... 20
   Building an Initial Feature List .......................................................................................... 20
   Feature Headline ................................................................................................................ 21
   Organizing Features ........................................................................................................... 21
   Accounting for Risk ............................................................................................................ 21
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test-First Technique and Tools</td>
<td>40</td>
</tr>
<tr>
<td>Refactoring</td>
<td>41</td>
</tr>
<tr>
<td>Code Hygiene</td>
<td>41</td>
</tr>
<tr>
<td>Specific &quot;Refactorings&quot;</td>
<td>41</td>
</tr>
<tr>
<td>Refactoring to Patterns</td>
<td>42</td>
</tr>
<tr>
<td>The Flow of Refactoring</td>
<td>42</td>
</tr>
<tr>
<td>Refactoring Automation in IDEs</td>
<td>42</td>
</tr>
<tr>
<td>Continuous Integration</td>
<td>43</td>
</tr>
<tr>
<td>Continuous Integration Technique, Tools, and Policy</td>
<td>43</td>
</tr>
<tr>
<td>Benefits of Continuous Integration</td>
<td>43</td>
</tr>
<tr>
<td>Simple Design</td>
<td>44</td>
</tr>
<tr>
<td>The Art of What Suffices</td>
<td>44</td>
</tr>
<tr>
<td>&quot;You Aren't Gonna Need It&quot;</td>
<td>44</td>
</tr>
<tr>
<td>Pair Programming</td>
<td>45</td>
</tr>
<tr>
<td>Pairing Mechanics</td>
<td>45</td>
</tr>
<tr>
<td>Spreading Knowledge</td>
<td>45</td>
</tr>
<tr>
<td>Pairing and Productivity</td>
<td>45</td>
</tr>
<tr>
<td>Productivity and Staff Turnover</td>
<td>46</td>
</tr>
<tr>
<td>Pairing Strategies</td>
<td>46</td>
</tr>
<tr>
<td>Common Codebase</td>
<td>47</td>
</tr>
<tr>
<td>Single Code Standard</td>
<td>47</td>
</tr>
<tr>
<td>Open Work Area</td>
<td>47</td>
</tr>
<tr>
<td>Are You Agile?</td>
<td>48</td>
</tr>
<tr>
<td>Evaluating Agility</td>
<td>48</td>
</tr>
<tr>
<td>You might not be agile if</td>
<td>49</td>
</tr>
<tr>
<td>General Resources</td>
<td>50</td>
</tr>
<tr>
<td>Must Read</td>
<td>50</td>
</tr>
<tr>
<td>Definitions of Agile Software Development</td>
<td>50</td>
</tr>
<tr>
<td>Websites</td>
<td>50</td>
</tr>
<tr>
<td>Books on Agile Development</td>
<td>50</td>
</tr>
<tr>
<td>Agile Software Development</td>
<td>50</td>
</tr>
<tr>
<td>Agile Development Management</td>
<td>51</td>
</tr>
<tr>
<td>Crystal</td>
<td>51</td>
</tr>
<tr>
<td>DSDM</td>
<td>51</td>
</tr>
</tbody>
</table>
Extreme Programming ........................................................................................................51
Feature-Driven Development (FDD) ..................................................................................51
Lean Development ...........................................................................................................52
Scrum ...............................................................................................................................52
Agile Development

What is Agile Development?

"Agile Development" is an umbrella term for several iterative and incremental software development methodologies. The most popular Agile methodologies include Extreme Programming (XP), Scrum, Crystal, Dynamic Systems Development Method (DSDM), Lean Development, and Feature-Driven Development (FDD).

While each of the Agile methods is unique in its specific approach, they all share a common vision and core values (see the Agile Manifesto). They all fundamentally incorporate iteration and the continuous feedback that it provides to successively refine and deliver a software system. They all involve continuous planning, continuous testing, continuous integration, and other forms of continuous evolution of both the project and the software. They are all lightweight (especially compared to traditional waterfall-style processes), and inherently adaptable. As important, they all focus on empowering people to collaborate and make decisions together quickly and effectively.

The Evolution of Agile Development

Many of the individual principles and practices that are promoted by Agile development have been around for years, even decades. As opposed to implementing these best practices piecemeal, Agile methodologies have "packaged" various customer, management, and in some cases, engineering practices and principles together in a way that helps guide teams through the process of rapidly planning and delivering working, tested software. Each of the Agile methodologies combines both old and new ideas into refinements that are certainly greater than the sums of their parts.

While it is true that many of the practices associated with Agile development have been around for quite some time, the average software development team has yet to embrace many of the principles and practices. Even today, the average software team does not iterate, does not deliver software incrementally, and does not practice continuous planning nor automate testing. Now that these practices have been combined in a manner that can more easily be understood and adopted, the trend appears to be rapidly changing for the better, especially during the last several years.

As with any new way of doing business though, Agile methods have generated quite a bit of controversy within the software community. Yet since their emergence, in project after project, they have continued to deliver higher quality software systems in less time than traditional processes. If you are a software development professional, you definitely owe it to yourself to at least become familiar with the theory and practice of Agile development. Hopefully the information presented on this site can assist.
Agile Methodologies

The various Agile methodologies share much of the same philosophy, as well as many of the same characteristics and practices (as we discuss separately on the site). But from an implementation standpoint, each has its own recipe of practices, terminology, and tactics. Here we have summarized a few of the main contenders these days:

- Scrum
- Extreme Programming (XP)
- Crystal
- Dynamic Systems Development Method (DSDM)
- Feature-Driven Development (FDD)
- Lean Software Development

Scrum

Scrum is a lightweight management framework with broad applicability for managing and controlling iterative and incremental projects of all types. Ken Schwaber, Mike Beedle, Jeff Sutherland and others have contributed significantly to the evolution of Scrum over the last decade. Over the last couple of years in particular, Scrum has garnered increasing popularity in the software community due to its simplicity, proven productivity, and ability to act as a wrapper for various engineering practices promoted by other Agile methodologies.

Ken Schwaber, has depicted the Scrum process in a diagram on the Control Chaos website.

In Scrum, the "Product Owner" works closely with the team to identify and prioritize system functionality in form of a "Product Backlog". The Product Backlog consists of features, bug fixes, non-functional requirements, etc. - whatever needs to be done in order to successfully deliver a working software system. With priorities driven by the Product Owner, cross-functional teams estimate and sign-up to deliver "potentially shippable increments" of software during successive Sprints, typically lasting 30 days. Once a Sprint's Product Backlog is committed, no additional functionality can be added to the Sprint except by the team. Once a Sprint has been delivered, the Product Backlog is analyzed and reprioritized, if necessary, and the next set of functionality is selected for the next Sprint.

Scrum has been proven to scale to multiple teams across very large organizations (800+ people).

Extreme Programming (XP)

XP, originally described by Kent Beck, has emerged as one of the most popular and controversial Agile methods. XP is a disciplined approach to delivering high-quality software quickly and continuously. It promotes high customer involvement, rapid feedback loops, continuous testing,
continuous planning, and close teamwork to deliver working software at very frequent intervals, typically every 1-3 weeks.

The original XP recipe is based on four simple values – simplicity, communication, feedback, and courage – and twelve supporting practices:

1. Planning Game
2. Small Releases
3. Customer Acceptance Tests
4. Simple Design
5. Pair Programming
6. Test-Driven Development
7. Refactoring
8. Continuous Integration
9. Collective Code Ownership
10. Coding Standards
11. Metaphor
12. Sustainable Pace

Don Wells has depicted the XP process in a popular diagram.

In XP, the “Customer” works very closely with the development team to define and prioritize granular units of functionality referred to as "User Stories". The development team estimates, plans, and delivers the highest priority user stories in the form of working, tested software on an iteration by iteration basis. In order to maximize productivity, the practices provide a supportive, lightweight framework to guide a team and ensure high-quality software.

Crystal

The Crystal methodology is one of the most lightweight, adaptable approaches to software development. Crystal is actually comprised of a family of methodologies (Crystal Clear, Crystal Yellow, Crystal Orange, etc.) whose unique characteristics are driven by several factors such as team size, system criticality, and project priorities. This Crystal family addresses the realization that each project may require a slightly tailored set of policies, practices, and processes in order to meet the project’s unique characteristics.

Several of the key tenets of Crystal include teamwork, communication, and simplicity, as well as reflection to frequently adjust and improve the process. Like other Agile methodologies, Crystal promotes early, frequent delivery of working software, high user involvement, adaptability, and the removal of bureaucracy or distractions. Alistair Cockburn, the originator of the Crystal, has created the Crystal website and has also recently released a new book “Crystal Clear: A Human-Powered Methodology for Small Teams”.
Dynamic Systems Development Method (DSDM)

DSDM, dating back to 1994, grew out of the need to provide an industry standard project delivery framework for what was referred to as Rapid Application Development (RAD) at the time. While RAD was extremely popular in the early 1990’s, the RAD approach to software delivery evolved in a fairly unstructured manner. As a result, the DSDM Consortium was created and convened in 1994 with the goal of devising and promoting a common industry framework for rapid software delivery. Since 1994, the DSDM methodology has evolved and matured to provide a comprehensive foundation for planning, managing, executing, and scaling Agile and iterative software development projects.

DSDM is based on nine key principles that primarily revolve around business needs/value, active user involvement, empowered teams, frequent delivery, integrated testing, and stakeholder collaboration. DSDM specifically calls out “fitness for business purpose” as the primary criteria for delivery and acceptance of a system, focusing on the useful 80% of the system that can be deployed in 20% of the time.

Requirements are baselined at a high level early in the project. Rework is built into the process, and all development changes must be reversible. Requirements are planned and delivered in short, fixed-length time-boxes, also referred to as iterations, and requirements for DSDM projects are prioritized using MoSCoW Rules:

- M – Must have requirements
- S – Should have if at all possible
- C – Could have but not critical
- W - Won’t have this time, but potentially later

All critical work must be completed in a DSDM project. It is also important that not every requirement in a project or time-box is considered critical. Within each time-box, less critical items are included so that if necessary, they can be removed to keep from impacting higher priority requirements on the schedule.

The DSDM project framework is independent of, and can be implemented in conjunction with, other iterative methodologies such as Extreme Programming and the Rational Unified Process.

Feature-Driven Development (FDD)

FDD was originally developed and articulated by Jeff De Luca, with contributions by M.A. Rajashima, Lim Bak Wee, Paul Szego, Jon Kern and Stephen Palmer. The first incarnations of FDD occurred as a result of collaboration between De Luca and OOD thought leader Peter Coad. FDD is a model-driven, short-iteration process. It begins with establishing an overall model shape. Then it continues with a series of two-week "design by feature, build by feature" iterations. The features are small, "useful in the eyes of the client" results. FDD designs the rest of the development process around feature delivery using the following eight practices:
1. Domain Object Modelling
2. Developing by Feature
3. Component/Class Ownership
4. Feature Teams
5. Inspections
6. Configuration Management
7. Regular Builds
8. Visibility of progress and results

FDD recommends specific programmer practices such as "Regular Builds" and "Component/Class Ownership". FDD's proponents claim that it scales more straightforwardly than other approaches, and is better suited to larger teams. Unlike other Agile approaches, FDD describes specific, very short phases of work which are to be accomplished separately per feature. These include Domain Walkthrough, Design, Design Inspection, Code, Code Inspection, and Promote to Build.

The notion of "Domain Object Modelling" is increasingly interesting outside the FDD community, following the success of Eric Evans' book Domain-Driven Design.

Lean Software Development

Lean Software Development is an iterative methodology originally developed by Mary and Tom Poppendieck. Lean Software Development owes much of its principles and practices to the Lean Enterprise movement, and the practices of companies like Toyota. Lean Software Development focuses the team on delivering Value to the customer, and on the efficiency of the "Value Stream," the mechanisms that deliver that Value. The main principles of Lean include:

1. Eliminating Waste
2. Amplifying Learning
3. Deciding as Late as Possible
4. Delivering as Fast as Possible
5. Empowering the Team
6. Building Integrity In
7. Seeing the Whole

Lean eliminates waste through such practices as selecting only the truly valuable features for a system, prioritizing those selected, and delivering them in small batches. It emphasizes the speed and efficiency of development workflow, and relies on rapid and reliable feedback between programmers and customers. Lean uses the idea of work product being "pulled" via customer request. It focuses decision-making authority and ability on individuals and small teams, since research shows this to be faster and more efficient than hierarchical flow of control. Lean also concentrates on the efficiency of the use of team resources, trying to ensure that everyone is productive as much of the time as possible. So it concentrates on concurrent work, and the fewest
possible intra-team workflow dependencies. Lean also strongly recommends that automated unit tests be written at the same time the code is written.

**Benefits of Agile Development**

Agile methods grew out of the real-life project experiences of leading software professionals who had experienced the challenges and limitations of traditional waterfall development on project after project. The approach promoted by agile development is in direct response to the issue associated with traditional software development – both in terms of overall philosophy as well as specific processes.

Agile development, in its simplest form, offers a lightweight framework for helping teams, given a constantly evolving functional and technical landscape, maintain a focus on the rapid delivery of business value (i.e., “bang for the buck”). As a result of this focus and its associated benefits, organizations are capable of significantly reducing the overall risk associated with software development.

In particular, agile development accelerates the delivery of initial business value, and through a process of continuous planning and feedback, is able to ensure that value is continuing to be maximized throughout the development process. As a result of this iterative planning and feedback loop, teams are able to continuously align the delivered software with desired business needs, easily adapting to changing requirements throughout the process. By measuring and evaluating status based on the undeniable truth of working, testing software, much more accurate visibility into the actual progress of projects is available. And finally, as a result of following an agile process, at the conclusion of a project is a software system that much better addresses the business and customer needs.

The diagram below displays the differences between agile and waterfall development processes. By delivering working, tested, deployable software on an incremental basis, agile development delivers increased value, visibility, and adaptability much earlier in the lifecycle, significantly reducing project risk.
Problems with Traditional Development

According to the Standish Group’s famous CHAOS Report of 2000, 25% of all projects still fail outright through eventual cancellation, with no useful software deployed. Sadly, this represents a big improvement over previous CHAOS reports from past years. And now there is more evidence of the same kind. In Agile and Iterative Development: a Manager’s Guide, renowned consultant and author Craig Larman does a thorough job of debunking the traditional waterfall model once and for all.

The numbers are overwhelming. A study in the United Kingdom shows that of 1,027 projects, only 13% did not fail, and waterfall-style scope management was the "single largest contributing factor for failure, being cited in 82% of the projects as the number one problem." A 1995 study of over $37 billion USD worth of US Defence Department projects concluded that "46% of the systems so egregiously did not meet the real needs (although they met the specifications) that they were never successfully used, and another 20% required extensive rework" to be usable.

Larman also points that in "another study of 6,700 projects, it was found that four out of the five key factors contributing to project failure were associated with and aggravated by the waterfall model, including inability to deal with changing requirements, and problems with late integration." And finally, another study of over 400 waterfall projects reported that only 10% of the developed code was actually deployed, and of that, only 20% was actually used.
These numbers reinforce what many of us have experienced personally: the waterfall approach is a risky and expensive way to build software systems. This is the real reason why the much of industry is investigating and/or implementing agile alternatives.

**Agile Hallmarks**

Summarized below are several of the key characteristics that successful Agile projects seem to share. For some methodologies these correspond exactly with individual practices, whereas for other methodologies there is a looser correspondence.

### 1. Releases and Fixed-Length Iterations

Agile methods have two main units of delivery: releases and iterations. A release consists of several iterations, each of which is like a micro-project of its own. Features, defects, enhancement requests and other work items are organized, estimated and prioritized, then assigned to a release. Within a release, these work items are then assigned by priority to iterations. The diagram below summarizes this.

The result of each iteration is working, tested, accepted software and associated work items.

Agile development projects thrive on the rhythm or heartbeat of fixed-length iterations. The continuous flow of new running, tested features at each iteration provides the feedback that enables the team to keep both the project and the system on track. Only from the features that emerge from fixed-length ("time-boxed") iterations can you get meaningful answers to questions like "How much work did we do last month compared to what we predicted we would?" and "How much work did we get done compared to the month before?" and our personal favourite, "How many features will we really get done before the deadline?"
The cruelty of several tight, fixed deadlines within a release cycle focuses everyone's mind. Face to face with highly-visible outcomes from the last iteration (some positive, some negative), the team finds itself focused on refining the process for the next iteration. They are less tempted to "gold-plate" features, to be fuzzy about scope, or to let scope creep. Everyone can actually see and feel how every week, every day, and every hour counts. Everyone can help each other remain focused on the highest possible business value per unit time.

The operating mechanics of an Agile development process are highly interdependent. Another way to represent an iterative development process is through a set of interlocking gears that turn at different speeds:

![Agile Planning & Delivery Cycle](image)

Each **day**, the team is planning, working on, and completing tasks (as communicated during daily stand-ups), and software is being designed, coded, tested and integrated for customer acceptance.

Each **iteration**, the team is planning, testing, and delivering working software. Each **release**, the team is planning, testing, and deploying software into production. And throughout the entire process, in order to coordinate and successfully deliver in such a highly adaptive and productive process, team communication and collaboration are critical.

As the iterations go by the team hits its stride, and the heartbeat of iteration deadlines is welcomed, not dreaded. Suddenly, once the team gets the hang of it, there is time for continuous process improvement, continuous learning and mentoring, and other best practices.

### 2. Running, Tested Software

Running, tested features are an Agile team's primary measure of progress. Working features serve as the basis for enabling and improving team collaboration, customer feedback, and overall project visibility. They provide the evidence that both the system and the project are on track.

In early iterations of a new project, the team may not deliver many features. Within a few iterations, the team usually hits its stride. As the system emerges, the application design, architecture, and business priorities are all continuously evaluated. At every step along the way, the team
continuously works to converge on the best business solution, using the latest input from customers, users, and other stakeholders. Iteration by iteration, everyone involved can see whether or not they will get what they want, and management can see whether they will get their money's worth.

Consistently measuring success with actual software gives a project a very different feeling than traditional projects. Programmers, customers, managers, and other stakeholders are focused, engaged, and confident.

3. Value-Driven Development

Agile methods focus rigorously on delivering business value early and continuously, as measured by running, tested software. This requires that the team focuses on product features as the main unit of planning, tracking, and delivery. From week to week and from iteration to iteration, the team tracks how many running, tested features they are delivering. They may also require documents and other artefacts, but working features are paramount. This in turn requires that each "feature" is small enough to be delivered in a single iteration. Focusing on business value also requires that features be prioritized, and delivered in priority order.

Different methodologies use different terminology an techniques to describe features, but ultimately they concern the same thing: discrete units of product functionality.

<table>
<thead>
<tr>
<th>Methodology</th>
<th>Feature Terminology</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extreme Programming</td>
<td>User Stories</td>
</tr>
<tr>
<td>Scrum</td>
<td>Product Backlog</td>
</tr>
<tr>
<td>DSDM</td>
<td>Requirements</td>
</tr>
<tr>
<td>Unified Process</td>
<td>Use Cases &amp; Scenarios</td>
</tr>
<tr>
<td>FDD</td>
<td>Features</td>
</tr>
</tbody>
</table>

4. Continuous (Adaptive) Planning

It is a myth that Agile methods forbid up-front planning. It is true that Agile methods insist that up-front planning be held accountable for the resources it consumes. Agile planning is also based as much as possible on solid, historical data, not speculation. But most importantly, Agile methods insist that planning continues throughout the project. The plan must continuously demonstrate its accuracy: nobody on an Agile project will take it for granted that the plan is workable.

At project launch, the team does just enough planning to get going with the initial iteration and, if appropriate, to lay out a high-level release plan of features. And iterating is the key to continuous planning. Think of each iteration as a mini-project that receives "just-enough" of its own planning. At iteration start, the team selects a set of features to implement, and identifies and estimates each technical task for each feature. (This task estimation is a critical Agile skill.) For each iteration, this planning process repeats.
It turns out that Agile projects typically involve more planning, and much better planning, than waterfall projects. One of the criticisms of "successful" waterfall projects is that they tend to deliver what was originally requested in the requirements document, not what the stakeholders discover they actually need as the project and system unfolds. Waterfall projects, because they can only "work the plan" in its original static state, get married in a shotgun wedding to every flaw in that plan. Agile projects are not bound by these initial flaws. Continuous planning, being based on solid, accurate, recent data, enables Agile projects to allow priorities and exact scope to evolve, within reason, to accommodate the inescapable ways in which business needs continuously evolve.

Continuous planning keeps the team and the system honed in on maximum business value by the deadline.

In the Agile community, waterfall projects are sometimes compared to "fire and forget" weapons, for which you painstakingly adjust a precise trajectory, press a fire button, and hope for the best.

Agile projects are likened to cruise missiles, capable of continuous course correction as they fly, and therefore much likelier to hit a target (a feature-set and a date) accurately.

5. Multi-Level Planning

Continuous planning is much more accurate if it occurs on at least two levels:

- At the release level, we identify and prioritize the features we must have, would like to have, and can do without by the deadline.
- At the iteration level, we pick and plan for the next batch of features to implement, in priority order. If features are too large to be estimated or delivered within a single iteration, we break them down further.

As features are prioritized and scheduled for an iteration, they are broken down into their discrete technical tasks.

This just-in-time approach to planning is easier and more accurate than large-scale up-front planning, because it aligns the level of information available with the level of detail necessary at the time. We do not make wild guesses about features far in the future. We don't waste time trying to plan at a level of detail that the data currently available to us does not support. We plan in little bites, instead of trying to swallow the entire cow at once.

6. Relative Estimation

Many Agile development teams use the practice of relative estimation for features to accelerate planning and remove unnecessary complexity. Instead of estimating features across a spectrum of unit lengths, they select a few (3-5) relative estimation categories, or buckets, and estimate all features in terms of these categories. Examples include:
• 1-5 days
• 1, 2, or 3 story points
• 4, 8, 16, 40, or 80 hours

With relative estimation, estimating categories are approximate multiples of one another. For example, a 3-day feature should take 3 times as long as a 1-day feature, just as a 40-hour feature is approximately 5 times as time-consuming as an 8-hour feature. The concepts of relative estimation and/or predefined estimation buckets prevent the team from wasting time debating whether a particular feature is really 17.5 units or 19 units. While each individual estimate may not be as precise, the benefit of additional precision diminishes tremendously when aggregated across a large group of features. The significant time and effort saved by planning with this type of process often outweighs any costs of imprecise estimates. Just as with everything else in an Agile project, we get better at it as we go along. We refine our estimation successively.

If a feature exceeds an agreed maximum estimate, then it should be broken down further into multiple features. The features generated as a result of this planning ultimately need to be able to be delivered within a single iteration. So if the team determines that features should not exceed 5 ideal days, then any feature that exceeds 5 days should be broken into smaller features. In this way we "normalize" the granularity of our features: the ratio of feature sizes is not enormous.

7. Emergent Feature Discovery

As opposed to spending weeks or months detailing requirements before initiating development, Agile development projects quickly prioritize and estimate features, and then refine details when necessary. As features for an iteration are described in more detail by the customers, testers, and developers working together. Additional features can be identified, but no feature is described in detail until it is prioritized for an iteration.

8. Continuous Testing

With continuous testing we deterministically measure progress and prevent defects. We crank out the running, tested features. We also reduce the risk of failure late in the project. What could be riskier than postponing all testing till the end of the project? Many waterfall projects have failed when they have discovered, in an endless late-project "test-and-fix" phase, that the architecture is fatally flawed, or the components of the system cannot be integrated, or the features are entirely unusable, or the defects cannot possibly be corrected in time. With continuous testing we more easily avoid both the risk that this will occur, and the constant dread of it.

At both the unit level and acceptance (feature) level, we write the tests as the code itself is written or (better yet) beforehand. The most Agile of Agile projects strive to automate as many tests as possible, relying on manual tests only when absolutely necessary. This speeds testing and makes it more deterministic, which in turn gives us more continuous and more reliable feedback.
emerging wealth of new tools, techniques, and best practices for rigorous continuous testing; much of the innovation is originating in the Test-Driven Development (TDD) community.

When is a feature done? When all of its unit tests and acceptance tests pass, and the customer accepts it. This is exactly what defines a running, tested feature. There is no better source of meaningful, highly-visible project metrics.

9. Continuous Improvement

We continuously refine both the system and the project by reflecting on what we have done (using both hard metrics like running, tested features and more subjective measures), and then adjusting our estimates and plans accordingly. But we also use the same mechanism to successively refine and continuously improve the process itself.

Especially at the close of major milestones (iterations, releases, etc.), we may find problems with iteration planning, problems with the build process or integration process, problems with islands of knowledge among programmers, or any number of other problems. We look for points of leverage from which to shift those problems.

We adjust the factory’s machines, and acquire or invent new ones, to keep doing it a little better each release. We keep finding ways to adapt the process to keep delivering a little more value per unit time to the customer, the team, and the organization. We keep maturing and evolving, like any healthy organism.

10. Small, Cross-functional Teams

Smaller teams have been proven to be much more productive than larger teams, with the ideal ranging from five to ten people. If you have to scale a project up to more people, make every effort to keep individual teams as small as possible and coordinate efforts across the teams. Scrum-based organizations of up to 800 have successfully employed a "Scrum of Scrums" approach to project planning and coordination.

With increments of production-ready software being delivered every iteration, teams must also be cross-functional in order to be successful. This means that a team needs to include members with all of the skills necessary to successfully deliver software, including analysis, design, coding, testing, writing, user interface design, planning, and management. We need this because, again, each iteration is its own mini-project.

Teams work together to determine how best to take advantage of one another's skills and mentor each other. Teams transition away from designated testers and coders and designers to integrated teams in which each member helps do whatever needs doing to get the iteration done. Individual team members derive less personal identity from being a competitive expert with a narrow focus, and increasingly derive identity and satisfaction from being part of an extraordinarily productive and
efficient team. As the positive reinforcement accumulates from iteration to iteration, the team becomes more cohesive. Ambient levels of trust, camaraderie, empathy, collaboration, and job satisfaction increase. Software development becomes fun again. These outcomes are not guaranteed, but they are much likelier in well-managed Agile projects than elsewhere.

Customer and Management Practices

You can divide Agile practices roughly into 1) customer and management practices, which address Agile requirements analysis, planning and project management, and 2) programmer practices, which address how to work with the code in an Agile way. Most of the Agile methods focus primarily on customer and management practices.

Planning and Managing the Project

Though specific practices may vary between the different Agile methods, there are many common practices and principles. The most basic management practice common to all Agile methods is to iterate early and regularly. Agile projects have releases of some length, and iterations of some length. These releases and iterations must be planned. An Agile team begins a project by identifying and prioritizing a superset of features. It then plans a release (roughly) and the first iteration. From then on, iteration by iteration, the team uses continuous planning to refine the scope of that release plan as new information is discovered and requirements change. The iteration plans should also get more and more accurate and precise as the team refines and normalizes its velocity (the measured work it accomplishes per unit time). So the management practices give managers and the entire team a better and better sense for exactly which features will be delivered by the deadline. We've covered a few of these customer and management practices in articles listed in the menu to the left. A separate menu to the lower left covers Programmer Practices.

Feature Estimation

What is a Feature?

In Agile development, a feature is a chunk of functionality that delivers business value. Features can include additions or changes to existing functionality. For planning purposes, some methodologies also use the notion of "work items" that can include features, bug fixes, documents, and other artefacts. But features are the main unit of planning. Ideally, a feature should adhere to the following criteria:

1. It should provide business value
2. It should be estimable - it must have enough definition for the development team to provide an estimate of the work involved in implementing it
3. It should be **small enough** to fit within an iteration - therefore, if it is too big, it should be broken down further
4. It should be **testable** - you should understand what automated or manual test a feature should pass in order to be acceptable to the customer

The different methodologies use different terminology to refer to features. It is up to the team to decide which methodology or terminology to use. Extreme Programming (XP) uses the terms User Stories or Stories to represent features; Scrum uses Product Backlog to describe a feature list; Feature-Driven Development uses Feature; and DSDM uses Requirement. Similarly, there are various lightweight versions of the Unified Process, or Agile UP, that use Requirement and/or Use Case to define incrementally deliverable functionality. Ultimately, the goal is the same - to deliver business value regularly in small increments, and sooner rather than later.

**Feature Breakdown Structure (FBS)**

During detailed planning, Agile development favours a feature breakdown structure (FBS) approach instead of the work breakdown structure (WBS) used in waterfall development approaches. Feature breakdown structures are advantageous for a few reasons:

1. They allow communication between the customer and the development team in terms both can understand.
2. They allow the customer to prioritize the team’s work based on business value.
3. They allow tracking of work against the actual business value produced.

It is acceptable to start out with features that are large and then break them out into smaller features over time. This allows the customer to keep from diving in to too much detail until that detail is needed to help facilitate actual design and delivery.

**Building an Initial Feature List**

Before release planning and iteration planning, the team needs to quickly draw up a list of as many potential features for the system as they can. There is typically a single person responsible for collecting features (e.g. a Product Manager, a Customer, a Program Manager, Business Analyst, or some other customer proxy), but feature requests can come from many sources. Users, customers, Sales, Marketing, RFP’s, development team members, management, competitors, and Government regulations can all be sources of features. The team’s central feature list should have some controls to prevent duplicate items, impossible features and overly vague requests. The team should be encouraged, however, to enter new features as they identify them, so that they can be folded into the prioritization and planning process.

An initial feature list can be a rough sketch, a superset, to be used as input for planning the release and first iteration. It represents the current potential of what the system could become, perhaps over several releases. You need not wait until all features are defined before getting started.
delivering software. And you need not adhere senselessly to the original list, original descriptions, or original priorities. One of the main points of Agile development is that this list (like everything else) evolves, iteration by iteration.

Let's pretend that a rough feature list is completed, a release plan and iteration plan are drawn up, and the first iteration is completed, before a couple of critical features are identified. These features simply get folded into the evolving release plan and a later iteration, and get delivered as soon as possible. But meanwhile, time is not wasted. The team starts delivering value as soon as possible, and creates the infrastructure to allow the project to adapt over time to new priorities, information, and business dynamics.

**Feature Headline**

When drawing up a feature list, features are initially described in a short paragraph, typically 2-4 sentences. This description represents a high-level summary of the feature, a placeholder for preliminary understanding and a basis for future communication. It's rather like a headline for an article that will be written later. The goal is to spend just enough time describing the feature to have a reasonable understanding of relative size, complexity, and priority compared to all other features.

A few more details will emerge during iteration planning. But the precise, concrete understanding of the feature is allowed to emerge during the iteration, as customers and developers discuss it enough to implement it, or (in some methodologies) to create automated acceptance tests that specify it deterministically.

**Organizing Features**

Managing a single long feature list can be difficult. It is very helpful to organize features by specifying categories, higher level functional groupings, business value or priority, and risk. Filtering and sorting by these various attributes can help break down a very large feature list into manageable chunks.

The entire list of features should be ranked in a single continuous sequence to provide the project team so that everyone can easily see which features are the most valuable. If a project starts out with 100 features on the list, it is not uncommon for 50 of those features to fall into a "High" priority category. A single sequential ranking of the features helps identify those that are the "highest of the high", and thus should be completed first in order to maximize delivered value.

**Accounting for Risk**

Additional consideration may be taken for the risk associated with certain features. Some features will involve designs, architectures, frameworks, or algorithms that are new to the team, or are otherwise risky. Even if such features do not deliver the highest business value, it often makes sense to bump their priority up enough to tackle them in early iterations. If a high-risk feature is addressed
early in the project, and for some reason proves unworkable, the team still has time to react and work around it. This minimizes the overall risk to the project. It is up to the development team to work closely with the customer to help identify these types of issues, risks, and dependencies. It is ultimately up to the customer to prioritize features, but this critical process should not occur in a vacuum. The best teams work together to both deliver value and reduce risk throughout the life of a project.

Estimating Features

After identifying features, the customer often works with key development stakeholders to define feature estimates. Feature estimates are meant to be preliminary high-level estimates that are used to drive release planning and iteration planning. The stakeholders involved in estimating may include architects, tech leads, developers, testers, writers, and managers. Many organizations have set up processes where groups work together to quickly provide initial estimates. This step can be helpful in initially determining whether the feature should be broken down further.

When initially estimating features, the goal is to quickly converge on a reasonable high-level estimate. Instead of focusing on whether a feature will require exactly 17.5 idea hours (or Gummi Bears, or NUTs, or whatever unit is being used; see below), the goal is to get reasonably close in a fraction of the time. If it takes 2 minutes to agree that the feature will take 2–3 ideal days to implement vs. 30 minutes to establish a precise 17.5 idea hour estimate, the former approach is preferred. To establish a single estimate when opinions in the group vary, teams can either take an average, develop a reasonable approximation, always use the best case scenario, or potentially use a calculation involving best case, worst case, and expected estimate if more complexity is appropriate.

In any case, the discussions about differing estimates will often yield useful knowledge.

This process of defining and estimating features can initially seem difficult, and when teams first implement it, they may require several meetings to get comfortable with a process that works well for them. Over time, it becomes easier to break down features into units of work that can be delivered within a single iteration. Teams get very good at what they practice and Agile development allows teams to practice estimation every release and iteration.

Estimation Units

Estimates by their very nature are inaccurate. And developers have historically had difficulty producing useful estimates of all of the time required to complete a development task. Estimates of actual programming time are often inaccurate (especially if they are not rigorously compared to actual numbers). But non-programming time is even more difficult to nail down. What do you say if someone asks you how long it takes to drive across town? You use a relative measure. "An hour during non rush-hour, in good weather, if there is no construction, otherwise maybe 2 hours," etc.
These external factors are impossible to control and difficult to predict. In addition to developing code, programmers spend time testing, writing documentation, designing, participating in meetings and reviews, doing email, and so on. Compared to programming work, non-programming work is difficult to predict or control. It can vary according to your industry, your organization, the time of year, and any manner of exchanging external pressures on the organization.

Some teams ask programmers to include each non-programming activity in their estimates. But as we’ve said, this is not easy to do. For a given Agile project, long before the team has an accurate measurement of time they spend doing non-programming stuff, they can know the relative work required to get different features done, and can plan accordingly. That’s why it is more typical of Agile teams to focus their estimating on what can be most straightforwardly estimated and measured: actual programming. They focus on how much work each feature and each technical task will take, compared to other features and technical tasks. They allow the amount of time consumed by that non-programming stuff to become clear as actual velocity emerges after a few iterations.

There are two main estimating units Agile teams use to concentrate the focus on programming in this way:

- Work Units
- Ideal Time

**Work Units**

A Work Unit is a relative measure that we hope cannot possibly be confused with actual time. Some such units:

- Points
- Gummi Bears
- Foot-Pounds
- NUTs (Nebulous Units of Time)

These represent the relative amount of work required to implement a feature (or task), compared to other features (or tasks). Only once the team has settled into a consistent velocity, usually over a few iterations, can they begin to map these work units to units of actual time. That is exactly the point of velocity: to describe how much work the team can do per unit of actual time.

Once the team or organization has agreed on an estimating unit, they should agree to make an effort to implement it consistently and stick to its original definition. Especially in the project’s early iterations, everyone should resist the urge to try to map these units to time units with any exact precision.
Ideal Time

Like Work Units, Ideal Time excludes non-programming time. When a team uses Ideal Time for estimating, they are referring explicitly to only the programmer time required to get a feature or task done, compared to other features or tasks. Again, during the first few iterations, estimate history accumulates, a real velocity emerges, andIdeal Time can be mapped to real, elapsed time.

Many teams using Ideal Time have found that their ultimate effort exceeds initial programmer estimates by 1-2x, and that this stabilizes, within an acceptable range, over a few iterations. On a task by task basis the ratio will vary, but over an entire iteration, the ratios that teams develop have proven to remain pretty consistent. For a given team, a known historical ratio of Ideal Time to real time can be especially valuable in planning releases. A team may quickly look at the required functionality and provide a high level estimate of 200 ideal days. If the team's historical ratio of ideal to real is about 2.5, the team may feel fairly confident in submitting an estimate of 500 project days.

In fixed-bid scenarios, this kind of estimate can be reliable.

Relative Estimation

Many Agile teams use the practice of relative estimation for features. Instead of estimating features across a spectrum of unit lengths, they select a few (3-5) relative estimation categories, or buckets, and estimate all features in terms of these categories.

Feature vs. Task Planning

While the emphasis at this initial stage of planning is on speed and on the relative work per feature, at some point features must be broken down to their respective tasks and estimated more precisely.

This happens during release planning and iteration planning. We discuss these in more detail in separate articles. In general, feature estimates and task estimates serve different purposes:

- Feature estimates help drive scheduling across releases and iterations
- Task estimates help drive resource loading within an iteration

Because they serve different purposes, a feature's estimate need not align precisely with the sum of its task estimates. Over a range of features, however, there should be at least a rough correlation between feature estimates and the sum of the task estimates for the features.

FAQ's

How big is a feature?
This can vary to a large extent based on the development work your team is doing. A general rule of
thumb is that a feature should be small enough to be completed within an iteration and big enough to warrant scheduling. You wouldn’t, for example, want to schedule nothing but 1 hour features for a team of 10 working on a 1 month sprint. That’s way too many items to schedule and track. If there are specific feature changes that are that small, it’s often best to group those smaller changes up into one larger item for scheduling purposes. Make each 1 hour of effort a task under that feature.

**Are bug fixes features?**

They can be. Scrum, for example, teaches that all work the team needs to accomplish should be on the backlog list. Common methods for handling bugs include 1) creating a feature item for each bug, 2) creating a feature item called 'fix bugs' within each iteration and detailing the list or types of bugs to be fixed, or 3) handling bugs separately and not tracking time against them. The number and size of bugs your team expects to encounter should be a large factor in determining which method you choose.

**Why Estimate Features?**

Feature estimates help drive the ranking and scheduling that happen in release planning and iteration planning. To know how much work to schedule within a given period, you must have an estimate of how big each piece of work is. Also see Velocity. If two features are of equal business value but one is half the size of the other, the team will be more efficient if it works on the first feature so it should be ranked higher than the second.

**Should we update feature estimates if the task estimates don’t add up?**

No, feature estimates drive scheduling. Task estimates drive resource allocations. While there should be a correlation between the values, they do not need to align precisely.

**Release Planning**

**What is a Release Plan?**

Planning and estimating in the Agile world depend on a single key metric: the development team’s velocity, which describes how much work the team can get done per iteration. (We describe velocity in detail separately.) Given a team’s known velocity for its last project (if it is known), a release plan represents how much scope that team intends to deliver by a given deadline.

Release deadlines are often fixed, imposed externally by such things as tradeshows, accounting exigencies, or contractual obligations. But since the goal is to get working software into the users’ hands as quickly as possible in order to make “course corrections” as soon as possible, every effort is made to keep release cycles as short as possible. Agile release cycles should certainly be kept shorter than a year, and are often as short as 6 months or 3 months. A release is, in turn, made up of iterations. For a given project, iteration length will typically be fixed at a length somewhere between a week and a month. If the release is in 6 months and iterations are going to be 2 weeks each, the release will consist of 13 iterations.
In some environments, software may be delivered to users, or at least a subset of users, incrementally at the end of each iteration or every couple of iterations.

After an initial feature list has been identified, prioritized, and potentially estimated, the team holds a release planning meeting to establish the overall release schedule and determine which features can likely be delivered. The overall release plan in terms of prioritized features is then used to directly feed individual iteration plans.

Some agile methods stress clear separation of responsibilities between programmers and customers. During planning, only the customer is responsible for business decisions and prioritization and only the programmers are responsible for task estimation and development execution. Agile methods also strongly discourage management from capriciously imposing technology choices on the development group, instead giving developers as much latitude as possible to choose the best tools for the system and the project.

**Preliminary Release Planning**

The goal of initial release planning is to estimate roughly which features will be delivered by the release deadline (presuming the deadline is fixed), or to choose a rough delivery date for a given set of features (if scope is fixed). We use this information to decide whether or not the project will produce enough ROI to at least pay for itself, and therefore whether or not we should proceed.

Initial release planning meetings rarely last longer than a day - or two half-days if you just can't stand to stay in a meeting for an entire day. First, the customer presents the prioritized features to be delivered. Ideally, the developers have already devised rough estimates of how much work is required to implement each of those features.

Using the developers' estimates and the customer’s feature priorities, the team lays out a release plan, mapping features very roughly to the first few iterations. Developers may find that fairly low-priority features that pose design or architectural risks, and may therefore ask customers to consider assigning them to earlier iterations anyway, in order to address those potential risks as early on in the project as possible.

It is enormously helpful if the development team's velocity on a previous release is already known. In that case, if scope is fixed, you divide the total estimate for all required features by the team’s velocity to provide the approximate number of iterations required to deliver the functionality, and thus the deadline. If the deadline is fixed (as is typical), then you multiply velocity by the number of iterations to get an initial sense of how many features can be delivered. If the development team's velocity is not known, then they must provide an estimate for it, and the release plan must be understood to be less precise for the first few iterations, until a reliable velocity number can be derived.
Preliminary Release Plan

The initial release plan rarely satisfies all parties - either not enough functionality will be delivered or too much time seems to be necessary to deliver the desired functionality. But in the agile world, the team looks these hard truths in the face, and plans around them. No-one believes in scope miracles that will satisfy everyone. Instead of practicing collective denial, the team uses real metrics and negotiation to make hard choices as close to the start of the project as possible.

Agile thought leaders agree that while it is possible to adjust scope, deadline, resources, and quality for a given project, the only variables that respond well to adjustment are deadline and scope.

Extensive research has shown that larger teams tend to deliver lower quality systems more slowly, while smaller teams tend to deliver higher quality systems faster. It may indeed be necessary to add programmers to a team, but that is likely to slow the team down for at least awhile, not speed it up.

Once we accept these findings, then we accept that in our release planning, we must adjust either the scope or the deadline to produce a release plan that is agreed to be workable by sponsors, customers, developers, managers, and other stakeholders. The fact that the team makes these hard choices up front reduces overall risk for the project. It increases the chances that the team will produce a feature set that returns stakeholder investment more than adequately by the deadline.

Planning the Release Continuously

The initial release plan is understood to be rough. It must be detailed enough only to get us started, by predicting that the project will deliver enough return on investment to more than pay for it. (If the initial release plan predicts ROI that is too low to justify the project, then we can cancel the project before we waste very much money.) In agile projects we plan continuously, and we correct our course as we go. One of the primary mechanisms for course correction is allowing the release plan to evolve in response to all kinds of feedback. It will take at least a couple of iterations for team velocity to settle down. Iterations will sometimes deliver less functionality than was planned for, and sometimes more. Features, architectural choices, design choices, or framework or technology choices might prove to be too risky or simply unworkable. The user interface might require revision.

Staff might be lost or added. Feature priorities might change. All of these factors will help us revise and refine the release plan continuously. When each new iteration plan is published, so should a revised release plan that reflects the new reality.

Start-up & Wrap-up

Many agile teams plan to deliver only a small amount of functionality in the first iteration (often called “Iteration 0”), in order to explicitly allow for working out initial technical and logistical issues, and perhaps also stressing the architecture from end to end. This can be critical for teams that have
little or no agile experience. For a team without a good velocity metric, this might mean that only at
the end of the second or third iteration has the velocity become stable and reliable.

Some teams also schedule up to two iterations at project close for stabilization, system-wide
integration and testing, bug fixing, and user documentation completion. In an ideal agile project this
would not be required, but in the real world it depends on the specific agile practices the team
follows, the organizational structure, the overall complexity of the system, the non-code deliverables
expected of the team, the complexity of system deployment, and similar factors.

FAQ’s

Do I really need to use releases and a release plan?
Some teams can get by without planning at the release level. For example, an ASP may simply
deliver software into production every iteration (i.e., every few weeks), thus every iteration
effectively is a release, so simple planning by iteration may suffice. If, on the other hand, some level
of visibility is required by management at the release level (i.e., progress, status, change from initial
plan, etc.), then release-level planning and management can be invaluable.

How big are releases?
Release typically range between 2 and 12 months. For longer releases, it may make sense to break it
down into several sub-releases.

How many iterations are in a release?
The number of iterations within a release is typically driven by the schedule. If a release is 6 months
long, and iterations are 2 weeks, then 13 iterations should be scheduled for the release.

Who participates in release planning?
For smaller teams, it may make sense for the entire cross-functional team to participate for both
input and accountability purposes. For larger teams and organizations, a subset of the team may be
selected or elected to represent the team.

How long do release planning meetings last?
Release planning meetings typically last between 4 and 8 hours.

How much work is done in preparation for a release planning meeting?
Generally, quite a bit of work has been done prior to a release planning meeting in terms of project
approval, budgeting, vision, team identification, etc. With respect to functionality, the customer has
likely spent time working with development to identify initial features, as well as potentially in
breaking them down and providing initial, high-level estimates.

Does the release plan change during the release?
As more information is uncovered, functionality is delivered, more about the system is understood,
business needs evolve, and priorities change, the overall make-up of the release will almost
definitely change. Although anticipated, the evolution of the release over time will need to be communicated to all appropriate stakeholders.

**Iteration Planning**

**Iteration Planning Meeting**

Iteration lengths typically range between 1 and 6 weeks. The team holds a planning meeting at the beginning of each iteration to break down each of the features scheduled for the iteration into specific technical tasks. Iteration planning meetings generally last from 2-4 hours - any more than that and you may be spending too much time in unnecessary planning; less time than that and you may not be doing enough planning and collaborating.

**Feature Selection**

Many teams set an overall goal for the iteration to help guide the selection of features. At the beginning of the meeting, the highest priority features are typically selected from the release plan. If the iteration does have an overarching goal, then some lower priority features may be selected if they better align with the goal. Prior velocity is critical to enabling the team to schedule a realistic amount of work. For example, if the team previously planned to get 40 days worth of product features, but only successfully delivered 30 days, then 30 days should be considered the current velocity for the next iteration. Past velocity estimates compared to actual numbers are useful at the iteration level, the feature level, and the task level. All of these help the team determine how much they can sign up for in the next iteration. If the iteration is overbooked, then the customer needs to select which features need to be delayed to a future iteration.

During the iteration planning meeting, the customer will discuss features with the team and attempt to answer any questions the team has. The team will break the features down into tasks. On some teams, developers will sign up for tasks and estimate them; on other teams, the development manager will assign tasks. Another popular planning method is to simply estimate the tasks as a team and queue them all up. Team members each select a task to start on and then select the next task off the top of the queue when the current one is completed. This helps to keep the focus more on team goals and less on individual achievement.

**Task Planning**

Tasks typically range in size from 4 hours to 2 days, with most tasks capable of being delivered within a day. Tasks larger than 2 days should generally be broken down into smaller tasks. Occasionally during task planning a feature is determined to be have been woefully underestimated in the original release plan. In this case, the team will need to work with the customer on providing a corrected estimate and determining what feature or features may need to be delayed as a result.
**Iteration Adjustments**

During the iteration, if there is remaining time after all features have been delivered, then the team can request that the customer identify additional feature(s) to add to the iteration. If, on the other hand, it is obvious that not all features can be delivered, then the team works with the customer to determine which features could be delayed or perhaps split in order to deliver the most value by the iteration deadline.

**Warning Signs**

If over a series of iterations, the team continues to push features out into the future, it is a sign that the team should pay closer attention to its prior velocity in order to minimize continuous overbooking and maximize planning accuracy. If the team keeps pushing the same features forward each iteration, it may be a signal that the team is purposely avoiding certain functionality and the root causes should be explored.

**FAQ’s**

*How do we handle dependencies between tasks?*

This question comes up quite a bit. As part of iteration planning, the team should strive to minimize task dependencies as they divide features up. Specific techniques abound in Mike Cohn’s excellent book *User Stories Applied*. Next, the team should strive to collaborate to minimize the effects of unavoidable dependencies. Agile teams typically embrace simple, loosely-coupled, adaptable designs that minimize dependencies. An excellent resource for devising and refining such architectures is Bob Martin’s seminal book *Agile Software Development: Principles, Patterns, and Practices*. Agile teams also use techniques, tools and practices that enable them to work concurrently on interdependent subsystems and modules. Test-driven development, automated test harnesses, and mock objects help teams minimize and cope with task interdependencies.

Continuous, close collaboration can be key here; co-located teams find it easier to work out dependency challenges together throughout the iteration in a just-in-time fashion. Iterations are only so long, reducing the risk that a single lurking dependency will kill the project.

PERT charts and CPM, while potentially valuable in terms of general system understanding, have a very high tendency to crumble under the stress of high-speed, iterative software development. The additional time and effort spent to build a dependency model for a two-week iteration is rarely worth the time. Automated tests and code will give you more accurate, executable feedback at least as quickly.

*How much should a team member sign up for?*
A team member should rarely sign up for more than the total estimate of the tasks they were able to complete in the prior iteration. If tasks are not being signed up for during iteration planning, then more emphasis is placed on making sure the team does not sign up for too much work by comparing to the prior iteration's feature and task velocity.

*How do you plan iterations if team size varies?*
Without the ability to rely on consistent team effort, no project approach, agile or otherwise, provides much insight. With iterative development, though, at least there is typically some history that is built up over time to use as a basis for planning. If you have delivered several iterations with a team of 10 with an average velocity of 20 ideal days or 200 hours per iteration, and you team is cut in half, then a simple calculation should lead you to plan no more than 10 ideal days for the upcoming iteration (at least initially). If key personnel have been removed, or you find you are wrong, you will find out within the next few weeks and be able to quickly adjust for future iterations.

*How do you account for overhead (meetings, email, etc.)?*
Teams generally do not spend much time tracking minor overhead items. Over the course of a few iterations, these interruptions are reflected in increasingly consistent (if unexpected) velocity actuals. Some teams incorporate larger interruptions and disruptions into their iteration plans explicitly, to reduce risk and increase visibility.

*How do you account for bug fixing during iteration planning?*
There are a couple of ways teams deal with bug fixing. One of the simplest is to include bugs as explicit input to iteration planning, prioritizing it, and estimating the tasks involved. Bugs and features are essentially equivalent units of work for planning purposes. Some teams elect to track bugs separately outside of their iteration process. This is somewhat riskier: if the bug-fixing effort varies between iterations, then the team's velocity will vary accordingly, throwing off estimates and plans. But if the bug-fixing effort is held constant, then this method can work reasonably well.

*Why should iterations always be the same length?*
Iterations with the same or very close lengths provide a rhythm that teams rely upon for estimating and planning. Without fixed-length iterations, it can be difficult to achieve and measure steady velocity. The discipline of cutting off production at the close of an iteration focuses everyone's minds, applying pressure to keep designs simple and resist gold-plating or scope creep. The entire organization quickly grows accustomed to a steady hum of input, planning, execution, output, and retrospection. Without this rhythm, the team is less efficient. There will occasionally be good reasons to stretch or compress specific iterations, to fit them around deadlines, major interruptions, or holidays. But these should be the exception, not the rule.

*How do I account for testing and documentation time?*
Testing and documentation updates should be prioritized, estimated, and planned just like any other major activity that require developer's time. They are often created as tasks under specific features, but may also be grouped as their own feature.

*Should feature estimates be revised during iteration planning?*
Feature estimates should only be revised during iteration planning if the original estimate is found to be way off base and the new level of effort will have a significant impact on the team’s ability to complete other work.

**Should task estimates be revised during an iteration?**
The original task estimate should not be revised once the iteration planning has been completed. On the other hand, the estimates for future iterations should be continually revised to reflect an accurate assessment of remaining work.

**Should all teams operate on the same iteration schedule?**
There are advantages to having all teams working on the same iteration schedule. Rolling up iteration status across teams is only meaningful if the teams are on the same schedules. It is not helpful to roll up any numerical status across a team that is just beginning its iteration along with another that is about to finish. The disadvantage to having all teams on the same iteration schedule lies in starting up and wrapping up iterations all at the same time. If common resources (e.g. a customer or management) are shared across projects, they may appreciate a staggered iteration schedule across teams.

**Velocity**

Velocity is an extremely simple, powerful method for accurately measuring the rate at which teams consistently deliver business value. To calculate velocity, simply add up the estimates of the features (user stories, requirements, backlog items, etc.) successfully delivered an iteration.

There are some simple guidelines for estimating initial velocity prior to completing the first iteration (see FAQ’s below), but after that point teams should use proven, historical measures for planning features. Within a short time, velocity typically stabilizes and provides a tremendous basis for improving the accuracy and reliability of both near-term and longer-term project planning. Agile delivery cycles are very small so velocity emerges and can be validated very early in a project and then relied upon to improve project predictability.

**Is it Really That Simple?**

Yes, it is. Do not try not to overcomplicate velocity - it really is a straight forward concept and a great deal of its value lies in its inherent simplicity. Many managers and teams new to Agile methods tend over-analyze the concept of velocity and heap too much complexity around it. After a few months of project experience, most people will experience an "ah ha" moment with velocity, shedding any baggage they’ve associated with it and appreciating its simplicity and intrinsic value.
Velocity Charts

Along with release and iteration burndown charts, velocity has proven to provide tremendous insight/visibility into project progress and status. A velocity chart shows the sum of estimates of the work delivered across all iterations. Typically velocity will stabilize through the life of a project unless the project team make-up varies widely or the length of the iteration changes. As such, velocity can be used for future planning purposes. While typically reliable for a couple iterations out, if you accept that priorities, goals, and teams may change over time and therefore the confidence level of an iteration far in the future, velocity can be used to plan releases much further into the future. A typical velocity chart for a project team might look like the image here.

Team Velocity by Iteration

Initially, teams just need to dive in a select an initial velocity using available guidelines and information. Very quickly (as fast as the next iteration), velocity can be measured and adjusted.

Velocity, along with granular features (e.g., user stories, backlog, requirements, etc.) and high-level and/or relative estimation (in terms of points, ideal days, or even hours), tremendously simplifies and accelerates the whole project planning, estimation, status tracking, and reporting process.

FAQ's

How is velocity calculated?
Velocity is the sum of the estimates of delivered (i.e., accepted) features per iteration.

What unit is used to measure velocity?
Velocity is measured in the same units as feature estimates, whether this is story points, days, ideal days, or hours - all of which are considered acceptable.

How is the first iteration's velocity estimated?
A general guideline is to plan initial velocity at one-third of available time if estimating in ideal programmer time in order to account for meetings, email, design, documentation, rework, collaboration, research, etc. As an example, with six programmers and two-week iterations, a total of 60 programmer-days (6 programmers x10 days) are available. In this situation, a good start would
be to plan 20 ideal days worth of work in the iteration. If using actual time, include enough of a buffer to account for standard project 1) overhead and 2) estimation inaccuracy.

Also, remember that velocity will quickly emerge during the first iteration. If underestimated, velocity in the first iteration will rise as new features are included; and if overestimated, velocity will decrease as features are removed. The second iteration should then use the first iteration as a guideline.

**Do meetings, phone calls, email get included in velocity?**
This depends on whether these items are estimated and included in the iteration plans. They are typically not included - a goal of velocity is relative consistency and predictability across iterations in terms of a team's ability to deliver.

**Should velocity be accumulated across teams or projects?**
Velocity is very much a localized measure. In addition to different team members with different team 'personalities', projects typically possess unique characteristics in terms of estimating techniques, detail process, technology, customer involvement, etc. As a result, this can make organization-wide analysis very inaccurate. If, on the other hand, all of your teams estimate exactly the same, develop exactly the same, test exactly the same, and track exactly the same, then by all means, maybe you are the exception.

**What if velocity fluctuates?**
Velocity will typically fluctuate within a reasonable range, which is perfectly fine. If velocity fluctuates widely for more than one or two iterations, the team may need to re-estimate and/or renegotiate the release plan.

**How long does it take for velocity to stabilize?**
Team velocity will typically stabilize between 3 and 6 iterations.

**How do I estimate future iterations?**
Future iterations use the proven history of the team to determine how much the team can do. Therefore, velocity is the right measure to use for planning future iterations.

**How do I estimate velocity if project teams change size?**
Velocity relies on team consistency in order to be most valuable. If a team changes, use common sense in planning future iterations. If 20% of your team is unavailable for a couple iterations, then reduce planned velocity by 20% or so. If this includes a couple of key players, in particular a customer that may be less available, then reduce the estimate a little more. It will only take the length of the next iteration to understand better what the team can deliver and thus their new velocity.

**Does maximum velocity mean maximum productivity?**
Absolutely not. In an attempt to maximize velocity, a team may in fact achieve the opposite. If asked to maximize velocity, a team may skimp on unit or acceptance testing, reduce customer
collaboration, skip fixing bugs, minimize refactoring, or many other key benefits of the various Agile development practices. While potentially offering short-term improvement (if you can call it that), there will be a negative long-term impact. The goal is not maximized velocity, but rather optimal velocity over time, which takes into account many factors including the quality of the end product.

**How do we measure velocity if our iteration lengths change?**

You don’t, at least not nearly as easily. Velocity’s value comes from its inherent consistency. A fixed iteration length helps drive the reliable rhythm of a project. Without this rhythm, you are constant revising, re-estimating, and reconciling, and the ability to predict out in the future is minimized due to inconsistent results. If, on the other hand, almost everyone is going to be out a week for the holidays or a couple days for company-wide meetings, then by all means simply use common sense and adapt iteration dates or velocity accordingly. Like most Agile practices, these are guidelines, not rules that are meant to prevent common sense.

**Charting Progress**

After a release has been defined, iterations have been scheduled, and features to be created during the iteration have been turned into tasks and tests, the real work begins. The team can actually start working through their tasks and building the intended product. Now, new questions arise: “How are we doing?” and “Are we going to make it?” The nature of iterative development require these questions to be asked at two levels: for each iteration and for each release (or the entire project itself if there is only one release).

**Measuring Iteration Progress**

When considering a team’s progress within an iteration, it is important to view a measure that reasonably reflects that progress throughout the iteration. Completed features are one possible measure to use, but the feature completion rate tends to be slanted (sometimes rather heavily) towards the end of an iteration, so it may not be a very good indicator when you’re only halfway through. Passed acceptance tests provide a less slanted metric, but the effort required to be able to pass a set of tests may vary widely, so it certainly should not be the only metric viewed. Task progress provides a very telling measure of overall iteration progress and has the potential (though it often does not) to remain at a constant rate throughout an iteration. The Burndown Chart shows a trended view of task progress and is the most common measure of iteration progress.

A simple burndown chart is used in Scrum to show a trend of the remaining To Do estimate for all tasks within the sprint (iteration). By adding the completed effort to a burndown chart, the total current estimate within the iteration becomes visible as the total height of the To Do and Done bars. See the chart below.
Simple Iteration Burndown (with Done included)

The classic burndown chart can be packed with even more information by showing historical context and projecting the burndown through the end of the iteration. The chart below shows the current iteration burndown as the thicker red line and the prior iteration's burndown in the thinner red line.

A projection is made (dotted line) for the current burndown based on the team's historical progress.

Iteration Burndown (with History included)

**Measuring Release Progress**

While iterations serve as the lifeblood of the development team, releases represent the lifeblood of the business. As a result, a team's ability to successfully deliver software to the business is of critical importance. Given the tremendous churn inside software projects, accurately communicating progress, status, and change throughout the development cycle remains a challenge. On the other hand, because Agile development uses the incremental delivery of working, tested software features as its primary planning and tracking asset, directly visibility into project status can be very straightforward.

Let’s say the original release plan included 100 features that were originally estimated at 250 ideal days (or story point, days, hours, etc.) of work. Since progress of the release is measured in terms of completed features, at any point in time during the release, insight into the amount of work (i.e., number of features and total feature estimate) completed and remaining is simple to calculate and display. Since Agile development is measured in a binary fashion - a feature is either complete or
not, with little to no credit given for 'partially' complete work - it is a fairly simple process to accurately convey status.

![Release Burndown](image)

While planning and executing at the iteration level is very important, teams should not lose track of where they are in context of the overall release plan. In their desire to make sure iterations are successfully delivered, some teams may lose sight of their progress against the release goals.

Additional scope may continue to creep into the plans as the team marches forward. As a result, each individual iteration appears relatively successful to the team, but the higher level goal of delivering the overall release is sometimes forgotten and changes in the release plan are not communicated to the organization. Consider the graph below, where the team continues to complete features each iteration, but new features are being added to the release just as quickly. If the team does not check its progress relative to the overall release, this situation may not be identified until the release is about over.

![Release Burndown - Scope Creep](image)

In addition to completed and remaining work, other considerations include new features, removed features, and changes in estimates. While Agile development thrives on being able to embrace and benefit from change, being able to analyze, understand, and communicate this change remains critical in any situation involving external sponsors and stakeholders.
Programmer Practices

Keeping the Code Agile

As the iterations start to flow for a newly-Agile team, a new kind of pressure is applied to them. They must deliver working, tested software every two weeks (or three, or four). They find themselves coding more, making more changes to the code, and staying focused on today's deadlines.

Under such production pressures, the team may find they cannot rely on their old programming practices. The programmers and the code itself must become more Agile, which is to say, capable of faster change, sooner. Many who have tried working this way would say that this requires not less discipline than traditional programming, but substantially more.

Agile teams have found in practice that Agile code is extensible, low-defect code with the simplest robust design that will work for the features currently implemented. It is well-factored and well-protected by unit tests. Among the Agile methodologies, Extreme Programming (XP) goes into the most depth concerning how programmers can keep themselves and their code Agile. It is increasingly common to find Agile teams that have adopted non-XP customer practices but have adopted at least some of the XP programmer practices:

1. Test-first programming (or perhaps Test-Driven Development),
2. Rigorous, regular refactoring,
3. Continuous integration,
4. Simple design,
5. Pair programming,
6. Sharing the codebase between all or most programmers,
7. A single coding standard to which all programmers adhere,
8. A common "war-room" style work area.

Such practices provide the team with a kind of Tai Chi flexibility: a new feature, enhancement, or bug can come at the team from any angle, at any time, without destroying the project, the system, or production rates. We've described the theory and techniques behind some of these practices in articles listed in the Programmer Practices menu to the left.

Test-First Programming

Agile teams often find that the closer the unit test coverage of their code is to some optimal number (somewhere between 75% and 85%, many teams find), the more agile their code is. Which is to say, it is easier for them to keep the defects in the code to very low levels, and therefore easier for them to add features, make changes, and still deliver very low-defect code every iteration.

After experimenting with different ways to keep test coverage up at those optimal levels, agile teams hit upon the practice of Test-First programming. Test-First programming involves producing
automated unit tests for production code, before you write that production code. Instead of writing tests afterward (or, more typically, not ever writing those tests), you always begin with a unit test.

For every small chunk of functionality in production code, you first build and run a small (ideally very small), focused test that specifies and validates what the code will do. This test might not even compile, at first, because not all of the classes and methods it requires may exist. Nevertheless, it functions as a kind of executable specification. You then get it to compile with minimal production code, so that you can run it and watch it fail. (Sometimes you expect it to fail, and it passes, which is useful information.) You then produce exactly as much code as will enable that test to pass.

This technique feels odd, at first, to quite a few programmers who try it. It’s a bit like rock climbers inching up a rock wall, placing anchors in the wall as they go. Why go to all this trouble? Surely it slows you down considerably? The answer is that it only makes sense if you end up relying heavily and repeatedly on those unit tests later. Those who practice Test-First regularly claim that those unit tests more than pay back the effort required to write them.

For Test-First work, you will typically use one of the xUnit family of automated unit test frameworks (JUnit for Java, NUnit for C#, etc). These frameworks make it quite straightforward to create, run, organize, and manage large suites of unit tests. (In the Java world, at least, they are increasingly well integrated into the best IDEs.) This is good, because as you work test-first, you accumulate many, many unit tests.

**Benefits of Test-First Work**

Thorough sets of automated units tests serve as a kind of net for detecting bugs. They nail down, precisely and deterministically, the current behaviour of the system. Good Test-First teams find that they get substantially fewer defects throughout the system lifecycle, and spend much less time debugging. Well-written unit tests also serve as excellent design documentation that is always, by definition, in sync with the production code. A somewhat unexpected benefit: many programmers report that “the little green bar” that shows that all tests are running clean becomes addictive. Once you are accustomed to these small, frequent little hits of positive feedback about your code’s health, it’s really hard to give that up. Finally, if your code’s behaviour is nailed down with lots of good unit tests, its much safer for you to refactor the code (as we discuss separately). If a refactoring (or a performance tweak, or any other change) introduces a bug, your tests alert you quickly.

**Test-Driven Development: Taking it Further**

Test-Driven Development (TDD) is a special case of test-first programming that adds the element of continuous design. With TDD, the system design is not constrained by a paper design document.

Instead you allow the process of writing tests and production code to steer the design as you go.
Every few minutes, you refactor to simplify and clarify. If you can easily imagine a clearer, cleaner method, class, or entire object model, you refactor in that direction, protected the entire time by a solid suite of unit tests. The presumption behind TDD is that you cannot really tell what design will serve you best until you have your arms elbow-deep in the code. As you learn about what actually works and what does not, you are in the best possible position to apply those insights, while they are still fresh in your mind. And all of this activity is protected by your suites of automated unit tests.

You might begin with a fair amount of up front design, though it is more typical to start with fairly modest design; some white-board UML sketches often suffice in the Extreme Programming world.

But how much design you start with matters less, with TDD, than how much you allow that design to diverge from its starting point as you go. You might not make sweeping architectural changes, but you might refactor the object model to a large extent, if that seems like the wisest thing to do. Some shops have more political latitude to implement true TDD than others.

**Test-First vs. Debugging**

It's useful to compare the effort spent writing tests up front to time spent debugging. Debugging often involves looking through large amounts of code. Test-First work lets you concentrate on a bite-size chunk, in which fewer things can go wrong. It's difficult for managers to predict how long debugging will actually take. And in one sense, so much debugging effort is wasted. Debugging involves time investment, scaffolding and infrastructure (break points, temporary variable watching, print statements) that are all essentially disposable. Once you find and fix the bug, all of that analysis is essentially lost. And if not lost entirely to you, it is certainly lost to other programmers who maintain or extend that code. With Test-First work, the tests are there for everybody to use, forever.

If a bug reappears somehow, the same test that caught it once can catch it again. If a bug pops up because there is no matching test, you can write a test that captures it from then on. In this way, many Test-First practitioners claim that it is the epitome of working smarter instead of harder.

**Test-First Technique and Tools**

It is not always trivial to write a unit test for every aspect of a system's behaviour. What about GUIs? What about EJBs and other creatures whose lives are managed by container-based frameworks? What about databases and persistence in general? How do you test that an exception gets properly thrown? How do you test for performance levels? How do you measure test coverage, test granularity, and test quality? These questions are being answered by the Test-First community with an ever evolving set of tools and techniques. Tremendous ingenuity continues to pour into making it possible to cover every aspect of a system's behaviour with unit tests. For example, it often makes sense to test-drive a component of a system in isolation from its collaborators and external resources, using fakes and Mock Objects. Without those mocks or fakes, your unit tests might not be able to instantiate the object under test. Or in the case of external resources like network connections, databases, or GUIs, the use of the real thing in a test might slow it down enormously,
while the use of a fake or mock version keeps everything running quickly in memory. And while some aspects of functionality may always require manual testing, the percentage for which that is indisputably true continues to shrink.

**Refactoring**

Refactoring is the process of clarifying and simplifying the design of existing code, without changing its behaviour. Agile teams are maintaining and extending their code a lot from iteration to iteration, and without continuous refactoring, this is hard to do. This is because un-refactored code tends to rot. Rot takes several forms: unhealthy dependencies between classes or packages, bad allocation of class responsibilities, way too many responsibilities per method or class, duplicate code, and many other varieties of confusion and clutter.

Every time we change code without refactoring it, rot worsens and spreads. Code rot frustrates us, costs us time, and unduly shortens the lifespan of useful systems. In an agile context, it can mean the difference between meeting or not meeting an iteration deadline. Refactoring code ruthlessly prevents rot, keeping the code easy to maintain and extend. This extensibility is the reason to refactor and the measure of its success. But note that it is only "safe" to refactor the code this extensively if we have extensive unit test suites of the kind we get if we work Test-First. Without being able to run those tests after each little step in a refactoring, we run the risk of introducing bugs. If you are doing true Test-Driven Development (TDD), in which the design evolves continuously, then you have no choice about regular refactoring, since that’s how you evolve the design.

**Code Hygiene**

A popular metaphor for refactoring is cleaning the kitchen as you cook. In any kitchen in which several complex meals are prepared per day for more than a handful of people, you will typically find that cleaning and reorganizing occur continuously. Someone is responsible for keeping the dishes, the pots, the kitchen itself, the food, the refrigerator all clean and organized from moment to moment. Without this, continuous cooking would soon collapse. In your own household, you can see non-trivial effects from postponing even small amounts of dish refactoring: did you ever try to scrape the muck formed by dried Cocoa Crispies out of a bowl? A missed opportunity for 2 seconds worth of rinsing can become 10 minutes of aggressive scraping.

**Specific "Refactorings"**

Refactorings are the opposite of fiddling endlessly with code; they are precise and finite. Martin Fowler’s definitive book on the subject describes 72 specific "refactorings" by name (e.g., "Extract Method," which extracts a block of code from one method, and creates a new method for it). Each refactoring converts a section of code (a block, a method, a class) from one of 22 well-understood
"smelly" states to a more optimal state. It takes awhile to learn to recognize refactoring opportunities, and to implement refactorings properly.

**Refactoring to Patterns**

Refactoring does not only occur at low code levels. In his recent book, *Refactoring to Patterns*, Joshua Kerievsky skilfully makes the case that refactoring is the technique we should use to introduce Gang of Four design patterns into our code. He argues that patterns are often over-used, and often introduced too early into systems. He follows Fowler's original format of showing and naming specific "refactorings," recipes for getting your code from point A to point B. Kerievsky's refactorings are generally higher level than Fowler's, and often use Fowler's refactorings as building blocks. Kerievsky also introduces the concept of refactoring "toward" a pattern, describing how many design patterns have several different implementations, or depths of implementation.

Sometimes you need more of a pattern than you do at other times, and this book shows you exactly how to get part of the way there, or all of the way there.

**The Flow of Refactoring**

In a Test-First context, refactoring has the same flow as any other code change. You have your automated tests. You begin the refactoring by making the smallest discrete change you can that will compile, run, and function. Wherever possible, you make such changes by adding to the existing code, in parallel with it. You run the tests. You then make the next small discrete change, and run the tests again. When the refactoring is in place and the tests all run clean, you go back and remove the old smelly parallel code. Once the tests run clean after that, you are done.

**Refactoring Automation in IDEs**

Refactoring is much, much easier to do automatically than it is to do by hand. Fortunately, more and more Integrated Development Environments (IDEs) are building in automated refactoring support. For example, one popular IDE for Java is *eclipse*, which includes more auto-refactorings all the time.

Another favourite is *IntelliJ IDEA*, which has historically included even more refactorings. In the .NET world, there are at least two refactoring tool plugins for Visual Studio 2003, and we are told that future versions of Visual Studio will have built-in refactoring support.

To refactor code in eclipse or IDEA, you select the code you want to refactor, pull down the specific refactoring you need from a menu, and the IDE does the rest of the hard work. You are prompted appropriately by dialog boxes for new names for things that need naming, and for similar input. You can then immediately rerun your tests to make sure that the change didn't break anything. If anything was broken, you can easily undo the refactoring and investigate.
Continuous Integration

Traditional software development methods don’t dictate how frequently or regularly you integrate all of the source on a project. Programmers can work separately for hours, days, or even weeks on the same source without realizing how many conflicts (and perhaps bugs) they are generating. Agile teams, because they are producing robust code each iteration, typically find that they are slowed down by the long diff-resolution and debugging sessions that often occur at the end of long integration cycles. The more programmers are sharing the code, the more problematic this is. For these reasons, Agile teams often therefore choose to use Continuous Integration.

Continuous Integration (CI) involves producing a clean build of the system several times per day, usually with a tool like CruiseControl, which uses Ant and various source-control systems. Agile teams typically configure CI to include automated compilation, unit test execution, and source control integration. Sometimes CI also includes automatically running automated acceptance tests such as those developed using FitNesse. In effect, CI means that the build is nearly always clean.

Continuous Integration Technique, Tools, and Policy

There are several specific practices that CI seems to require to work well. On his site, Martin Fowler provides a long, detailed description of what Continuous Integration is and how to make it work.

One popular CI rule states that programmers never leave anything unintegrated at the end of the day. The build should never spend the night in a broken state. This imposes some task planning discipline on programming teams. Furthermore, if the team’s rule is that whoever breaks the build at check-in has to fix it again, there is a natural incentive to check code in frequently during the day.

Benefits of Continuous Integration

When CI works well, it helps the code stay robust enough that customers and other stakeholders can play with the code whenever they like. This speeds the flow of development work overall; as Fowler points out, it has a very different feel to it. It also encourages more feedback between programmers and customers, which helps the team get things right before iteration deadlines. Like refactoring, continuous integration works well if you have an exhaustive suite of automated unit tests that ensure that you are not committing buggy code.

Skilful CI means that integration is never a headache, because your work reflects only slight divergence from the codebase. If the team must regularly deal with small-scale divergences, they never have to deal with truly scary ones. They also get the opportunity to discuss different design approaches on the day that they arise, because integrating that day brings them to the entire team’s attention.
Simple Design

Agile teams are under pressure to deliver working, tested code at the end of each iteration. They are also available to their customers for potentially radical requirements changes at any point in the project. They and their code must be capable of turning on a dime at any moment. So agile teams place enormous value on the extensibility of their code: the extent to which they can easily maintain and extend it. Elsewhere we discuss how important refactoring is to keeping code extensible. The other key component of extensibility is design simplicity. Extensibility seems to be inversely proportional to design complexity.

The Art of What Suffices

In any agile context, simple design means, to paraphrase the poet Wallace Stevens, "the art of what suffices." It means coding for today's specified requirements, and no more. It means doing more with less. But this is not necessarily a natural inclination for us programmers.

We are surrounded by technologies that are supposed to assist us with scalability, performance, security, concurrency, etc. Many mainstream shops seem primarily concerned with our knowledge of specific technologies: database technologies, application server technologies, framework technologies, meta languages and scripting languages, and on and on. Each day these technologies grow in number, complexity, and sometimes sheer bloat. We work in vast oceans of these technologies.

So we can be forgiven if we are more than occasionally tempted to introduce some of this complexity into our designs in an anticipatory way, or if we kid ourselves about the true necessity of these additions. It is sometimes easier to introduce a known gizmo than to experiment with making the design work without it. It is the devil we know. Sometimes we even have such technology choices dictated to us by executives or customers. And sometimes the introduction of a little more complexity, a layer of abstraction in the form of a design pattern, for example, actually does make the code more extensible.

"You Aren't Gonna Need It"

But the truth about design complexity of all kinds is that we often find that the extra abstractions or technologies do not become wings that free us, but instead shackles that bind us. Whatever extra stuff we add, we are indeed clamping to our legs, to lug around thereafter from feature to feature, from iteration to iteration, and from release to release. In Extreme Programming, the saying is "You Aren't Gonna Need It" (YAGNI). There are mountains of painful old lessons behind this maxim.

Agile shops are often willing to trade expertise in specific technologies for expertise in the craft of extensibility, which often boils down to expertise in unit testing, refactoring, and design simplicity. Agile shops tend to wait until a current requirement (a requirement for the current iteration) fairly
hollers for an extra layer of complexity before introducing it. So by mainstream standards, agile
codebases tend to be lean per function point, measured in numbers of classes, or in lines of code.

Many if not most systems eventually require drastic changes. We cannot anticipate them all, so
perhaps we should stop trying to anticipate any of them. If we code for today, and keep our code
and our team agile, we keeps ourselves most capable of turning on a dime as the customer requires.
These are the agile wages of simple design.

**Pair Programming**

Agile teams, committed to frequent, regular, high-quality production, find themselves striving to find
ways to keep short-term and long-term productivity as high as possible. Proponents of pair
programming ("pairing") claim that it boosts long-term productivity by substantially improving the
quality of the code. But it is fair to say that for a number of reasons, pairing is by far the most
controversial and least universally-embraced of the agile programmer practices.

**Pairing Mechanics**

Pairing involves having two programmers working at a single workstation. One programmer "drives,
operating the keyboard, while the other "navigates," watching, learning, asking, talking, and making
suggestions. In theory, the driver focuses on the code at hand: the syntax, semantics, and algorithm.

The navigator focuses less on that, and more on a level of abstraction higher: the test they are trying
to get to pass, the technical task to be delivered next, the time elapsed since all the tests were run,
the time elapsed since the last repository commit, and the quality of the overall design. The theory is
that pairing results in better designs, fewer bugs, and much better spread of knowledge across a
development team, and therefore more functionality per unit time, measured over the long term.

**Spreading Knowledge**

Certainly as a mentoring mechanism, pairing is hard to beat. If pairs switch off regularly (as they
should), pairing spreads several kinds of knowledge throughout the team with great efficiency:
codebase knowledge, design and architectural knowledge, feature and problem domain knowledge,
language knowledge, development platform knowledge, framework and tool knowledge, refactoring
knowledge, and testing knowledge. There is not much debate that pairing spreads these kinds of
knowledge better than traditional code reviews and less formal methods. So what productivity
penalty, if any, do you pay for spreading knowledge so well?

**Pairing and Productivity**
Research results and anecdotal reports seem to show that short-term productivity might decrease modestly (about 15%), but because the code produced is so much better, long-term productivity goes up. And certainly it depends on how you measure productivity, and over what term. In an agile context, productivity is often measured in running, tested features actually delivered per iteration and per release. If a team measures productivity in lines of code per week, they may indeed find that pairing causes this to drop (and if that means fewer lines of code per running, tested feature, that's a good thing!).

**Productivity and Staff Turnover**

Proponents of pairing claim that if you measure productivity across a long enough term to include staff being hired and leaving, pairing starts to show even more value. In many mainstream projects, expertise tends to accumulate in "islands of knowledge." Individual programmers tend to know lots of important things that the other programmers do not know as well. If any of these islands leaves the team, the project may be delayed badly or worse. Part of the theory of pairing is that by spreading many kinds of knowledge so widely within a team, management reduces their exposure to this constant threat of staff turnover. In Extreme Programming, they speak of the Truck Number: the number of team members that would need to be hit by a truck to kill the project. Extreme Programming projects strive to keep the Truck Number as close as possible to the total team size. If someone leaves, there are usually several others to take his or her place. It's not that there is no specialization, but certainly everyone knows more about all of what is going on. If you measure productivity in terms of features delivered over several releases by such a team, it should be higher than if pairing does not occur.

**Pairing Strategies**

In by-the-book Extreme Programming, all production code is written by pairs. Many non-XP agile teams do not use pairing at all. But there is lots of middle ground between no pairing and everyone pairing all the time. Try using pairing when mentoring new hires, for extremely high-risk tasks, at the start of a new project when the design is new, when adopting a new technology, or on a rotating monthly or weekly basis. Programmers who prefer to pair might be allowed to, while those who do not are allowed not to. The decision to use code reviews instead of any pairing at all is popular, but we don't know of any reason not to at least experiment with pairing. There is no reasonable evidence that it hurts a team or a project, and there is increasing evidence that it is a helpful best practice.
Common Codebase

A common codebase is, simply, shared by as many programmers on the team as possible. This is only practical if the team is using test-first programming, refactoring, continuous integration, and a single coding standard. It is also greatly assisted by pair programming.

So what are the benefits of sharing the codebase? Its proponents claim that, as with pairing, sharing the codebase reduces management’s overall vulnerability to staff turnover, and makes it easier to deploy staff. If everybody more or less knows what is going on in all of the codebase, then you have more flexibility when figuring out who will do what in the next iteration. And if someone leaves the team, it is much less of a crisis; other team members can fill the gap pretty straightforwardly.

Other, perhaps more compelling benefits of sharing the codebase include simpler, better overall design, more design consistency, and better mentoring of programmers. All of these things, again, help make the code more extensible and the team more Agile. They help the team deliver more running, tested features per iteration, and deliver them more consistently, with least risk.

Single Code Standard

If programmers all adhere to a single coding standard (including everything from tabs vs. spaces and curly bracket placement to naming conventions for things like classes, methods, and interfaces), everything just works better. It’s easier to maintain and extend code, to refactor it, and to reconcile integration conflicts, if a common standard is applied consistently throughout. The standard itself matters much less than adherence to it. Fortunately, modern IDEs make it trivial to apply many kinds of formatting, even after the fact.

Open Work Area

During the Second World War, when England absolutely had to learn how to break very challenging Nazi cryptographic algorithms, they assembled the best mathematicians and cryptographers they could find at Bletchley Park. And they did not, then, place them all in separate offices (cubicles had not yet been "invented"), to slave away at their own pace in isolation. They placed them together as much as possible, encouraging and requiring them to interact with each other. These guys were, in fact, isolated from the rest of the world. They worked, ate, and lived together. It was from this tumultuous convergence that the brilliant ideas emerged that led to code breaking machines (computers!) that eventually cracked the Enigma. These ideas eventually helped win the war.

Many of us have experienced the team-building, "barn-raising" effects of a small community of folks working on the same problem in a fairly small space. Open-area "war-room" arrangements have repeatedly been shown to facilitate team communication enormously. When a programmer has a technical question, has a feature question, encounters an integration conflict, or just plain gets
stuck, help is immediately available. People can more easily stay on process, stay focused on business value, and celebrate successes. When people can talk to each other at exactly the point when they need to, problems get solved faster, and get solved better. When really big problems arise, the best minds in the group are right there, available to brainstorm and throw big ideas at it together. Those who have given it a good-faith attempt typically report substantial benefits. Team cohesion, camaraderie, ambient trust and respect, listening skills, and other measures of social health all improve.

And this camaraderie translates directly into higher productivity. Teams that like, trust, and respect each other, and that actually enjoy working together, do better work, and do it faster. You would think that we would not even need research to verify something so self-evident.

In Extreme Programming, programmers are expected to all work within earshot of each other in a single room full of workstations, and at least one customer representative is also encouraged to work with them in the room. And while "facilities issues" are notoriously problematic politically and emotionally, lots of teams have made open areas work. There are ways to encourage people to work in such open areas, without asking them to give up all of their private space or "space status."

Are You Agile?

Evaluating Agility

The most agile teams we know of are consistently delivering the highest business value the fastest, and over the longest period. The most agile teams do seem to be the most efficient. But agility is not easy. While built on a set of very simple concepts, agility represents a fundamentally different way of doing business. High levels of agility require a substantial commitment to education and training, new tools and techniques, patient transition, and cultural change. This commitment must flow from management downward, and from the team upward.

Such change is not for everybody. Organizations that don’t promote and value open communication, trust, and teamwork may not provide the most fertile ground for agility to succeed. This does not mean that agile cannot work for such organizations. Agility is a spectrum. Some teams are more agile than others, which are in turn more agile than yet others. It may be well worthwhile for your team to start with modest increases in agility, and to gradually seek a most comfortable and workable spot on the agile spectrum.

Ultimately each organization is responsible to itself for its agility. With any process or methodology implementation, practices will need to be adapted and localized. The key is not the specific practices, but the mindset and manner in which they are applied. In Ron Jeffries’ words, "The practices are not agility. They are a way to find agility."
You might not be agile if.....

1. The 'Send/Receive' and 'Save As' buttons initiate most team communication
2. Your white boards are mostly white
3. "Test-driven" still refers to your car
4. You don't yet know what PHB stands for (hint)
5. You know what CPM stands for, and continue to rely upon it
6. You spend more time trying to manage project dependencies than remove them
7. Someone still believes the Can't Chart (oops, that's Gantt Chart)
8. Developers only develop, testers only test, and managers just manage
9. Simplicity is presumed to be simple, and
10. A Change Control Board meets...ever
General Resources

Must Read

The New Methodology by Martin Fowler

Definitions of Agile Software Development

Wikipedia Definition
c2.com Definition

Websites

<table>
<thead>
<tr>
<th>Agile Alliance</th>
<th><a href="http://www.agilealliance.org">www.agilealliance.org</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>Agile Manifesto</td>
<td><a href="http://www.agilemanifesto.org">www.agilemanifesto.org</a></td>
</tr>
<tr>
<td>Agile Modelling</td>
<td><a href="http://www.agilemodeling.com">www.agilemodeling.com</a></td>
</tr>
<tr>
<td>Crystal</td>
<td>alistair.cockburn.us/crystal/crystal.html</td>
</tr>
<tr>
<td>DSDM</td>
<td><a href="http://www.dsdm.org">www.dsdm.org</a></td>
</tr>
<tr>
<td>Extreme Programming</td>
<td><a href="http://www.xprogramming.com">www.xprogramming.com</a></td>
</tr>
<tr>
<td></td>
<td><a href="http://www.extremeprogramming.org">www.extremeprogramming.org</a></td>
</tr>
<tr>
<td></td>
<td><a href="http://www.xp123.com">www.xp123.com</a></td>
</tr>
<tr>
<td></td>
<td><a href="http://www.industrialxp.org">www.industrialxp.org</a></td>
</tr>
<tr>
<td>Feature-Driven Development (FDD)</td>
<td><a href="http://www.featuredrivendevelopment.com">www.featuredrivendevelopment.com</a></td>
</tr>
<tr>
<td>Lean Development</td>
<td><a href="http://www.poppendieck.com/index.htm">www.poppendieck.com/index.htm</a></td>
</tr>
<tr>
<td>Scrum</td>
<td><a href="http://www.controlchaos.com">www.controlchaos.com</a></td>
</tr>
<tr>
<td></td>
<td><a href="http://www.mountaingoatsoftware.com/scrum">www.mountaingoatsoftware.com/scrum</a></td>
</tr>
<tr>
<td></td>
<td><a href="http://www.scrumalliance.com">www.scrumalliance.com</a></td>
</tr>
</tbody>
</table>

Books on Agile Development

Agile Software Development


Agile Development Management


Crystal


DSDM


Extreme Programming


Feature-Driven Development (FDD)

Lean Development


Scrum
