$40

The Agile Planning Horizon in Professional Software Development

Managing at the dynamic boundary where business necessities meet software development realities.

David A. Penny, *Ph.D.*

15 September, 2012 (12th edition)

ABOUT THE AUTHOR

Dr. Penny has had a mixed industrial and academic career spanning over 25 years involvement in software development.

As a Ph.D. student at the University of Toronto specializing in languages, operating systems, and software engineering, Dr. Penny was involved in software production initiatives including chemical-physics simulation software, robotic control software, variable-precision numeric libraries, language run-time environments, the Object-Oriented Turing IDE, the Mini Tunis teaching operating system, the Software Landscape, a program verification system, and the Polyx multi-processor operating system.

Following graduation, Dr. Penny took a position at IBM working on an integrated development environment for C++ on the AIX platform.

Dr. Penny then joined Algorithmics Incorporated becoming its CTO and VP Software Development. He led the development of RiskWatch™, the industry's leading middle-office financial risk management software for global banks, and related products.

After Algorithmics, Dr. Penny brought his ideas to a larger population as a software development management consultant and as an Associate Professor of Computer Science at the University of Toronto.

Dr. Penny then joined Electronics Workbench, a provider of packaged Windows-based software to electronics professionals and educators, as Vice President of R&D. He is currently CIO at Ceryx Inc., an applications service provider where he is adapting his ideas to fit delivery of Software-as-a-Service.

ACKNOWLEDGEMENTS

I would like to express my thanks to the professional software development colleagues I have encountered during my career and who have informed and inspired so much of this book.

Special thanks to Arthur Tateishi, a long-standing colleague and friend with whom, over the course of numerous professional software development engagements, we have developed many of the ideas forming the basis of this book.

My thanks to Professor Richard C. Holt for mentoring and assisting me in the early days of my career, and providing leeway to allow me to try out my ideas and try my hand at running software projects.

Thanks to Steve Rosenberg for many discussions on software development management, and for recommending me for my first executive management role. Thanks also to Ron Dembo, Bill Dinardo, Bill Wignall, and Gus Harsfai, for believing in these ideas and allowing me free reign to implement them, and to Jonathan Rose for hiring me to teach a software engineering course to his 4th year students which inspired the creation of this book.

To Matt Medland, Braulio Lam, Derek Mock, Tom Berenstein, Erich Klein and my numerous students for taking these ideas and putting them into practice so wonderfully within their software development organizations.

And finally, a big thanks to Matt Medland for his detailed comments on the book. Any remaining errors are entirely my own.

Contents

Preface

The programming of computers is still a relatively young field. Up until the 1970's, the scarcest commodity required for programming was computer time. Everything that could be done to conserve computer time was worthwhile, even at the expense of vast person effort.

With the advent of more affordable computers, that situation changed. As a graduate student working at the University of Toronto in the 1980's, we had Sun™ workstations running Unix™ at our desks. We self-organized into teams who interacted every day, we designed as we went, and we had continuous feedback with stakeholders. We continuously integrated our work with other team members using source code control systems developed by our contemporaries, and we strove to put in place automated testing frameworks because manual testing was ineffective and just plain boring.

We were shocked to learn that there was another world, a Corporate world, a Department of Defense world, where very large teams of people wrote volumes of documentation and had them signed off according to rigorous processes, not approaching an actual computer until much later. Testing was manual and performed at the end, only then exposing significant problems that caused massive delays as code designed by different teams had to be made to work together.

Many of us at the time were able to avoid this monstrosity in its entirety. After completing my Ph.D. in the field of software engineering I moved on into industry at the height of the dot-com bubble in the mid 1990's where I became a manager/architect/coder, learning my

professional trade as I went, leading the development efforts of one of Canada's most rapidly growing software companies. We did not look back. We used what worked for us on the large University projects, learned from one another, and learned how to work effectively with important customers and the business side of the organization.

It was during those early professional years where I first developed many of the concepts and ideas captured in this book, and later honed them by applying and adapting them to other organizations I managed, and by evaluating other companies and helping them as a consultant.

The genesis of this book was in the summer of 2005 upon a request from the Chair of the Department of Computer Engineering at the University of Toronto, Jonathan Rose (a TA of mine when I was an undergraduate, and the very successful founder of his own company that wrote innovative FPGA design software based on his research).

Jonathan asked if I would teach a *practical* course on software engineering to their 4th year students. I told him if I did, I would throw out the traditional software engineering textbook (essentially unchanged since I had taken the course two decades earlier) and start fresh based on what I had learned myself. Jonathan was encouraging of this approach, so I wrote the first iteration of this book in that summer, and I have been teaching the course and refining the book in my spare time ever since.

My hope is that you may find something within these pages to assist you in our shared passion: developing great software without drama!

David A. Penny
Toronto, Ontario
September 2012

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.

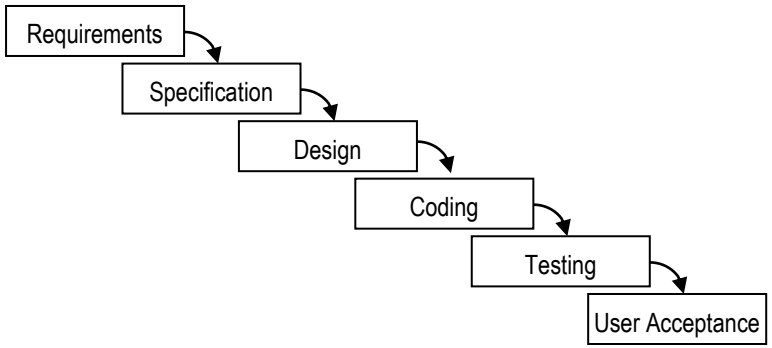
This is not a book specifically about how to design or code or test, but rather the focus is on the *management* of software projects. Building large-scale software systems is a problem of coordination, which in our case lies under the purview of technical software development management. However, I 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.

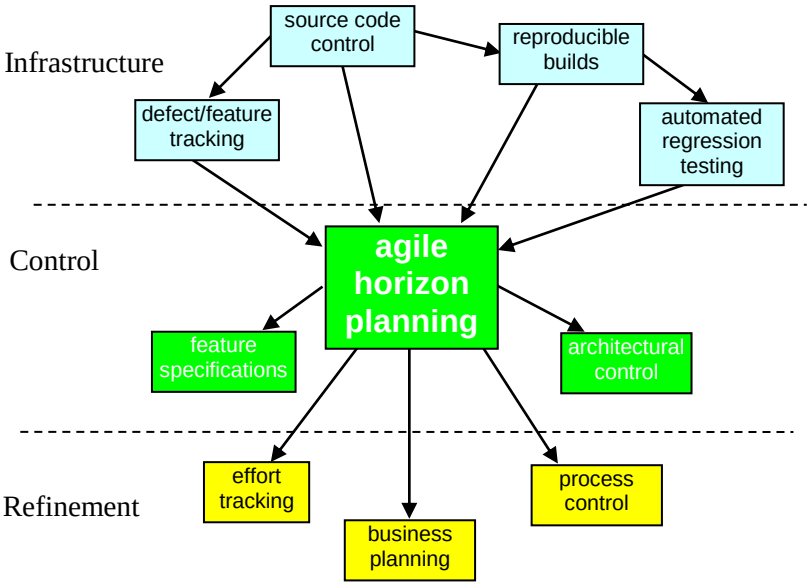
Finally, as an umbrella for accomplishing all of the foregoing activities, a development manager must think like any other business manager and build a business plan for the development group. The purpose of such a plan is to build confidence within the executive team that the situation is well-assessed, 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,

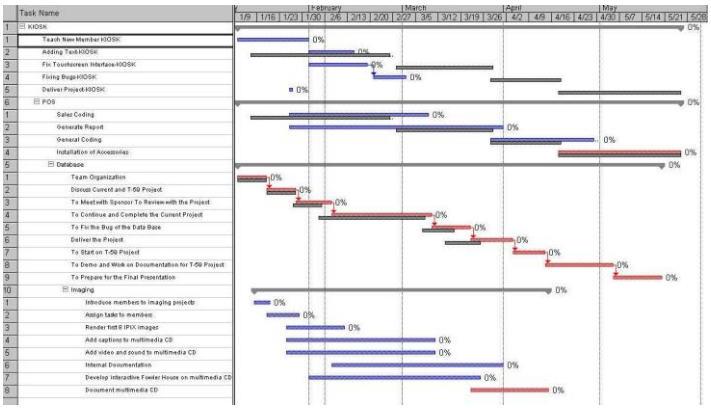
and sales are dependent upon the organization having a unified view of the expected functionality and release date. If there are to be any changes, it is essential that the software development department provide warning well in advance to allow the company to take mitigating actions as early as possible.

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.

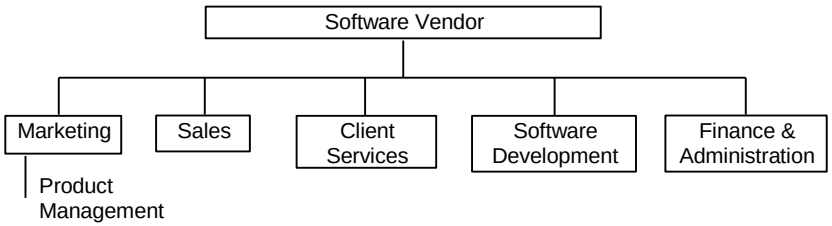
We will be discussing agile horizon planning in the context of a software vendor organization wishing to better control follow-on releases of their software product. The median of such product ventures will involve around three dozen people and a code base of a million lines of code or so. However, the 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.

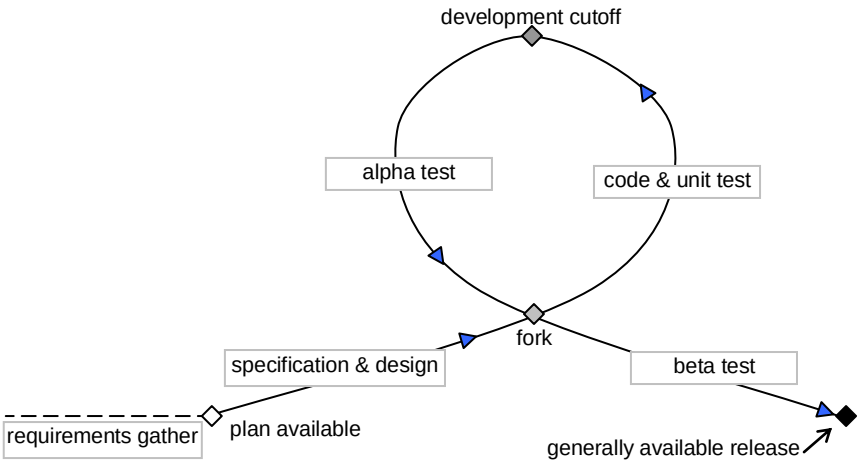
The lifeblood of the software company is the flow of new features in their software products going out the door. If the rhythm of these new features falters, it jeopardizes the health of the company.

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.

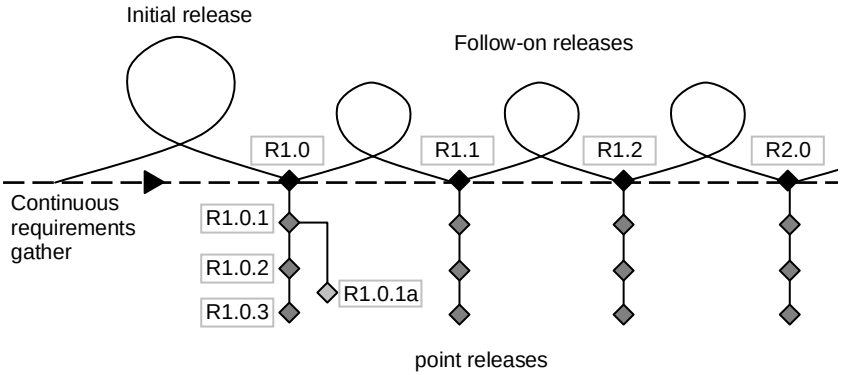
When design and specification has produced enough output, development is ready to begin ***coding and unit testing***. For traditional release lifecycles this point is called "fork" because the source code is logically forked into the release currently being supported in the field, and the new release 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.

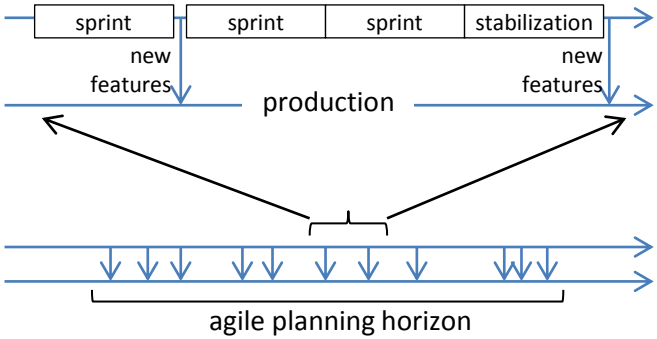
Each feature release will require ongoing maintenance, meaning that customers who take a release will have made available to them periodic point releases that fix any problems they may encounter. These *point* releases are pure corrective maintenance, and contain no new features.

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.

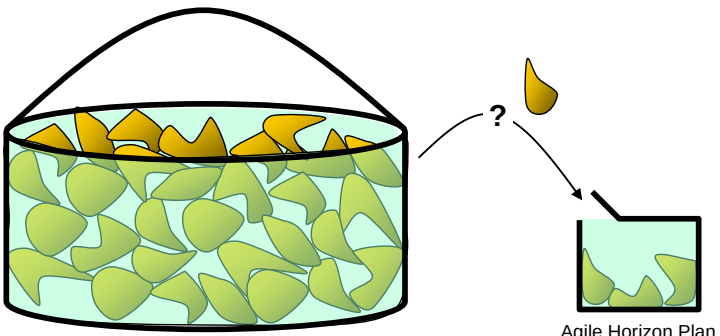
We view the agile horizon plan as an abstracted version of the implementation plan. It comes to the same conclusions, but is stated and manipulated in summary form. It should always be the case that if we can put together a valid agile horizon plan (one that appropriately balances capacity and requirement), 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.

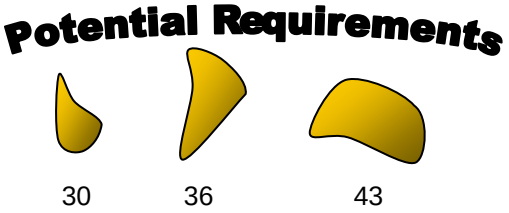
Coming out of the first release effort is a set of features that we put off together with some significant areas where we know we could improve the architecture. These form the basis for a wish list of potential requirements for the next planning cycle. When users start applying the software to practical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

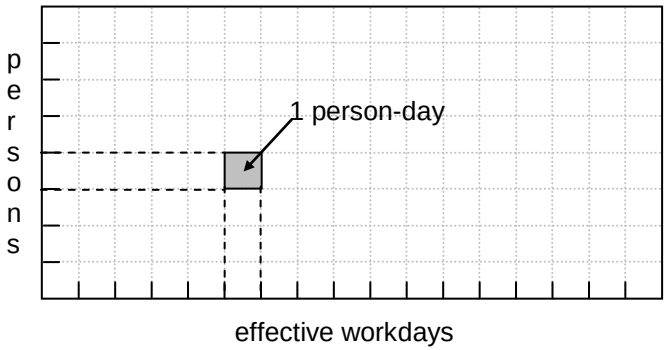
For each developer we then estimate a *work factor*. This factor converts 8-hour workdays into dedicated, uninterrupted hours available to work on putting new features into the next release. This is typically on the order of about 0.6 or so, meaning that for each workday in the release, they can on average 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

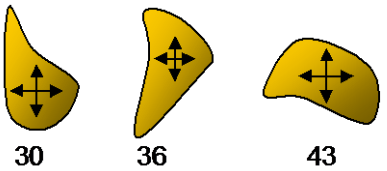
Once we have an estimate for the average number of dedicated developers available during a given planning horizon, the planned capacity we have for coding features is that number multiplied by the effective number of workdays dedicated to coding during the planning horizon.



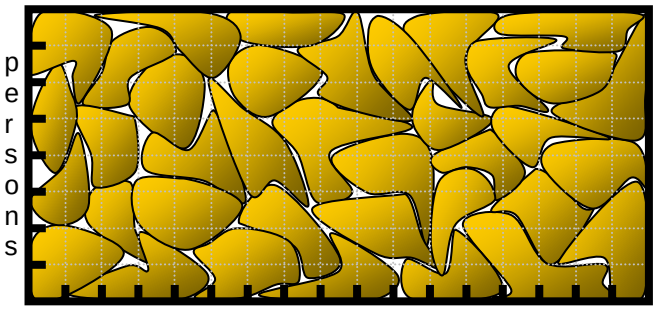
For a traditional release cycle, that is the number of days in the coding phase. For a more continuous SaaS-based release strategy, that is a pre-estimated fraction of the total workdays in the planning horizon. In other words, one can either concentrate the coding in a phase, or spread it out, but either 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.

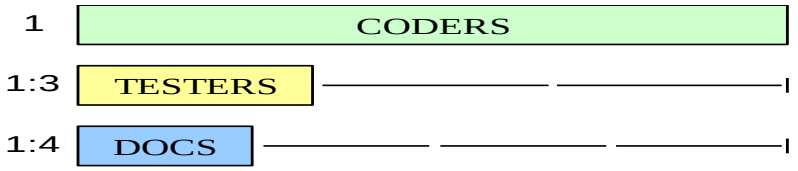
By consequence, when planning it is worthwhile for us to devote the majority of our attention to the coding activities of the project (which, for future reference, such term shall always include the development of automated tests as well). For this reason, we size explicitly only the coding activities 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.

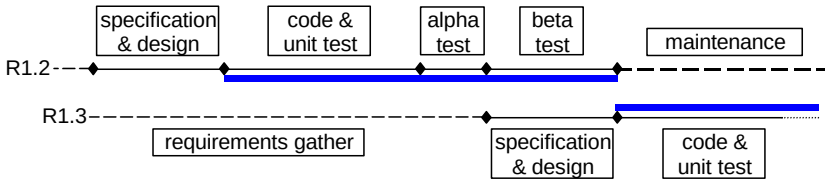
Nonetheless, the agile horizon plan must take into account the amount of time and number of people available for non-coding activities. To account for this time and these resources, we require that they be in certain pre-determined ratios to the amount of coding time and the number of coding resources.



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

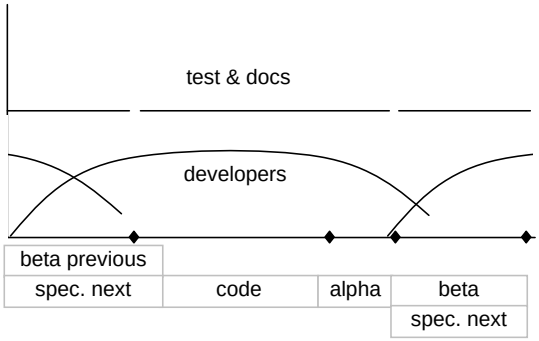
overlap releases, starting specification and design on the next release as the previous one goes into beta test.



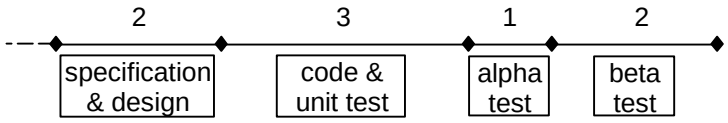
This same overlapping of releases also serves to smooth the use of coding resource. The same can be said on a feature by feature basis in the case of a SaaS lifecycle.

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.

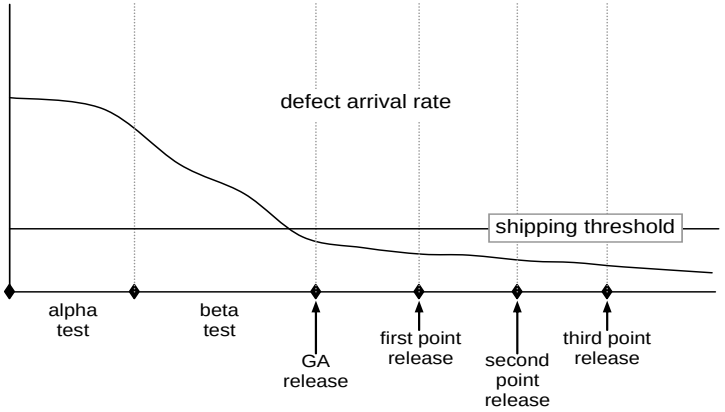
In the case of a more continuous release method, we use an overall ratio of coding days to other days, typically in the range of 3:1 or 3:2 or so. These "other days" are considered overhead days that account for release preparation, stabilization sprints, delayed start sprints due to lagging specification 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.

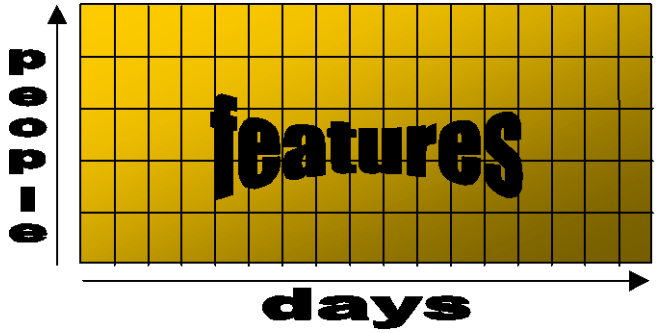
4.1. A Geometric Analogy

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.

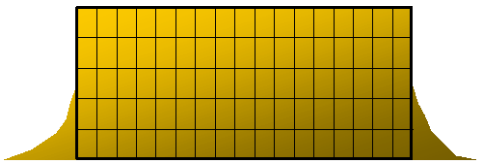
In practice, the dimensions of planning are the base and the area: time and features. The height, the number of developers available, is not usually something we can adjust easily. It is a given, and tends only to

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.

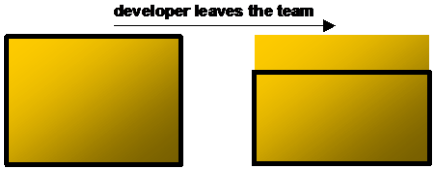
Increasing the strength of the development team operates on a longer-term planning horizon than the next release. We must address these sorts of issues in the software development department's annual business plan, which we will cover in Chapter 16. "Business Planning", on page 323.

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.

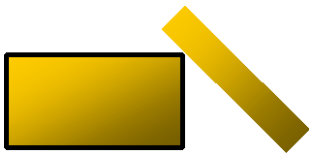


This plan violates the capacity constraint. When this happens, we must take action. We must either increase the time to accommodate the desired features, or cut features from the plan.

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

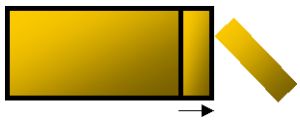


Assuming it is impossible to replace these developers in the short term, we have three options. The first option is to cut features from the plan, holding firm to our dates.

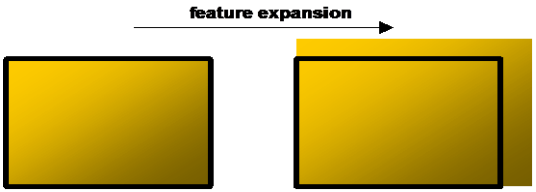


The second option is to hold firm to our feature set, and move the dates out.

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

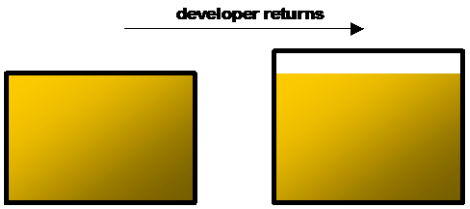


The following diagram represents another mishap. This could represent either additional features coming into plan, or a re-estimation upwards of the amount of effort we require to get the remaining features implemented.



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.

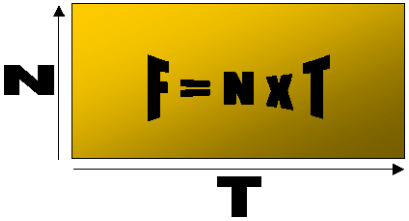
While it may seem excessively detailed, experience has shown that a rigorous definition of the numerical quantities involved is necessary to apply agile horizon planning in practice. Only with precise definitions can we determine the validity of an agile horizon plan. Without these precise definitions, 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.

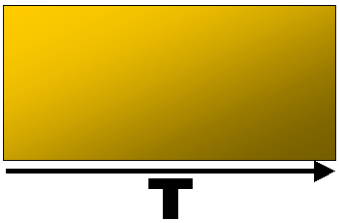
This does not happen by coincidence, it happens by definition. The capacity constraint is a true *constraint*. It is not a *suggestion* that we ought to balance the capacity and the requirement. It is a law of nature that constrains us to being able to put into a plan only as much effort on features 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.

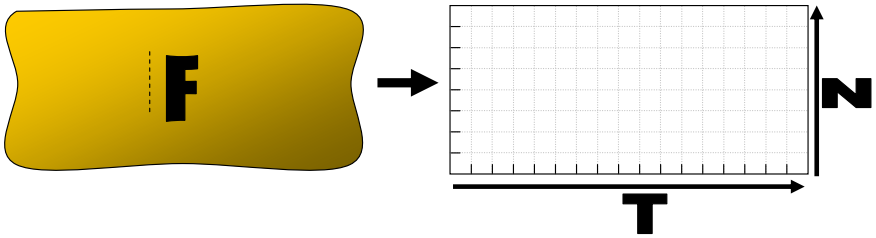
Say there are 3 hours left. With those scant 3 hours, if the phone keeps ringing, if people keep dropping in on us, if those 3 hours gets spread willy-nilly across the 8-hour day, then we will get considerably less development done than if we can closet ourselves away for 3 solid hours with no disruptions.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

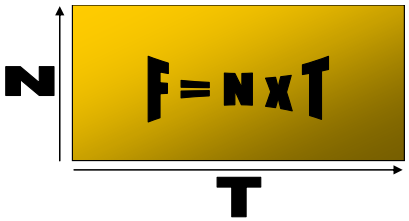
The effort estimate therefore combines three independent variables:

- 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.

6. The Stochastic Capacity Constraint

In the previous chapter, we gave precise, *post-facto* definitions for the quantities that make up the capacity constraint. While this is useful for measuring these quantities, the essence of planning is estimating them in advance.

When we give an estimate, we never give a number and say we are 100% certain that the number is accurate. For example, if we estimate the size of a feature to be 20 dedicated person-days, we are not saying that we are 100% sure it will actually take 20 person-days. Our best guess is that it will take this long, but it may take more or less time in practice.

What is this 20 person-days, though? Is it an estimate we are confident in? Is it a pessimistic estimate? Is it an estimate that is optimistic? Is it an average sort of estimate? Sorting this out is very important to planning.

A quantity whose value is uncertain, that depends on chance, is called a *stochastic variable*. Statistics provides the mathematical language for dealing with stochastic variables. In this chapter we will introduce statistics, discuss how to apply statistics to estimation and the planning problem, and look at some of the organizational issues when dealing with uncertainty in the planning context.

6.1. Confidence Intervals

At the heart of statistics is the concept of a *confidence interval*. To illustrate confidence intervals, we will use the familiar notion of flipping a coin. Of course, we will be flipping it 5000 times, so maybe it is not very familiar to you!

When we flip a fair coin 5000 times, we would expect it to come up heads about 50% of the time, or about 2500 times. We know that it will not be exactly 2500, just more or less.

However, what is the probability that it will be exactly 2500? The answer is that it is not very likely. To be precise, there is only a 1.1% chance that it will come up heads exactly 2500 times (not one more, not one less).

What is the chance that it will come up heads something less than 2500 times? This is a lot more likely. In fact, the answer is 50%. If we repeat this experiment repeatedly, then, on average, half the time we'll get less than 2500 heads, and half the time we'll get more.

What is the probability that it will come up heads something less than 2530? This is a higher still at 80%. There is an 80% chance that the number of heads will be fewer than 2530. There is a 92% chance that it will be fewer than 2550.

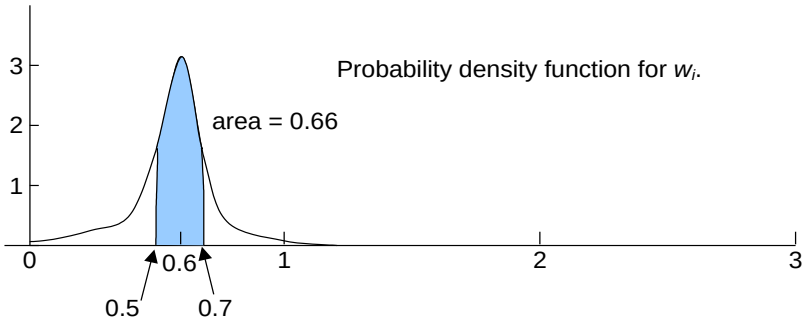
We call these numbers, the 50%, 80%, and 92%, *confidence intervals*. With 80% confidence, we can say that the number of heads will be less than 2530.

6.2. Stochastic Variables

This above coin toss example illustrates the notion of a confidence interval. Confidence intervals are key to describing the more general notion of an estimate.

As an example, consider estimating the work factor, w_i, for a given developer. When estimating, w_i is considered to be a *stochastic variable*: a thing that is described by a statistical distribution. For any range of values for w_i, the statistical distribution gives the probability that the actual w_i will be in that range.

One way of describing a statistical distribution is by giving its *probability density function*. This is a curve where the x-axis covers all possible values for the stochastic variable, the y-axis are numbers bigger than 0, and the total area under the curve is precisely 1. The probability that the stochastic variable lies between two x-axis values, *a* and *b*, is the area under the curve from *a* to *b*.



We show here one possible probability density function for w_i. In this example, the probability of w_i lying between 0.5 and 0.7 is 66%. This looks to be a realistic curve for w_i. For example, the probability of

w_i being more than 1 is close to 0, and there is a finite probability it can be as low as 0.

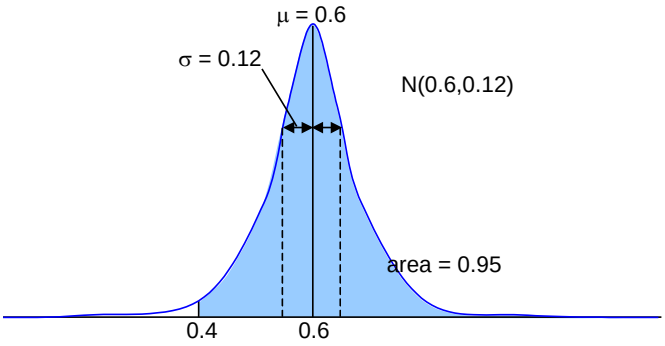
The fully accurate way of giving an estimate is to supply such a curve. Based on the shape of the curve, we know with great precision everything there is to know about the estimate.

Getting someone to draw us a curve is not the usual way to ask them for an estimate! For one thing, nobody knows the actual distribution in such detail. Usually we make an assumption about the overall shape of the curve, choosing a shape that we can describe by a mathematical formula. We then ask for a few parameters that we can use to fit the curve. Because estimates are somewhat inaccurate, the relative error introduced by choosing from one of a set of mathematically tractable probability density functions is small.

For example, we may assume that all *w's* are adequately described by a bell-shaped Normal distribution. We can then ask our estimators to give us average case and worst case values for *w*.

For average case we can be more precise by asking the estimator to pick a value that half the time will be higher and half the time lower that the actual value. Say they pick 0.6 for this value.

For "worst case" we could ask for a value for which 95% of the time *w* won't be that bad. Say they pick 0.4. Given these two numbers, we have enough information to precisely fit a Normal curve. If we do the math, the Normal curve is the one with mean $\mu = 0.6$ and standard deviation $\sigma = 0.12$, also called *N(0.6,0.12)*, and shown on the page following.



The standard deviation, σ, for a Normal distribution with mean μ is defined such that there is an approximately 68% chance of the actual value lying in the range $\mu \pm \sigma$. By convention, we give Normal distributions using the mean and this 68% confidence interval. There is no reason why we cannot give them using any two confidence intervals we choose.

A Normal curve, while mathematically the easiest to deal with, may not be the best fit. For example, because a Normal curve is symmetric about the mean, the distribution above predicts a best case of 0.8 (that is, that 5% of the time the actual value of *w* will be greater than 0.8). If in reality the estimator believes the 95% best case value is something more like 1.0 we will need a skewed (tilted) distribution to represent *w*, and would have to move to something more complex than a Normal distribution.

As well, Normal distributions never end. That is, there is a non-zero probability that *w* could be anything at all. This is not appropriate, as there is no chance of *w* being less than 0 or larger than 4.2. However, given the small probabilities of these events, it may be something we are willing to live with as a source of error.

6.3. Estimates

From all this we see that an estimate is not such a simple thing. Say we ask a developer to estimate the effort required to implement a certain feature and get an answer of one week. Is that one week of just one person, or are more involved? Does he mean $5 \times 8 = 40$ hours of work, or is he assuming longer days and weekend work? Is he assuming those hours are dedicated only to putting this feature into the release, or are they just regular work hours?

We address questions such as these with the rigorous *post-facto* definitions we gave in the previous chapter.

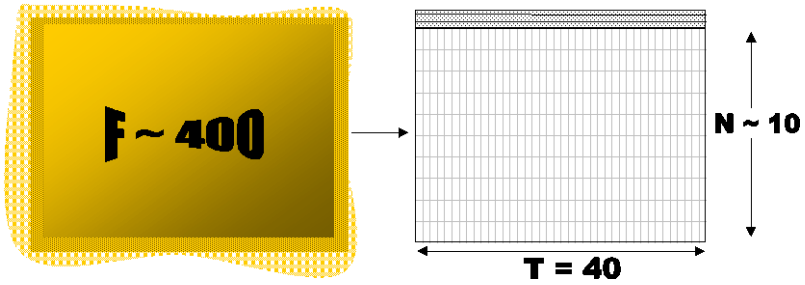
Once we have answered these questions, we should then ask at what confidence interval is he giving the answer. Is that a 50% confidence interval? That is, is he assuming he will be under half the time and over half the time? Is it an optimistic, best-case sort answer? Something like about 20% of the time he would expect to come in under that time and 80% over. Is it a conservative estimate? That is, would he expect to come in within that time 19 times out of 20?

We address these questions by making certain that we use proper estimates, as described in this chapter, and not just a single-number "estimate".

Depending on the answer to all these questions, that one week estimate could be anything from two days to one month at a 90% confidence interval! It is very important to clearly establish this sort of thing up-front when doing estimates. The differences could be large.

6.4. The Capacity Constraint

The capacity constraint is uncertainty embodied. *T* is fixed, but both *N* and *F* are uncertain.



N and *T* define the shape and size of the rectangle. *F* is the "goo" we use to fill up the rectangle.

We can imagine that the height of the rectangle, *N*, vibrates up and down, so we are never sure how much we can fit into it from moment to moment. At the same time, the goo is vibrating: we are never quite sure how big it is. There is uncertainty about the size of the rectangle, and there is uncertainty about the amount of goo we have.

In other words, there is uncertainty in our capacity estimates, and there is uncertainty in our feature size estimates.

Under these circumstances, all we can speak about is the chance of the goo fitting into the rectangle: the chance of getting all the planned features done by the planned date.

For example, say we come up with a set of features and estimate that the total effort required, *F*, will be 400 dedicated person-days. Say we

then estimate that our resourcing, N, will be 10 dedicated developer equivalents per day. If we have 40 working days ($T = 40$) to get the release done, one would think we are fine, as we have 400 days of capacity and 400 days of requirement.

However, we may not be fine. Based on what we have said so far, we do not have enough information to answer the question "with what probability will we come in within the T days". We will need to know the precise distributions for F and N to get any farther.

6.5. Summing Distributions

Both F and N are sums and averages over many contributing largely independent stochastic variables. In statistics, we cannot just add things up as we can in simple math.

For instance, say our release is comprised of only two features whose sizings are f_1 and f_2, respectively. Each of these sizings is an estimate and hence has an associated statistical distribution. The question is, what is the distribution of $F = f_1 + f_2$?

This is a difficult statistics question in the general case. If f_1 and f_2 are Normally distributed, then we know that F will be Normally distributed as well. The mean of F will be the sum of the means of f_1 and f_2. The standard deviation of F will be the square root of the sums of the squares of the standard deviations of f_1 and f_2. This is difficult enough. If f_1 and f_2 are not Normally distributed, it gets more difficult still.

Suffice to say that we need statistical simulation software tools to come up with the distributions of F and N given estimates of the contributing factors.

Without the use of such software, we can still take into account estimates by using a quick but conservative rule of thumb.

No matter the distributions of the constituents, we know that the mean of the result is the sum of the means of the constituents. In addition, we know that in practice the sum of the *p*% worst case confidence intervals of the constituents provides a worst case bound for the *p*% worst case confidence interval of the result.

Using these facts, we can solicit mean and 90% worst case estimates for the contributing stochastic factors, and compute the results (non-statistically) using both of these sets of numbers. These provide mean and 90% worst case estimates for the resulting distribution we wish to calculate. If we assume the result is Normally distributed, we can use these numbers to fit a pessimistic Normal curve for the resulting distribution. From this curve, we can compute confidence intervals for our resulting distribution.

This is overly pessimistic, but is easy to apply in practice.

6.6. The Delta Statistic

Now that we understand how to get distributions for *F* and *N*, we can begin considering the planning problem: how best to choose the feature set and the dates.

As a preliminary, we shall define a new quantity *D(T)*, for *delta*. *D(T)* is a stochastic variable whose distribution depends on *T*. We define it as follows,

$$D(T) = N \times T - F$$

We can determine this distribution if we know the distributions for *N* and *F*.

From an *a priori* estimation standpoint, we are interested in the probability that $D(T) \geq 0$: the probability that all the features will be completed on or ahead of time. For small *T*, that would be a small chance. For big *T*, that would be a high chance. In settling on a *T* for our plan, we will want to first choose a confidence level, say 80%, and then pick a *T* such that *D(T)* is just positive with that degree of confidence.

Let us continue our previous example to illustrate this. Assume *F* and *N* are both Normally distributed with means of 400 and 10 and with 90% worst case values of 500 and 8, respectively. The table following shows the value of *D(T)* for various *T's* at the indicated confidence level.

	confidence level						
	25%	40%	50%	60%	80%	90%	95%
30	**-39**	**-77**	**-100**	**-123**	**-177**	**-217**	**-250**
35	**14**	**-26**	**-50**	**-74**	**-130**	**-172**	**-207**
40	**67**	**25**	**0**	**-25**	**-84**	**-128**	**-164**
45	**121**	**77**	**50**	**23**	**-38**	**-85**	**-123**
50	**174**	**128**	**100**	**72**	**7**	**-41**	**-82**
55	**228**	**179**	**150**	**121**	**52**	**1**	**-41**
60	**282**	**231**	**200**	**169**	**97**	**44**	**0**

Of special interest are the points where the columns transition from negative to positive values. For instance, looking at the bottom of the last column we see that to be 95% certain of hitting the dates, we

should plan for the release to take 60 days. Another way of saying this is that if we plan for T to be 40, then only 5% of the time will we be late by more than 20 working days.

When we first used the numbers 400 and 10 for F and N, we blithely assumed T would be just 40. In fact, if we wish to be 95% certain of hitting our dates, we should make $T = 60$. This is a large difference.

To be a bit less conservative, we can look at the 80% column and say that to be 80% sure of hitting the dates, T should be about 49 days (we interpolate for T between the two vertically adjacent cells where the value changes from negative to positive).

If we are a gambler, we can look at the 25% column and set T to about 33 to have a 25% fighting chance of hitting the dates.

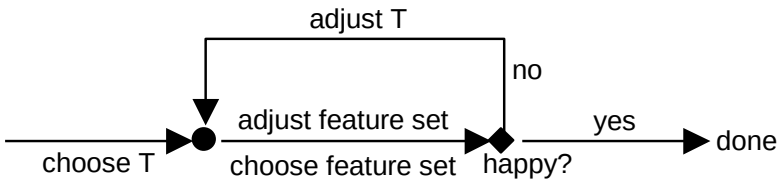
6.7. The Initial Planning of the Release

In practice, in coming up with a plan we start by picking a reasonable date for the release. For a traditional, shrink-wrapped software product, for instance, that might be somewhere between 6 and 9 months from the previous release date. This will imply a T, the number of workdays in the coding phase. To be concrete, let us use $T = 70$ days as an example.

According to our risk preferences, we pick a confidence interval, say 80%, and choose a set of features such that $D(T=70) \geq 0$, 80% of the time. This defines the initial set of features and the initial dates for the release. There is an 80% chance of releasing those features by those dates.

To hone in on the plan, we will iterate. If the features we had our heart set on did not quite fit, we would move the dates out a bit and

choose a broader feature set. If everything fits comfortably, then we might move the dates in a bit.



If we were to look at the problem of optimal agile horizon planning mathematically, we would discover it to be extraordinarily complex. We would need to consider the benefit associated with promising various features sets on various dates, and actually delivering different features sets on different dates. Moreover, it is a dynamic problem as we can adjust the agile horizon plan as we go, making optimal decisions as we proceed through the agile horizon plan. Suffice to say that a mathematically optimal solution will always evade us, and so we must use intuition, experience, and good judgment for agile horizon planning.

6.8. Adjusting the Agile Horizon Plan

The initial agile horizon plan is a best guess at the features we can get into the next release, by the given date, with a certain contingency as captured by the confidence interval we are using.

As development of the release unfolds, it may be that we will need to add extra features into the plan, or (less frequently) that certain features may no longer be required. It is also likely that we will adjust our estimates for how long various features would take, most often

upwards (human nature being optimistic in the case of feature estimation).

To deal with such situations, it is good to have flexibility built into the plan.

One way of achieving this flexibility is to start with a pessimistically high confidence interval (*e.g.,* 95%). However, this has the problem that we would be very likely to be finished ahead of schedule. If we finish ahead of schedule, we can either release early or add some extra features into the release.

Usually there is not much upside to releasing ahead of schedule. People will generally be upset that the plan was "overly padded". Even if it was a fixed price contract, the customer will be upset at being overcharged.

The alternative is to then add extra features into the release. Unfortunately, if the agile horizon plan achieved its slack by using a high confidence interval, there will have been no planning or up-front work done for those new features.

To avoid this situation, it is usually wise to have two confidence intervals in mind, and two feature sets.

We divide the feature set into an "A-list" and a "B-list". The B-list are nice to have features that we can easily drop from the release should we find ourselves falling behind. The complement to this is the A-list of features without which the release is no longer meaningful.

We should have a high degree of confidence that we can get the "A-list" done on time. We should have a low degree of confidence that we can do the entire B-list on time as well. For example, we might plan to

a 95% confidence interval for the A-list, and a 40% confidence interval for the full list.

To use this technique, it is necessary that no work on "B-list" features occur before "A-list" features.

In practice, having a good-sized "B" list and being willing to use it is the best way of hitting our dates. It is remarkable how a company can hit the dates by dropping half the features and still have everyone calling the release a success. On the other hand, if a company misses its dates by even a few weeks, the release will be classified a failure, even if all the features originally planned for were included.

6.9. Advanced Planning I

While the basic planning method given in the previous section is the one that will be used in practice, it is instructive to consider other, more quantitative methods of planning. This will enable us to better understand what we are trying to achieve in the seat-of-our-pants style.

In this section we'll consider in more quantitative detail how to choose an optimal *T*. In the next section we'll look at the even more difficult problem of choosing an optimal feature set.

When we have a certain feature set in mind, choosing a date for the release is a tug-of-war between announcing its availability for an earlier date on the one hand, and possibly missing those dates on the other hand.

This can be quantified using a benefit function, *B(T,e)*. This is the benefit (positive or negative) associated with announcing a certain

feature set for time *T*, but actually delivering it at time *T+e* (*e* for *error*).

In theory, to estimate *B* we would start by assigning a value to *B(T,0)* for each *T* (it is understood that we have already settled on a given feature set). This is the benefit of announcing and then actually getting the release out on time, for various times. In general, the closer in is *T*, the higher is the value of this function (within reason). As *T* gets out farther and farther, the benefit goes down.

$$B(T,0) \geq B(T+e,0) \text{ for } e \geq 0$$

For each of the *T*'s, we then need to assess the effects of delivering either earlier or later than we said. That is, of taking into account non-zero values for *e*.

Earlier is always either the same benefit or more (if it was less, then we could just wait and deliver on time).

$$B(T,e) \geq B(T,0) \text{ for } e \leq 0$$

Generally, promising it earlier and delivering on time is better than promising later and delivering it at the same time.

$$B(T+e,0) \geq B(T,e) \text{ for } e \leq 0.$$

These two considerations imply the following bounds on *B(T,e)* for negative *e* (being early).

$$B(T,0) \leq B(T,e) \leq B(T+e,0) \text{ for } e \leq 0.$$

For positive *e* (being late), we have similar considerations. For an ethical software company, being late is always worse than promising late, but being on time.

$$B(T,e) \leq B(T+e,0) \text{ for } e \geq 0.$$

Unfortunately, not all software companies are ethical, and will perceive a benefit in promising something for earlier than they can possibly deliver it. In these cases, mostly the same benefit can be had by announcing a later date (or no date at all) earlier on, which is ethical, rather than announcing availability for an impossibly early date (which is not – but at what confidence interval does it become so?). This implies an upper bound on *B(T,e)* for positive *e*. Ominously, there is no lower bound!

Using these bounds as guidelines, it is theoretically possible to fill in a guess at *B(T,e)* for many possible values of *T* and *e*. Once we have B, it is then possible to solve for *T*. *T* is the value that maximizes the expected benefit over the possible range of values for *e*.

Before we can give the equation, we need to define the function *P(t)*. This is the probability that *D(t)* = 0, or in words, the probability that we can deliver the feature set at time *t*. We can get this from the distributions for *F* and *N*. The equation for *T* is then,

$$T = \max_t \left[\sum_{e=-t}^{\infty} P(t+e) \cdot B(t,e) \right]$$

In the square brackets is the expected benefit of announcing at time *t* but delivering at a wide range of possible times. *B(t,e)* is how well we can expect to do promising at *t* but delivering at time *t+e*. *P(t+e)* is the probability of actually delivering at time *t+e*. The sum of the product of the probability and the benefit over all possible values for *e* is the net expected benefit.

We should choose a *T* that maximizes this net expected benefit.

While all of this is fine in theory, one would not advocate taking such a quantitative course in practice. For one thing, this is an extreme simplification of the planning problem and ignores some of the best planning options that are available (as discussed in the next section). As well, as we proceed through an agile horizon plan everything changes as we go, and we would have to continuously re-estimate *B* and *P*.

In practice, and stripped of the mathematics, the lesson is that we should always be aware of the risks of coming in late. We should choose a date and arrange our commitments in such a way that we won't be unduly punished if we miss our dates. This may mean moving the date a bit farther out, to both reduce the probability of being late, and the harm if this does happen.

6.10. Advanced Planning II

In the previous section, we assumed that the feature set was chosen, and we were trying to decide on an optimal *T*. Even if we were running late, we would stick to our guns and deliver the original feature set. This is unrealistic. In practice, we have a wonderful planning tool available in our choice of feature set and the related choice of what features to drop if we start getting into trouble.

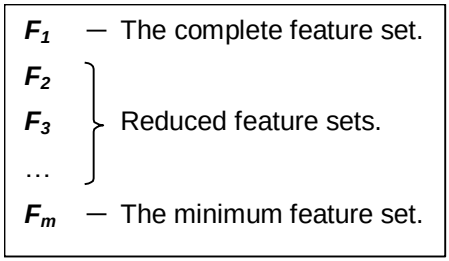
To quantify this, the benefit function should be re-stated as follows. Let

$$B(F_1, T, F_i, e)$$

be the benefit associated with announcing feature set F_1 for time *T*, but actually delivering feature set F_i at time $T+e$. This formulation of *B* raises the possibility of trading out features for time as the release cycle proceeds, and is considerably more realistic.

In practice, the dimensionality of this new *B* is too high to be useful in any practical sense. It does, however, capture the essential planning variables: features and time.

To work with this new formulation of *B*, we identify a sequence of feature sets from F_1, the complete feature set, to F_m, the minimum feature set we would still be willing to ship. Each succeeding feature set would lop off more features (or reduce the scope of features).



The assumption is that if we start getting behind, we would make some compromise between moving the dates out a bit, and reducing the feature set a bit.

The general planning problem is to pick an initial feature set, F_1, and an initial date, *T*, to maximize our expected benefit given that we will make optimal choices down the line in trading off dropped features for late dates.

We won't be giving any equations for this, as it is a very difficult thing to quantify mathematically. This difficulty goes a long way towards explaining why planning remains very much an intuitive, seat-of-the-pants effort.

Experience dictates that it is unwise to stuff the release full to the brim with "must-have" features. We should always remain cognizant of an "A" list and a "B" list of features. The "A" list comprises the feature set in F_m: those features that we simply cannot drop and still ship a meaningful release. The "B" list comprises all those features that bring F_m up to F_1: those features we would be willing to drop if necessary.

A good agile horizon plan will have a balance of "A" list and "B" list features, and will balance the relative size of the "B" list with the tightness of the dates.

6.11. Appreciating Uncertainty

In order to plan well, one must have a good gut-feel for probability and statistics. Unfortunately, not many people *really* appreciate probability and statistics all that well.

Two types of people who do have this understanding are successful financial traders and successful gamblers. Gamblers have it a bit easier than traders. Gamblers at least know the probabilities going in. Traders have to first guess at the probabilities, and then act accordingly.

Both groups will tell you that the most important lesson to learn is not how to make money, but how not to lose it. That is, knowing when to cut your losses.

By plying their trade day-in and day-out, they develop a feel for probabilities. They know they are almost as likely to lose as to win. They do not distort the odds based on false optimism.

So, what does all this have to do with agile horizon planning? Every plan is a gamble. In most types of plans, we never see any elements of

uncertainty. Yet, they abound. Somehow, it seems shameful for us software professionals to have to admit that we do not know something with certainty.

When plans start slipping the first thing we do is blame the estimates. We say that an estimate is "bad". Actually, that is not even a meaningful statement. We give honest estimates with the understanding that almost anything can happen. Each estimate is its own little statistical experiment that we will never repeat again. There is a finite probability that almost anything can happen.

To be fair, if every estimate in the plan is off we can do some statistics and see that the chance of them all being so far gone is practically nil. If this is the case then we can say that the developers doing the estimation are *almost certainly* not very good at it.

Nonetheless, we should probably not be as quick as we usually are to blame the estimate and point fingers at the developer who gave it. Bad luck happens, on average, half the time. Moreover, we take a lot more notice of the bad luck than of the things that go right.

Where the gambler comes in is in our reaction when bad luck does hit us.

When a feature is slipping, there is a very human tendency not to "take our losses". It is a lot like a failing gambler. When things start slipping, hopeless optimism sets in. After all, there is always a chance that the plan will recover. True enough. However, is it a good bet? By taking the optimistic approach we do not have to admit to our losses right away. We do not have to admit, "We were wrong" just yet. There is still a chance that things can turn.

The problem is, most people involved in agile horizon planning do not have the finely honed skills of the professional gambler or trader. They do not really understand at a gut level the implications of, for instance, a 25% chance. They will often err on the side of optimism. The good gambler, on the other hand, knows when to cut his losses, announce that the software will be late, and deal with the business consequences immediately. More often than not, holding onto a slim chance will drive us deeper and deeper into the hole, like the compulsive gambler who winds up losing his life's saving on the one chance that he can recoup his losses and come out a hero.

On the other hand, when the business is at risk of failing entirely, the 25% chance may be the best bet around. Again, the experienced gambler will know which situation applies, and act accordingly.

Aggressive plans are another example of statistics at work. We all know of projects where nobody in development thought the thing could reasonably come in on time, and yet the business presses on regardless with the doomed deadline. It is not necessarily true that anybody is acting irrationally, just optimistically.

Ask the developers the chances of the project coming in on time and they will say, "Poor: 60% at most". Ask the entrepreneurial CEO the same thing and he will say, "Looks good" (thinking to himself that there is least a 60% chance).

This sort of divergent thinking is responsible for a lot of tension between the business side of a software organization and the development department.

Entrepreneurs, by their nature, are risk-takers. Most software developers, on the other hand, are risk averse. When an entrepreneurial

CEO insists on a date and a feature set, and the developers cannot see how it can possibly come in, what might be happening is that different risk tolerances are coming into play.

The entrepreneurial CEO enjoys risk and is eager to take it on. The software developers dislike risk, and are eager to avoid it; they may even be de-motivated by it. If both sides could explicitly agree on the risk they are taking, they could largely eliminate this tension.

The best practical advice is to state the chances explicitly and discuss them in advance. If the company decides to take a chance, it should not start blaming the estimates when things go bad. Rather, management should know when to take their losses and re-plan. Remember, even when planning to hit the dates 80% of the time, we should still expect to miss the dates or drop features 20% of the time.

6.12. Loading the Dice

Up to now, we have been treating the capacity constraint as purely stochastic, like rolling a pair of dice. There is actually a key difference between these two things. Unlike for a pair of dice, the actions we take subsequent to planning a release have a major effect on the outcome.

Up front, we treat as stochastic variables things like our ability to retain staff, the work factors, the number of raw developer-days we have, and so on. We also treat as stochastic our feature sizings. During the release cycle, however, we can actively manage these variables, thereby loading the dice in our favor.

It is analogous to the odds on a football game. The odds of a certain team winning may be 3-to-1 against. Everybody accepts, however, that an exceptional effort by that team can result in a win. It is just that in coming up with the odds most people are not expecting that exceptional effort.

It is the same for planning software. When we give the odds, we are basing them on history and giving a realistic assessment of our future chances given our past efforts. This is entirely proper and prudent. However, when we take off our planning hats and put on our management and development hats, we should be expecting much better from ourselves than we have ever been able to achieve in the past. This is the right attitude going into a new horizon.

By exceeding expectations and delivering more than anybody expected, we have added a new historical data point. Future estimates will consider this new-found productivity.

There is a danger in confusing prudent estimation and optimistic attitude. A common pitfall is to make estimates based on *expected* productivity increases. This will usually lead to disaster.

For instance, say a company has just instituted a new tool of some sort and expect to cut testing time in half. Should the company consider this in their planning? It would be unwise to do so. The company should base its plans on what it has achieved historically. If this new tool pans out in practice, they can adjust their estimates downwards on the next planning cycle.

We should always be aggressively seeking productivity increases, and we should be highly confident of achieving them based on our course of action. These expected productivity increases should not,

however, be taken into account in planning. Planning should use what we have proven in the past. There are simply too many things that go wrong to prudently count on any sort of productivity improvement during the planning.

Nonetheless, the goal in execution is to always do better than the plan would indicate. The best guide in achieving this is in looking at the planning itself, and concentrating our efforts where they will do the most good.

For example, if the work factors appear to be particularly low, we should take measures to remove distractions from the developers. If the staff retention rate seems low, we should take measures to ensure our developers do not quit on us.

In a pinch, if the number of vacation days during the release cycle seems high, we can ask developers to put off their vacations until after the release cycle is done. We can even attempt to increase the work factor by asking for weekend and late night work. In general, these sorts of measures are counterproductive in the end, but they may help us out of a tight spot.

Even on the feature sizing side of things, there are measures we can take. A clever design can reduce the sizing considerably over that of a more mundane design. In one instance, the chief architect came up with a brilliant software design that had two features use one common implementation. During planning, everybody thought these features were entirely independent of one another. The net result of this clever action was therefore to cut 20 days out from the feature sizings.

Recall also that developer productivity, the efficiency with which one dedicated hour is used, is included in feature sizings. By helping less experienced developers at key moments, it may be possible to increase their net productivity drastically. A team approach to implementing the features, where each developer helps where they are best able, can increase net productivity and therefore act to reduce total feature sizings.

Even though when planning we will count on only what has been achieved historically and base estimates on that, it is nevertheless also necessary to work hard to ensure that we take every measure possible to improve our chances as we go.

It is a subtle thing to only count on what has been achieved historically with one part of our brain, the planning part; while simultaneously expecting to improve things with another part of our brain, the getting it done part.

To confuse the two, however, is to set ourselves up for failure.

6.13. Summary

To use the capacity constraint in practice, we need to understand it not only from a rigorous, post-facto definitional framework useful when collecting statistics, but also from an *a priori*, estimation standpoint useful for planning a release ahead of time.

We began with a generic discussion of stochastic variables and estimates. We saw how these basic statistics concepts applied to the capacity constraint, where T (number of days) is fixed, but F (feature sizings) and N (average developers) are uncertain.

In applying statistics to the capacity constraint, we discussed the problem of combing fine-grained stochastic variables into distributions for *F* and *N*, and ultimately how to arrive at a distribution for the *delta* statistic, $D(T) = N \times T - F$, whose confidence intervals tell us our chances of hitting a planned date as determined by *T*.

Using the delta statistic, we discussed the problem of planning software releases. We described how it is useful to plan such that we have a high degree of confidence in delivering an "A-list" of features by the planned date, but a low degree of confidence in delivering a "B-list" in addition. If the release begins slipping, we can sacrifice the "B-list" to maintain a firm delivery date.

We closed by discussing various issues regarding the management in the presence of uncertainty. In particular, we discussed the idea, well-known to successful financial traders and gamblers, of "taking our losses" as opposed to hanging-on and hoping for a recovery. We also discussed the difficulties of simultaneously planning for average case productivity while managing in such a way to achieve better productivity. By managing in this manner, we effectively "load the dice" in our favor.

7. Software Releases

For agile horizon planning to have meaning, a software company must be committed to the notion of distinct, well-spaced planning horizons of their software. Without this commitment, the agile horizon planning process has no opportunity to become established, and its benefits are lost.

In this chapter we will discuss obstacles to establishing such a culture, and what can be done to overcome them. We will discuss the tradeoffs between having well-spaced planning horizons on the one hand, and being responsive to customers' needs on the other. By the end of the chapter, the reader should appreciate the need for horizon plans, and understand how to mitigate the negative effects this can have on responsiveness to customers' needs.

Implementing a sound horizon planning process is a culture shock for the young software company. The entrepreneurial startup considers itself agile and fast moving, responsive to the changing business situation and the needs of its customers. This culture leads to the situation where the company will release an unplanned continuous sequence of changes to its software. The company identifies these "dribbling releases" by the hour of the day they built the software. The company will typically be incapable of re-creating a previously-shipped build. Indeed, the company sees the idea of going back to a previous build as a step backwards.

While this dribbling release approach will work to a certain extent, as the company grows, continuing these practices will result in

inconsistent quality, the inability to deliver on promises, and ultimately dissatisfied customers. Solid release methodology will go a long way towards clearing up these quality and control problems.

Thus, the first commitment a company must make is to abandon the idea of dribbling releases, and adopt the idea of planned and disciplined feature release to the field. Without this commitment, the software company cannot improve its practices.

For the bulk of this chapter, we will discuss larger feature releases, where the cost of shipping a release is relatively high (shrink-wrapped and enterprise software, for example). Towards the end of the chapter we will discuss more continuous release methods applicable to SaaS.

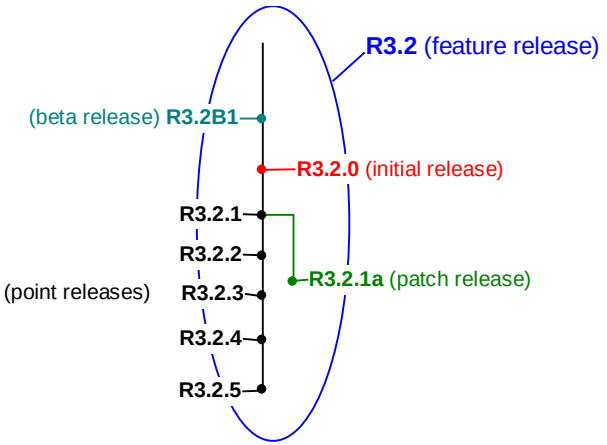
7.1. Concepts & Terminology

Let us start our discussion of releases by defining some basic concepts and terminology.

The first definition is that of a *feature release*, or just *release* of a software product. A release is an abstract thing. It does not correspond to any CD-ROM that customers can buy or file that they can download. Rather it is the notion of an effort whose goal is to make available within a software product a well-defined set of features. This set of features is realized by the feature release's *initial release* and its subsequent *maintenance releases.*

Looking at the example following, the feature release **R3.2** corresponds to the vertical line and all the dots on it. The dots correspond to concrete software deliverables.

In this example, the software vendor uses two digits for the release number (as in **R3.2**) to correspond to the commonly used marketing



practice of having *major feature releases* (such as **R3.0**) and *minor feature releases* (such as **R3.2**). From the point of view of agile horizon planning, they are the same. The reason for the distinction is so that marketing can choose to emphasize significant new functionality when they so choose (and, possibly, charge extra for it).

The first customer-visible artifact of a release is typically its *beta release*, intended for customers and partners to have an early look at the new or enhanced software. Those taking the beta understand that the software vendor will not maintain it, and hence that they should not attempt to use it in production. Often the software vendor will have its beta testing partners sign an agreement to this effect. There may be one

or more beta releases. In the example, there is only one beta, designated **R3.2B1**.

After the betas comes the first generally available release. In the example, it is designated **R3.2.0**. Once this release is available, the software vendor, by default, will ship this release to all new customers. This also implies that client services will be capable of answering questions about the release, offer training on this new release, consulting, and so on. Marketing will need to update its product literature to reflect this new release, and all the sales people will have to begin selling the benefits the new release. The vendor will also begin a campaign to get all its existing customers to upgrade to this new release.

Following on from the initial release are a series of maintenance releases that fix everything that is wrong with the dot-0 release. There are two types of maintenance releases: *point releases* and *patch releases*.

Point releases are regularly scheduled occurrences (for example, once a month) whose purpose is to collect all the defect corrections since the last point release. In the example, the point releases are **R3.2.1** through to **R3.2.5**.

Point releases should contain only defect corrections, not new features. Simultaneous with putting out a sequence of point releases that fix defects in the previous release, the company will also be working through the release cycle for its next feature release, with one

development team typically dividing its time between these two activities.

Patch releases are unscheduled events, and will occur when a customer requires an urgent fix to correct a defect that makes the software unusable by them. A customer will require a patch rather than a point when the defect has not yet been corrected in any previous point release, and they can't wait for the next scheduled point release.

If the software is mission critical, the customer may request a patch to the point release they are currently using in production rather than to the most current point release. They do this to minimize the possibility that the patch introduces unforeseen problems.

Referring to the example, **R3.2.1a** is an example of a patch release. The vendor released it after the second point release, **R3.2.2**, was already available. Patching **R3.2.1** involved retrieving its source code and build environment, putting a targeted defect correction into that code base, re-building, doing a targeted testing of the patch, and then shipping it. Once the developers finish the patch, they will apply the same defect correction to the next scheduled point release (**R3.2.3** in the example).

The release numbering scheme and terminology applied to these concepts will vary from place to place. What is important is that the concept of distinct feature releases, each having a maintenance stream is in effect.

7.2. New Releases

Once the "dot-0" point release for a new feature release is available, the software company should not continue to ship older feature releases to new customers.

Sometimes the company must do this if the first few point releases are unstable. However, this is an embarrassment to the vendor, and is a nuisance to their new customers who will not receive the newest features and will have no desire to upgrade in such a short time. In all likelihood, the vendor should not have released such defective software, but rather issued another beta instead and continue stabilizing until the defect discovery rate was sufficiently low to allow them to confidently ship the new release.

Many would consider it bad business practice, and even unethical, to knowingly ship a defect-prone release just to hit a promised first release date. At the very least, it is bad management.

When the new release is generally available, the software vendor should begin a campaign to get their existing customers to upgrade.

As we shall see later in the chapter, it is in the software company's best-interests to minimize the number of distinct feature releases that are simultaneously in the hands of their customers. As well, there is sometimes a potential for increased revenue associated with getting customers to move onto a new release. Thus getting existing customers to move on is a priority for most software vendors.

The typical software license agreement that customers will sign entitles them to receive a certain feature release of the software along with

maintenance releases for some fixed period. Some licensing agreements will specify how quickly the software company must respond with a maintenance release in the case of various severities of defect. This is called a *Service Level Agreement* (SLA).

According to the licensing agreement, new feature releases may or may not be chargeable. Often this is at the discretion of the software vendor, who can distinguish non-chargeable minor feature releases (*e.g.*, **R3.2**) from chargeable major feature releases (*e.g.*, **R4.0**).

In any case, for bigger-ticket software, there will usually be an annual maintenance fee, typically some fraction of the initial licensing fee (e.g., 20%), which entitles customers to help desk support, maintenance releases, and new minor feature releases.

Software vendors will often limit the amount of time they agree to support an older feature release. Typical terms might be "up to one year after the next feature release is made available". This limits the number of simultaneous feature releases that are in the field.

Whether or not new releases are chargeable, the software vendor must make it a priority to get their existing customers to upgrade to the latest release. The most basic way to do this is to make the new release a sufficiently large step-up from previous releases that the customers wish to do so. This combined with good marketing of the new release and favorable licensing terms in the case of an upgrade, are effective tools.

7.3. The Cost of Feature Releases

For many types of software, each distinct feature release of a product is costly. There is considerable overhead associated with each one: extensive system testing, new marketing collateral, launch events, press releases, customer and partner briefings, new training courses and materials, CDs to burn, and so on. An unexpectedly large cost, however, is the increased maintenance burden that occurs when the company supports simultaneous feature releases in the field.

When a tester or a customer finds a defect, development must fix it in all feature releases then extant in the field. This first requires that the company determine in which feature releases the defect manifests. Once identified, development must then re-instantiate the complete source code and build environment for each feature release in turn, re-build, verify the presence of the defect, decide what to change in that release to fix it, re-build, test, and release to system test. System test must then restore the regression testing environment for that release, re-run their regression tests against the new point release, and then test the new fixes to verify that development has corrected the defect. The company can then make available the new point releases.

Note that it is not the number of maintenance point or patch releases that are important. It is the number of simultaneously maintained feature releases that are costly. For maintenance releases, system testing consists simply of re-testing previously tested functionality via a suite of regression tests. There is not a lot of costly marketing or training overhead associated with maintenance releases. Most importantly, unlike feature releases, the issue of a new maintenance release does not imply a new maintenance stream where developers must fix defects.

There is usually no alternative but to deal with at least two simultaneous feature releases: the one currently in the field, and the one currently under development. As well, with nine months or so release cycles, there is usually at least one previous feature release under maintenance still in the field. Software companies should try to hold the line at these three feature releases, and not support any more owing to the maintenance costs involved.

The ultimate result of increased maintenance costs, as we shall see, is the erosion of the company's ability to respond in a timely fashion to changing market situations, exactly the opposite of what good agile horizon planning should bring.

When considering the implications of an increased maintenance burden, accountants will think in terms of the average loaded cost of a developer, amounting to somewhere between $100K and $200K per year. Actually, the true cost lies elsewhere.

When the maintenance burden increases, one of two things can happen. Either we have to hire more developers, or we have our existing developers do more maintenance and less new development. While the latter is not a monetary cost, it is an opportunity cost, and, in the software business, opportunity cost is what hurts us.

The company can deal with monetary cost by raising more money. The company cannot deal with opportunity cost as easily. Opportunity cost means that the company cannot do the things the business requires, and means they might fall behind their competition or miss windows of opportunity.

We cannot easily trade opportunity cost for monetary cost. The skills of the developers are unique. Only they fully understand the software. It takes many months to hire a new developer and bring them

up to speed on the software. Moreover, during their apprenticeship they use up time from the experienced developers: more opportunity cost. Hiring developers that are more expensive or high-priced consultants helps a little, but surprisingly not all that much. What the company needs is experience working with their code, and they cannot hire that.

This is the true cost of an excessive maintenance burden: the software vendor winds up losing responsiveness to the market as a whole. This negates the benefits they were attempting to realize through the proper planning of releases.

7.4. Being Responsive to Customers

From the previous section, we have seen that each new feature release is a costly undertaking. This argues for longer, well-spaced feature releases. However, while putting out a series of well-spaced feature releases that address the bigger-picture market needs is essential to the longer-term outlook, the software company neglects the needs of individual customers and new prospects at its peril. Unfortunately, an inevitable tension arises between a nine month (or longer) feature release cycle and being responsive to individual customers' needs. There are good arguments both for and against a longer release cycle. In this section, we will consider some of the tradeoffs involved.

Customers will upgrade to new feature releases only so often. Customers have a love-hate relationship with new feature releases. They love the new features, but they hate installing a new release. Apart from just the time-wasting mechanics of it, they get used to a particular release and learn how to work around its issues and exploit

its broken features. When a customer installs a new release, they introduce subtle (and not so subtle) incompatibilities that reduce the productivity of their end users. Under these conditions, end-users see "improvements" as steps backward. As well, the way the software works has probably changed and so there is a new learning curve end-users will have to climb.

More significantly, in cases where it applies, if the customer has gone to some effort and expense to integrate the vendor's software into surrounding systems, the customer's IT department will be reluctant to undertake a project to upgrade the vendor's software. They will insist on a cost-benefit analysis, comparing the benefits accrued from the upgraded software against the costs of upgrading. Many organizations outsource integration efforts to consulting firms. In this case, the cost of upgrading is concrete (and significant).

The net result is that users are stickier on old releases then the software vendor would want them to be. If a customer is sufficiently powerful, they may even get the vendor to commit in writing to supporting an old release indefinitely. "Supporting a release," means putting out a continuous series of point releases and patch releases to correct defects.

This behavior from users argues for a longer release cycle. The long cycle allows the software company to get significant new functionality into their product; hopefully enough new functionality that the customer base as a whole will wish to put up with the pain of upgrading. If release cycles are too short, there will be customers remaining on more of the previous feature releases, necessitating costly multiple maintenance streams.

A second argument for a longer release cycle, as we have discussed previously, is the overhead associated with shipping a new feature release: intensive system testing, a new release to maintain, new marketing collateral, new training courses and materials, new CDs to cut, and so on. This also tends to push release dates out farther.

On the other hand, if an individual customer wants or needs some new functionality in the software, they may not be willing to wait for the next feature release.

As well, market pressures may force us to get some new features out quickly; for example, to respond to a competitor's move, or to respond to a shift in our customers' regulatory environment.

In cases where an annual maintenance fee is paid for the software, if release cycles are too long, customers may begin feeling as if they are not getting their money's worth from the maintenance fee. The customers may choose not to upgrade, but at least they will have the choice, and they will want that.

Finally, a most significant reason to release quickly is a big prospect that refuses to sign the contract unless we put specific new features into the software for them.

These pressures are very real, and will force the software company's hand more often than not.

There is therefore a tension between cost-efficient, long feature release cycles on the one hand, and customer responsive, short feature releases on the other. What can we do to reconcile these two points of view?

Unfortunately, there is no easy answer. Because of the leveraging involved, it is never good to put out a new feature release (available to

all) just for the sake of a few customers or prospects. The costs involved do not justify it. It only makes sense to deviate from the well-spaced release cycle when market pressures demand it. In those cases, where there are severe competitive pressures, a shorter release cycle may be justified. If it is a one-off occurrence, in some such cases the vendor can instead extend the previous release cycle, and add in the extra functionality required by the market.

Unfortunately, these considerations leave us no room to address the needs of individual customers and prospects. Satisfying their needs must involve shipping enhanced functionality outside of the regular release cycle. There are various mechanisms available to us to do this. We shall consider them in due course. However, before going to any lengths to satisfy their perceived needs, it is always wise to first understand if there is a true need there or not.

7.5. Pushing Back

When faced with the requirement to distort release cycles or insert new functionality outside of the regular release cycle, the first thing to do is to healthily resist such requests. In the software business, we call this "pushing back".

Sales push for features that make their prospects happy. Client services push for features that satisfy their existing customers. Marketing pushes for features that make the products more marketable. With all this pushing, there is usually only software development and (hopefully) product management pushing back.

In many cases, it is surprising what just a little push-back can accomplish. Often, others are just seeing what they can get away with,

and will back-off easily enough. In other cases, they are genuinely convinced that a big sale will be lost or a big customer will defect to the competition unless a new feature appears quickly. In these cases, it is often worthwhile to track the request back to its source.

When we follow the trail back to the ultimate end-user requestor, they will sometimes tell you that the request is not very important. The urgency of the request; in moving up the client's organization, over to your company's sales person, up the ladder to the VP Sales, back down through VP Marketing, a side trip to the CEO, and then back down the management chain in Development; can easily become exaggerated. It is good advice to anybody managing a product, from development or product management, to always go back to the source.

Pushing back will not always work, and we have to be careful that we do not push back so much that we are harming our own company's best interests, but it is worth a try.

7.6. Features in Maintenance Releases

Given that we have tried pushing back and it did not help, it may ultimately be necessary to accommodate an important customer or prospect by adding features outside of the regular release cycle.

One way of doing this is to slip these new (or changed) features into (what is billed as) a maintenance point release. In theory, maintenance releases should contain only defect corrections and modify no correctly-functioning customer-visible aspects of the software (such as file formats, GUI's, customer API's, program behavior, and so on). Putting a feature into a maintenance point release goes against this.

Especially when a company first adopts an agile horizon planning culture, there is a strong temptation to use maintenance releases as vehicles for shipping enhanced functionality to customers in this way. The advantage of this approach is that it avoids the overheads associated with new feature releases. In particular, it avoids the need to maintain a new feature release in the field. These advantages are tempting.

Unfortunately, while tempting, putting features into point releases will lead to negative consequences that will ultimately cancel the advantages of sound agile horizon planning. Let us see how this happens.

A basic rule of software is that we cannot introduce new code into a release without also introducing new defects.

There are two reasons for introducing new code into the software: to correct a defect or to implement a feature. When correcting defects, we will typically introduce fewer new defects than what we fix. While this situation is not ideal, the trend is good. The software will tend to get more stable through time and eventually development will fix all the defects (asymptotically – in the limit).

If we begin introducing features into point releases, this will no longer be the case. In implementing new features we will inevitably introduce new defects. This has a negative effect on quality. In fact, we can easily get to the stage where the number of defects in point releases is steady or even increasing.

Moreover, there is a kind of leveraging that works against us. New features we introduce are typically useful to only a few customers. Instability in a release affects all our customers. In the worst case,

putting features into point releases creates the possibility of breaking functionality that has worked previously. This is not good. Customers begin counting on the functionality they use, and get irate when a "maintenance release" breaks features they were using in production. It may even be that we fixed a hundred defects and broke only one thing. Chances are that the customer was either unaware of those defects (we wish) or had learned to live with them (more likely). They had also learned to count on the feature we just broke.

This is a danger even when just fixing defects, but we increase it significantly if we add new features into point releases. The bulk of our customers will rightly ask why we did this to them needlessly (at least from their point of view).

Is it not possible that we can avoid this problem? Surely there are development practices such that we can introduce new features without breaking anything? Practices such as debugger walkthroughs, code reviews, extensive unit tests, or full suites of automated regression tests? Is it not the case that these best practices will eliminate any possibility of introducing new defects?

While theoretically possible, even the best practices still leave a residue of defects that escape into the field. There have even been instances of defects in software that has had far more resources thrown at it to ensure correctness than is even remotely feasible for typical commercial software. Examples include software for space probes, nuclear reactors, jet planes, and high-powered radiology machines (the loss of the Mars Lander comes to mind due to a software error involving fuel calculations).

Development should always strive to minimize the number of defects that they introduce. Experience, however, has shown that the residual is usually sufficiently high that putting new features into point releases will remain a risky proposition.

All software starts with defects. Point releases ought to take the release to a less defect-laden state. If the software company starts putting features into points, that delicate balance is upset. If they do too much of it, the software quality will diverge distressingly.

7.7. Release Proliferation

The intent of agile horizon planning is to provide a mechanism whereby, given the right plan, we can address the needs of the marketplace while not neglecting the needs of individual customers. The repercussions of putting features into point releases are the negation of these planning benefits.

The most obvious repercussion of putting features into point releases is that the software company will get a bad reputation for quality that will hurt sales.

A more subtle repercussion is that customers will be slow to upgrade to the next feature release. Knowing the software is defect-ridden, they will hold off upgrading as long as they can. If we charge for new releases, this hurts revenues. Even if not, customers will stay on old releases for longer than we would like. As a result, there will be more releases to maintain in the field. We call this *release proliferation*, and it has a significant maintenance cost associated with it.

While one form of release proliferation occurs when customers refuse to upgrade to the next feature release, a more catastrophic form occurs when they refuse to upgrade to the latest point release. If we carelessly add features into point releases, this can easily happen.

In the usual scenario, only feature releases contribute to release proliferation. It does not matter how many distinct point and patch releases there are. There may be sixty-eight of those. If they all came from the same feature release, then a newly discovered bug must be fixed in only one place: the latest point release. However, this holds *only so long as customers are willing to upgrade to that point release*.

If a customer refuses to upgrade to the latest point release, we will need to issue a patch to the point they are using. This is especially likely if the customer integrated the software into a larger system. If we issue a patch to the point they are using, then instead of fixing the defect in one place, we have just had to do it in two places. If all our customers refuse to upgrade to our latest point we will have to patch all sixty-eight previously issued patches and points. This constitutes massive release proliferation. Even a little of this can drastically reduce the developers' productivity.

The surest way to get customers balking at moving to our latest point is by putting features into these point releases.

The reason they would normally move to a new point is the promise that the software is more stable. If a persistent proclivity to put features into points destroys this, they will not upgrade. If we force them, they will be dissatisfied. The only alternative to forcing them is to fix the precise point or patch they are currently using.

7.8. Mitigating the Consequences

Despite these dire consequences, it is sometimes the case that the only reasonable way to proceed is to put features into points. Given the inevitability, it is wise to take pains to ensure that the new feature has a minimal effect on the quality of the point releases to the majority of our users.

To help ensure this we need a good regression testing environment that will catch incompatibilities. We need code reviews, extra focused compatibility testing, and we should use run-time switches and/or dynamic load modules to ensure that the feature is accessible only to the requesting customer. Development should use code reviews to ensure that when the feature is disabled, the bulk of the customers will execute no new or changed code.

While it is not possible to completely disallow features in point releases, applying the above-mentioned measures will greatly reduce the number of times we have to do it and increase our margin of safety in those cases where we absolutely must.

7.9. Impact of SaaS

The situation alters considerably for SaaS software where we advise a more continuous release methodology. There are two pertinent differences between SaaS and traditional software releases: forced upgrade and low-cost deployment.

Forced upgrade with SaaS means that as soon as new code is pushed out, every customer is instantly using it and they have no choice about going back. With packaged software, usage will spread more gradually

and customers often have the option of continuing to use the older release until they are happy with the newer one. Sometimes SaaS gives the option of "go back to the old interface", but even in this case it is typically the new code that is being used.

SaaS has very low-cost deployment as compared to packaged software. There is only one target environment (the one on which it is being run, for example a specific release of MySQL, Linux, Apache, and Java, for example). With packaged software, the new release may need to run on many environments with all sorts of previous installation history outside of the control of the software vendor. This therefore requires extensive system testing after any substantial change to the code. This is still the case with packaged software that automatically updates itself. Developers must take care to avoid system incompatibilities. Not so with SaaS, where the software must be confirmed to run in only one environment.

These two differences, forced upgrade and low cost release, push us towards a more continuous approach to releasing software for the following reasons.

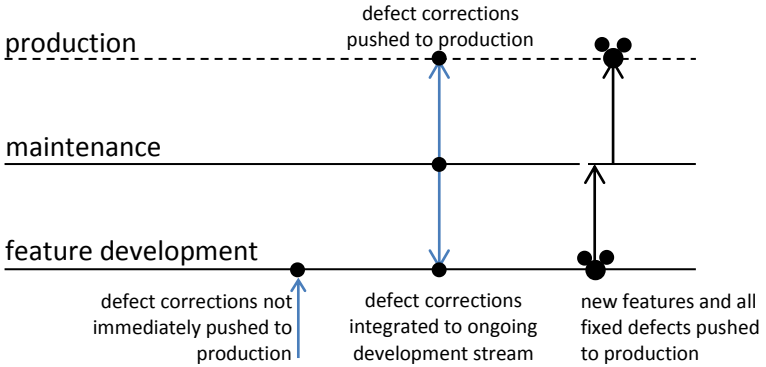
A "big bang" release of SaaS is a very dangerous thing because of forced upgrade. Releasing a lot of code into production all at once concentrates the risk into a tiny window, analogous to sitting on a single nail. It hurts. Releasing code more continuously into the field is like lying down carefully on a bed of nails. Not entirely comfortable, but quite safe. Low cost release makes it possible to push out code more often and yet still be cost effective.

Outside of the safety concerns, more frequent releases to the field allow us to gather feedback that matters (from real customers using the

software in production). This feedback can guide both the details of future planned development, but can also cause us to rethink our agile horizon plan and make changes to it to reflect the feedback we receive.

Therefore we must reconceive the notion of "releases" in the SaaS world. We still require a mechanism to very quickly address a problem in the production code. So a maintenance stream is still required, but it is never very long-lived. Any urgent changes must be made into the maintenance stream, and that tested quickly using automated regression tests and pushed out into production. It should also be possible to quickly revert the production code back if the patch causes an issue. Some organizations have even automated this process to the extent that if production monitoring systems pick up a problem (*e.g.*, many failed operations) the code can be reverted automatically.

Defects that need to be corrected in production immediately need to be both pushed into production, but also integrated into the ongoing feature development stream as shown in the diagram below.



Some defects fixed in the feature development stream may not be so urgent, and do not need to be pushed to production immediately, but can wait and go with the next feature release bundle. It is efficient to do this, as a release to production generally has associated overhead.

Every several weeks, code from a feature development stream would get pushed into maintenance and from there to production. Comprehensive nightly automated regression tests and an even more extensive pre-release regression suite need to be used to safely maintain such a release pace.

If a release of features to production requires a database schema upgrade, the release preparation time must be considerably increased to accommodate this. Moreover, the code and database should ideally be designed to work together in such a manner that the old code can work with the new database schema. This is because reverting code is relatively easy, but undoing a database schema change is difficult as soon as the first customer makes the first change that goes into the database. There are some clever design techniques that can be used to maintain backwards compatibility.

The feature code that gets pushed to production may not be made available to customers. It is often wise to deploy the code with a configuration switch. In this way, new features can be production tested before being made generally available as part of a feature bundle. Details as to what the bundle will contain, and when the master switch will be flipped is a product management detail. When new features get pushed into production is a software development detail. Governing everything is the agile horizon plan, which rises above this level of detail and addresses the question of what features will be released within a longer planning horizon.

7.10. Summary

For packaged software, before a software company can start down the course towards a solid agile horizon planning process, it must first establish a discipline of well-spaced feature releases with associated maintenance releases that just fix defects. This is a cultural transition for the starting software company, but becomes necessary as the company grows if they are to balance responsiveness to individual customers with responsiveness to the market.

In satisfying individual needs, we must be careful to avoid release proliferation, which results in increased maintenance work for our developers. If left to get out of hand, these can have a surprisingly bad effect on developer productivity.

Many development managers watch their departments grow from five developers to a hundred, and wonder at how much more productive the department was in its early days. The manager may be unwittingly thinking of productivity in terms of putting new features into the software, and not as overall productivity including maintenance.

This distressing loss of productivity is almost always due to the increased maintenance burden brought on by rapid growth in customers, possibly combined with lax controls on release proliferation. The wise manager will attempt at every turn to control the situation, and the wise software company will back her up.

SaaS software is different, and a quicker release to production tends to be both safer and more beneficial for gathering fast feedback. Developers must however use a disciplined approach to maintenance and new feature releases involving extensive automated regression tests and careful attention to backwards compatibility issues.

8. Software Versions

In the previous chapter, we considered the costs associated with supporting multiple feature releases simultaneously in customers' hands. In this chapter, we will examine the related issue of the costs and tradeoffs associated with supporting many simultaneous *versions*. One of the great inherent benefits of SaaS is that the software only exists in a single version, so the material in this chapter applies mainly to packaged and enterprise software.

8.1. Concepts & Terminology

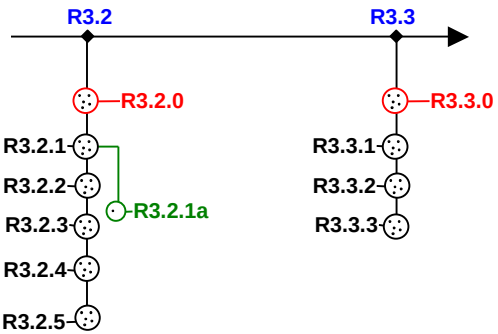
We define a *version* of a software product to be a variant of a product, each version differing in some, typically small, way from one another.

One of the most common reasons for having versions is to support multiple hardware and/or operating system platforms: Windows'2000, Windows'95, Windows CE, Linux, Apple Macintosh, Sun UNIX, SGI UNIX, IBM UNIX, HP UNIX, AS/400, mainframes, Palm Pilots, and so on. Combine this with different releases of the OS; such as Windows '95, '98, Me, and XP; or Solaris 4.1 and Solaris 4.2; and the number of versions can escalate rapidly.

Sometimes products must interface to other vendor's software products. In this case, there will be one version for each feature release of the other vendor's product that we support. For example, multiple database backends and multiple releases of those backends. A vendor may choose to have different versions of their software for different

markets: 128-bit versus 56-bit encryption, or "Standard" and "Professional" versions of the software for different market segments. Another common reason for versions is translations of the software to other languages: English, French, Arabic, Chinese, and so on.

Usually, development builds many different versions of the software from the same code base. Generally, the notion of a version carries the connotation that the vendor must support it in two ways. First, there will be a stream of maintenance releases associated with the version. In particular, each point release will ship all the various versions being supported. Second, each feature release will continue to support that variant. The diagram below illustrates these concepts.



Each point release of each feature release ships every supported version of the software. Note that patch releases only need to patch the specific version the customer is using.

Once a software vendor commits to supporting a new version of their software, it is a difficult decision to undo.

For example, suppose a customer requests a new version of the software to run under Linux. The agreement to deliver implicitly carries the commitment of also porting all new feature releases and their maintenance streams to that platform. It is rarely a one-off occurrence (even though the request is presented as such). If the vendor, after the fact, decides that the revenue stream does not justify the costs, they cannot stop supporting that version. The customers who rely on that version will have made costly decisions around the commitment that the vendor supports, in this example, the Linux platform.

8.2. Costs of Versions

One of the most crushing weights a company can put on their development staff is having them support an excessive number of versions of the software.

The cost is never just the development costs involved in first creating a new version. The more significant costs are those involved with supporting the version going forward.

When different versions have different code written for them, there is a cost involved in producing that code. In other cases, the code can be the same but developers must compile and link them differently. In yet other cases, the versions are binary-compatible: for instance an application that can run on both Windows'95 and Windows XP. In all cases, development must test the versions individually and track down defects peculiar to that version.

Developers must always be aware that they are writing code, build environments, test scripts, install scripts, and documentation that

applies to multiple versions. It is all too easy for the development group to get the code right on their development platform, but inadvertently neglect to get it right for a different platform. To solve this requires process discipline, a sophisticated build environment, extra computing equipment, a large test department, system administration expertise for all supported platforms, and supporting libraries and development tools for all platforms. Ideally, these resources should be conveniently available to client service representatives, developers and testers.

Supporting such an infrastructure carries considerable costs, both monetarily and in developer-days (opportunity cost).

8.3. Version Proliferation

Many pressures come to bear on the software vendor pushing for increasingly more versions.

A software vendor will support new versions in the hope that sales will increase. For example, the vendor may hope that supporting multiple versions of the software for various hardware platforms will increase the size of their target market and thereby increase revenue.

They may also support new versions in order to forge better relationships with partner companies. For example, a consulting firm may have a relationship with a hardware manufacturer. In this case, it would be awkward for the consulting firm to advise their clients to use the vendor's software if it does not run on their partner's hardware platform.

If left unchecked, the number of versions may increase dramatically: a situation we refer to as *version proliferation*.

It is a danger for software vendors to hastily commit to supporting too many versions. Sometimes software development themselves are the culprit. An inexperienced developer or manager may indicate that porting the software to, for example, another flavor of the UNIX operating system is technically straightforward. However, the ongoing maintenance is the larger part of the cost, and the inexperienced developer may not have fully considered this aspect.

As for its cousin, release proliferation, the only sensible thing to do is to be aware of the costs and push back when pressures mount to support a new version of the software. The main mechanism for pushing back is to ask if the incremental revenues from this new version cover the costs, and especially the opportunity costs.

For example, say a hardware vendor offers to cover all development and maintenance costs to support the product on their platform. Even under these conditions, it may still not be worthwhile to do so. The most significant cost is not the financial cost. Rather, it is the opportunity cost of the developers. As those within a software company are always more bullish on its prospects than those without, the two parties are liable to value the opportunity cost differently.

If a software vendor does find itself in a situation where they must support many different versions, it is wise for them to invest heavily up-front in constructing an excellent build environment and technical infrastructure that can cope with it.

8.4. Static Versus Dynamic Versions

There are several methods whereby new versions of software may be created, presented in order of increasing desirability:

- Parallel code streams
- Conditional compilation (**#ifdef**'s in the code or macros)
- Run-time switches

Parallel code streams means maintaining parallel versions of source files (one, more than one, or even all source files). This can certainly accommodate any variations amongst the versions, but is costly to maintain, as defects must be corrected in many different places.

Conditional compilation is essentially the same as parallel code streams. It at least has the benefit that when a coder is doing some work, she can physically see the alternate code paths in the same file. However, it is still the case that multiple builds of the software must be made and organized and tested and shipped individually.

The best approach is dynamic modification via run-time switches. In this case, different versions may be tested easily by coders and testers simply by flipping some run-time switches. As well, install scripts may be consolidated and inventory (if shipping CD's for instance) is much simplified. As well, it opens the possibility for the versions to be switched based on dynamic license key information. This enables relatively easy up-selling of higher-cost versions.

Note that while the dynamic approach is the best of the three, it does not solve the issues inherent in supporting multiple versions, and the bulk of the cost remains.

8.5. Customized Software

One compelling reason to support a new version that we have not previously looked at is to provide customized software to important clients. For example, say a prospect will not buy the software unless the vendor modifies it in some manner to suit their peculiar needs.

There are two broad classes of technical methods to achieve customizations: static and run-time. Static methods require that the vendor build a distinct executable. Dynamic methods enable the vendor to ship only one executable, but provide the customizations by means of alternate dynamic load modules or run-time switches.

In the dynamic case, if the vendor makes the changes as part of the next feature release of the product, then the implications are the same as those involved with supporting any new feature in the software. The vendor should weigh the cost of developing and maintaining the new functionality against the revenue potential. If the customizations are not generally saleable, either because of their unique nature or because of prohibitions against shipping the customizations to others, then the vendor must negotiate with the customer to see if the revenue from them alone can justify the new feature.

While a software vendor can often justify such decisions based on financial cost, it must also be the case that the revenues justify the opportunity costs. If the same developers put in a more generally useful feature instead, there is leverage to the revenues available because of the large potential market not available if the feature is only useful to one potential customer.

If the prospect requires the customizations outside of the regular release cycle, then the vendor must choose whether to do it at all, issue a new version of the software until the next feature release, or carefully insert the alternate functionality into a point release.

Because it is such a risky proposition (and such a slippery slope), we would advise the vendor to stay well away from adding features into point releases. If the vendor chooses to take this approach anyways, then they must do it carefully so that there is no impact on code executed by the majority of their customers. The cost of taking these precautions versus the cost of issuing a new, static version of the software is roughly equivalent. Of course, the vendor can always hack the customizations sloppily into a point release. However, the longer term costs associated with this approach are considerable, as we have discussed previously.

Whether based on the preceding considerations, or whether it is the only technological way to proceed with the customizations, the vendor now faces the prospect of issuing a new, static version of the software.

The issue now is whether the development can merge the modifications into the next standard feature release executable, whether the customizations must be maintained as a parallel static version through all future feature releases, or whether the customizations are so great that the modified software is essentially a new product, with its own independent feature releases. The costs increase exponentially with each alternative, respectively.

From these considerations, we see that is rarely wise for the software vendor to support alternate versions of its software for particular

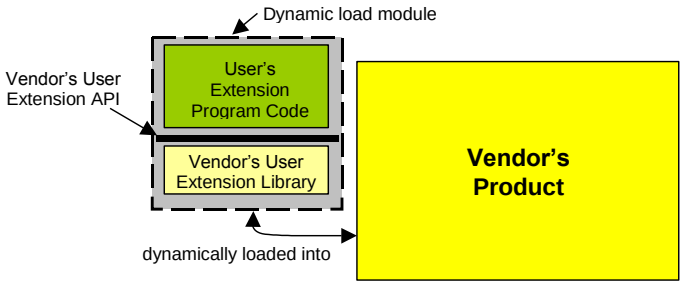
customers. The best approach is to build enough configurability into the software (accessible through the user interface or scripting support) so there are not so many requests for customizations in the first place. The second best approach is what we will consider in the next section.

8.6. User-Extension APIs

There is a way, in theory, of simultaneously satisfying the need for well-spaced releases, the desire to avoid too many customer-oriented versions of the software, and the desire to be responsive to new feature requests. This is by supporting a *user-extension API* in the product.

The acronym API stands for *Application Programming Interface*. A user-extension API allows technically sophisticated end-users to extend or modify features in the product for themselves.

To use the API, the customer writes program code with this API as one of the libraries to which they link. They then compile the extension, and indicate at run-time to the product that it should load the resulting *dynamic load module*.



Other ways of supporting a customer extension API are by means of an embedded scripting language, inter-process messages (such as a Windows COM API accessible via Visual Basic), and even by allowing users direct access to database tables. In all these cases, the considerations are similar.

These mechanism allows customers to implement their own features into the product, allows third-party consulting companies to do this work on the customers' behalf, allows the software company's own consulting group to do this work, and allows other software vendors to market value-added extensions to the product.

Scripting solutions are an excellent way of providing extension APIs. The requirement is only that the API's remain consistent from release to release. This is usually technologically feasible, even in the face of changing implementations, by always re-implementing compatibility API's at the same time as providing newer API's to exploit the enhanced functionality. However, scripting interfaces can be very slow, and dynamic loading of compiled code may be the preferred alternative for this reason.

Depending on the technology, the dynamically-loaded extension may or may not be *binary-compatible* or *source-compatible* with future point releases or feature releases.

Binary compatibility means that customers can use the same dynamic load module to extend the next version of the software. This is the best from the customers' point of view.

If binary-compatibility is not technologically feasible, the vendor should strive for at least source-code compatibility. This means that when the customer moves to a new release, they need not change the

source code for the extension in any way, but that they must re-build the extension to support the later release. This is next best and is still reasonable.

Worst is if the vendor breaks source code compatibility, meaning that the customer will have to modify the source code and re-build in order to use the extension with a new release. This implies the need for applications programmers to go back in, understand how the extension functions, understand the changes the vendor requires, and understand how to modify the extension to accommodate these changes. If a consultant built the original extension, this implies extra expense to the customer.

The more successful is the API, the more negative leverage there is when the vendor breaks compatibility. If the extension is successful, and they have a large number of users, breaking binary or source-code compatibility in even the smallest way implies that thousands of users must put in extra work to use a new release. As well as adversely affecting customer satisfaction, this can also contribute to release proliferation, as they will be reluctant to move on to a new release, fearing the work involved and the uncertainty.

There is a significant distinction between using dynamic loading technology internally and using it to provide an externally supported API.

There is no question that it is good practice to use such dynamic-loading APIs for internal, development use. They reduce build time, define solid boundaries between development groups, and encourage a better architected product.

However, it is a big step from supporting such an interface internally to supporting it for the world at large.

A software vendor should not take the decision to support a user-extension API lightly: it is a costly undertaking.

Even if we assume the best case scenario, where there is already a suitable API that development uses internally to dynamically load extensions for themselves (and hence there is no new technology that needs to be created), the step-up to supporting this API for externals is a big one.

First, development must clean up the API and keep the code for the interface files pristine. They must carefully distinguish those parts of the internal API that they support externally, and those parts that are private. Next, documentation must produce an enhanced level of API manuals, including user manuals, tutorials, and reference manuals. These are typically far in excess of anything the vendor may have been using for internal documentation. The vendor must provide training courses, hiring trainers who themselves are competent programmers.

The vendor must provide their customers with support in using the API. This implies the need to staff the help desk with competent applications programmers. It also will imply the need to run a consulting practice to assist users in developing enhanced functionality. For a company unaccustomed to supplying consulting services, this can be a major cost and a management distraction. If the company does not provide such a consulting service, they will find themselves providing such services free of charge, lacking a mechanism by which they can charge for the help or even effectively manage it.

Finding good programmers to act in training or support roles is often difficult. Moreover, as a growing software company is often limited in what it can do to its product by a lack of developers, shunting programmers to support roles constitutes a significant opportunity cost (what else could the company have had those programmers work on?).

On the sales and marketing side the vendor will need to develop marketing collateral to tell the company's prospects about the API, including technical white papers and brochures. The vendor must instruct sales in how to sell the API, and provide them with pre-sales support (again, using developers for this).

Debugging customer problems regarding the use of an API is costly. When supporting an API it is common for problems to be due to misuse of the API by customers. However, because there is always the chance that the vendor's software is in error, customer services find themselves in the position of having to debug their customers' code in order to determine the cause.

Maintaining binary or source-code compatibility puts a burden on the developers. For all defect corrections and new features, they must make certain they do not affect the behavior of the API in any way. This is difficult to accomplish, and requires upgraded code reviews and regression testing. Regression testing an API requires more work, especially as the number of potential interactions is so great.

Supporting an API constrains the types of architectural enhancements that development can make. If for internal reasons development considers it necessary to change some aspect of the API, they must develop new wrapper code to preserve the old form for the customers while offering the new form as an alternative.

If the vendor mistakenly breaks binary compatibility or source-code compatibility, the repercussions are great. Customers will be dissatisfied, and the vendor may have to bear a large part of the cost of upgrading customers to the new version of the API.

Any software vendor, in considering whether to support a user extension API, should bear these points in mind.

On the other hand, an extension API can be a selling point for the software, and may be a big competitive advantage. Even when customers do not use the API (or do not even buy it), they are happy to know it is there in case the need arises

An API can support a third-party extension market around the product which can contribute to it becoming a market standard. In addition, it opens a market for systems integrators to provide additional value-added services around the product. This can encourage these systems integrators to favor a particular vendor's product in competitive situations.

If done well, an API can satisfy customers or prospects that want or need new functionality in the software to appear quickly despite the fact of a long feature release cycle. It helps especially when a customer requests a feature that the vendor would not normally include in the software fearing that the functionality is too specific and has no general marketability.

When supporting an API and providing consulting services around it, the vendor must take particular care in distinguishing who supports an extension. If a customer or third-party supports the extension on an

ongoing basis, then this is just the sort of leverage the vendor wants from the API.

If, on the other hand, the burden for ongoing support of an extension falls to the vendor, then this is similar to supporting a new version of the software. The company is forever-more committed to supporting this new version of the software: the version that has the extension installed.

Given that the vendor must run a consulting group, one might think that ongoing support of an extension by the vendor (provided it is all chargeable) is a good thing. This is often not so. The vendor runs a consulting group because it must in order to leverage sales. The vendor would typically much rather deploy programming resource on leverageable product features than on un-leveraged consulting. The best situation is when either the customer or a third-party supports the extension.

By supplying a customer extension API, it is possible to have long feature release cycles amenable to good agile horizon planning for packaged software, yet still be responsive to individual customer needs and late-breaking market events. The vendor must balance the benefits against the costs, which are surprisingly high even when a suitable API is already in use internally.

The best advice is to consider the question carefully in advance, bearing in mind a realistic assessment of the costs. If the decision is to go forward with marketing a customer extension API, then it should be done properly, or else it can easily backfire causing customer dissatisfaction and a greater than expected distraction to the development team. The company should be especially careful

concerning committing to ongoing maintenance of an extension, as the increased revenues it brings rarely justify the opportunity costs.

8.7. Summary

In this chapter, we looked at the costs and tradeoffs associated with supporting multiple simultaneous versions of a software vendor's products. While desirable from a market and customer responsiveness point of view, the opportunity costs of deploying developers to maintain many versions can be large. This is especially damaging if the situation is uncontrolled by management, and version proliferation sets in.

One cause of versions is creating customizations to the software demanded by individual users. We discussed how, for a software vendor company intent on marketing software to a broad audience, this practice will rarely be justified, owing to the opportunity costs involved.

We then saw how by supporting a user extension API, a vendor could avoid the issue of software customizations, passing these instead on to customers or third-parties. However, there are significant costs associated with maintaining such an API that management will need to appreciate, and there is always the risk that a customer's extension will wind up being supported by the vendor, which carries with it all the costs of any customized software.

9. Source Control & Build

The previous two chapters on releases and versions set the stage for a discussion of source code control, build management, and testing. In this chapter we will discuss the importance of such systems, explain how they are used, and give some practical suggestions for setting them up.

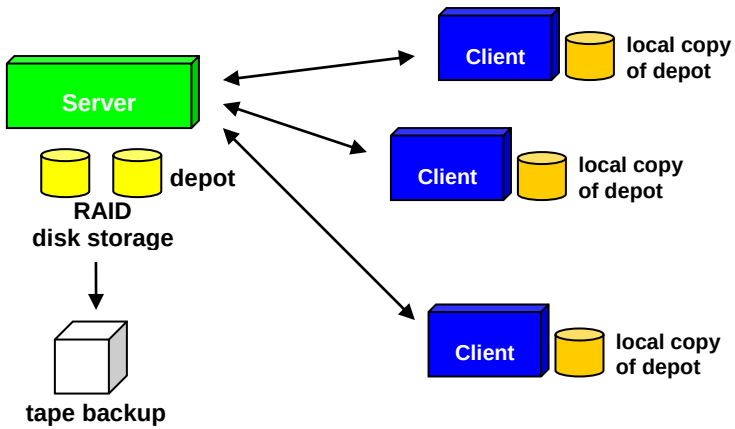
9.1. Requirements for a Source Control System

In this context "source" is used in its English sense, meaning that a source file is one that is not generated by the action of an automated tool, but rather is the source from which these tools generate other files. For example, object files and executables, which are generated by running a compiler and linker against source code are not considered "source". Rather, these are "intermediate" or "final" files.

Source files also comprise documentation, unless it is automatically extracted from other files.

A source code control system maintains a central repository of version-controlled files (often nicknamed the "repo" or "depot"). The central repository usually consists of a collection of files containing the version data, a database that indexes the files and other related information (such as the relationships between files), and a service that can be accessed over the network for performing source code control operations from diverse remote client computers.

The depot should be kept on a server-class computer, with redundant power supplies and network connectivity, and using RAID disks for storage to protect against disk drive failures. As well, the source code control system's data should be backed up each night to tape, and regularly rotated to an offsite location in case of fire or similar catastrophe.



The various clients connect to the server, and establish a local "mirror" of a subset of the depot at a certain revision level. By contacting the server, the clients can update their local copy to the latest revisions of the files, can indicate to the server that they intend to edit, add, or delete a file, and can then "check-in" changes back to the depot, making them available to others.

More recently, distributed revision control systems have emerged (such as such as Git and Mercurial) that eschew the centralized server in favor of keeping multiple copies of the entire source code data, one

copy on each developer's workstation. Whatever the technology, functionality concerns are similar.

Ideally, check-ins should be atomic, meaning that when other clients synchronize to the latest revision, they either get all of a check-in or none of it (not half the files at their newer revision, and the other half at an older revision).

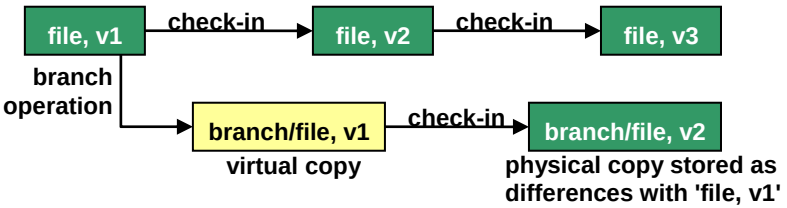
The system should store files and changes to files efficiently. This may mean storing only the differences from one revision to another, or compressing the files or the differences. The system should have a means of associating meta-data with each check-in: for example, a check-in comment, who performed the check-in, when, and for what reason.

The system should have a means of "branching" sets of files. Branching means making a virtual copy of a set of files in one part of the depot to another part of the depot, and then understanding that the branch happened. This is necessary in order to support multiple maintenance streams (*e.g.,* simultaneous work on release **R3.0** while maintenance proceeds on revision **R2**).

Branches will often encompass all of the source files of a large system (*i.e.,* tens of thousands of files) and hence branch creation should be an inexpensive operation (not using much disk space and not taking very long) as branches are common occurrences.

A good source code control system accommodates this by not actually copying files, but by making a notation in a database and making it appear to the clients that two copies exist.

For example, in the diagram below, after the branch operation a virtual copy of **file** at revision **v1** was made into the branch named **branch**. The database recorded the fact that the branch occurred, but no physical copy of **file** was required at the time of the branch operation.



As long as the branch does not check-in a modification to a file, this will work. As soon as the branch checks-in a modification, then the file differences will have to be physically stored separately from subsequent differences to the original file. The branch on which the change occurs will not need to copy the entire change history of the original file as the system will be able to the show clients a complete version history that includes history prior to the branch copy because of the information stored in the database.

Once one or more branches for a file exist, the system should provide tools that aid the coder in applying the same changes to all branches. This is a common operation that occurs every time a defect is fixed in a maintenance stream. The changes must be "merged" back into the main stream and other parallel maintenance streams.

This is straightforward if the modified file is still identical in the branches. However, if one or the other streams have modified a file uniquely (without the same modification going into all the other branches) then the merge may be challenging, and will require an

analysis of the way in which the files differ. Knowing the branching structure and the modifications that have occurred should in theory allow a merge tool to do a better job. However, in all such cases it would be unwise to allow the tool to perform the merge without checking its suggested merge carefully.

Finally, a source code control system should provide a powerful labeling facility that allows a set of files, each at their own revision, to be labeled (*e.g.*, as "Release 3.2").

It should then be possible to synchronize a local copy of a depot to a label (to debug previously shipped releases), and it should be possible to retroactively cause a branch to occur at a label (in case a patch must be created to a specific previous release). The latter is important because it should be a hard and fast rule never to ship any software that was not built from source control. The practice of syncing to a label, making a quick change on the local machine (without checking the change back into source control), re-building on the spot, and then shipping a quick patch file (*e.g.*, a newer version of a dll), should never be used. In order to make it practical to eradicate this practice, branching from a label is important.

9.2. Uses For Source Control

A good source code control system helps a software development organization in numerous ways.

9.2.1. Repository

Most importantly, a source code control system acts as a repository for the source. Rather than being scattered across numerous hard drives on various developers' computers, the repository keeps the most up-to-date revisions of all the source required to build, ship, and document the software system. This eliminates ambiguity in where to look for a file. It also ensures the safety of the source. The source code control system should be implemented on RAID disks and have full backups done. With frequent check-ins, the organization limits its vulnerability to a failed hard-drive on a coder's workstation.

9.2.2. Structure

The directory structure within the source code control system helps define the modular structure of the software. All non-trivial software systems store their source files in a complex directory structure that promotes logical grouping and eases the task of finding particular files. The source code control system becomes the definition of this directory structure.

The directory structure is important as it is often used to define the module structure of a complex software system. For example, all files stored in the **DeviceDrivers** directory would be sources for device drivers. Sub-directories within that directory would be used to group the source files that implement each individual device driver. If a

developer were to store the code for a new device driver elsewhere, it would be hard to find and would clearly be a violation of the module structure of the software system as defined by the directory hierarchy.

The module structure is important not only for grouping related code, but also to provide a basis for defining architectural rules. For example, it may be a violation of an architectural rule for code within a device driver to call code within a graphical user interface module. When source code files are appropriately grouped into modules, such rules can be stated in machine-readable format, and simple tools can be built to ensure compliance (*e.g.*, a simple parser that parses only **#include** directives).

9.2.3. History

A source code control system maintains complete history of all changes that occur to sources. This is useful if a developer, for example, makes an accidental check-in with negative consequences and would like to revert to a previous revision. It therefore gives confidence to developers to know that none of their changes are irrevocable.

The history is useful for tracking down defects. For example, if a defect is discovered on a Tuesday morning, and it was known to be absent Monday morning, a developer can review all changes to the sources between Monday and Tuesday morning and quickly hone in on the likely candidates for changes that caused the regression.

History is useful for software archaeology: understanding how a system got to be the way it is. For example, if a coder is asked to modify a particular file but does not understand a segment of the code, he can go back in history and find where the code was first introduced and by whom. If the original coder is still accessible, she can be asked.

If not, it is possible that check-in comments identify the reason for the change, and this can provide additional clues.

9.2.4. Control

Unsurprisingly, one of the features of a source code control system is to provide control over what gets changed.

With such a system, a manager can review all of the previous days check-ins to ensure that all developers' work is aligned with corporate priorities. In some more strict processes, changes to the sources must be approved before they are allowed in.

Automatic controls are also available. For example, tools can be run to monitor changes for adherence to coding standards, to prevent changes that are, for example, too high on some source code complexity metric, and to enforce architectural compliance (e.g., no references from the **DeviceDrivers** module to the **GUI** module).

In conjunction with a defect/feature tracking system, management can enforce the stricture that all changes are associated with either a defect or an in-plan feature. In this way, only approved changes will make their way into the code. Such control is especially needed if the culture is a "cowboy" culture where coders are used to doing as they please without guidance from corporate priorities as to where they should spend their time.

9.2.5. Collaboration

A source code control system allows many team members to collaborate on a project. Changes by one developer can be brought into another developer's workspace on demand. This enables two developers to share their work quickly and conveniently.

Contrariwise, If a developer (or a team of developers) needs to work in isolation on a feature until they are ready to integrate back into the main flow of the source code, that is possible as well.

Multiple developers may work on the same file simultaneously. The first developer to return the file does so in the normal fashion, subsequent developers will be prompted to merge their changes with the other developer's changes, with the assistance of merge tools.

9.2.6. Multiple Streams

A source code control system allows multiple streams of development to occur simultaneously. For example, release **R2** can be readied for shipment while aggressive new feature work is being performed on release **R3**. Likewise, post shipment, point releases and patch release of **R2** can be worked on while developers continue to work on **R3**.

To allow for this, the source code control system must make it convenient to make a defect correction in one of the source streams and then apply the same (or a similar) change to the other source streams. When the files are unmodified between the two streams this is not an issue. When the files differ, the same tools required to have multiple developers merge their changes into one file may be used to merge the changes from one stream into the other.

9.2.7. Reproducible System State

When a release of the software is shipped, the source code control system allows the precise set of source files that went into creating that release to be labeled. This is useful for two reasons.

Often, a defect reported in the field can be reproduced on the most current build of the system. If not, then there is a dilemma. Do we assume the defect is now fixed, or do we assume that the defect is still latent in the current code but, because of other changes, is no longer reproducible by the same means? Unless developers have good reason to believe the former is the case (*i.e.,* the exact symptoms identified were explicitly fixed in a previous defect correction), it is wise to assume the latter. In this case, the best course of action is to debug the problem using the precise set of source files that went into creating the system in which the defect was identified. This is the first reason why labeling is useful

The second reason is for patch releases. If a defect correction is sufficiently critical to customers, it is wise to get a correction out very quickly with high confidence that nothing unexpected will break. The fastest way of doing this is going back to the exact source code that was used to build the release, apply the minimal code changes to fix the problem, test intelligently around those code changes, and then ship a small binary patch to the system. With poor systems, one is never 100% sure that the sources being used were the precise ones used to build the original release (that underwent original extensive testing), and thus one is loathe to ship a quick patch in the manner described. With good systems, this is not a problem.

9.2.8. Coder/Build Communication

The source code control system exists as a handoff point between development and QA/Build.

The organization must make a hard and fast rule that no software (including even small utilities or small patches) should ever be shipped

unless it comes through a build and QA function. With such a rule in place, coding can concentrate on coding, and the QA/Build function can concentrate on testing, packaging, shipping, licensing, inventory, keeping track of which releases are extant, communicating with client services, and so on.

Mitigating against this is the reality that there is considerable back and forth between the QA/Build team and the coding team prior to any release. Once the n^{th} iteration has been reached, and a coder has on his desktop the exact working executable that will solve a customer's problem, the temptation to "just ship it" is strong. This temptation must be avoided at all costs.

A likely scenario, were this to occur, would be that the coder emails off the desired changed executable, the customer is now happy. Having worked eighteen hours straight, the coder then goes home. Hopefully, next day the coder will remember not to make any subsequent changes to the source files before checking them in. It's now a month later, and the customer complains of another problem. The help desk cannot reproduce the problem. They use the exact same release ("release 3.2.9 - build 2357 - English" as reported by the "About..." dialog box), but they cannot reproduce the issue. The reason is of course, this changed executable. After several days, everybody in the team puts their heads together as the customer is now upset. Collectively, the team remembers the patched executable. They attempt to recreate the source code for the build the customer has, but are unable to. The issue was that the developer had some older versions of files synchronized to his depot as he was working on the patched executable. The exact configuration used is nowhere in source control - the customer cannot be patched this time.

To prevent this type of scenario, which can waste considerable time, lead to poor quality shipments, and may prevent certain operations, the source code system must be the means (and the only means) by which coders convey their changes to the QA/Build group.

The QA/Build group should only ever build releases and patches from sources taken out of source control. As soon as they release the software, the exact configuration should be labeled. This will eliminate much potential confusion.

9.3. Codeline Policy

Establishing codeline policy is central to the use of a powerful source code control system. A codeline policy specifies the conventions whereby a source code control system is used to maintain multiple simultaneous streams of development, and the rules for who may check-in what types of changes into the system. In this section we will look at a codeline policy suitable for a packaged software release. At the end of the section we will discuss modifications for a more continuous SaaS release cycle.

9.3.1. The Main Codeline

Generally, one starts with a **main** codeline. This is typically the only codeline into which new features are coded, and only at certain points in the release cycle (between fork and dcut). Defect corrections are coded into this codeline at any time, and all defect corrections (if applicable) must be checked into this codeline. The main codeline is consistently used for ongoing development, across all releases of the software.

After dcut new features are no longer allowed in the main codeline; only defect corrections. This persists as long as possible up to GA. However, often before GA the software stabilizes sufficiently, and coders need more features work to do. When this happens, a branch should be created off main, and named for the release (*e.g.*, **R3**). As soon as the branch is created, coders may begin again checking-in new features into the main codeline.

The timing of the main codeline branch is a delicate thing. Too soon, before the next release is stable, and developers must fix too many defects in two places rather than one. Too late, and coders wind up doing new feature work off to the side, in private branches, and have a difficult and error-prone merge later on to re-integrate their work with all the defect corrections that have since taken place.

The best rule is to delay as long as possible, until the branch is definitely needed, but then not to hesitate to branch. For example, if a coder begins work on a new feature, but the new feature's code is very disconnected from the rest of the code, it may be reasonable to have that coder work within their own private branch, as the subsequent merge will not be overly painful. However, if a new feature cannot be delayed any longer and requires work to existing files, then the branch should be made.

9.3.2. Maintenance Codeline

The codeline created from a branch from **main** under these situations is called a **maintenance** codeline.

Developers typically have free access to the maintenance codelines. The rule is that only defect corrections go into a maintenance codeline, no new features. Furthermore, all applicable defect corrections must go into the *active* maintenance codelines. As opposed to active maintenance codeline, *retired* maintenance codelines are codelines for previous releases that are now off maintenance and are no longer being supported. Software companies should make every effort to retire maintenance codelines at the earliest opportunity.

As point releases get closer and recede, the codeline policy for defects may vary. For example, as a point release nears, the rule may be that only high priority defects may be fixed and checked into the maintenance codelines. As the point release recedes, the codeline policy may become more open, and any defect may be fixed and checked in.

Usually, software is not shipped directly from maintenance codelines. When nearing a release date, another branch is made from the maintenance codeline called the **shipping** codeline.

As with the first branch off the main codeline, the timing for branching shipping codelines from maintenance codelines is delicate. Too soon, and there is a lot of defect merging to perform. Too late, and lower priority defects may not be worked on at all for a time.

9.3.3. Shipping Codeline

The **shipping** codeline is used by the QA/Build group, and they must maintain tight control over it. If the system allows, the codeline should be locked down so that only build group members are allowed to make changes to this codeline, not coders.

The rule for the shipping codeline is that only the most critical, last minute defect corrections should go into it.

The reason it is required is to promote stability immediately before a shipment. Typically, testing has put a lot of work into validating the build that is about to go out the door. Each new defect that is reported is carefully triaged to determine if it is of sufficient importance to merit re-starting the testing cycle. Given a 2-day final testing cycle, if all newly discovered defects were to get added to the release, the release would, quite literally, *never* get out the door, as chances are almost 100% that some defect will be discovered in any 2-day period.

If a defect of sufficient importance is discovered, it will first be fixed in the **maintenance** codeline and merged into the **main** codeline. The QA/Build group will then very carefully merge the change from the maintenance codeline into the **shipping** codeline (generally with an anxious developer looking over her shoulder), and only limited re-testing will be performed around areas of potential concern given the nature of the change.

The shipping codeline is also a place to put in trivial last minute changes to less critical components, such as splash screens, samples files, release notes, and so on.

Any required patch releases can be shipped from the shipping codelines.

9.3.4. Private Codeline

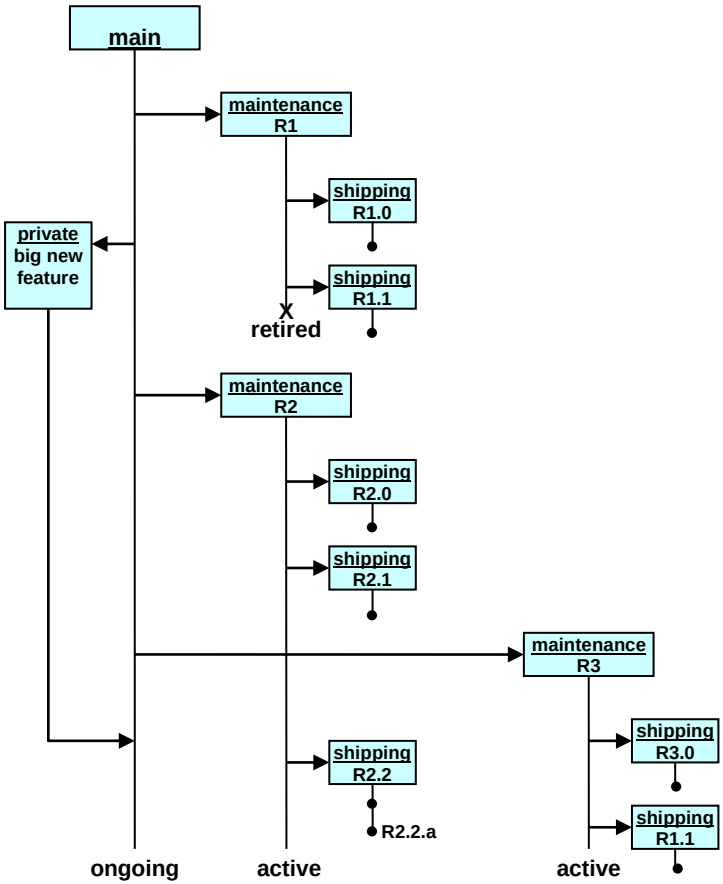
Finally, **private** codelines, alluded to earlier, are used by individual developers, or small groups of them, for feature work that occurs outside of the main release cycles. For example, if a new feature would take longer than the standard release cycle, a private codeline would be branched off of main (or just a portion of main, if possible) for performing this work. At the start of the release cycle in which this feature is due to be shipped, the private codeline will be merged back into main.

9.3.5. Example

The diagram on the page following illustrates how codelines are used. A maintenance codeline **R1** was branched off the main codeline. Two shipping codelines, **R1.0** and **R1.1** were branched off of it.

Similar situations occurred for the currently active maintenance codelines, **R2** and **R3**. In the case of the shipping codeline **R2.2**, two builds were shipped from it, the second a patch release designated **R2.2.a**

Meanwhile, a big new feature was being worked on in parallel, due to be shipped with an upcoming **R4**.



9.3.6. SaaS Codelines

In the case of the more continuous release approach typical of SaaS, we would typically use three codelines: *main*, *maintenance*, and *production*.

Production is a "trivial" codeline. It is always an exact reflection of what exists on the production servers. We build check in rules that will automatically check out anything we put into this codeline onto the production servers. So it is mainly just an interface to a deployment method.

We put emergency defect corrections into the *maintenance* codeline. It acts as a place for multiple parties to collaborate on a fix, and a place to gather up a small number of disparate defect corrections prior to pushing to production. We set things up so that code put into *maintenance* can be easily subjected to automated regression tests. When we are ready to deploy to production, we will integrate all changes from *maintenance* into *production*. There should never be any conflicts when doing this as nothing gets directly checked into *production* without going through this route.

Developers code new features into the *main* codeline. Defect corrections put into *maintenance* must also be backwards integrated into *main* so as not to lose them. If the company has multiple teams, it is advisable they all collaborate within the *main* codeline to ensure that no later integration problems arise.

When it is time to push new features from *main* to production, either the entire *main* codeline, or parts of it, are pushed to *maintenance*, and from there to *production* where they will be mirrored onto the production servers. If there is a problem with the code, the *maintenance* codeline can be reverted and pushed back to production.

9.4. Builds and Installs

A *build* of a software system involves running build tools against the sources to create executable files (and any other support files, such as static databases). The end-result of a successful build is a set of files, gathered together in a convenient location that may be installed onto a computer (or several computers) to enable end-users to run the applications and servers comprising the software system.

Once the files are generated for a build, a subsequent step involves creating an *install image* for distribution. An organization will typically use an installer tool to create an install image. The install image will be burned onto optical media for distribution, or posted on a web site for download. When the end-user acquires the install image, he will run the installer script which will unpack the files, copy them to the appropriate locations in the file system, and add any system entries required (such as registry settings). The application should then be ready for use.

Installers may be created either for complete releases, or for patches that bring software from one release up to a newer release. Tools used for patch creation can be different from tools used for a base install. Good patch tools will compute binary differences to files, and include only the minimum necessary information to bring software from one release to the next. This is desirable if the patches are delivered over the Internet (as they almost always are) in order to minimize download times. Ideally, the software will be built with a facility to query the company's servers for patches, download them, and install them automatically.

All of the files used to feed the build and the install tool must be kept under source control. As well, any custom tools required to perform these steps should also be kept under source control. Most organizations would exclude the basic tools, such as the Integrated Development Environment, from this rule for convenience's sake.

Most importantly, the source code for the scripts that perform the builds, perform the creation of the install images, and burn the images to CD should be kept under source control.

It is unacceptable to have any manual steps in the build and install creation for three reasons.

First, it creates an exposure. Often knowledge of the steps will be held in the mind of a single individual. If that person is away or leaves the company, software cannot be shipped. This can be mitigated by documenting the steps. However, the correct steps for builds and install creation rapidly change, and documentation risks getting out of date. Better in these situations to "document" the steps in an executable form. One is then assured that the "documentation" and the reality are kept in-line with one another.

Second, builds and install creation are notoriously error-prone. If files are being copied manually, it is easy to accidentally pick up the wrong copy of a file and use it. If this occurs, because the file was not taken from the correct location in the depot, the company is shipping a release for which the sources will not be available in the depot. As we have seen previously, this can cause distress, and may introduce last-minute errors that are not tested for.

Finally, using scripts that use only sources from the depot ensures consistency in the builds. Without this, it is possible for QA/Build to discover a problem with the software. The defect is then reported to the coders, but they cannot reproduce it. The cause is often inconsistent builds. The QA/Build team is using one approach to building the software, the coders are using a different approach. This can cause delays in shipments as the problem may be very important to the QA/Build team, but the coders may not take them seriously because they don't see the issue at all in their builds.

There are two types of builds and installs. One type is development-oriented, the other end-user oriented. In the next section, we examine the development-oriented build, and the one following the end-user oriented build.

9.5. Development Builds

Because end-users only ever see a fully installed application, the most accurate testing and debug environment would be a full install. However, these are so inconvenient and slow to work with that organizations take a tiered approach. The first tier is the development build (which includes a mini "development install").

It would be inconvenient for coders to track down defects based on installed software. Coders maintain a development environment whereby (ideally) they can run one command to create a build and run the executable. These builds contain copious debugging information and switches which enables the developers to single step through the

source code, examine variable values, do memory checking on all allocation and deallocations, dump tracing information, and so on. These types of "debug builds" are never shipped to end users, both because they run inefficiently and because they contain enough information to enable hackers to reverse engineer the code.

If a problem manifests in a debug build, it is usually an easy and quick fix. The vast majority of problems fall into this category. If they do not, then chances are good they will manifest in a "release build". A developer can usually turn a switch in their development environment and create a release build. The release build is much closer to what end-users will run on their systems, but are more difficult to debug. Typically, binary debugging must be used to examine memory locations, which is tedious work. Fortunately, only a very small (but finite) proportion of defects fall into this category.

Occasionally there are problems that manifest in an installed application but not in a release build. These problems are almost always due to a faulty install script, which is usually under the purview of the QA/Build group, and not the coding group.

For this reason, coders will want to work as much as possible with debug builds. QA/Build should be running tests against release builds (because occasionally problems show up in release builds but not in debug builds, and release builds are shipped to customers). Mostly this works fine, as the vast majority of problems found by QA/Build will be reproducible by the coders in their debug builds.

The first tier of testing and stabilization therefore takes place on development builds and development installs.

Developers must create a build and development install system that with the execution of a single script performs all the necessary build steps, moves files into appropriate locations on their machines, and does the minimal necessary system settings to allow the application to run.

QA/Build will then use this script to make their testing builds leading up to a release. Because testing and coding use the exact same build scripts (with the only difference being a switch to "debug" for coders and "release" for testers), there is a high degree of confidence that problems found by testing can be reproduced by the coders.

As well, as we shall see later, automated regression tests should be run against the same development builds (in release *and* debug modes) so that problems found using these automated tests suites can easily be reproduced manually by both testing and coding.

9.6. Production Builds

A production build generally starts with a release-mode development build, but then carries it further to create an install image and possibly even an ISO image (for direct burning to a CD) or a downloadable image. This may be used either for a full install, or for a patch.

Production builds are what are eventually shipped out the door. A final "release checklist" is carried out against a candidate production build to ensure that all major functionality (and especially functionality that if broken cannot be patched) works as intended. This is usually a manual process that takes one or two days. While automated regression testing is a necessary part of a testing program, there is no substitute for

human common-sense as a final check lest we create 20,000 CD's of "coaster-ware" because the machine said it was ok!

The shipping of production builds should follow a defined workflow process. We will discuss tools to assist in building and controlling workflow processes in Chapter 11, "Defect Tracking" and Chapter 12, "Feature Tracking". A similar workflow management tool can be used to track production builds through development build, testing, install image creation, final testing, and CD mastering.

The system should keep a list of all the production builds (including both releases and patches) the company is intending to ship out the door in the foreseeable future. A regular "release priority" meeting involving senior management should occur that prioritizes this list, attaches dates to the items on it, re-prioritizes as required, and checks up on progress.

The list should also include scheduled production build candidates. For example, for a new major feature release of a product, QA/Build might recommend that there be weekly trial production builds for the six weeks prior, with the last build being intended as the "Gold" build (the one that will make it out the door). For a point release, perhaps one or two trial releases should be scheduled.

The list should be stored in the workflow management system, and automatically posted on a corporate intranet site so that all parties have visibility against progress.

9.7. Automated Builds

With many developers making changes to the software code, and with these changes possibly affecting many different versions and codelines, there needs to be a mechanism of catching any build problems early on. This is especially important when there are many different versions of the software, as developers will typically only ever test on the version they are developing against. Introducing build errors into the depot is called "breaking the build".

To minimize build problems, before checking in any changes coders should synchronize their sources to the latest files and attempt a re-build. If the software builds and tests out, they should then check in their changes. The astute reader will notice that there exists a race condition here. Two coders may be doing the same thing at the same time, with the result that nobody will have tested the system with both sets of changes simultaneously. These race conditions introduce the possibility of breaking the builds. The other (and more common) reason for breaking the build is coder carelessness.

To guard against broken builds, every night a scripted process should fire off, completely delete all of the sources, intermediate files, and final files from the previous night's runs, and return the system state to pre-install. Then all the sources should be automatically checked out, both debug and release development builds fired off, and then development installs performed for each. As the number of versions of the software increase, each version can be built and installed in this manner (possibly using a swarm of computers).

The build process should report its progress, as it goes, into a relational database. Detailed output from the build tools will typically

collect in text files. These files should be captured, automatically analyzed to count errors and warnings and extract the messages, and copied to a server. An Intranet web-app can then be built that can report on the progress of the build, and on the final results. Especially important is a summary page indicating what builds were made, and giving the numbers of errors and warnings on each.

Each morning, developers should check the build results, correct any problems and re-initiate the automated builds if necessary. The automated build environment should provide a means via the results page on the intranet of drilling down to the exact error or warning messages, and then viewing the code that the error or warning refers to. Developers responsible for "breaking the build" by checking in erroneous changes should be punished in some creative fashion (*e.g.*, they have to come in early every morning to check the build until the next developer breaks it!). These builds should occur against both the main codeline, and all active maintenance codelines.

Using automated builds, with easily queryable results, and having strict social censures against breaking the build are necessary to maintain the integrity of the automated build system.

9.8. Summary

In this chapter we looked at some of the most important core infrastructural elements required to control software development in a commercial software organization.

We started by describing the attributes of good source code control systems, the ways in which they benefit the organization, and the codeline policies used to control their use. We then discussed the need to automate builds and installs, and the difference between development builds and installs and production builds and installs.

In the next chapter, we will discuss the next logical infrastructural elements: testing.

10. Testing

In this chapter we discuss the various types of testing that take place during a release cycle, and concentrate especially on automated regression testing that enhances the automated builds discussed in the last chapter.

Humans make mistakes. It is generally fruitless to try to improve quality by trying to tackle the problem at its source; preventing mistakes in the first place. Rather, the most fruitful areas for improved testing are in instituting processes and practices that double and triple-check the work of fallible humans.

There are two general types of checks: testing and reviews. Testing means running tests against the software. Reviews means examining documentation and/or source code produced by developers. In this chapter, we will concentrate on testing as it comes first. If software development does not have an effective testing program, spending time and money on reviews will be less effective than spending it on testing for the same gains in quality.

There are many different types of testing that take place during the coding phase. While the terms used to describe these testing phases vary, the ideas are common. We will organize this chapter around the various types of testing that can be done. We end with automated regression testing, and devote some considerable space to discussing this all-important type of testing.

10.1. Unit Test

When a coder is first coding a feature, generally a tester will not get involved until the feature is relatively complete and relatively stable. While this is going on, the coder is doing their own testing of the feature. This testing is called *unit testing*.

Unit testing may be testing that takes place within the context of the developer's private copy of the application, or it may involve the coder building *test scaffolding* to more thoroughly test elements of the code that they develop. A test scaffold is a small, stand-alone application intended to test an internal API the coder has developed as part of the overall solution. Ideally, these unit tests are also checked into the depot, and a facility is provided whereby they are re-built and re-run every night as part of an overall automated testing infrastructure.

10.2. Component Test

Once a feature is relatively complete and stable, an independent tester will test the feature with reference to the feature specification. This is called *component testing*.

Initially, this testing will be manual, using the full application. The tester should never use a private build for this testing. Rather, he should always use the release version of the nightly development build, and test across the various different versions of the software. In this way, we guarantee that the tester is testing what will eventually appear in the application, and not code that only accidentally appears in a developer's private workspace. This practice also encourages maintenance of the automated build facility and the frequent checking-in of changes.

During function test, the tester will be constantly feeding back problems encountered and ideas for improvement to the coder. As the feature assumes its final form and begins stabilizing, the tester should be thinking in terms of how to develop an automated test script to test the functionality. In the absence of this, or in addition to it, the tester should be writing down the sequence of manual tests that should be performed to test the feature thoroughly.

During function test, the coder should also be working on testing API's that integrate to a regression testing environment (more on this later), and the tester should be working on test scripts that use this API.

10.3. Integration Test

After dcut is reached, the new features and the old features should all be re-tested in light of the interactions between the various new features that could not be uncovered during component test. This test phase is called *integration test*. During integration test, any automated tests should be finalized and checked into the system.

10.4. System Test

When the system nears completion and the production install scripts are ready, the testing group will test the software using full installs, and test for how this new version interacts with previous versions and complementary software. This is called *system test*.

10.5. Final Release Test

Finally, immediately before release, a final checklist must be gone through that verifies all the most important aspects of the system. This is called *final release testing*.

10.6. Automated Regression Testing

Running across all this testing, and providing an underpinning for it, is the most important type of testing: automated regression testing.

"Regression testing" is so-named because it is intended to check for "regression" in capability: things that used to work before that now no longer work.

In practice, regression testing cannot be performed manually. For most software there are so many things to check for that humans would take months to run all the necessary tests.

Some organizations perform regression testing by examining all the previous defects reported against the software and verifying that those defects have not re-appeared. This is a fertile source of defects for the test team. Things that are broken once have a tendency to break in the same manner again. Historical defects are an important source of regression tests, but are only a part of the complete picture.

A good regression test should be entirely automated. It should be a natural extension of the automated builds. However, whereas coders will check and correct the build results, it will the QA/Build group that checks the results of the regression tests. This is because many problems surfaced by automated regression tests are not defects.

Human judgment is required to determine if the deviances of results from a previous run are a defect, an improvement, or a test error. Any genuine defects can then be reported to the coders.

As for automated builds, the regression tests should similarly report their progress and final results into a centralized relational database. Web applications can then be built to query the results and drill-down on deviances.

Whenever new functionality is added to the application under test, new regression tests should be produced for testing that functionality. Thus it becomes the focus of the test team not to only test manually, but to test manually with the intent of understanding how to build a good suite of automated test cases.

Whenever a defect is reported against the software, chances are that whatever triggered the defect was not captured by the automated regression tests. Thus there should be a rule that a part of resolving any defect is to ensure that going forwards there will be a regression test that can catch that defect. In this manner, defects will not re-appear (which often happens without this practice).

10.7. Performance Regression Test

The automated regression tests should not only check for correct operation of the software, but should also check for *performance regression*: *i.e.*, check if the software is running slower today than yesterday.

This is important because sometimes a coder will check in a change that fixes a problem, but badly deteriorates the performance of the application. Because the developer will typically only check with one

example, or a small example, the performance regression may not be immediately visible.

For example, if an operation that usually takes 0.01s takes 0.1s, the developer may not notice this at all during her testing. Similarly, if the automated nightly regression tests only perform 20 or so tests on this operation and we do not explicitly track execution time, we will also not notice the change. If we ship this software and a customer has a batch script that performs this operation 1,000,000 times, the difference in performance is from 3 hours to 30 hours. The 3 hours comfortably fit into an overnight window. The 30 hours assuredly do not!

Performance regression requires that we measure the time taken to do specific performance tests. Measuring the time to perform individual regression tests (or the whole suite of tests) is not adequate as the time may be dominated by test system overhead. Therefore testers must design specific tests that are not dominated by testing system overhead. These tests must then be measured, and the performance numbers made available individually, by test category, and aggregated into some sort of an overall performance metric using weights to stress the importance of one aspect of performance over another.

The performance data must then be kept historically and made comparable over time (using normalized measures in some way). Only in this manner will we spot trends. Developers will also be able to spot the exact day performance went awry, and will be able to use the source code control system to track down what changes were made to the code on that day to possibly account for the slowdown. These results can also be used as a metric if the company decides it needs to spur the coders to improve the performance of the application.

10.8. Memory Leak Test

Once functional and performance regressions are in place, we must next look to memory leak regression. Every night, tests should be run to verify that the application is not leaking memory, or, if it is due to a component out of the company's control, that it is not getting any worse. This can be as simple as monitoring virtual memory used during tests, or special tools can be used that check for improper use of memory including memory leaks and point as directly as possible to the problem.

10.9. Benefits of Regression Testing

Regression testing is important because it helps lock in quality. Once component testing and integration testing is passed, we can have good confidence that the feature works as intended. However, the biggest problem with leaving things there is that as other development and defect corrections proceed, existing features, and especially newly created ones, are delicate and tend to break.

Because coders and testers are focusing on the new features and the new defect corrections, nobody is paying attention to features that have previously been determined to work. It is very common, therefore, for these other features to regress.

Automated regression testing means that the machine is always checking and re-checking this existing functionality, and will raise the alarm when and as soon as it breaks.

Raising the alarm early is another benefit of automated regression tests. Defects are less expensive to fix the sooner they are found after having been introduced. This is because developers have fresh in their minds what they just did to break the software. If something is shown to be broken the very night of a big check-in by a coder, chances are good that it was the big check-in that broke it.

Automated regression testing is a development aid. When coders must make changes to inherently complex, hard-to-understand, central parts of the code (*e.g.*, a main simulation loop, for example), it is beneficial to have an exhaustive suite of tests that can be run against the application to help assist the coder in making sure nothing unexpected was broken.

Without such a facility, coders become loathe to change these central parts of the code, whereupon hacks are accumulated around it to deal with specific issues without upsetting things that are known to work. This leads to rapid architectural degradation and ultimately defect-laden code that "can’t be fixed".

Regression testing is especially important as we near the final stages of a release. Say the release date is upon us and a number of distributors and/or customers are anxiously awaiting the new CD. At the eleventh hour a show-stopping defect is found. The correction looks dangerous, in that we are unsure what other functionality it may break. Without a complete set of automated regression tests, the testing cycle for this change might be days. The organization is then faced with a dilemma; should it leave the defect in (not acceptable), should it delay the release by another 2 or 3 days (not acceptable), or should it take a chance and

just do some cursory testing around the change. Often, it is the last option that is selected, with potentially disastrous consequences. With a full suite of automated regression tests, within a few hours the organization can build confidence in the fix, and greatly reduce the dilemma.

10.10. Regression Coverage

Central to managing a regression testing progress is having some metric for *coverage*. Coverage is an estimate of the completeness of the regression tests. An unambiguous metric is the percentage of the lines of code that are executed during a regression test run. Profiling tools may be able to give this metric. Other metrics may be more human-oriented. For example, one may list all the functions that need to be tested, and then put a check-mark beside each one that has one or more automated regression tests. Generally, automatically-produced metrics are best, as they cannot be argued with.

It is not critical that the metrics actually represent absolute coverage. What is important is that an increase in the metric should correspond to increased coverage. This provides management with a simple metric to determine the rate at which progress is being made, and to detect any backsliding in the face of new feature additions. Progress against a metric also provides the basis for a business justification for increased resourcing if that is what is required to properly staff the test group.

There are two types of regression test results: *baseline* and *validated*. Establishing a baseline means that the software has not changed its behavior from yesterday to today. It does not say if the behavior is

correct or not. This has considerable value in itself in that it fulfills the basic function of regression testing: that once something is working it does not cease working without someone being made aware of it.

A validated baseline goes a step further. Not only is the software consistent from day to day, but the results it produces have been validated to be correct by human examination.

The reason for the distinction is that with many automated regression environments, increasing the number of test cases can be done by adding pre-existing test files. In most organizations, this first step can be done mechanically and will bring value in that the regression tests will catch any core dumps and any subsequent changes in behavior.

The QA department can come along later and validate the results produced.

Baseline results can be established quickly and automatically. All that needs to be provided is the input to the test case; the regression testing system will cause the software under test to generate the output. If the system finds no previous baseline to compare against, it can automatically save these first results as the baseline going forwards.

Thus, if a certain operation on a certain input file fails, then once it is fixed, that input file (possibly originating with a customer) can be deposited into the test bin and baselined results obtained easily.

Thus, two coverage metrics can be produced: baseline coverage and validated coverage.

10.11. GUI Versus Scripting

In practice, regression testing requires an ability to run the software under test by remote control. There are two general approaches to this, which can be complementary. One approach is to drive the user interface by simulated mouse and keyboard input (GUI-level testing - GUI = graphical user interface). The other is to interface to a scripting layer in the software.

The GUI-level testing approach can be problematic. GUI's are designed for human interaction. If, for example, something goes wrong the user interface may put up a dialog box requesting confirmation before proceeding. The GUI-level test scripts may not be aware of this, and may continue trying to press buttons that are not operative until after the dialog is dismissed. Or, a GUI may reconfigure itself under certain situations (*e.g.*, adaptive menus). Unless the GUI script is completely aware of all the details of this, its mouse clicks may be "off".

With GUI-level testing, output is sometimes only graphical. The tools can do a bit-by-bit comparison of the graphics, and signal errors if anything is different at all. This may be overkill.

The problem with all these examples is that the chances for "false-positives" are too high. There are many situations where the output may differ by a few bits (e.g., a subtle change of color), or the menus may be slightly re-arranged, or an extra confirmation dialog may appear, and these are not considered errors. A human tester would pass over these easily and continue testing. GUI testing tools will stop dead in their tracks and report an error.

As well, the output of these tools is usually difficult to get at and customize and report on in a fashion that suits the organization. The tools are typically very expensive. The large number of staff needed to maintain the scripts in the face of changes to the software is also very costly.

The other approach is to architect into the software a scripting API that can be driven by various scripting languages such as Perl, Python, VB, and so on. On Windows-based systems, providing a COM API is an excellent way of doing this.

The COM API should be designed from a testing perspective. It should cover all functionality that needs to be tested and produce output in a textual form that can be easily differenced against previous results. Errors can be returned by means of error codes from the API that allow the script to continue.

Building a regression testing API at this layer bypasses the GUI. This has both a negative and a positive connotation.

On the negative side, because the scripts are not doing exactly what the end-user would do, there is a chance of errors slipping by. As well, with an API every new feature must expose its functionality both as a GUI and as an API, which is extra work for the coders. Finally, writing test scripts against an API requires programming skills, whereas using GUI-testing tools do not.

On the positive side, many false-positives are bypassed by not testing the GUI layer. The organization can be more free in changing the GUI without breaking all the regression tests. Providing a scripting API will improve the architecture, as it will promote a division between

GUI-independent functionality and GUI. Finally, and most importantly, the integrity of the scripting API can be maintained in the face of dramatic changes to the software. This implies that once a test script is written, it will work forever without modification so long as the API behavior remains consistent. This last consideration trumps all others, and is the chief reason for favoring an API-based regression testing approach.

With SaaS, it is often critical to perform GUI-based tests, as considerable functionality of a web application is contained in the executable JavaScript code than runs in the browser. The open source framework Selenium is often used for in-browser testing, augmented by other tools that can manage and report on the test progress. There are a number of cloud-based services that run these tools against a URL target you supply, which is an ideal application of cloud computing owing to the bursty nature of the computing resources required.

Even so, these tests are subject to the same caveats as any GUI-based testing, and while necessary, if there is some testing that does not involve JavaScript or GUI elements, it is wisest to also have a regression testing framework that does not involve any GUI elements, as these tests will be far less work to maintain.

10.12. Regression Testing Architecture

Regression tests should be runnable on a developer's computer, and on dedicated computers used for nightly tests. The developer needs access in case the defects discovered cannot be reproduced in a standalone fashion, and to enable them to run a suite of tests when they are working on dangerous code. When run on a developer's computer, the testing infrastructure should not report progress into the central testing results database, but rather report them locally (into a file, for instance).

The regression testing infrastructure should be cross-platform, meaning that it can run on any conceivable platform that the application under test may ever run on. For software that has many versions running on many different OS platforms, using virtual machine technology for the testing is a cost-effective and convenient way to proceed.

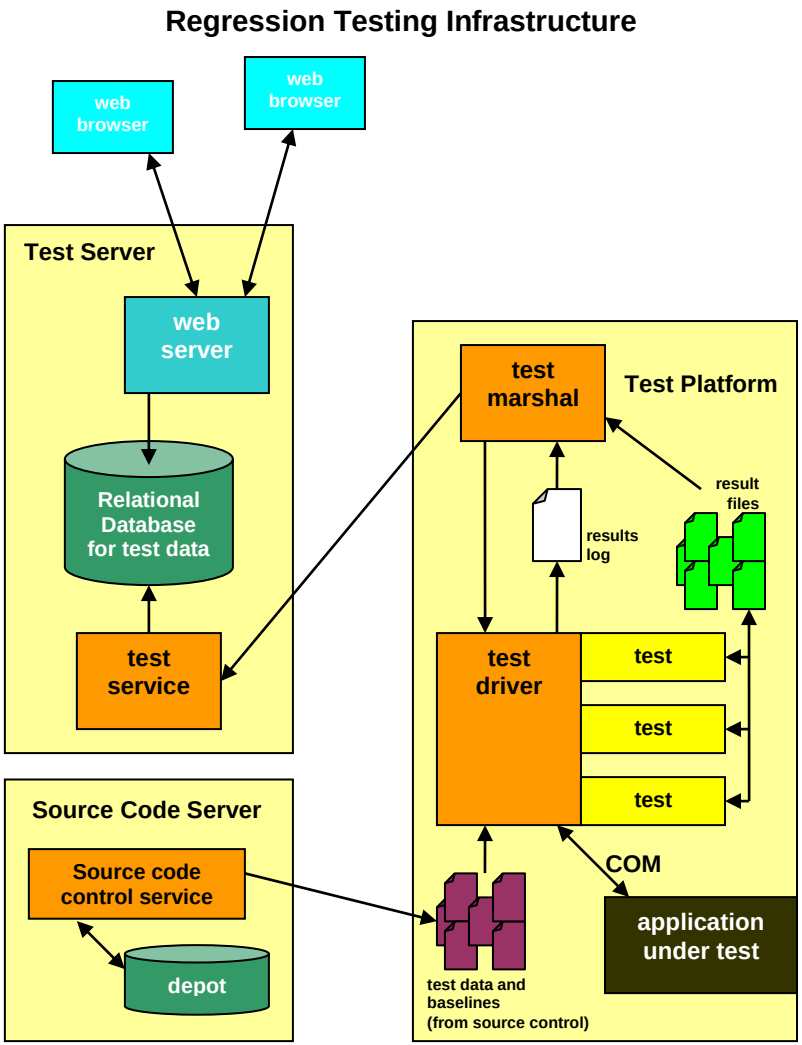
Scripts for individual tests should "plug into" a framework architecture that can run defined sets of tests and provide support functions (such as starting up the application, doing diffs, storing raw and processed results, and so on).

The framework should provide for extreme fault tolerance. Because we are testing an application under development that we expect may core dump or hang, the infrastructure needs to be clever about constantly monitoring the state of the application and recovering from core dumps and timing out hung applications in order to complete the test suite. In other words it is not acceptable to stop the test suite after the first error. Having results for all tests help greatly in tracking down problems (even if it is the case that every test core dumped!).

If a test fails, it may be that the test always fails, or that it only fails because it is in sequence with other tests (that may have succeeded themselves, but caused some memory corruption that caused future tests to fail). It is very valuable to detect this latter type of error, as these are typically hard to track down errors that plague the customers. For this reason, the testing infrastructure should as much as possible use the same instance of the application to run a whole series of tests. If the application fails on a particular test, the infrastructure must be capable of restarting the tests starting with the test that failed. If the test fails again on a clean invocation of the application, chances are it is a defect exposed by that particular test, and the application should then be restarted starting with the next test. If the test runs clean the second time, then it was probably a previous test that caused the problem.

Therefore, when a test fails, it is good to collect the history of actions that took place leading up to the failure in a readily accessible log file (*i.e.*, linked to the failed test result on the web app).

The following page shows a sample architecture for a regression testing infrastructure.



The test server runs a relational database and a test service, to which the test marshal communicates. The test service is in charge of writing results to the database and storing results files locally for comparison to baseline files. The web server and an appropriate web application are used for reporting results. The test service may be eliminated in favor of file shares and direct communication to the RDBMS from the test marshal.

The test marshal oversees the operation of the test driver. The test driver is an application that communicates using COM (if in a Windows environment) to the application under test. A number of test suites plug into the test driver, and get their test files and baseline data from the source code control system mapped onto the test platform. The test driver dumps results to its standard output. This architecture enables developers to run the test driver on their local machines and get a quick output without all the complexity of the server environment which is not required.

The test plugins do intelligent diffs with the baseline files to determine of tests pass or fail. We require "intelligent diffs", because if, for example, an output includes a date and time, differences in this part of the output should not be flagged as errors.

The test marshal sequences the tests, parses the log output from the test driver, and monitors the test driver and restarts it as necessary if the application core dumps (which, under a COM regime, will generally bring down or hang the test driver).

10.13. Summary

In this chapter we considered the various types of testing, including unit test, component test, integration test, system test, and final release test. We ended with a discussion of the most essential type of testing: automated regression testing.

We described how automated regression testing is helpful because it allows us to efficiently re-check all functionality after any small change. This benefits quality as a whole, enables developers to make dangerous changes in core parts of the software with confidence, and enables quick response in emergency patch situations.

We continued by describing the various types of regression testing, methods of setting up systems to perform regression testing, and how to measure regression testing coverage, which is necessary when putting in place a management initiative to improve automated regression testing.

In the next chapter we look at systems to keep track of the various defects found in testing and guide the process of fixing them.

11. Defect Tracking

The previous two chapters have discussed infrastructure necessary for conducting professional software development in a commercial software vendor setting. The only significant piece missing is the defect tracking system, however this system is distinguished from the others as, when done right, it is the first step towards getting control of the software lifecycle process.

11.1. Introduction to Defect Tracking

Defect tracking is the act of keeping track of all the defects that have been reported against the software, and then keeping track of the steps to go through to correct them.

Such a system is necessary for even the most rudimentary software product. Without it, all the hard work that goes into finding defects risks being lost, and all the pain that customers experience discovering these defects in the field risks being repeated by those customers and others yet to come.

An effective defect tracking system helps ensure that coders do not start work on un-validated reports of defects rather than actual defects, and helps prioritize which defects to work on at any given time.

At its core, a defect tracking system consists of a database containing defect records, and a workflow. A workflow is driven by a **state** field, containing states such as **New**, **Valid**, **Fixed**, **Closed**, and so on; and a current defect owner, who is the person who is currently responsible for

the activities required to drive the defect from its current state to its next state.

In the next section we look at the information contained in a defect record.

11.2. Defect Information

The following types of information would be typical of those kept for a defect.

Where Found

The software product, release, and version in which the defect was first identified; and information identifying the hardware and operating system platform that was being used when the defect manifested. As well, the general area of the software in which the defect was found (*e.g.,* GUI, database, calculations, and so on). This is useful when choosing a developer to fix the defect.

Who Found It

The name of the customer and the person within the organization who reported the defect. Even if reported by a customer, there should be an employee within the organization who liaised with the customer and is able to reproduce the defect who will act as the customer's proxy.

Description of the Defect

A summary line, briefly describing the defect in a quick phrase; a more extended description of the defect fitting within a paragraph; a

detailed list of steps to go through to reproduce the defect; and any accompanying data files required to reproduce the defect.

Triage

Information that can be used to prioritize this defect relative to others. Usual data fields to capture include the severity of the defect and the probability of occurrence.

The severity would be listed as high, medium, or low. A high severity defect might be one that crashes the software or causes incorrect results to be posted. A medium severity defect might be one that prevents the end-user from accomplishing a task. A low severity defect might be a cosmetic annoyance or a misspelt word.

A probability of occurrence should be included as a separate field. Typical values might be **always**, **often**, **sometimes**, or **unlikely**.

Severity plus probability of occurrence can be used to auto-generate a priority (*e.g.*,1 to 5) from a matrix. The priority should not be exclusively tied to these fields, however, as human judgment might intervene to say a defect is higher priority than the matrix would indicate.

Audit Trail

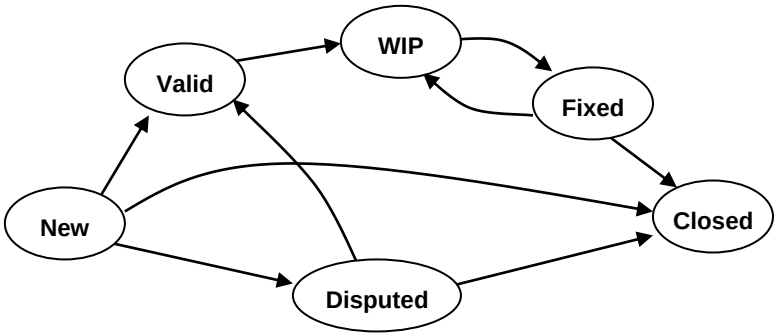
The defect record should include a discussion log with entries tagged by date and person for any discussions relating to this defect. All changes to any data fields in the defect should have an audit trail entry detailing when the field was changed, by whom, and the from and to values for the field. If there were any code check-ins into the depot made to correct this defect, these should be linked to the defect record.

State Information

The current state of the defect. The current owner of the defect. A record of who the submitter of the defect was, who the coder who fixed the defect was, and who the tester who tested the fix was.

11.3. Defect States

Defining the states a defect goes through towards resolution is the most important aspect of defect tracking. A typical state transition diagram is shown below. Note however that this is an example only, and the most appropriate defect workflow will depend on the teams involved, where there are weaknesses in the organization, and in what order we wish to address these weaknesses by implementing additional state-driven process control.



New

When a defect is first introduced it enters the system with a **New** state. The defect should not be entered into the system if the submitter knows the defect has already been fixed in a more recent release, if it is a duplicate defect record, or if it cannot be reproduced.

At this point, the defect is automatically assigned to an appropriate developer, based on the area of the software to which it applies. If the developer, upon seeing the defect, believes that a different developer would be better suited to investigating it, she will forward the defect.

The developer to whom the defect is eventually assigned has the task of verifying for himself if the defect is in fact a defect, and if it can be reproduced. If so, he will move the defect to the **Valid** state.

If the defect cannot be readily reproduced, the developer will leave it in the **New** state, and change ownership back to the submitter with a request for more information. If, after several back and forths, the defect still cannot be reproduced and the submitter refuses to close it, it should be moved to the **Disputed** state.

If the developer can reproduce the behavior, but believes that the behavior is correct, she will also move the defect to the **Disputed** state if the submitter does not agree to close it.

All defects, no matter the assigned priority, should be quickly (within a day or two) moved to either **Valid** or **Disputed** from **New**.

Disputed

If a defect is disputed in any way, it will wind up in the **Disputed** state. At the time of this transition, ownership of the defect will shift to a person in charge of following up on disputed defects (e.g., the QA manager or a development manager).

One common reason is a disagreement on whether or not the reported defect is in fact a defect, or correct operation of the software. Often in these cases, the developers will say that the software was intended to operate in this manner, and hence is not a defect. Often it will be left to a manager to decide whether to treat the issue as a defect, or as a feature request.

If the decision is as a defect, the state will be moved to **Valid** and the defect re-assigned to the developer. If as a new feature, the defect record will be closed, and a new feature request will be submitted.

Valid

If the assigned developer concurs that it is a defect, she will move it to the **Valid** state where it will await work. At this time, the developer may question the assigned priority of the defect, and request a manager to change the priority up or down.

Generally, depending upon the phase within the software cycle, developers will be given guidelines to, for instance, begin work on all defects above a certain priority; not to do any work on defects below a certain priority; and to optionally do work on defects in a certain priority band if they seem quick and easy or if the coder is working "in that neighborhood" of the code anyways.

WIP

Once a developer begins work on fixing a defect, he will move the defect to the **WIP** state, standing for **Work-In-Progress**. Having this state gives management visibility into which defects are actively being pursued, and how long they have been worked on to date.

Fixed

Once a developer believes that a defect has been corrected, she will move it to the **Fixed** state. At this time, ownership of the defect will pass to a tester to verify that the defect has been fixed. If the tester cannot verify the fix, she will transition the defect back to **WIP** and re-assign the defect back to the coder. If she can verify a fix, she will add the test to the set of regression tests, and transition the defect to the **Closed** state.

Closed

The terminal state is **Closed**. From this state, a defect cannot be resurrected. If a mistake was made and more work is required, a new defect record must be submitted.

A "close reason" will always be associated with the **Closed** state. Possible reasons are "closed - duplicate", "closed - cannot reproduce", "closed - not a defect", and "closed - fixed".

11.4. Management Controls

The main purpose of a defect tracking system is to control the defect process to ensure that low priority defects are not worked on in favor of higher priority defects or feature work, and that high priority defects are addressed quickly and completely.

To enable this, management must overview all active defect records on a regular (daily) basis using reports built for this purpose. Ideally, these reports should be available upon demand by navigating a Web browser to an Intranet URL. If important defects are seen to languish for too long in a given state, the reports should make this clear. This

allows management to discuss the situation with the appropriate individuals and take action.

Likewise, if management notices too much time being spent on low priority defects, they can also take appropriate action, such as changing the defect engagement rules or reminding developers of the existing ones.

11.5. Metrics

Another reason for having a defect tracking system is to enable metrics on the quality of the software, and on the productivity and quality of work of defect finders (testers) and defect fixers (coders).

Clean defect data is critical to these sorts of metrics. For example, if there was no process or mechanism to record reported defects that did not turn out to be valid defects, management might focus all their attention on reducing the number of defects introduced in the code with barely any effect on the metrics. This would be because the high defect numbers might rather be attributable to errors made in the finding and reporting of defects. These errors, in turn, eat into developer productivity which slows down everything. Thus we would have the situation where the developers become less and less productive, while a sequence of ineffective measures are taken to reduce defects, further eating into developer productivity. Thus clean data is important to management to allow them to identify and address the right problems.

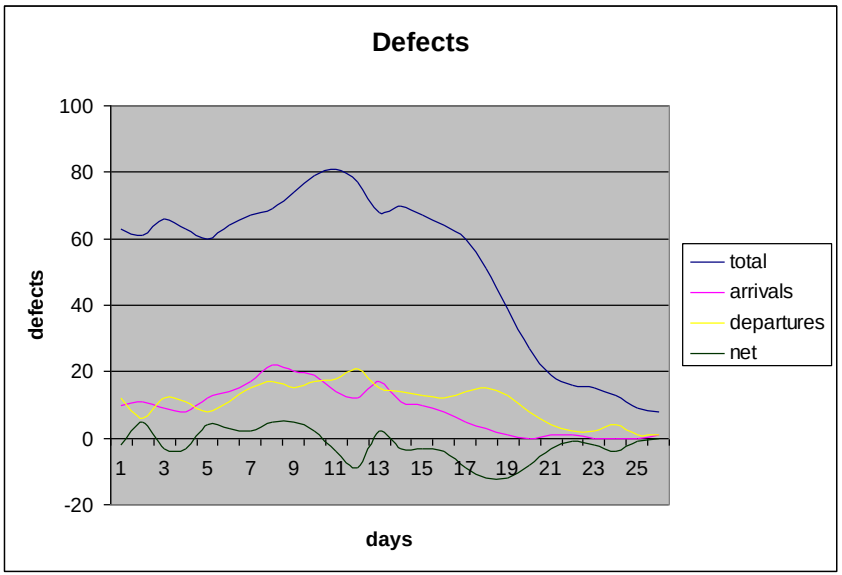
Another example of clean data is having developers quickly assess any **New** defects and move them to the next sate, and having testers quickly jump on **Fixed** defects and move them on.

With clean data, the information in the defect tracking system can be pulled out and graphed and analyzed. The three most important metrics are total valid unfixed defects, and the contributing partial derivatives: defect arrival rate and defect departure rate.

Defect arrival rate is the number of defect records per day transitioning into the **Valid** state.

Defect departure rate is the number of defect records per day transitioning from **Fixed** to **Closed** states.

Total valid unfixed defects are the count of the number of defects in any of the states **Valid**, **WIP**, or **Fixed**.



The example chart above shows a typical scenario leading up to product shipment.

All of the metrics should be available by product and by priority. A typical priority scale would be from 1 to 5, with 1 being highest priority. The organization should have threshold ship values on defect metrics. For example, a release cannot ship with any known priority 1 or 2 defects, and priority 3 defects must have an arrival rate lower than 1 defect per day. As an organization improves their software's quality, these thresholds may be ratcheted downwards in a systematic fashion.

After dcut, management will closely scrutinize the defect metrics to get a sense of whether the projected GA date is still viable. If it looks to be in danger, management may react by removing other work from developers, by focusing the most productive resources on defect fixing (*e.g.*, recruit the chief architect back into the trenches), or by requesting a few weekends of bug blitz by testers and coders prior to the ship date.

In a more extreme case, management might decide to postpone the GA date of the release.

In the most extreme case, management may conclude that the high defect rate has underlying causes in the poor architecture of the product and devote a release cycle to architectural clean-up efforts.

11.6. Relationship to Source Code Control Systems

Fixing defects and coding new features are the only two reasons developers should be making changes to the code. By tying together the source code control system and the defect tracking system (which, as we shall see in the next chapter, will also be used for feature tracking), we can go most of the way towards ensuring that this is actually the case, and that no unauthorized code changes occur.

Whenever a change is checked-into the depot, the developers should be prompted for a defect (or feature) record on behalf of which the change was made. The list of defects (and features) should include only those for whom that developer is currently assigned.

Once given, the system should make a record of the association between the defect (or feature) record and the code check-in, and store it persistently. This level of integration is easily accomplished with modern-day tools.

Providing such a tie-in means we can extend management overview from only "what" was changed (made possible by a source code control system) to also "why" it was changed (made possible by the integration to defect/feature tracking).

Skeptics might argue that management does not really have control. A developer may make an arbitrary change and associate it with any random defect assigned to them. Of course this is possible, but it is not significant for two reasons. First, professional developers are not dishonest. If management clearly asks them to do something, and makes it a part of their jobs, of course they will do it. Secondly, if they do not, there is a clear record of the violation, easily spotted by management and stored for posterity. In the unlikely event that a developer flouts the procedures, they can be dismissed with a clear conscience and an excellent "paper trail" of their repeated violations. Management does indeed have control.

Of course, management cannot merely make up the rule and then never check up on it. Check-ins must be audited for compliance. Once developers learn that management actually cares that their check-ins are associated with appropriate defect and feature records, they will

comply. If, on the other hand, by experience they learn that management does not care (*i.e.*, they do not perform the requested actions and nobody ever notices), the developers will question why they bother going to the extra work, which detracts from their productivity and hence makes them look bad in those things management does care about, and they will turn the process into a joke and rightly so. Management should never request anything of developers that they do not, in fact, care about. Management demonstrates they care about a practice by noticing when it is not done.

One of the most valuable reports that a tie in between the defect/feature tracking system and source code control is a "work done" report:

Date Range:	Last 24 hours

	Matt	Arthur	Mark	Derek
D100203	23			
F100350		108	34	
D155401			56	
D100343				10
D100453	1			
F100782			508	
Totals:	**24**	**108**	**598**	**10**

A typical work done report is accessible via the corporate Intranet using a Web browser. It allows the manager to choose a date range, and then shows a matrix of lines of code added and modified by developer and by feature or defect.

The system ID's of the defects and features are listed down the left of the matrix. Each is a web link that can be followed for more detail on the defect or feature.

The coders who made check-in into the depot over the time period indicated are listed across the top of the matrix (these reports can be hierarchical for larger organizations).

In the center of the matrix are the number of lines of code added and modified (deletions are not a good measure of productivity, though very useful to track in a separate report as deletions usually have a beneficial effect on the architecture). These numbers are web links as well. When followed, they will give details of the check-ins, such as what files were affected. Following links on the affected files, one should be able to see the source code annotated with the changed lines (a number of diff tools are available that can compare two revisions of a file from source control and display the results graphically).

With such a report, it becomes possible to audit all the changes going into the depot. This is useful for managers, charged with ensuring that the most important company priorities are worked on first; and chief architects, charged with ensuring that no changes going into the depot violate the architectural integrity of the system.

11.7. Defect Attribution

One of the most powerful refinements to a defect tracking system is the attribution of defects to root causes. From a system point of view, this is quite easy. From an organizational point of view, it is very difficult.

The tie-in of check-ins to defect records enables one to track where the corrections for a defect are placed in the source code. Thus one can

analyze which source files or modules are most defect-prone, which can guide architectural clean-up efforts.

One can also attribute defects to the development phase in which it was first introduced. At the time the cause of the defect is identified, the developer is in a position to identify if the root cause was a specification issue, a design issue, or a coding issue. By gathering this information, management can determine if process enhancement efforts are better spent adding a review step to specifications, to design, or to coding.

Finally, and most organizationally challenging, if the defect is due to a coding error the coder who fixes it can attribute the defect to the coder who first introduced it. This information is usually easily discoverable once the lines of code causing the defect are discovered. The source code control system keeps a complete history of changes, and thus the moment at which the breaking change was introduced can be discovered and attributed. While this information is very useful in helping coders to benchmark themselves against their colleagues, it can also be used to reward staff and/or fire poor performers, and hence the organizational challenges. In an organization with good management and mature, confident, and professional developers, this can work beneficially.

11.8. Relationship to Customer Issue Tracking

A customer issue tracking system is similar to a defect tracking system. Sometimes the same generic workflow management system can be used for both purposes, although not always, depending upon the requirements of the issue tracking system.

Issue tracking software is used by the customer support help desk function. Whenever a customer contacts the company with an issue, the issue is entered into the system and assigned a ticket number. These issues can be prioritized according to impact to the customer.

Some of these issues are due to defects in the software, however some can be resolved by shipping to the customer a newer patched version of the software, and hence there is no need for a corresponding defect record. In other cases, the issue may be the customer's misuse of the software, the configuration of the customer's computer, or a host of other issues that are not related to current defects in the software.

If the issue the customer is facing is a current defect in the software, it may have been previously reported, but no patch has yet been produced. In this case, the customer issue should be linked to the existing defect record. When a fix to the defect is available in a patch, the link allows customer service to follow up with the customers to resolve the issue from the customer's point of view.

The quantity of such links to a defect record, and the importance to which the customer ascribes the issue, are factors that will affect the software company's internal prioritization of the defect. Thus good systems for tracking the links between issues and defects are important.

If a customer reports an issue that does turn out to be a previously unknown defect, a new defect record can then be created. The fact that a defect escaped into the field should elevate the importance of the defect when attributing to root causes.

If an organization has no customer issue tracking in place, a second instance or a customization of the defect tracking system will probably do well. If the defect tracking system is implemented on a more

general-purpose workflow management platform, this approach can be carried quite far.

However, if the customer issue tracking system needs to incorporate additional requirements then using the same underlying system as that used for defect tracking may not be wise. Such requirements may include a searchable issue knowledge base; customized reporting; web presence for customers (to look up solutions on their own, to submit issues, and to track them); the ability to identify callers who have purchased support contracts; telephony integration (as soon as a customer calls into the help desk, an automated system will create a new ticket and fill in the customer information); or integration with a more general-purpose CRM (customer relationship management) system to integrate the help desk better into the sales cycle (for treating prospects demo'ing the software differently, or for up selling customers on new solutions).

11.9. Shipping Software With Known Defects

Ideally, a software vendor would not wish to ship software with any defects at all. However, for most types of software applications one cannot build a sustainable business on shipping 0-defect software. The effort required to do this would cause the cost of the software to exceed the budgets of potential buyers. Under a 0-defect regime, the software industry as we know it today could not exist. Computers and software would exist for the use of only the most elite consumers.

Given that professionals will ship software with defects, the next question to be raised is how many defects at the various priorities are acceptable.

A software company cannot know how many unknown, latent defects exist in the software it ships. Various schemes, such as *defect seeding*, have been proposed to get a handle on this. Defect seeding means artificially injecting defects into the software and then seeing how many are discovered through testing. The ratio of detected seeded defects to total seeded defects is then assumed to be the same ratio as the total number of known defects to the total known and as yet undiscovered defects. The practical difficulties of this approach comes in injecting "typical" defects into the software, as opposed to easy to find or very hard to find defects, which would skew the results.

In practice, such schemes are not required. given a consistent testing effort, we generally can accept that the number of known defects is proportionate to the total number of defects. This allows us to use known defects as a proxy for total defects.

If we ship software with 350 known defects, and the customer reaction is very negative, we know that it is too many. If, on the other hand, we ship software with 50 known defects (of commensurate priority) and the reaction from customers is "this is the best release ever - very problem free", then we know we are doing something right and might be encouraged to use 50 known defects as the shipping threshold for future releases.

Thus arriving at the appropriate thresholds is a matter of measuring customer satisfaction in their perception of the stability of the software, and relating it to the number of known defects at various priority levels that we ship.

If we significantly enhance the testing effort, we can assume that the ratio of known to total defects increases, and we can therefore allow the shipping thresholds to rise while expecting the same level of customer satisfaction.

For example, assume the ratio is ½. That is, for every defect we know about, there is another lurking in the software. If we ship with 50 defects, we therefore assume there are a total of 100 defects actually shipped. If we improve testing, and raise the ratio to ¾, then if we continue to ship with 50 defects, we are actually shipping only a total of 67 defects. Thus we can allow the threshold to rise to 75 known defects without impacting customer satisfaction. This is good to know, because if we increase a focus on testing without commensurately increasing the number of coders available to fix the defects, we will be hard pressed to meet the old thresholds under unchanged coding time to testing time ratios.

On the other hand, if we increase the coding effort devoted to defect correction (or decrease the testing effort), we can expect the ratio to increase, and thus we must compensate by lowering the shipping thresholds to experience the same level of customer satisfaction.

If we increase testing and coding effort in proportion to one another, we should keep the same historic shipping thresholds, but expect to reach them faster.

11.10. Release Notes

When shipping a new point or patch release it is useful to include *release notes* indicating what defects have been corrected in the shipping build, and what defects remain.

Well-integrated source code control and defect tracking systems can make this easier. Every check-in to the code will carry a defect (or feature) record. An automated tool can be built that examines all check-ins between the last point release and the current one and automatically generates a list of all the defects corrected.

Many of the defect records will not be "customer-readable". For example, the defect might be described as "program crashes when I open this file". That is a perfectly adequate description for a coder to correct the defect, however, it is not customer-readable.

In order to make it customer-readable, the coder who fixes the defect must trace it to its root causes, and then anticipate the likely scenarios that would lead to exercising this defect. For example, the "crashes" defect might be re-described, after some analysis, as "program will crash when opening a file previously saved by release 3.8 of the software if it contains custom fields whose name begin with a non-letter".

It will then be left to product management or the QA/Build group to generate customer-readable release notes from the defect lists and these additional analyses.

The effort in performing these analyses is a considerable additional burden on the coder. However, one may also argue that the additional insight into the defect demanded by this practice improves the quality of the defect fix.

Once these release notes are in place they can be placed on the Web, and customers using the software can see if a particular point release addresses an issue they are currently experiencing. Customer service

can use the information as well when customers call in. This practice is beneficial to end users of the software.

A further practice that is beneficial is listing on the Web site the set of all defects that are known but whose fixes have not yet been released in a point or patch release.

If a customer knows they are dealing with a known defect in the code, this can save them much time and aggravation, and they can work around the defect for the time being and await a fix.

Providing this information in a manner that is customer-readable is even more difficult than providing the fixed defect information, as in many cases the root cause may not yet have been determined.

11.11. Automated Patching Facilities

Increasingly popular, automated patching facilities work to keep the customer up-to-date with the most stable current release of the software at all times.

A facility is built into the software that either on demand or on startup (depending upon user configuration choices), the software will pass the current release number they are running to the company's Web server and query if there are any updates available.

This will occur silently, behind the scenes, if a Web connection is available. If the software is up-to-date, or if no "critical updates" are available, the software will silently complete its initializations. If the facility discovers a critical patch, it will ask the user if she wishes to have the software automatically download and install the patch. Also,

there may be non-critical patches that the user can configure the software to search for as well.

Generally, it is recommended that such a facility be run immediately after installation of the software as the last step of the install process. In this way, software companies are not saddled with giving the users the experience that they initially burned onto the CD, but rather, as time progresses, give them a better and better experience "out of the box".

As long as the facility exists, the software company can extend its use to also display messages they deem important to the end user, or to perform targeted marketing of related products and services, for example, based on the customer's geographic location and products purchased. However, privacy issues must be seriously considered.

When updating software in this fashion, it is desirable to keep the download sizes small, and hence the best approach is to download only the binary differences in changed files. However, this requires a known initial release and final release. There may be very many point releases in the field when the next recommended point is released. The server-side software should therefore compute the "least-cost" (in terms of download size) sequence of patches to get them from where they are at to where they need to go.

This requires a server-side database of all patches available, their download sizes, and the various "from" releases that they may be applied against to get to the "to" release.

The software company should designate certain defect corrections as "critical" and others as "optional". Critical defect corrections are those recommended for all users to install. Optional are those that are

expected to only affect relatively few users and/or effect them in only minor ways.

When automatically determining whether a customer should move from the release they are currently at to a newer one, the presence of a critical defect correction in the patch sequence should be an important factor. If present, the upgrade should be hi-lighted to the customer as "critical" or "important". If not present, the upgrade can be designated as "optional". The auto-update software may be configurable differently by the end-user to deal with these two situations. For example, the customer might set it to "inform me always of critical updates" and "show me only once every 2 weeks any optional updates".

Implementing such an automated patch system requires great discipline in the build environment, and great confidence in the automated regression testing facility.

When patching release **R2.1.a** to **R2.3** using binary patches the software organization must be extremely sure of the exact versions of all files on the customer's computer at level **R2.1.a**. This requires an excellent source control and build environment. For greater certainty, a wrapper around the binary patch applicator should check and refuse to run if the "from" file is not at the right version number and/or if a checksum does not match.

The regression testing environment must be top-notch as well. If the software is pro-actively recommending to the end-user that it update itself, it is very obnoxious if more defects are introduced than are fixed. At least if the user had to download and apply the patch manually they can at least partially blame themselves!

11.12. Summary

In this chapter we made a close examination of the process of tracking defects. We looked at why this is important, what information should be kept with defect records, and a state-transition process model for dealing with defects. We then discussed management controls and metrics that may be used to control the defect correction process and track whether a release is good to go for its GA date after dcut.

We saw how relating defect records to source code control check-ins helps to provide great control over software development, and incidentally, helps in generating useful release notes.

We discussed the relationship between a development defect tracking system and a customer service issue tracking system. While all defects are potentially issues, not all issues are new defects.

We discussed the issues involved in shipping software with known defects and discussed how appropriate shipping thresholds may be arrived at and how they are affected by resource changes.

Finally, we analyzed the benefits of automated patching facilities, how they may be used to advantage, and potential pitfalls.

12. Feature Tracking

In the previous chapter we discussed defect tracking and the workflow management systems that supported finding and fixing defects.

In this chapter, we turn the focus to feature tracking. That is, tracking new features from their inception, to their inclusion in an agile horizon plan, through their implementation, and finally into testing and out to the customer.

The process underlying feature tracking is the agile horizon planning process that was first introduced in Chapter 3, "Agile Horizon Planning Overview" on page 47. This chapter discusses the systems and the practical considerations required to implement and control that process.

12.1. Feature Tracking System

The fundamental requirement to control the next release lifecycle is a system capable of tracking individual features through their lifecycles. The requirements for such a system are similar to those for the defect tracking system, and hence the two are often the same. The chief requirements of both are

- storing data about features,
- being state driven,
- having feature owners vary through time,
- linking to source code control check-ins,
- having a full audit trail of all changes.

Ideally, the defect tracking system is a more general-purpose workflow management system in which a second, feature, workflow may be added beside the defect workflow.

The main additional requirement for a feature tracking system is the ability to associate version controlled documents with feature records (for example, a feature specification document). Barring this facility built-in, documents can be stored in the depot and the path to them referenced in a one of the feature record's text fields.

12.2. Feature Information

The following information should be associated with each feature record.

Description

The feature details what product it is for, and what broad area of functionality it enhances. This can be used later in determining the amount of effort management wishes to expend on various categories of features. A short summary phrase is provided that acts as a mental cue for the feature in question. This allows people to refer to the feature in a consistent manner. This is augmented by a description, which consists of a paragraph or two of information describing the feature. The description should include what customers (if any) are requesting the feature, and why it is important.

Priority

All features (or feature requests, the two terms are used interchangeably here) are prioritized using some scheme, for example 1 to 5, with 1 being the highest priority, most desirable features. Generally marketing product management will assign priorities to features.

If the feature is an architectural feature, without direct end-user impact, it is marked as such and its priorities are not comparable to the other features' priorities.

Target Release

A field is available to indicate the target release that this feature is in-plan for. If undecided, this field is left blank. There is also an entry meaning "definitely not in the next release, but should be considered for future releases".

Effort

All features include an estimate in effective coder days for the amount of effort remaining to finish the feature.

Process Information

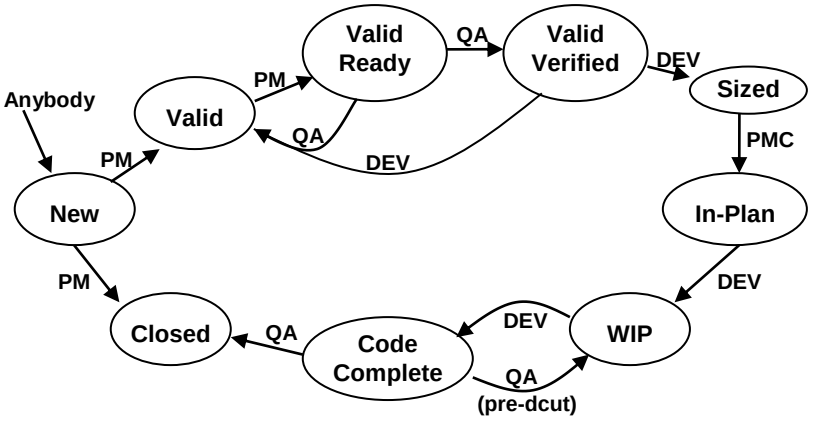
Fields are included that relate to aspects of the process that are done or not yet done. For example, if a meeting is required to discuss each feature, a checkmark is included for this. If a feature requires a written specification, it is indicated here, and so on.

Personnel

The feature lists the person who requested the feature, the coder who will work on this feature, and the tester who is assigned to function test the feature.

12.3. Feature States

A typical state transition diagram for features is given below. The labels on the arrows indicate who is permitted to effect the state transition. **PM** stands for product management, **PMC** for product management committee, **QA** for the quality assurance group, and **DEV** for the coding group.



Note that this is an example only, and in fact this example illustrates some more advanced process steps (in particular, everything between **New** and **Sized**) that might need to be put in place under certain circumstances in certain organizations.

In general, an organization should start with the simplest possible workflow, and only add additional complexity to remedy a problem that occurs persistently. For the sake of illustration only, I have included additional complexity which was required to solve a problem that I once encountered. I included it here in the hopes that this can give readers some guidance on how to do similar sorts of refinements to deal with problems that may afflict them.

New

Anybody at all in the organization is permitted to submit a new feature request. When initially submitted, they start in the **New** state. It is the responsibility of the product manager to move it along from here.

Valid

The product manager may choose to close a new feature if she feels the feature request is ill-informed in any way. She will not do so if she disagrees with feature, rather only if the feature is not a valid request as would be the case if the requested functionality is already present, if the requested functionality makes no sense whatsoever, and so on. If the PM chooses not to close the feature, than she will transition it to the **Valid** state. This means that the feature is a valid and reasonable feature request that can be considered for inclusion in a subsequent release of the software.

Valid Ready

It is then the PM's responsibility to flesh out the feature with additional detail sufficient to allow the development team to properly size the feature. At this stage, the feature should be not open-ended in any way. The PM puts the feature into this state when she believes that she has done this.

Valid Verified

The quality assurance team should be involved very early on in the process. When a feature is in the **Valid Ready** state, it is the QA team's job to validate that the feature is in fact well-described and not open-ended. If so, QA will transition the feature to this state. Otherwise, they will send the feature back to the **Valid** state.

Sized

Once a feature is in the **Valid Verified** state, development can consider the feature and attach a sizing in effective coder days to it. If they feel they have insufficient information to do so, they will move the feature back to the **Valid** state. This, however, will be rare, given the preceding QA review of the feature's validity for sizing.

In-Plan

Once a feature is sized, it becomes a candidate for inclusion into a future scheduled release of a product. The process captured by the first part of this state diagram is intended to culminate in an agile horizon plan. If a feature makes it in-plan for the next release, it is moved into

this state. Otherwise, it languishes in the **Sized** state awaiting consideration for a future release.

The product manager, as chair of the product management committee (the PMC) will physically transition the feature. However, the PMC, comprising experts in the domain and on the product, will collectively form the agile horizon plan. It is the job of the PM to move this process along.

WIP

At some point in the feature lifecyle a development manager will assign the feature to an appropriate coder. Generally this is done when it is initially being sized with a particular coder in mind. The assignment may change later on, in which case the sizing may need to be reconsidered in light of the capabilities of the new coder.

When the assigned coder is ready to begin work on the feature, he will move it to the **Work-In-Progress** state. This is done so that management has visibility into who has started which features. Prior to a feature being **WIP**, management has the option to move the feature to another developer (while reconsider sizing) to better balance the load, or to drop the feature in order to balance the agile horizon plan.

Code Complete

When a developer believes that there is no code that she knows of that remains to be written for the feature to be finished, she will move the feature to **Code Complete**. Once all features in an agile horizon plan have achieved this state, the release has achieved dcut.

Even prior to moving a feature to this state, the QA manager will have assigned a component tester. The component tester will work with the coder to test various cuts of this feature prior to it being set **Code Complete**.

If after the feature is moved to this state, the tester discovers more issues, and if the release is still before dcut, the tester will transition the feature back to **WIP** with informal instructions to the coder on what to fix. After dcut, the feature remains in **Code Complete**, and full-fledged defect records are opened against the feature to record any problems. This rule is made to not skew the defect tracking data when coders and testers have different styles for interacting pre-dcut.

Closed

A feature reaches the **Closed** state either by being abandoned from the **New** state, or by being fully integration tested post-dcut.

12.4. Specifications & Designs

As the features flow through the lifecycle discussed in the previous section, more detail is added. At a certain point, the group must decide if a feature is worthy of a written specification and/or a written design document.

As a base minimum, each in-plan feature should be discussed. In the feature tracking system, a checkbox can be added to indicate whether a feature meeting has been held for a feature, and a text field added to record any notes taken from the meeting.

These meetings may be scheduled by the QA group who will control the agenda and move the meeting along. Each meeting will discuss

several features, and will consist of a group of people knowledgeable on all of the features under consideration. Typically the chief architect, the product manager, the QA manager, and the development manager will attend all such meetings. Others will be invited if they contribute something to the features under discussion.

After discussing the feature for a time, the group may feel it is an easy to understand feature with a straightforward software design. If this is the case, they will switch yes/no fields on the feature record indicating that no written specification or written design is required. The few notes they add to the feature record will fill in any holes.

On the other hand, after a few minutes of discussion the group may feel that either a written-out feature specification is required, or a written-out software design document is required. In this case, they will so indicate using the yes/no fields on the feature record.

A feature specification document will detail all externally visible behavior of the new feature. It will give at least rough designs for all menu items and dialog boxes, explain how all the options work, and give any and all implications to other parts of the code, such as how the new feature will interact with reports, old data files, databases, and so on.

A good specification will often start with a conceptual Object-Oriented Analysis given in UML that names the important concepts in the feature (and/or reverse engineers existing concepts from the current release of the software). This UML then forms the basis for the terminology used throughout the document, and helps inform the GUI design and, later, the software design.

Generally a specific individual will be charged with writing the specification document. This person must be familiar with the domain, familiar with the existing software, write well, and have an understanding of development concerns.

Throughout the process of creating the document, the specification writer will have periodic discussions with product managers, development managers, coders, and the chief architect to attempt to achieve consensus on the general approach to the feature. Occasionally, meetings may be called to resolve logjam issues.

A feature specification is finished when the approach reflects the group's consensus on how the feature should be surfaced in the software; when the specification is *complete* in that there is no aspect of the software operation that is undefined; and when it is *consistent*, meaning that it is not self-contradictory or contradicts the requirements of other features.

Completeness and consistency must be judged not against the existing software, but against the agglomeration of all the new features to appear in the next release. This latter point can often trip up a group in that two seemingly unrelated features may be specified in a way that is complete and consistent with respect to the existing software, but that contradict one another. The more subtle of these faults will only be uncovered during integration testing once both features are code complete.

A software design is a document that explains how a feature will be implemented in the code. This document will typically be created by the chief architect, or by a developer under close supervision by him.

Often the reason a software design is required is because it is not immediately apparent how to implement the feature in order to achieve adequate performance. Therefore, the designer will typically do some experimentation on a private codeline before deciding on the ultimate approach and documenting it for whomever is going to implement it.

Another reason would be that the feature is quite complex, and/or will require more than one coder to work on it. In this case, a class design in UML and a database design in an ER (Entity-Relationships) notation may be required before the coder(s) can effectively begin on their pieces. This may also be the case if the chief architect is delegating the design to a less experienced developer, and would like to review the planned design prior to it being coded.

Whenever a specification or a design is deemed to be required, the fact should be recorded in the feature record. Once the appropriate document is completed, it should be attached to the feature record and other fields switched to indicate that the documents have been completed.

Note that it is almost always overkill to require this level of documentation for every new feature in a release. So long as a responsible group of knowledgeable individuals explicitly decides that these documents are not required, that is fine as well.

12.5. Reviews

Once the basics are in place, in the further quest for improved quality and productivity, reviews are a fertile source of process maturity. True QA starts at testing, and continues by pushing earlier and earlier into the development cycle.

The earlier a defect is found, the less expensive it is to correct. If a half-baked feature can be killed early on in the proposal stage, that is best. If a poor specification would lead to a poor-quality and inconsistent feature that users will dislike, then it is good finding that out sooner rather than later. If a poor design would result in a feature whose performance is unacceptable, again, better to know earlier. If code is badly written and would result in many defects, then better to review the code directly to find that out. If a defect in code can be detected the day after it is introduced, then that is much less expensive than if it is only found much later. If a defect is discovered by a customer in production, that is worst of all.

In this section, we discuss various reviews that may take place at earlier stages in the feature lifecycle.

12.5.1. Feature Review

In the previous section that presented a sample feature state diagram, we touched on a very early QA step designed to ensure that feature proposal would not get past an early stage unless it was well-defined. Inserting this QA step helps ensure that nobody's time is wasted in sizing, agile horizon planning, or subsequent implementation on a poorly thought out feature request.

Such a step is especially important when an agile horizon planning methodology is first being introduced into an organization. Product managers not familiar with the approach tend to submit features for consideration that do not correspond to implementable concepts. An example might be "add more wizards". Users may well be requesting "more wizards", but for the feature to be actionable, the company requires more specific features, such as "add a csv file import wizard".

This review step may be implemented by adding a state to the feature workflow, being clear on what it takes to pass this review step, and assigning an individual or group to be in charge of carrying out these reviews.

12.5.2. Specification Review

The process described previously calls for a meeting regarding each feature to determine if a written specification is required. If one is required, then the process could be enhanced further by adding a specification review.

A specification review would consist of a group of people who would take the time to read the specification and find incompleteness and inconsistency in it. A specification review should not have as a goal second-guessing how the feature is to be surfaced in the software. Well prior to the stage of a reviewable specification, consensus should have been arrived at on this point. Dissenters must now hold their tongues and accept the group's decision.

QA should schedule a specification review meeting and chair it. During the meeting, the chair should ask the reviewers to prioritize their comments, and then proceed round-robin around the room.

Comments should confine themselves to incompleteness and inconsistency, not to suggestions on how to improve the feature. If such suggestions arise, the chair should ask the person to connect one-on-one with the specification writer and lobby for their change to be included. When a potential problem is uncovered, the group should discuss whether it is a valid problem or not. If the consensus is that it is, the problem should be noted and the review should move on. By no means should suggestions on how to resolve the problem be considered at this point. Keeping the review focused on uncovering problems is essential if the meeting is not to deteriorate into a three-hour exercise with little to show for it.

The chair will then record all the review meeting comments and attach it to the feature record.

The original specification writer should then be trusted to review all the issues and correct the document. It is usually counter-productive to hold yet another review meeting after this, or to "check-up" on the fact that the specification writer did indeed solve the problems. We would want to encourage the specification writer to want to get the feedback from the meetings in order to produce a better document. Additional review steps tend to diminish this desire.

12.5.3. Design Review

If a software design is deemed to be required, and if it is not the chief architect who writes the document, then the chief architect should review the document and sign-off on its completion.

It is also possible to have a broader review meeting, along the lines of the specification review meeting described above.

The fact that a design document is completed, and the fact that it is signed off, should both be stored in true/false data fields associated with the feature record.

12.5.4. Code Review

After source code is written, a code review would involve a number of individuals stepping through the code in a meeting trying to find problems with it. Again the fact that a code review meeting was held, and any problems so uncovered could be attached to the feature record.

An alternative to a full code review would be a "buddy system" whereby a second coder would step through the code (at their desk, possibly with the original coder beside them, ideally with the help of a debugger) and search for potential problems. Again, the feature tracking system would be used to record the fact that this activity took place.

Organizations wishing to institute code reviews should start with the simpler version, which is easier to manage and easier for everybody to get used to, and later see if the more elaborate code review improves the defect finding rate.

12.5.5. Feature Demo

A very effective review step is a feature demo meeting. These meetings will be scheduled by QA after a number of features are approaching or have passed **Code Complete**.

In this meeting, a number of interested parties including the coders, managers, product managers, testers, and documentation are brought together and the coder will demonstrate the new feature on a development build of the software. It is important that this be a

development build and not a private build, as management will want assurance that the feature may be built and tested as part of the regular development build process.

The main purpose of the meeting is to provide a concrete point at which the readiness of a feature may be assessed. Scheduling this meeting also tends to bring focus to the coders in polishing off a feature for the scheduled date. A secondary purpose is to get suggestions to the coder to fine-tune the feature.

A scribe should be appointed for the meeting who will write down people's suggestions for improvements. Afterwards, the coder and their manager will sit down to decide which of the improvements to incorporate. The scribe will attach their notes to the feature record and mark a true/false field indicating that the milestone was passed. If the feature is a disaster, crashing all over the place, that feature should be reviewed again at a later meeting.

12.6. Effort Tracking

An important aspect of the process is to track the actual effort in dedicated hours spent on the coding activities.

Of most importance is tracking the number of dedicated hours spent by each coder on each feature. This is possible to do using a numeric field in the feature record. Each time some additional hours are spent on a feature, the assigned coder will increment the field.

It is also necessary to collect total time spent fixing defects by each coder. Unless some very sophisticated defect attribution analysis is planned, it is sufficient to collect this information across all defects, and not ask coders to separate out time spent on each individual defect.

In addition to this, we must have some means of recording days off by a developer (including company-wide holidays).

Rather than shoe-horning this into the feature/defect tracking system, a better solution is to build a custom-purpose fine-grained time tracking utility that keeps its records in a centralized relational database.

It is important not to get carried away and attempt to account for every single hour of the day of each coder. The coders will feel overly-controlled and "watched". Therefore the system should not force coders to clock-in and clock-out, not force them to record break time, and not force them to account for every hour during the workday. This is onerous on the coders, conveys a feeling of mistrust from management, and will negatively impact job satisfaction (and thus productivity as well).

Rather the system should insist on collecting the following information that is important for managing the agile horizon planning process:

- Dedicated hours spent working on each assigned feature each day.
- Dedicated total hours spent working on all assigned defects each day.
- Days or parts of days taken as vacation.

As well, not related to collecting effort, but coders must have a means of updating the estimate of the number of effective coder days required to finish each feature they are working on. This can be a numeric field in the feature tracking system, however as it is desirable to analyze the changes in this quantity side-by-side with the effort data, it is

reasonable to have them both collected by the same system and stored in the same place.

Each coder should log into the system at the start of each day. They should be able to choose from a menu of activities for that day to log time against. Timers should be available whose values can later be edited before submitting them (in case a coder starts a timer, then wanders off for an hour forgetting to pause the timer).

Activities should include all features assigned to that coder in the **WIP** state. It should also have a generic **defects** entry.

Whenever time is recorded against a **WIP** feature, the system should prompt the coder to re-estimate the time remaining to finish the feature. As a default, the system might provide the coder with their last estimate with the new time spent subtracted from it, although this is dangerous as it encourages the coder to accept that suggested value rather than re-thinking for themselves an estimated time remaining based on new information they may have gleaned while working on the feature.

At the end of each day, the coder will log out of the system. At this time it should present the coder with the time recorded that day, allowing them to edit it before submitting it.

The system should also provide a facility that enables coders to estimate at what points in the future they plan on taking vacation, and to update the system with their vacation actually taken.

12.7. Management Control

With feature tracking and effort tracking in place, as described in the preceding sections, management is in an excellent position to bring hard quantitative data to bear on the release cycle.

12.7.1. Coder Work Factors and Vacation Estimates

Key predictive quantities in the agile horizon plan relating to total coder capacity available to the end of the coding phase are the work factor for each coder and their estimated vacation prior to dcut.

The effort tracking system described previously gathers this data. Work factor to-date by coder can be computed by summing the total feature effort from fork to the current date, and dividing by the number of workdays less vacation taken. This gives a historical measure of what proportion of each day each coder has been taking to code new features into the next release. This information can then be used in putting estimated work factors into the agile horizon plan going forwards.

Sometimes, measuring this information will show certain coders with extremely low work factors. the first thing to look into is whether the coder has been accurately recording time spent against each feature. If not, then the coder should be re-encouraged to do so. The fact that management has noticed the lapse will usually be sufficient encouragement.

If this is not it, then typically the coder has been spending all their time on other sanctioned activities. If the agile horizon plan is in jeopardy in any way, freeing up the coder from these other duties will

be a fertile source of extra capacity. Usually, this will require a dialog with more senior management to defer whatever other project has been consuming their time. If this is not possible, then senior management must address the issue by allowing a smaller estimated work factor in the agile horizon plan for that coder, and adjust the plan features or dates accordingly to re-establish the plan's capacity constraint.

Occasionally, there will be no explanation whatsoever for the low work factor. This may or may not be cause for alarm. Some coders work in spurts, and may have very low work factors while the pressure is off, and then sprint to extremely high work factors as deadlines approach.

One of the most common causes for low work factors is the time spent working on defects. It is generally advisable to have a rule of some sort stating that defects above a certain priority have precedence over new feature work. If there is a flood of such defects, this can eat into the work factors considerably.

Measuring the total time spent on defects is what allows management to see if this is the case. If high defect rates from previous releases are causing the problem, management can react by lowering estimated work factors and re-balancing the plan, or by raising the priority level of must-fix defects, or by explicitly constraining the amount of time each day coders spend on defects.

Graphing the group's cumulative work factor since the start of coding can yield fascinating insight into the group's behavior as it relates to next release development. Management should keep a close watch on

this metric, as it is arguably the most important one relating to the future health of a commercial software vendor.

12.7.2. Actual Versus Estimated Feature Time

Another important purpose for the collected effort data is to compare estimates versus actuals for feature effort.

After a number of features have been completed, a statistical view of estimation accuracy can be taken. Usually, certain classes of features (*e.g.*, small simple ones) might have better estimation accuracy than other classes. Based on this data, management may revisit not yet completed features belonging to the inaccurate classes and adjust the estimates to compensate for the systemic errors.

This should be done in concert with all those involved in arriving at the original estimates, so that they can learn from the experience.

The data will also reveal those coders whose features always come in to estimate versus those whose features are always under-estimated.

Coders who come in right at their estimates are usually finishing earlier and then polishing either the feature's functionality or its underlying code. In this case, management must look at the nature of the polishing activity and decide if it is appropriate or overkill.

Coders who consistently under-estimate their features are usually doing a poor job at anticipating all the detailed work that needs to be done to complete a feature.

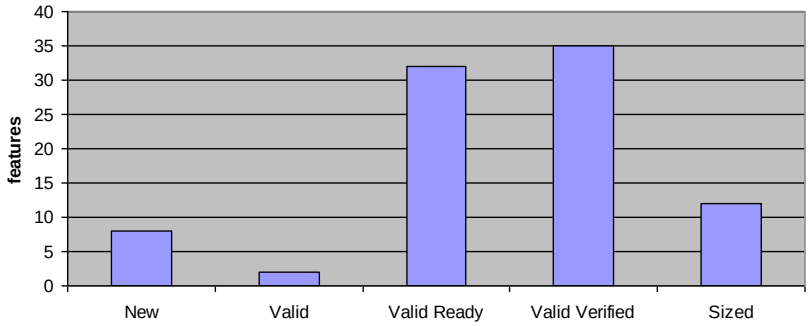
Sometimes the reason for this is they feel in their hearts that if they actually did this exercise, the feature estimate would come in much higher than that in the agile horizon plan. This in turn would cause themselves, their colleagues, and their managers grief. Management can

help allay this feeling by not acting out when estimates are increased mid-plan, but by dealing maturely and calmly with the situation. At the same time, coders who consistently under-estimate should be asked to provide written estimates of the work remaining, broken out by work item, and then have this estimate reviewed by others.

12.7.3. Progress to Process

With the steps in the process marked off on a feature-by-feature basis in the feature tracking system, management reports giving progress to process are possible.

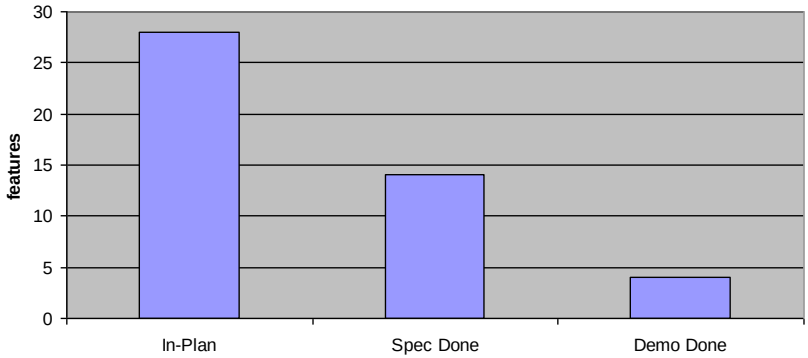
An initial report would show the number of features in the various pipeline stages as a bar chart. Clicking on the bars would link to a list of all the features. The first bar would display the quantity of **New** features, the second the **Valid** ones, and so on.



This report would be used to manage the activity leading up to an initial agile horizon planning session. For example, the report above shows that product management is doing a good job moving features along to the **Valid Ready** state. However, there appears to be a bottleneck in the QA department moving features to the **Valid Verified**

state, and in the coding department sizing features. This information allows management to take appropriate action leading up to an initial agile horizon planning meeting.

A next report might detail activity relating to in-plan features:



This report indicates a total of 46 features in-plan. Of those, 4 have passed a successful code complete demo milestone, 14 are not yet completed to this stage but either have a specification and design written and reviewed, or don't need one. 28 features have neither of those things and are still in some other stage of the process.

A report like this can help management determine if the organization's process initiatives, such as writing specifications, is effective or not.

12.8. Summary

In this chapter we looked at processes and systems to assist in tracking and applying metrics to the agile horizon planning methodology.

We started by considering the requirements of a workflow management system used to keep track of feature request records. We itemized the type of data that needs to be kept by such a system, and the detailed workflow as expressed by a state transition diagram.

We then considered specification and design documents. We detailed what these were, who creates them, and how to decide when they are needed.

Next we examined reviews. We looked at the various kinds of reviews that have shown to be useful and how to conduct them

We then looked at the thorny issue of tracking effort. We considered both systems to use to track time at a fine-grained level, and dealt with some of the organizational issues involved.

We ended with a discussion of various types of management controls that are enabled through good data tracking of clean data and good reporting based on that data.

13. Process Control

By the time an organization has its infrastructure functioning effectively, has good defect control, and is tracking the software development process well as described in the previous chapters, the organization is overdue to begin documenting its software development process.

13.1. The Process Document

A good process document describes concisely the steps in the process of developing and releasing software. The document should capture what is going on now, not just what management wishes would go on sometime in the future.

Even that can be problematic. Usually, there are steps that management believes are taking place that are only occasionally taking place. These steps should be documented, and appropriate reporting put around them. That way, if management is serious about these initiatives (*e.g.*, producing a specification for each feature that needs it), then it can monitor and take action when non-compliance is detected, or rather when the extent of non-compliance is beyond where it should be.

Capturing the process in a process document is important to this type of clean-up effort. The process document makes clear to everybody what is expected and how it will be monitored. When new employees enter the organization, they can be given the process document to read to understand how they fit in and get them started on the right foot.

Only when the current process is documented, solidly in use, monitored, and measured, should management consider adding a process enhancement, such as another review step.

When the organization is ready to enhance the process, the process document will be edited first, making clear to everybody what the new step will be, the criteria for entering and leaving the step, and how the fact that the step was carried out will be recorded.

The remainder of this chapter will explain how to write a process document, giving an example of a typical process document that fits with the ideas in this book.

13.2. Documenting Process

A process is documented as a series of process steps. To document a process step it is necessary to cover the following information.

13.2.1. Scope

The process step's scope will detail the unit upon which the step will take place (e.g., feature by feature, or an entire build), the duration of the step in the release lifecycle, and how often the step will be repeated and under what conditions.

13.2.2. Actors

The actors are the staff involved in carrying out this step. If one or more actors are more central, they will be indicated in boldface.

13.2.3. Inputs

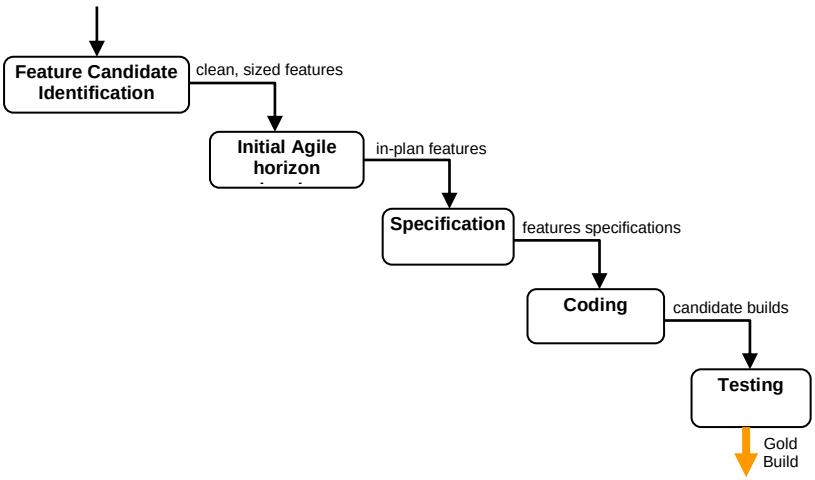
The inputs identify what information is required in order to validly begin the current process step, and any requirements on those inputs.

13.2.4. Outputs

The outputs state how the results of executing the step are captured, any requirements on the outputs, and how the fact that the step was completed is indicated.

13.3. Sample Process Document

The major steps in a sample process are shown below. Each major step groups a number of activities. These activities are described below.



Feature Candidate Identification

Initially, candidate features for the next release are identified, their suitability for use in the following steps are validated, and the features are given an initial sizing.

1) Feature Request

- Scope:
  - feature-by-feature
  - duration: continuous
- Actors:
  - Marketing Product Manager
  - Staff with ideas
  - Partners
  - Customers
  - **Champion**
- Inputs:
  - Ideas for product features
  - Competitive research
- Outputs:
  - A feature in the feature tracking system in state **New**
  - There is a short meaningful title for the feature (1-5 words)
  - There is a < one paragraph description of the feature
  - The feature has the **product** set appropriately
- Description
  - In this process step, we capture ideas for new features that may go into our products. These features may be originated by anybody, but should have a champion within the organization. The informational requirements are light at this stage: just a short general idea of the feature and a meaningful title.

2) Feature Triage

- Scope:
  - feature-by-feature
  - within 5 days of a **New** feature being submitted
- Actors:
  - **Product Manager**
- Inputs:
  - Features in state **New**
- Outputs:
  - Feature moved to state **Closed** if already doable, a duplicate, or makes no sense
  - Feature moved to state **Valid** if a reasonable request for that product
- Description
  - The product manager will triage the features submitted in step 1, searching for those with merit. This step is intended to be performed very quickly after a feature has been submitted, and is intended to keep the data clean. The product manager will quickly determine if a requested feature can already be done in the software, if it is a duplicate feature request, or if it just makes no sense. If so, he will close it. Else, he will move it to the **Valid** state.

3) Feature Identification

- Scope:
  - feature-by-feature
  - only for those likely to be in-plan on the next release
  - repeat until the feature passes Feature Validation
- Actors:
  - **Product Manager**
- Inputs:
  - Features in state **Valid** for the product in question
- Outputs:
  - A feature that is a candidate for the next release
  - Any marketing requirements are listed
  - The feature is cohesive (only grouping highly-related functionality)
  - The feature cannot be reasonably be divided into meaningful, stand-alone sub-features
  - The feature is constrained in scope, not open-ended
  - The feature has the **target release** set appropriately
  - The feature is placed into a priority class relative to other features
  - The feature is in state **Valid-Ready** indicating it is ready for feature validation (see next step)
- Description
  - At this stage in the process we are contemplating the next release. This is the work done beforehand by the product manager to clean up the features that are likely candidates for inclusion in an agile horizon plan. This step entails considerable work, and so is only undertaken once a feature looks like it has a reasonable chance of getting in-plan. The information is necessary in order to have a clear sense of the feature both for sizing and for agile horizon planning discussions.

4) Feature Validation

- Scope:
  - feature-by-feature
  - repeat until pass
- Actors:
  - **QA**
- Inputs:
  - A candidate feature marked for the appropriate **target release** in the **Valid-Ready** state
- Outputs:
  - If passed this will be indicated by moving the state to **Valid-Verified**
  - If failed this will be indicated by moving the feature back to state **Valid** with the **Log** field indicating the no-pass reason
- Description
  - This is an early-stage QA validation step to ensure that features being considered for the next release are well-defined. This step ensures that the output criteria from the previous step are met. Without this step, agile horizon planning meetings tend to degenerate into extended discussions on what a feature is, rather than discussions about whether to include it in plan. However, that meeting will contain the wrong people to engage in that type of discussion, and will waste the time of the decision makers who were brought together to decide matters of feature priority. As well, ensuring the features are well-described will lead to more accurate sizings, which will improve agile horizon planning accuracy.

5) Sizing

- Scope:
  - feature-by-feature
  - repeat whenever new information arises for a feature
- Actors:
  - **Coding Manager**
  - Developers
- Inputs:
  - A feature in the **Valid-Verified**, state marked for the appropriate target release
  - A feature in the **In-Plan** or **WIP** state if resizing is called for
- Outputs:
  - A (new) sizing in ECD (effective coder days) attached to the feature
  - (optional) one (or more) assigned coders for whom the sizing was made
- Description
  - All features entering this step have already been validated as being useful, well-described features that have a fighting chance of getting into the next agile horizon plan. At this step, developers will come up with a rough initial design for the feature, then divide the implementation into appropriate tasks, size each one and sum them to come up with a feature sizing in effective coder days (ECDs). At this stage, the team will have a rough idea of which coder will be assigned this feature, and may make a note of that in the feature record. If more than one, the sizing will be cumulative across all the developers. At a later stage, the feature may be sub-divided by coder if required to track it better.

Initial Agile horizon planning

An agile horizon plan is put together which lists the chosen in-plan features, gives a final release date, and respects development constraints (balancing capacity with requirement).

6) Agile horizon plan Prep

- Scope:
  - all sized, valid-verified features
- Actors:
  - **Product Manager**
  - Coding Manager
- Inputs:
  - sized, prioritized, valid-verified candidate feature list for this release
  - an initial, suggested end-date for the release
  - an understanding of the initial assignment of coding resource to the release
- Outputs:
  - A preliminary, prioritized suggestion for a feasible agile horizon plan (delta=0)
  - A prioritized list of alternate features
- Description
  - This step is preparation for the agile horizon planning meetings to come in the next step. At this step, the product manager will create an initial suggestion for an agile horizon plan based on her own ideas and those gathered informally from other decision makers. As well, she will also give a list of features that she knows certain decision makers may want in the release. With this preparation work done, the agile horizon planning meetings have a starting point and tend to be much more effective as a result.

7) Agile horizon plan Meetings

- Scope:
  - all sized, valid-verified features
  - repeat when current plan predicts a gold build slip > 1 month
- Actors:
  - **Product Manager**
  - Coding Manager
  - Agile horizon planning Committee
- Inputs:
  - A preliminary suggestion for a feasible agile horizon plan (delta=0)
  - A prioritized list of alternate features
- Outputs:
  - A feasible agile horizon plan (delta=0)
  - In-plan features moved to the **In Plan** state
- Description
  - These are the key meetings held by the decision makers in the organization to decide on the next release date and feature content. Each product has a "Agile horizon planning Committee" that is empowered to collectively make these decisions. Coming into these meetings, the product manager has already prepared a suggested agile horizon plan and a set of alternate features. In these meetings, the product manager will start by explaining her rationale, and describing the importance of each feature. The meeting participant will then suggest changes to this plan and debate them. These meetings are organized and driven by the product manager.

Specification

All features are discussed in meetings, and some are designated as requiring a more extensive written specification. For these features, a written specification is produced and formally reviewed.

8) Specification Meeting

- Scope:
  - feature-by-feature for in-plan features
  - may deal with multiple features at once
  - repeat as required by the spec writer
- Actors:
  - **Coding Manager**
  - Spec Writer
  - Product Manager
  - staff with ideas
- Inputs:
  - An in-plan feature
- Outputs:
  - A decision recorded with the feature on if a written specification is required
  - Notes taken by spec writer attached to the **Spec Notes** field of the feature
- Description
  - Once a feature is designated as in-plan, each feature will need to be reviewed in one or another specification meeting. This group should involve people knowledgeable about the feature request, and the implementers. The purpose is to cement a solid understanding of what is being implemented. Some features may be dealt with quickly. Others will be deemed to require a full written specification.

9) Specification Creation

- Scope:
  - feature-by-feature
  - only those features marked as requiring a written spec
  - refine as spec defects are identified prior to code start
- Actors:
  - **Spec Writer**
  - Staff with ideas
- Inputs:
  - An in-plan feature requiring a spec
  - Spec notes from the specification meeting
- Outputs:
  - A specification document attached to the **Spec Notes** field of the feature
  - The specification must describe all user-visible aspects concerning how the feature will work
- Description
  - If a feature is not clearly understood coming out of the specification meeting in the previous step, a full-blown written specification is marked as required. A specification writer will be assigned and he will gather input and create a specification for the feature detailing all end-user visible aspects of the feature (but not at all delving into implementation concerns).

10) Specification Review

- Scope:
  - feature-by-feature
  - only those features marked as requiring a written spec
  - repeated only if a feature spec fails miserably
- Actors:
  - **QA**
  - Spec writer
  - Reviewers drawn from qualified staff
- Inputs:
  - An in-plan feature that has a spec
  - The specification document
- Outputs:
  - A list of defects with the specification:
    - spec fails to specify what happens under certain conditions
    - spec does not satisfy all the user requirements
    - spec does more than satisfy the user requirements
    - spec is internally inconsistent or inconsistent with how things already existing or specified function
    - The list is saved into the **Spec Notes** section of the feature
- Description
  - After a written specification has been created, it is mandatorially reviewed in this step. The review meeting is designed to find concrete flaws in the specification, and not to discuss how the feature is surfaced. Flaws include any inconsistencies and any missing parts of the specification (usually coders are excellent at finding these).

Coding

Features are coded, unit-tested by the developers, and component tested by testers. As features become code complete, the functionality is demonstrated in a feature demo meeting. During component testing, testers develop automated regression tests and/or manual test scripts for testing each new feature.

11) Coding and Unit Testing

- Scope:
  - feature-by-feature
  - repeat as defects are identified
- Actors:
  - **Developer**
  - Architect
  - Spec Writer
- Inputs:
  - An in-plan feature with a reviewed specification document (or marked as not requiring a spec)
- Outputs:
  - Code that fully implements the spec and in conformance with architect's technical vision
  - COM API code that can be called by a test script and that executes the specified functions
  - Tested by the developer
- Description
  - At this step, the feature is coded by the assigned developer. As coding proceeds, the developer will update time spent to-date and estimated time remaining on the feature. The coder will test the feature to the best of her ability before releasing it to be tested.

12) Feature Demo Meeting

- Scope:
  - feature-by-feature
  - may deal with multiple features at once
  - repeated only if a feature fails miserably or requires very extensive changes
- Actors:
  - **QA**
  - Developer
  - Spec Writer
  - Product Manager
  - Interested staff
  - Scribe
- Inputs:
  - A new feature nearing the code complete state
  - A nightly build clean on all regression tests containing the new feature
- Outputs:
  - a list of defects/corrections to be made to the feature saved into the **Log** field of the feature
  - A determination on if this step is passed
- Description
  - This process step involves a meeting in which a nearly ready feature is demonstrated to the group. Feedback from the audience is given and the developer may choose to change some aspects of the feature as a result. A scribe is assigned to take notes, as the developer is usually busy demonstrating. An additional purpose of this meeting is to determine the fitness of a feature for inclusion in the release at a relatively early stage in the overall process. This step therefore provides a milestone using which the coding group's progress can be assessed.

13) Component Test

- Scope:
  - feature-by-feature
  - repeated as desired by tester after further code changes
- Actors:
  - QA
- Inputs:
  - a nearly code complete feature
  - nightly build containing reviewed code, clean on all regression tests
- Outputs:
  - A list of defects with the feature
  - The list is saved into the **Log** field of the feature
  - Automated regression tests are added to the regression testing system to fully or partially test the feature
- Description
  - As a feature nears completion, a tester will be assigned to the feature to begin testing this one feature in isolation and providing informal feedback to the developer. During this time, the tester will come up with a plan to test this feature, ideally using the automated regression testing facility to the maximum extent possible.

Testing

During the testing phase, builds containing the entire feature set are used to test the features one-by-one according to the plans previously produced during component test. Eventually, gold master candidates are produced which go through a more extensive testing regime which considers all aspects of the user's experience.

14) Integration Test

- Scope:
  - requires all in-plan features to be finished
  - feature-by-feature
  - repeated on each new build if judged necessary
- Actors:
  - QA
- Inputs:
  - A post-DCUT build, clean on all regression tests.
- Outputs:
  - Defects recorded in the defect tracking system.
- Description
  - Before this step starts, the release will have achieved DCUT (development cutoff) meaning that no developer knows of any additional code that needs to be written for all their assigned features to be complete. During integration test, the test plans developed during the component testing step will be executed. As well, the testers will look for possible adverse interactions between the various new features in the release.

15) System Test

- Scope:
  - requires all in-plan features to be finished
  - feature-by-feature
  - repeated for each new build
- Actors:
  - QA
- Inputs:
  - A candidate gold master CD, clean on all regression tests
  - Complete with installation scripts
  - Other products that need to work with this one
- Outputs:
  - go/no-go decision on ship
  - Defects in the defect tracking system
- Description
  - Final tests designed to assess the suitability of a particular build for release.

16) Regression Test

- Scope:
  - continuous throughout release cycle
  - repeated each night
- Actors:
  - QA
  - Development
- Inputs:
  - A nightly build that compiles
- Outputs:
  - a report on tests passed and failed
  - Defects reported to developers or new baselines
- Description
  - Continuous regression testing each time a new build is produced.

13.4. Process Enhancement

The sample process detailed in the previous section is a basic agile horizon planning methodology implemented on a sound infrastructure supplemented by:

- A QA step to validate suggested features,
- a specification meeting for each feature with notes taken,
- written specifications when called for,
- a specification review step,
- a feature demo meeting.

This was intended as an example only. No canned process fits all organizations, or even all projects within an organization. What is crucial is for the process document not to overstep (by too much) the process maturity level at which the team is currently working.

Further enhancements to this process might include:

- A design meeting lead by the chief architect for each feature with notes taken,
- written design documents when required,
- design document reviews,
- A GUI prototype completed first, with a demo meeting for that
- A debugger walkthrough by a "code buddy"
- A formal code review
- An explicit process step to construct and then review the automated regression test for each feature,
- and so on...

Coming up with a list of all the best practices that "ought to be" followed by a team is an exercise involving a few hours with some software engineering textbooks and a Web search engine. Making the process that management believes is happening *actually happen* is a huge challenge. Improving on the process, by even a small amount, is an even bigger challenge.

Anytime a process is first documented, chances will be that it is not being followed as completely as management would like. The fact that there was no prior written process documentation practically guarantees this. Documenting what is supposedly already happening, and then putting in place all the associated process monitoring and measurement is a good first step. Having management actually monitor and measure the process, and then putting the results to a constructive use, will already be a stretch for the team.

In such an environment, inserting additional process steps would be counter-productive.

Any team has only a limited capacity to change and adopt new habits. If management attempts too much change all at once, nothing at all will be the result. Having a documented process and monitoring it is already a huge change. No sense changing even more at first.

Once the written process is established and monitored, and non-conformance to it is the exception that is promptly dealt with, then it is time to sit back, consider the existing process, and decide where to improve upon it.

A process improvement should be one extra initiative. The steps involved should be very carefully defined. There should be a definite,

machine-readable way of determining if the step was taken and what the results of the step were.

If the step involves producing something, there should a quality assurance step associated with it to determine if the thing was produced in a satisfactory fashion. Initially, the QA step should be quick and informal. As a later initiative, a more formal QA step may be introduced. For example, if management decides that designated features should have written design documents, it would be a mistake to insist on the documents and a full-blown design review meeting as part of the same process enhancement initiative. On the other hand, it would also be a mistake to add a step that produces a design document and not have any second pair of eyes give it a quick once over, making a record of acceptance or rejection.

Once management decides to endorse an enhancement to the process, they must focus their energies on ensuring that it gets accomplished. A common management mistake is to launch an initiative; put in place training, tools, and procedures to support it; announce it as a significant step forward to the rest of the company; and then go on to the next thing.

All process enhancements require additional work from the staff on a regular basis. They will quickly determine if it is the case that nobody seems to care if the step is done or not, and will devote their energies to things people do seem to care about in a more direct fashion, such as coding many features and fixing many defects, for instance.

Management must be dogged in its determination to put in place a new process step. They must set up all necessary monitoring and then perform the monitoring on a regular basis. If staff fail to comply or do a

poor job at the process step, management must raise it as an important issue. This must occur for *every single* violation.

This amount of management focus on implementing a new process initiative is a significant bottleneck to process improvement. If management is constantly fighting fires, or spending all their time struggling to enforce existing process steps, or spending all their time playing politics within the organization to keep a project alive, then it is unlikely that the necessary management bandwidth will be available to effect a process change.

13.5. Summary

In this chapter we examined the idea of documenting a process. We described the benefits of doing this, and how to do it. We gave as an example a fully-elaborated process for a basic agile horizon planning methodology. We ended with thoughts on process enhancement, and the limited ability of any organization to change too much all at once.

.

14. Architectural Clarity

Up to now, we have discussed the correct management of software ventures and how to setup the right infrastructure and tools. However, a software venture has as its goal the production of source code. All the effort, all the process, all the management, needs to result in a collection of text files containing source code at the end of the day.

In order to control costs, that source code needs to be as clearly written as possible.

14.1. The Efficiency of Clarity

A disorganized muddle of source code is unlikely to build into a well-functioning software system. If, by some chance, it does, then the cost of finding and fixing defects and of adding new features will be prohibitive. Thus, one of the most important attributes of a system is the clarity of its source code.

There is clarity "in-the-small" and "in-the-large". Clarity "in-the-small" means that whenever a programmer examines lines of source code, the meaning of it is clear to her and it is easy for her to see that the code is correct, meaning that it does what it is clearly intended to do.

Clarity "in-the-large" means that the overall operating principles of the system are clear, and the organization of the source code (into directories, files, classes, methods, and so on) clearly reflects the operating principles, making it easy to find the source code that

performs each of the various functions. This is often referred to as clarity in the "architecture" of the software.

It is vital to the ongoing health of a software venture to retain and enhance the architectural clarity and the code clarity of a system.

As we discussed previously, the cost of maintaining successful software (adding features and correcting defects) dwarfs the initial cost of creating it. Therefore, any improvement in maintenance efficiency will yield leveraged dividends. The largest improvement in maintenance efficiency comes from having a clear architecture and clearly written source code.

With clear architecture and code, defect frequency is reduced. Defects arise when coders cannot convince themselves by examining the code they have just written that it is or is not correct. If they can clearly see it is not correct, they will fix it. If they can see that it is clearly correct, the probability that it is correct is greatly increased. If they cannot see one way or the other (which happens often) then it probably has defects which will go un-corrected until testing uncovers them.

When defects are discovered, clarity in the architecture and source will make it easier for a programmer to locate suspect areas of code, to convince himself whether or not the suspect area is indeed the cause of the defect, and to see how to correct it if it is. This decreases the time required to correct defects, thus decreasing costs given the same level of quality.

New features become faster to design and code as well. Clear architecture will make it straightforward to understand and then document the design of a desired enhancement. Clear code will make it

easier for the programmer to add the required code into the system while containing unexpected side-effects.

Having clearly understandable architecture and source code is the single most effective means of improving next release efficiency.

14.2. Code Clarity

If the desire is to create clear code, the natural first question is how to go about achieving it?

It starts by having appropriately trained and experienced staff creating code; and, where sufficient experience is not yet achieved, having more experienced staff review it.

Creating code is an exercise in mathematical logic. There is no more unrelentingly logical environment as a software program. Any fuzzy, illogical lapse on the part of a programmer is mercilessly punished by the system with hour upon hour of additional debugging.

This is why experience has shown that typically University graduates with strength in maths and sciences are required for the job. Writing a program requires the same attention to detail, quickness of mind, and extreme logical thinking required to do well in school exams. University graduates are selected for these traits.

Training in the computer sciences at the University level is required. There is a vast body of knowledge (too much to fit into an undergraduate University program, in fact) that a professional programmer needs to know. A good deal of the more basic knowledge is mathematics, logic and logical programming. As well, writing

program after program and having their correctness and clarity critiqued (via a grade) is essential to the development of a strong programmer.

Programmers without a good education will tend to make two mistakes. The first mistake is to fail to understand what is happening at every level of the system. The second mistake is to think operationally rather than logically about a program.

An appropriate education should cover off how hardware and software systems function from the level of the NAND gate upwards. Students should get at least a rough understanding of how computers are constructed from gates and the hardware architecture of computers. They should understand machine-language programming, assembler code, calling conventions, virtual memory and protected mode operation, operating systems, compilers, linkers, loaders, interpreters, object-oriented languages, system calls, user-level libraries, XML, web servers, databases, networking, remote procedure calls, object request brokers, LDAP, and on and on.

With an understanding of how these various technologies function and fit together, writing code becomes an exercise in adding the "final touches" to a system. When something fails to work as expected, understanding all levels of the system helps the programmer to find the problem and correct it. More importantly, it gives the programmer confidence that the problem is not some random, unexplainable occurrence but rather a logical consequence of something going wrong somewhere in the system stack.

Programmers who fail to have this understanding tend to try random "corrections" to a program until they find something that seems to work. Without understanding why it did not work before, nor why it works now, the code is very fragile. Future programmers working on that code may easily break it. Programmers who have a solid understanding will track down the issue and resolve it.

Well-trained programmers will think of their programs in logical terms, not in operational terms. Thinking operationally means understanding a program in terms of its flow of control. First this happens, then this, then this, and so on. Thinking logically about a program means understanding the program as a *predicate transformer* independent of flow of control.

A *predicate* is a logical expression characterizing the state of a system. The logical way of looking at a program is as an entity that transforms these characterizations.

Given a predicate that describes the state of the system before program execution, the program will transform that predicate into another which gives the state of the system after the program executes.

Each statement in a program, in turn, can be viewed as a predicate transformer as well. In a correct program, each sequential statement transforms the starting predicate (the *pre-condition*) closer and closer to the ending predicate (the *post-condition*).

Subroutines especially can be viewed as predicate transformers that given a pre-condition describing the system before execution, will guarantee a post-condition describing the state of the system after execution.

For example, consider the following simple example to illustrate the point. It's a fragment of Perl code for printing out to a Web page the first 5 elements of an array:

```
pre-condition: array has >= 5 elements
my $elementsLeftToPrint = 5;
foreach my $element (@array) {
    invariant: # of elements printed + $elementsLeftToPrint == 5
    print "$element<br>\n";
    $elementsLeftToPrint--;
    last if $elementsLeftToPrint == 0;
}
post-condition: # of elements printed == 5
```

The operational way to think about the program is to mentally step through the code, line-by-line, and count on our fingers each time an element is printed.

The logical way to think about the program is in terms of its pre and post condition predicates.

The desired post-condition is for 5 elements of the array to be printed. Given this code, the weakest necessary pre-condition for this to be achieved is that the array has 5 elements. If it does not, the loop will exit early with **$elementsLeftToPrint** non-zero, and the post-condition cannot be assured.

The way to reason about this program is to think about the *loop invariant*. The loop invariant is a predicate that is always true at the top and bottom of the loop (in the middle of the loop body it can be violated for a time).

If the loop invariant **# of elements printed + $elementsLeftToPrint == 5** can be proven to hold, then when this is combined with the exit condition

$elementsLeftToPrint == 0 it implies (mathematically) **# of elements printed == 5** which is the post-condition we are trying to establish.

To convince oneself of the loop invariant is an exercise in mathematical induction. Assuming the loop invariant **# of elements printed + $elementsLeftToPrint == 5** is true at the top of the loop, then in the loop we print one item and decrement **$elementsLeftToPrint** re-establishing the truth of the loop invariant. Thus if the invariant is true at the start of any loop iteration, it is therefore true at the bottom of that same loop iteration as well.

On entry to the loop nothing has been printed and **$elementsLeftToPrint** was initialized to 5, hence the loop invariant is true on entry. Given it's true on entry, it is therefore true at the bottom of the first iteration and hence at the start of the second. Given it is true at the start of the second iteration it is therefore true at the bottom of the second iteration and hence the top of the third as well, and so on. By mathematical induction, we have proven that the loop invariant holds always in this program.

This was the final step in proving that the program does what it claims to.

A programmer trained in computer science at the University level will have an appreciation for this type of reasoning. While they will not prove all their programs correct in this manner, the good ones will think in this manner as they program, which will benefit correctness and clarity.

Understanding that a subroutine is like an extension to the programming language's statement set, this type of thinking will benefit

the design and documentation of the interface. A sub-routine will be thought of and commented in terms of:

- what must be true of the parameter values for the function to behave correctly (the pre-condition);
- given the pre-condition, what will be true after the subroutine is called (the post-condition).

The programmer will tend to avoid the use of global variables and will carefully document the meaning of every parameter and the return values.

14.3. Coding Standards and Metrics

Most organizations have some form of coding standards to help to preserve the clarity of the code. Coding standards should cover:

- The hard tabstop setting and the size of indentation
- Bracketing conventions
- Commenting conventions
- Variable naming conventions
- Maximum sizes for subroutines, files, classes
- Avoidance of cloned code.

The coding standards should be written and published and adhered to. Ideally, the development environment should auto-format the code according to the local coding conventions. Reviews and/or buddy walkthroughs can be used to check for non-compliance. Checkmarks on the feature records can be used to indicate that a coding standards review has taken place.

Every class and method should have comments. The comments for a method should refer to every parameter by name. These comments should be written using technology that can extract them into html documentation (*e.g.*, javadoc, doxygen, pod, and so on).

Variables, constants, classes and methods should be named according to their function. Variables should not be re-used for more than one purpose. Constants should always be used in preference to literal values in code and named well. Longer names should not be discouraged if necessary. Some conventions require naming variables according to their type (*e.g.*, so-called "Hungarian notation"). This is not very helpful and leads to obscure names.

Comments should be used sparingly within subroutines, however not neglected either. Generally, it is good to comment a block of code designed to accomplish some distinct purpose. Also, a particularly tricky line of code that cannot be re-written more simply should be commented.

If subroutines get too long it is usually for lack of an appropriate level of design thinking. Therefore, the sizes of programming entities such as subroutines, files, and classes should be constrained.

Cloned code is identical code copied from one place to another. Cloned code is notoriously error-prone, as several months later nobody will remember about the cloning, and a change in one copy will not get changed in the other, leading to a problem. Cloned code should not be tolerated, and particular attention should be placed on avoiding it anywhere.

Even two branches of an **if** statement should contain no cloned code. For example,

```
if( $count == 0 ) {
   print "<table border='1'><tr><td>none\n";
} else {
   print "<table border='1'><tr><td>$count\n";
}
```

should be replaced by the non-cloned version:

```
print "<table border='1'><tr><td>";
print ($count==0) ? "none" : $count;
print "\n";
```

14.4. Architectural Clarity

The biggest first-step in having a clear architecture is to organize the source code appropriately into files and directories in such a way that the directories correspond to logical modules.

Each module should then be documented with its purpose and how it fits into the overall architecture: what other modules it is allowed to access, what other modules access it.

This organization then provides the underlying structure for a reasonable architecture document that describes this module architecture at a high level.

Additionally, architectural documentation should describe the strategies for storing persistent data, and the run-time organization of the system: what processes and threads are running and how they communicate.

Going through this exercise will reveal which parts of the system are architecturally murky.

14.5. Architectural Degradation

Over time, systems tend to deteriorate in architectural integrity. This is because programmers come on who either don't understand the current architecture, disagree with it, or simply don't care; and they start making changes to the system. Over time, the architecture becomes murky.

It is a full-time job for one of the programmers, generally designated "Chief Architect," to stay on top of the architecture, to consult with other programmers on how they intend to effect changes, and to review all code going into the system for adherence to the architectural direction. Sometimes, such a person might observe some code being checked in and will overnight re-write it to conform better to an architecture. Entire systems worth hundreds of millions of dollars in revenue have been written in this fashion.

Another reason for architectural murk is an architectural idea that didn't work out, or a changed direction for a system. In this case, the system is saddled with an inappropriate architecture. As programmers begin fixing the defects and adding new features, the bad architecture becomes entrenched.

Generally, systems with a good clear architecture are controlled by a single individual involved for a long time with the system.

Even under the best of care, however, the architecture of a system will tend to degrade over time.

One mechanism to overcome this issue is the so-called "architecture tax" in the agile horizon planning methodology. The idea is that if the total capacity in a plan is, say, 500 ECD's, then a percentage of that is taken off the top and reserved for architectural maintenance and cleanup. For instance 10% or 50 ECD's. Marketing product management is then given 450 ECD's for new features. 50 ECD's, are reserved for architectural "features". These features can have no direct impact on the end-user experience. They must only be used to change the code in order to improve the architecture of the system. Decisions on how to spend those ECD's are made by the chief architect.

Using "The Tax" offsets the inevitable architectural degradation that takes place as many coders work on a system over time.

Another way to maintain the architecture is a commitment to not accept "hacks" to implement features quickly. In many cases there is a way to code a feature that is quick but dirty (*i.e.*, doing it would lead to architectural regression). There is always a right way to do something like this, but it would cost more in terms of ECD's.

To maintain the architecture it is important that the development department, in giving estimates, always gives the "correct" estimate for the feature done the right way. Giving the company a quick hack might seem to be beneficial, but it always costs in the long run. It is also un-professional.

Professional developers must balance this by realizing that "the absolute best way" does not exist. They need the judgment to distinguish "gold plating" from "done right" from "quick hack". Professionals will eschew equally the gold plated version and the quick hack.

However, when a developer's professional judgment is that a feature will take 10 ECD's to do right, they should stick by their guns and not even consider the 2 ECD quick hack. Mostly, business people don't want all the details, they just want to know how long a feature will take. For this example, 10 ECD's is the right answer. There is no need to say "well, we could do it in 2 ECD's but it’s a hack". Just say 10 ECDs.

There will be occasions when a quick hack is necessary to save the company's bacon. Developers also need the business judgment to see when this is the case, and only then suggest the quick hack. The quick hack should then be done and released quickly, but not mainlined. The mainline code should be written to implement the feature the correct way so that the hack does not live on.

14.6. Summary

In this chapter we considered the matter of architectural and code clarity, why they are vitally important, and schemes for building and maintaining clarity.

15. The Software Vendor Business Environment

Up to now we have been discussing technical aspects of software development and management. In this chapter we begin a study of how a software development department integrates into a business.

15.1. Managing

Managing a software development organization consists of four distinct activities:

- Managing downwards:
  dealing with the software organization itself.
- Managing outwards:
  dealing with other groups within the company.
- Managing upwards:
  dealing with organizational superiors.
- Managing externally:
  dealing with customers, partners, and investors.

When most people think of managing a software organization, they think primarily of the first item on this list: managing the software organization itself. While important, it is not possible to be successful at it unless we also are successful with the other three.

To see why this is so, and to begin to understand how to be successful at this multi-faceted approach to managing the software

organization, we must understand the operation of the company in which the software development organization exists.

In this chapter, we will explain how a prototypical company involved in software development operates. As the focus of this book is primarily on managing within the context of a software vendor organization, we will discuss only this business context. Those readers concerned with other business contexts will see the parallels for themselves.

15.2. The Software Vendor's Business

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

The key ingredient is the software itself. The software must satisfy some need or some want of one or more target markets. Moreover, the combination of market size and willingness to pay must be sufficient to pay the software vendor's costs, plus extra on which to re-invest in growth and (ultimately) provide profit for the investors.

Customers will license the software from the vendor, which gives them the rights to use the software within certain constraints. The software vendor company will not often *sell* its software outright, as the implication of this is that the buyer would gain all rights to the software (and, in particular, the right to license its use to others).

Individuals and smaller customers will purchase a license to use one copy of the software. Larger organizations will buy multi-user site licenses, which is essentially a bulk discount provided by the software

vendor. Usually single-user and smaller multi-user licenses have a non-negotiable licensing fee associated with them. The vendor will establish the cost of larger site licenses through a negotiation process with the customer that will deal with not only license cost, but also with other questions such as the cost of upgrades, the level of service the customer can expect, consulting help, and so on.

While initial license fees from new customers fuel the growth of the software company, follow-on revenues from existing customers is the ultimate goal.

These follow-on revenues consist of the purchase of additional licenses, the purchase of licenses for related software products, the purchase of upgrades to the software, and recurring maintenance revenue. Maintenance, charged annually at perhaps 20% of the license fee, entitles the customer to ask questions of the help desk, to get priority problem resolution, to receive periodic bug-fix releases of the software, and entitles them to periodic minor upgrades to the software. Major upgrades may not be included under some maintenance agreements.

These follow-on revenues associated with successful products provide a stable, long-lived, and predictable core of revenue for the software company, and can fund the development of new products. We sometimes refer to products that have achieved this status as "cash cows" (in that after the cow is bought and fully grown, it provides the farmer with a continuing supply of profitable milk for little ongoing cost).

The reason why good software provides ongoing revenues is that when a customer buys into a software product, their investment usually goes far beyond the cost of acquisition. They commit to learning how to use the software, and they commit to integration work to make the software work in their environment. Thus, once the customer chooses to acquire software, and once they have fully accepted it into use, they will be reluctant to switch to a competing product, and will want to capitalize on their all-in investment to-date by buying incremental products and services that add value atop their initial investment. They will therefore expect the vendor to keep supplying the service, incremental upgrades, and related software solutions that will keep them satisfied with their initial purchase decision.

The implication of this is that the software company must not only sell its software, but also ensure that the customer gets it into production (lest it become "shelfware": purchased software that is never used).

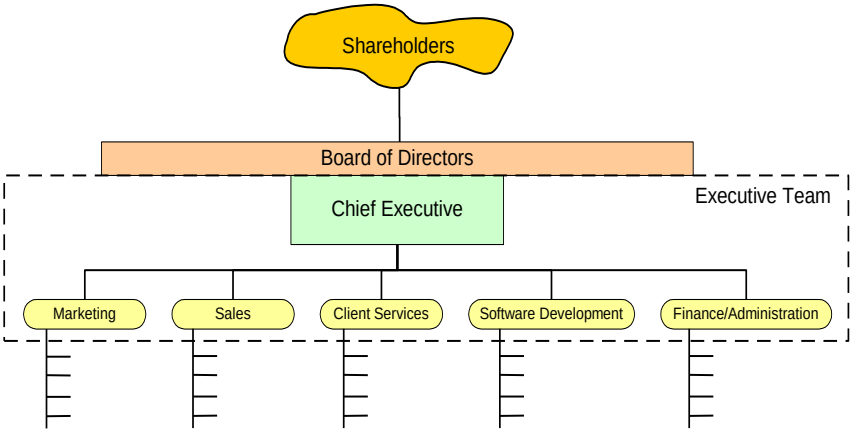
As we see from these considerations, the generic mission of the software vendor is as follows.

- To produce software that meets the needs and/or wants of their target market sufficiently well that enough customers will buy the product at a given price.
- To provide related services to assist their customers in putting the software into production and effectively using the software on an ongoing basis.
- To make available incremental upgrades and new, related products that enable customers to capitalize on their initial investment.

If the company fails at any of these goals, their products will not become ongoing sources of revenue, which is the ultimate goal. The customers may even cease using the vendor's software, which given the size of the customer's all-in investment is usually a serious indictment of the vendor's product.

15.3. Software Vendor Structure

In order to fulfill its mission, a mid-sized software vendor company will typically organize itself around a small number of functional units. While no two companies are the same, an industry-wide average for the structure of a software vendor would be somewhat as follows.



The **shareholders** are the ultimate owners of a company. In a *publicly traded* company, anybody can buy shares. In a *privately-held* company, people can only buy shares with the approval of the existing shareholders. The typical situation in a privately-held company is that

the founders of a company start with all the shares, and they then give some of these shares up in exchange for financing (to a venture capitalist, for example), or as an incentive to their hired staff.

The shareholders elect a **Board of Directors** to represent their interests. In a closely-held private company (for instance, where only a lone founder has shares), the board will often act in an advisory capacity. In a public company or a private company with several large investors, the board plays a more significant role, and votes on important decisions according to the company's written constitution. In particular, the board appoints the chief executive (who is often then brought in as a board member as well) and the rest of the executive team.

Board members are also responsible for the correct conduct of the company according to any applicable laws of the land, and can be held personally liable in case of certain gross violations.

The board will typically meet on a quarterly basis (once every three months or so) to review the status of the company, offer suggestions and advice to the chief executives, and set expectations for future performance. As well, there are certain decisions, such as the issuing of new stock for example, that only the board and shareholders can make.

Board members are generally a company's most important founders, plus representatives of significant stakeholders, plus prominent and experienced businessmen able to offer advice and provide business connections. The company (on behalf of its shareholders) will typically pay board members a stipend and issue them some stock in exchange for their services.

The **chief executive** is in charge of running the company on a day-to-day basis. He or she will assemble an executive team (subject to approval by the board) and lead them. Sometimes the chief executive (or CEO, for Chief Executive Officer) will split their job in two, with themselves concentrating more on external relations and hiring another to oversee internal company operations. In such cases, the chief executive will retain the title of CEO, and designate the other as either *President* or *COO* (Chief Operating Officer).

The chief executive is responsible to, and reports to, the board of directors. If the board is dissatisfied with the CEO's performance, they can elect to replace him or her. In particular, the CEO must commit to certain financial targets on gross revenue and profits, and then deliver on them.

The **executive team** is composed of the senior executives within the company. The executive team will cover all areas of responsibility with no gaps and no overlaps, thus providing clear accountability. Each member of the executive team (other than the chief executive) has a defined area of responsibility and a designated staff and budget to carry out those responsibilities. In addition to these individual *functional* responsibilities, they also have a responsibility to contribute to the management of the company as a whole by means of their participation on the executive team.

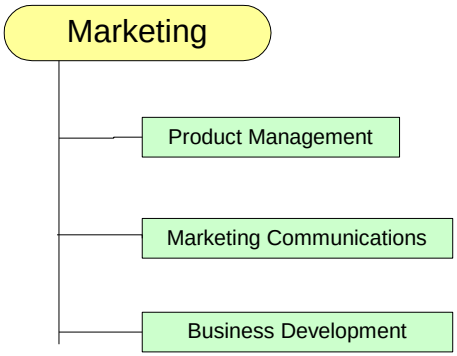
The members of the executive team will often have the title of "Vice President". In companies suffering from title inflation, the executive team members may carry the title "Executive Vice President", leaving the title of "Vice President" available to attract more junior managers who will not participate on the executive team.

The executive team will meet weekly, reviewing status, coordinating activities, and making important decisions. The strength of the executive team is critical to the operation of the software company. A weak executive team can ruin a company. A strong team can make it successful.

In the remainder of the chapter, we will look in detail at the individual business functional areas shown on the preceding diagram.

15.4. Marketing

The **marketing** group divides itself into the three distinct areas of responsibility shown below.



The **product management** group within marketing is responsible for identifying market segments and determining what software products (or enhancements to existing products) and related services the company should be providing. They will do this both at a strategic,

long-term level, and at a more detailed, tactical, feature-by-feature analysis level. Product management should decide on the timing of new releases, how much to charge for them, and what features should be included in them.

There will be a Director of Product Management in charge of the group, and a number of product managers reporting into him. Each product manager will manage an assigned portfolio of the company's products. The company holds product management accountable for the profitability of the products they manage.

As part of their jobs, product managers will spend a lot of time talking with customers, understanding their needs. Making them the go-to point for all requests for new functionality in the products they manage facilitates this. The product manager will maintain a list of all these requests, and coordinate (and, ideally, lead) the activity of deciding what set of features will be included in the next release, driving the company to a participative consensus. The company expects product managers to bring sound, business thinking to this problem, and not only unbridled creativity (of which there is usually plenty elsewhere in the organization).

Product managers must do their research, understanding how the competition compares, and making an educated guess as to where the market is heading.

To successfully release a product, it is necessary for all the various groups in a company to coordinate their activities. Often, product managers will be responsible for overseeing all the activities, across all the groups, involved in the release of a product. They will act as a coordinator and large-scale project manager, raising the red flag to management if certain groups appear to be slipping behind schedule.

Finally, product managers must develop the written content for promotional materials, such as advertisements, brochures, and white papers, that explain the benefits of their company's products to their various target markets.

Product management is one of the key jobs in the software vendor company. Good product managers are hard to come by, but very valuable when found.

The **marketing communications** group (called *marcom* for short), are in charge of communicating with the external world. They do this by means of advertising, web presence, press relations, trade shows, and promotional literature. Marcom is in charge of the company's overall image.

Marcom will work with advertising firms, design firms, and press relations firms to achieve their mandate, which is to generate "buzz": a lot of people talking about the company in a lot of places.

Marcom will generally not be experts on the company's products and markets. They will rely upon product management to fill in these gaps. Rather, they will be expert in knowing how to reach people with a message.

The company measures Marcom in terms of the number of sales leads they generate, and the number of column-inches of publicity appears because of their efforts.

The **business development** group (spelled *busdev* and pronounced "bizdev" for short) is in charge of developing new business opportunities for the company.

This is usually a small group, and by no means always reports into marketing. Often busdev is its own functional area, independent of marketing, with its own Vice President reporting into the CEO.

The main function of busdev is to work with other organizations, striking strategic alliances, helping to negotiate acquisitions, and running partner programs.

Especially important is for this group to develop alternate *channels to market.* For example, if there is a market for the company's product in Taiwan, and yet the company is leery of opening a sales office to service the region, busdev might negotiate a deal with a company that already has a presence in Taiwan to resell the products and offer first-line support.

As another example, suppose there is a software product from another vendor that customers often use in conjunction with this company's software. In such a case, busdev might negotiate a deal, whereby the complementary company can resell their software as part of a "productivity bundle".

Busdev must be excellent negotiators, be good networkers, and have sound, business minds. Increasingly, busdev is becoming a critical part of a software company's overall strategy in attacking their markets.

15.5. Sales

The **sales** group, typically organized by sales region, is responsible for the company's revenue targets. There are three main approaches to selling:

- **High-Level Direct Sales**
  Sales people identify individual prospects, visit with them,

organize demonstrations, negotiate terms with them, and consummate sales.

- **Dialing-For-Dollars**
  Sales people get leads from marketing initiatives and call out to prospects. They make many calls a day and hope to close a few sales quickly.
- **Channel Sales**
  Sales occur indirectly via alternate channels to market, such as through distributors local to a geography.

Sales people are motivated by means of a commission. A commission is a bonus paid to the sales person worth a certain percentage of the revenue when the business is closed.

Often commissions are 100% revenue based. The sales person will receive the same commission regardless of whether the deal is profitable for the company. There are therefore controls in place through sales management to ensure that prices are not discounted too heavily and to ensure that only an appropriate amount of time and money go into each sale.

One danger for the software development department is to have their people pulled into sales opportunities on a regular basis to assist in the sales process. To a certain extent, this is normal. However, unless it is carefully controlled, the sales person, motivated by revenues and not costs, will demand more time than it is reasonable to spend on any given sales engagement.

15.6. Client Services

The **client services** group is responsible for helping customers get up and running with the company's software. As such, they provide pre-sales support, training, help desk services, and consulting services. They are ultimately responsible for customer satisfaction.

Often, because of the problem of sales pulling too much resource from software development, a *pre-sales support* team will be formed within the client services organization. This group will be composed of technically-oriented non-developers who understand the company's software very well, and understand the typical process by which a customer will implement the software into their environment and make the best use of it.

15.7. Finance and Administration

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 usually report here as well.

15.8. Summary

In this chapter we set the business context in which the software development organization functions.

In the following chapter, we will see how software development integrates into the rest of the organization from a fiscal perspective.

16. Business Planning

Software development is a cost center. Other departments will take the software and sell it and claim the revenues. These are the profit centers. A profit center can say "I'll spend this extra money to generate more sales and will come out ahead in the end". A cost center cannot ever say this. The best they can say is "I'll spend this extra money and produce more and better software". The former is easier to judge than the latter. What is the meaning of "better"? What is the meaning of "more"? How is this measured? How much money should it cost to make "more and better" software? What is reasonable?

The truth is that "more and better" cannot be effectively measured or quantified. Nor can a dollar amount be attached to them in any meaningful manner even if they could be measured.

As a consequence, software development organizations must construct their arguments for their cost budgets in other ways.

In this chapter, we will discuss how this is done.

16.1. Proposals

Once there was a small software business that lacked effective source code control and defect/feature tracking. The development managers told their new VP that they had repeatedly asked the CEO for money for such systems but that they had always been refused. Within a month, the new VP had acquired the necessary budget to put such

systems in place. The development managers were stunned! They asked what mysterious Svengali-like power the VP possessed to get the money they had been repeatedly asking for. He had written a proposal.

Technical people tend to think that if they clearly need something, they should get it. If they don't get it, the powers that be aren't really interested in seeing the venture succeed and/or are too technically illiterate to realize the right thing to do. Whenever something goes wrong as a lack of these things they need, they will say "see, I told you we needed that". This is unproductive.

Technical people must convince themselves that unless they put a written proposal together, they are not really asking for anything. They are just griping.

Even with a well thought-out and well-researched written proposal, nobody can guarantee that the budget for the initiative will come or will not be cut back from what the proposal suggests. However, it is safe to say that without such a proposal there is no good chance of getting the budget.

Management will not allocate money to a venture that it does not believe has been thought through. Doing the necessary research and thinking, and then proving it by writing a proposal will provide management with the confidence they need to approve expenditures.

The managers from whom the budget is being requested are responsible for that money. If they approve a budget on the basis of "trust me" alone, they should probably be removed from that position of responsibility.

A proposal should set out the goal of the project, the expected benefits, the monetary costs, the amount of people's time it will take to implement and run, and a detailed timeline divided into phases. The proposal should list several different means of achieving the goal, compare them, and come up with a recommendation. If external systems are part of the proposal, the proposal should state which systems were considered, compare the strengths and weaknesses of each, and give the prices quoted. All costs should be included, both setup and ongoing, both in money and in time.

Following on from a well-written proposal, good managers will want to discuss ways of trimming costs from it and achieving the same, or only somewhat reduced, results. There are often ways of cutting costs out of a proposal without compromising the essence of it. If money is tight, the proposal writer should work cooperatively with the manager to see what can be done.

For example, a production-grade server might cost $10,000. However, a $5,000 computer might be adequate to run the systems, with the only risk being a failed power supply or NIC where the production server might have had redundancy. The risk is therefore being down for a day. It may be that management is willing to take that risk in exchange for the savings.

While a proposal is an essential part of acquiring budget for an initiative, it may be that the timing is unfortunate. The timing is unfortunate if for some reason the business feels it cannot afford an expenditure at that time. To deal with this requires a rudimentary knowledge of what motivates corporate budgets.

16.2. Corporate Budgets

Most corporations have more than enough money to fund any given initiative at any given time. However, for some reason they always seem to say that they don't. To understand this requires understanding what motivates corporate budgets.

Businesses operate to a budget. The budget says, for the fiscal year, what are the expected costs on a month-by-month basis, and what are the expected revenues. The surplus of revenues (money taken in) over costs (money flowing out) are the profits. The percentage fraction of revenues that the profit represents is called the profit margin.

The business goals behind the budget are fueled by the desire to add value to the business. The value of a business is how much it can be sold for. For a publicly traded business, the public is constantly giving their opinion of this as they agree on prices at which to buy and sell the company's stock. However, people's opinion on how much the business is worth depends upon the business's financial performance (and other, less tangible, qualities).

For privately-held companies, the goal is similarly to build value in the business. Financiers will typically use multiples of yearly revenue combined in some manner with profit margin to ascribe a value to the business. The multiples used will be determined by looking at comparable companies and how much they were sold for (or are trading at).

For example, if a private company's revenues are $5M per year, and if a company in a similar space with $10M revenue was sold last year for $50M, then the multiple of revenues is 5x. This would imply that

the business is worth $5M x 5 = $25M, all else being equal. This represents the amount of money the owners can sell the business for.

The best-case scenario for any business is to have shown steady revenue growth with healthy profit margins. In the software business, healthy profit margins should be in excess of 15% and healthy revenue growth should be in excess of 50% per year.

However, it is not only the final results at the end of the fiscal year that count. If revenues or profit are down in a given quarter (the fiscal year is divided into four 3-months *quarters*) that may be taken as a bad sign and reduce the value of the business.

Prudent managers will also be tracking their revenues and profits on a month-by-month basis. If a month falls short of projections, that may cause a change in behavior in a business. Good managers know that bad years are made up from bad months, and will not let a bad month slide without taking some action, which may involve cutting spending.

On the other hand, a particularly good month will not be seen as a spending spree. That month will be put in the bank to offset any potentially bad months later on in the quarter or the year. If all the rest of the year is to-target or exceeding target, the good months will accumulate to being a good year. The owners, via executive management, may show their gratitude to the employees via bonuses, and/or to stock holders via a dividend.

Thus even though there may be plenty of cash available to pay for an initiative, the numbers may be such that it cannot be spent in that month, that quarter, or that year.

16.3. Funding Initiatives

A good way around the numbers is if cost offsets can be found. For example, if a new hire was in a budget, then delaying the hire by 2 months might save the company $20,000 that can be re-directed to a new initiative.

A development manager may even suggest reducing her staff by one in order to pay for an initiative. In a staff of ten, for example, the benefits accruing from an initiative might offset the productivity lost by letting go the poorest developer (in fact, sometimes this can gain productivity!).

Note however, that money saved in one month is not necessarily available for spending in a different month. This must be discussed with the business manager.

Once budget is allocated to a department, if mid-year that department can make a case that spending the money differently than was originally foreseen is more efficient, then this is a good business argument and one that is easy for management to agree to. Efficiency here is measured against corporate goals. If the corporate goals that were originally agreed upon can be reached to a greater extent with less money by spending it differently, then that is more efficient. On the other hand, if the argument for the money involves changing one corporate goal for another, then this is a more difficult argument to make, and may not be successful.

For example, if the corporate goal was to produce more features faster, then trading off a new hire in development for a better regression testing system is a more difficult argument to make.

It may be the case that budget for an initiative is agreed to, but when it comes time to commit to spending it the situation may have changed. It might be that a different department has overspent for that month[*], or revenues are lower than expected that month, and hence the software development department may be asked to delay an initiative or reduce its cost further.

These things happen and the mature manager will take it in stride, re-group, and come up with an alternate plan.

The best way to fund an initiative, however, or to have money in budget to trade-off at all, is to get pride of place in an annual corporate budget. That involves participating effectively in the budget-making process, which involves creating a good software development department business plan.

16.4. The Annual Budget Cycle

In most companies there is an annual exercise near the end of the fiscal year whereby a new budget is arrived at for the following year. This initiative is usually driven by the CEO and the VP Finance, and involves significant contributions from the rest of the executive team (the executives are the top functional managers who report into the CEO: VP R&D, VP Sales, VP Marketing, VP Client Services, and so forth).

The purpose of the exercise is to come up with month-by-month revenue projections and month-by-month cost projections. The budget

[*] probably the Marketing department

is then tracked against actuals as the year proceeds, and the CEO will make adjustments as necessary.

Input for the revenue side comes from the sales and marketing functions, together with targets set by the CEO. This is backed by a sales and marketing plan with a certain amount of cost in order to drive the desired revenue.

Other departments must also come up with annual business plans that clarify the status quo, say how much money is required to maintain the status quo, and proposes new projects to improve upon the status quo.

The CEO must weigh the initiatives proposed by all the functional areas to come up with a final budget that gets translated into detailed financials by the VP Finance.

In the next section, we shall look at the ingredients that make for a business plan for the software development department so that they can compete effectively in the CEO's mind for budget.

16.5. The Software Development Business Plan

The purpose of the annual business plan (which may be revisited more frequently than annually, but will have a horizon of one year) is to help the CEO make tradeoff decisions about how much budget to allocate to one department or another during the next fiscal year.

The document should therefore be written in such a fashion so as to facilitate this task. The following information should be included.

16.5.1. Introduction

The introductory section of the document should restate the mission of the department, and then drill down into the details of all the various products that are under active maintenance, describe all the various platforms that are being supported, describe all the new product initiatives that are under way, and any process oriented initiatives that were started but have not yet been completed.

This is done in order that the readers of the document are fully apprised of all the significant ongoing commitments required of the department.

The introduction should summarize accomplishments in the department during the previous year. This is done so as to give a sense that money for new initiatives is well-spent.

It should self-assess the department in terms of process maturity, and indicate where the department needs to improve with regards to process maturity and tools.

Finally, it should give a directional statement indicating where are the areas for improvement, and what departmental goals will form the basis for the requested budget.

For example, if the development department has understood from the rest of the management team that software quality is an issue, this should be identified as a key area for improvement with specific initiatives being proposed for improving quality. If on the other hand, the VP Software Development understands that the pace of development should be increased, they will propose projects that will enhance that.

In order to understand which way the business wants the software development department to move, the VP must be in constant communications with the rest of the executive team and the CEO to formulate ideas for where the department needs to improve in order to better support the business.

Even more importantly, the VP Software Development should have considerable customer contact to understand first-hand areas for improvement important to customers. This input is valuable for suggesting corporate goals that will align with customers and hence improve customer satisfaction, product attractiveness, and thus sales.

It is wise to come to a clear meeting of minds ahead of time with the CEO and the rest of the executive team as to where the software development department should head. Without this, the business plan may make a great case for something the business has no interest in doing.

The annual business plan is a valuable opportunity for the VP and the department to re-align themselves with the goals of the company, and to make it all explicit and carefully thought through by means of the act of writing it down.

As well, it is a valuable opportunity to reflect on progress over the past year, and to decide on departmental goals for progress during the next year.

16.5.2. Baseline Budget

The baseline budget is the current status quo for the department. If nothing is changed (no new initiatives engaged in, no hiring, no firing,

no new consultants, and so on) then the baseline budget is what would be spent throughout the fiscal year.

The section spelling out the baseline budget should give a breakdown of the salaries of staff and how they are allocated both by functional area (coding, testing, build, documentation, management, and so on), and by product. This requires a spreadsheet that allocates people's time to functional areas, and to products. Only the final percentage breakdowns should be shown.

The breakdowns will then require a few paragraphs of explanation, describing how the numbers were arrived at. This breakdown is especially useful for the CEO to compare budget allocated to product, versus revenues expected from products.

Various staffing ratios (in time, not dollars) should also be produced. The ratio of testers to coders (by product) and the ratio of documentation to coders (again by product).

These ratios can provide justifications for projects to increase the number of testers, for example.

The most important part of this section is the baseline budget in dollars required to keep the department operating at the same level throughout the next year as it was operating at the end of the last month of the previous year.

Staff costs are the most important component of this. The costs include both the salary paid to the employee, and the overhead costs necessary to pay for their benefit plans and to provide them with the tools and environment they need to be productive.

VP Finance should be able to supply to the department a quick estimate of overhead costs. For example, if overhead is 50%, then a $60K salary actually costs the company $90K after one takes into consideration phone bills, computer leases, rent, utilities, and so on.

This is called the "fully-loaded" cost. To be better comparable with the costs of other initiatives, salaries should use the fully loaded cost. However, when high-level budgets are eventually translated into detailed budget spreadsheets, the overhead may not be allocated to the software development budget (*e.g.*, the phone bill may be paid from the General and Administration, or G&A, budget line).

In a growing company, the baseline budget will typically be higher than last year's entire budget. When staff is added, they are added throughout the year and hence their salaries are only paid for a part of the year. For the next year, however, those staff will be paid for the entire year.

The importance of the baseline is to show the CEO that the baseline is the bare minimum budget that can be considered unless a contraction is desired. It indicates to the CEO that fiscal support for the department's growth and improvement must be incremental to the baseline (which is already higher than last year's budget). If the CEO allocates $0 above the baseline, he is indicating no fiscal support for improvements in the department. He may still expect improvements via increased efficiency, but will be providing no fiscal support.

It also helps to make clear to the CEO and the rest of the executive team how money is currently being allocated. Often, when the team sees how software development resources are divided out into the

various company commitments, it becomes clear how little resource is allocated to key new business initiatives. Desiring no cuts in the other commitments, this is a powerful argument for more budget for the new initiatives.

16.5.3. Organizational Structure

The business plan should have a section describing the current organizational structure of the department.

It should identify roles and titles, and explain them in terms of their scope of responsibility.

If the department is sufficiently small (less than about 50), the organizational structure can include all staff. If larger, then only down to the management level. This helps other executives to see what the job responsibilities are for the various people they may encounter at company picnics.

If changes in the organizational structure are contemplated, then these should be shown (including where proposed new management hires would exist in the responsibility structure).

Executive management is a lot about "responsibility design". The VP should divide the department into sub-groups, each with a manager (usually titled "Director") as its leader. Each of the sub-groups must be assigned a scope of responsibility. The Directors of the sub-groups must be given appropriate leeway to succeed in their responsibilities. The VP will generally assist the Directors and be closely involved with their decisions to further sub-divide their areas into smaller sub-groups, each managed by its own manager reporting to the Director. This repeats until the groups have five or six people.

The VP must think in terms of handing general responsibility for an area to a Director. Consequently, Directors must be hired who are capable of taking on that responsibility without hand-holding by the VP. As the organization grows, new areas of responsibility will come into existence, and these must be parceled out, resulting in re-organizations. As well, certain responsibilities will inevitably slip through the cracks, and nobody will take them on. As soon as the VP notices a situation like this, she must take action to re-assign or expand areas of responsibilities so that this lapse does not recur.

The annual business plan is a valuable time for the VP to reflect upon the organizational design of the department, and to consider changes to it.

16.5.4. New Project Summary

The goal of the annual business plan is to clearly enunciate the baseline budget for the new year, and then to propose a series of projects on top of that for the next year.

These projects could involve hiring staff, hiring managers, buying hardware, engaging consultants, buying software, and so on. However, they should not be stated in this manner. The new projects should be stated in terms of their benefit to the company.

For example, "improve co-ordination of documentation" project might involve hiring a documentation manager and buying some new software tools. Naming projects in these terms helps clarify their alignment with corporate goals. If inconsistency in product documentation is the problem, a project named in this manner will be more resonate than one called "hire more documenters". It makes it

clear that the goal is not in the hiring. Rather, that the goal is to solve the problem, which as a component might involve hiring.

As another example, a project called "increase the pace of product X development by 20%" might call for four new coders to be hired into the group along with two new testers and one new documenter (to keep the ratios status-quo).

A project called "improve quality" might involve hiring six new testers to bring the ratios to 2:1 coders to testers across the board.

Each of these projects should be named in this fashion and listed, with their associated costs given. The costs should be given in three parts:

- expenses this year
- capital spending this year
- implied baseline for next year

Expenses this year are the salaries, overhead, and consulting fees to be paid out during the budgetary year. For example if a project calls for a new manager to be hired midway through the year at a salary of $100,000 (implying a fully loaded cost of $150,000, for example), then the expense in the first year is only half that, or ($75,000).

Capital spending is money spent purchasing hardware and software. This kind of money is different than expense because finance can spread out the expense impact over several years (usually three for hardware and software). For example, if $30,000 worth of new hardware is to be purchased mid-year this corresponds to an expense of $5,000 in the first fiscal year. The hardware can be "paid for" over three years. The first six months of having it will therefore cost one sixth of that, or $5,000. Capital should be identified separately, however,

because there may be good business reasons to conserve capital different than the reasons for keeping expense down.

The final part is the amount of cost this project adds to the baseline. If the project involves hiring a consultant for a few months during the year, the impact on next year's baseline budget is $0. If however, the project involves hiring a new coder at $120,000 fully loaded cost in the last month of the year, the impact on the current year's budget is only $10,000. The amount it adds to next year's baseline is, however, the full $120,000.

The projects should therefore be given in a list, with business-relevant titles, and with the spending broken out as identified.

There should be a grouping of projects into related categories and an analysis. For example, projects can be grouped as "pace increasing projects", "quality increasing projects", and "management projects" and the costs shown by way of these categories.

This way the business plan starts to show how the proposed projects mesh with the corporate objectives stated in the introduction, and in what proportion budget is requested for each.

16.5.5. Project Details

Following the summary, the business plan should then go into detail on each of the proposed projects.

It should explain the project in more detail, explain how the money will be spent, and justify how spending that money will achieve the goals of the project.

16.6. Establishing The Budget Request

The VP should have a good idea before writing the annual business plan the approximate size the software development budget is expected to be for the next fiscal year.

There will generally be a series of meetings leading up to the initiation of the budget process. In these meetings, the executive team will come to an agreement on how large expected revenues need to be next year, and what is possible to achieve.

Once the expected revenues are decided, a target for profit should also be decided. Subtracting the profit from the revenue yields the expenses. Depending upon the business climate and the kind of company the organization wishes to be, the executive team will decide upon a percentage of the cost budget to allocate to software development (called R&D for consistency with other types of technical organizations – financial people usually do not distinguish between the types.).

For example, suppose the revenue target is $100M for the year and the target profit margin is 15%. This leaves a cost budget of $85M. If R&D expenditures are 18%, the target budget for the R&D group will be $15M. If baseline is at $13M, that will leave $2M for new initiatives to improve upon the baseline.

In these early meetings, the largest influencer of the R&D budget will be the percentage of costs allocated to R&D. The well-prepared VP will come into such discussions armed with information about what percentage comparable companies, and especially competitors, are spending on R&D. As well, the executive team will debate what is the

appropriate amount of spending for R&D for the company. In many cases, sales and marketing executives will be some of the most vocal supporters for more R&D spending, as new products drive new revenues.

But be careful what you wish for. If the VP R&D gets the extra budget, there will need to be something extra to show for it at the end of the year. Otherwise, those formerly supportive sales and marketing VP's will be lobbying for a change in R&D management!

The calculation above provides a target for the R&D budget. Generally, the business plan should include sufficient projects to exceed that target budget by 10% or so. If the CEO suddenly feels the need to increase R&D spending, there should be projects there.

On the other hand, the plan should also clearly indicate priorities. Projects making up target plus 10% should be proposed, but the plan should show which projects to keep (or how to scale them down) if at target, and if at 5% or 10% below target. There is no need to be concerned with making it too "easy" for the CEO to cut the R&D budget. The CEO will cut if she needs to. Better to be a team player and make it easy to see what is given up if 5% or 10% is cut.

16.7. Finalizing the Budget

Once the annual business plan from the software development department is available, the CEO will take that and other departments' business plans and create a high-level budget from them. When the numbers begin coming together into a larger picture, the CEO will

typically start feeling conservative, and will ask all the executives to take a second pass and cut out, say 5% of the budget.

After this, one department may make a more compelling case for not being cut than the others. Or, the CEO may have her own initiative in mind for, say, the marketing department, or for new business development, for example. In this case, the R&D department may be asked to cut further.

During this time, the VP R&D must be making the case for why the money is required in order to fend off initiatives from other departments and protect forward momentum in software development. Good relations and frequent communications with the CEO is necessary for this.

Towards the end of the process, the VP Finance will create detailed budget spreadsheets from the high-level plan formulated by the CEO. The executives will each be asked to check to ensure that the budget they have negotiated with the CEO is accurately reflected in the detailed spreadsheets. It never is precisely, and the VP R&D must work with the VP Finance and the CEO to smooth things out and bring everything in-line.

16.8. Summary

In this chapter we looked at how a software development organization integrates into a business from a financial perspective. We looked at the act of writing proposals, and how department budgets are made.

17. Concluding Remarks

This book has covered a lot of ground. In it is a lot of what can make a software development team and its managers successful.

We covered the requirements to practice a disciplined, process-oriented approach to software development that integrates well into the business environment. These requirements included the essential practices, which were,

- Source Code Control
- Defect and Feature Tracking
- Reproducible Builds
- Automated Regression Testing
- Agile horizon planning
- Feature Specifications
- Architectural Control
- Effort Tracking
- Process Control
- Business Planning

We concentrated especially on the agile horizon planning framework, wherein the common-sense balance between effort required and effort available is maintained throughout the release cycle.

It is our fondest desire that readers make use of the knowledge and experience in this book to elevate their chosen profession of software development.

Appendix A Sample Deterministic Agile Horizon Plan

The first sample agile hoirizon plan is a non-stochastic presentation that assumes all estimates are at an 80% worst-case level. Appendix B will present a stochastic version. Following the two plans, Appendix C provides a glossary of the underlined terminology used in the plans.

These plans can act as models for an organization building systems to implement the agile horizon planning methodology.

The plan is for a fictitious software company that makes and sells software that simulates the movement of stars and planets in the night sky

Appendix C should be referred to regularly as these plans are read for an indication of the meaning of the various terms.

Planetaria 2.4 Agile Horizon Plan

Planned Release Dates

Release:	2.4	
Design Start:	Mon Sep 5	
Code Start:	**Mon Oct 30**	(38 workdays)
Development Cut:	**Fri Feb 16**	(**74 workdays**)
Beta Availability:	Fri Mar 23	(25 workdays)
General Availability:	Fri Jun 1	(49 workdays)

Mission

This release will be a significant upgrade from 2.3. The ability to view the sky from various locations in the universe, the ability to cope with orbiting entities, and the ability to simulate the passage of time are three of the major upgrades in this release. Many other features will be included as well. This release will satisfy many of users' most frequent requests, and satisfy a key NASA request for realistically rendered planets.

Current Status Summary

Estimates: are made at an 80% worst case level.

As at: EOD Fri, Nov 10 2000
(**Coding** phase: **10** of 74 working days elapsed).

Remaining coding capacity:		394	effective coder days
Average coders:		6.8	effective coders per day
A+B Features	Remaining coding requirement:	400	effective coder days
	Delta:	-6	effective coder days
A Features Only	Remaining coding requirement:	334	effective coder days
	Delta:	60	effective coder days

Capacity

Estimates: are made at an 80% worst case level.

As at: EOD Fri, Nov 10 2000
(**Coding** phase: **10** of 74 working days elapsed).

Coder	*Class*	*Days*	*Vacation*	*Work Factor*	*Effective Days*	*To Date*	*w (act.)*	*A*	*A+B*
Philip	Mgr.	64	5	0.1	5.9	1	0.1	-0.1	-0.1
Tracy	Arch.	64	4	0.2	12	2	0.2	+10	+3
Sam	Lead	64	5	0.4	23.6	5	0.5	+4.6	-1.4
Al	Lead	64	5	0.4	23.6	4	0.4	-1.4	-1.4
Gil	Lead	64	5	0.3	17.7	3	0.3	+0.7	+0.7
Chris	Coder	64	4	0.6	36	6	0.6	+5	-3
Shakur	Coder	64	0	1.2	76.8	12	1.2	+10.8	-1.2
Helen	Coder	64	5	.6	35.4	7	0.7	+7.4	-2.6
Ted	Coder	50	4	1	46	9	0.9	+19	+11
Cal	Coder	64	5	0.6	35.4	5	0.5	+4.5	-1.6
Britney	Coder	60	3	0.7	39.9	7	0.7	+3.9	+3.9
Bob	Coder	64	4	0.7	42	7	0.7	+6	-2
totals:			49	6.8	394	68	6.8	+71.3	+6.7
					Unassigned:			-11	-11
					Final Totals:			+60.3	-5.7

Requirement

Estimates: are made at an 80% worst case level.

As at: EOD Fri, Nov 10 2000
(**Coding** phase: **10** of 74 working days elapsed).

fid	*description*	*prereq*	*prio*	*assigned*	*status*	*to date*	*remain*	*spec*	*design*
345	show angular separation		A	al	DONE	4		y	y
304	Field-of-View indicators		A	helen	CC	7		y	y
234	NGC/IC objects		A	brit	WIP	7	18	y	y
389	time simulation		A	sam, cal	WIP	10	27	y	y
230	change location		A	helen	NYS		5		
704	set location to city	230	A	bob	WIP	7	3	y	y
298	set elevation	230	A	helen, cal	NYS		21	y	p
239	set location to planet	230	A	sam, brit	NYS		19	p	y
456	set location from map	230	A	bob	NYS		11	y	y
301	orbit framework		A	bob, al	NYS		32	y	y
303	orbit display	301	A	chris	WIP	6	9	p	y
906	orbit editor	301	A	shak	WIP	12	8	y	p
959	proper motion		A	phil, brit	WIP	1	14	y	y
508	selective constellations		A	tracy	WIP	2	2	y	y
102	images for objects		A	shak	NYS		10		

fid	*description*	*prereq*	*prio*	*assigned*	*status*	*to date*	*remain*	*spec*	*design*
294	show planets	102	A	ted	WIP	9	27	y	y
459	rendered planets	294	A	good	WIP	3	17	y	p
873	planet atmosphere	459	A	shak	NYS		8		y
939	custom images	102	A	shak	NYS		5	y	p
986	constellation boundaries		A		NYS		8	y	y
934	classical constellation		A	al	NYS		15	y	
904	clip movies		A	chris	NYS		22	y	
848	H-R diagram		A	helen	NYS		15	x	
509	night vision		A	shak	NYS		17	y	y
937	absolute motion		A		NYS		3	y	p
394	light pollution		A	shak	NYS		3	y	y
367	limit stars by distance		A	shak	NYS		15	y	y
A Totals:					1/27	68	334	78%	63%
735	comet database	303	B	chris, cal	NYS		15	y	y
640	milky way display		B	helen	NYS		10		
491	guides framework		B	shak	NYS		12	y	p
493	Galactic guides	491	B	ted	NYS		4	p	p
412	Equatorial guides	491	B	ted	NYS		4	p	p
458	Ecliptic guides	491	B	sam	NYS		6		
345	bookmarks to Web		B	tracy, bob	NYS		15	y	y
B Totals:					1/34	68	400	71%	56%

Document Version Control Information

version:	2.345	
last plan change:	Mon Nov 6 11:47 2000	DAP
last approval:	Wed Nov 8 13:21 2000	DAP
last update:	Fri Nov 10 18:34 2000	DAP

Appendix B Sample Stochastic Agile Horizon Plan

This second sample is a stochastic agile horizon plan that explicitly states the means and standard deviations for Normally distributed estimates.

As for the deterministic plan, Appendix C should be referred to regularly as these plans are read for an indication of the meaning of the various terms.

Planetaria 2.4 Agile Horizon Plan

Planned Release Dates

Release:	2.4	
Design Start:	Mon Sep 5	
Code Start:	**Mon Oct 30**	(38 workdays)
Development Cut:	**Fri Feb 16**	(**74 workdays**)
Beta Availability:	Fri Mar 23	(25 workdays)
General Availability:	Fri Jun 1	(49 workdays)

Mission

This release will be a significant upgrade from 2.3. The ability to view the sky from various locations in the universe, the ability to cope with orbiting entities, and the ability to simulate the passage of time are three of the major upgrades in this release. Many other features will be included as well. This release will satisfy many of users' most frequent requests, and satisfy a key NASA request for realistically rendered planets.

Current Status Summary

Estimates: are Normal with the given mean and standard deviation.

As at: EOD Fri, Nov 10 2000 (**Coding**: **10** of 74 days elapsed).

		mean		*sdev*	
Remaining coding capacity:		394	effective coder days	±	28
Average coders:		6.8	effective coders per day	±	0.5
A+B Features	Requirement:	400	effective coder days	±	16
	Delta:	-6	effective coder days	±	32
A Features Only	Requirement:	334	effective coder days	±	14
	Delta:	60	effective coder days	±	31

A+B Features					
Confidence:	**43%**	50%	80%	95%	99%
DCUT Slip (workdays):	on-time	-1	-5	-9	-12
Projected GA:	Jun 1	Jun 5	Jun 15	Jun 27	Jul 5
A Features Only					
Confidence:	**97%**	50%	80%	95%	99%
DCUT Slip (workdays):	on-time	+9	+5	+1	-2
Projected GA:	Jun 1	May 8	May 18	May 30	Jun 7

Capacity

Estimates: are Normal with the given mean and standard deviation.

As at: EOD Fri, Nov 10 2000
(**Coding** phase: **10** of 74 working days elapsed).

Coder	*Class*	*Days*	*Vacation*	*Work Factor*	*Effective Days*	*To Date*	*w(act.)*	*A*	*A+B*
Philip	Mgr.	64	5 ± 1	0.1 ± 0.1	5.9 ± 3	1	0.1	-0.1	-0.1
Tracy	Arch.	64	4	0.2 ± 0.1	12 ± 3.6	2	0.2	+10	+3
Sam	Lead	64	5 ± 1	0.4 ± 0.1	23.6 ± 5.9	5	0.5	+4.6	-1.4
Al	Lead	64	5	0.4 ± 0.1	23.6 ± 5.9	4	0.4	-1.4	-1.4
Gil	Lead	64	5	0.3 ± 0.1	17.7 ± 4.7	3	0.3	+0.7	+0.7
Chris	Coder	64	4	0.6 ± 0.1	36 ± 5.4	6	0.6	+5	-3
Shakur	Coder	64	0	1.2 ± 0	76.8 ± 0.6	12	1.2	+10.8	-1.2
Helen	Coder	64	5 ± 1	.6 ± 0.2	35.4 ± 11.8	7	0.7	+7.4	-2.6
Ted	Coder	50	4 ± 2	1 ± 0.4	46 ± 18.5	9	0.9	+19	+11
Cal	Coder	64	5 ± 1	0.6 ± 0.2	35.4 ± 11.8	5	0.5	+4.5	-1.6
Britney	Coder	60	3 ± 2	0.7 ± 0.1	39.9 ± 4.8	7	0.7	+3.9	+3.9
Bob	Coder	64	4 ± 1	0.7 ± 0.1	42 ± 4.9	7	0.7	+6	-2
totals:			49 ± 3.6	6.8 ± 0.5	394 ± 28	68	6.8	+71.3	+6.7
						Unassigned:		-11	-11
						Final Totals:		+60.3	-5.7

Requirement

Estimates: are Normal with the given mean and standard deviation.

As at: EOD Fri, Nov 10 2000
(**Coding** phase: **10** of 74 working days elapsed).

fid	*description*	*prereq*	*prio*	*assigned*	*status*	*to date*	*remain*			*spec*	*design*
345	show angular separation		A	al	DONE	4				y	y
304	Field-of-View indicators		A	helen	CC	7				y	y
234	NGC/IC objects		A	brit	WIP	7	18	±	1	y	y
389	time simulation		A	sam, cal	WIP	10	27	±	4	y	y
230	change location		A	helen	NYS		5	±	1		
704	set location to city	230	A	bob	WIP	7	3			y	y
298	set elevation	230	A	helen, cal	NYS		21	±	4	y	p
239	set location to planet	230	A	sam, brit	NYS		19	±	2	p	y
456	set location from map	230	A	bob	NYS		11	±	0.5	y	y
301	orbit framework		A	bob, al	NYS		32	±	5	y	y
303	orbit display	301	A	chris	WIP	6	9	±	2	p	y
906	orbit editor	301	A	shak	WIP	12	8	±	0.5	y	p
959	proper motion		A	phil, brit	WIP	1	14	±	1	y	y
508	selective constellations		A	tracy	WIP	2	2			y	y
102	images for objects		A	shak	NYS		10	±	0.5		

fid	*description*	*prereq*	*prio*	*assigned*	*status*	*to date*	*remain*			*spec*	*design*
294	show planets	102	A	ted	WIP	9	27	±	2	y	y
459	rendered planets	294	A	good	WIP	3	17	±	3	y	p
873	planet atmosphere	459	A	shak	NYS		8	±	4		y
939	custom images	102	A	shak	NYS		5	±	0.2	y	p
986	constellation boundaries		A		NYS		8	±	1	y	y
934	classical constellation		A	al	NYS		15	±	3	y	
904	clip movies		A	chris	NYS		22	±	10	y	
848	H-R diagram		A	helen	NYS		15	±	1	x	
509	night vision		A	shak	NYS		17	±	5	y	y
937	absolute motion		A		NYS		3	±	0.3	y	p
394	light pollution		A	shak	NYS		3	±	0.3	y	y
367	limit stars by distance		A	shak	NYS		15	±	1.5	y	y
A Totals:					1/27	68	334	±	14	78%	63%
735	comet database	303	B	chris, cal	NYS		15	±	1	y	y
640	milky way display		B	helen	NYS		10	±	2		
491	guides framework		B	shak	NYS		12	±	4	y	p
493	Galactic guides	491	B	ted	NYS		4	±	0.3	p	p
412	Equatorial guides	491	B	ted	NYS		4	±	0.3	p	p
458	Ecliptic guides	491	B	sam	NYS		6	±	2		
345	bookmarks to Web		B	tracy, bob	NYS		15	±	5	y	y
B Totals:					1/34	68	400	±	16	71%	56%

Document Version Control Information

version:	2.345	
last plan change:	Mon Nov 6 11:47 2000	DAP
last approval:	Wed Nov 8 13:21 2000	DAP
last update:	Fri Nov 10 18:34 2000	DAP

Appendix C Agile Horizon Plan Definitions

Agile Horizon Plan Definitions

The agile horizon plan is the company's current understanding of what features are going into the software by when, how many effective developers are deployed on it, and the current status of the development effort (ahead, behind, on-time).

Planned Dates

Key milestone dates for the plan.

release	The release designator for the software release being planned. The first digit is changed when a more substantial upgrade is being planned (usually accompanied by a larger marketing campaign - once every couple of years or so). A change in the second digit is used to denote a more standard feature release (every six to nine months or so).

design start	The date detailed specification and high-level design activity starts. Before this date, features have only been described to the point of a one paragraph description. Based on these initial descriptions, initial sizings were made and an initial plan was put together. At design start, each of the features is specified in detail, and preliminary software designs are proposed where needed.
code start	By this date, most of the features should have been fully specified and designed (if appropriate). The developers then begin coding and unit testing. The delta in brackets refers to the number of working days for the phase ending on this date (in this case, design).
development cut	Also referred to as "dcut". The date at which coding ends and system test and debug begins. We test if *development cut* is achieved by asking each developer if they know of any remaining code that needs to be written for the release to be feature complete. If the answers are all "no", then dcut is achieved. The delta in brackets refers to the number of working days for the phase ending on this date (in this case, coding).
beta availability	The date the release can be shipped as a beta. Prior to this date the release was being alpha tested. As of this date, the release is not perfect, but it is no longer embarrassing to let some others have a look with the caveat that it is not to be used in production. The delta in brackets refers to the number of working days for

	the phase ending on this date (in this case, alpha test).
general availability	The date the release can be made generally available to new and existing customers. The significance of this date is that prior to it, the previous release of the software was shipped to all new customers. After this date, the new release will be shipped to all new customers. The delta in brackets refers to the number of working days for the phase ending on this date (in this case, beta test).

Mission

A paragraph that describes the key goals for this release.

Current Status Summary

A summary of where the release stands in terms of its chances of being shipped to schedule.

We divide the features in the release into an essential "A-list" and a nice-to-have "B-list" (see priority of features). We then present the chances of being on-time for both the A-list and the A+B-list. Ideally, the chances of delivering the A-list on-time should be high, but the chances of delivering the B-list as well should be low, but still attainable.

estimates (stochastic)	All measures stated as *mean*±*sdev* in this section are assumed to be stochastic variables whose distribution is Normal with the given mean and standard deviation. The *mean* is a number whereby half the time we will expect the realized actual number to be over, and half the time under. The quantity *mean*+*sdev* is a number whereby we expect the realized actual number to be less than or equal to it 84% of the time. There are many other possible ways of giving estimates. For example, giving a 40% best case and 60% worst case. For less-sophisticated, easier to get going agile horizon planning, much of the benefit is available from using only a single number for an estimate. We recommend that those new to agile horizon planning start with this approach. In this case, there should be agreement when soliciting the estimates that they should all be comparable. For instance, that they should all be 80% worst case estimates (*i.e.*, the actual number is expected to be worse only 20% of the time). In this case, when the agile horizon plan balances capacity and requirement, it will then indicate there will be a

	better than 80% chance of hitting the dates.
estimates (deterministic)	When the plan is given in a deterministic mode, all estimates are assumed to be taken on a worst case basis, with the confidence level described in this field.
as at	The plan is up-to-date as-at the date given. Also indicated is the current phase we are in, the number of working days elapsed in that phase, and the total length of the phase in working days. A plan change would not modify this date. Rather, a plan status update would modify the date. A status update is performed by re-estimating the stochastic quantities given an advance in time. - To re-estimate feature requirement, we first determine the status of each feature (whether the feature is done, work-in-progress, or not-yet-started). For each work-in-progress feature we ask the assigned coders how much time they have spent to date on it, and a re-estimate of how much remaining effort they think it will take. - To re-estimate coder capacity we decrease the available days, and ask coders to re-estimate their vacation and work factor, possibly using information as to what their realized actual work factor has been to-date during the coding phase.
average coders	The average number of dedicated coders available

	per-day during the remainder of the coding phase. A "dedicated coder" is an idealized worker who spends 8 uninterrupted hours each and every working day working at nothing more than coding new features into this release. Flesh-and-blood coders are not expected to live up to this ideal. The units are in dedicated coder equivalents per day.
remaining coding capacity	The total number of dedicated coder days (see average coders) available during the remainder of the coding phase.
remaining coding requirement	The total number of dedicated coder days (see average coders) required for the remainder of the coding phase to implement all the features in the plan.
delta	Remaining coding capacity less remaining coding requirement. Negative delta indicates a danger of slipping the dcut date by this many effective coder days divided by the average number of effective coders available per day.
A+B features	Remaining coding requirement and delta for all the features listed in the plan, including both the more essential features (the A list) and the less essential features that it would not be overly painful to drop from the plan if the need arises (the B list) (see priority of features).
A features only	Remaining coding requirement and delta for just

	the essential features listed in the plan (the A list), and excluding the less essential features that it would not be overly painful to drop from the plan if the need arises (the B list). (see priority of features).
confidence	Various confidence levels. We would expect to slip dcut or the projected GA date by no more than the indicated amounts, with this level of confidence.
DCUT slip	Estimates with the given level of confidence for the worst-case number of working days expected to slip (-'ve numbers) or be ahead of (+'ve numbers) the planned dcut.
projected generally available	Estimates with the given level of confidence for the worst-case generally available date for the release. This is computed by assuming that any slip in dcut is augmented by a slip in the following phases in order to respect the inter-phase elapsed workday ratios.

Capacity

A breakdown of how the available coding resource is computed, along with a collection of time spent to-date by coders, and an indication of how balanced is the remaining workload.

coder	The name of the coding resource. Anybody on this list is capable of, and available to, do coding work on this release.
class	The type of the coder. Certain types of coding resource would be expected to have different ranges of work factors than others. This is (in some way) an explanation of the work factor.
days	Working days remaining during the coding phase to work on the release. Coders not available for particular days in the coding phase (with certainty) will have lower numbers. The maximum is the number of days left until dcut.
vacation	An estimate of the number of vacation days for this coder during the remainder of the coding phase.
work factor	An estimate of a multiplicative factor used to convert working days into dedicated coder day equivalents for the remainder of the coding phase. By definition, on average during the coding phase this coder is expected to put in 8 hours times this work factor of uninterrupted

	hours working on nothing but coding new features into this release. The work factor does not take into account any differences in productivity (this is accounted for in the feature estimates that take into account who will be assigned to each given feature).
effective days	An estimate of the total number of dedicated coder day equivalents this coder will work during the remainder of the coding phase.
to date	The (measured) actual number of dedicated coder day equivalents that this coder has put in so far during the coding phase. This "to date" and the "to date" on the requirement side are two different ways of slicing the same effort spent, similar to the two entries in a double-entry accounting balance sheet.
w (act.)	Based on the to date measure, the actual measured work factor for this coder so far during the coding phase.
A features	For the given coder, the difference between their available effective days and the number of effective days assigned to them in the plan for the remainder of the coding phase. Negative numbers indicate an over-commitment. This column is for the A priority features only.
A+B features	For the given coder, the difference between their available effective days and the number of effective days

	assigned to them in the plan for the remainder of the coding phase. Negative numbers indicate an over-commitment. This column is for both the A and B priority features.
totals	The sums of the columns. The ± numbers are the square root of the sums of the squares.
unassigned	The number of dedicated coder days for features that have not yet been assigned to any coder. Always negative.
final totals	The total over or under -commitment of dedicated coder days for all features in plan. These must tally with the delta measures in the current status summary section.

Requirement

A list of what features are planned to be in the release with estimates for the number of effective coding days required to complete them. Also includes additional information on the features, such as their priority and current status.

fid	A unique feature identifier used for cross-referencing back to this feature.
description	A brief description of the feature, intended to be an evocative mnemonic.

pre-req	Indicates a dependency amongst features for the purposes of agile horizon planning. If any of the listed pre-requisite features are removed from plan, then the features listing them must either be removed as well or have their scope changed significantly.
priority	A letter-grade assigned for the priority of a feature. Letter "A" is the highest priority. If possible, all higher priority features should be completed before any work is begun on lower-priority features. "A" features are assumed to be very painful to have to remove from the plan. "B" features are considered expendable in a crunch, but nice to have if they can be managed.
promised	A list of (external) customers who have been promised this feature and are counting on its delivery.
assigned	The coders that are assumed will work on this feature. The initial sizing estimate and remaining estimate is made assuming that this set of coders will be working on the feature. If the set of assigned coders changes, the sizing estimate should be re-considered. If this field is blank, no decision has yet been made and the sizing estimates assume the average coder's productivity.
initial	The initial estimate (as at the start of coding) of the total number of dedicated coder days it will take for the assigned coders to complete this feature during the entirety of the coding phase.

	The purpose of this column is to act as a historical baseline for comparing estimates against actuals. It should represent the best sizing available at the start of coding. If subsequent to the start of coding the scope of the feature is explicitly modified, this column may be adjusted accordingly to remain comparable. All other changes in sizing estimates after coding start (including assigned personnel changes and refined estimates) should be made into the remaining column (even if no work to-date has been done on the feature).
status	One of four possible status values for the feature. - DONE: The feature has been fully coded, unit tested, and reviewed. - CC: (Code Complete) The feature has been fully coded and unit tested. - WIP: Work-in-progress. - NYS: Not yet started. Generally features that are NYS are vulnerable for removal from the plan.
to date	The (measured) actual number of dedicated coder day equivalents that has been put into working on this feature from the start of the coding phase to the as-at date. This "to date" and the "to date" on the capacity side are two different ways of slicing the same effort spent to-date, similar to the two entries in a double-entry accounting balance sheet.
remaining	A re-estimate of the number of dedicated coder days it

	will take for the assigned coders to complete this feature during the remainder of the coding phase.
spec	Indicates if a specification for this feature is available, has been reviewed, and is approved. - A blank entry indicates that a specification document is not yet available. - An "x" indicates that the specification has been reviewed but rejected. - A "p" indicates the specification is available, but has not yet been reviewed (pending). - A "y" indicates either the specification has been reviewed and is acceptable, or that no specification is required and the reviewers agree. A specification should come in the form of a document that describes all of the (even potentially) user-visible aspects of the feature. When available, the document should be provided as a link from this field.
design	Indicates if a software design for this feature is available, has been reviewed, and is approved. - A blank entry indicates that a design document is not yet available. - An "x" indicates that the design has been reviewed but rejected. - A "p" indicates the design is available, but has not yet been reviewed (pending). - A "y" indicates either the design has been reviewed and

	is acceptable, or that no design is required and the reviewers agree. A design should come in the form of a document that describes how the feature will be implemented into the software, and that includes a task breakdown used to come up with a feature sizing. When available, the document should be provided as a link from this field.
test	Indicates how many test cases for this feature are available and have been successfully executed. The entry is in the form x/y/z, where z is the total number of test cases planned, y is the number that exist and have been run against the feature, and x is the number that have been successfully run.
docs	Indicates if end-user documentation for this feature is available, has been reviewed, and is approved. - A blank entry indicates that documentation is not yet available. - An "x" indicates that the documentation has been reviewed but rejected. - A "p" indicates the documentation is available, but has not yet been reviewed (pending). - A "y" indicates either the documentation has been reviewed and is acceptable, or that no documentation is required and the reviewers agree.
totals	- For status gives the number DONE relative to the total number of features.

	- For spec, design, and docs, gives the percentage of features with a "y" in this column. - For all others, gives a sum. The ± numbers are the square root of the sums of the squares.

Document Version Control Information

Gives the current version of the agile horizon plan, when the last change was made, and whether or not that change was approved.

version	The plan's version number (not the number of the software release). A given agile horizon plan will go through many different revisions before the software is eventually shipped.
last plan change	The date of the last change to the feature list or dates. There are two kinds of revisions to the plan: revisions that take place routinely to bring the status up-to-date (see as-at), and revisions that reflect changes to the plan (feature additions, deletions, and scope changes, and date changes). This date is the last time the latter (the plan) was changed.
last approval	If the last approval is after last plan change, the plan is valid. Otherwise the change is understood to be just a proposal.
last modified	The last time this document was modified in any manner.