Many organizations use cycle time as a measure of process capability. Cycle time is the time it takes an organization to produce a new software version. It is usually applied to organizations that are producing a series of product releases or versions. The "cycle" begins with the receipt of the customer's request (e.g., a software trouble report) and ends with the successful completion of product acceptance test. The time to produce a software release clearly depends on the amount of availability of the engineers who perform the work.

This paper discusses how we might define a truly general definition of cycle time suitable for comparing organizations. What we desire are analogies to the Cost Performance Index and Schedule Performance index used in earned value tracking. We will briefly describe the total software product lifecycle. This encompasses incremental, phased, and evolutionary development. Next, we will identify the milestones (process anchors) that define the end points of a "cycle". Then we will discuss possible definitions of "cycle time" and the factors that affect the computed value. This discussion provides insight into ways to normalize the actual cycle time by the product size and the average staff level to obtain a standardized measure. We find that when completely normalized, cycle time simply reduces to productivity. We also describe an alternate definition that is useful for organizations interested in time to market.
1.0 INTRODUCTION

Many organizations use cycle time as a measure of process capability. Cycle time is the time it takes an organization to produce a new software version. It is usually applied to organizations that are producing a series of product releases or versions. The "cycle" begins with the receipt of the customer's request (e.g., a software trouble report) and ends with the successful completion of product acceptance test. The time to produce a software release clearly depends on factors such as the availability of the engineers who perform the work. This paper discusses some of the factors that affect the computation of "cycle time" and provides insight into possible definitions that may be useful in practice.

1.1 Preconditions

The word "cycle" denotes a repeating sequence of activities. In order to define a cycle time, therefore, we need to have a production process which produces a set of equivalent products. Most often, we think of cycle time in the context of producing a series of releases. In some cases, however, the term cycle time may be appropriate for production processes which develop similar versions of a product which are members of a common product line. This second case is essentially the production of a set of instances or instantiations from a common pattern or template. This paper will focus on the more usual case, namely the production of a series of releases.

To define a meaningful cycle time the production process used to produce each release must be the same. In the vernacular of statistical process control, this means that the process is stable. Essentially, stability means that the process is well defined and operating within the expected control limits defined on the basis of statistical measurements of the process. For more information see [Florac, 1999].

1.2 Total Software Life Cycle

Figure 1-1 shows the total software life cycle. The usual software product life cycle covers concept definition, development of the initial release, subsequent sustainment, and, in some cases decommissioning. The activities of these last three are shown in the figure. Since the sustainment cycle is repeated over and over, it usually consumes the bulk of the total effort expended during the life of a software product. Initial development is the line along the bottom of the chart starting with requirements analysis and completing with system operation and ultimately close-out. During the initial creation of the system, the engineers and designers define the operational concept for the system, the architecture, and the top level product design. These activities correspond to two of the "process anchors" defined by Boehm [Boehm, 1996]. Specifically, the anchors are life cycle concept of operations (LCO), life cycle architecture (LCA), and initial operational capability (IOC). During sustainment, the staff corrects and enhances the product, producing a series of releases according to some specified schedule. The activities are represented by the loop above the horizontal line. The activities begin with screening problems reported by the users, and then proceeding to analyze these problems, plus any new requirements from the system's owner, and
finally making the necessary changes to the code. Sustainers of the system make modifications for four purposes: correct, perfect, adapt, and enhance. For more details see [Boehm, 1981, page 536].

**THE TOTAL SOFTWARE LIFE CYCLE**

Although several boxes in figure 1-1 have the same names in the sustainment loop as in the initial development sequence, the nature of these activities is actually somewhat different. For example, the requirements analysis activity in sustainment involves not only analyzing requirements provided by the user or customer in the form Software Change Requests (SCRs), it also includes analysis of Software Trouble Reports (STRs). Compared to new development, software sustainment expends a greater fraction of the total effort performing requirements analysis. One reason for this increase is that the analyst must assess the impact of the proposed change on all modules of the existing system. A second reason is that sustainment is not usually performed by the original designer, and so additional time is needed to understand the structure of the existing system. Since the software architecture is defined and the size of existing modules are known, less effort is spent on design. One of the main assumptions of sustainment is that the product structure remains unchanged. Of course, if the required modifications are extensive it may be necessary to change the structure of the system. If the architecture of the product must be changed, then the cyclic production process is halted and the team must return to the initial creation stage and repeat the standard activities of requirements analysis and product design.

Some systems have a decommissioning activity at the end of their useful life. In most cases, the users of the product merely shut off the system and discard the product.
In some cases, however, it is necessary to replace the existing product with some newer version. In this case there may be some activities involved with cutting over the old system and its associated databases to the new system. This is a totally different estimation problem and of course is very situation dependent. There is no cyclic activity in this final stage of the product's life.

The end points which make sense for the definition of a "cycle" are as follows. The cycle should start when the developers receive a set of validated and approved change requests. As mentioned before, these can deal with correction, perfection, adaptation, and enhancement of the product. This must be the starting point for the process because only when the change requests have been validated and approved can we define the build content. This is necessary in order to determine the amount of code and the effort associated with producing the new release of the system. (As we will see later, size is an important contributor to the cycle time.)

The end of the cycle is defined as the point when all acceptance tests have been successfully completed. It is at this point that the team has produced a complete package of technical data which includes code, databases, and associated documentation. At this point, depending on the desires of the customer, the baseline of the package of data is then distributed via various means. These could include mailing out new versions of the system in the case of a shrinkwrapped product. It could include actual travel to customer sites to install and check out the new version. These details are situation-specific and do not belong in the calculation of a general purpose cycle time.

2.0 COMPONENTS OF CYCLE TIME

This section discusses the various components which make up the cycle time and serves to identify factors which must be considered in calculating any meaningful numbers.

2.1 Measures of Time

The following equation shows that the apparent time between releases of a product consist of the sum of the development and non-development time:

\[
TTBR = TDEV + TNONDEV \quad \text{[calendar-days]}
\]

where

- \( TTBR \) = Total Time Between Releases
- \( TDEV \) = Time to develop the product
- \( TNONDEV \) = Time when no production is occurring

The time between releases is measured in calendar days. Note that we have distinguished time when the team is actually producing software from time when no production is occurring. This non-development time could be due to the analysis of the change requests and trouble reports, planning, deployment, and slack. We want to remove it from consideration because it varies widely between releases and organizations. For example, the analysis of change requests depends on the number of users, the number of
sites, etc. The presence of the non-development time is the first indication that there are various types of anomalous conditions which must be accounted for and removed if one is to come up with a general definition for cycle time. Another important variable which should be considered is the fact that different organizations operate on different work schedules.

We can further decompose $T_{DEV}$ as:

$$T_{DEV} = \text{work days} + \text{weekends} + \text{holidays} + \text{down time}$$

The apparent development time, $T_{DEV}$, is measured in calendar days and equals the sum of the number of work days, the weekends, holidays, and "down time". Down time could be due to failures of the development equipment during the production process or lack of funding (stop work). The factors relating to the failure of equipment and lack of funds are what are usually called "special causes" in the language of statistical process control. (For more details see [Florac, 1999].)

2.2 Duration and Effort Delivery Rate

The best candidate for a project independent measure of development time is the actual time spent working. We call this time the "duration" and define it as:

$$TDUR = \frac{\text{Effort Required}}{\text{Effort Delivery Rate}} = \frac{E}{R} \quad \text{[work-days]}$$

where

- $E = \text{Effort Required} = E_{\text{build}} + E_{\text{inspect}} + E_{\text{test}} + E_{\text{rework}}$
- $R = \text{Effort Delivery Rate} = (N_{\text{workers}})(AA)(8 \text{ phrs/work-day})$
- $AA = \text{Average Availability (ranges from 0 to } 2)$

Basically, the duration is defined as those days when the project staff delivers effort to perform the project tasks. We measure duration in units of work-days. Clearly, the apparent schedule ($T_{DEV}$) has to be longer than the duration ($TDUR$). A typical ratio if we subtract out holidays and weekends shows that the ratio of duration to schedule is approximately two-thirds.

The effort required consists of the direct charged effort to perform the building, inspections, testing and rework for the production process. For some organizations, the effort also includes the effort required for support activities such as configuration management and quality assurance. For other organizations, these activities are considered indirect costs and are covered out of overhead. The choice of course affects the apparent amount of effort which must be delivered by the project team. Does the team include CM and QA or not? This must be clearly stated in order to have meaningful definitions. (This also will affect productivity as we will soon see.)

The effort delivery rate is defined as the numbers of person-hours of labor which are delivered to the project tasks per work day. The effort delivery rate (denoted by $R$) is
defined as $N_{\text{workers}} \times AA \times 8 \text{ phrsl/worker-day}$ where AA denotes the average availability of the workers. The value of AA ranges from as low as 0 up to probably 2 or so. If AA is greater than 1 then on the average the staff is working overtime. The number of full time equivalent (FTE) workers is simply equal to $AA \times N_{\text{workers}}$.

Note that there is actually a coupling between the value of AA and the units of time. Some authors choose to define effort delivery per calendar month. I believe that this obscures the effects of the different work calendars, seasonal variations and any outages. For this reason, I prefer to represent these calculations in terms of work-days instead of person-months and calendar-months.

The effort to perform the analysis, design, code, and test activities needed to produce a release is often estimated in terms of some measure of equivalent size divided by a productivity: $EE = \text{Size}/\text{Productivity} = S/P$. The size can be measured in Source Lines of Code (SLOC), Function Points, etc. The productivity, $P$, is measured in compatible units such as SLOC/person-hour, FP/person-hour, etc. The definition of productivity is somewhat slippery and consists of the following factors:

- **Size** = Amount of product produced. For software, this is usually source lines of code or Function Points.
- **Effort** = Direct labor expended by the project staff to build the product. Typically measured in person-months or, better, person-hours.
- **Productivity** = Size divided by effort.

There are a number of factors which affect the productivity values. These deal with size and with effort. The factors affecting size are:

- Product architecture (choice of platform, technology, and decomposition)
- Amount of reuse
- The software being counted (reused, prototypes, tooling, breakage)
- The choice of size unit of size (blank lines, comments, physical or logical lines)

The factors affecting effort are:

- Project formality ("rigor")
- Process architecture (life cycle)
- The activities and phases included (CM & QA direct/indirect; P&R)
- Degree of automation
- Staff skill and experience
- The choice of units (person-hours per person-month)
- Scope (paid overtime, uncompensated overtime)

Any meaningful definition of cycle time must be based on precise definitions for the activities and tasks included in calculating the effort, as well as the issue of direct versus indirect charges for various types of labor as I have already mentioned.
2.4 Summary of Factors

The following factors affect the computed value for cycle time:

- Choice of cycle's end points
- Product size
- Productivity of process (automation, rework)
- Staffing level (number and availability)
- Work calendar (weekends, holidays, planned shutdowns)
- Outages (unplanned shutdowns)

We have discussed most of these factors already. Outages, however, require some additional explanation. We consider outages to be unplanned shutdowns when production can not occur. In some cases, the staff may continue to charge the project even though they are accomplishing no useful work. The effort associated with such anomalous events needs to be subtracted out in order to obtain a standardized measure of cycle time. Even if the staff does not bill effort to the production activities during unplanned shutdowns, there still will be some additional effort consumed as the staff returns to the project and must restart their activities. (There is "friction" associated with restarting the work and reestablishing the mental context.) Whether or not outages cause extra effort to be consumed affects how we choose to define TDEV and TDUR. This affects the form of the equations. My equations assume that outages do not affect the total effort. That is, the managers quickly reassign the staff to do other things. (This still does not eliminate the problem associated with the friction of restarting a team after an outage.) Some organizations may, however, accept some level of outages as inevitable, and may include a "pad" when estimating the effort, E, and will record the unproductive effort for the purpose of computing the productivity, P. For example, Cusumano and Selby describe a 20-30% "schedule buffer" used at Microsoft [Cusumano, 1995, pages 204-206].

3.0 QUANTITATIVE DEFINITION

This section presents a working definition for cycle time and then discusses how this value could be normalized to permit comparison between various projects. Then we discuss some interesting consequences of the normalization.

3.1 A Working Definition

We believe that duration is the most general quantity for representing cycle time. Using the notation introduced above, we can define:

$$TDUR = \frac{S/P}{Nw*AA*8}$$

This equation shows the factors which determine the duration. The units are of this equation reduce to work-days as expected:
(SLOC)/(SLOC/phrase)\(\text{work-day}\) = work-day

We can reduce TDUR by adding staff and in principle could even reduce the duration (cycle time) to 0 by adding large numbers of workers. In practice, however, there are natural constraints to the amount of schedule compression that can be achieved. This is analogous to the situation of attempting to assign nine women the task of producing one baby in one month. Other relevant issues here are limits to schedule compression (e.g., see [Boehm, 1981, pages 466-472]) and constraints associated with adding staff to a project which is already in progress, i.e., Brooks’ Law. For a mathematical form of Brooks’ Law see [Stutzke, 1994].

3.2 Normalized Cycle Time

In order to obtain a standardized measure to compare different projects we would like to normalize the cycle time. The following equation illustrates how we can normalize TDUR for the effects of the number of workers, their availability, and the size of product to be produced:

\[
\text{NCT} = \text{Normalized Cycle Time} = \frac{TDUR \times N_w \times AA}{S} = \frac{1}{8P}
\]

The use of duration as a measure of cycle time is desirable because duration is independent of weekends, holidays, outage, etc.

Alternately, we can express the same thing using COCOMO’s notation:

\[
\text{NCT} \propto \frac{(TDEV)(FTE)}{S} = \frac{MM}{S} = \frac{1}{P}
\]

since \(TDEV = k \times TDUR\)

and \(FTE = N_w \times AA = MM/TDEV\)

The alternate calculation (shown above) is another way of illustrating what is shown in the NCT equation. The conclusion is that when completely normalized, cycle time is identical to productivity. Thus, normalized cycle time is not an independent measure which characterizes a development process.

3.3 A Revised Definition

Cycle time is a valid measure for some industries where time-to-market is critical. For example, products must be delivered to meet desired market windows. Thus, some measure of the organization’s ability to deliver functionality on time is of interest. A commonly used definition is:

\[
\text{NCT} = \frac{TDUR}{S} = \frac{1}{8 \times N_w \times AA \times P} \quad \text{[work-days/SLOC]}
\]

© by R. Stutzke 1999

15SEP99
This revised definition measures the time to deliver a "unit of functionality". Planners adjust size, productivity, staffing, and work calendars to achieve the desired TDUR. Planners have a number of options which are summarized in Table 3-1. The usual goal is to keep NCT' constant while increasing the size, S. Typically, commercial developers assign more workers to the project, thereby increasing N_w. Note that this may cause the productivity, P, to decrease due to the diseconomy of scale associated with the larger staff size. (Cost is not the primary or even the secondary consideration for producers of mass market software products.) We note in passing that more mature organizations with well-defined production processes can increase the number of workers with less loss of productivity than less mature organizations. Usually, the use of larger staffs is planned from the start. Options for recovering or shortening schedule are discussed in [Stutzke, 1993] and [Stutzke, 1994].

<table>
<thead>
<tr>
<th>TABLE 3-1: WAYS TO REDUCE CYCLE TIME*</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Eliminating tasks (reducing size, automating some operations)</td>
</tr>
<tr>
<td>2. Reducing time per task (productivity, overlapping)</td>
</tr>
<tr>
<td>3. Avoiding single-point task failures (hardware outages, loss of key people)</td>
</tr>
<tr>
<td>4. Reducing rework (&quot;backtracking&quot;)</td>
</tr>
<tr>
<td>5. Activity network streamlining (reduce task dependencies)</td>
</tr>
<tr>
<td>6. Increasing the duration (number of workdays)</td>
</tr>
<tr>
<td>7. Better people and incentives</td>
</tr>
<tr>
<td>8. Transition to learning organization (process improvement)</td>
</tr>
</tbody>
</table>

*Adapted from [Boehm, 1997]
References


