A Systems Analysis of Complex Software Product Development Dynamics and Methods

by

Gregory B. Russell

Bachelor of Science, Computer Science and Bachelor of Arts, English
University of Tulsa, 1990

Submitted to the System Design and Management Program
In Partial Fulfillment of the Requirements for the Degree of

Master of Science in Engineering and Management

at the

Massachusetts Institute of Technology
September 2007

Signature of Author [illegible]
Gregory B. Russell
System Design and Management Program
September 2007

Certified By [illegible]
Professor Michael A. Cusumano
Thesis Supervisor
Sloan Distinguished Professor of Management

Accepted By [illegible]
Patrick Hale
Director
System Design and Management Fellows Program

Acknowledgements

I first wish to thank my parents, Robert and Sharon Russell, for their support and for teaching me the love of learning.

I wish to thank Professor Alan MacCormack of the Harvard Business School, Chris Kemerer of the Katz Graduate School of Business, Michael Cusumano of the Sloan School of Business, and Bill Crandall whose paper on the trade-offs between productivity and quality in selecting software development practices has served as the key reference for this thesis due to its combination of keen insight and objectivity that is so rare in software development.

At the Massachusetts Institute of Technology, Professor Michael Cusumano who advised this thesis, and whose books, teaching, and research have been a rare beacon of reason and insightful analysis throughout my career in software development and my time at MIT.

I am grateful to my fellow graduate students from whom I have learned much and who have made my time at MIT well worth taking an 18 month vacation from work.

Thanks to Pat Hale, Helen Trimble, Ted Hoppe, and William Foley for all of their help and patience along the way.

Most of all, thank to Erin without whose help, support, and editing skills, none of this would have been possible.

A Systems Analysis of Complex Software Product Development Dynamics and Methods

by

Gregory B. Russell

Submitted to the System Design and Management Program on
August 10, 2007 in Partial Fulfillment of the Requirements for the
Degree of Master of Science in Engineering and Management
at the

Massachusetts Institute of Technology

Abstract

Software development projects and products have long shouldered a reputation for missed deadlines, blown budgets, and low quality. Unfortunately, this negative reputation appears to be supported by more than just anecdotal evidence; quoting an industry study[i], respected software development expert and author Steve McConnell reports in his book *Professional Software Development*[ii] that "Roughly 25 percent of all projects fail outright, and the typical project is 100 percent over budget at the point it's cancelled." What's more, notes McConnell, "Fifty percent of projects are delivered late, over-budget, or with less functionality than desired."

Exactly why software development projects and products have historically performed so poorly and with arguably little if any improvement over the past 40 years, however, is a subject on which there is less agreement. While blame often aligns along functional (product marketing and sales) versus technical (software development) lines, the increasing popularity of different and often contradictory software development methodologies seems to suggest that no real consensus exists within the software development community itself.

The goal of this thesis is twofold:

1. To describe a set of key factors to consider when analyzing software processes
2. To outline an organizational framework that is optimized for implementing and managing software development practices

Table of Contents

List of Figures

Chapter 1: Introduction

"Businesses in the US spend about $250 billion annually on software development with the average cost ranging from $430,000 to $2,300,000[iii]Error! Bookmark not defined.."

"Roughly 25% percent of all projects fail outright[iv]."

"Another 31% are cancelled, primarily because of quality problems, creating losses of about $81 billion annually[v]."

"Only 16% of these projects are completed on schedule and within budget.[vi]"

1.1 The Slow March Forward

Countless software industry watchers can and frequently do quote such now-ubiquitous statistics, even though the connection of such insight to any real understanding of how these numbers were generated is debatable at best. The more salient point may be that few people, least of all those in the software industry, thought to even question these clearly damning numbers, much less research their origins. In light of the greater industry's seismic shifts (as represented by the advent of the World Wide Web, Agile Methodologies, and Java to name a few) since initial publication of these oft-referenced studies some 15 years ago, it's reasonable to wonder if the field of software development has experienced improvement proportionate to that of the overall technology sector.

Dr. Winston W. Royce[vii]'s groundbreaking 1970 article "Managing the Development of Large Software Systems" is often cited as the first salvo in the age of formal software processes, a subject about which more books seem to have been published in the past 10 years than in the previous 30 combined. Interestingly, Royce never used the term "waterfall" although he did make use of the famous "descending step" flowchart, in reference to which the process is commonly believed to have been named. In fact, Royce's article offered an early hint at the potential for something resembling iterations in the case of complex projects.

With or without a name, the waterfall process proved to be very easy sell, especially given how nicely it fit into the popular "assembly-line" model of product manufacturing where everything could be broken down into a set of ordered, discreet and homogeneous steps. In view of what had been in place prior to its introduction, the waterfall model did likely represent a big improvement. In his book on the history of the software industry, Martin Campbell-Kelly describes managing software projects in the pre-waterfall era as "...something of a back art. There were no formal methodologies until the late 1960s, and then they tended to little more than prescriptive checklists[viii]."

In the 37 years since Royce's article was first presented at the Western Electronic Show and Convention, software development and its supporting processes and practices have undergone numerous radical changes. And today, with the arrival of the Agile models and the competing CMMI for Development V1.2, both camps seem to ready to lay claim as Brooks' "Silver Bullet[ix]."

1.2 Comparing and Contrasting Processes

As the methodology debates began to heat up in the mid-to-late 1990's, I found myself drawn repeatedly to two core questions:

1. How could the leading contenders be so at odds with each other?
2. Why was the software community unable to reach an agreement as to best development methodology?

As my career led to direct interaction with a variety of different software projects and different people, however, I began to respect the distinctively complex and diverse nature of the software development discipline and to appreciate the challenge of creating any kind of a controlled test. Not only is each software development project unique, but in my experience, the talent and motivation of the developers and other key stakeholders has more impact on success than process.

1.3 Factoring the Human Factor

Having said that, however, I've also found that process can do much to help or hurt a development effort. Given the stated importance of the people on the project to its success, it follows that the quality of the collaboration and communication between those individuals is also critically important. Other than the culture at a company, processes and practices probably have the biggest impact on the how well development and cross-functional teams collaborate with each other and on the quality of the communication throughout the larger product organization.

In other words, while a truly controlled study or comparison of different software methodologies would be impossible largely impossible due to the inability to control all variables, the benefits of such a study would be significant enough to render an attempt at such an effort worthwhile. Unfortunately, the quality of the most of the readily available development methodology research is disappointing and rarely even worth the effort required to glean any usable kernels of information.

Indeed, many existing software development process studies are done by the advocates of a particular methodology and may include only a few small projects as sample data. Other studies are published by companies or individuals with a direct financial stake in the success of a particular set of practices such as Rational/IBM and the Agile community led by Kent Beck.

1.4 Process Gone Personal

So far in my software career, I have enjoyed the opportunity to experiment with many of the different practices from the leading methodologies, and have been able to do so at a range of different software organizations both large and small. This has allowed me to benefit from old-fashioned trial and error. Over time I've collected a number of what I categorize as best practices that essentially draw from all of the major methodologies and applies sets of them based on the specific characteristics of the software project or product itself, and I had initially planned for this thesis to focus on these practices.

I realized, however, as I continued to stay current in my reading and to do research for this thesis that there was no shortage of similar publications. In fact, it seems that the idea of making informed choices and creating a selective "best of" set of software practices is becoming evermore widespread, to the point some might say of commoditizing the very concept.

Ivar Jacobson, widely considered the "father" of the Rational Unified Process, is one industry thought leader who has publicly commented on the growing overabundance of process material. Recently *Dr. Dobbs' Journal* published a series of three articles co-authored by Jacobson in which his sentiments were none too subtly conveyed by the series' title, "Enough of Processes: Let's Do Practices[x]." Too, several books have also been published within the last couple of years that also suggest this proliferation of methodologies be regarded as toolsets rather than gospels.

In the end I decided that simply adding my own opinions and experiences- however valid- to the rising din of promising but still largely unstudied and under-researched publications and blogs was unlikely to be of much value to anyone. As I contemplated how to most effectively package and contribute my personal experience, I encountered the 2003 study focused on productivity and quality around eight development practices. Authored by a team of leading academic and corporate researchers, "Trade-offs between Productivity and Quality in Selecting Software Development Processes[xi]" accomplished several critical objectives that I believe no previous study has achieved, including:

- Authored by evaluators with no ties to a specific methodology or set of practices
- Conducted in as controlled a setting as possible by:
    - Selecting out marginal or outlying projects
    - Limiting cultural and/or other variations by containing the study to a single organization
    - Choosing an environment large enough to "average out" many miscellaneous factors
- Correlated practices from different methodologies both individually and in multivariate arrangements against a set of the two most critical performance variables

Providing objective data comparing a number of multiple cross-methodology practices should be seen as a watershed accomplishment and should also be required reading for anyone involved in software project or product development. Due in large part to the import and impact of the HP study to the area of software development in which I have spent the majority of my career, I restructured my thesis to position my original contribution as a complement to the "Trade-offs between Productivity and Quality in Selecting Software Development Processes" work.

To that end, this thesis aims to accomplish two primary goals:

1. Provide descriptions and analyses of two additional critical factors that I believe can add relevant and applicable insight to the study's correlations in terms of how different sets of practices would perform given these factors

2. To describe an organization structure or framework that I believe can be helpful in enabling software development managers and leaders to more effectively choose and more easily integrate new sets of software development "best practices" into their organizations

Chapter 2: Key Software Product Development Factors

2.1 Key Factors Overview

This section describes two key factors involved in software development that are important in understanding what practices will be most effective in on a particular software development project. In selecting and analyzing these factors I have chosen to focus on a recent study done with data collected from 29 software projects at HP over approximately nine months [xi].

As discussed briefly in the introduction, there are many reasons why this study is virtually unique in its contribution to the practice of software development today and the many competing and seemingly incompatible methodologies. These include, but are not limited to the following:

- Inclusion of the key representative practices that are most important to the most popular software development methodologies (Agile, CMM, RUP, etc.)
- Limiting of extraneous influences and factors allowing for a relatively controlled environment, especially given the high level of complexity and variability presented by the study of software development projects
- Inclusion of a relatively high number of software development projects allowing for statistically meaningful analysis
- An empirical and objective approach that is extremely rare for the subject

As a veteran of the software development industry and as someone who has been directly involved in multiple process improvement efforts, it is my sincere hope that others will build upon these results in the future. In an effort to provide some potential help to such future studies, I have attempted in this section to provide additional input into the interpretation and analysis of a few of the key results. Where applicable, I also make specific suggestions as to how future studies could potentially add to the value of this study by including additional factors in the data collection and the subsequent correlations.

The discussion of each factor includes a description of the factor in terms of its individual characteristics in relation to software development as well as an analysis of the effects of the single factor on both product quality and productivity. I have also provided wherever possible suggestions or recommendations regarding potential "best practices". In cases where interactions between multiple factors can result in a collective dynamic that is different from that presented by any of the single factors, a further discussion of the appropriate practices may also be included.

2.2 Factor One: Differentiating Between Different Types of Software Development

In the HP study, three different types of software products or projects were included in the testing results. The correlations, however, were not presented separately for each type of software project. In my experience, the dynamics can vary significantly based on the type of software or project or product, and while limiting the number of different correlations helped to simplify the results and to focus the discussion on the different practices which were understandably the focus of the study, I suspect that some additional patterns would have emerged or existing ones would have been made more discernable if the single variable and multivariate correlations had been run against the two performance measure separately for each of the three different types of software development projects. In other words, could there be valuable additional information to be gleaned by having looked at the application, systems, and embedded software systems separately to determine if any additional subtleties may have emerged that are specific to each discipline? Based on my experience and analysis of the results, I suspect this may have been the case in at least a couple of instances.

The first example is based on the fact that, on average, it can often be more difficult to have customers review prototypes earlier on systems software projects as compared to applications software projects (since I'm not familiar enough with embedded software to make an intelligent observation, I'm limiting my discussion to application and systems software). Application software typically has a higher degree of direct user interaction usually in the form of a graphical interface. Systems software, on the other hand, would typically consist of more complex algorithms and a greater number of interactions with different systems, both of which are often hidden from the user as opposed to application software's simpler and more discrete business processes. Given these same attributes, systems software may also be difficult to test early on as incomplete modules may not be testable separately prior to the "code completion" of additional software modules or components due to the higher likelihood of technical and functional dependencies.

If either of these characterizations of the differences between application and systems software development is accurate, then a scenario could result in which the composite data and the resulting analyses may actually be skewed, at least in part, based on specific characteristics unique to different types of software development rather than only on how effective or ineffective a particular practice or combination of practices is in general.

This could be the case if either of the performance measurements varies due to the nature of the type of software rather than something essential to the practice itself, or if certain practices are not compatible with one or more types of software development. In both cases, one of the performance measures may be correlated positively or negatively in the composite in a way that isn't true for one of more of the different types of software development.

2.2.1 Impact on Performance Measures (Software Quality and Productivity)

To better illustrate this scenario, I'll include an example from my own experience. In order to test software that has dependencies on other software that isn't ready for integration testing, developers will often take the time to build what are commonly called stub functions. In the case of a systems software project, building stub functions can be time consuming due to their complexity or just the number of them required to do an accurate test.

On average, however, building stub functions in order to test earlier, whether using a prototype or not, will almost always have a positive effect on quality. Whether or not the costs or hit to productivity is worth the improvement in quality is a difficult call to make in advance as this would depend on a number of other factors including how much productivity would be gained by having fewer bugs later that need to be fixed, which brings me back to my point. Assuming that early testing/prototyping was also beneficial to software quality for the other types of software development and there was a consistent net benefit to productivity. When included in the overall results are calculated, they may reflect accurately that, on average, early testing or prototyping was both a benefit to software quality and a positive for productivity. For the systems software manager, however, these results could be misleading.

In addition to the above example, the same effect could happen in reverse. In this case, practices that are uniquely beneficial on balance to a specific type of software development may be buried in the data and overshadowed by the lack of measured performance benefit to other types of software development. This same logic also applies to multivariate correlations to the performance measures and could occur when either one set of practices is not practical or possible on a given development effort or in the case where different sets of practices are mutually exclusive or counterproductive in combination. In essence, when certain "best practices" are either impractical, or costly in other ways for a particular type of software development, then more subtle improvements may be helpful if they can be discerned.

This dynamic may help to explain a couple of the variances seen in the HP study where individual practices demonstrated different correlations to the target performance measures based on whether they were observed alone or as part of a set of specific practices. I'm specifically thinking of the following example from the study:

> "Some practices that are correlated with performance when considered in isolation do not appear as significant in multivariate models. For example, we

> initially found that having a less complete functional specification when coding begins is associated with lower productivity, lending support to the waterfall model of development. When we accounted for the variance explained by releasing early prototypes and using daily builds however, however, this relationship disappeared. (A similar pattern exists for the relationship between a more complete detailed design specification and a lower defect rate.) In a sense, these other practices appear to 'make up' for the potential trade-offs arising from an incomplete specification."

Again, it would be interesting to see how these results compared across the different types of software development as incomplete documentation may not, on average, be compensated for by daily builds and early prototyping on systems software projects where the effectiveness or "costs" of these methods may differ from other types of development. The importance of this distinction is similar to the previous example in that without this additional analysis, a systems development manager may choose to reduce the amount of time spent creating documentation in order to invest more time on practices that might not be favorable for that specific project.

Making a similar point, the authors summarize these observations by stating that:

> "This suggests that development models should be considered as coherent systems of practices, some of which are required to overcome the potential trade-offs arising from the use (or absence) of others."

I agree strongly that understanding how different practices interact in either favorable or unfavorable ways is extremely important. I would add, however, that these multivariate interactions may vary based on the type of software of software being developed or on other factors that may not have been part of the research. Further, the lines of coherent interaction may not always be drawn based on a specific model. For many software companies or project managers, there may be organizational constraints or other factors that make one or more of a particular model's practices impractical or otherwise detrimental to overall performance. In these cases, it may be important to the project's success, to be able to combine practices from different models in ways that result in better performance that could be had by adhering to a single model

2.3 Factor Two: Maturity of the Software Product

For software projects or products of the same type, the maturity of the product is probably the single most important factor in determining what practices are most likely to be effective. Books on software methodology, however, rarely focus much time on how the constraints and dynamics differ between new development and maintenance. This is especially true of the Agile methodologies which tend to focus almost exclusively on new development in one form or another One could draw the conclusion from this that many of the Agile authors are more likely to have a background in software consulting rather than in the commercial software.

For the purposes of this discussion, I am using maturity as an average measure of the amount of development on a software project or product that would typically be considered maintenance versus new development. While there has been some very impressive and in-depth analysis of what qualifies as new development versus maintenance development[xii][xiii] [xiv](e.g., [Boehm, 1981], page 54, [Lientz and Swanson, 1980], and [Fowler et al., 1999]), for the purposes of this discussion, the simple metric that I've used for quantifying this factor is the percentage of new lines of code (LOC) written as part of the new effort versus the number of existing LOC.

Using this approach, software is "immature" when the percentage of new lines of code being developed is 50% or more of the total lines of code at the end of the project (e.g., more than 10,000 lines of new code on a product that previously consisted of 10,000 lines or less). If the percentage of new LOC is between 25% and 50%, this product is considered to be "moderately mature" with some additional legacy-related constraints as compared to development efforts where the percentage is below 50%.

Contrastingly, when the percentage of new LOC in development is below 25% this could be classified as a mature product. Taking the data from the HP study for example, the average maturity of each project was 41%, meaning that the maturity level for each of the software project's would be moderate. Extrapolating out the number of previous development cycles based on the data, I would estimate that average project was on its third major release cycle.

For many existing or well-established software companies, most of its developments efforts likely fall into the "moderately" or "fully" mature classifications. Software companies like this are likely focused on either maintaining or extending an existing software product or set of products and are therefore bound by the often unique constraints and dynamics that relates to this type of development.

Ironically, despite the high percentage of software product development taking place at any given time that falls into the category of “fully mature” with all of the related constraints, many of the existing methodologies give little if any consideration as to how the dynamics and practices of maintaining and enhancing an existing product may differ from creating a new software product from scratch. As mentioned earlier, this is especially true of the newer Agile group of practices and may partially explain why many larger and established software product companies have been slow to adopt Agile practices either in part or in whole. The differences, however, are considerable as are the implications on how each part of the development cycle, from requirements through release, should be approached.

2.3.1 Impact on performance measures (software quality and productivity)

In many ways, the maturity of software differs from other factors in that it’s not necessarily clear what specific definition of “software quality” and “productivity” is most appropriate. This makes direct comparisons across the different types of development problematic. For example, in terms of software quality, development efforts on fully mature products focus partly if not wholly on fixing or resolving existing bugs, missed requirements, and design flaws. While this doesn’t completely immunize this type of software from of new bugs as part of a bug fixing cycle, my experience had been that fewer new bugs and design flaws are introduced per lines of code than with new development.

This is likely due in part to the more "targeted" nature of fixing known bugs as well as the benefit provided by developing against existing software whose nature and behavioral flaws have had a chance to show themselves over time and in a number of different working environments rather than in the more theoretical and often limited environment that is used during pre-release unit and system testing. In fact, the specific code that needs to be fixed is often known and documented long before the fix actually applied. This dynamic can positively affect the measurement of software quality by serving to decrease the average number of bugs introduced per LOC. One way to adjust for this difference when comparing software quality measures of new versus mature development is to apply a calculated constant based on historical data (ie software equivalent of Planck's constant). Another option is to simply avoid making an "apples to oranges" comparison and to rate the measurement of "software quality" for bug fixing efforts only against other bug fixing efforts. This would require each software project to be classified based on its level of maturity or similar characteristic. While this factor, unlike the type of software development, was not included in the correlation data, it does appear to have been captured for each project.

As opposed to the higher measure of software quality associated with an increase in project size or maturity, one might conclude that productivity would follow a different curve in which productivity would most likely be lower as compared to new software development. Based solely on the emphasis placed on new development by the Agile methodologies, for example, it would seem logical to conclude that in the same way software quality benefits from working within the constraints imposed by the existing body of code that the productivity would be likewise constrained or reduced. This might seem especially true when also considering the amount of time that might be spent searching and parsing through the existing code's logical structure before making any changes or extending the software by writing new lines of code. And while both of the previously assumed dynamics likely have a legitimate and negative impact on productivity, the reality is that the benefits to productivity provided by the additional stability, familiar functionality, and amount re-useable code afforded by mature software products far outweigh the concerns on average.

Newer development efforts, for example, are more likely to have to deal with significant design issues that may require a large amount of rework. And attempts to avoid the need for future rework will likely mean multiple prototypes or the creation of entirely new test scenarios all of which, while helpful in improving product quality, take time away from the number of lines of code that can be written during the same time period.

Additionally, if the same developers are still working on the project, they will likely be very comfortable with both the technology and the functionality of the software, requiring a much smaller learning curve and greatly reducing the likelihood that requirements will be misinterpreted or improperly implemented. Pre-existing test scripts can also speed development as it is much easier to simply add to or modify a test script than it is to create one from scratch.

There are, however, some additional exceptions to the general rule which warrant attention and would also be candidates for consideration as variables or factors in later studies. For example, depending on the length of the release cycle, mature products may require extra time to be spent regression testing the entire code-base even if only a small segment of the code has changed in order to ensure that newer and unanticipated bugs have not been introduced. Probably the largest exception to the rule is when the subsequent development or maintenance of a mature software product is done by a different person or team than originally developed the software, as this would cause many of the benefits mentioned above to either be reduced or to disappear altogether.

I would like to have been able to find the data needed to calculate what percentage of all software development would classify as new versus maintenance. Even without this data, however, it's clear that the majority of software products are older than the two years or two cycles it would take qualify as a mature product. Given this and the degree to which the dynamics between the two types of development can differ, having solid and objective data as to which practices work well for both types of development and which practices are only effective in a particular situation would extremely valuable to software industry. Even companies like Microsoft that to large degree built their success on having some of the most original and effective practices in the industry are struggling with the massive constraints placed on them by the sheer volume of legacy code along with other issues. While Microsoft may be an extreme example, the value of knowing how different practices perform based on the maturity of a software product and how even the performance measures themselves might differ would be helpful to a significant percentage of the software industry.

Chapter 3: A Value-Based Organizational Framework

3.1 Reframing an Old Picture

The idea for creating a new type of organizational framework originated during my experience at a software startup company where I was tasked with leading a software development organization while also building out a new product management group within the organization. This came at a time when I had just spent several years finding an organization template that worked for the software development group which went through several iterations before finally finding one that worked well for our company. I quickly realized that creating a structure for the product management organization required thinking about the structures and processes of the company as a whole since this new group was to be responsible for translating the company's strategic objectives into an executable software development plan.

Unfortunately, since none of the off-the-shelf models worked well, the development group had already headed in a more agile process direction, whereas the sales and marketing groups were a steadfast hierarchical bunch that didn't like the idea of spending any more time talking to developers than was absolutely necessary. Through some careful picking and choosing of different practices from various schools of thought and a lot of hand-holding, the company moved forward more or less successfully.

Soon thereafter, I found myself in charge of creating new development processes at a very large enterprise software company, where I also quickly discovered that there was much less room to officially redefine roles and structures than there had been in the previous startup environment. Working very closely with other core stakeholders, I was again able to achieve some degree of success but these two experiences together provided the impetus for creation a new type of software product development organizational template for that had the following two primary goals:

1. To provide a structure which would enable software organizations to create a collaborative environment at all levels of the company, tackle software development's unique complexities and rapidly changing requirements by focusing decision making in the areas where the appropriate skill and knowledge resided regardless of seniority or status, and provide the conduits for the needed two-way flow of information between strategic and tactical groups

2. To make the structure flexible enough that companies of different types and sizes could adopt it quickly without a major re-organization or a large scale change in role definition and hierarchy.

Most of the ideas described in this section are based on real world experience and have been implemented to one degree or another at various types of software companies both large and small and whose focus ranged from enterprise software to web application development. In fact, I was recently able to put my thesis subject to good use when I implemented a more specific version of the framework at a software company with approximately 60 employees. The overview that was used during this implementation can be seen in Appendix A.

3.2 Basing "Value-based" on Values

The selection of the term "value-based" as a way to describe the collective ideas in this chapter was designed to focus the idea of structure on what it should be doing: helping an organization achieve its objectives rather than simply providing a way to put all of the jobs and reporting relationships on a single chart. In my experience, organizational structures tend to create an undesirable resistance to change, as employees become comfortable and even protective of the status quo where the organizational structure becomes something you have to "work around" in order to get anything done. In contrast, I wanted it to be clear that the goal of the "value-based" structure was, to put it simply, to add or enable the creation value as defined by the company.

3.3 Analysis of Existing Organizational Structures

3.3.1 The Hierarchical Organization

Hierarchical organizational structures are still the norm in the software industry as in most manufacturing industries. As companies like Google and Amazon continue to grow, making their development models likely to be copied elsewhere , organizational structures at other software companies are generally on track to get flatter and consist of smaller and more independent teams. In all likelihood, however, most organizational structures will continue to be some variation on the hierarchy theme with variations in levels and role definitions. A typical hierarchical structure for a software company can be seen below in Figure 1.

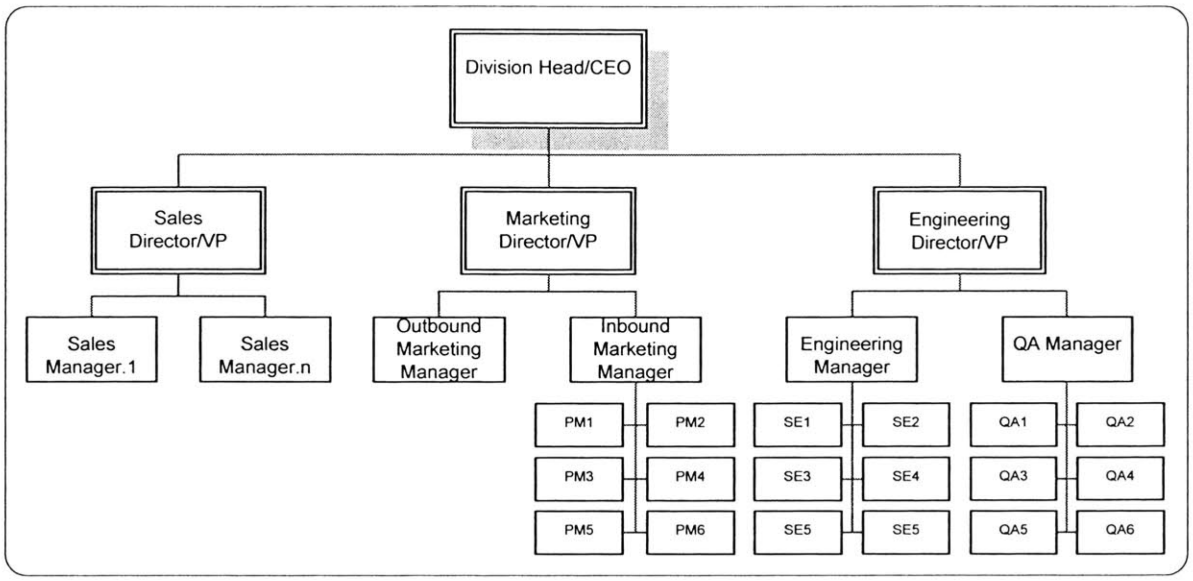


Figure 1. A typical hierarchical organizational structure for a Software Product Development Company

In my experience, there have been two primary weaknesses in software companies that adhere to strict hierarchical organizational structures:

- Decision-making tends to be concentrated at the highest possible level rather than at the most appropriate level where the decision-maker would have the most experience and information
- There is no structural path for cross-functional coordination and communication which is critical in software development where the work is especially complex and tasks can rarely be broken down into discrete responsibilities

This is especially true of any decisions that have high visibility or may impact the "bottom line" to one degree or another. While this may work in certain types of organizations, the complexity of issues within a software development organization are such that decisions need to be made at whatever level and by whatever group or person has the best knowledge or experience relating to that issue.

Others have argued that this dynamic requires that upper management step in more frequently in order to decisions that better adhere to a company's bottom line. In software development, issues are often technical or architectural in nature and arguably best answered by the responsible senior developer or software architect. I refer to these types of issues as "tactical" issues which deal specifically with "how" a strategic objective can best be accomplished or executed, whereas "strategic" issues focus on "what" needs to be accomplished and may be best suited for upper management.

In a hierarchical organization, when tactical decisions of consequence are made by the highest level manager or executive with awareness of the issue despite potentially being many levels removed from those that have a detailed knowledge of the issue, decisions tend to be made based on factors that may have nothing to do with the actual merits of the issue, but are rather based on out-of-date past experience or the advice of the most influential person or group. This creates a pattern whereby tactical decisions are made based solely on the higher-level strategic objectives.

In turn, this dynamic often has the unintended consequence of lowering morale among developers and others who may be painfully aware of how much or how little understanding the decision maker had of the issue. Over time, this can also build a strong mutual distrust among the different hierarchical levels as each feels unappreciated and undervalued by the other.

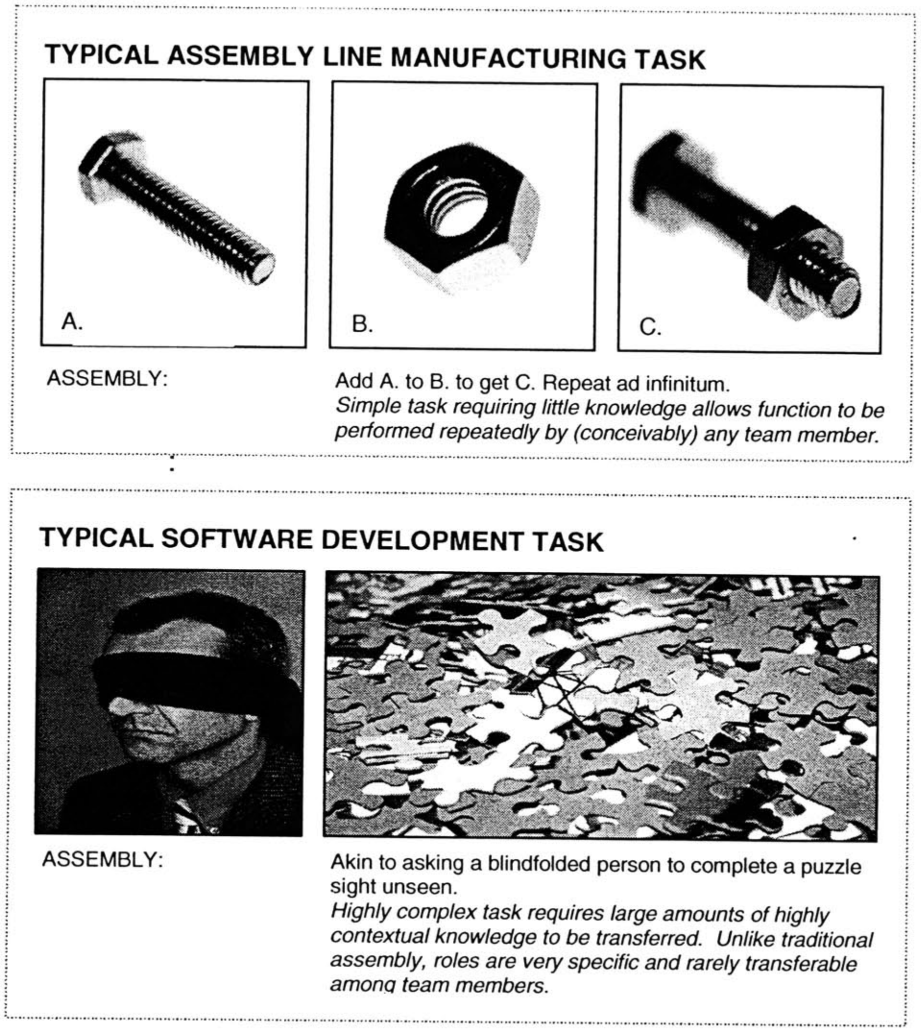
This is not to say that hierarchical structures are bad per se, or that there aren't software companies which have been successful with this model. On the contrary, it's reasonable to assume that most software companies of the past few decades have been hierarchical in nature, but I also suspect that the more successful companies have developed the ability to act outside of the hierarchy when necessary to optimize the quality of decision-making.

One way to enable this dynamic is to create a conduit or steady flow of critical strategic and tactical information through hierarchical boundaries and across levels which often increases trust and provides for better alignment of strategic and tactical decisions. One of the most notable and successful examples is the role of the "program manager" as defined and practiced by Microsoft and described in *Microsoft Secrets —How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People.*[xv]

In essence, the purpose of this role was to align strategic and tactical objectives where strategic objectives are represented by product requirements and tactical objectives are represented by the actual software development designs and project plans. The role and the practice of program management is notable for both the degree of success that Microsoft achieved with it and that it was largely the first time this type of role was defined and formalized within a software product development company of Microsoft's stature. In essence, the purpose of this role was to align strategic and tactical objectives where strategic objectives are represented by product requirements and tactical objectives are represented by the actual software development designs and project plans.

In regard to deficient structural support and communication, most software companies find ways around this limitation out of necessity. A more discrete or "assembly-line" approach to product development may be appropriate for certain product manufacturing, but for software developers, this would be analogous to assembling a complicated puzzle blindfolded (see **Figure 2.**)

There is simply too much functional and contextual information embedded in software requirements to develop software that meets those requirements. In fact, software is one of the few "product" industries where the product is actually "manufactured" virtually and entirely by people. Fred Brooks wrote very eloquently on this topic in his "No Silver Bullet" article when he stated:



TYPICAL ASSEMBLY LINE MANUFACTURING TASK

ASSEMBLY: Add A. to B. to get C. Repeat ad infinitum.
Simple task requiring little knowledge allows function to be performed repeatedly by (conceivably) any team member.

TYPICAL SOFTWARE DEVELOPMENT TASK

ASSEMBLY: Akin to asking a blindfolded person to complete a puzzle sight unseen.
Highly complex task requires large amounts of highly contextual knowledge to be transferred. Unlike traditional assembly, roles are very specific and rarely transferable among team members.

Figure 2. Manufacturing v. Software Task Comparison

3.3.2 The Matrix Organization

It's probably not accurate to say that very few companies have been able to successfully implement a "pure" matrix based organizational structure organization as there's ample evidence to suggest many companies have some degree of cross-function team participation and/or consider themselves matrix organizations. One could even argue that the "value-based" structure described in this thesis is just another way of viewing a matrix organization given the strong emphasis on cross-function teams and communication.

Some facet of the matrix based approach, however, must have presented difficulties for many companies during a period of backlash and abandonment against "matrix" structures amid much early hope. A search of scholarly articles on matrix-based organizational structures turns up articles almost entirely in the 1970's and 1980's. And having been responsible at one point for introducing a matrix-based organizational structure into a software development group, I can identify firsthand with the challenges and weaknesses commonly associated with such an approach. For an example of what a matrix organization might look like as applied to a software company, see **Figure 3.**

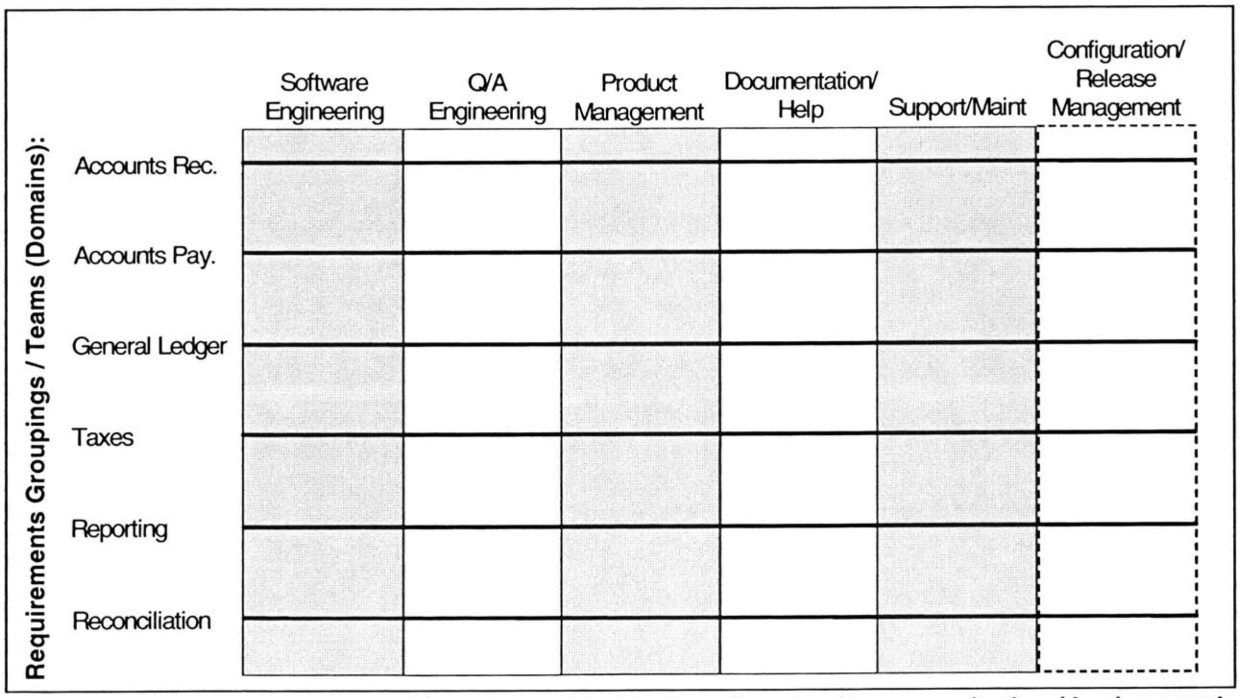


Figure 3. A matrix-like view of a value driven organizational implementation

Like many other people, I thought at the time I introduced it that a matrix structure seemed to be a logical if not obvious solution to combining different existing functions and organizations into a single cross-functional or cross-team structure. In the end, I think the failure of the "pure" matrix-based approach is not its "cross-functional" nature per se, but rather the combining of two essentially flawed structures.

Strictly matrix-based failures must, however, be differentiated from failures connected to cross-functional approaches in general because the matrix based-approach simply overlaid two existing and ultimately incompatible structures without re-thinking or fundamentally re-examining the nature of the structural relationships. Without stepping back and actually rethinking the larger question of what was needed to best deliver a product to the market within a specific organization, one can clearly discern a "two wrongs don't make a right" scenario.

The "value-based" framework was designed to avoid this scenario by emphasizing the creation of a cross-functional approach or set of key principles related to software product development that both recognizes the underlying dynamics of the work and seeks to best enable the cross-functional relationships necessary to create the value and meet the company's strategic goals.

3.4 A Value-based Organizational Framework

When designing this framework I actually started by defining the principles before thinking about the actual structure. In this same spirit the discussion is organized in the same way.

3.4.1 Essential Principles and Dynamics

Principle #1 – Strategic and tactical decoupling

The structural and procedural decoupling of the strategic and tactical functions within an organization

This allows the tactical groups or functions within an organization to follow a "bottom up" approach to optimizing processes and decision-making as opposed to the hierarchical or top-down approach favored by the strategic (executive) layer. A "bottom up" approach means that the focus of the tactical organizational is the work of the individuals or teams at the "bottom" of a standard organizational chart. This should also enable the strategic layer to maintain focus on the issues of strategic direction and the development of higher-level objectives.

Principle #2– Creation of a coordination layer

Creation of a cross-functional coordination layer within the larger organizational structure that ensures and enables ongoing alignment between the strategic and tactical layers of the organization

This approach allows strategic and tactical groups to optimize their own processes without interfering with each other as the needed information flow among those two layers is managed by the middle coordination layer (i.e."value-added" middle management.) Examples of this might include the following:

- Managing the flow of requirements and priorities from the strategic to the tactical layer of the organization
- Managing the reporting of "real time" status updates and changes from the tactical to the strategic layer

Sharing a coordination layer within the organization also means that this layer can focus on more proactive risk management and "push" issues in need of resolution to the strategic or tactical layers on an "as needed" or real-time basis rather than at the end or beginning of the week during a two-hour status meeting and after the issue has already festered and incurred costs for two-to-five additional days.

This layer also provides a clarified role for the "technical management" position which was lost or virtually moot within the standard matrix approach, but now emerges as a technology advocate while staying geared to tactical execution. The example below shows the three separate layers at their most basic form:

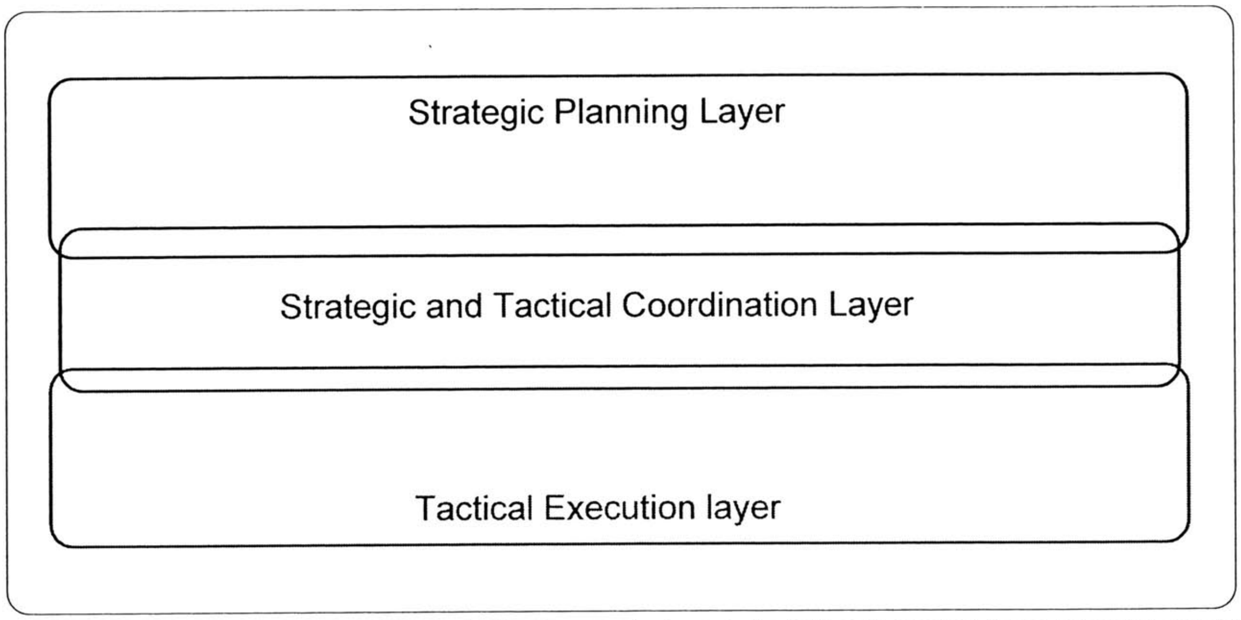


Figure 4. The three meta-layers of a Software Product Development Company.

Figure 5 illustrates the simplest way to map the value-based framework to a typical hierarchical software organization.

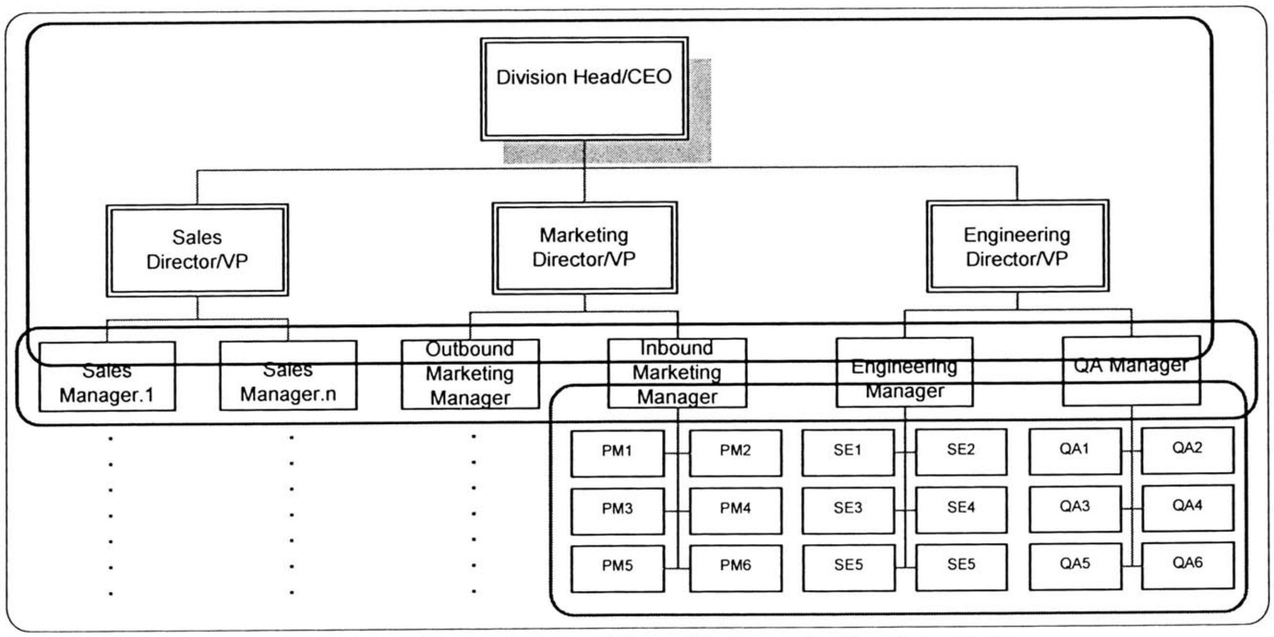


Figure 5. The three meta-layers overlaid on a standard hierarchical organizational structure

In reality, the effectiveness of the value-based approach is dependant on the effectiveness of the cross-functional team. Figure 6 shows a more advanced implementation in terms of how a software organization could most effectively utilize the framework.

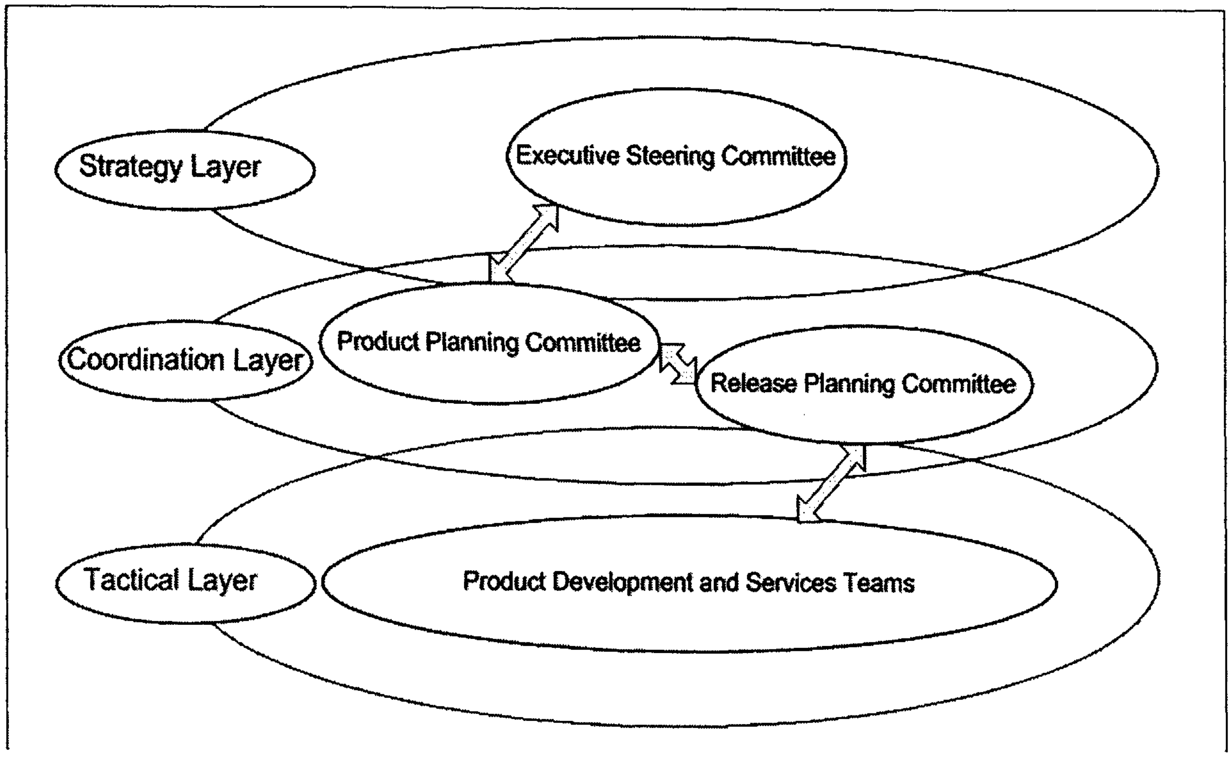

Strategy Layer
Executive Steering Committee
Coordination Layer
Product Planning Committee
Release Planning Committee
Tactical Layer
Product Development and Services Teams

Figure 6. Advanced implementation of a value-based organizational structure

This particular implementation utilizes functioning or working committees to represent the coordinating layer's middle, which is made up of managers or director-level employees who are just above the detailed tactical work, but not in the higher level executive circles. Smaller startup companies will essentially organize this way implicitly and may have only one or two people in the middle layer. It is only when companies have to begin creating explicit structures that they lose this flow because they adopt a more standard hierarchical structure which serves to limit rather than facilitate the flow of information across multiple levels. By focusing an entire layer of the organization on ensuring proper alignment between the strategic objectives on one hand and the tactical plans on the other, a company is more likely to achieve them and much less likely to focus on just the near term objectives or to let political issues get in the way.

It's worth noting that if mistrust already exists within the organization or between key individuals, just putting them on the same "team" doesn't automatically mean that they will be more effective or suddenly get along. These types of issues still have to dealt with, ideally directly and separate from the team meetings as they can otherwise fester and impact others on the cross-functional team. I have, however, witnessed this type of coordinating team help create trust and build healthy relationship by doing a couple of things well:

- Issues and concerns related to the work are discussed openly creating a greater understanding of team members' various perspectives and why they may be fighting so hard for a specific feature or target date. This helps to build empathy for co-workers' motives rather than suspicion
- Shared success is a very effective way for people from different groups within the company to build trust and to share in more of the company's successes

Principle #3 – Focus on value creation

All individuals, teams, divisions, roles, structures, processes, requirements, tasks, etc. should be judged based on its ability to add value to the company's strategic objectives, its customers, or its employees

While this principle does not provide any specific guidance in terms of creating new structures or processes, it should be the final measure by which any employee, requirement, decision, process or structure should be judged. If something does not provide value, then it should be modified to do so, or replaced by something that does.

This may seem like an obvious point, but in my experience companies frequently create or maintain structures, processes, policies, etc., that do not add value to the company, its customers, or its employees or, more often, they conflict with higher priorities or value-adding structures, processes and policies. A common example of this would be a compensation policy that rewards employees solely on individual performance and essentially punishes employees if they fall behind on their own tasks because they helped a different team complete a higher priority (higher value) task. The primary definition of value within an organization should be derived from or at least be consistent with its Vision or Mission Statement as well as its product's primary value proposition

Principle #4 – Continuous planning and execution

Decoupling the Planning (strategic) and Execution (tactical) functions within the separate organizational layers can also increase the benefits of an iterative approach to software development by allowing each function to be continuous and therefore minimize the negative impacts of certain dependencies.

There are two primary practices that a value-based framework can help a software company stay productive rather than suffering through the usual up-and-down cycles as each group hurries between releases to complete work in order to hand it off the next.

The most important practice is continual product planning, designing, and reprioritizing based on real-time market demands rather than what the plan might have directed six months or a year ago (which was likely based on even older market conditions.) Product planning will sometimes shut down for months after a feature list is agreed to for an upcoming release. But, since the coordination team in the middle layers is persistent over multiple releases rather than temporarily partnered only be to separated again, each function can operate at full capacity all the time and can simply draw from the other the necessary or required information (status and priorities) on an as-needed basis rather than having to wait for each other to complete before beginning the next cycle.

The other benefit is that resources can be more easily shifted from lower priority tasks or tasks where there is less risk of a dependent logjam. Having a cross-functional coordination team with members from each major group within the company, everyone needs to agree on the company's priorities rather than focusing on the priorities for just their team. See **Figure 7** below for an illustration of this effect.

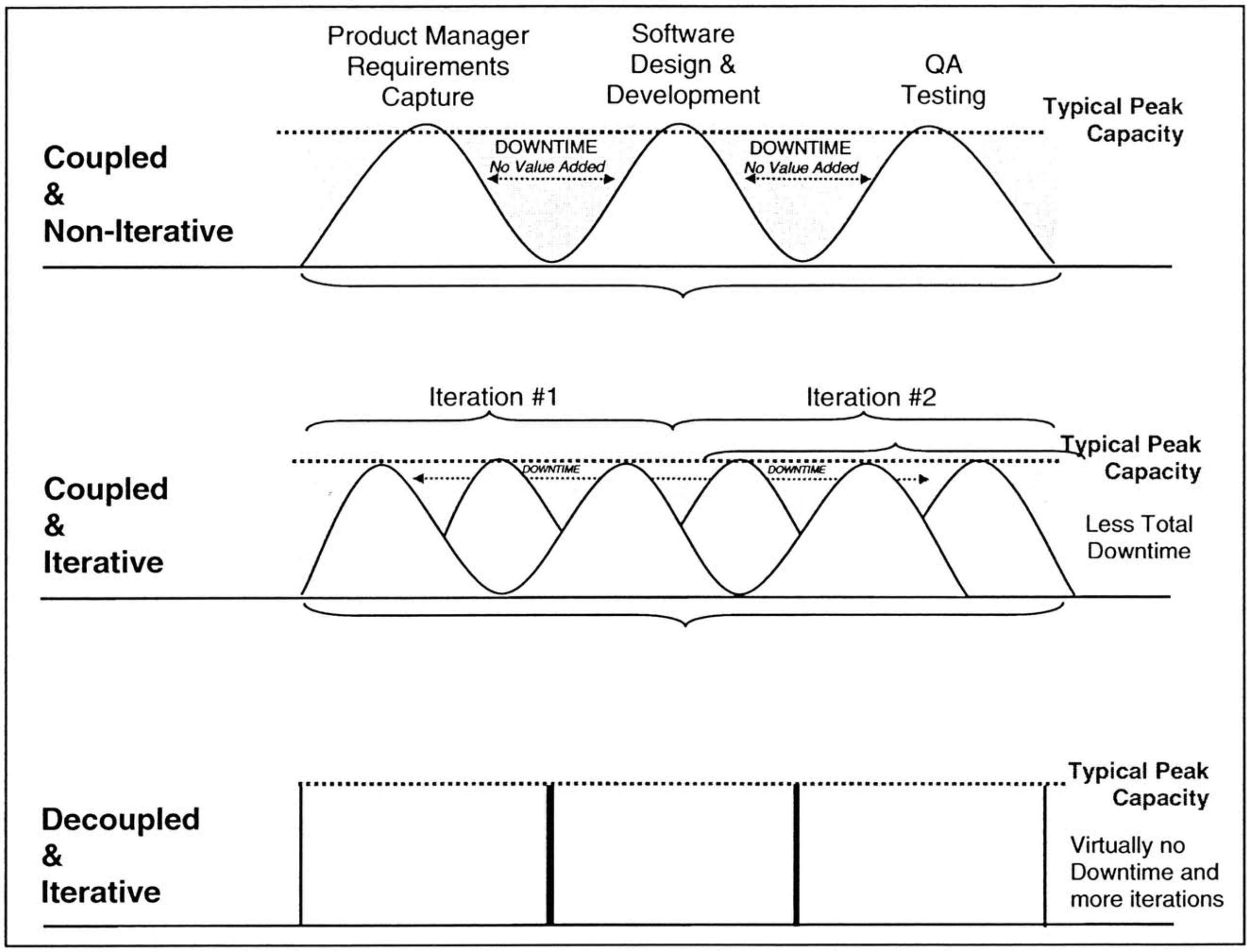


Figure 7. Decoupled and Iterative Model Effects on Team Capacity and Efficiency

Principle #5- Meta-teams and Meta-roles

Utilize the concept of meta-teams and meta-roles as a way to maintain staffing flexibility and to more easily move back and forth across group boundaries if needed in order to staff certain shared efforts.

This is an important method of applying new teams or structures over an existing organizational structure without necessarily having to officially change the existing structure and fight the political battles and other common difficulties associated with changing employees' roles or creating new ones.

The most common example in my experience involved requirements teams. In a "value driven" approach, requirements are the main currency of value for a product company, and rather than organizing around temporary projects or releases, cross-functional requirement teams are formed to own the successful specification, design, testing, delivery, and maintenance or requirements for the life of that requirement. And while most requirements are functional in nature, there are also technical or non-functional requirements that may be just as important for the success of the product, but often get ignored by product management because they lack the expertise or understanding necessary to drive that requirement through.

By specifying the roles within a requirements team as meta-roles rather than specific positions or titles within the company, the role of subject matter expert (SME) can be filled either by someone from product management (in the case of functional requirements) or by someone from the development group in the case of non-functional or technical requirements. Please see **Figure 8** for an example of how meta-roles and meta-teams can be used to help create and validate requirements.

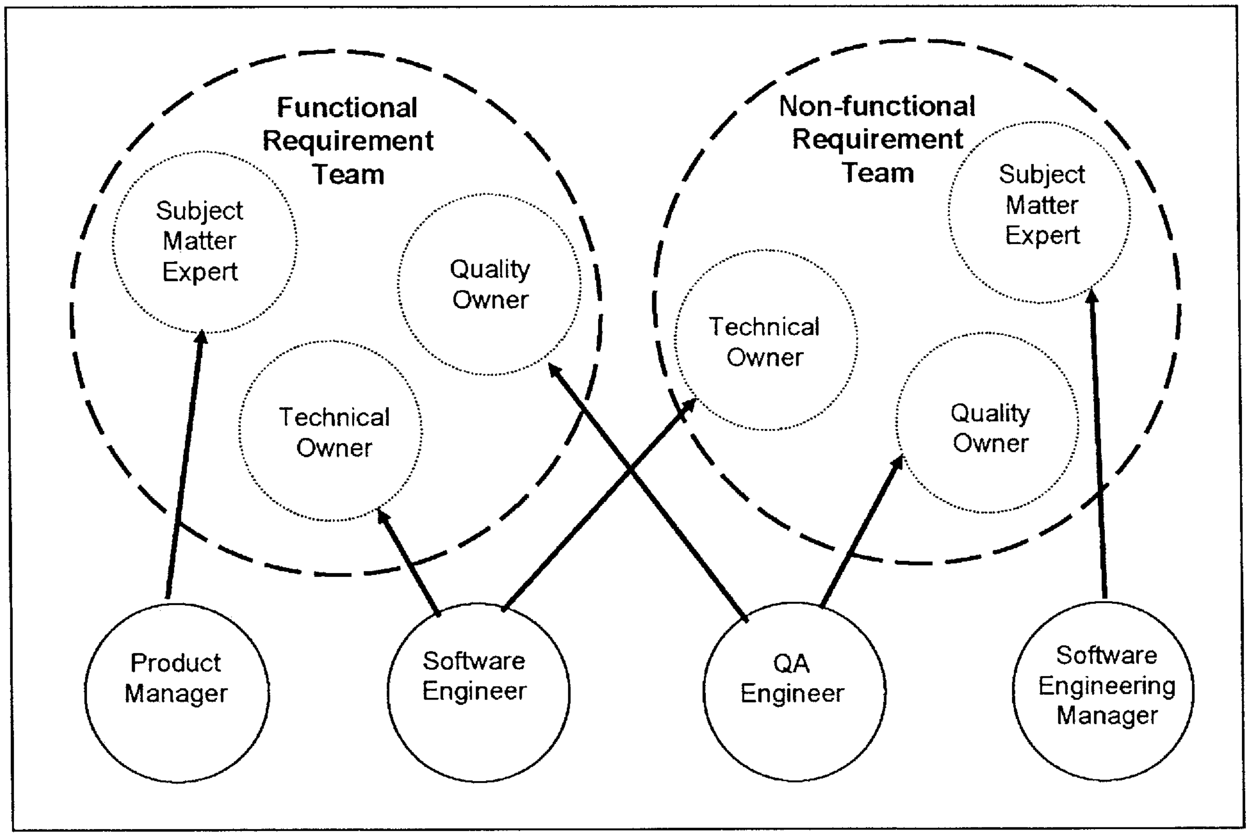


Figure 8. Meta-Roles" and Assignments within Requirement Teams

Figure 9 illustrates a higher level view of a complete meta-team structure on all three levels. This example was actually implemented as-is at a previous company in late 2006.

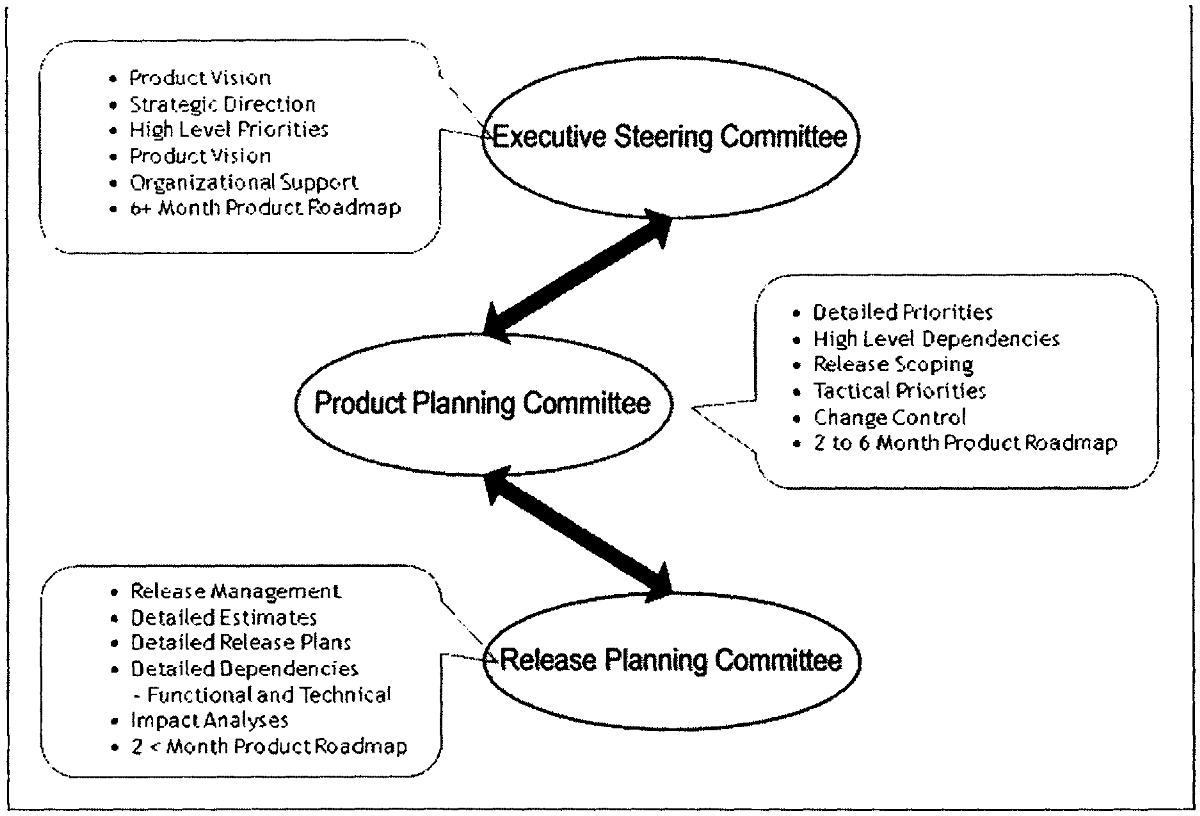


Figure 9. High- level view of a complete meta-team structure

Principle #6- Manage forward

Project management should focus more on adjusting and optimizing the plan forward rather than on maintaining the original project plan as new knowledge is gained and understanding of the effort is increased

The original plan is usually out of date by the second day of an average software development project. Refusing to recognize or record this fact as the original plan continues to become less relevant does nothing to help the project or the company to meet its original objectives when circumstances have changed. However, this is a situation where separating the tactical and the strategic can be very useful.

In my experience if a plan is referred to as a tactical rather than a strategic plan (and the strategic plan remains static), management tends to be more likely to allow the tactical plans to be updated and optimized as needed in order to adjust to everything that is constantly changing during a software development project. Ironically, this leeway actually increases the odds that the original plan's objectives will be met since the team is able to make important adjustments that can help keep them or get them back on track including:

Principle #7- Manage risk early and often

Managing and minimizing risk on software development projects requires daily, hands-on attention rather than a standard approach which relies on weekly status meetings.

If this principle sounds similar to XP's daily stand-up meetings or Scrum's daily scrum meetings, that's probably because it is. It's also the single most effective practice I've come across for managing risk during the development phase on a software project I've been able to implement daily development meetings at four separate software companies and each time I did so, there was a noticeable improvement in software performance measurements across the board. It's also easy to understand why as managing risk earlier means that less time is wasted either by heading down the wrong path or by struggling with an issue that additional input could have resolved immediately. It also means that the meetings can average fifteen minutes rather than the two hours that weekly status meetings typically take.

Principle #8- Value must be valued

Ensure that the compensation system and the performance review process recognize and equitably reward value creation across the organization

While the larger external market for jobs primarily determines standard compensation levels for different roles, companies can still be creative. For example, rather than simply rewarding the sales group based on commissions from sales of product that may result as much from a good product design as a good sales strategy, also compensate those in responsible in the development group with "ownership" of the product via revenue sharing or actual stock (not just futures).

While this may seem relatively extreme, it is also fair given the other collective principles, because without good products there would be nothing to sell and ownership in the product/company itself is a natural incentive to build better and "value-added" products for developers in the same way that commissions are a natural incentive for sales to sell the product.

Conclusion

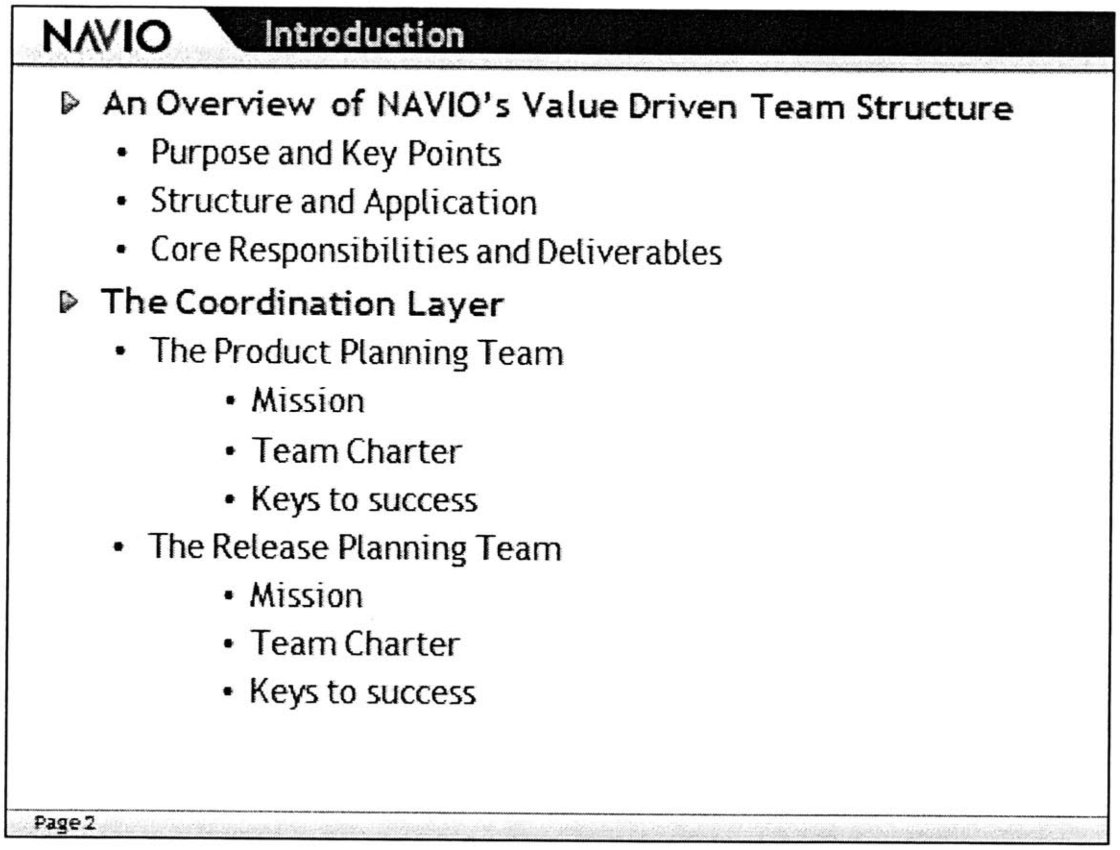
Having benefited so much from the writings of others throughout my career including Steve McConnell, Gerald Weinberg, and Michael Cusumano just to name a few, my hope in writing this thesis was to contribute something original that would make use of my experience and my time at MIT and that could be of some help to others in the future. Having spent much of the last two years culling through research and papers on software development going back to the 1950's, it quickly became clear that there is truly very little related to software process and practices that has not been suggested or tried in one form or another.

This realization only reinforced what I had already been coming to understand after fifteen years of experience in software development, much of it working with process, which is that the real key to success for those who are responsible for choosing, implementing, and managing software development processes is to have a direct understanding of why specific practices are or are not helpful and to be able to predict in what situations those same practices might cease to be helpful. This is not something that can be learned by simply picking up a book and implementing a process through rote adherence or by reading someone else's blog entries.

In discussing the additional factors related to the HP study my hope was to add some additional insight that would complement the existing analysis and to suggest additional factors for consideration in future studies. In creating the value-based framework, my goal was to provide a working framework based on organizational principles that could enhance an average software organization's ability to implement and manage a wide range of different processes based on what works best for that organization. It is my sincere hope that others will be able to bring these two ideas together in a way that will benefit his or her company's software development efforts.

Appendix

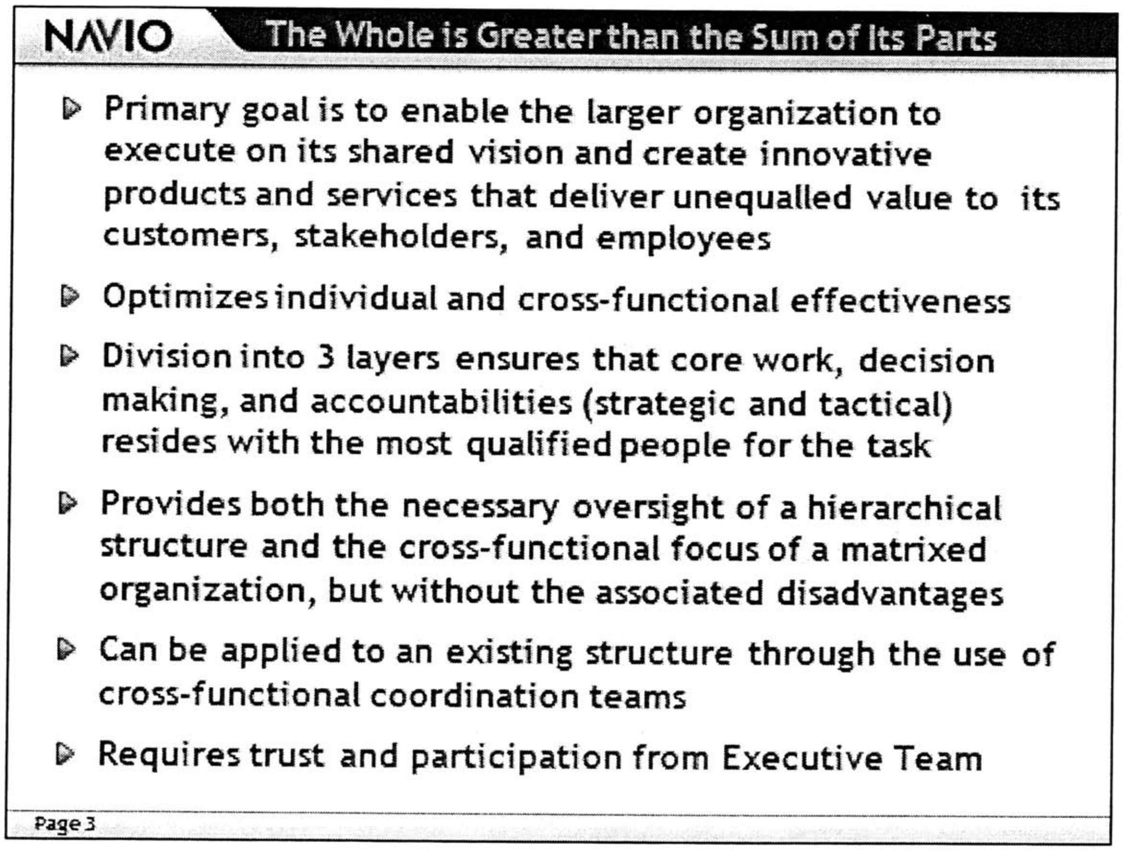
Value-based structure exhibits. These documents were used to help implement a value-based organizational framework at Navio, Inc. in late 2006.


NAVIO

Introduction

- An Overview of NAVIO's Value Driven Team Structure
  - Purpose and Key Points
  - Structure and Application
  - Core Responsibilities and Deliverables
- The Coordination Layer
  - The Product Planning Team
    - Mission
    - Team Charter
    - Keys to success
  - The Release Planning Team
    - Mission
    - Team Charter
    - Keys to success

Page 2


NAVIO

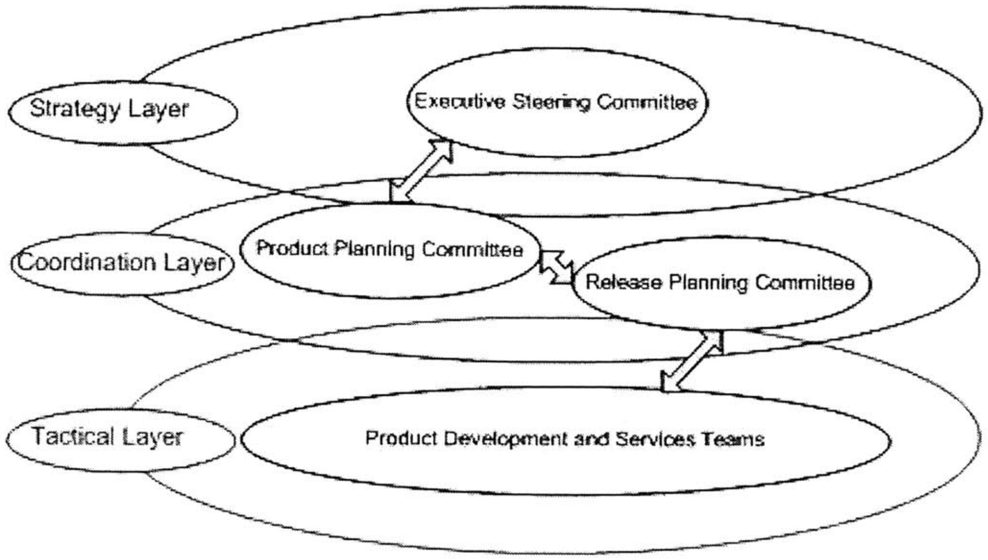
The Whole is Greater than the Sum of Its Parts

- Primary goal is to enable the larger organization to execute on its shared vision and create innovative products and services that deliver unequalled value to its customers, stakeholders, and employees
- Optimizes individual and cross-functional effectiveness
- Division into 3 layers ensures that core work, decision making, and accountabilities (strategic and tactical) resides with the most qualified people for the task
- Provides both the necessary oversight of a hierarchical structure and the cross-functional focus of a matrixed organization, but without the associated disadvantages
- Can be applied to an existing structure through the use of cross-functional coordination teams
- Requires trust and participation from Executive Team

Page 3

NAVIO A Layered or Value-Driven Structure

The value-driven structure optimizes NAVIO's ability to execute on its shared vision and create innovative products and services that deliver unequalled value to its customers, stakeholders, and employees

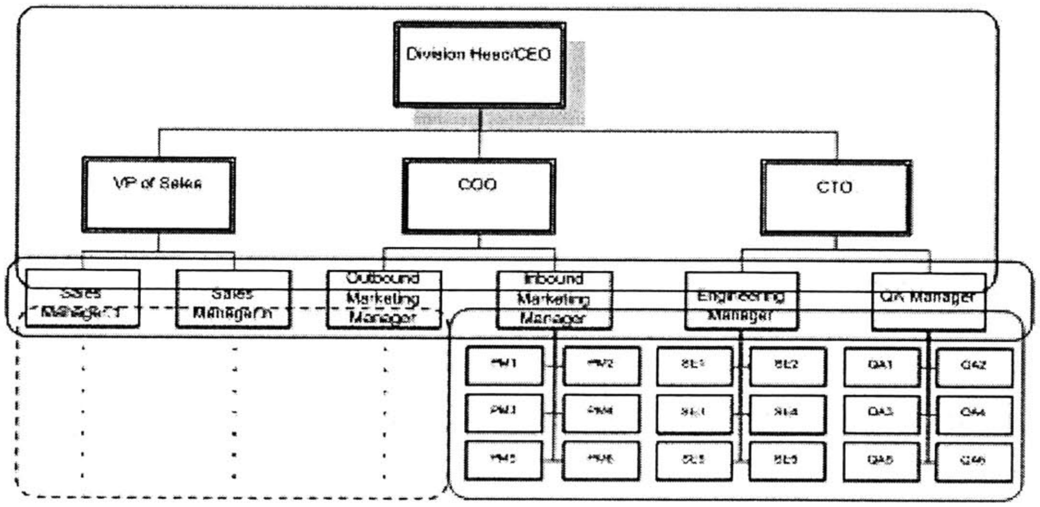


Strategy Layer
Executive Steering Committee
Coordination Layer
Product Planning Committee
Release Planning Committee
Tactical Layer
Product Development and Services Teams

Page 4

NAVIO Applying the Value-Added Structure

The Value Driven Structure can be applied or to a typical hierarchical software product development structure

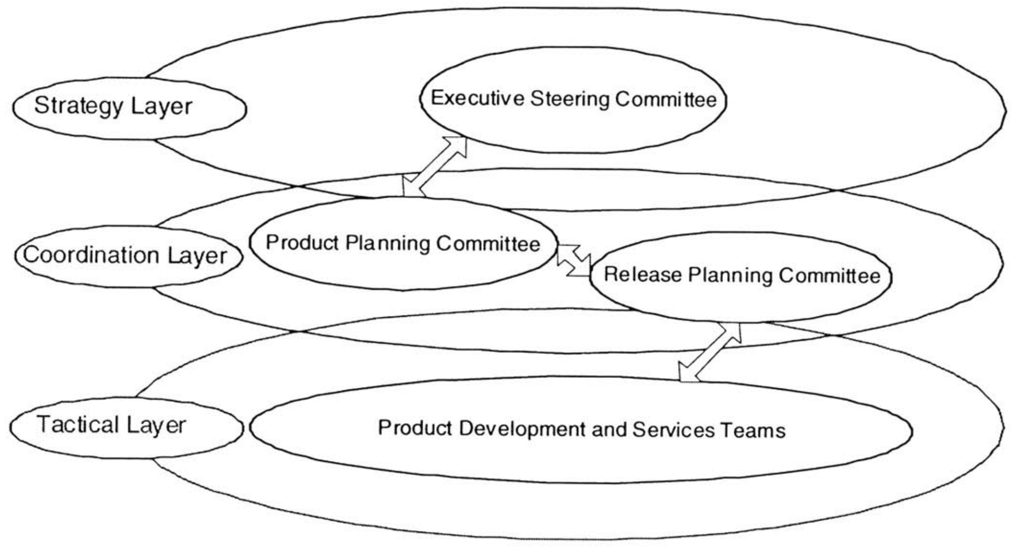


Page 5

NAVIO

A Layered or Value-Driven Structure

The value-driven structure optimizes NAVIO's ability to execute on its shared vision and create innovative products and services that deliver unequalled value to its customers, stakeholders, and employees



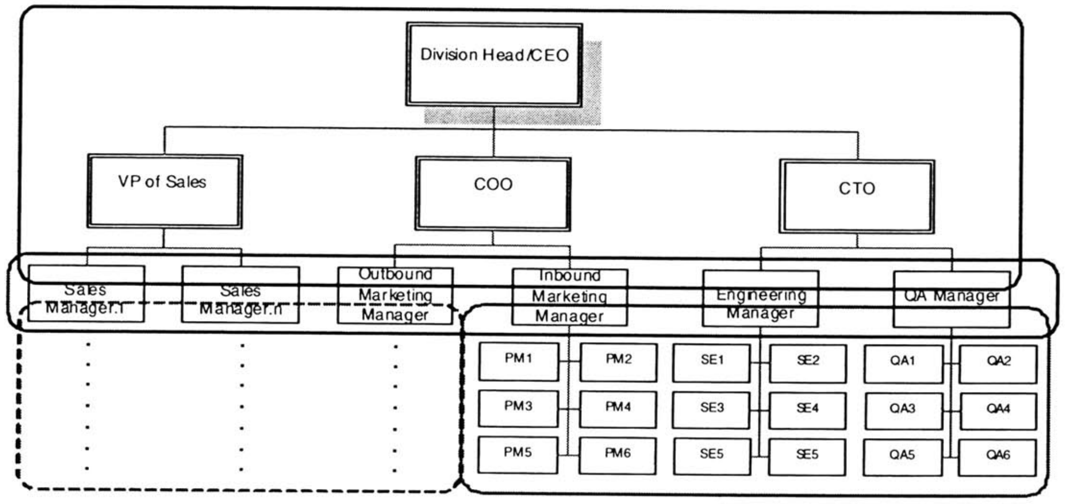
Strategy Layer
Executive Steering Committee
Coordination Layer
Product Planning Committee
Release Planning Committee
Tactical Layer
Product Development and Services Teams

Page 4

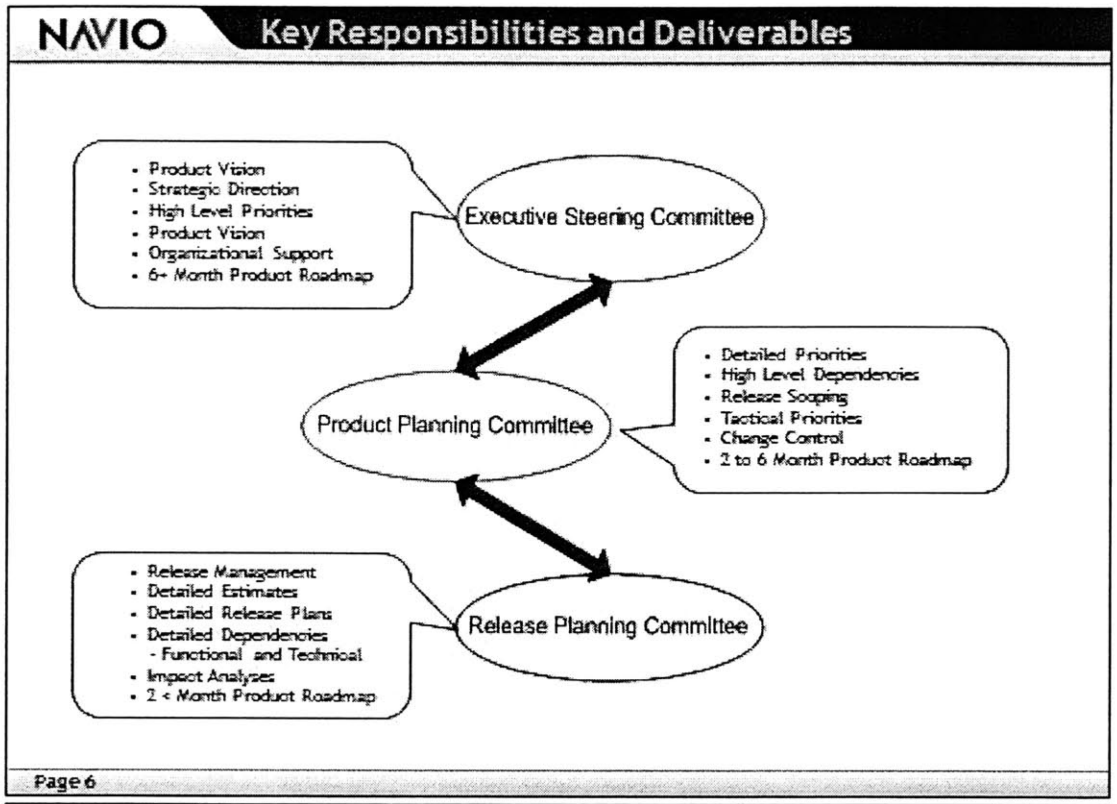
NAVIO

Applying the Value-Added Structure

The Value Driven Structure can be applied or to a typical hierarchical software product development structure



Page 5



NAVIO

Key Responsibilities and Deliverables

Executive Steering Committee

- Product Vision
- Strategic Direction
- High Level Priorities
- Product Vision
- Organizational Support
- 6+ Month Product Roadmap

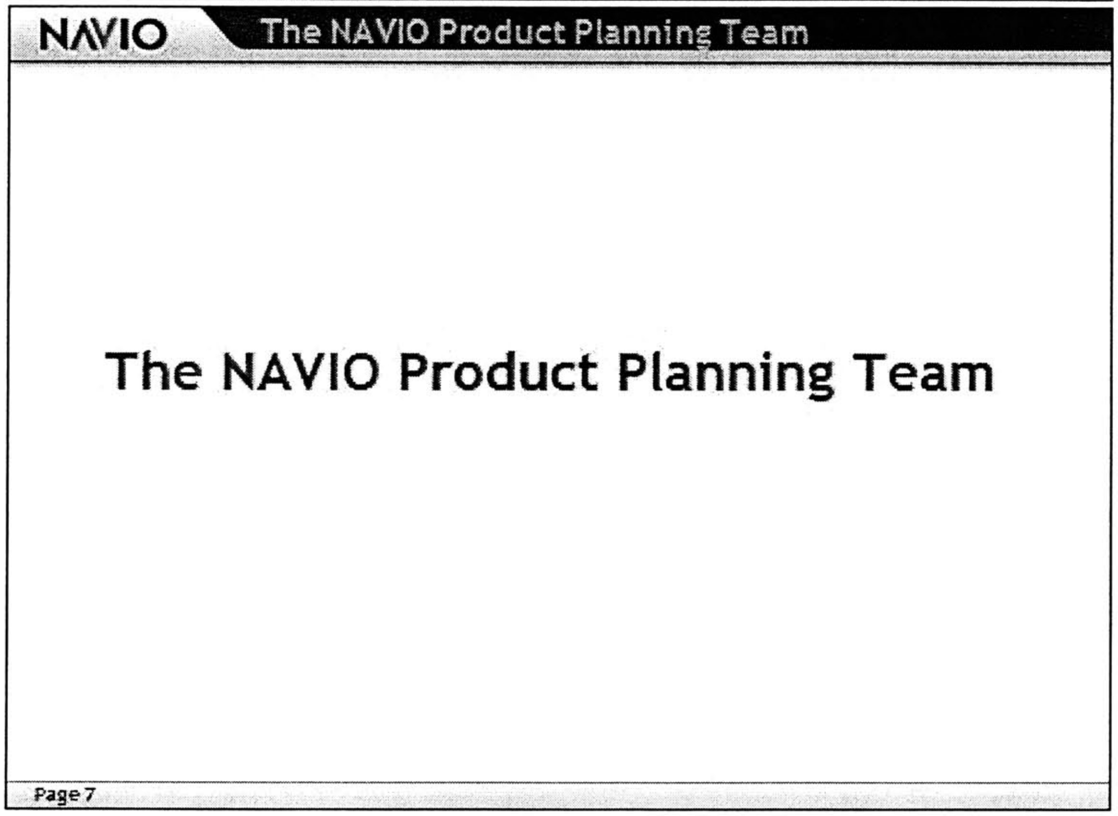
Product Planning Committee

- Detailed Priorities
- High Level Dependencies
- Release Scoping
- Tactical Priorities
- Change Control
- 2 to 6 Month Product Roadmap

Release Planning Committee

- Release Management
- Detailed Estimates
- Detailed Release Plans
- Detailed Dependencies
  - Functional and Technical
- Impact Analyses
- 2 < Month Product Roadmap

Page 6

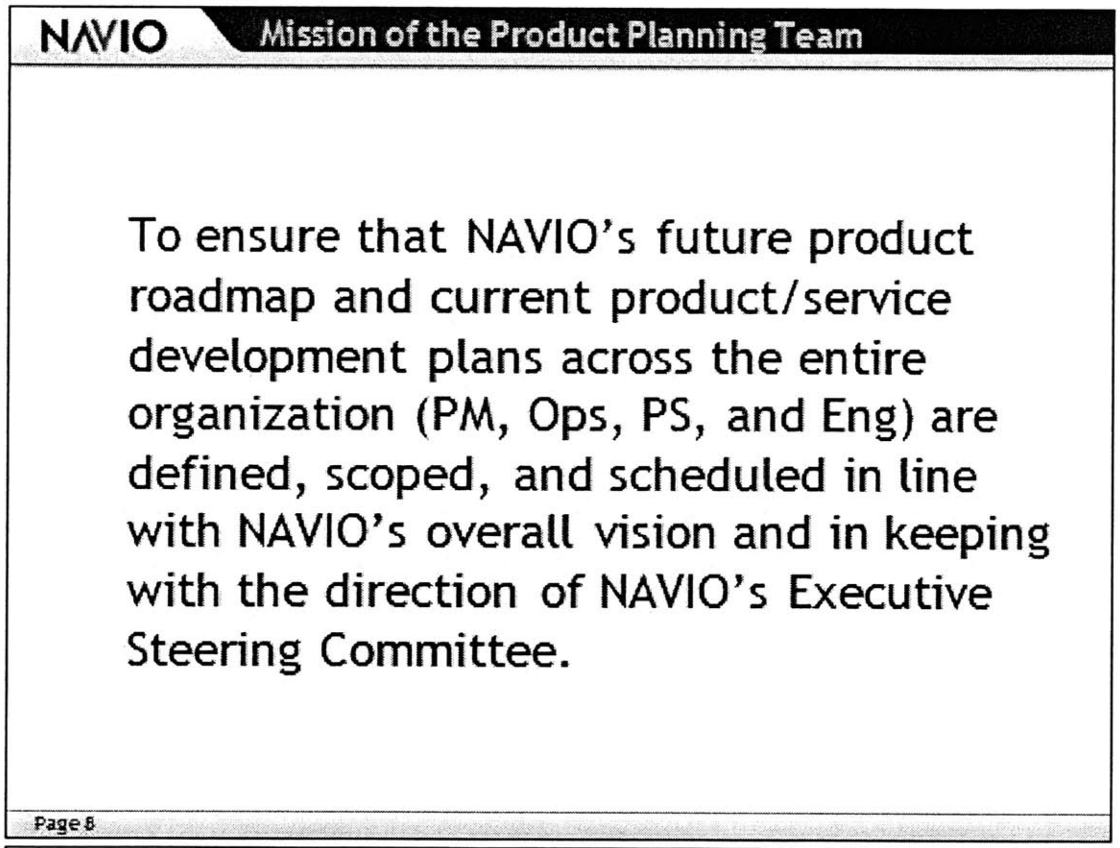


NAVIO

The NAVIO Product Planning Team

The NAVIO Product Planning Team

Page 7

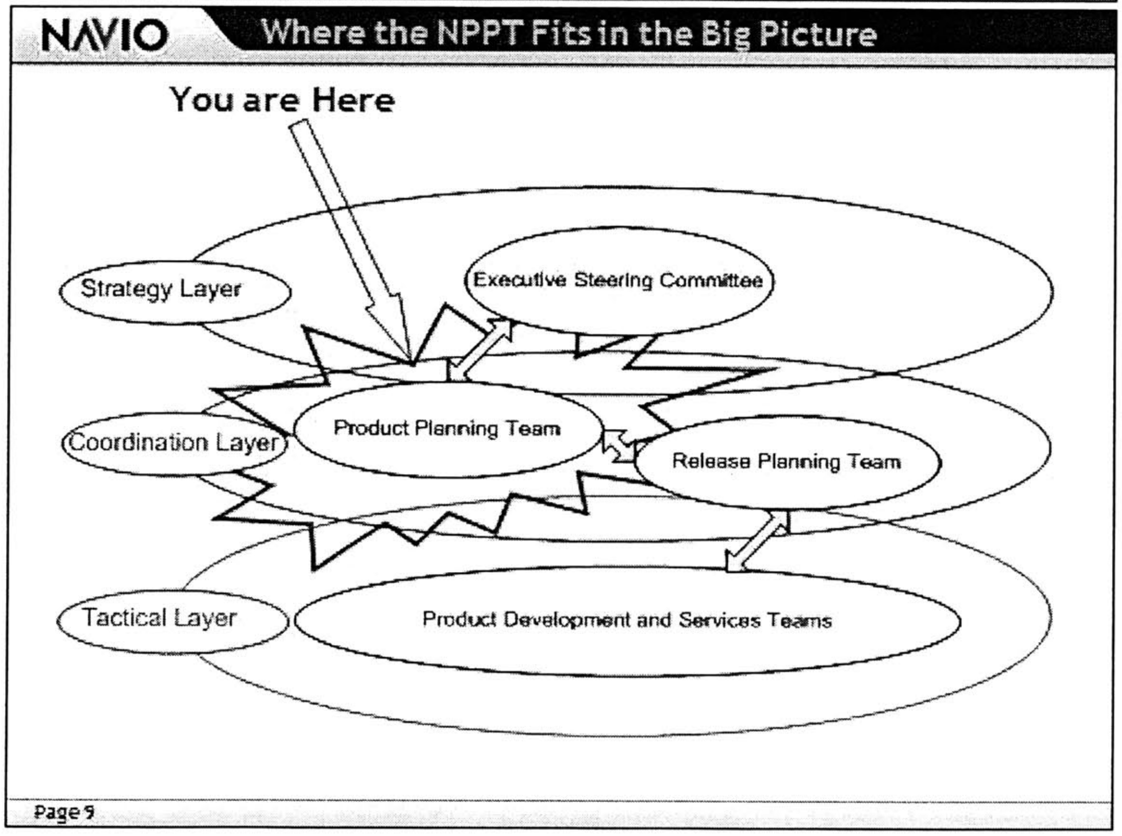


NAVIO

Mission of the Product Planning Team

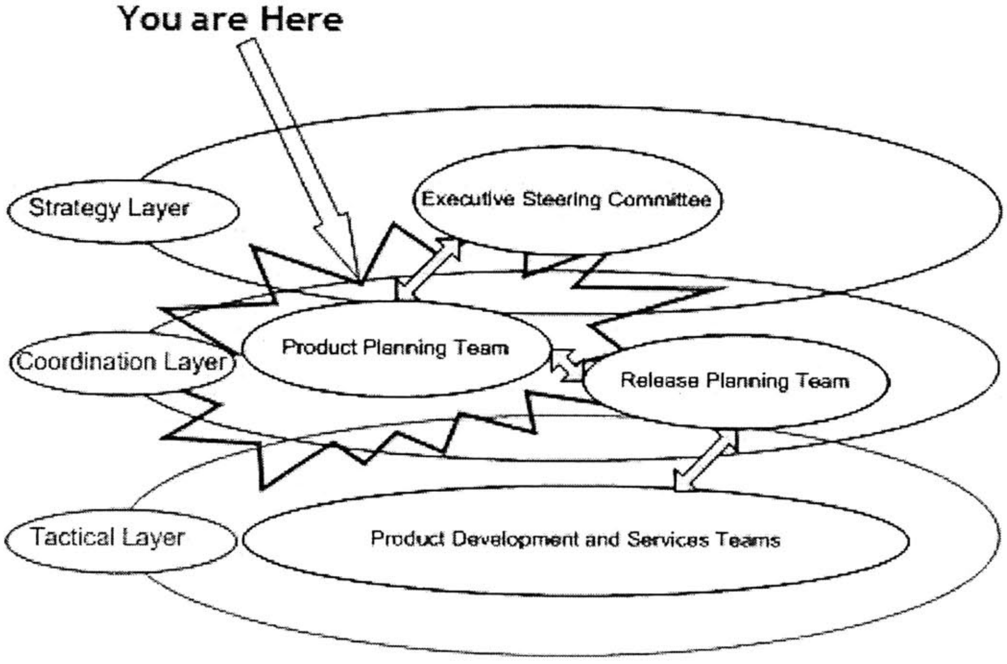
To ensure that NAVIO's future product roadmap and current product/service development plans across the entire organization (PM, Ops, PS, and Eng) are defined, scoped, and scheduled in line with NAVIO's overall vision and in keeping with the direction of NAVIO's Executive Steering Committee.

Page 8

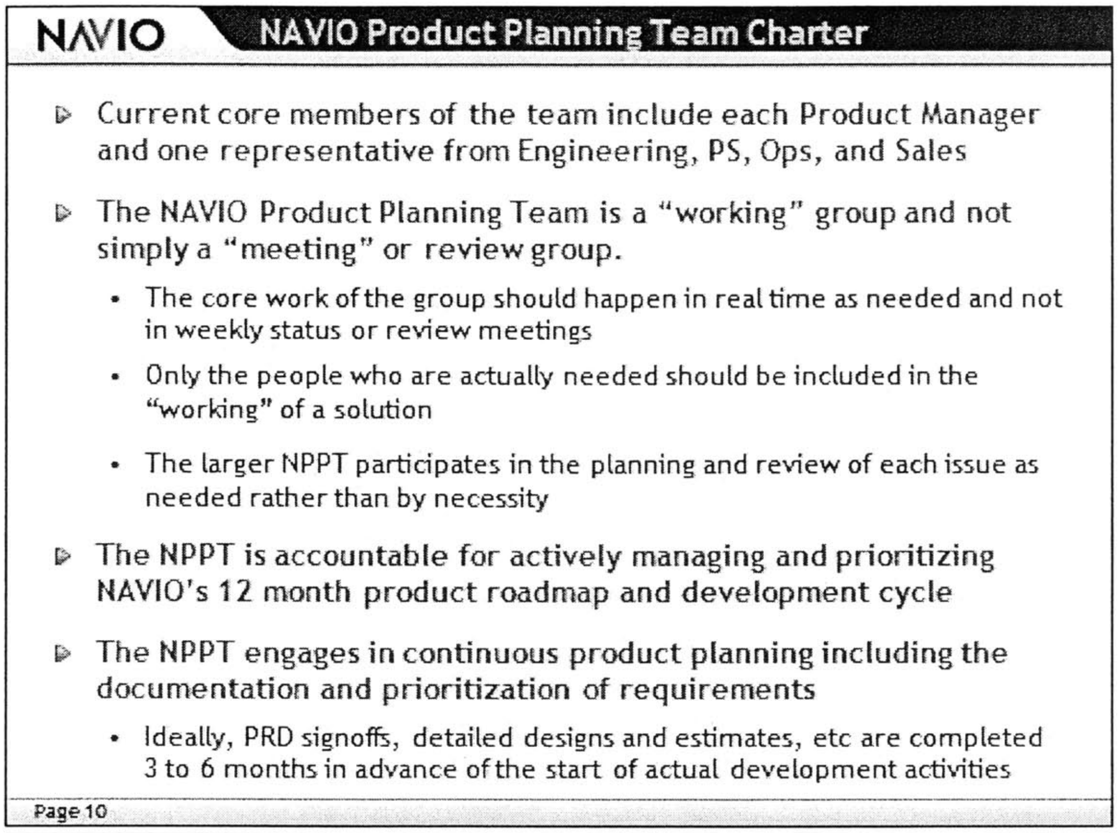


NAVIO

Where the NPPT Fits in the Big Picture



Page 9

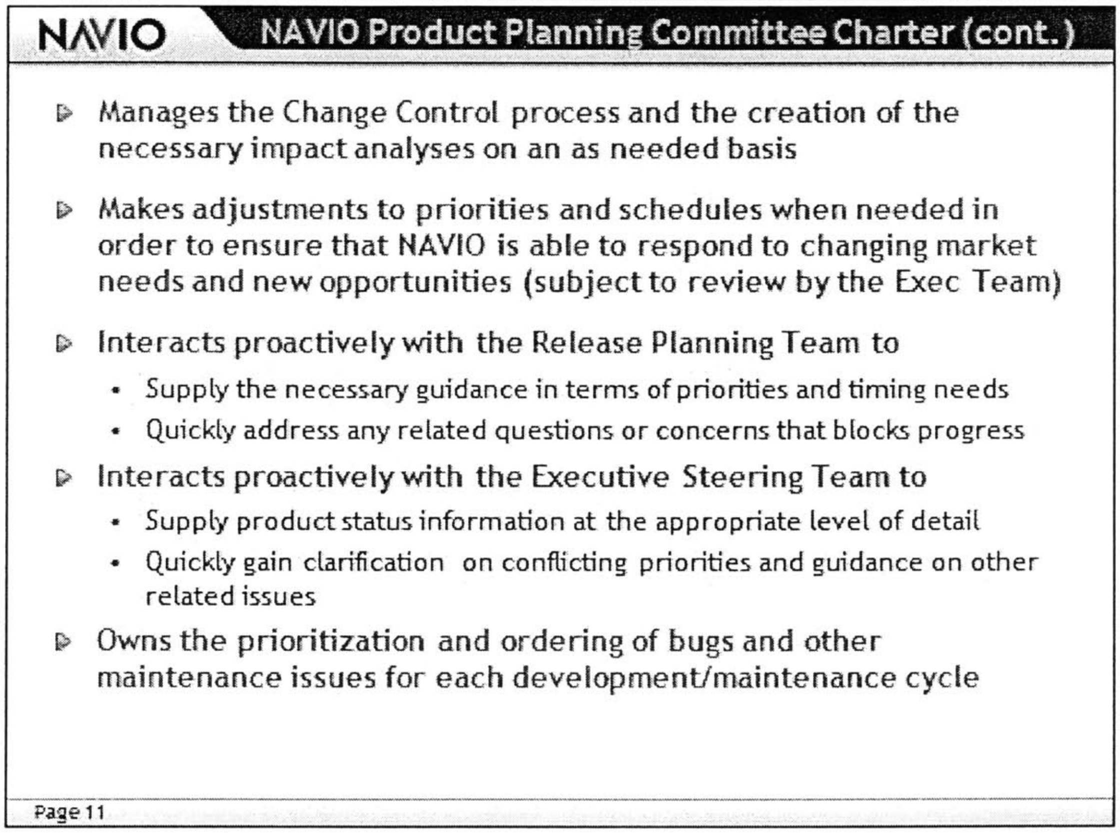


NAVIO

NAVIO Product Planning Team Charter

- Current core members of the team include each Product Manager and one representative from Engineering, PS, Ops, and Sales
- The NAVIO Product Planning Team is a "working" group and not simply a "meeting" or review group.
  - The core work of the group should happen in real time as needed and not in weekly status or review meetings
  - Only the people who are actually needed should be included in the "working" of a solution
  - The larger NPPT participates in the planning and review of each issue as needed rather than by necessity
- The NPPT is accountable for actively managing and prioritizing NAVIO's 12 month product roadmap and development cycle
- The NPPT engages in continuous product planning including the documentation and prioritization of requirements
  - Ideally, PRD signoffs, detailed designs and estimates, etc are completed 3 to 6 months in advance of the start of actual development activities

Page 10

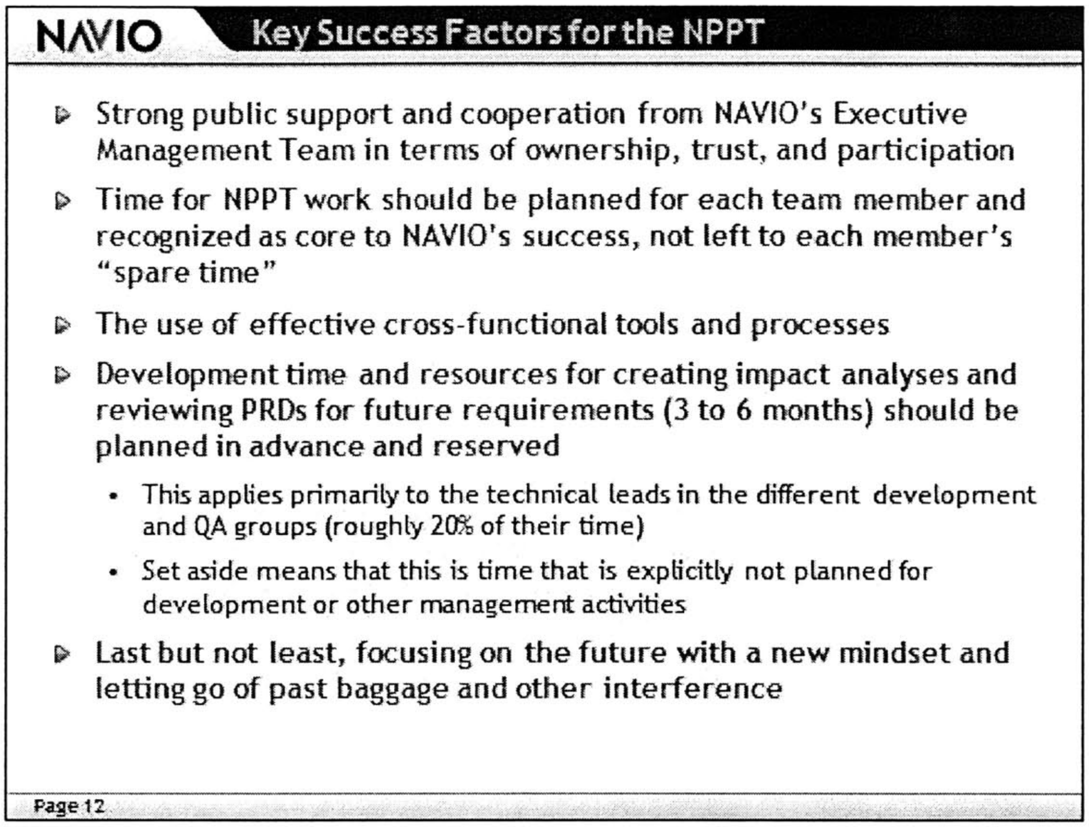


NAVIO

NAVIO Product Planning Committee Charter (cont.)

- Manages the Change Control process and the creation of the necessary impact analyses on an as needed basis
- Makes adjustments to priorities and schedules when needed in order to ensure that NAVIO is able to respond to changing market needs and new opportunities (subject to review by the Exec Team)
- Interacts proactively with the Release Planning Team to
  - Supply the necessary guidance in terms of priorities and timing needs
  - Quickly address any related questions or concerns that blocks progress
- Interacts proactively with the Executive Steering Team to
  - Supply product status information at the appropriate level of detail
  - Quickly gain clarification on conflicting priorities and guidance on other related issues
- Owns the prioritization and ordering of bugs and other maintenance issues for each development/maintenance cycle

Page 11

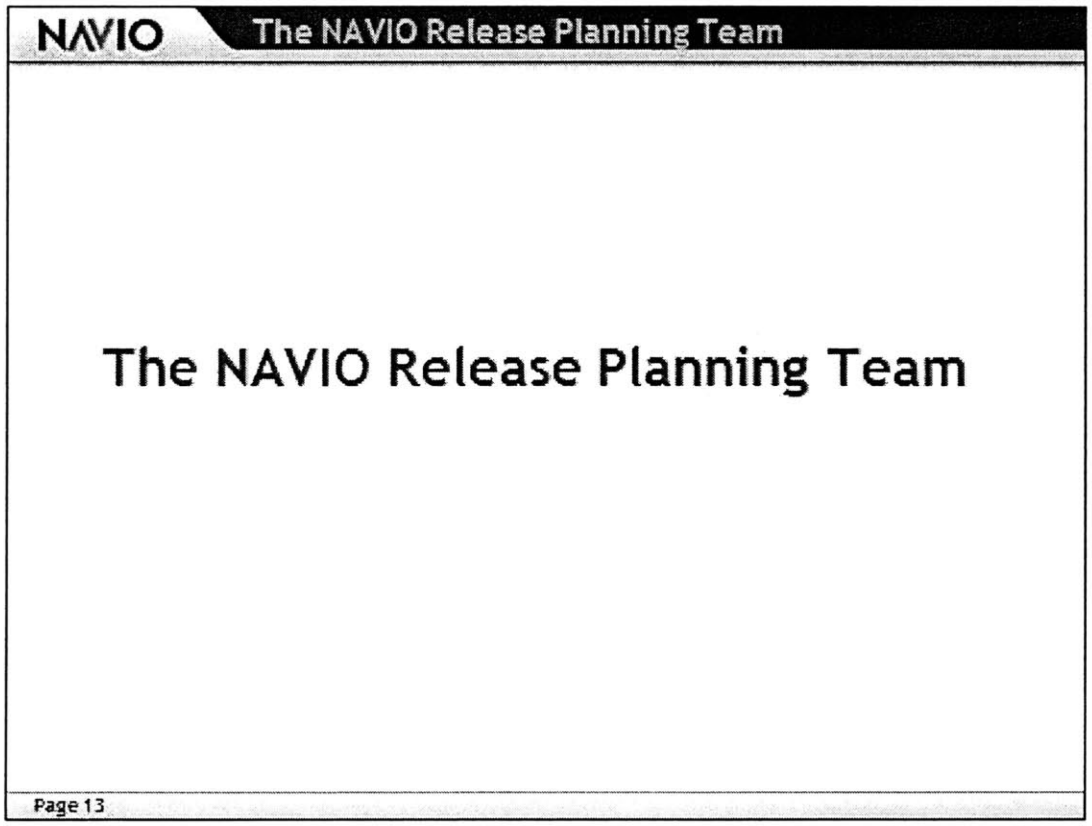


NAVIO

Key Success Factors for the NPPT

- Strong public support and cooperation from NAVIO's Executive Management Team in terms of ownership, trust, and participation
- Time for NPPT work should be planned for each team member and recognized as core to NAVIO's success, not left to each member's "spare time"
- The use of effective cross-functional tools and processes
- Development time and resources for creating impact analyses and reviewing PRDs for future requirements (3 to 6 months) should be planned in advance and reserved
  - This applies primarily to the technical leads in the different development and QA groups (roughly 20% of their time)
  - Set aside means that this is time that is explicitly not planned for development or other management activities
- Last but not least, focusing on the future with a new mindset and letting go of past baggage and other interference

Page 12

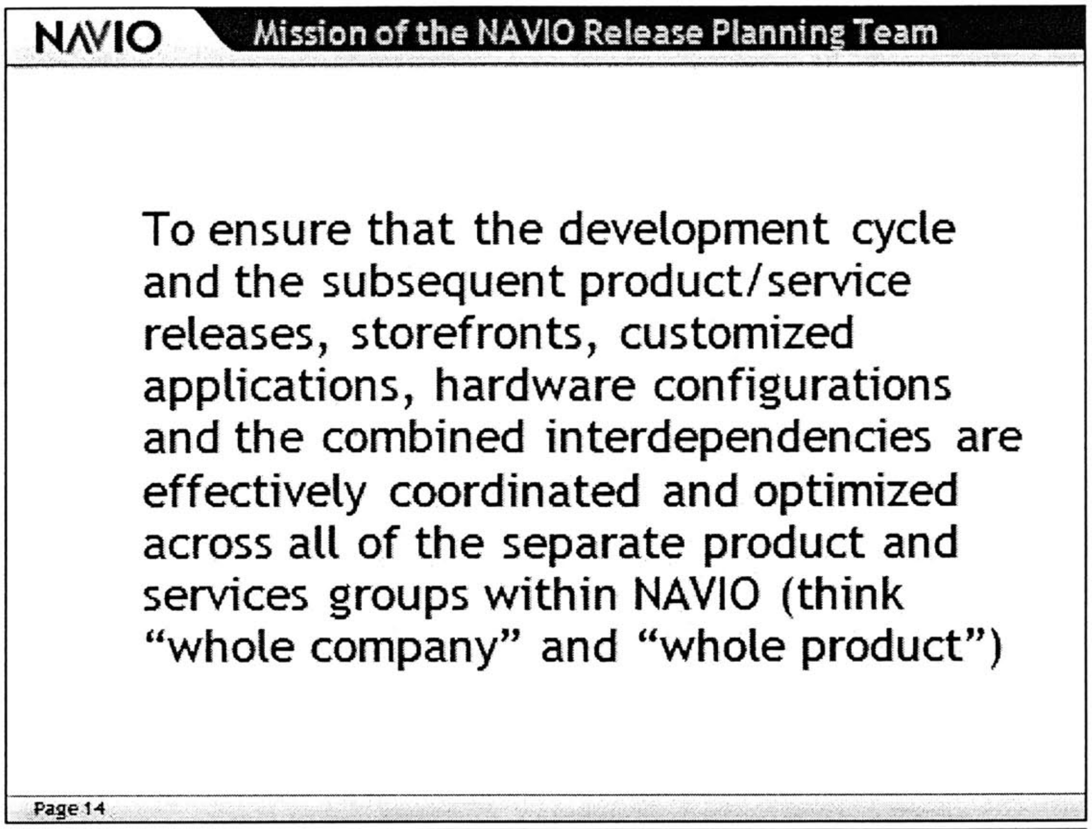


NAVIO

The NAVIO Release Planning Team

The NAVIO Release Planning Team

Page 13

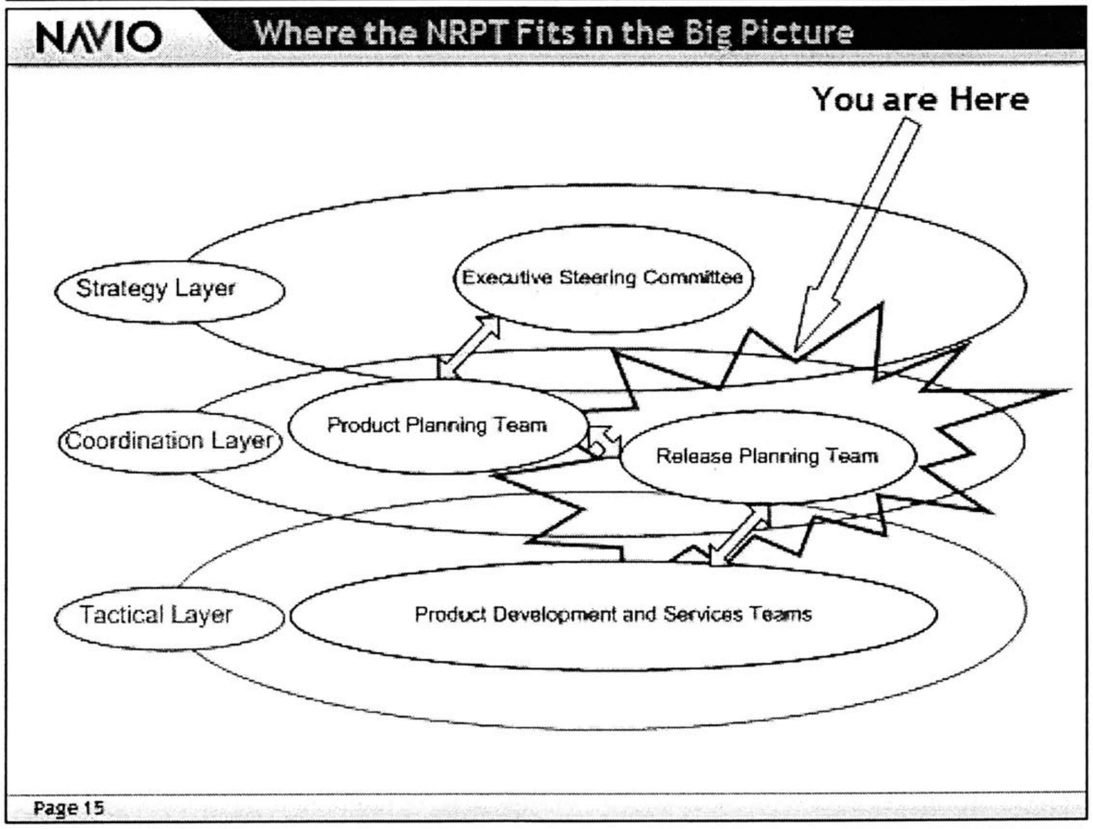


NAVIO

Mission of the NAVIO Release Planning Team

To ensure that the development cycle and the subsequent product/service releases, storefronts, customized applications, hardware configurations and the combined interdependencies are effectively coordinated and optimized across all of the separate product and services groups within NAVIO (think “whole company” and “whole product”)

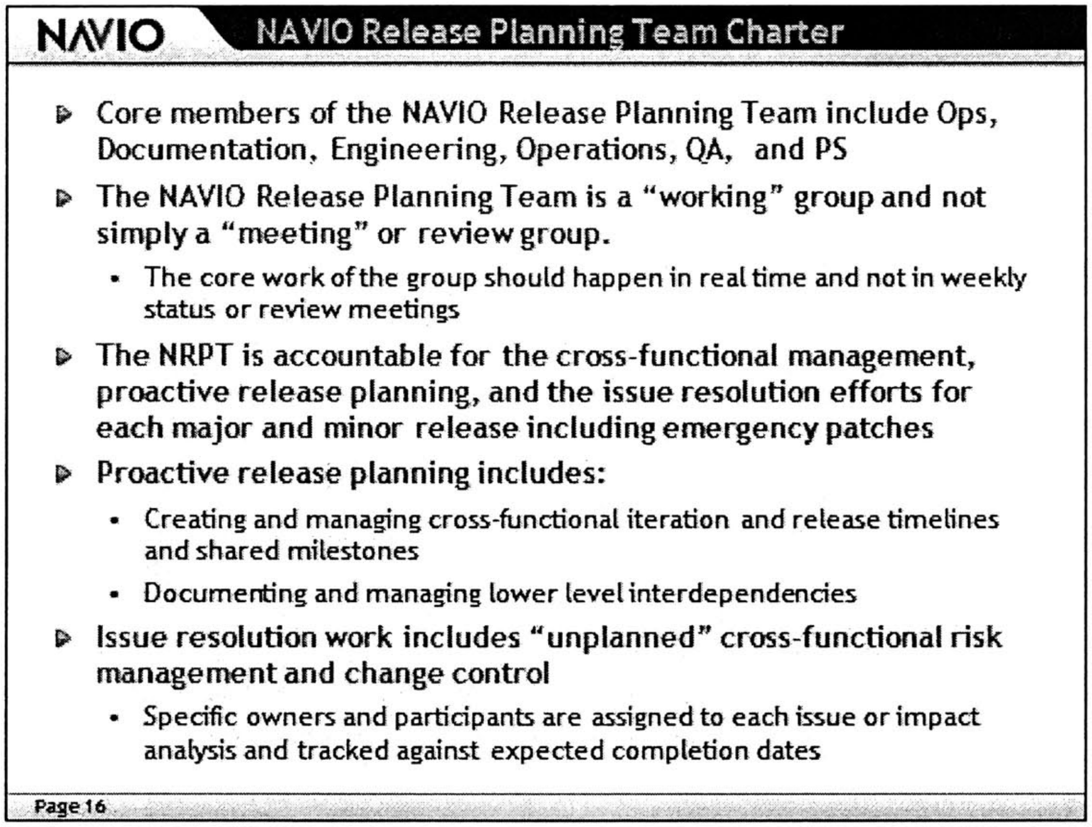
Page 14



NAVIO

Where the NRPT Fits in the Big Picture

Page 15

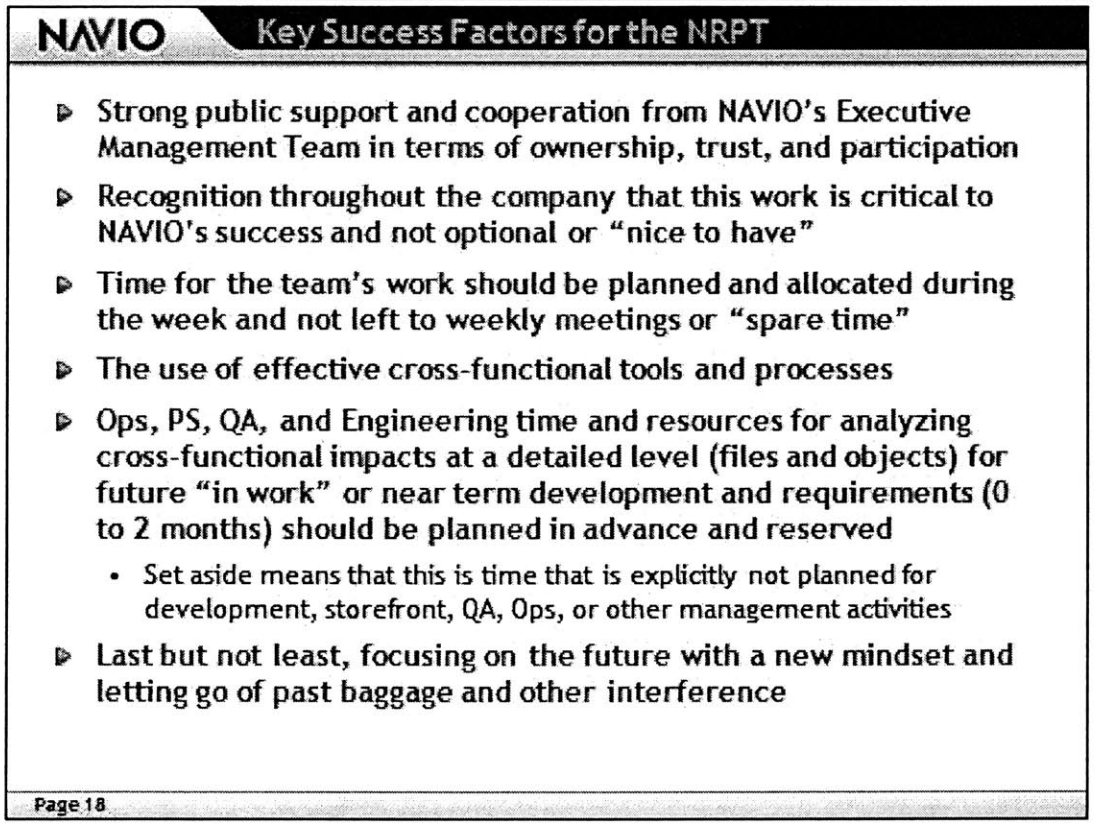


NAVIO

NAVIO Release Planning Team Charter

- Core members of the NAVIO Release Planning Team include Ops, Documentation, Engineering, Operations, QA, and PS
- The NAVIO Release Planning Team is a "working" group and not simply a "meeting" or review group.
  - The core work of the group should happen in real time and not in weekly status or review meetings
- The NRPT is accountable for the cross-functional management, proactive release planning, and the issue resolution efforts for each major and minor release including emergency patches
- Proactive release planning includes:
  - Creating and managing cross-functional iteration and release timelines and shared milestones
  - Documenting and managing lower level interdependencies
- Issue resolution work includes "unplanned" cross-functional risk management and change control
  - Specific owners and participants are assigned to each issue or impact analysis and tracked against expected completion dates

Page 16



NAVIO

Key Success Factors for the NRPT

- Strong public support and cooperation from NAVIO's Executive Management Team in terms of ownership, trust, and participation
- Recognition throughout the company that this work is critical to NAVIO's success and not optional or "nice to have"
- Time for the team's work should be planned and allocated during the week and not left to weekly meetings or "spare time"
- The use of effective cross-functional tools and processes
- Ops, PS, QA, and Engineering time and resources for analyzing cross-functional impacts at a detailed level (files and objects) for future "in work" or near term development and requirements (0 to 2 months) should be planned in advance and reserved
  - Set aside means that this is time that is explicitly not planned for development, storefront, QA, Ops, or other management activities
- Last but not least, focusing on the future with a new mindset and letting go of past baggage and other interference

Page 18

References

[i] The Standish Group, "Charting the Seas of Information Technology," Dennis, MA: The Standish Group, 1994. Jones, Capers, *Patterns of Software Systems Failure and Success*, Boston, MA: International Thomson Computer Press, 1996.

[ii] McConnell, Steve, *Professional Software Development: Shorter Schedules, Higher Quality Products, More Successful Projects, Enhanced Careers,* Reading, MA: Addison-Wesley, 2006

[iii] The Standish Group, "Charting the Seas of Information Technology," Dennis, MA: The Standish Group, 1994. Jones, Capers, *Patterns of Software Systems Failure and Success*, Boston, MA: International Thomson Computer Press, 1996.

[iv] McConnell, Steve, *Professional Software Development: Shorter Schedules, Higher Quality Products, More Successful Projects, Enhanced Careers,* Reading, MA: Addison-Wesley, 200

[v] ibid

[vi] ibid

[vii] [Royce, 1970]. W.W. Royce, "Managing the Development of Large Software Systems: Concepts and Techniques," *1970 WESCON Technical Papers, Vol.14*, Western Electronic Show and Convention, Los Angeles, August 25-28, 1970, pp. 1 – 9, Reprinted in the *Proceedings of the 9th International Conference On Software Engineering, March 30 – April 2, 1987, Monterey, California*, IEEE Computer Society Press, Washington, D.C., 1987, pp. 328 – 338.

[viii] M.Campbell-Kelly, *From Airline Reservations to Sonic the Hedgehog—A History of the Software Industry*, The MIT Press, Cambridge, Massachusetts, 2003

[ix] [Brooks, 1987]. F. P. Brooks, Jr., "No Silver Bullet: Essence and Accidents of Software Engineering," *IEEE Computer*, Vol. 20, No. 4, April 1987, pp. 10 - 19.

[x] [Bach, 1994]. I. Jacobson, P. Ng, and I. Spence, "Enough of Processes: Let's Do Practices," *Dr. Dobbs' Journal*, No. 395, April 2007, pp.38 – 46.

[xi] [MacCormack et al., 2003]. A. MacCormack, C. Kermerer, M. Cusumano and Crandall, "Trade-offs between Productivity and Quality in Selecting Software Development Processes," *IEEE Software*, Vol. 16, No. 4, September/October 2003, pp. 78-85.

[xii] B.W. Boehm, *Software Engineering Economics*, Prentice-Hall,

Englewood Cliffs, New Jersey, 1981.

[xiii] [Lientz and Swanson, 1980]. B.P. Lientz and E.B. *Swanson, Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*, Addison-Wesley, Reading, Massachusetts, 1980.

[xiv] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, Reading, Massachusetts, 1999.

[xv] [Cusumano and Selby, 1995]. M.A. Cusumano and R.W. Selby, *Microsoft Secrets —How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People*, The Free Press, New York, New York, 1995.