# An Estimation-Based Management Framework for Enhancive Maintenance in Commercial Software Products

*By Prof. David A. Penny*
*Department of Computer Science, University of Toronto*

## Abstract

In commercial software vendor organizations, evolution of a software product by means of scheduled feature releases is a central activity. Relying on existing work on maintenance task effort prediction, this paper proposes a management framework for periodically capturing updated estimation data and using it as a basis for initial planning and subsequent re-planning of releases. The framework is founded upon a mathematically-stated, metrics-based model of the release cycle tuned to the software vendor environment. It allows non-technical product managers to make sensible, fast-paced and fine-grained release decisions without undue overhead or undue reliance upon the software development organization. This approach has been used successfully in practice by the author as vice president of the software development division of a mid-sized enterprise software vendor company.

## 1. Introduction

A software vendor organization is a commercial enterprise that licenses a common set of software products to its customers. A central activity within the software vendor organization is deciding when the next releases of their software products should be made generally available and what feature enhancements they should contain so as to maximize future revenue. A plan that states these things for a given product is called a *release plan*. Thus, *release planning* is the activity of determining a feasible combination of dates, features, and resourcing for the next release of an existing software product.

The release plan describes the contract between product management (who coordinates the plans on behalf of the business) and software development (who implements the release) as to what new features will be delivered by what date.

In the fast-paced business environment of most software vendors, there is pressure to form the release plan early on, before requirements are worked out in detail, and pressure throughout the release cycle to make changes to the plan owing to the volatility of early effort estimates, human resource issues, and sudden changes to the desirable feature set due to shifting business priorities. If not managed carefully, these frequent changes can lead to a problematic relationship between product management and software development leading to sub-optimal decision making.

In this paper, we present a management framework for release planning. It is centered around a set of continuously updated release plan documents typically deployed as Web pages on a company's intranet. We describe these documents in Section 6, "The Release Plan Document". Supporting these documents is an abstracted, mathematical model of a release cycle that is used to frame the planning constraints and define the quantities we will need to estimate and will want to subsequently measure. We describe this model in Section 4. "A Mathematical Model as a Basis for Release Planning" and then show how to use it for planning in Section 5, "Using the Model for Release Planning". In Section 3, "Principles of Release Planning", we describe qualitatively the constraint that the model states mathematically: that effort available be equal to effort required.

The framework is intended to satisfy and perfect some key CMM practices for level 2 process maturity: project management and management oversight [1]. It also helps to move an organization closer to higher levels by encouraging the definition of a release cycle and guiding the organization in choosing appropriate process metrics to measure.

## 2. Justification

It has long been recognized that software maintenance is a large part of any ongoing software development effort [2]. In the context of software vendor organizations shipping successful software products, these costs dominate. Here, we refer to all types of maintenance, but especially corrective, perfective, and enhancive [3]. To illustrate this point, 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 five developers, a tester, and a documenter. Using a nominal loaded cost per employee of $100k, the cost of this hypothetical initial development is approximately $700k.*

*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 team and five documenters. This follow-on effort represents approximately 100 person-years, at a cost of $10M, and is burning $3.5M a year for maintenance of the product.*

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

There are many worthwhile approaches to improving efficiency during maintenance [4]. In this paper, we concentrate on improving the efficiency of enhancive maintenance by providing effective management controls that enable the business to ensure that the right things (the things that maximize revenue in the case of a software vendor) are being worked on.

This is related to ensuring that requirements are well stated, but takes place earlier than this activity. The purpose of requirements analysis is to ensure that a requirement stated at the highest-level is described in a sufficiently complete and consistent fashion so as to support the development of the new feature [5]. If the detailed requirements are stated incorrectly, the organization will waste effort implementing incorrect requirements thereby reducing the efficiency of enhancive maintenance.

Release planning involves deciding on what “one-liners” (*features* hereafter) will be included in the next release of a software product. Features are usually enhancive, but may be perfective as well (such as performance improvements, modifying the architecture for better maintainability, and so on). If release planning decisions are sub-optimal they have a greater impact on productivity than incorrect detailed requirements (in addition to the wasted implementation effort, the requirements effort is wasted as well).

Only enlightened market and customer-oriented product management can assure that the right new features are going into follow-on releases. We must assume the existence of such a capable organization as a pre-condition. Given their existence, they must be empowered to make decisions. This is only possible through an effective and fine-grained communications mechanism between product management and software development. Software development must be allowed to estimate and implement. Product management must be enabled to decide on dates and feature sets, but only within constraints set by available software development resourcing. The current approach is aimed at improving this communication thus providing a necessary pre-condition for optimal release planning.

## 3. Principles of Release Planning

The central principle of release planning is the *capacity constraint*: a release plan can only be feasible if the requirement for effort to put features into the release equals (or is less than) the available capacity. A plan satisfying the capacity constraint ensures that there is a fair chance of success. A release not planned in such a way that the constraint is satisfied will lead to what Edward Yourdon terms a “death march” [6], something not uncommon in the software industry.

After a release has been concluded we will find (in a *post-mortem*) that the requirement to do work (the total effort expended implementing all features, broken out by feature) was equal to the capacity to do work (the total effort expended by all those working on the release, broken out by worker). The justification is straightforward bookkeeping: each quantum of effort spent by a worker is counted both towards the worker and the feature they worked on.

The issue in release planning is to estimate the total effort requirement for all suggested in-plan features (in appropriate units: for example, person-days), and to estimate the total worker capacity available within the planned dates (in matching units), and to ensure they are equal within some confidence level.

Estimation must be done for the initial planning, and must be re-done on a regular basis (for instance, weekly) until the release is completed. Anytime the release plan slips out of balance past some company-mandated confidence threshold, action must be taken to correct the imbalance, either by moving dates, dropping features, adding resource, or any combination thereof.

In this paper we do not propose new techniques for estimation. We rely upon whatever is situationally appropriate and is currently considered state-of-the-art [7] [8] [9] [10], and build a management framework aimed at getting the estimates as early as possible, updating the estimates as soon as new information is available, and providing the necessary information for on-the-fly re-planning.

As it is typically not the implementation arms of the company that make the decisions regarding next release dates and content, but rather the business arms as represented by a product marketing/management function, we seek a release planning method that is primarily aimed at the latter. This has two implications. First, it means that the units of work that can be traded off one against another in a release plan should be stated in business terms: features of benefit to customers; and not in development terms: tasks that need to be executed to implement the features. Second, it means that product marketing must be enabled to consider many different feasible alternative release plans with minimum involvement from the implementation arms, where a feasible alternative is defined as one that satisfies the capacity constraint.

## 4. A Mathematical Model as a Basis for Release Planning

It is intractable to consider every detail of the implementation to decide if a release is feasible. We must devise a model that states what should be considered and what should be ignored. The model must be sufficiently accurate for it to be likely that a feasible release plan can be implemented, and sufficiently lightweight and easy to manipulate that it can accommodate rapid “what-if” scenario planning on-the-fly.

Ironically, aiding us is the relative inaccuracy of data upon which the model will be built. We will need to estimate in advance resourcing requirement and capacity. The potential estimation errors early in the release are large [11]. Having a release planning model that is more accurate than the estimation errors justify is not useful.

To define a model we abstract the software process into a few key metrics that in application we will estimate in advance. For the purposes of definition, we state the metrics in an operational manner: how will we measure them. This ensures that we know the definition of what it is we are estimating, and also provides a basis for collecting actuals so that they can be compared against estimates both as software development works their way through a release, and for when they conduct post-mortems to prepare themselves for upcoming releases.

Depending upon the situation, the best model will differ, however the general approach will still apply. This section describes a model the author has successfully used as the head of software development at a software vendor (approximately 400 employees globally) producing packaged enterprise software for large investment banks.

The model starts by defining an abstracted release cycle, continues by identifying the independent phases and human resourcing that will form the basis for planning, and then states the capacity constraint in terms of these independent phases and resources. To satisfy the capacity constraint across all phases and resourcing, other phase lengths and resourcing must then be sized relative to their independent counterparts.

## Step 1: Document the Abstracted Release Cycle

As the first step we define the release cycle for new feature releases. This may be an abstracted release cycle as the details are not important. The release cycle below is a common one, but not universal. As discussed previously, it must be adapted to the situation.

- **Phase 1: specification and design**
  The requirements for each feature are elaborated. Medium-level software designs are worked out. Prototyping may occur during this phase.
- **Phase 2: coding, unit testing, and ongoing integration testing**
  Features are coded, unit tested, and continuously integrated. Coding starts at a point called *fork*, which is the point at which the source code is diverged from that of the previous release. This phase ends at *dcut*, development cutoff, the point at which no developer knows of any more code that needs to be written for the release to be feature complete (also known as *code complete*). It is typical that there is some residual work from the previous phase done during this one.
- **Phase 3: alpha test**
  The software is handed off to a QA team that executes a test plan. Developers are available to correct defects as they are found. This phase ends with beta release.
- **Phase 4: beta test**
  The system test plan continues to be carried out and problem reports are collected from the beta testers. Developers remain available to correct defects. This phase ends with the generally available release of the product.

The beta test phase of the previous release overlaps the specification and design phase of the next release in order to smooth the resource requirement placed on the documentation, testing, and development staff assigned to the product.

## Step 2: Identify Dependent Phases and Resourcing and State Ratios

Once the release cycle is defined, we simplify the model by assuming that the lengths of certain phases (*dependent phases*) are dependent upon the length of other phases (*independent phases*). We also assume the number of human resources with a particular skill required (the *dependent resources*) are dependent upon the number of other resources (the *independent resources*). This simplification allows us to concentrate our planning efforts on a subset of the activities and resources.

For ease of later estimation, the more concrete activities should define the independent phases and resources: those that we vary during planning. Usually coding is the most concrete activity. To implement the desired feature set, the appropriate code must get written. On the other hand, how much testing, documentation, requirements, prototyping and design is enough? These are more flexible, and can more easily be made to fit supplied time windows and resourcing (assuming a certain minimum standard is met)

In the ongoing example, the independent phase will be the coding phase and the independent resource will be the coding resource. We will deal explicitly with the capacity constraint only for the independent phase and resources. We will deal indirectly with dependent phases and resourcing by insisting that they be sized relative to the length of the coding phase and the number of coders. In particular, for the example model, we use the following ratios.

- length of specification and design phase = length of coding phase
- combined length of test phases = length of coding phase
- length of beta test phase = twice the length of the alpha test phase
- number of coders throughout = same as in coding phase
- number of dedicated testers throughout = one third the number of coders
- number of dedicated documenters = one quarter the number of coders.

To test if a release plan is feasible, we need to first decide on the amount of time spent on the coding phase, and the number of coders to use. Based on this information, our plan must have these particular coders available throughout the release, must have the appropriate number of testers and documenters available throughout the release, and must allow for the appropriate length for the other phases as determined by the ratios above. If the planned dates or resourcing fall short, the release plan is considered infeasible (not satisfying the capacity constraint).

If it turns out the dependent work will not fit given the desired level of quality, or there is more than enough time and resourcing, the ratios should be adjusted when planning the next feature release.

## Step 3: Plan the Independent Phase

We now turn out attention to the planning of the independent phase (coding) using the independent human resources (those contributing to coding the release during the coding phase).

An important consideration is the units of coding effort to be used. Similar in spirit to the work of Boehm in COCOMO for defining effort [9, p.29], we define a *dedicated coder-day* to be the equivalent of eight uninterrupted hours spent by a coder during the coding phase working on nothing other than coding (and unit testing) new features into the release in question. In practice a productive coder in a good working environment may net only about 0.6 or so dedicated coder-days per workday. Using these units as a basis, we state the capacity constraint mathematically as follows.

$$F = N \times T \qquad \textbf{Eq. 1}$$

- $F$ is the total requirement for coding effort during the coding phase (measured in dedicated coder-days),
- $T$ is the length of the coding phase expressed in workdays (excludes weekends and holidays),
- $N$ is the average number of dedicated coders available throughout the coding phase.

The total requirement for coding resource during the coding phase, $F$, is measured by the sum of the effort for each in-plan feature:

$$F = \sum_{i=1}^{k} f_i \qquad \textbf{Eq. 2}$$

We indicate in the release plan cases where the effort sizing for one feature is dependent upon another feature being in-plan. Product managers are thereby made aware that if they remove certain features from the plan, it may increase the effort required for dependent features. Feature effort is given in units of effective coder-days for a particular coder.

To measure each $f_i$ requires that at the end of each working day every coder records how many dedicated hour equivalents they spent working on each of their assigned features during that day. The coder must make an estimate as to how distractions affected his or her productivity. For example, and hour spent uninterrupted is not equivalent to one hour spent over the course of two being frequently interrupted [12].

$N$ is the average number of dedicated coders available for each day of the coding phase. To measure this we require:

- $v_j$ – the number of vacation days the $j^{th}$ coder took during the $T$-day coding phase.

- $w_j$ – the *work factor* for the $j^{th}$ coder: the ratio of effective days to workdays.

$N$ is then the sum of the ratios of effective days to working days for each coder:

$$N = \frac{\sum_{j=1}^{n} w_j \times (T - v_j)}{T} \qquad \textbf{Eq. 3}$$

The number of vacation days should constitute only official vacation days, not days or parts of days taken off the coder planned to make-up, or days taken off in lieu of working overtime prior to that point in time. Vacation time is paid time off the developer had no intention of "making up" in any way.

The *work factor* is the average number of dedicated days per workday for a given coder. It is measured by totaling the number of dedicated hours spent throughout the coding phase by that worker, dividing by eight to convert to days, and then dividing by the number of non-vacation workdays during the coding phase to arrive at the ratio. To measure this, we require that the $j^{th}$ developer record the number of dedicated hours spent during the 24-hour period of the $d^{th}$ working day: $h_{j,d}$

$$w_j = \frac{\sum_{d=1}^{T} h_{j,d}}{8 \cdot (T - v_j)} \qquad \textbf{Eq. 4}$$

We can gather both the time spent working on a given feature, and the time spent on any given day, by implementing a time tracking system capable of gathering the quantity $h_{j,d,i}$ which is the number of dedicated hours spent by the $j^{th}$ developer on the $d^{th}$ day working on the $i^{th}$ feature. This quantum of dedicated effort winds up on both sides of Eq.1, constraining them to be equal under the formulation given (which is required of any release planning model that is intended to satisfy the capacity constraint). This equality may be demonstrated by expressing both sides of Eq.1 in terms of $h_{j,d,i}$ using Eqs.2-4, simplifying, and noting the equality.

## 5. Using the Model for Release Planning

Release planning involves estimating the various quantities in the formulation given above, and then determining if, within a given confidence level, the capacity constraint is satisfied. In particular, the input stochastic variables for the model given above are as follows.

- $\tilde{f}_i$ – estimate of the coding effort required for each feature.
- $\tilde{w}_j$ – estimate of the work factor for each coder.
- $\tilde{v}_j$ – estimate of the vacation to be taken during the coding phase for each coder.

In the ideal case we would be provided with a probability distribution for each of these stochastic variables. We could then combine the distributions to come up with a composite distribution, *delta*, parameterized by the number of workdays in the coding phase, defined as follows.

$$\tilde{D}(T) = \tilde{N} \times T - \tilde{F} \qquad \textbf{Eq. 5}$$

When delta is positive (a "good" thing), capacity exceeds requirement. When delta is negative (a "bad" thing), the requirement is greater than the available capacity.

Company policy must dictate a certain confidence interval, $c$, at which the company is willing to plan, and make a release plan (choosing resourcing, release date, and feature content) such that the probability that we have sufficient capacity to get the coding done on time with the given resourcing is just $c$.

$$\Pr[\tilde{D}(T) \geq 0] = c \qquad \textbf{Eq. 6}$$

In this example we do not consider the number of coders available to be a variable for the purposes of next release planning. In the typical software vendor situation the useful coders are those that are familiar with the code. In bringing on new coders for the next release, we hope only that they contribute as much as they consume in training time. Thus changing the number of coding resources is a viable planning decision only across a multiple-release planning horizon.

The management framework does not constrain the manner in which feature effort estimates should be arrived at. The development organization must supply 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 [13]).
- Which coders will work on the feature.
- The productivity of those coders working on that feature (in LOC 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.

The majority of previous work on effort estimation has been done in the context of estimating the effort required to complete a new software project from scratch [9] [14]. However, there has been some recent work on estimating maintenance tasks [7] [9] [10], and when combined with methods for estimating a coder's individual productivity [8] we may arrive at a feature effort estimate.

In practice, any more statistically oriented approach to effort estimation will be an adjunct to expert-based estimation [10]. In the case of enhancive maintenance where the architecture is well-understood, the code is well-understood by the coders, and the coders are well known to the experts, it has been the author's experience that direct, expert-based prediction of effort in dedicated coder-days has been fairly reliable. This issue should be investigated in future research.

Pre-requisite to accurate estimation and effective measurement are well-founded and well-understood definitions of the quantities being estimated. The mathematical framework helps ensure this. Another consideration not often discussed is the stochastic nature of an estimate. For example, assume we are given an effort estimate of 20 effective coder-days for a feature effort estimate. Is this a pessimistic estimate, an optimistic estimate, or an average-case estimate? If the 20 days is an average-case estimate, then what is the 90% worst case? Is it closer to 22 days (low variance) or to 35 days (high variance). Is the 10% best case the same distance from the mean as the 90% worst case (i.e., what is the *skewness* of the distribution).

When we get two numbers for an estimate, say 18 effective coder-days as an average case and 23 effective coder-days as a 10% worst case, we can fit the estimate to a standard distribution (for example, Normal if appropriate) and have a better-defined stochastic estimate than just one number alone can provide.

## 6. The Release Plan Document

The mathematical, metrics-based model presented above forms the basis for the release plan document. This document is typically maintained on an internal Web site, one per product (or per major resource pool within a product effort). Space precludes inclusion of a complete sample release plan, however the most important sub-sections are summarized below. A complete sample release plan may be found on the author's Web site [15].

The release plan takes the form of a balance sheet. On one side are the assets (the capacity to do work), on the other are the liabilities (the requirement to do work). A summary section subtracts the requirement from the capacity to determine if the release is in good shape or if the release dates and/or feature set are in jeopardy.

## Current Status Summary Section

A release plan starts with the product name, the new feature release designator (for instance, *Release 4.3*), the planned start/end dates for the various phases (or actual dates if any are in the past), the number of working days in each phase (to assess conformance to the ratios discussed above), and an overall mission statement for the release.

Near the top of the plan is the current status summary. In the example given below, the release plan reflects a status as at 10 days into a 74 day coding phase for a release. It summarizes the capacity to do work expressed both as the total number of effective days available, *"Capacity"*, and as the average number of dedicated developers available per day remaining in the coding cycle, *"Average coders"*. The total requirement for effort is given, *"Requirement"*, also in effective coder-days. *"Delta"* expresses the amount of excess (>0) or shortfall (<0).

| ***SUMMARY*** | ***mean*** | ***sdev*** | ***units*** |
|---|---|---|---|
| *Capacity* | 394 | 28 | effective coder-days |
| *Remaining coding days* | 64 | | working days |
| *Average coders* | 6.8 | 0.5 | effective coders per working day |
| *Requirement* | 400 | 16 | effective coder-days |
| *Delta* | -6 | 32 | effective coder-days |

As part of the summary may also come an indication of the implications of these numbers on the dates assuming the current plan.

| *Confidence* | **43%** | 50% | 80% | 95% | 99% |
|---|---|---|---|---|---|
| *Dcut slip (workdays)* | on-time | -1 | -5 | -9 | -12 |
| *Projected GA* | Jun.1 | Jun.5 | Jun.15 | Jun.17 | Jul.5 |

The *"Dcut slip"* represents the numbers of workdays by which the development cutoff date (the end of the coding phase) will be delayed. The *"Projected GA"* is the projected date for a generally available release given the slip in dcut. This is computed by assuming that if the coding phase slips then the assumed ratio holds true for the testing phase and so it will slip as well. Combining the two slips then counting workdays on a calendar gives the projected GA.

In this example, there is only a 43% chance of delivering on or before the originally planned Jun.1 date. However, there is an 80% chance of being no later than Jun.15 for the generally available release. These projections are based on the delta statistic from the table above.

The reason for the focus on dcut rather than GA is because after dcut pro-active re-planning is no longer possible. By dcut, all the code is written and all the defects have been injected. After dcut, all the organization can do is to monitor the defect arrival rates to ensure the release remains on track for its end dates. The only planning alternatives are shipping late or shipping with unacceptably high defect arrival rates.

The ratio-oriented nature of dependent phase planning helps organizations avoid the pitfall of delaying dcut but holding firm on the GA date thereby reducing the testing time. This is tantamount to claiming that if coding has taken longer than expected, testing will take less time than expected. In fact, the opposite must be assumed as is captured in our framework. The practical result is that the dcut date becomes the focus of management and of developer attention: all the "panic" occurs at dcut time. The GA is by comparison anti-climactic.

For this example release, product management will need to take action to re-balance the plan. If, for example, the software company is comfortable with an 80% chance of hitting the dates, and they wish for the GA date of Jun.1 to hold firm, they will have to cut 5 workdays from the coding plan, equivalent to approximately 34 effective coder-days (the conversion rate is 6.8 effective coders per working day, as shown above). This means that product management will have to go through the plan, look at the items that (preferably) have not yet been started, and cut a total of 34 effective coder-days worth of features from the plan.

## Requirements Section

The requirements section of the report gives details regarding how the total requirement for dedicated coder-days is broken out into features. This table contains sufficient information for a product manager to make a first-cut decision as to what features to drop in order to re-balance the plan.

| ***id*** | ***description*** | ***status*** | ***remaining effort*** | |
|---|---|---|---|---|
| | | | ***mean*** | ***sdev*** |
| 345 | Show angular separation | code complete | 0 | 0 |
| 123 | Field-of-view indicators | in progress | 18 | 3 |
| 43 | Orbit display | not yet started | 22 | 5 |
| 265 | Change location | not yet started | 12 | 2 |
| | | **…** | | |
| | | ***total requirement:*** | 400 | 16 |

For example, if the product manager must remove 34 effective coder-days from the plan, removing features 43 and 265 would suffice.

In practice, the requirements section would have more detail, such as pre-requisite relationships, priority information, any customers to whom the feature was promised, initial effort estimate, number of days to-date spent on the feature, and hyper-linked quality assurance information. For instance, whether specification and design documents have been completed for the feature and have been reviewed. This information as important, as the variance in the effort estimates is likely highly-dependent upon whether or not good specifications and designs are available [11] [16].

## Capacity Section

The capacity section gives details regarding how the total capacity was arrived at. There is a line for each person who will contribute to coding the release during the coding phase. It lists the remaining number of workdays available (the same for everybody), the expected amount of vacation to be taken (given as a "deterministic" estimate in this case) , and the work factor for converting workdays into effective days expressed as a Normal estimate. The effective days available for each worker are then computed, and the total is (stochastically) summed.

| ***coder*** | ***class*** | ***workdays remaining*** | ***vacation (workdays)*** | ***work factor*** | | ***effective days remaining*** | |
|---|---|---|---|---|---|---|---|
| | | | | ***mean*** | ***sdev*** | ***mean*** | ***sdev*** |
| Tracy | Manager | 64 | 4 | 0.1 | 0.05 | 6.0 | 3.0 |
| Sam | Architect | 64 | 5 | 0.4 | 0.1 | 23.6 | 5.9 |
| Shakur | Coder | 64 | 0 | 0.7 | 0.3 | 44.8 | 19.2 |
| Britney | Coder | 64 | 7 | 0.6 | 0.2 | 34.2 | 11.4 |
| | | | **…** | | | | |
| | | | ***totals:*** | 6.8 | 0.5 | 394 | 28 |

In practice, the reasons for work factors less than one involve distractions and other work (such as corrective maintenance on previous releases), leaving only a fraction of the day available for concentrated work on coding new features into the current release. For certain classes of resources. Such as managers or lead architects, the actual amount of coding time they can devote is usually low, even though they may be concentrated 100% on the effort.

This section of the report helps those reading the plan to understand why the capacity numbers are what they are. This aspect of traceability in the report demonstrates the rational process by which a capacity estimate was arrived at, cutting short a potentially non-productive discussion regarding the issue of how much capacity is "truly" available to work on the release.

## 7. Final remarks

Planning the availability dates and feature content of new releases of existing software products is a critical activity for any software vendor organization. In this paper we presented a high-level approach to release planning founded on a mathematical, metrics-based model of software development that helps us to define the precise units of quantities we are estimating. The example model formulation is one case in a more generic approach that involves making a model appropriate to the situation at hand. All such models must provably satisfy the capacity constraint, which insists that capacity to do work and requirement to do work are in balance.

The metrics-based approach encourages the use of tools to facilitate the gathering of actual versus estimated information. The main tool is a fine-grained time-tracking system that understands about features in-plan for a given release and to which developer they are assigned. Such information can be used to refine estimation on future release cycles, and can be used to determine appropriate standard deviations for average case estimates even when developers may not be aware of this information directly themselves.

The framework remains effective even when estimates near the start of a release cycle are inaccurate. Once requirements are fully documented and a design worked out, certainty increases. Once coding begins on a feature, certainty in remaining effort estimates increase again. The release plan is a place in which the increasingly certain estimates can be recorded and aggregated as they become available. This gives the business increased visibility into release status, and hence more time to respond to shifting estimates. It also provides constantly up-to-date information as a basis on which to consider plan changes necessitated by shifting business priorities.

This framework also suggests a non-traditional approach to future estimation research. In addition to being concerned with the accuracy of estimates, we should also be concerned with the rate at which they converge to accuracy during the release cycle.

---

1. Humphrey WS, Managing the Software Process, SEI Series in Soft. Eng., Addison-Wesley, 1989
2. Boehm B, Software Engineering Economics, Prentice-Hall, 1981
3. Chapin N, Software Maintenance Types – A Fresh View, Proc. of the IEEE Int't. Conf. on Software Maintenance, 2000 pp.247-252
4. Pigoski TM, Practical Software Maintenance, Wiley, 1997
5. Gause DC, Weinberg GM, Exploring Requirements: Quality Before Design, 1st ed. New York: Dorset House Publishing, 1989.
6. Yourdon E, Death March, Prentice-Hall PTR, 1997
7. Ramil, JF, Lehman MM, Metrics of Software Evolution as Effort Predictors – A Case Study, Proc. of the IEEE Int't. Conf. on Software Maintenance, 2000 pp.163-172
8. Humphrey WS, Predicting (Individual) Software Productivity, IEEE Trans. on Soft. Eng., Vol.17, No.2, Feb. 1991, pp.196-207
9. Boehm BW, *et. al.*, Software Cost Estimation with COCOMO II, Prentice-Hall PTR, Upper Saddle River, NJ, 2000.
10. Jørgensen M, Experience With the Accuracy of Software Maintenance Task Effort Prediction Models, IEEE Trans. on Soft. Eng., Vol.21 No.8, 1995, pp.674-681
11. Low GC, Jeffrey RD, Function Points in the Estimation and Evaluation of the Software Process, IEEE Trans. on Soft. Eng., Vol. 16-1, pp.64-71, Jan. 1990.
12. DeMarco T, Lister T, Peopleware: Productive Projects and Teams, Dorset House, 1987
13. Albrecht AJ, Gaffney JE, Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation, IEEE Trans. on Soft. Eng., Vol SE-9, No.6, June 1983, pp.639-648
14. Jones TC, Estimating Software Costs, McGraw-Hill, New York, NY, 1998
15. Penny, DA. http://www.cs.toronto.edu/~penny/research/release-planning/samplerp.html
16. Wang AS, Dunsmore HE, Early Software Size Estimation, Proc. of the 3rd Symp. on Empirical Foundations of Information and Software Sciences, Oct. 1985