1. Introduction

In my experience as a manager and consultant, I have observed that not enough commercial software development is conducted in a truly professional manner. This lack leads to extended development time, defect-laden software, and considerable frustration. This situation is often due to a lack of application of basic professional practices.

While there is no universal agreement on what constitutes basic professional practice, there is a core body of practice that accomplished professionals can agree upon. Some of these practices have come to be known as *agile* practices, and others just make good sense. The importance of these core practices and details for using them are not generally taught at the Universities where professional software developers are educated. Universities provide essential underpinnings for software professionals; however their focus is not on preparing a student specifically for commercial software development. This phase of the student's education has been left to an informal apprenticeship.

My intent in writing this book is to provide professionals with the practical body of knowledge required to operate effectively.

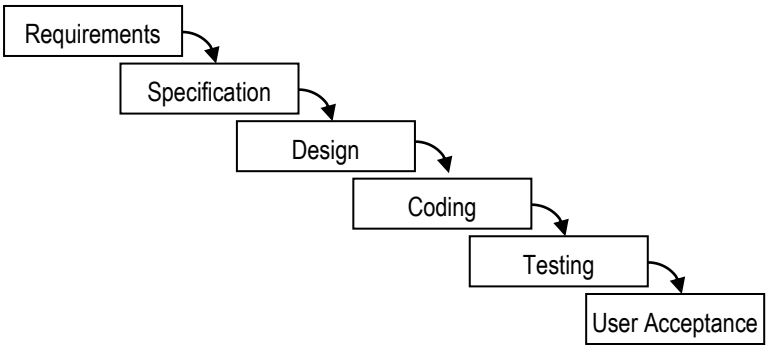
This is not a book specifically about how to design or code or test, but rather the focus is on the *management* of software projects. Building large-scale software systems is a problem of coordination, which in our case lies under the purview of technical software development management. However, I did not intend this book only for managers. For individual contributors as well it provides invaluable guideposts in how to effectively contribute to a well-functioning team.

1.1. Agile Development

Increasingly, professional software development organizations are embracing *agile* development methodologies. These encourage frequent human interaction, iterative development in small units of functionality, a continuously working state for the software proven through automated testing, and embracing late-breaking requirements changes.

These ideas are not new. I used practices that have come to be known as agile as early as the 1980's. The term "agile" came about more recently (in 2001) to contrast these methods to the *waterfall* development methodology (and other process-heavy methods).

The waterfall is a strict step-by-step document-driven method starting with full-system requirements documents, going on to full-system specifications documents, software design documents, finally getting to coding, and then relegating testing to the end. The next step in the waterfall could not start until the previous one was signed off as being fully completed. Complex change processes were required to go back and re-work a step.



Waterfall never worked well. In fact, the very first description of it in the literature by Royce in 1970 in his survey paper "Managing the Development of Large Software Systems" explained why it could not work well because of the inherent feedback problems.

Nevertheless, because it seemed reasonable, it was widely used "by default", despite other more agile approaches suggested in the literature (such as Barry Boehm's "Spiral" or James Martin's "RAD"). While the waterfall was increasingly showing itself to be problematic, at the same time it paradoxically became increasingly entrenched in bureaucratically-imposed approaches to software development. It was policy in large industry and government that contractors must submit work products according to the waterfall model, and the steps were even embedded into standards documents.

With the accessibility of computing increasing in Universities, and the independent software vendor coming into prominence, smaller software development shops accountable only to themselves could choose what methodology they wished. In the absence of a reference framework for software development other than the waterfall, all such organizations ran the risk of being painted with the "cowboy hacker" brush. When things went wrong (as they often do in our business), the "solution" was to impose the dreaded "tried and true" waterfall, which only made matters worse.

To counteract this and to make professionally acceptable many of our best practices, in 2001 a group of like-minded thinkers came together at a ski resort in Utah to publish the "Agile Manifesto" (see http://agilemanifesto.org). A number of agile methods have been put forward, and increasingly many organizations, including large corporations, are embracing them.

Agile methods are not document driven. They are driven by producing small increments of working programs that stakeholders can try for themselves to ensure they are getting what they want. In this way, the development can never go too far off track. The program is always kept in a working state by means of a strong focus on automated testing throughout the development effort. The program is only ever designed to meet the requirements of the next iteration of functionality, but is designed in such a way that is easy-to-modify when new requirements come up.

The most popular agile methods have been Kent Beck's Extreme Programming and Sutherland and Schwaber's Scrum, as popularized by Mike Beedle. Both of these methods were developed during the 1990s.

While agile methods are the clear way forward, when I hear from a software development organization that they "use agile", it can mean almost anything, and sometimes means almost nothing. Agile is emphatically not the lack of waterfall, or avoiding writing documentation, but rather the consistent and repeatable application of its practices embedded in a management framework that acts as a control system over the whole. In this book I talk about the practices and the management controls necessary to ensure they are provably carried out.

This book is not about specific agile methods. There are a number of excellent books that deal with that. All agile methods, all software development in fact, need to be supported by certain core development practices, and need to be controlled by certain core technical management practices. In this book we discuss these core practices essential to *any* agile development method.

1.2. Continuous Release Methods

When I started my career, the software lifecycle was governed by the notion of the big bang release. Every year or so a new major release of the software was shipped, containing many user-visible changes with respect to earlier releases. Consider for example the Microsoft Office products such as Word and Excel in the 2003, 2007, and 2010 big bang releases.

Owing to the methods by which software was delivered to its consumers, a new release had a lot of overhead. For example, shrink wrapped consumer software required system testing across all the various platforms it might install on, burning physical CD's, putting them into boxes, shrink-wrapping them, sending them out through distribution channels, and then having them hit store shelves. Different, but equally lengthy and high overhead processes apply to large enterprise software deliverables, as another example.

These big bang releases provided an ideal planning horizon. By managing what the product would look like in nine-month increments, the business could plan its future. Deciding all the features to go into the next major release was a significant event, as these were make or break decisions for a product line.

While this approach is still relevant in certain circumstances, a lot of commercial software has now moved to more continuous release methods. We need software management methods that can address release and distribution models across the spectrum.

Some of the most aggressive continuous distribution methods are enabled by SaaS (Software-as-a-Service). The idea of SaaS is that the

software vendor not only writes the software but also runs a large centralized instance. Customers access the software remotely using web browsers.

With SaaS, businesses do not need to buy large compute and database servers, house them, power them, maintain them, and back them up. Rather the customer simply signs up for the service on a web site, and begins using the software immediately. They pay on a usage basis model: so many dollars per user per month.

All customers share the centralized common instance, with the software itself maintaining separation between customers using the "multi-tenant" model (an analogy with the idea of many tenants occupying one large apartment building).

With the SaaS delivery method combined with agile practices, it is possible (and advisable) to almost continuously release functional increments to the field. These functional changes need not necessarily be presented to customers immediately, but can be held back using configuration switches and released just to certain customers initially (recall a "Would You Like to Try our New Interface?" type of question you sometimes see when using a web-based application).

Even certain software that installs onto desktop computers is coming close to continuous release methods with downloadable patches available on a frequent basis (my son tells multi-user gaming software is particularly know for this, as is software such as Adobe Acrobat which has frequent new releases). Indeed it is typical for your cable box, smart thermostat, or even your operating system to continuously download and apply patches, without you being aware of it.

With the advent of SaaS and continuous release methodologies, the traditional notion of the big bang software release is disappearing, and along with it the easy and obvious planning horizon of "the next major release". However, something must replace it in order for us to properly manage a software venture.

1.3. The Agile Planning Horizon

A complaint I often hear from business leaders whose development teams embrace agile methods is that they do not know what is going on. They do not know what will be delivered by when. They feel a lack of control of their business's destiny.

At first glance, this seems to be an odd criticism of agile methods that were designed to provide stakeholders with continuous visibility and continuous feedback into what was being built.

But is the CEO trying out the software every two weeks? Even if she were, does using the next iteration every two weeks give her an understanding of what will be delivered by the end of the fiscal year, or by the end of the quarter?

The reason for this lack is that "first generation" agile methods did not particularly concern themselves with this issue. They focus on developing small increments in functionality in a high-quality and predictable manner. The original design point for agile methods is the desire to build software that satisfies the needs of an identifiable user set in the most efficient manner possible. Knowing what was going to be delivered on a nine-month time horizon, for example, was not a goal, and only slows things down to try to "guess".

This haziness over the longer time frame, however, flies in the face of business necessity which insists that we know the timeframe and the main feature set of the software product under development.

In moving away from "big bang releases" in favor of more continuous release methodologies, we risk throwing out the baby (longer-term planning) with the bathwater (big bang releases). While the notion of the big bank release is going by the wayside in certain environments, we must still retain a method for planning our future.

I was faced with this challenge myself as my talented Director of Software Development began moving our SaaS software to more continuous release methods. In order to reconcile the two, we decided that while big releases were gong away, what we formerly called "release planning" could be renamed "agile horizon planning". Indeed many software development organizations have adopted a similar notion, be it Altassian's JIRA Version concept (http://www.altassian.com), or Sutherland's MetaScrums [Sutherland 2005, "Future of Scrum: Parallel Pipelining of Sprints in Complex Projects"].

With horizon planning we identify a fixed planning horizon (our business finds a quarterly planning horizon to be convenient) and we decide before the quarter starts what set of features would be most beneficial, yet still feasible to deliver, within that planning horizon.

The details of when exactly the various features would get released to the field (a software development concern), and how they would be bundled and presented to customers (a product management concern) were unimportant. What was important was managing the set of features that would be shipped within the planning horizon.

At first glance, this seems straightforward, but it is anything but. We need to have a high degree of confidence that everything we were planning to ship would ship with high quality and within the horizon. We are not willing to just take the team's word on it. We need to see the background work that went into the team convincing themselves that it is feasible. This required a methodical approach, and this is the approach we outline in this book.

However, software development is not a sure thing. We are often building things that have never been built before, and managing the uncertainty in the horizon plan is equally critical as having a feasible plan in the first place. Thus as we work through the planning horizon, we insist that new and up-to-date information continuously flow into the plan, and that the plan itself be updated continuously as a result.

Just as software development is uncertain, so is the business climate in which we develop the software. New customers or partners force us to reconsider our priorities mid-stream, as do competitive pressures. We need to ensure our horizon plan is always ready to be reconfigured to meet updated business needs. But also always within the constraint that at all times the updated horizon plan would maintain the feasibility of delivering the features with high quality within the horizon with the available team.

Thus not only should our software development methods be agile, but our larger horizon planning must be equally agile, hence the name "agile horizon planning".

1.4. Dynamic Estimation Equilibrium

Agile horizon planning is therefore a *dynamic* framework where we continuously keep track of our best estimates on how long features will take to code, how long we need to test and stabilize them, how long we need for release preparation, and how much development resource is available to us. I say *best* estimates in the sense that, especially earlier on, we really do not know exactly what we want, exactly how to build it, exactly how long it will take, or even know with certainty the human resources that will be available to us over the planning horizon. Even if we did, the business will impose new requirements midway through the horizon that will change the entire situation.

The best we can hope for is a dynamic equilibrium where at all times the business stakeholders maintain their best guess as to what is required within the agile planning horizon, and the software team maintains their best guess as to the specification and design of the features and the timeframe to implement them. At any given point, if the guesses as to the timeframe given by the development team and the guesses as to the dates required by the business stakeholders do not jibe, then the stakeholders need to jointly agree on tradeoffs and the dynamic balance restored. As the two sides march together, closer and closer to the end of the planning horizon, these "guesses" become better and better and our confidence in our feature set, delivered quality, and final date increases.

Making sure we bring to bear all available expertise and gather and synthesize all pertinent information in real-time is the essence of the dynamic balance we strive for.

1.5. Essential Practices

With this background on agile methods and agile horizon planning, we can now discuss the core essential practices I suggest are required for effective commercial software development. When I am asked to evaluate a software development organization, I use this list as a template against which to compare the organization. These are:

- Source Code Control
- Defect / Feature Tracking
- Reproducible Builds and Deployment
- Automated Regression Testing
- Agile Horizon Planning
- Feature Specifications
- Architectural Control
- Effort Tracking
- Process Control
- Software Development Business Planning

If a software organization can put a checkmark beside each of these practices, it is on solid footing in my opinion.

1.5.1. Source Code Control

Source code control is the basis for solid professional software development. It enables a complete repository for all the ingredients of a shipping software product, it keeps a complete history of all changes, it enables simultaneous development by many developers, and it efficiently enables multiple maintenance and development streams.

A good rule of thumb is that if something does not exist under source code control, it does not exist at all.

1.5.2. Defect / Feature Tracking

To manage a software development effort, one must control what changes go into the source code. A workflow management system capable of keeping track of all the defects discovered against the code, and all the features that are slated to go into the code, and manage the process of getting them done correctly, is an important tool to enable this level of control. The source code control system and workflow management system should be integrated in such a way that every change to the source code *requires* a reference to either a defect or a feature record.

1.5.3. Reproducible Builds and Deployment

Building on solid source code control, reproducible builds and deployments are the next major practice necessary for software development sanity.

Building a software product for testing, setting up a test environment, creating a shippable ISO image for burning onto a CD, or publishing the next iteration of SaaS out to production, should ideally involve no more than issuing a single command.

This guarantees a consistently reproducible build of a product which is necessary for efficient maintenance activities. It also is the necessary prerequisite for automated regression testing.

1.5.4. Automated Regression Testing

Once reproducible builds are in place it is then possible to support an automated regression testing environment. Such an environment ideally tests every aspect of the software in a fully automated fashion. In case of error, it should concisely report on exactly where and how the program under test has failed.

Passing these tests should give confidence that the software has not been broken in any way. The focus of the testing group should be in developing and maintaining more and more complete and torturous automated regression tests (not in manual testing).

This enables software development to move quickly and confidently when deploying production changes.

1.5.5. Agile Horizon Planning

Effective agile horizon planning is the most important of the practices given in this book. It is the means by which resourcing, dates, and feature content are initially established and continuously tracked and updated. Agile horizon planning must at all times preserve the integrity of the *capacity constraint*: that expected remaining work requirement at all times equals expected remaining work capacity.

With good agile horizon planning, stakeholders can be kept up-to-date on what they can expect can be delivered to the field with good quality within a chosen planning horizon, and they can adjust that as they go.

With this practice, quality can be maintained, "elbow room" can be manufactured for continuous improvement initiatives, product management is empowered to make late-breaking decisions about

feature content without upsetting quality, and the uncertainties inherent in software development can be managed.

1.5.6. Feature Specifications

One of the surest ways to upset a software project is to be unclear about what is being built.

A professional software organization must have practices in place to decide whether or not a given feature requires a written specification, a capacity to produce and review written specifications, and a mechanism to ensure that the final product adheres to the specification.

Because agile methods prefer human interaction over written documentation, there is sometimes a misconception that writing a feature specification is not "agile". This is not true at all, individual feature specification documents should be written when they help to clarify matters.

1.5.7. Architectural Control

All software products require an architecture to which coders must adhere. The major architectural structures are the module structure (how the source code is organized), the process structure (how the application/process/thread structure is organized at run-time), the data architecture (how/where external data is stored), and the software design (major elements in the class, procedural, and/or in-memory data structure design).

Every software project should have a coherent and evolving architectural vision and a way of protecting the integrity of that vision in the face of rapid change and multiple (possibly inexperienced) developers.

Because agile methods have a philosophy of not architecting in advance of requirements, there is again sometimes a misconception that exerting control over the architectural is not "agile". This is not true. It is quite proper to have a mechanism to guide the architectural evolution of the software with a steady hand.

1.5.8. Effort Tracking

Making estimates on resource availability, on effort required to implement features, and on ongoing effort required to fix defects is central to agile horizon planning. However, without a means to track the actual amount of effort expended, quantitatively-based decision making is difficult. Therefore, we require some means of tracking time spent against various activities to a highly granular level (*i.e.*, fractions of hours).

1.5.9. Process Control

Well-run software projects follow a certain defined sequence of steps to get new features or groups of features from inception to final ship. To have control of this process means that the steps are written down and known explicitly to all participants, that there are automated systems into which records of passing each step are kept, and therefore that summary reports can be produced showing the extent to which the process is being adhered to.

This is a pre-requisite for effectively inserting quality steps into the process. For example, defining a *specification review* step and recording the pass/fail and action items from this step. Without process control and associated automatically generated reports, skipping these steps is the usual outcome.

1.5.10. Software Development Business Planning

All of the other activities listed (and especially horizon planning) exist in a business context expressed in terms of budgets. It is critical for a software developing organization to understand explicitly its budget, how it will be used, available flexibility within the budget, and that budgets can change according to business conditions.

Having a written business plan enables a software development organization to get a budget from higher management and then to manage to it.

1.6. Effective and Defective Organizations

If one can imagine a defective organization in which none of the above-listed essential practices are in use, the software professional must work to remedy the situation as described below.

1.6.1. Infrastructure

The four infrastructure practices (source code control, defect/feature tracking, reproducible builds, and automated regression testing) form a firm foundation upon which more can be built.

If absent, source code control, must be on the top of the list of improvements. We must decide on a good system, purchase and/or install it on an appropriately sized server with redundant disks and good backups, and migrate all the existing source code into the system. This exercise can be accomplished within a few weeks with minimal disruption, and is utterly non-negotiable. As an alternative, many organizations are moving to SaaS-based source code control, though

some companies take issue with having their valuable source code residing in the cloud.

Even if source code control is in place, but the system used is inadequate, it should be replaced with a better one as a first priority, with the change history imported.

The next thing to look for is a system for keeping track of defects found in the code, and doling them out to the appropriate developers (automatically, to the extent possible). Management must then take the stance that defects above a certain severity level must be fixed first, before a developer may work on a new feature. The defect tracking system should be implemented on a more general purpose workflow management system to enable process refinement and enforcement later on.

The same system can provide a repository for feature requests as a basis for tracking new work. It is also necessary that the source code control system and the defect/feature tracking system be integrated. That is to say, the tool should enforce that all source code check-ins be made against either a specific feature or defect. The selection and rollout of a defect/feature tracking system can usually be accomplished within a month.

We now turn our attention to the manner in which developers or testers make builds of the software and release it into the field.

If the process of building a release or assembling an installer has many manual steps, with files copied all over the place in the process, problems will arise.

Professionals should strive for nothing less than a fully-automated build facility, with all scripts and source (and no intermediate files) pulled from source control. The goal must be a push-button build. Only in this manner can we guarantee quality and consistency.

If the concept of "build" does not apply (as for some simple script-based Web technologies, for example), or if the organization offers software-as-a-service, "build" should be understood to include fully automated deployment of the development system to staging and then to production.

Implementing such a build facility for internal testing builds is the first priority. Implementing it for the installer images and CD creation for shrink-wrap software, or for pushing to production for SaaS, is the second priority. If the build system is in rough shape, the first step can take several weeks of dedicated effort by the top developer most familiar with the system. The second stage can stretch for months if it must take into account all the variants of the software that typically need to be shipped, or all the complexities of deploying SaaS to production.

The next priority must be an automated testing facility. Quality products depend upon the extent of automated testing. No amount of manual testing can make up for the lack of an automated testing facility. Often the most effective way of doing this is architecting the software in such a way that it has an automation API that as closely as possible follows the code paths of GUI-based commands.

Putting in place such an API is a major feature effort (months of development). Building an automated testing facility around it may take several months more. It will then be a constant struggle to build a

library of automated tests and to maintain it in the face of software corrections and enhancements.

This effort is made considerably easier if the API was contemplated at day-one of the architecture, and if all features in the software are coded to be accessible to the API.

While the application regression testing infrastructure is the first priority, full suites of automated unit tests that test internal code are also a help in maintaining quality and tracking down defects quickly to their source. These types of automated unit tests are usually maintained within a framework that allows developers to contribute new test scaffolding and test cases whenever they create a new program module (for example, a new set of classes in C#). These suites test for correct functioning of the programmer's interface to these program modules, and will typically test more completely the code paths within these modules than can the main regression tests, as code that is currently unused within the application is also tested, and obscure error conditions that might never occur in the application regression tests can be tested as well.

With Web-oriented Software-as-a-Service (SaaS), automated testing can often take the form of cloud-based testing services that call a test URL and emulate the behavior of client-side code running in the various browsers.

At this stage, the basic infrastructure for solid software development is in place. However, there is as yet little management control over the development effort. Practices around this are the next priority.

1.6.2. Control

In many software developing organizations, medium to longer term planning is surprisingly lacking. I have found recently that the lack of this is being legitimized under the banner of being agile.

The argument goes that features are being continuously released to the field in short sprints of effort. For any sprint, the small set of features is chosen just before the sprint starts from an unsorted backlog of such features. If a business executive asks what is planned for the next six months, they risk being told that the question itself is not very agile.

Companies cannot do business like this. Independent of feature release to the field frequency, we must always have a longer term planning horizon.

First priority is to set the next planning horizon and then estimate the person-days of developer effort available. Then one may consider a portfolio of features to include during this planning horizon, estimating the effort involved in each one, and making sure the sum total effort required is balanced with the effort available.

As time proceeds, re-estimation should be repeated regularly and appropriate corrective actions taken well before the end of the horizon. In this manner, management will have visibility into larger-scale progress and can adjust direction as they go.

While it is a goal that the software be ready to ship at the end of every small sprint, it is rarely achieved. The agile correction is often the "stabilization sprint", where no new features are worked on for a period of time. As well, with packaged software there are many activities that must occur prior to releasing a new major release out into the field. For

SaaS, the actual act of releasing a significant change into production has considerable preparation associated with it. Finally, it is often wise to include a beta testing period before declaring the new feature set generally available.

All of these types of activities need to be factored into a planning horizon, not just the coding and unit testing. The exact ordering of these activities is less important. However, I should say that it is generally unwise to leave all stabilization, release preparation, and even beta testing of features to the end of a planning horizon. Rather, it is better to intersperse these activities throughout, but to keep the total time spent on them in some proportion to the coding time. If the coding time during a planning horizon increases, so must this other time increase in proportion to it.

Putting such a system into practice is a team effort involving development, management, and product marketing. Initially, the tools used need be no more than a spreadsheet. As an organization grows in sophistication it may build custom tools that update agile horizon plans in real-time on the corporate Intranet. Putting such practices into place with rudimentary tools generally requires a full planning cycle to work out the kinks, and then the following planning cycle will proceed more smoothly.

Once the organization decides what features to work on, the next most important practice is writing feature specifications.

Under the guise of agile methods, some development organization will eschew written documentation. This is not the spirit of agile. Agile values working software over comprehensive documentation, but does not rule out documenting a complex feature.

A feature specification is a document that describes how a new feature will appear to the end-users of the software. Writing them improves quality, reduces costs, and aids release predictability. A typical planning horizon of a major software product may contain hundreds of new features. Not all of them will deserve a full-blown feature specification, though all will deserve at least a meeting with notes recorded. For those that do require a specification document, not having one is a serious mistake.

Institutionalizing the appropriate creation of feature specifications is difficult in a coding-centric environment; however a capacity to write and review specifications must be developed. Existing developers are not always the best people to write specifications. Management must set budget aside to hire or develop people with the required domain knowledge, business analysis abilities, and writing skills.

With a solid infrastructure, horizon planning, and feature specifications in place, the organization now has good control of the quality and predictability of its software. However, there is a longer-term, lurking menace in any significantly-sized software effort (*i.e.*, more than a few hundred thousand lines of code). The lurking menace is architectural deterioration.

While there are exceptions, generally a first release of a successful software product will have a fairly coherent architecture. This is because the architecture of a first release is often tightly held by a creative mastermind taking full ownership. However, as developers are added to a project, as the original architectural leaders move on to other projects, and as defect corrections and new feature additions pile onto

an architecture not originally envisaged to cope with the requirements now placed upon it, the architecture has a tendency to deteriorate.

Written architecture documents become out of date, which then, in a downward spiral, further precipitate the lack of attention paid to these documents. This causes them to become increasingly out of date and hence more and more useless to developers.

To reverse this trend requires management action to ensure there is always one person or one group in charge of the architecture. That person should spearhead the development of tools and techniques to document the architecture and, where possible, automatically extract it and enforce it in the source code. As well, allocating a certain effort budget not for new feature development, but for changes to the source code to improve the architecture (or return it to compliance) is also imperative.

1.6.3. Refinement

Once infrastructure and control is in place, refinements can be made. The first refinement is to put in place a system for measuring the actual effort expended on various development activities: a fine-grained time-tracking system. There is no need to measure when an employee gets into work or leaves; rather, for coders, it is necessary to measure how much time they devote to each individual feature, and how much time they each devote to fixing defects.

These measurements can then be used to refine assumptions that go into forming a horizon plan, such as on average how much time each workday a coder is able to devote to coding new features.

Many managers are fearful of insisting on this much discipline from their coders. However, as long as the coders understand how the data

will be used (*i.e.*, in support of their efforts and not against them) they will welcome it.

Some commercial and open source time tracking tools have appropriate functionality, and are capable of being integrated into an existing environment. Putting in place such a system and integrating may take a few weeks. Management must then insist on its use and monitor the data closely. Within three months of management dedicating attention to it, the habit of tracking time will be firmly in place, requiring only infrequent reminders to staff to keep it up.

The next refinement is to get control of the software development lifecycle process by defining it thoroughly, writing it down, publishing it on an Intranet, and instrumenting it to assess the extent to which it is being followed.

A simple process with the controls discussed previously may comprise a dozen or so steps. These steps should be written down, and the systems (especially the workflow management system used for defect and feature tracking) should be customized to capture features moving from one stage in the process to the next. Management can then develop reports that indicate the flow of features through the various stages. Non-conformance becomes immediately visible to all.

Writing down the process and instrumenting it is a few month's effort, however it cannot be started until the informal process is being used consistently.

The formal measured process will then bring to light situations where the informal process is not being followed. Some of the time, there is a good reason for this, and the process can be refined to account for these situations.

Then, if the organization wishes to add process steps to correct problems (*e.g.*, a specification review, a feature demonstration meeting, a documentation signoff, a code review, and so on) it can do so in a controlled fashion, and can monitor and adjust for compliance.

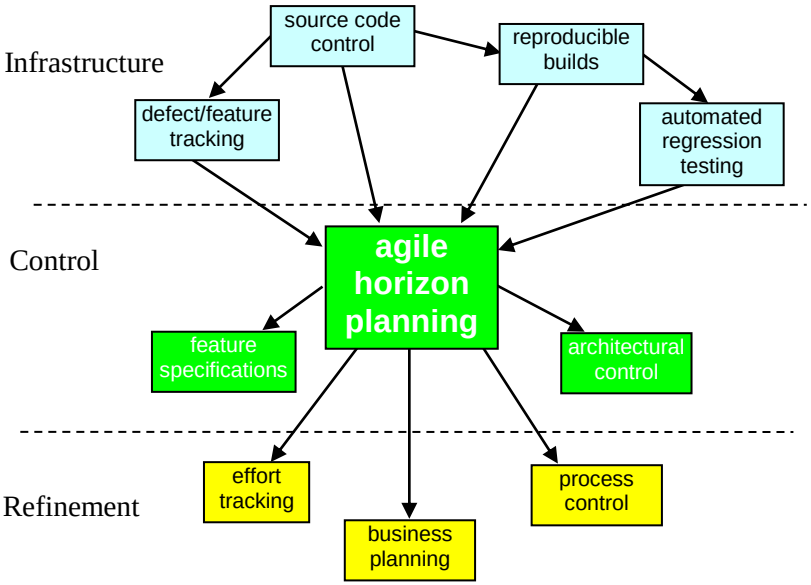
Finally, as an umbrella for accomplishing all of the foregoing activities, a development manager must think like any other business manager and build a business plan for the development group. The purpose of such a plan is to build confidence within the executive team that the situation is well-assessed, the development group is organized with defined areas of responsibilities, development priorities are aligned with corporate priorities, and plans for improvement are well thought-out with concrete timelines and resource requirements translated into budgetary terms. This plan then forms the basis of a negotiation with executive management on an appropriate budget.

While this exercise may seem mundane, without doing it, no progress will likely be made on any of the other points.

1.6.4. Relationships

Of the ten core practices described above, the key practice is agile horizon planning. Agile horizon planning is a goal towards which an organization should strive, and is the key marker that separates well run from poorly run organizations.

The four infrastructural practices, of which source code control is the prime enabler, while having benefit in their own right exist in large part to serve agile horizon planning.



Agile horizon planning, in turn, enables feature control and architectural control, two practices that are difficult to implement in an organization always scrambling to catch up to poorly planned, unrealistic deadlines.

The three remaining practices are considered refinements in the sense they improve upon an already well-run operation.

The outline of the book will follow this rationale. We first proceed with a detailed discussion of agile horizon planning. By covering this material first we lay the groundwork to discuss the importance of the infrastructural practices, the control practices, and the refinements.

1.7. Intended Audience & Scope

The ideas contained in this book are most applicable in the context of a commercial independent software vendor organization that has produced the first release of a product, and is now interested in shipping new features with increased quality and predictability.

Within such an organization, the ideas can be championed either by decision-makers within the software development organization, or at a more grass-roots level by developers on a practice-by-practice level.

The ideas will be of interest to those who wish to learn to manage a reliable software organization, including executives in related areas, individual contributors, product and project managers, and students.

I primarily stress follow-on development as opposed to "green fields" new product development. The reason is that follow-on development makes up the vast majority of effort expended in the software industry, and is thus the type of software development most likely to be encountered by the software professional.

As well, much has been written on green fields development; less on the more economically significant topic of follow-on development. To illustrate the economic significance of follow-on development as opposed to green fields, consider the following hypothetical but reasonable scenario.

> *A software vendor develops a modest new software product over a one-year period using a team of three developers, a tester, and a documenter. Using a nominal loaded cost per employee per year of $100,000, the cost of this hypothetical initial development is approximately $500,000.*

Assume that the product is successful at launch, and is still shipping new releases and maintaining older releases five years later. During this time, the software has been ported to multiple platforms and has been significantly enhanced. To support the effort, the development team has ramped up to a staffing of twenty developers, plus an additional ten in the test and build teams and five documenters. This follow-on effort represents approximately 100 person-years, at a cost of $10,000,000, and is spending $3,500,000 a year for maintenance of the product.

As this scenario illustrates, follow-on costs dominate initial release costs. To make the most impact, effort should be directed at ensuring follow-on activities are efficient. For example, a 10% increase in productivity during initial development would have saved the hypothetical vendor company only $70,000. The same 10% increase in efficiency during follow-on development would have saved the company $1,000,000 to date, and $350,000 or more per year going forwards.

This is not to downplay the importance of the initial release. It sets the stage for everything that follows. If done well, the follow-on releases will proceed smoothly. If done poorly, the entire product lifetime will be a continuous game of "catch-up". Thus one of the most important attributes of initial release development is how well it sets the stage for subsequent development. Understanding the practices essential to subsequent release development is therefore important to carrying out the initial release development in an effective manner.

1.8. Professional Experience

Before continuing on our journey I will digress, if my readers allow, for a quick word concerning professional experience directed at newly minted professional software developers and students of our field.

Whether software developers find themselves thrust into leadership positions, or whether they are one of many on a team, all software professionals have a collective responsibility to improve the state of practice within their organizations. To know how to do this requires education and experience. To qualify as an experienced commercial software developer one requires:

- A solid formal education in the computer sciences.
- To have been involved in several release cycles of a software product, from release inception to ship and maintenance.

Shipping a software product that a large number of customers have paid money to purchase is an important experience. The software developer is no longer in the driver's seat as he or she is for school assignments. The expectations and pressures associated with a commercial release are much higher.

However, simply shipping the software and then moving on is not adequate. Experience comes from having made certain decisions, and later having to correct the problems that arise, sometimes only years later. Only in this fashion, by learning from mistakes in a business setting, can a software professional gather relevant experience.

The recommendations in this book stem from this type of experience. To the inexperienced, many of the recommendations will seem practical and sensible, and the aspiring professional will no doubt

wish to put the ideas into practice. However, without the experience to know that the problems these techniques solve are important ones, it is all too easy to lose this enthusiasm and gradually drift towards the activities that seem to lead most directly to the end goal (*i.e.*, coding and debugging).

The experienced professional will recognize the importance of the problems and how failing to address them early in the project will come back to haunt the project later. They will be loathe, therefore, to start any software development activity without the solutions to these problems in place.

While experienced professionals may have different ways of addressing the various problems, they will generally agree that the problems being solved are important ones that must be addressed.

It is not so much knowing specific prescriptive solutions that make the professional. Rather, it is knowing the problems and how severe they can become if left untreated that is the true mark of the professional.

In this book I will strive to identify these problems and propose solutions that have worked in practice in various settings.

To the inexperienced professional or the student who has yet been troubled by these problems, I can only plead indulgence and hope they will take me at least somewhat on faith.

2. Planning

Of all the activities that ought to take place in the commercial software development organization, the planning and subsequent tracking of software progress is one of the most critical. The Carnegie Mellon Software Engineering Institute identifies planning and project management as the most basic of all software practices [see *The Capability Maturity Model*, Carnegie Mellon University, Software Engineering Institute, 1994]. Indeed, project planning is what separates the chaotic "initial" organizations from the more sound, second-level "repeatable" ones. Any company that has achieved CMM Level 2 is doing well, and has a firm foundation for further improvement.

Despite its importance, good software planning and tracking is a constant struggle. Many times the typical software organization will make no plans at all. Other times it will make plans that seem to have no connection to reality. Sometimes management will make a plan but not update it as events unfold. If development makes a plan and keeps it properly updated, the business stakeholders will often complain that it is difficult, if not impossible, to make necessary changes to it. These planning problems will take their toll on the software company, and can even threaten its continued existence.

The software company that persistently finds itself in these situations can take heart in two facts. First, they are by no means alone. Many software companies seem to go through these chaotic early stages. Second, when the time eventually comes to do so, it is surprisingly easy to correct the situation.

I will describe an overall approach to the management of the software development organization that centers on the notion of a well-defined planning horizon, the rational planning and tracking throughout that time period, and dynamically managing requirement changes. I discuss these practices in the context of a living, fast-paced software organization that has little time or management focus to devote to process enhancement. I will include practical advice on effecting change and making the techniques work.

The practices in this book have been developed and refined within enterprise software vendors, shrink-wrapped software vendors, and SaaS organizations. The ideas have proven to be simple to implement and apply, and yet effective in practical applications.

2.1. Planning Overview

I will start in this introductory chapter to planning by discussing some general considerations regarding plans and planning. We discuss what happens when we do not plan, how planning and tracking necessarily go hand in hand, and why good planning is so elusive.

Chapter 3, "Agile Horizon Planning Overview", overviews a typical planning process for software, and I introduce my particular approach to determining what software the development organization can write in what timeframe with what resources.

The following three chapters go into increasing detail on the horizon planning framework.

Chapter 4, "The Capacity Constraint", discusses the planning framework qualitatively; Chapter 5, "The Quantitative Capacity Constraint" delves into it from a quantitative perspective; and Chapter

6, "The Stochastic Capacity Constraint", looks at how we cope with the uncertainties inherent in planning.

Planning having been dealt with, we proceed with chapters covering the other key practice areas discussed in the previous chapter.

2.2. Why Plan?

In the contemporary software development environment where the small start-up with no processes to speak of can become highly successful, we can legitimately ask, "Why bother planning at all?"

Planning is not always a good thing. For the sole developer, alone in her basement working on the next killer application, there is little reason to plan. In fact, planning will only slow her down and hamper her creativity. We can say the same of small, entrepreneurial start-ups and skunk-works projects within larger organizations that wish to emulate them.

Similarly, if the goal is to as efficiently as possible continuously develop software that assists a targeted group of users in doing their jobs, it is good enough to see a constant gradient of improvement, and knowing what particular features will come out in a year's time is inessential. I believe this scenario is the true design point for agile methods, and why a naïve agile approach can therefore be a danger when planning is required in a commercial software development environment.

However, so long as there is little or no external pressure to produce a given set of functionality by a given date, planning is inessential and just slows you down.

Planning becomes useful to a business when external pressure comes to bear on the software organization.

This external pressure is usually weak in the beginning stages of a company's existence. A company can exist for a considerable time marketing their software only to what Geoffrey Moore refers to in his ground-breaking high-tech marketing book, *Crossing the Chasm*, as the innovator and early adopter market segments. These market segments wish only new functionality delivered as soon as possible, are tolerant of defects in the software, are resigned to poor or nonexistent documentation, and will suffer less than adequate support. In general, these early customers have great patience with the fledgling software company. Once the company "crosses the chasm" and wishes to sell to the majority market segments, the software organization's planning practices must change dramatically.

These majority customers will want to plan their purchase and schedule their acceptance testing and rollout. They expect high quality documentation and customer service, are intolerant of defects, and expect that if they report a defect that the vendor will promptly fix it in a maintenance release that contains no extraneous feature enhancements. Under these conditions, planning is one factor that separates successful software vendors from those that disappear into Moore's chasm.

When targeting the majority markets, the successful software company must set expectations and then deliver to them. If an important new customer requests specific functionality, there must be a mechanism in place to determine if software development can deliver within the timeframe, and then monitor to make sure that it happens. Good documentation, customer training, marketing, customer service,

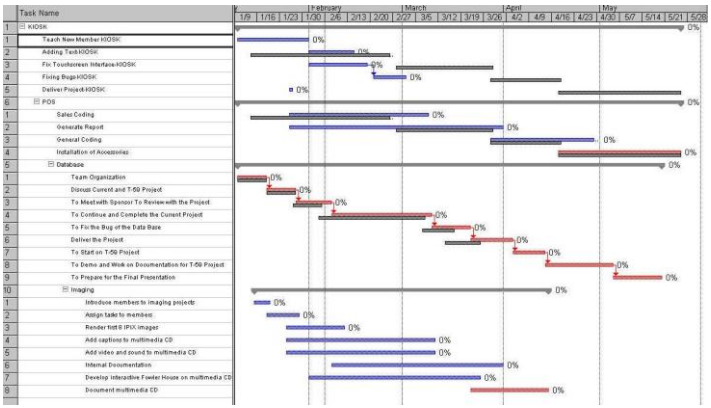
and sales are dependent upon the organization having a unified view of the expected functionality and release date. If there are to be any changes, it is essential that the software development department provide warning well in advance to allow the company to take mitigating actions as early as possible.

Whether the pressures come from new customers, venture capitalists, or industry analysts, the common thread is external expectations. In all cases the software organization must make commitments, manage expectations, and deliver on those expectations. If it does not, the company will cease being successful.

In these circumstances, proper planning is essential. Why then are there so many software companies that fail to do a good job of it?

2.3. Gantt Charts Considered Harmful

Too many software developers and project managers look at plans in one certain way: as a Gantt chart.



Practically all project-planning software we see revolves around this concept. To many classically trained project managers, it is inconceivable to start a software project without first listing out all the detailed tasks that need to be performed, assigning personnel to these tasks, taking into account the inter-dependencies between tasks, and only then assessing the feasibility of getting a software release out the door by a planned date. Typically, these plans wind up becoming large and unwieldy, if done non-trivially at all.

The problem with the large Gantt chart plan is that it is too clumsy a tool for the sort of agile horizon planning that needs to go on in the typical software vendor organization. Because the plan is oriented towards documenting a set of activities, it is awkward for performing fast-paced "what-if" tradeoff analysis between features and dates.

More particularly, effective use of a Gantt chart implies a level of knowledge of how events will unfold that has no justification at the time initial planning is taking place.

A complex Gantt chart is overkill at the start of a new release initiative. Fortunately, it is not necessary to use one.

Software organizations often lose sight of their most important objectives when planning. These are to know what they are building, by when, and using how many people.

- **What are we building?**
- **By when will it be ready?**
- **How many people will it take?**

Coming up with a plan that answers these questions (and no more) must be the focus in agile horizon planning.

Later, when the vendor has clear answers to these questions, it can develop an implementation plan using Gantt charts. This plan can be used to divide work amongst people, assign people to tasks, and sequence those tasks. In our framework, implementation plans flow from agile horizon plans. The framework proves itself when these more detailed plans do not contradict the agile horizon plans.

That having been said, it is my opinion that agile horizon planning combined with agile development methods renders any implementation plan unnecessary.

2.4. Of Mice and Men

Developing an initial plan that answers the important agile horizon planning questions is less than half the effort. One of the biggest culprits is not neglecting to have a plan in the first place, but rather neglecting to update it as it unfolds.

There is one certainty in any planning effort: the plan will change. Some would have us believe that this is a flaw with the planners. If only they had planned properly in the first place, they would not have to keep changing the plan.

However, it seems to be a universal truth that all sorts of plans have a distressing habit of going astray. The Scottish poet Robbie Burns expressed it well in 1785:

> *But, Mousie, thou art not alone*
> *In proving foresight may be vain:*
> *The best laid schemes o' mice an' men*
> *Gang aft a-gley,*
> *An' lea'e us naught but grief an' pain*
> *For promis'd joy.*

[To A Mouse, on turning her up in her nest, with the plough, November, 1785, Robert Burns, from *Poems, Chiefly in the Scottish Dialect]*

When formulating plans it is important to understand that these plans represent only a current understanding of one possible way reality may unfold. Unless we accept, indeed embrace, the uncertainties in a plan, we will be disappointed by our planning success. Embracing uncertainty implies that as time passes, as better information becomes available, as the business situation changes, as we uncover flaws in the initial planning, we must update the plan. It most particularly does not mean sticking to our plan through thick and thin, and hoping to make up for lost time somewhere, somehow.

Plans change for a variety of reasons. The reasons divide into *internal* and *external* causes. Internal causes are those resulting from changes to the estimates we made during the initial planning. External causes are those resulting from late-breaking changes in the requirements. Let us consider internal causes first.

In the computer industry, even professionals are notoriously bad at being able to predict how long it will take to build cutting-edge software. Therefore, any initial estimates upon which we base a plan must contain a significant margin of error. As time goes by and as development proceeds, it would be rather odd if re-estimates of how much longer a feature will take to implement were consistent with the initial estimate. Every time we uncover an estimation error it is a change that upsets the plan. Unless we update the plan it descends into meaninglessness.

A particular cause for distress is that once "padding" is removed (as I advocate here in favor of honest estimates and rational reaction to estimation changes by management) estimation inaccuracies are more often than not skewed towards overly optimistic estimates. This is because developers will rarely imagine a piece of work that needs to be done which is not required. However, in coming up with estimates it is likely that they will forget some work that would need to be done. Putting a bit of padding into an estimate to account for this is advisable; however, the tendency even in that case will be to estimate low.

Other internal causes that require plan updates are developers quitting, personnel being re-assigned away from the project, unexpected changes to vacation plans, and changes in the number of hours per day a developer can devote to adding new features into the release.

External causes can affect plans as well. Suppose a major new prospect has arrived on the scene and will only purchase the vendor’s software if they implement a certain new feature. For the kind of revenue they are

expecting from that customer, chances are they will change the plan. This is an external cause.

Other external causes for plan changes are competitor moves, collaboration opportunities, and changes in a regulatory environment.

Whatever the reasons for change, we can be certain that we will need to update the plan regularly throughout the development effort. It is in this situation that the large Gantt chart shows its weakness. With so unwieldy a document, updating it regularly is impractical.

Yet we cannot sufficiently stress that making an initial plan without updating it is worse than useless. It actually wastes time that we could have better spent on productive tasks. Only if we keep a plan up-to-date in the face of reality does it continue to have meaning.

2.5. The Difficult Question

A software development organization needs to make a good initial plan and needs to keep it up-to-date. Unfortunately, the software industry, collectively speaking, is not good at planning. While there are three easy questions to answer when planning, there is at least one difficult question.

Recall the three planning questions the software vendor needs to have answered.

- **What are we building?**
- **By when will it be ready?**
- **How many people will it take?**

The first question, "what are we building?" can be hard to answer for the first release of a product, but is usually much simpler for follow-on work. The product managers will have lists of features that the customers have requested or that the sales force says are necessary to close future sales. Choosing a set of these features for the next planning horizon and refining their specification is straightforward.

The second question, "by when should it be done?" is easy enough. The software vendor must pick a reasonable date that is not so far out that the customers think the vendor has given up on the product.

That leaves the third question: "how many developers?" For most ongoing software ventures, the development organization knows how many developers they have to work with. They can think about hiring, but unless they can get developers who are already familiar with the code base, the newly hired developers will not contribute more than what they consume from the other team members for on-the-job education. Therefore, resourcing, at least for the next planning cycle, is usually very constrained.

The easy questions are therefore "what", "when", and "how many". There is one more thing to consider. Can the software development organization build "what" with "how many" by "when"? This is the difficult question; nonetheless, the software company will need to address it. If they do not, they are likely to run into trouble as the following tale illustrates.

2.6. A Software Vendor Fable

While the development organization often raises the essential question of whether or not resource balance exists for a proposed plan, it is surprising how often this question remains unanswered.

One common approach is "our developers are the best; of course they can pull it off." If this flattery does not convince, the next line of defense is often "it needs to get done; the business is riding on it." Management asks development what they need to make it happen. Do they need more money to hire consultants? Do they need extra pay for working weekends?

The final line of defense is often: "the release has already been promised and we can't go back on our promises now." To this, unfortunately, there is no rebuttal.

We are now at the point where practically nobody in the organization thinks the release can happen on time, but nonetheless everybody is working away on it. Edward Yourdon describes this situation in his book *Death March [Prentice-Hall, 1997]*. Development knows it cannot happen on time. Product marketing knows it will not come in on time. The CEO hears "we're going to really have to push ourselves to get this one done, but we know how important it is to the company."

As time marches on it becomes increasingly clear that the features cannot all be done on time. Balancing this is the fact that it is also getting increasingly painful to have to admit this. Trapped in a difficult situation the human psyche prevails: hopeless optimism sets in. It will be their finest hour.

When the planned date passes, development management characterizes it as a short delay, a couple of weeks at the outside to correct the most important problems. This will typically happen more than once. Eventually, things come to a head. Development admits that it now appears as though it will take another couple of months.

Development, however, is blameless. Nobody agreed to the ridiculous plan in the first place. Development said it was next to impossible but that they would work as hard and as smart as they could. The obstacles were too much to overcome.

Product Marketing as well is blameless. They made the plan, but Development said it was doable. Nobody heard any complaints for the last three months either. They were under the impression that everything was going fine.

Unfortunately, this little fable happens far too often in the software industry. With a little knowledge, foresight, common sense, and commitment to change, such situations are entirely avoidable.

Read on.

3. Agile Horizon Planning Overview

For a software organization to be successful, it needs to accurately plan and track the development of its software. To enable this, the software organization must make two commitments. The first is that they subscribe to the notion of a longer term horizon plan. The second is that they follow a repeatable development process. For the purposes of introducing agile horizon planning it is not necessary that we take into account the low-level details of the process. A sketch will do.

In this chapter, we give an overview of how agile horizon planning operates in the context of a specific type of technical release cycle. In subsequent chapters, we go into the details in greater depth, discussing tradeoffs, business issues, and advice on how to make the process work in a software organization.

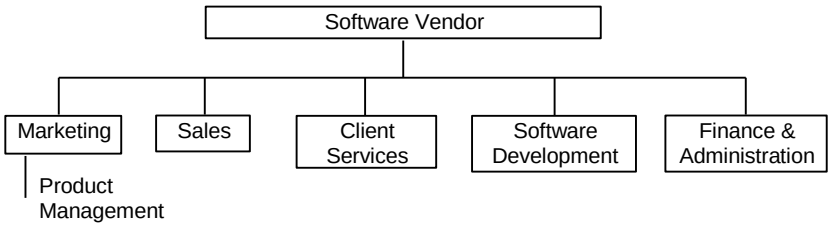
We will be discussing agile horizon planning in the context of a software vendor organization wishing to better control follow-on releases of their software product. The median of such product ventures will involve around three dozen people and a code base of a million lines of code or so. However, the ideas are applicable on larger and smaller scales, as well as in software development contexts other than a software vendor organization.

To set the stage, we start by describing the business environment of a typical software vendor organization.

3.1. Software Vendors

A software vendor is a business that makes its money by providing software and related services such as help desks, training, product-related consulting, user groups, and so on.

The typical medium-sized software vendor organizes itself as follows.



The **marketing** group is responsible for identifying market segments and determining what software features the market would be willing to pay for. They are also responsible for communicating the existence and benefits of the software to these target markets, and for identifying new channels to market. Ultimately, they are responsible for the profitability of products.

The **product management** group within marketing will manage and coordinate the horizon plans for the various products, and coordinate the launch of new products, or new releases of existing products.

The **sales** group, often organized by sales region, identifies individual prospects, negotiates terms with them, and consummates sales. This group is responsible for the company's revenue targets.

The **client services** group is responsible for helping customers get up and running with the company's software, and dealing with any ongoing issues they may have. As such, they provide pre-sales support,

training, help desk services, and consulting services. They are ultimately responsible for customer satisfaction.

The **software development** group is responsible for delivering high quality product with a promised set of features by a promised date. This group will have limited direct contact with customers. Client services will deal with customer issues, and product management will deal with customer requests for product enhancements.

The **finance and administration** group ensures that the company has adequate funding, provides oversight over spending by establishing budgets, and takes care of the day-to-day operations.

Human resources and **internal IT** are two departments that tend to report wherever it makes most sense, given the experience of the various executives.

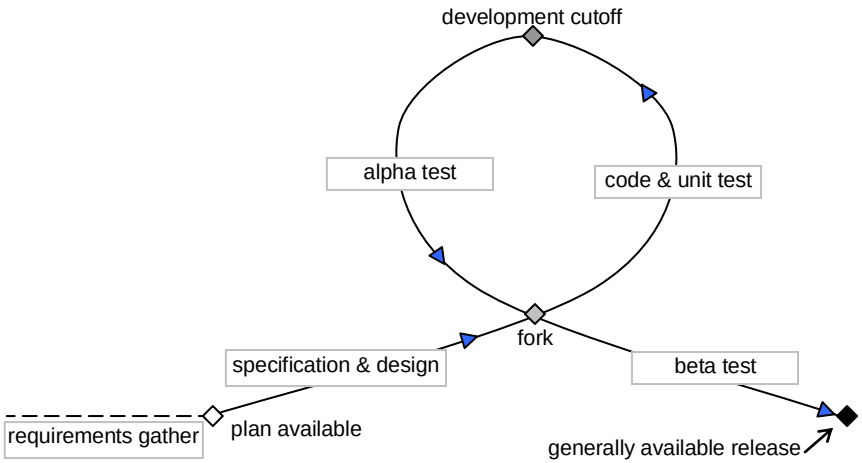
The lifeblood of the software company is the flow of new features in their software products going out the door. If the rhythm of these new features falters, it jeopardizes the health of the company.

Customers depend upon the new features, plan ahead of time to receive them, and make commitments based on promised deliveries. To satisfy its customers, the software company must successfully combine the efforts of the entire organization to meet the challenge of shipping on time and with high quality. Therefore managing the horizon plan is central to the management of the software company.

Managing the horizon plan means deciding what new features are to appear by what date, and adjusting this plan as the situation changes. The company measures success by whether it can consistently make and meet its commitments. Agile horizon planning, as described in what follows, is the means by which this can occur.

3.2. The Traditional Software Product Lifecycle

Before discussing the increasingly important continuous release methods advised for SaaS-type software, I will first describe a more traditional release method, suitable for most other types of delivery where the cost of releasing a new feature set to the field is relatively high.



The preliminary stage, ***requirements gather***, consists of gathering potential requirements for the release. The marketing product management group will coordinate this effort. Once they have brought together and prioritized this wish list, the initial agile horizon planning can begin. It consists of working with software development and key decision makers to decide on the dates for the release and what features from the wish list will make it *in-plan*.

Once the company has settled on an initial agile horizon plan, detailed ***specification and design*** work begins on certain in-plan features. This can involve both prototyping and written work. As analysts, coders, and architects refine the features into specifications and designs, they will update their initial sizing estimates, necessitating re-planning. It is not necessary that all such work be done before coding can begin. This phase is used to get a head start.

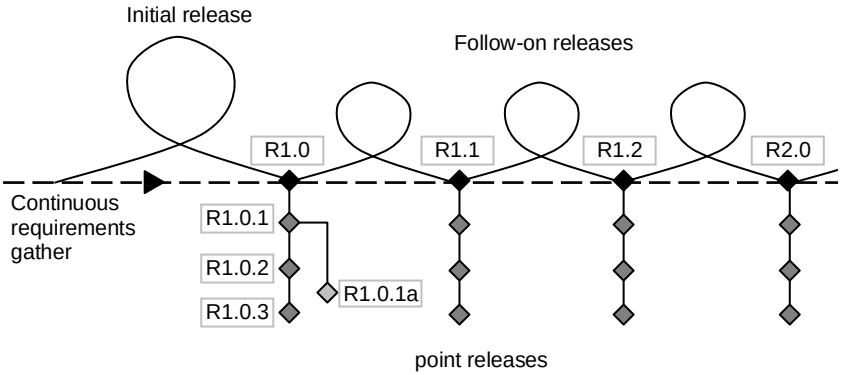
When design and specification has produced enough output, development is ready to begin ***coding and unit testing***. For traditional release lifecycles this point is called "fork" because the source code is logically forked into the release currently being supported in the field, and the new release that is under development. Throughout this phase, software development and product management update the horizon plan as they uncover new information, and shift features and dates once the plan is no longer within the comfort zone. Coding and unit testing continues until the team arrives at the milestone known as "development cutoff". At this point, no developers can identify any remaining code they need to add to make the release feature-complete.

After development cutoff, ***alpha testing*** begins. The testing staff uncovers defects and the developers fix them. A common variation is to have the alpha testing phase spread out and interspersed into coding and unit testing, in order to minimize the total time to do both.

When the new software is sufficiently stable, the organization will release a ***beta testing*** version both internally and to select customers so that they can get an early look at the software, try it out in the field, and suggest improvements. At the end of the beta testing period, the company will make the new release generally available.

Throughout the testing phases, management keeps a close eye on defect arrival rates to determine if the end-dates remain feasible.

The software vendor will iterate such a release cycle many times during the lifetime of a commercially successful software product.



After the initial release, typical lifetimes would exceed five years and have at least ten or more distinct follow-on releases, each of which adds significant functionality to the product. Follow-on releases would be spaced anywhere from six months to a year or more apart, closer in certain situations.

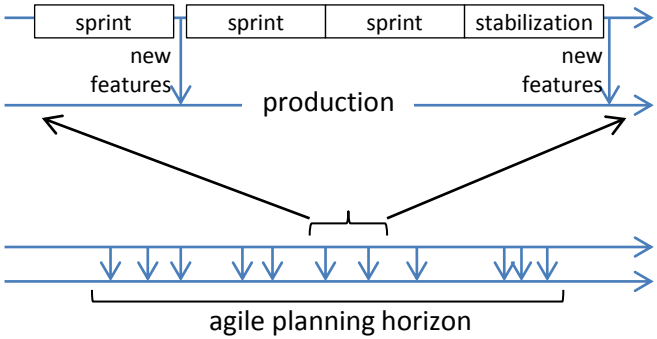
Each feature release will require ongoing maintenance, meaning that customers who take a release will have made available to them periodic point releases that fix any problems they may encounter. These *point* releases are pure corrective maintenance, and contain no new features.

Requirements gather is outside of the release cycle, and proceeds at a constant pace throughout the lifetime of the software. Product management will consider those features that do not make it into the current release for the ones following.

3.3. SaaS Lifecycle

The typical lifecycle for Software-as-a-Service differs from the traditional lifecycle described in the previous section.

The SaaS lifecycle is more linear and continuous. Agile teams will pick up a set of features to work on and deliver a completed set of functionality within a two or three week sprint. Specification and design, coding, and alpha testing are generally handled on a feature by feature basis and attain 90% closure by the end of a sprint. Sometimes several such sprints may be chained together, followed by a stabilization sprint to ensure the software is ready to be put into production.



This is repeated continuously throughout the life of the software. This may also be mixed with the more traditional lifecycle when a very large change needs to be made.

For the sake of longer term planning, all the features that the teams will work on during a given time horizon will be planned out such that elapsed time, effort required, and effort available will be correctly balanced.

3.4. The Agile Horizon Plan

The central document used in managing the software is the *agile horizon plan*. Here we show a simplified example agile horizon plan below. A full version suitable for the traditional software lifecycle is shown in Appendix A on page 345).

The horizon plan takes the form of a balance sheet. On one side are the available development resources. On the other side is the amount of work required to code all the features. For the plan to be valid, the available resources must balance the required resources.

Simple Agile Horizon Plan

Planning Horizon: Jul.1—Sep.30 (64 workdays)

Coding Ratio: 3:1 (48 coding days)

Capacity:		*days available*
	Fred	**25** effective-coding-days
	Lorna	**33** effective-coding-days
	…	…
	Bill	**21** effective-coding-days
	total	***317 effective-coding days***

Requirement:		*days required*
	AR report	**14** effective-coding-days
	Dialog re-design	**22** effective coding days
	…	…
	Thread support	**12** effective-coding-days
	total	***317 effective-coding-days***

Status:		
	Capacity:	**317** effective-coding-days
	Requirement:	**317** effective-coding-days
	Delta:	**0** effective-coding-days

Planning Horizon

The first section of the plan shows the duration of the planning horizon and the number of workdays available within that horizon.

Coding Ratio

The next section shows within that planning horizon the typical proportion of days available for coding and unit testing versus other activities that need to be carried out (such as specification and design work by coders, stabilization, and release preparation). A development shop will arrive at this ratio through historical measurement.

Capacity

Next comes the computation of the capacity to do coding work expressed in *effective-coding-days*. We explain this quantitatively in Chapter 4. For now, the units can be simply understood as the number of solid, 8-hour coding days (devoid of all other distractions) available for coding.

To compute this measure we start by listing all the coders who will contribute to the plan. For each, we count workdays available for coding and apply a factor to convert to effective days. We then sum to come up with a total that represents the capacity available to put features into the release.

Requirement

Balancing the capacity to do work is the requirement for work to be done. For consistency with the capacity measure, we also express this in units of effective coding-days.

To compute the requirement we list all the features that we wish to include in the release, attach a sizing to each, and sum them. The individual feature sizings are a combined estimate of the intrinsic size of the work item, an estimate as to which developers will work on the feature, and an estimate of how productive they will be with the hours they dedicate to the feature. For the purposes of the horizon plan, we combine these uncertainties into an aggregate feature sizing given in effective-coding-days: the total number of uninterrupted hours required to code and unit test the feature divided by a nominal eight-hour day.

There is complexity here, such as what constitutes a feature, how to size them, how to deal with uncertainties in the estimates, and so on. We shall discuss all of these issues in detail later on. For now, it is sufficient to note that the end-result is a list of features understandable equally by software development and by the business stakeholders; not a list of tasks that software development alone can understand.

Status

At the bottom of the plan we bring together the capacity and the requirement, subtract them, and give a *delta*, the number of effective-coding-days by which we expect to come in ahead of (positive numbers) or behind (negative numbers) our target date.

Negative delta indicates a growing need to take action to re-balance the plan, either by extending dates, dropping features, scaling back the scope of certain features, or adding effective resource. Positive delta indicates that there is room in the plan to do more.

We do not view positive delta as a contingency. We deal with contingency explicitly in the stochastics of the full plan, which we will discuss in Chapter 6.

For more traditional software lifecycles, we focus our planning efforts towards development cutoff. After this date, the die is cast. As we pass this milestone, our proactive control of the plan is reduced to either delaying the generally available release or shipping a poor quality product.

To ship a good quality release on time, it is essential that the software company hit their planned development cutoff date and thereby maintain their planned course of testing and debugging. It is not reasonable to expect that if coding has taken *longer* than expected, then testing will take *shorter* than expected. Yet, many organizations will default to this behavior, holding end-dates firm despite slips in development cutoff, no doubt hoping for better-than-usual luck during testing. Long experience has shown that if the development cutoff date slips, it will be necessary to extend the end-dates both by the length of the slip plus by an extra proportionate amount of time for testing. Therefore, it is important to ensure that development cutoff does not slip, and manage the plan to that date accordingly.

Though the sample horizon plan that we show here is simpler than the full horizon plan shown in Appendix A, it captures the essentials of agile horizon planning, which is a balancing of capacity and requirement. While we must consider many details and make a study of how to make agile horizon planning work in practice, we must not lose track of the simplicity of the situation.

Planning in greater detail is a temptation we must avoid, as this extra detail is not justified by the precision of the capacity and requirement estimates.

3.5. Implementation Planning

When planning a release we are dealing at a level of abstraction above that required by the detailed plans used to get the job done. At a certain stage in the development cycle, we will need to plan *all* activities in detail. We must divide jobs into tasks, assign these tasks to individuals, sequence them appropriately, and track them. All of this requires detailed planning, or *implementation planning*.

When we are doing agile horizon planning, this level of detail is counter-productive. What we are seeking is a planning method that gives us a minimum of complexity while losing the least planning accuracy. In this way, the horizon plan becomes easier to cope with: it is easier to put together, easier for developers and business stakeholders to understand, and easier to keep up-to-date on a regular basis.

We have designed the agile horizon plan to be amenable to rapid "what if?" analysis: "what if we added this feature?", "what if we took away that feature?", "what if we moved the date out?" We have also designed it to be easy to keep up-to-date on a weekly basis so that at all times we have an understandable statement of the status. If the status is "behind schedule", the what-if capabilities of the agile horizon plan provide us with a mechanism to consider plan adjustments.

The biggest fault with planning is not the shortcomings of a particular planning methodology, but rather either not planning at all, or making an initial plan but then not tracking and adjusting it. A simplified agile horizon planning method addresses these issues.

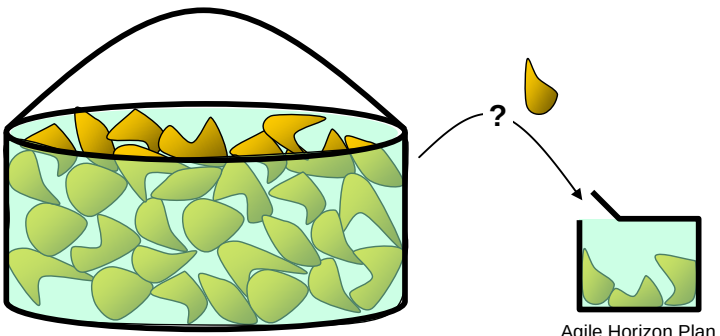
We view the agile horizon plan as an abstracted version of the implementation plan. It comes to the same conclusions, but is stated and manipulated in summary form. It should always be the case that if we can put together a valid agile horizon plan (one that appropriately balances capacity and requirement), then we can produce a valid implementation plan (one that gets all the tasks done with the planned resources by the planned end-date).

To serve the abstraction we take shortcuts and make assumptions when coming up with the agile horizon plan. These simplify the planning. We have found that these simplifications do not reduce the validity of the plan. In other words, even given the simplifications there is little chance that a detailed implementation plan will contradict the agile horizon plan.

That having been said, I consider it to be entirely unnecessary to carry the baggage of a more detailed implementation plan in most circumstances, and find that agile horizon plan combined with agile development methods render a detailed implementation plan an unnecessary burden.

3.6. Eliciting Potential Requirements

Closing on an agile horizon plan involves first identifying the set of potential requirements for the time horizon: a wish list. These potential requirements are such that we can reasonably either include them or omit them from the software. We state them at the level of business requirements. That is, features that have a business benefit when we implement them into the software. The benefit is usually to the customer, but it can be to the software vendor in the case of architectural enhancements that make the code easier to deal with going forward.



We can imagine the potential requirements as each being unique, and all contained in a large bucket. Agile horizon planning involves selecting a subset of features from the bucket for delivery within the planning horizon.

For the first release, it takes effort to concisely define a set of potential requirements. First, we must model the domain, for instance using use cases and UML. Then we can define a set of potential requirements. As well, in a first release, we ought to develop at least an

architectural concept we think will be required for subsequent releases. Therefore, the first release requirements usually come down to a minimum set of features, with a maximum amount of architectural work. This limits the scope of planning activities.

For follow-on releases, the job gets easier. The problem here is not so much eliciting requirements as keeping track of them all.

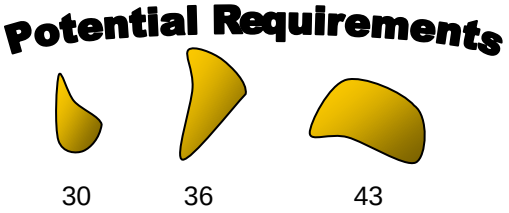
Coming out of the first release effort is a set of features that we put off together with some significant areas where we know we could improve the architecture. These form the basis for a wish list of potential requirements for the next planning cycle. When users start applying the software to practical problems, they have many sensible ideas for improving it. This adds to the wish list.

For the follow-on development, the problem is to keep track of the wish list, prioritize it, size it, and decide which features will make it into the next planning horizon.

3.7. Sizing Potential Requirements

Once product management has identified and prioritized the wish list, it will be necessary to determine the merits of each potential feature by means of a cost-benefit analysis. The benefit is what incremental revenue or reduced costs the proposed feature will bring. The cost is using the development staff to implement the feature. This has both a financial cost and an opportunity cost. The opportunity cost is the benefit of other software the team might have produced during that same time.

For software development, cost is almost entirely proportional to the number of people we have working. Thus, estimating cost requires *sizing* the potential feature or architectural enhancement in units of person-days. For example, a potential requirement may take 30 effective person-days in total to implement.



To size a feature a developer will first seek to understand the specification for the feature and how she will add it to the software. If a specification and design is available, she will use it. Otherwise, she will take an educated guess as to the nature of the specification and from that work out a rough design. She will then divide the implementation into component tasks, and estimate the time she will take to produce debugged code for each component task individually based on her prior experience with the code. Summing the timings for the component tasks, she will arrive at an overall sizing for the feature.

The software organization can improve sizing accuracy over time by conducting post-mortems that compare estimated to actual sizings. This requires that the company keep track of the sizings throughout the project, and that all the developers track the time they spend working on the various features. A few such iterations can greatly improve sizing accuracy.

Another consideration when sizing is the precise nature of the units that we use. Does the estimate include testers and documenters? Is management overhead included? Are days seven hours or eight, or as long as a developer is willing to work? Is it dedicated time or time where other work activities are expected to interfere?

Is 30-days an average-case estimate or worst-case estimate? If it's worst case, what is the confidence interval: will we expect to come in under 30 days 90% of the time, 95% of the time, or something else? If it is average case, then what are the best and worst cases? Are they symmetric?

There are many details surrounding the use of sizings. If we neglect any one of them, the results may be far off. We consider these issues in detail in Chapters 5 and 6.

A shortcut we take for agile horizon planning is to size explicitly only the coding work associated with a feature. We shall see later that we size other tasks, such as system testing, documentation, specification, and design, indirectly from the code sizings.

3.8. Sizing the Available Resources

While feature sizings estimate how much coding work is required for a candidate agile horizon plan, resource sizing estimates how much coding person-power is available to do that work.

We start by examining the pool of capable developers available to work on the software within the planning horizon. For follow-on work within the next planning horizon this is often limited to those who have

experience with the code. Most others will use as much of the other developer's time for training as they will in producing useful work.

For each potential developer we estimate when they are free to start (at least partially) working on the plan, and when they are no longer available. During this time, we count the available workdays and deduct any vacation the developer indicates they will be taking.

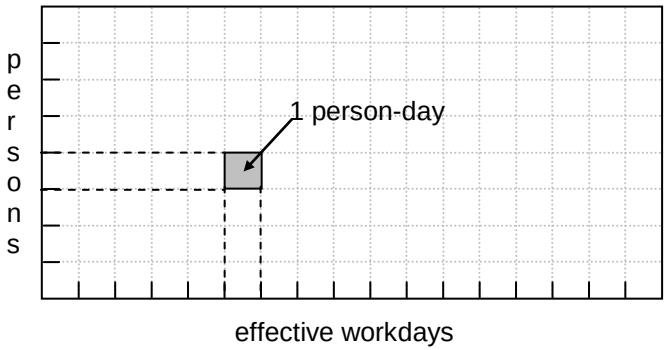
For each developer we then estimate a *work factor*. This factor converts 8-hour workdays into dedicated, uninterrupted hours available to work on putting new features into the next release. This is typically on the order of about 0.6 or so, meaning that for each workday in the release, they can on average spend $8 \times 0.6 = 4.8$ dedicated hours putting features into the release. This work factor accounts for other assigned tasks, sick days, corporate functions, meetings, training, and a poor work environment. On the positive side, it accounts for extra hours spent working evenings and weekends. We combine all of these things into this one work factor.

The result of all this estimation is the average, dedicated number of developers available to us. The units are "ideal developers" who never quit, are available for every workday, and work an uninterrupted eight hours putting new features into the release. We will call these ideal troops "dedicated developers" (no disrespect intended to those 99% of developers who, under this definition, are not considered "dedicated"!).

If we have a pool of ten developer bodies available to us, it is usual for the average number of dedicated developers per day to be as low as four or so.

3.9. The Capacity Constraint

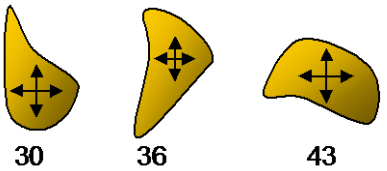
Once we have an estimate for the average number of dedicated developers available during a given planning horizon, the planned capacity we have for coding features is that number multiplied by the effective number of workdays dedicated to coding during the planning horizon.



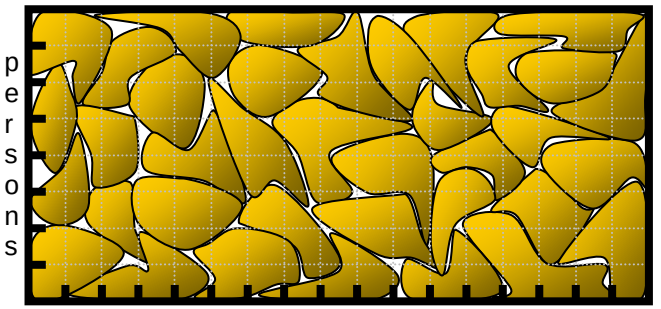
For a traditional release cycle, that is the number of days in the coding phase. For a more continuous SaaS-based release strategy, that is a pre-estimated fraction of the total workdays in the planning horizon. In other words, one can either concentrate the coding in a phase, or spread it out, but either way not all of the workdays go to pure coding work.

As for feature sizing, we give feature capacity in units of dedicated person-days. We can think of our available capacity as a rectangle. The height of the rectangle corresponds to the average number of dedicated developers. The base of the rectangle corresponds to the number of effective workdays. The capacity to do work corresponds to the area of this rectangle, measured in units of dedicated person-days.

On the other side of the balance sheet are the sizings for the features we could add to the software. We should think of the area of each of the features as its sizing in effective person-days.



The basic agile horizon planning problem is to choose a set of features that just fit into the planning horizon. Ensuring the features fit is called satisfying the *capacity constraint*.



The dimensions of planning are which features to include, and the length of the horizon plan. The number of developers is too constrained to be of much use for next-horizon planning.

When we have a plan that balances people, days, and features according to the capacity constraint, we have a partially valid agile horizon plan: "Partially" because we need to consider other resourcing and time using a method of ratios.

3.10. Ratios

As much as other, non-coding activities are important, the target of our efforts must in the end be debugged code with automated tests.

Moreover, coders are usually the scarce, rate-limiting resource in a plan. This is especially true for follow-on work where we are constrained by the availability of developers familiar with the code base. Coders also tend to be the most expensive type of resource.

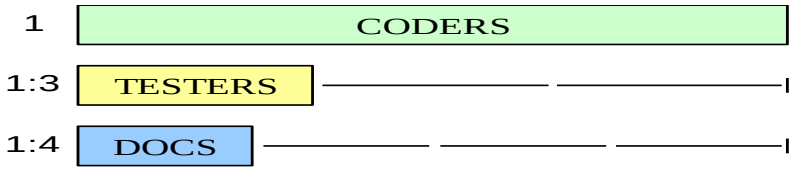
By consequence, when planning it is worthwhile for us to devote the majority of our attention to the coding activities of the project (which, for future reference, such term shall always include the development of automated tests as well). For this reason, we size explicitly only the coding activities involved with adding new features, we look carefully at the amount of time available for coding, and we carefully consider the number of coders who are capable of working with the code.

However, not devoting sufficient time or resources to non-coding activities can have just as bad consequences as for coding; all the same, explicitly planning these other activities with the same care as the coding is not necessary. The complexity that it adds is not worth the increase in accuracy it brings.

The reason for this is that with other, non-coding activities such as system testing, documentation, and the up-front specification and design work, there is latitude regarding how much we need to do. This latitude is not present for coding. To implement a given set of features into the release, all of the necessary code must be finished. However, for specifying, designing, documenting, and system testing a given set of features it is never clear exactly when enough is enough. Therefore,

for these activities we have a greater ability to accept a more fixed deadline and do the best job possible within the constraints given.

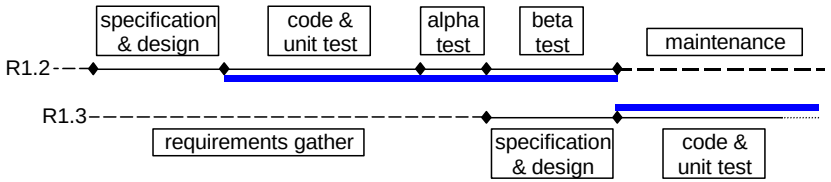
Nonetheless, the agile horizon plan must take into account the amount of time and number of people available for non-coding activities. To account for this time and these resources, we require that they be in certain pre-determined ratios to the amount of coding time and the number of coding resources.



In order to plan for the required test and documentation resource, we use a method of ratios. We have found that ratios of about 3:1 coders to testers, and 4:1 coders to documenters are reasonable. For example, if a plan calls for 12 coders, we must have 4 testers and 3 documenters available to us. In practice, the appropriate ratios will differ from product to product and company to company, and so we must use experience coupled with historical data to determine the situational appropriate ratios.

We assume that we deploy these resources throughout the agile horizon plan. In the case of traditional release cycles, in order to smooth out resource utilization and to shorten the inter-release time gap, we

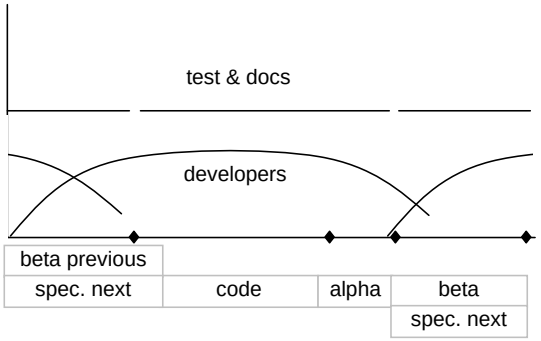
overlap releases, starting specification and design on the next release as the previous one goes into beta test.



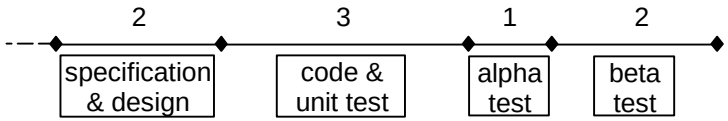
This same overlapping of releases also serves to smooth the use of coding resource. The same can be said on a feature by feature basis in the case of a SaaS lifecycle.

For a well-run development effort, we find that coding work drops off during beta testing as the defect discovery rate drops off. At the same time, demand for coders to work on specifications and designs increases steadily throughout the specification and design phase as we get more into the details.

In practice, overlapping of releases in the manner indicated above allows us to have a smooth deployment of both coding and non-coding resources.



To determine the lengths of the non-coding phases in traditional release cycles, we again use ratios. In practice, we have found that the following ratios of working days in each phase are realistic.



As an example, if we assume coding takes 90 working days, then we estimate that specification and design head-start phase will take 60 working days, and the entire release cycle 240 working days (about one calendar year). Again, the ratios will differ from project to project and company to company, and, so again, we must use experience coupled with historical data to determine the situational appropriate ratios.

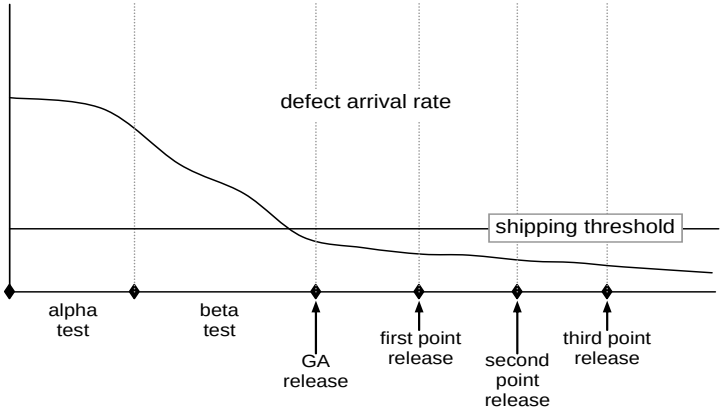
In the case of a more continuous release method, we use an overall ratio of coding days to other days, typically in the range of 3:1 or 3:2 or so. These "other days" are considered overhead days that account for release preparation, stabilization sprints, delayed start sprints due to lagging specification and design work, and so on, but spread all across time and all across developers. It is important to measure the total number of days spent coding versus the total number of days spent on other related activities to hone in on the correct ratio for any specific organization, software product, and release process.

Using such ratios, we work out the number of workdays within the overall planning horizon available to do coding. The effort within those coding days is planned explicitly using the capacity constraint, and is controlled by determining the feature content delivered within the planning horizon.

Therefore, in addition to satisfying the capacity constraint, two further requirements for a valid agile horizon plan are that the number of non-coder resources is in appropriate proportion to the number of coders, and that the number of non-coding days is in appropriate proportion to the number of coding days. If we find on any given planning cycle that any of these ratios are excessively optimistic or pessimistic (*e.g.*, we found that the documentation was of poor quality due to lack of time and resources), then we must adjust the ratios for the next planning cycle.

3.11. Shipping the Release

For a traditional release cycle, once we are at the end of the coding phase, we have done everything we can to ensure the release includes all intended features and will come out on time. We must now monitor the stability of the release to ensure that we can ship on schedule.



We do this by tracking the defect arrival rate: the rate at which we find defects measured in new defects discovered per unit time. Ideally, this statistic should decline during alpha test, and decline further (though at a lesser rate) leading up to the ship date. After the ship date, it should be tolerably low. The company should have policies regarding what arrival rate constitutes "tolerably low".

After the GA release, there will be sequence of maintenance point releases scheduled for once a month or so. During this maintenance period, the defect arrival rate should continue to decline.

To determine whether the software is on-track for its beta and GA releases, management compares the defect arrival rate with historical values for the same product. If the arrival rates are overly high or not declining sufficiently, we should consider postponing the ship date.

To avoid this problem going forward, the company ought to use this new baseline data in determining appropriate coding to test ratios for future releases. Of course, they should also be thinking of ways to decrease the total number of defects that they inject into future releases. However, management would be unwise to count on such improvements until they actually see them. Thus, the company should adjust their ratios nonetheless.

For SaaS based releases, there is often a stabilization time that comes about when we start to notice that defect arrivals are too high. However, these stabilization sprints will typically be scattered throughout the planning horizon in order to keep the software in continuous good repair. However, the sum total of all these stabilization days must be counted as "non-coding" time and our ratios set accordingly to take them into account.

3.12. Summary

In this chapter, we presented an overview of the agile horizon planning process that we will describe in detail in the next several chapters.

We began by describing the business context, iterative release cycle, and overall lifecycle typical of a commercial software vendor company, and another lifecycle more applicable to continuous release SaaS environments.

We then presented a simplified version of an agile horizon plan, stressing its essence as a balance sheet with feature sizings on one side and available resourcing on the other. We emphasized how the heart of agile horizon planning is keeping requirement and capacity in balance with the agile horizon plan acting as our scorecard.

We continued by discussing the relationship between the agile horizon plan and the implementation plan. An implementation plan is a document containing detailed task breakdowns, task orderings, and assignment of personnel to tasks. We can view the agile horizon plan as a higher-level abstraction of the implementation plan. We intend that if the agile horizon plan is valid then we can produce a valid implementation plan in due course.

We then went on to introduce the three cornerstones of agile horizon planning: eliciting potential requirements, sizing those requirements, and estimating how many dedicated developer equivalents we will have available. With this information, we produce the plan guided by the capacity constraint that governs the relationship between time, resources, and features.

Next, we discussed additional validity constraints on the plan involving the use of ratios that relate effort and time for coding days to effort and time required for other purposes.

We then described how, after the code is complete, it is necessary to track defect arrivals, comparing them to historical values to determine whether it remains feasible for the release to be made generally available on its scheduled date. For SaaS-based lifecycles, we stressed the need to monitor this continuously and introduce stabilization sprints where required to bring these values back into line.

The basic approach to agile horizon planning described here works well in practical situations. It is sufficiently lightweight that the fast-paced software organization is not slowed down and yet sufficiently rigorous as to make a real difference in how the company manages its software. The approach is an enabler that allows marketing product management and software development to come to terms, dividing their responsibilities in a way that fosters a good working relationship and results in a successful outcome for both the software vendor and its customers.

Putting the approach into practice is straightforward, requiring only a relatively small commitment as compared to other areas of process improvement. As far as tool support goes, a company can get by with as little as one internal web page per product to hold its horizon plans.

While the basic process is simple to understand, there are pitfalls that can sabotage our efforts. The following chapters 4 through 8 are devoted to a careful study of the details required to make agile horizon planning work in practice, and to some of the larger organizational issues.

4. The Capacity Constraint

A good plan respects always the "art of the possible". It may push up against the limits of what is reasonably possible, but if these limits are broken outright, the plan becomes irrelevant.

As we have seen from the previous chapter, the art of the possible is governed by a relationship between the number of people we have, the amount of time they use, and the effort involved in putting features into the release. We call this relationship the *capacity constraint*. In this chapter, we will talk in general terms about it and some of the considerations surrounding it.

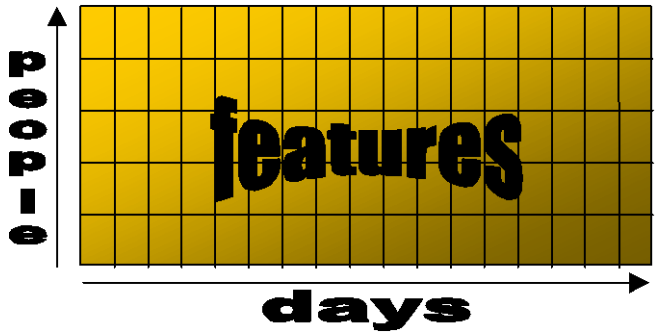
4.1. A Geometric Analogy

As we mentioned previously, the purpose of agile horizon planning is to answer the following three questions and no more.

- **What are we building?**
- **By when will it be ready?**
- **How many people will it take?**

The difficult part is to ensure the "what", "when", and "how many" are in balance. We primarily use the capacity constraint to determine if these things are in balance.

To help explain the capacity constraint, we use a rectangle as a geometric analogy.



The number of people working is the height of the rectangle. The number of days is the length of the base. The number of person-days it takes to implement all the features corresponds to the area.

For a rectangle, the area is the base times the height:

$$area = base \times height$$

For software releases, the analogous relation holds:

$$features = time \times people$$

The more time we have the more features we can put in. The more developers we have (within reason), the more features we can put in. For a given set of features, if the number of developers decreases, the time must increase. This is all common sense.

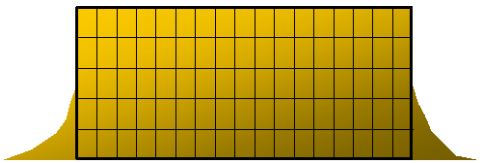
In practice, the dimensions of planning are the base and the area: time and features. The height, the number of developers available, is not usually something we can adjust easily. It is a given, and tends only to

shrink due to unexpected personnel losses (had the losses been expected, we would have planned on them).

The reason it is difficult to increase staffing is that only developers who understand the code base are useful to us. We cannot hire these developers; we must train them. Unfortunately, the act of training them uses as much, if not more, productivity than what they contribute during their training period.

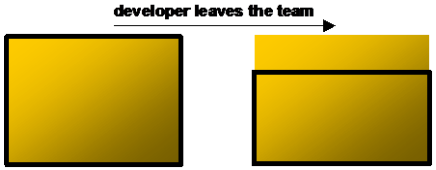
Increasing the strength of the development team operates on a longer-term planning horizon than the next release. We must address these sorts of issues in the software development department's annual business plan, which we will cover in Chapter 16. "Business Planning", on page 323.

The key to planning is that time and features are dependent on one another. With a given a set of developers, if we try to set the base and the area independently, more than likely we will wind up with features overflowing the sides of our rectangle.

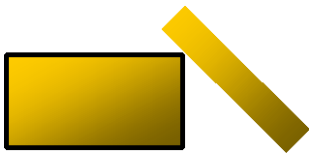


This plan violates the capacity constraint. When this happens, we must take action. We must either increase the time to accommodate the desired features, or cut features from the plan.

A typical mishap is show below. Here one or more developers have left the team leading to a violation of the capacity constraint.

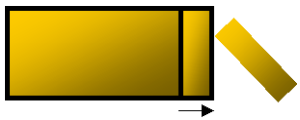


Assuming it is impossible to replace these developers in the short term, we have three options. The first option is to cut features from the plan, holding firm to our dates.

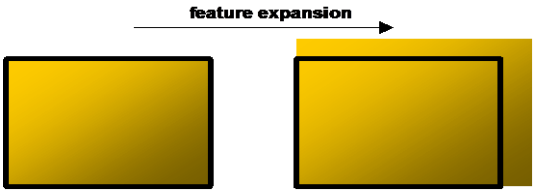


The second option is to hold firm to our feature set, and move the dates out.

The final option, and the most practical, is to combine the two, cutting the less essential features and moving the dates out slightly.

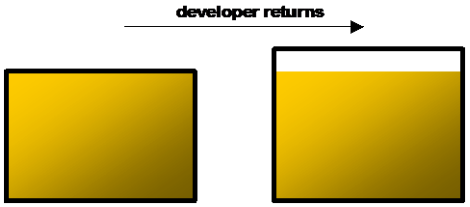


The following diagram represents another mishap. This could represent either additional features coming into plan, or a re-estimation upwards of the amount of effort we require to get the remaining features implemented.



Whatever the cause of the violation of the capacity constraint, the reactions are the same. We can either cut features, extend dates, or both.

Occasionally, good things happen. In the diagram below a developer who knows the code base has come back to us. We can now move in the dates, add features to the plan, or do both.



Whatever the causes or reactions, maintaining the integrity of the capacity constraint requires always that

$$features = time \times people$$

4.2. Organizational Issues

So far, this has all been common sense. Unfortunately, the trouble with common sense is that it is not very common.

Divergences from common sense will generally take one of two forms. The first form is not even trying to estimate how long features will take to implement, or how much actual time developers have available to work on new features. The second form is to deliberately overstuff the plan in hopes that the developers will rally and get it done on time anyway.

Closely related are the two forms of divergence from common sense subsequent to the initial planning. The first is to not track the plan, blithely assuming that everything is going to plan. The second is to know that something significant has just happened (a feature has just been added to the plan, a feature came in very late, or a developer left the team), and still take no action to adjust the plan.

This last can arise all too easily, and stems from a lack of appreciation for the true importance of the capacity constraint to the software organization.

At a certain point in time, the company closes on an initial plan. Assume that development signs off that this is a reasonable plan. As work proceeds, the developers may realize that their initial estimates were off for one reason or another. When they bring up this fact, the response is sometimes, "You agreed to the plan. The business is counting on it. We've made commitments around the plan. Don't tell us your problems. Get it done anyway, that's what you're paid for."

While this attitude is highly "accountability-driven", it is counter-productive.

The software company as a whole must respect the fact that the capacity constraint is central to its business. In other words, the capacity constraint is not a problem for development to overcome; it is a fact for the software company to deal with.

Development is the messenger of the bad news, but the company must rally to solve its problems. This typically means facing the situation squarely, re-planning, and mitigating as best as possible the business consequences. There is no choice.

In fact, it is precisely in these situations where the strength of the business managers can shine through. Developers can let us know the bad news in a blunt way. However, this is not the most appropriate way to pass on the news to our customers. The business side of the operation – sales, marketing, the CEO – must spin things and take concrete action to minimize the impact of the change in plan to the business. If the attitude is one of denial it will be too late to do any of these things when the inevitable happens.

Better still, the business can understand the risks up front and take pains to mitigate their business exposure before the plan slips by setting expectations lower and then over-delivering on them.

4.3. Setting Expectations

Software companies sometimes frustrate their customers by saying very little about future releases. This is sometimes the company's reaction to having made previous commitments on which they failed to deliver. However, customers will not be satisfied with this approach, and the successful business must give out certain information.

On the other hand, it is also surprising how many times a software company will come up with an initial plan and then let their customers and prospects know every detail of the plan. A proviso is always given that the plan is "subject to change". The trouble is, expectations have been set.

In the software business, client and market expectations are key to everything. When we release our plan, we set expectations. If we fall short of those expectations, there will be unfortunate consequences.

The software company must set expectations in such a way that they can be satisfied. This does not mean they have to be completely tight-lipped about their next release. This will backfire also. Rather they must make general statements about the release, and get into specifics as little as possible, and only when there is a compelling reason to do so.

The company must only say as much as it takes to satisfy their customers and prospects, and no more. In this way, they minimize the number of commitments they must make and retain flexibility in planning that will be required as events unfold.

4.4. A Web of Commitments

One of the most important things a company can do is to respect its commitments. In this way, the company will garner the respect, admiration, and goodwill of its customers and partners.

Many customers, and for good reasons, would rather deal with a software company that has poorer software but honors its commitments than deal with one with better software that does not. This goodwill can act as a valuable buffer for the business when times are bad.

The reason for respecting commitments is not solely moralistic. There is a good deal of pragmatics involved as well.

When the software company gives its customer a commitment, the customer turns around and gives commitments themselves. For example, if a software vendor promises to provide a new release of its financial accounting software with a certain set of new features by a given date, the customer will start making plans around that. The purchaser in IT will tell his boss that he can expect the new business functionality to be up by a certain date. The IT department promises the business side this new functionality. The business side promises its partners that new information will be available by a certain date. It goes on and on, extending a surprising distance.

A "web" of further commitments is built around the initial commitment made by the software vendor. If the software vendor reneges on their commitment, everybody looks bad, misses their bonuses, misses their revenue targets, loses that promotion, upsets their customers, upsets their bosses, and so on.

This is hardly the way to treat loyal customers!

4.5. Managing the Plan

The key to avoiding these sorts of issues is to go into a planning cycle with a good plan and track that plan as it unfolds.

The plan will identify dates by when specified new features will be made available. The plan will balance capacity and requirement for both the coding activities (planned explicitly) and the non-coding activities (planned using ratios). The company will formulate the plan in such a way that there is a high likelihood of achieving it.

This agile horizon plan, so constituted, will act as a document that defines what the company is committing to itself to do at the current instant in time. However, as external and internal events unfold, the company will update the plan to reflect these events, and re-balance the plan once it leaves the comfort zone.

The company will use the agile horizon plan internally by its various groups to plan their activities. Development will do detailed implementation planning based on the plan. Documentation will formulate what changes they will need to make to their publications. Client services will determine new training needs both for their own support people and for their customers. Testing will develop their test plans and test cases. Marketing will begin preparing collateral materials (brochures, white papers, advertising). Product management will brief sales on the new features and their benefits. In short, the activities of the entire company will coalesce around this plan.

Development will indicate their status by updating the agile horizon plan on a regular basis. As the plan delta holds at zero, the company

knows that the plan is on track. If the delta drops negative, management will understand there to be a growing need to re-balance the plan.

Externally, the company will use the agile horizon plan as a guideline for determining what to say about new functionality being made available.

While the plan gives specific expected end-dates, in the early stage the software company will only announce availability in general terms, such as by a certain quarter. As the end-dates get closer, and as customers require more certainty, the company can tighten its externally announced dates. The company can do this safely because as they get closer to their end-dates, the probability of hitting the planned dates in the balanced horizon plan rises.

Early announcements prepared by marketing will give the general themes of the production releases, but not disclose specific features and functionality. This is both to retain competitive advantage, and to avoid setting specific expectations as much as possible.

By means of client services, certain customers will be demanding features. Where necessary, and if in-plan, these features can be transformed into commitments, and marked accordingly on the plan. If subsequent to this the company must drop any features from plan, those committed to explicitly must be the last to go. The company does the same to manage commitments required by sales to close new deals.

From the external point of view, the operative concept is to say only that which is required and no more in order to retain flexibility. This flexibility is required in case uncertainties in the plan go against the company, to address issue raised by customers, to enable the inclusion

of functionality required to close new sales, and to give the company room to react to competitive threats and market opportunities.

Thus, the well-run software company embraces the concept of *agile horizon planning*, and orients its activities around it.

By contrast, the poorly run company rarely knows what it is building, or when it will be ready, yet makes more commitments (which it cannot meet) than does the well-run company. Moreover, it finds itself constantly hampered by these commitments and therefore unable to effectively react to customer and market conditions. It often does not realize it is missing its commitments until it is too late to do anything about it. The result is dissatisfied customers, missed opportunities, and, ultimately, a failed business.

In order to improve its ability to engage in well-planned feature releases to the field, a software company must adopt a horizon planning mindset. However, having only a general idea of the horizon planning process is insufficient. The company must have a solid, definitional basis on which to base agile horizon plans so that there be no confusion regarding its terms of reference. Thus, the company must adopt a quantitative view of their plans.

In the next chapter, we will again discuss the capacity constraint, but this time quantitatively, giving the precise definitions that will enable the willing software company to put the ideas into action.

5. *The Quantitative Capacity Constraint*

Up to now, we have looked at agile horizon planning qualitatively, examining the context, benefits, and overall process. In this chapter, we begin to look at quantitative definitions for the terms we have been using informally up to now.

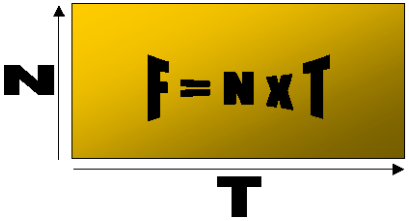
While it may seem excessively detailed, experience has shown that a rigorous definition of the numerical quantities involved is necessary to apply agile horizon planning in practice. Only with precise definitions can we determine the validity of an agile horizon plan. Without these precise definitions, there is room for interpretation that can lead us to believe that our plan is balanced when in fact it is anything but.

To avoid this situation, and to eliminate any questions of interpretation, we present a quantitative formulation of the capacity constraint here, with rigorous definitions of all the quantities involved.

To avoid confusion, for the bulk of the chapter we will quantitatively treat a planning cycle where all coding is concentrated into a single coding phase. Necessary modifications must be made when dealing with other methods of release, such as a more continuous SaaS-based release approach. We will end the chapter with a demonstration of how to modify the capacity constraint accordingly.

5.1. Basic Definitions

We have seen previously that a rectangle filled with features provides us with a visual analogy for the capacity constraint.



The base of the rectangle is the number of working coding days available over the course of the planning horizon. We will refer to this quantity as *T*.

The height of the rectangle is the average number of dedicated developers available to us during each of those *T* days. We will refer to this quantity as *N*.

Therefore, the total number of dedicated developer-days available to us in the plan is $N \times T$.

We fill the interior of the rectangle with features. We will refer to the sum of all the individual feature sizings (expressed in dedicated developer-days) by the quantity *F*.

Expressed in these terms, the capacity constraint requires that

$$F = N \times T$$

or, in words, that the requirement to get work done *(F)* is exactly balanced by the capacity to do work *(N × T)*

5.2. *Post-Facto* Considerations

In formulating a precise definition for the numerical terms in the capacity constraint, we will adopt a *post-facto* (after-the-fact) mindset. In other words, we will define the quantities with a view towards gathering certain metrics during the time horizon in question, and then computing realized values for *N*, *T* and *F* using these metrics.

Even though when we are planning we will need to estimate these quantities in advance, adopting a *post-facto* definitional framework is a good way to proceed for two reasons.

Most obviously, we *will* gather these metrics during the planning horizon, and we will want to compute these numbers so that we can compare them to our initial estimates. It is only by closing the loop on our estimates that we can improve them going forwards.

A second reason is for the sake of clarity in estimation. With good *post-facto* definitions we know exactly what it is we are trying to estimate.

The definitions for *N*, *T*, and *F* will be such that after we have completed a release, if we were to perform a *post-mortem* we would find that the capacity constraint held. That is, if we were to measure the average number of full-time equivalent developers who worked on the release *(N)*, count the number of working coding days they had on the release *(T)*, and examine time logs to determine the total effort put into coding all features *(F)*, we would find that $F = N \times T$.

It would not matter if the release were a disaster or a great success. After the dust has settled, we will always be able to verify that the capacity constraint held.

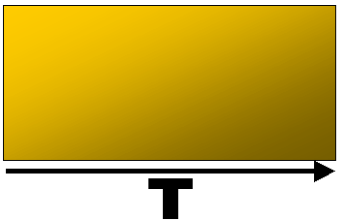
This does not happen by coincidence, it happens by definition. The capacity constraint is a true *constraint*. It is not a *suggestion* that we ought to balance the capacity and the requirement. It is a law of nature that constrains us to being able to put into a plan only as much effort on features as we have person-power available to us, no more and no less.

However, making things work out so that, *post-facto*, $F = N \times T$ always, takes a certain precision of definition. If we do not make the effort and define these terms precisely, the constraint would not hold, and, by consequence, we could make errors in our planning. It is well worth the effort in being explicit and making our definitions rigorous.

With that said, let us now define *N*, *T*, and *F* in detail.

5.3. Number of Workdays, *T*

Let us start by defining the simplest quantity, *T*.



T is the number of full-equivalent working days from the start of coding to the end of coding. For example, if we intend to start coding on December 10, 2007 and finish on February 8, 2008, then there are $T = 39$ working coding days (this excludes weekends and the time from Christmas to New Year's).

It will not necessarily be the case that all developers work all of these *T* days. Some may have vacation, or some may take sick leave.

This does not matter. Only if we know with certainty that no developers can work on a certain day should we remove that day from *T*.

What about developers working weekends and holidays? As a rule, statutory days off should never be included in *T*. If developers persist in working days not included in *T*, the extra work will be taken into account elsewhere. *T* can be fractional in those cases where, for instance, all developers have a half-day off.

5.4. Developer Power, *N*

From the simplest, *T*, let us now move on to the most subtle, *N*.

N is the average number of *dedicated developers* per day working during the period *T*. The term "dedicated developers" takes some explanation. As a simple example of *N*, say we have five developers available for every one of the *T = 39* days from the previous example, each working 8 uninterrupted hours per day adding features to the release, then, under these circumstance, *N = 5*.

Central to the definition of *N* is the notion of a *dedicated developer*. A dedicated developer is one who works 8 uninterrupted hours for every one of the *T* days of the coding phase doing nothing but adding features

to the release. A dedicated developer is an idealization that does not exist in reality.

The reason for this is that we do not assume that *body time* is the same as *dedicated time*. Body time is the time during which a developer's body is available to work each day. Let us assume that one workday consists of 8 body hours. That is a 9 to 6 job with one hour for lunch. Dedicated time is the uninterrupted time during the day used to get new features into the release. Depending upon what else is on the go, this might be, for example, only 4 hours.

The word "uninterrupted" is important. Consider two developers. They both come to work for 8 hours. They both are doing something else for 4 of those hours. If the first developer had his 4 hours of feature work all in one adjacent chunk of time, then the number of uninterrupted hours he worked was 4. If the second developer had her 4 hours of feature work come in fits and starts throughout the day, she may have gotten the equivalent of only 2 hours uninterrupted work done because of all the distractions. Her dedicated time is therefore only 2 hours.

For a developer to have fewer dedicated hours than body hours does not necessarily carry a negative connotation. The developer's manager may be asking him to be doing other important things during the day. This serves to reduce the amount of dedicated time available for the release, but reflects well on the developer.

Given that each developer understands what a dedicated hour is, and given that the climate of the organization is such that they have no fear in being honest about reporting it, then it is possible to compute a *post-facto N* for the project.

To do this, each developer must carefully track all the dedicated time spent adding new features into the release during the T day coding phase. Say there are a total of n developer bodies working during the coding phase. For the i^{th} developer, $1 \le i \le n$, we will call the measured number of dedicated hours h_i. We then have,

$$N = \frac{\sum_{i=1}^{n} h_i}{8 \cdot T}$$

In words, we sum all the hours spent over all of our pool of n developers, divide by 8 hours to convert dedicated hours into dedicated days, and then average the result over the T days of the coding phase by dividing by T. The result is the average number of dedicated developers deployed per day during the coding phase.

If each developer actually spends 8 dedicated hours for each of the T days, then $N = n$. The simple example at the start of the section assumed this. It is unlikely to happen in practice.

5.5. Attributing *N*

For estimating N in advance, the numerical formulation given above is not very useful. It is difficult to estimate h_i, the total number of hours that we expect a given developer to work during a release cycle, in advance. Moreover, if we find that our measured h_i differ widely from our estimated ones, it is unclear how to go about attributing the estimation error. To make the capacity constraint more usable, we need to develop a breakdown of N in which the terms are more intuitive.

An improvement is to think in terms of a *work factor*, w_i that for the i^{th} developer converts body days into dedicated days.

For example, a typical work factor might be 0.5, indicating that a developer can, on average, find half a dedicated day, equal by definition to 4 dedicated hours, each workday. It is easier to estimate a work factor than to estimate the total number of hours a developer will work during the release cycle.

A developer's work factor will be applied to the number of days they, in particular, were expected to have worked during the T day coding phase. For the i^{th} developer we will call this quantity t_i.

This quantity is always less than T. We subtract from T any days not allocated (at least partially) to the project, and any vacation days taken during the time allocated. Here vacation days mean discretionary time off that the developer does not intend on making up. Another way of stating t_i is as follows,

$$t_i = d_i - v_i$$

where d_i are the number of days at least partially allocated to the coding phase of this release, and v_i are the number of vacation days taken during that time. This leads to the following definitions for w_i and N.

$$w_i = \frac{h_i}{8 \cdot t_i}$$

$$N = \frac{\sum_{i=1}^{n} t_i \cdot w_i}{T}$$

If we substitute w_i into the equation for N, we'll get back the original equation for N.

Let us now look at some examples that illustrate these definitions.

Assume that the coding phase is $T = 39$ workdays long. A certain developer, Bob, tells us that he had 35 workdays on coding the release (he started 4 days late because of other projects), and then took 5 days of vacation. Bob's workdays were therefore 30.

$$\begin{aligned} T &= 39 \\ d_{bob} &= 35 \\ v_{bob} &= 5 \\ t_{bob} &= d_{bob} - v_{bob} = 35 - 5 = 30 \end{aligned}$$

Say Bob called in sick for two days. That does not affect these numbers in any way. The workdays are still $t_{bob} = 30$. The hours lost during those sick days will go to reduce Bob's hours (h_{bob}) and hence his work factor, (w_{bob}) but not his days worked (t_{bob}). The only thing that can reduce days worked is taking extra vacation (v_{bob}), or re-assigning Bob to different projects (d_{bob}).

Say Bob took a morning to run some errands and an afternoon to see Star Wars Episode 3. These were not vacation days, but rather time he intended to make up. Again, even though Bob was not at work for the equivalent of a day, the workdays are still 30. The non-vacation time-off day he took reduces his hours and therefore his work factor.

Say Bob makes up the time on a Saturday. Say even that Bob has no life whatsoever and works every weekend as well as during the week. Even though Bob worked all these extra days, the workdays in the formula remain at 30. The extra time gained goes to increase Bob's measured h_{bob}, and hence increase his work factor, w_{bob}.

If Bob took one fewer or one more vacation day, this would affect v_{bob}. If we take Bob off the release prematurely so that he can work on a different project this would affect d_{bob}.

Bob tells us that during the release cycle he put in a total of 120 hours of dedicated time on new features in the release ($h_{bob} = 120$). This number should include *all* dedicated time, including dedicated time during the workday and dedicated time working after hours and weekends. Bob's average number of dedicated hours per workday over his 30 workdays is therefore $^{120}/_{30} = 4$. Dividing to convert dedicated hours into 8-hour dedicated days, we get that Bob's work factor is $^{4}/_{8} = 0.5$.

$$h_{bob} = 120$$

$$w_{bob} = \frac{h_{bob}}{8 \cdot t_{bob}} = \frac{120}{8 \cdot 30} = 0.5$$

Work factors can theoretically be greater than 1. If a workaholic developer tells us that their average was 12 hours of dedicated time each workday, then their work factor would be $^{12}/_{8} = 1.5$.

Unexpected days off, sick days or family emergencies, will reduce w_i. Discretionary days off, planned vacation, will not. If a developer takes off the morning but then works late at home, this has an entirely neutral effect. Vacations are those days that are *gone*, not those that the developer makes up.

Note that some developers will be able to accomplish more in a dedicated hour than others. We do not consider this in the work factor. We consider this in the feature sizings later on.

When we go beyond the initial, simple definition of N to include d_i, v_i, and w_i, what we are doing is defining quantities that help us to attribute the contributors to N in an intuitive fashion. If we have a pool of 20 developers working on our release, there are then $20 \times 3 = 60$ individual contributing factors: d_i, v_i, and w_i for each of the 20 developers.

Attributing N in a fine grained, intuitive fashion will later enable us to hone in on the causes of estimation error and understand those factors that reduce our developers' productivity.

There are many alternative ways of dividing N into contributing factors. For example, say that sick days taken by developers were of particular concern to the organization. Then we might introduce an s_i analogous to w_i which is the developers propensity for sickness ($s_i = 0$ is perfectly healthy, $s_i = 1$ is deceased).

The details of any particular attribution of N should be suitable to the needs and concerns of the developing organization. The particular division we described in this section is a reasonable possibility that has proven effective in practice. In any case, the exact formulation is less important than having some division that is rigorously defined and self-consistent.

5.6. Factors Affecting w_i

In our experience, a typical factor for a developer not yoked with any management chores, working on only the one product (new features in the next release and maintenance on previous releases), and in a good working environment, will have a work factor of around *0.6*. The

reason that a developer's factor is typically less than 1 is intuitive. During the day, most people need to do more than only work on new features for the next release.

While there are different schools of thought on the issue of who does maintenance on previous releases, one common and efficient approach is to have developers simultaneously work on new features in the next release and fix defects in the previous release. If this is the case, then this is a large contributor to reducing the work factor (although, as was mentioned earlier, there is no negative connotation associated with this).

If we factor in training, team leader duties, company parties, meetings with clients, time spent reading this book, and so on, pretty soon we are down to only about 2 or 3 hours of dedicated time left out of the 8-hour workday.

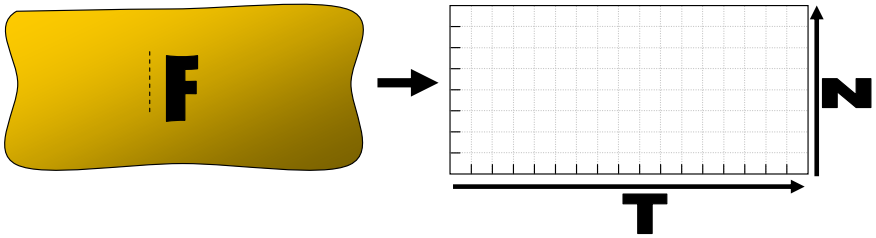
Say there are 3 hours left. With those scant 3 hours, if the phone keeps ringing, if people keep dropping in on us, if those 3 hours gets spread willy-nilly across the 8-hour day, then we will get considerably less development done than if we can closet ourselves away for 3 solid hours with no disruptions. For this reason, those 3 hours turn into an effective 1 hour of dedicated time.

Say that we have 16 developers like this working on our release. That is, each of the 16 developers has only 1 hour of dedicated time available each day. Each of their work factors is therefore $^1/_8$ and $N = 16 \times {}^1/_8 = 2$. Our 16 developers just got reduced to 2! Factors like this are not unheard of, and they can cripple our productivity.

This illustrates why it is worthwhile for an employer to do everything they can to get their developers' work factors up. For

example, private offices with doors that close, the ability to turn off phones, fewer and more focused meetings, and so on. It is also good to have the developer work on only one project at a time. Working on two different projects at the same time will typically more than halve the work factor for each project, owing to the overhead of switching between the two. DeMarco & Lister have an excellent discussion of these topics in the book Peopleware, [Dorset House, 1987].

5.7. Effort, *F*



Let us now go over to the other side of the capacity constraint and define *F*, the total number of dedicated 8-hour person-days *required* for coding all the features into the release. In our geometric analogy, *F* corresponds to the area that we are trying to fit into the rectangle defined by *N* and *T*.

In practice, *F* is something we have to estimate ahead of time. After the fact, however, it is relatively easy to measure.

$$F = \sum_k f_k$$

Where f_k is the effort required to put the k^{th} feature into the release measured in dedicated days (1 dedicated day = 8 dedicated hours). We measure it by asking the developers (after they are all finished) how many dedicated work hours in total they spent during the coding phase on the k^{th} feature. We then divide by 8 to get the total number of dedicated workdays they spent on the feature. We do this for each feature and then sum to get F.

The measured f_k should include all work done during the coding phase by the developers in regards to that feature, and not just coding work. In particular, we should include any extra specification work and extra design work. The implication for *a priori* estimation is that we should estimate all work that we would expect to go on during the coding phase, and not purely the coding work. Ideally, if we complete all the specifications and designs on time, this will be only coding and unit testing work.

5.7.1. Common Work and Abandoned Features

In measuring a *post-facto* f_k, it will sometimes be the case that a developer does work common to two or more features. In this case, it is not possible for the developer to allocate the work to just one feature, as we have required.

We try to deal with this ahead of time by either combining the features into one, choosing the most fundamental feature to bear the cost of the common work, or by explicitly listing the common work as an architectural enhancement "feature", whichever makes most sense.

In the latter two cases, the agile horizon plan should indicate a pre-requisite relationship amongst the features so that nobody gets the idea

they can just delete the common work and still get the dependent features at the same estimated cost.

If the common work was unexpected (which is usually delightful), it should be intelligently pro-rated to whatever features required it most and that remained in plan. This will involve the developer tracking this common time separately, and at the end of the coding phase pro-rating the work in a sensible fashion.

Another difficulty occurs when a developer starts coding on a feature that we later abandon from the release.

As this is an inefficient use of resource, we try to avoid this situation as much as possible. In particular, we have a rule that says only features that the developers have not yet started are liable for removal from plan.

All rules have exceptions, however, and if we nonetheless abandon a feature in mid-development we deal with it by leaving the feature in-plan in an "abandoned" state. We charge the developer's time to-date against that feature, and we sum it at the end of the release cycle with all the other features to determine *F*. Naturally, even though it is in-plan, the "abandoned" marker indicates the feature is not implemented in the release. These measures retain the integrity of the capacity constraint in the face of abandoned features.

5.8. Developer Productivity

Nowhere yet have we discussed the issue of *who* will be doing the work. Different developers have different levels of productivity. Some of this is due to different work factors, but some of it is due to other factors, such as training, experience, and raw talent. As well, a certain developer may be more productive on one kind of feature than on another because they are more familiar with that domain, that technology, or that part of the code.

It is interesting that the *post-facto* capacity constraint does not care about this point. Everything evens out in the end. The total time spent on features will equal the total time available to work on features, no matter if we hire the best or the worst to do the work.

Surely then, we are missing something in the capacity constraint? The problem only comes when we think of feature sizings as independent from who is doing the work. This is the usual approach, and is why academics suggest we do not size features in terms of person-days directly, but rather in terms of lines of code, function points, or some other metric.

When we estimate a feature sizing in person-days, we are also making an estimate as to who will be working on the feature, and how productively they will use each dedicated hour during that work cycle. We have found that, in practice, estimating simultaneously who will work on the feature and how much time they will spend on it is eminently practical, and in fact more natural than indirect approaches.

In the common commercial software development situation, we are familiar with the capabilities of the developers, and they can participate in estimating the feature sizings. Developers are usually more comfortable saying how long *they* will take to implement a feature than in making some abstract sizing estimate (say in terms of function points) and then another abstract estimate as to how productive they are in coding function points into software. Likewise, technical managers are often quite comfortable estimating how long a particular individual will take to implement a feature.

If, for some reason, the developer we were expecting to work on a feature is not available to work on it, and if the replacement takes much longer, then we consider this to be, in fact define it to be, a sizing error. Therefore sizing errors can come both from poorly estimating the amount of work involved in a feature, and from poorly estimating who will work on the feature and how productive they will be.

The estimation framework we discuss here does not constrain the manner in which feature effort estimates should be arrived at. It only requires that, at the end of the day, that the development organization supplies an estimate in effective coder-days.

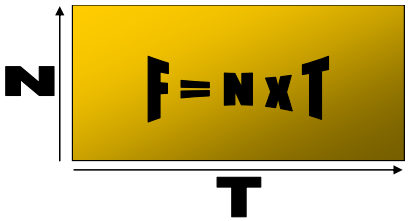
The effort estimate therefore combines three independent variables:

- The size of the feature (for example in lines of code or function points).
- Which coders will work on the feature.
- The productivity of those coders working on that feature (in lines-of-code or function points per effective coder-day for each coder).

If the estimate of any of these three things is inaccurate, the feature effort estimate will be inaccurate. For example, if the coder originally expected to work on a feature must be replaced by a different coder with a different productivity rating on that type of feature, it is an estimation error.

If the organization has a preference for how to estimate feature sizings, it can use it to advantage, combining it with the agile horizon planning approach.

5.9. $F = N \times T$



Given the definitions to date, we will now have *post-facto* agreement between F and $N \times T$.

We defined N such that *only* dedicated time working on new features was included. Moreover, we defined it so that *all* dedicated work on new features was included.

Similarly, we defined F so that it consists of *all* dedicated time working on new features, and *only* that time.

Given these complementary definitions, it is not surprising that, *post-facto*, $F = N \times T$.

The situation is analogous to accounting balance sheets. If you are like me, and you happen to be exposed to balance sheets, every time you look at a corporate balance sheet you are amazed how the assets always work out to be *exactly* the same as the liabilities plus the shareholders' equity.

Well, if you are an accountant, it is not that astounding. An accountant knows that these things are designed so that they must equal one another. If they do not it is because something on the assets side was not accurately reflected on the liabilities side: that there was an error in the book-keeping. This is the essence of double-entry book-keeping.

Similarly, we are using a sort of double-entry book-keeping to keep track of F (the assets) and $N \times T$ (the liabilities). Every hour that every developer spends working on new features in the release must appear in both F and in the computation of N. The way F and N are designed then requires that $F = N \times T$.

Only when we have this solid definitional basis for our quantities can we try to estimate these quantities in advance. If we defined the quantities inexactly, we would never be sure exactly what it was we were estimating. As it is, we know what we are trying to estimate, and after the fact, we can check to see how close our estimates came to what actually happened. In this manner, we have a firm basis for improving our estimation accuracy.

Note that we do not estimate F and N directly. Rather we estimate each of the contributing f_k , n, d_i, v_i, and w_i separately. After the fact, we then measure each of these quantities individually to attribute our estimation errors in a fine-grained manner.

For example, our *post-mortem* might reveal that our estimate for w_i, an employee's work factor, was off by a significant amount as compared to the actual. We then know to apply our effort to figuring out how to better estimate *w* in future.

5.10. Proof of the Capacity Constraint

As stated in the previous section, it is a requirement that *post-facto* $F = N \times T$. In this section, we shall prove it mathematically.

Assume that the i^{th} developer records the number of dedicated hours working on the k^{th} feature during the 24-hour period of the d^{th} working day. Call this quantity: $h_{i,k,d}$. In practice, it is a good idea to implement a fine-grained time-tracking system that is capable of tracking this quantity. We discuss this in Section 12.6, "Effort Tracking", on page 264.

By definition, the time in effective coder days spent on the k^{th} feature is given by summing the time spent by all developers on all days on that feature expressed in effective coder hours, and dividing by a nominal 8 to convert to effective coder days:

$$f_k = \sum_i \sum_d \frac{h_{i,k,d}}{8} \qquad \text{(Equation 1.)}$$

Given that

$$F = \sum_k f_k \qquad \text{(Equation 2.)}$$

Hence, the time required for all features is:

$$F = \frac{\sum_{k}\sum_{i}\sum_{d} h_{i,k,d}}{8} \qquad \text{(Equation 3.)}$$

As we have seen in Section 5.4, N is defined as:

$$N = \frac{\sum_{i=1}^{n} t_i \cdot w_i}{T} \qquad \text{(Equation 4.)}$$

And, from Section 5.5, w for the i^{th} developer is defined as:

$$w_i = \frac{h_i}{8 \cdot t_i} \qquad \text{(Equation 5.)}$$

Where h_i is the number of hours spent by the i^{th} developer on all features on all days:

$$h_i = \sum_{k}\sum_{d} h_{i,k,d} \qquad \text{(Equation 6.)}$$

Combining Equations 4, 5, and 6 gives:

$$N = \frac{\sum_{i}\left[t_i \cdot \frac{\sum_{k}\sum_{d} h_{i,k,d}}{8 \cdot t_i} \right]}{T} \qquad \text{(Equation 7.)}$$

Simplifying yields:

$$N = \frac{\sum_{i}\sum_{k}\sum_{d} h_{i,k,d}}{8T} \qquad \text{(Equation 8.)}$$

or:

$$N \times T = \frac{\sum_i \sum_k \sum_d h_{i,k,d}}{8} \qquad \text{(Equation 9.)}$$

Substituting Equation 3 into Equation 9 yields

$$N \times T = F \qquad \text{(Equation 10.)}$$

which is the capacity constraint — *Q.E.D.*

What this demonstrates is that each quantum of coder work on a new feature in the next release, $h_{i,k,d}$, goes both to the right side and the left side of the capacity constraint, and serves as a check that our terms are well-defined in a consistent manner.

The work quantum contributes to work done on each feature across all developers and all days. The work quantum simultaneously contributes to work done by each developer across all features and all days. The contribution to each side of the equation is equal, and therefore requires that the capacity constraint be maintained.

5.11. Modifications for Continuous Release

When using a different style of software release, the details of the definition of the capacity constraint will differ. For example, for a more continuous SaaS release methodology the multiple teams of developers will work in sprints, taking time between the sprints to prepare for the next one, and inserting stabilization sprints from time-to-time as the software quality warrants.

In such a case, we might choose to model it using the following modified quantitative capacity constraint:

$$F = N \times T$$
$$T = cD$$

Where D is the number of workdays over the entire planning horizon, c is the *coding factor* which converts workdays into "predominantly coding days" (PCDs), and N is the average number of dedicated developers deployed across those PCDs.

We can define a PCD in any number of ways. One way is to declare that a PCD for any individual coder is any day where they spend more than 1 hour coding new features. Alternatively, for each member of a team it could be any day management declares to be a PCD for that team. Or it could be defined for an individual coder as any day that coder declares to be a PCD. Any of these definitions will work, provided there is some way of recording it. The "1-hour" rule is particularly easy to measure given we are tracking time.

Let P_i stand for the number of PCDs of the i^{th} coder (as determined using any of the methods discussed previously, but applied consistently to all coders), and let P (un-subscripted) stand for the average P_i over the n coders as follows.

$$P = \frac{\sum P_i}{n}$$

Then the coding factor is computed as.

$$c = P/D$$

This is the average PCDs across the *n* coders in ratio to the total number of workdays in the planning cycle, which intuitively corresponds to a "coding factor" which takes into account all of the predominantly non-coding days spread throughout the planning horizon. Note that this same formulation can be used for the "Big Bang" release cycle as well, though there is no need to estimate *c* as it is planned.

We must then redefine the work factor in terms of PCDs as follows.

$$w_i = h_i/8P$$

Where h_i is the total feature coding hours of the i^{th} coder. Any hours spent coding new features outside a PCD will go to increasing the coder's work factor, the same as would coding on a weekend, for example. Note that a coder's work factor is now influenced by the average behavior of other's, as this formulation implicitly includes how this coder's PCDs compare to all others.

N is then simply the sum of the work factors.

$$N = \sum w_i$$

In this formulation of the capacity constraint, the things we would estimate on the capacity side of the equation are the work factors and the coding factor, whereas before we only estimated the work factors.

5.12. Summary

In this chapter, we assigned quantitative meaning to the capacity constraint that we had looked at only qualitatively until now.

In attaching quantities to the capacity constraint, we conceptualized a division of the contributing factors that is intuitive, amenable to estimation in advance, and that provides us guidance in attributing estimation errors and in leading us to ways to improve productivity.

Next, we will consider the capacity constraint from the estimation standpoint, which inevitably brings in interesting questions concerning randomness and probability.