NORTH-HOLLAND

Controversy Corner

It is the intention of the *Journal of Systems and Software* to publish, from time to time, articles cut from a different mold. This is one in that series.

The object of the CONTROVERSY CORNER articles is both to present information and to stimulate thought. Topics chosen for this coverage are not just traditional formal discussions of research work, but also contain ideas at the fringes of the field's "conventional wisdom."

This series will succeed only to the extent that it stimulates not just thought, but action. If you have a strong reaction to the article that follows, either positive or negative, write to Robert L. Glass, Editor, *Journal of Systems and Software*, Computing Trends, 1416 Sare Road, Bloomington IN 47401. We will publish the best of the responses as CONTROVERSY REVISITED.

CHANGE-POINTs: A Proposal for Software Productivity Measurement

Vernon V. Chatman, III

This article describes a productivity measurement approach based on a common output for design, implementation, and testing. It relies on the traditional definition of productivity: Output ÷ Input = Productivity. It presents the concept of a unit of work — an abstract work item — which reflects that, to manage a development project, we must make logical subdivisions of the work effort. The notion of CHANGE-POINTs is derived from these subdivisions. Because we need to view productivity within the context of effectiveness, the article introduces the productivity, interference, and effectiveness matrix for that purpose.

1. INTRODUCTION

In 1985, Dr. Ed Altman, then IBM General Products Division Vice-President, Software, commissioned a task force at IBM's Santa Teresa Laboratory (STL) to look into the issue of software productivity measurement. The task force leaders were Dr. Ursula Richter, who came to STL in 1984 from IBM Research to work on this problem, and myself. Dr. Altman wrote in an internal memorandum in 1984:

> The traditional measure of productivity (CSI/PY[1]) is of marginal use to management as it does not equitably measure many of development's variables. CSI/PY does not address the considerable investment in testing for large products such as IMS and DB2, nor does it fairly measure the testing effort required for products built for multiple operating environments.

Dr. Altman's remarks raise two fundamental productivity measurement issues. First, a productivity metric must address the mission of the activity to which it applies, that is, managers and performers of that activity must believe the metric relates to their responsibilities. Second, to aggregate productivity measurements across project activities, the metric must have a common "output" for the activities

Address correspondence to Vernon V. Chatman, III, 5984 Via Madero Drive, San Jose, CA 95120. E-mail address: vvchatm@IBM.net. Mr. Chatman recently retired from IBM.

[1]CSI means "new and changed executable source instructions." PY means "person years."

J. SYSTEMS SOFTWARE 1995; 31:71–91

655 Avenue of the Americas, New York, NY 10010

0164-1212/95/$9.50
SSDI 0164-1212(94)00088-5

included in its scope. CHANGE-POINTs addresses these and other important productivity measurement issues. Additionally, this article makes the point that we need to view productivity within the context of effectiveness.

2. PRELIMINARY MATTERS

Before presenting the ideas and metrics associated with CHANGE-POINTs, there are some topics that must be dealt with to put these notions in context. This section addresses those topics.

2.1 The Meaning of Productivity

Discussion of "productivity" in the literature of software metrics has strayed from the traditional formula for productivity[2] and the traditional interpretation of productivity as a period metric[3]. The paradox of lines of code (LOC) as a productivity indicator (Jones 1986, 1991) is no paradox. As implied by Arthur (1985), given the same start date, the same number of developers assigned to each task, and equal available time to work on each project, the project using the higher level language will complete in less calendar (elapsed) time.[4] Comparison of productivity ratios should deal with calendar time differences; thus, there is a fallacy in the so-called paradox. "The tendency to use 'productivity' loosely as a synonym for other concepts can create confusion and misdirect improvement efforts" (Heyel, 1982).

As Packer (1983) noted: "While outputs are usually easier to quantify than outcomes, we often care more about outcomes than outputs." "Outcomes" lead us to another concept:

> The distinction between output and outcome is mirrored in the twin concepts of efficiency and effectiveness. Efficiency refers to how well the enterprise converts its input resources into *immediate* outputs—how *productive* the organization is in doing whatever it does. Effectiveness, on the other hand, relates to how well the enterprise uses its input resources to meet its ultimate goals and purpose—how *productive* the organization is in accomplishing what it should be doing [(Packer, 1983); emphasis added].

If effectiveness is improved (e.g., reduced total development cost), then reduced efficiency (e.g., LOC per person-month) is acceptable.

The key problem with many uses of LOC per person-month as a productivity metric is that "programming development [involves]... a significant amount of... [effort] not affected by source language" (Jones, 1986); thus, LOC are not an *output* of those efforts. In addition, as in Arthur (1985), seemingly incorrect results arise from merging the concepts of efficiency and effectiveness, i.e., an expectation that productivity metrics should reflect how well an organization is accomplishing what it should be doing: e.g., "the aim of this paper is to explore a variety of measures... which have been advocated for measuring IS productivity; i.e., their efficiency and effectiveness" (Scudder and Kucic, 1991).

The CHANGE-POINT approach does not merge the concepts of efficiency and effectiveness; it defines separate metrics for each concept (and one other covered later). "Traditional formulas for measuring productivity stress efficiency and neglect effectiveness" (Packer, 1983). This keeps productivity analysis distinct from productivity measurement.

2.2 LOC in Language *x* is Not Equal to LOC in Language *x*

This section's heading does not contain a typographical error. It correctly reads "language *x*" in both cases and applies when using the *same* LOC counting convention. The point is that while some "items" are not equal, this fact may not prevent us from combining, manipulating, or comparing them; the significance of any differences must be proven. Figure 1 shows three semantically identical sets of code (in C). The first example is three LOC, the second example is one LOC, and the third example is five LOC. Clearly, there is no useful counting rule that will yield the same count for the three sets of code. Thus, even when there is semantic identity, it is difficult to argue that a LOC in language *x* equals a LOC in language *x*.[5] Inspection of programs written

[2] *Post hoc ergo propter hoc*. Assume a high-level design is produced, but no source code is produced, and the design is sold to another company. For this example, lines of code metrics would seem to preclude measurement of design effort productivity, or imply that design effort productivity is zero. In general, any "output" that is dependent on attributes of the implementation source code suffers from this problem.

[3] It is common to compare annual productivity rates. Productivity comparisons should be normalized for calendar (elapsed) time intervals.

[4] The examples in Jones (1986, 1991) bear this out. See Appendix 1 for a more detailed discussion.

[5] Some projects use multiple languages [e.g., Grady and Caswell (1987), pp. 21–22]. Adding LOC in different source languages may be more like adding US 25¢ and US 10¢ coins to make a purchase than it is like adding US dollars and British pounds to make the same purchase (however, even the latter can be done in some places of trade).

```
if ( I == 1 )
  {
  spec_code(A,B,C);
  J = 2;
  }
else
  J = 3;
```

```
J = ( I == 1 ) ? (spec_code(A,B,C),2) : 3;
```

```
switch ( I )
{
 case 1:
  spec_code(A,B,C);
  J = 2;
  break;
 default:
  J = 3;
  break;
}
```

Figure 1. Semantic identity.

in other languages will also reveal inequalities of LOC.

To repeat the point, it is acceptable to add apples and oranges if one is counting fruit. Simply observing that counted elements are different is not a conclusive objection: a valid objection requires the demonstration of the significance of any differences, within the context of the purpose being served. As a practical matter, we count, add, and manipulate unequal things all the time, e.g., planets, people, cars, planes, words, papers, books, etc.

2.3 Productivity Measurement versus Cost Estimating

Estimating the cost for a project is an important activity, but this is distinguishable from measurement of the efficiency of the project. Cost estimating involves a prognostic use of a productivity metric. In cases where estimated cost is incorrect, the error may be due to misunderstandings regarding factors affecting cost (e.g., particular design or implementation errors), and not a deficiency in the productivity metric for measuring efficiency. Thus, attempts to evaluate a productivity metric based on its use in estimating cost may not be appropriate and perhaps arise "*out of a ... misconception that a software measure must always be part of a prediction system*" (Baker et al., 1990). The situation may be that "productivity" relates primarily to the number of potatoes in Campbell's sack of potatoes (the quantity of output), and cost estimating involves also understanding whether the potatoes are "good cookers" [the qualities of the output (Campbell, 1921)]. This would support the conclusion that generally "one software development environment ... [cannot] use the algorithms developed at another environment to predict resource consumption" (Bailey and Basili, 1981).

2.4 The Meaning of Size

"Researchers in software measurement have failed to take advantage of measurement theory" (Fenton and Melton, 1990). "From the standpoint of measurement theory, many of the derived measurements of software that have been proposed ... are meaningless" (DeMillo and Lipton, 1981). "Size," especially as it relates to "effort," has been "a component of almost all [software] cost and productivity models" (Baker et al., 1990): e.g., "we will take the system size scale to be related explicitly to the *effort* to analyze, design, and develop the functions of the system" (Symons, 1988); "for a measure of software size to be useful for software productivity and ... software cost estimation ... it would have to correlate well with the measure of software development effort" (Yu et al., 1990); "if we have a generally accepted product (size) metric P we can estimate the process (cost) metric C" (Rask et al., 1993).

The view that size should correlate with effort in some predictable (formula) way preempts an aim of software measurement, namely, to find which of the competing alternatives is most effective for producing a given program of size s. Thus, size and effort should be independent. We cannot assume that a specification of size for a program predetermines the effort required for its creation (or the reverse). Effort should be dependent on the techniques and technology (e.g., COBOL versus Assembler) used for the creation of the program.

An additional observation concerning size is that the unit of size might not be the unit of output for a productivity metric. An example outside the software arena is a shoe factory that produces shoes in a variety of shoe sizes. Shoe factory productivity measurement will use the number of pairs of shoes produced, not the sum of the shoe sizes.

In this article I do not intend to cover all of the many issues in the software metric area. I do share the view that "many misunderstandings surrounding software measures are due to the failure to make clear the distinction between a) product/process/resource attributes and measures, and b) internal and external attributes and measures" (Fenton, 1991).

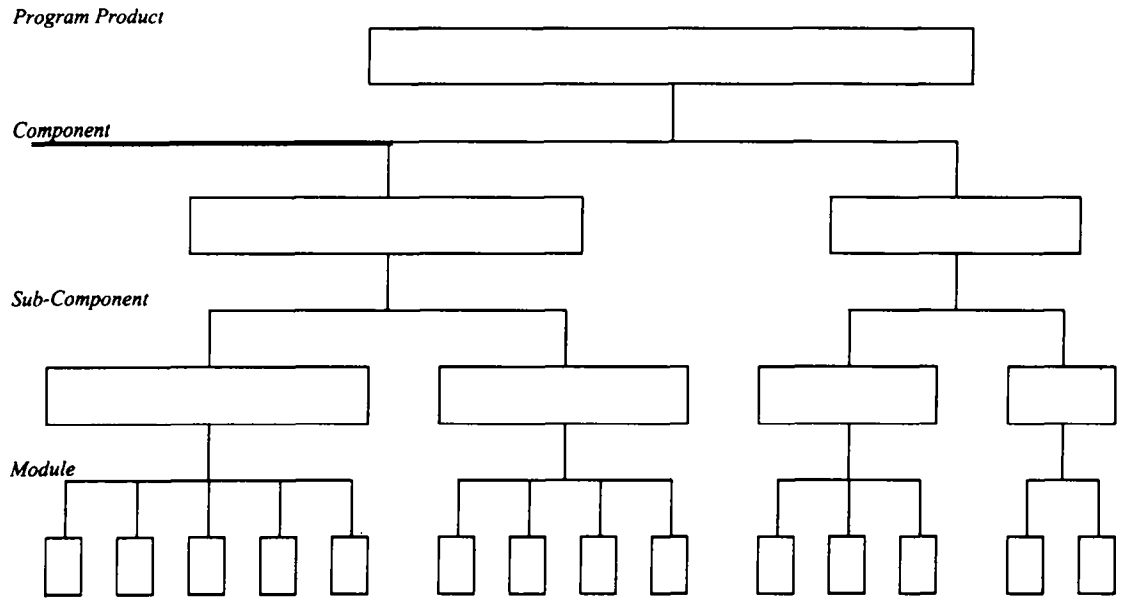

Figure 2. Program structure.

3. UNIT OF WORK

Figure 2 is a common picture of how we describe a program product or system and is accurate for what we ship. It does not, however, best represent what we work on; Figure 3 is a better representation. What we work on is formal [an abstraction (Brooks, 1986)]; this is illustrated by the rectangle outside the hierarchical product structure in Figure 3. The θ symbol shown inside the module level of the hierarchical structure represents the embodiment of this abstract work item; implementation of the abstract work item requires each θ. Of course, we work on

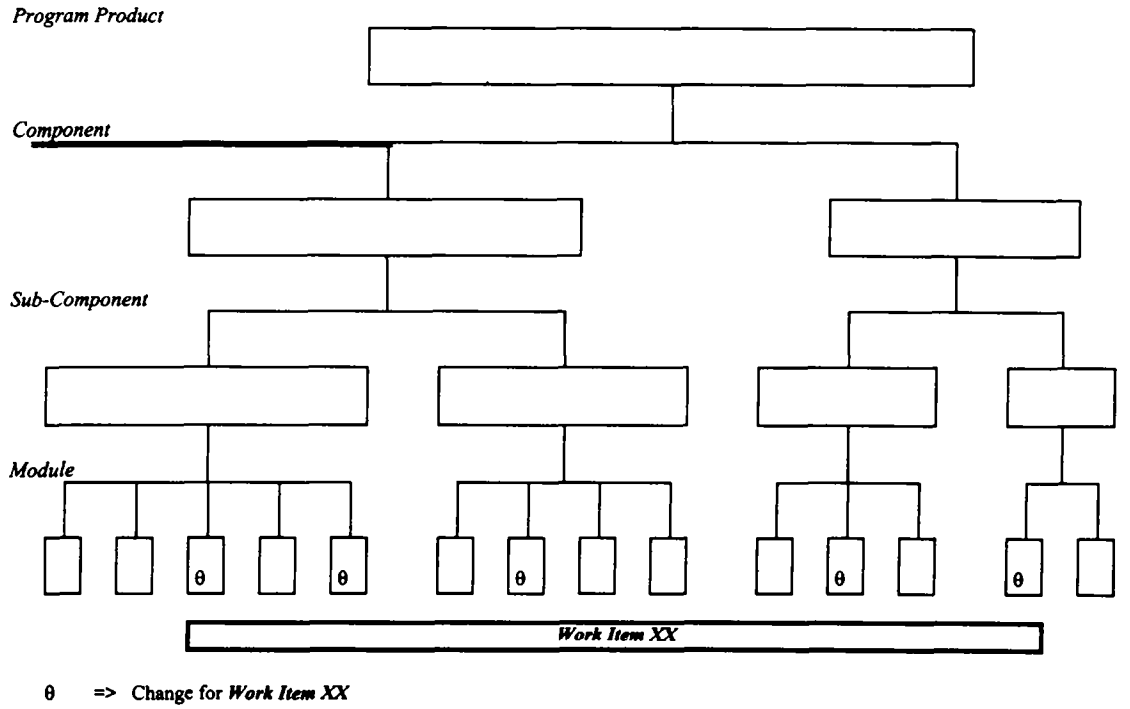

Figure 3. Work item.

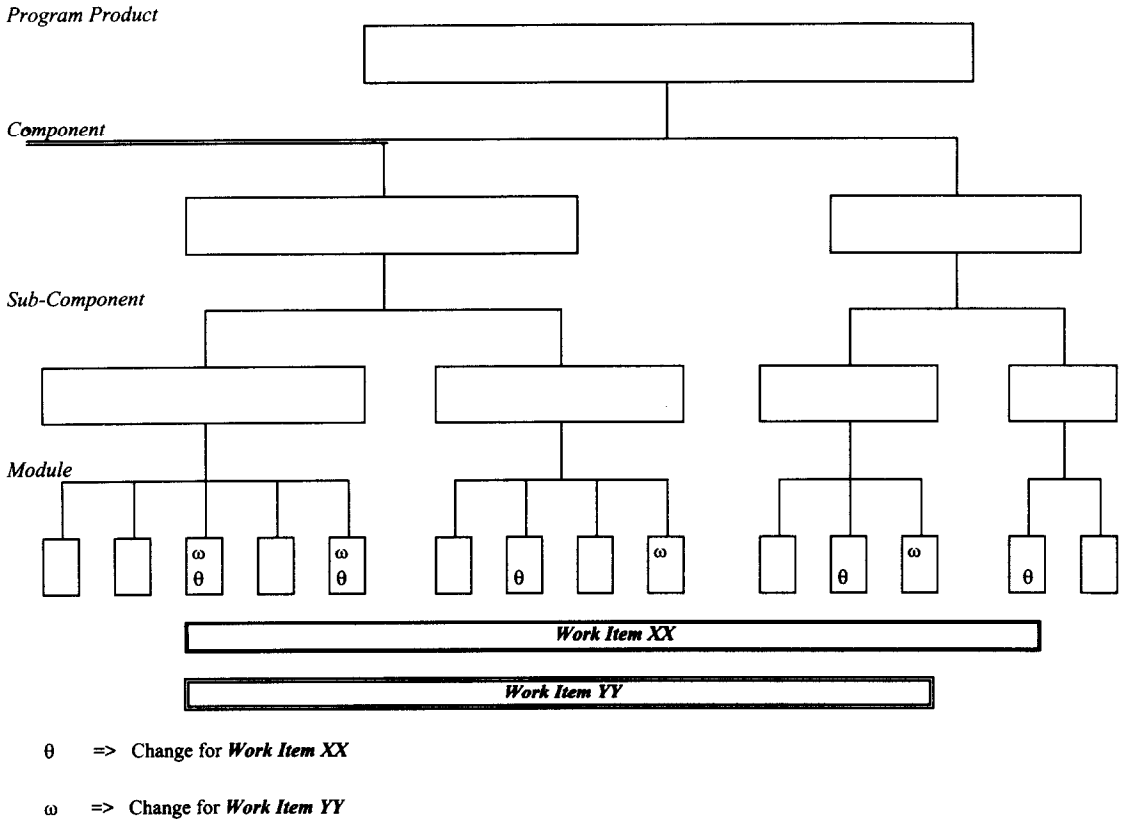

Figure 4. Multiple work items.

multiple items in a release, and they may have an impact on the same modules; Figure 4 shows this.

Figure 5 is an excerpt from what IMS calls a TU (transferable unit) and is a description of work for XRF Restart: "backup" system MFS block load/release. The important things to notice are: (1) there are several program units involved (three new, five modified {REPLACE}, and three affected {REASSEM}); (2) the work involves multiple individuals, here two (TF and JFW); and (3) some modules require zero new or modified LOC.

Figure 6 is a higher level view that illustrates that we work on multiple abstract work items in a release. The example is from the DB2 Release 2 System Plan, and it is a summary of one of their line item workbooks, which contain more detailed information. A line item (abstract work item) can require work in multiple components (sets of modules) within a product that may not all be integrated into the product simultaneously.[6] Some abstract work items may require actions in other products. For example, certain capabilities in DB2 require changes in MVS. We normally call these dependencies. I make this observation merely for completeness and, in this article, make no special use of this characteristic.

In summary, our unit of work

- is a management- or organization-defined abstract work item;
- is a definable set of changes to a definable set of modules (and may involve the creation of new modules);
- can require changes/additions in multiple elements (products, subsystems, etc.);
- may be subdivided if experience suggests that is prudent (usually when we get into trouble).

Additionally, in each of these cases, IMS and DB2, the *GPD Programming Development Process Handbook* does not even refer to TUs or line item workbooks. These things exist because the organization believes them to be necessary to manage its work. The common thread here is that each organization is attempting to manage change. To fulfill their

[6]A "spin" represents an integration point for selected functionality (see Figure 6).

```
&bptu.D05 HS RESTART 1212 0
&bpt."Backup" System MFS Block Load/Release
&mp.FUNCTION:
&p.Provide for the loading and releasing of the MFS BLOCKS
by the "backup" system based upon the events occurring
in the "active" system.
&tupreq. TUD01 TUD02
&tuopreq.
&mp.DESCRIPTION:
&p.Based upon the user Terminal activity
occurring in the "active" system, the "backup" will
mirror the loading and releasing of the MFS Blocks.
This is driven by the x'31' Communication Message Queue Get Unique and
the x'35' Communication Message Queue Enqueue log records.

===> We have a requirement that MFS support multiple TCBs
     because this is running under the Restart TCB concurrent with
     Communication during the CONTINUOUS TRACKING PHASE.
&mp.RESTRICTIONS:
&p.
&mp.TESTING SUGGESTIONS:
&p.Cancel with a dump a "backup" system which was tracking terminal
events occurring on an "active". Inspect the 'FRE's to determine
whether the MFS Blocks were correctly loaded and released.
Also, check all 'CIB's for correctness.
&tufpfs.'1/28/83' '-' '1/28/83'
&tulogic.'3/01/83' '-' '3/18/83'
&tucode.'4/01/83' '-' '3/31/83'
&tuut.'4/15/83' '4/29/83' '4/29/83'
&tuxfer.'4/15/83' '4/29/83' '5/2/83'
&tudr2.'1/28/83'  '1/31/83' '1/31/83'
&tudr3.'3/15/83' '-' '3/18/83'
&tui0.'3/15/83' '-' '3/18/83'
&tui1.DFSCRSP0  '4/15/83' '4/22/83' '4/29/83'
&tui2.'4/15/83' '4/22/83' '4/29/83'
&tuutc.'4/15/83' '4/29/83' '4/29/83'
        * * * * * * * *
&tui1.QLOGENQU  '4/15/83' '4/22/83' '4/29/83'
&tui2.'4/15/83' '4/22/83' '4/29/83'
&tuutc.'4/15/83' '4/29/83' '4/29/83'
&tumm.DFSCRSP0 MODULE  18 REPLACE TF
&tumm.DFSFFRT0 MODULE 318 NEW     JFW
&tumm.DFS[illegible]0 MODULE 228 NEW     TF
&tumm.DFSICV50 MODULE  40 REPLACE JFW
&tumm.DFSIINF0 MODULE  62 REPLACE JFW
        * * * * * * * *
&tumm.DFSQLOG0 MODULE  76 REPLACE TF
&tumm.DFSFRT   MACRO  282 NEW     JFW
           * * * * * * * *
&tumm.QLOGENQU MACRO    2 REPLACE TF
&tumm.DFSCRPV0 MODULE   0 REASSEM TF
&tumm.DFSCRP00 MODULE   0 REASSEM TF
        * * * * * * * *
&tumm.DFSQRST0 MODULE   0 REASSEM TF
```

Figure 5. IMS TU—*Replica*.

responsibilities, organizations factor and document these responsibilities to whatever degree thought required for the success of the organization. Change management is extremely important to success; thus, organizations create *work breakdown structures*.[7]

Let me call this unit of work an AWI (abstract work item). A working definition for an AWI is as follows: the intended *result* of a *set of actions* on a software product's source text. The result is vitalized through a set of *internal* or *external* product attributes; the actions are process activities performed to achieve the desired product attributes. The level of detail (degree of factoring) in a project's documented work breakdown structure may not always explicitly reveal the result (AWI) as specific product attributes (e.g., "update the payroll to reflect the new tax laws" or "improve performance by 5%"). Thus, the documented expression of an AWI may be subject to some ambiguity; nevertheless, clearly, such decomposition must occur.

3.1 CHANGE-POINTs

The use of AWI as the unit of output in the productivity metric has a certain appeal. However, to man-

[7] "The purpose of a WBS {*work breakdown structure*} is to divide the total project into small pieces, sometimes called *work packages*. Dividing the project into work packages makes it possible to prepare project schedules and cost estimates and to assign management and task responsibility" (Nicholas, 1990).

DB22PLAN

LINE ITEMS (CONTINUED)

(DB22LINE)

DEPT	LINE ITEM	DESCRIPTION		LOC
	LI40	SPUFI as a DSN Subcommand -- Spin 3		
		SPUFI (TAC)		1157
	LI Total			1157
	LI41	ISPF Panel Restructure -- Spins 2,3		
		ISPF/DB2I Panels (TAC)	&	5038
		Installation	&	414
		Precompiler/Parser	&	555
	LI Total		&	6007
	LI52	Interpreter Rework -- Spin 3		
		RDSM		28
		RDSI	&	409
	LI Total		&	437
M92 cont	LI61	ENCODE/DECODE Exit Support -- Spin 5		
		RDSI	&	2205
		RDSM	&	2238
		PC	&	265
		UC	&	1157
		BMC	&	344
		CAT	&	546
	LI Total		&	5650
	Department Total		&	40977
M09	LI02	Sequential Prefetch -- Spin 2		
		Data Manager		11
		RDSM		214
		Utilities		20
		Buffer Manager		56
		SPMC		200
	LI Total			501
	LI05	Restartable Load w/o logging -- Spin 3		
		Utilities		641
	LI Total			641
	[illegible]			[illegible]
	LI Total			575
	LI10	Recoverable REORG without Logging -- Spin 3		
		Utilities	&	79
	LI Total		&	79
	LI11	Discard File for Load -- Spins 2,4		
		Utilities	&	600
		TAC		20
	LI Total		&	620
	LI12	Delete Keyword -- Spin 1		
		Utilities		490
		Data Manager		9
	LI Total			499

Figure 6. DB2 System Plan—*Replica*.

age change, there clearly are many more elements to deal with than are represented by a count of AWI. As shown in Figure 5, an AWI implementation might use multiple program units. Specific program units will be created and/or used in support of implementing that AWI. Those program units are expected to produce certain effects (under specific conditions) within the program in support of that AWI.

Thus, recalling Figure 3, we can recognize the building blocks of an AWI—what I call CHANGE-POINTs. Please review Figure 5. Each new module will have a set of execution-time effects. Each modified module will have its set of execution-time effects altered to accommodate the new AWI. Each macro will have a set of compile-time effects.[8] Each reassembled module will have its set of execution-time effects modified in some way that accommodates this new AWI. If the correct set of effects needed from any of these program units is not specified or implemented, then there is an error in the AWI design or implementation. If not all the needed new, modified, or affected program units are identified or implemented, then there is an error in the AWI design or implementation. Each of these program units is a locus of change in support of this

[8]Some macros are conditional; some header files are conditional.

AWI. A CHANGE-POINT is defined as the intended *result* of a *set of actions* on a single new, modified, affected, or deleted program unit used to materialize a particular AWI. The result is vitalized through a set of *internal* or *external* program unit attributes; the actions are process activities performed to achieve the desired program unit attributes. More specifically, a CHANGE-POINT is vitalized through a *subset of* (*possible*) *effects* within the *set of* (*possible*) *effects* in or from a single new, modified, affected, or deleted program unit used to materialize a particular AWI.[9] Context (internal and/or external) affects which effects are expected to occur (e.g., is a particular symbol defined? is the value of a particular variable equal to 1? did a divide exception occur? did the user click on "Close File"?), and these expectations will be used to judge the correctness of the AWI and CHANGE-POINT design and implementation.[10]

"Program unit" is imprecise, because it can reference any level of the hierarchy shown in Figure 3, but mixing of levels should not occur. Within the context of this article, "program unit" and "module" are synonyms and refer to the lowest level of the hierarchy, assumed to be source modules (including merely definitional source, e.g., DSECT, INCLUDE, #include, and macro files). "Source modules" is also imprecise; I discuss this shortly. New, modified, affected, and deleted program units have somewhat different roles regarding change in a program; this is discussed below.

3.1.1 Affected modules. Figure 7 shows some C code modules; for the moment, just consider ModuleA and ModuleB. ModuleA is purely definitional. Assume that the design for some new AWI requires J and I to be long instead of int. To support implementation of the new AWI, we must modify ModuleA by changing "int" to "long." Merely changing ModuleA is not sufficient: ModuleB is affected by this AWI design.[11] For this AWI implementation, the (compile-time) effect in ModuleA is new semantics for aTYPE (and for paTYPE), and the (compile-time) effect(s) in ModuleB from ModuleA ensure that the (execution-time) effects from ModuleB statements using I or J remain valid.

```
ModuleA:
typedef int aTYPE;
typedef aTYPE *paTYPE;
```

```
ModuleB:
#include "ModuleA"
extern aTYPE J, aTYPE I;
void function1(void)
{
                        if ( I == 1 )
                         {
                         spec_code();
                         J = 2;
                         }
                        else
                         J = 3;
}
```

```
ModuleB':
#include "ModuleA"
extern aTYPE J, aTYPE I;
void function1(void)
{
                        J = ( I == 1 ) ? (spec_code(),2) : 3;
}
```

```
ModuleB'':
#include "ModuleA"
extern aTYPE J, aTYPE I;
void function1(void)
{
                        switch ( I )
                        {
                        case 1:
                         spec_code();
                         J = 2;
                         break;
                        default:
                         J = 3;
                         break;
                        }
}
```

Figure 7. C code.

Affected program units are altered in support of some AWI that changes the product design or implementation, but the alteration is of a different nature than for modified modules. The source text of an affected module remains the same (e.g., ModuleB[12]), whereas the source text for a modified module does not (e.g., ModuleA). The action taken with an affected module is intended to preserve (restore) its integrity and in some cases will add or remove capability.

3.1.2 New or modified modules. New or modified program units are created in support of some AWI that changes the product design or implementation. A program unit may have overlapping or nonoverlapping subsets of effects regarding multiple AWI. A

[9] Cf. C/C++ usage of "side effect(s)" and "side effect operator."

[10] See testing suggestions in Figure 5.

[11] If the size of int is smaller than the size of long, tests of I and assignments to J can be erroneous if ModuleB is not recompiled.

[12] Preprocessor output will differ, as will the object code, but like the object code, preprocessor output is not the source text for ModuleB.

module might contain some entry points used only in support of a particular AWI and others that are used only in support of another AWI. As a practical matter, there are many reasons to combine into a single module the needs of multiple AWI, for example, (1) performance improvements from internal subroutine sharing, (2) much of the module is reusable without replication, or (3) storage savings from sharing static data.

3.1.3 Deleted modules. Design and implementation of an AWI can require deletion of program units. Deleted program units are alterations to the program (structure) intended to have a *null* set of effects (e.g., removing support for old devices). Sometimes deleting a program unit merely means allocating to other (new or modified) modules whatever was previously done using the deleted module. Splitting a program unit can be desirable for many reasons, one of which is simply to create units of more manageable length. Not deleting unneeded program units is not always benign.

The designers and implementors of OS/2 2.1 Special Edition had to strip out Windows; testing had to verify that the actual deletions did not have undesirable effects. For legal reasons, Borland has to delete any modules whose only purpose is to support Lotus 1-2-3 emulation in their spreadsheet product.

3.2 Design, Implementation, and Test of Change

High-level design (see Appendix 3) identifies CHANGE-POINTs by (1) assigning a set of responsibilities to a new, modified, or affected program unit relative to an AWI, or (2) indicating actions needed (e.g., recompile) for an affected program unit relative to an AWI, or (3) indicating deletion of specific program units relative to an AWI. According to *Programming Process Architecture* (1986), component level design "defines: all new, changed, and affected modules, macros, and their functions[;] control and function flow to the intermodule level[;] all intermodule interfaces, including parameter values[; and] all data definitions." In this sense, high-level design produces *specified* CHANGE-POINTs, so in the context of productivity, it makes sense to think of CHANGE-POINTs as an output of design effort.

During implementation (see Appendix 3), a CHANGE-POINT is materialized by a set of source text (in one or more languages[13]) within a program unit, or by an action performed with a program unit relative to an AWI (e.g., recompile because of a change in a data structure shared with another program unit). Assume ModuleB in Figure 7 is created in support of some new AWI. ModuleB′ and ModuleB″ are semantically identical to ModuleB and invoked identically. They are examples of possible alternative materializations of the CHANGE-POINT ModuleB implements; ModuleB, ModuleB′, and ModuleB″ produce the same set of *effects*. An optimizing compiler might generate identical object code for these variations. Materialization of a particular CHANGE-POINT might use alternative sets of source text; in this sense a CHANGE-POINT is LOC independent (see also Appendix 2). It is important to note that the source text used to implement a CHANGE-POINT might not be contiguous within a program unit. Furthermore, because source text changes used to implement a CHANGE-POINT in a modified module often take advantage of surrounding source text, a CHANGE-POINT is not, in general, materialized *only* by new source text.

The implementation technology and the design that selected the technology determine what makes up source or source modules. For example, the input to an application generator or report writer (e.g., RPG) can be considered a higher level of source than input to a compiler (e.g., PL/I). Currently, great interest exists in raising the level of implementation above high-level languages (HLL) to get benefits above and beyond what is achievable with current HLL.

Mixing of implementation technology also occurs. It is common, for example, for development projects to use both assembler and one or more HLL. In theory, designers should minimize the number of CHANGE-POINTs for a project (set of AWIs), but practical trade-offs can exert an upward push on the number of CHANGE-POINTs needed to implement a project.

Test (see Appendix 3) validates specified and implemented CHANGE-POINTs for an AWI using a set of test cases. Currently, validation uses machine execution of the developed product, and I speak about test in this context. I leave proofs and other techniques to the future.

Test does not produce CHANGE-POINTs except in the undesirable sense of detecting incomplete design. In the ideal case, test should execute sufficient test cases to ensure that the CHANGE-POINT implementations accomplish the intended objective, the AWI (function test, system test), without undesirable side effects (function test, regression test,

[13]Some high-level language compilers support embedded assembler language, e.g., IBM's proprietary PL/AS and Borland's C.

system test). In this sense, test produces *validated* CHANGE-POINTs, so in the context of productivity, it makes sense to think of CHANGE-POINTs as an output of test effort.

During a project, a CHANGE-POINT is one of the sets of changes that management and the responsible developers must design, implement, and test (on schedule, with high quality, and at lowest cost). A CHANGE-POINT is more granular than an AWI (in that by definition it relates only to a single program unit) but more global than a LOC (in that implementation might use multiple LOC). CHANGE-POINTs seem usable as the unit of output in our productivity metric.[14]

3.2.1 Counting CHANGE-POINTs. To have consistent and repeatable CHANGE-POINT counts for a given design, we need a counting rule. The rule should be simple and seem consistent with the number of changes made and managed. The rule to be used is this: for each AWI, count 1 for *each* module modified, created (new), deleted, or affected that is used to vitalize that AWI; only count a module once for that AWI.[15] Summing gives the *total* CHANGE-POINT count for the design; total CHANGE-POINT count less the sum of the counts for deleted modules gives the *net* CHANGE-POINT count. Summing the total or net CHANGE-POINT counts of multiple AWI gives, respectively, the total or net CHANGE-POINT count for that set of AWI (Figure 8).

Total and net CHANGE-POINT counts for multiple AWI are distinguishable from a simple count of *unique* modified, new, deleted, and affected modules for that set of AWI. Given the above counting rule, it should be clear that, generally, total and net CHANGE-POINT counts will exceed a simple module count. One reason to go beyond a simple module count is that such a count would not necessarily reflect any or significant change because of removal or addition of an AWI.

3.2.2 CHANGE-POINTs and function points. Because many readers may have heard of function points, I must comment on some distinctions between CHANGE-POINTs and the function point concept as described by Albrecht and Gaffney (1983).

Five user function types are the building blocks for a function point count: (1) external input; (2) external output; (3) logical internal file; (4) external interface file; and (5) external inquiry. Fundamental to identifying the user function types is the notion of "the external boundary of the application being measured" (Albrecht and Gaffney, 1983). "The amount of the 'function' the software is to perform ... is quantified as 'function points,' essentially, a weighted sum of the number of 'inputs,' 'outputs,' 'master files,' and 'inquiries' provided to, or generated by, the software" (Albrecht and Gaffney, 1983).

Total and net CHANGE-POINT counts, on the other hand, depend on the partitioning of the AWIs for the application or program being measured and have no dependency on recognition of the user

[14] Design, implementation, or testing of a CHANGE-POINT could each be a work package (or subcategory) in a work breakdown structure, because these are activities with a duration and a timing/position relative to other activities in a project, but CHANGE-POINTs are not themselves elements (work packages or subcategories) of a work breakdown structure; rather, they are logical elements of the software product.

[15] New, modified, affected, and deleted assume some reference program structure (e.g., as in Figure 2) and program unit properties (e.g., deleted assumes the module is *in* the reference program structure; affected assumes the module is *in* the structure and does *not* have certain properties). The CHANGE-POINT count for an AWI is (should be) relative to the program structure and program unit properties assumed in the AWI design.

	AWI	New	Modified	Affected	Net Count	Deleted	Total count
	A	1	1	1	3	1	4
	B	3	5	3	11	0	11
	C	5	2	2	9	0	9
	D	1	10	0	10	0	11
AWI Set	4	10	18	6	33	1	35

Figure 8. Counting.

function types. Certain classes of rework of an operational application would result in zero [Mark II (Symons, 1988)] function points but nonzero total and net CHANGE-POINT counts: for example, performance enhancements.

"Albrecht... developed a methodology to estimate the amount of the 'function'... software is to perform... quantified as 'function points'" (Albrecht and Gaffney, 1983). CHANGE-POINTs relate to *change* in the software and its *structure*.[16] Function points are widely accepted as a size metric by both practioners and academic researchers (Kemerer and Porter, 1992) and have approached the position of being a de facto standard as a size metric (Symons, 1988), although Albrecht and Gaffney (1983) proposed function points "as an alternative to 'size.'"

Symons (1988) identified limitations of function points (including Mark II function points). Kemerer and Porter (1992) addressed reliability of function-point counts. An additional difficulty with function points is as follows: the meaningfulness of computations using the simple (low), average (medium), and complex (high) weights, and the degrees of influence (impact), is not well specified (by definition, these are *classifications* and so might be better represented as letters). Furthermore, even if these factors can be reliably assigned, this does not mean that total unadjusted function points (UFP) or total degree of influence are meaningful (e.g., summing [1] the result of a length in inches times a factor that converts it to feet and [2] the result of a length in centimeters times a factor that converts it to meters can produce a numeral {[1] + [2] = [?]}, but this numeral is not meaningful, although the factors can be reliably assigned).

Symons (1988) considered "function points... dimensionless numbers on an arbitrary scale." This seems inconsistent with the use of weights. The presumed purpose of the weights is to convert items to a common scale (e.g., L number of low-complexity external inputs is equal to M number of medium-complexity external outputs and H number of high-complexity logical files, on some nonarbitrary scale). If the scale is arbitrary, then, at least, each triplet of weights is independent of the others (e.g., there is no reason for the weights for external inputs and external inquiries to be identical) and, therefore, different sets of triplets are usable to compute UFP; this means UFP and FP are not unique for a given design.

Convention might lead to a completely reliable method for counting user function types and for the assignment of weights, and thus for computing function points. If, as Symons (1988) maintained, different technologies require the use of different weights for FP, then relating FP_{T1} to FP_{T2} requires some function N to map FP_{T1} to FP_{T2} or some function $\tilde{N}$ to map FP_{T2} to FP_{T1}, to be meaningful; the scales involved may be complex, but they cannot be arbitrary.

If FP does not require different weights for different technologies (and/or one maintains that FP is itself a scale), then we still have the *zero UFP problem*: whenever UFP is zero, FP is also zero, thus implying that productivity is zero; this can occur for certain classes of work, e.g., software defect repairs that do not affect the definition of or quantity or complexity of user function types, and some changes to macros in C #include files.[17] Also, user function counts are integers, so the current standard weights cause UFP to be a discontinuous count that sets an upper and lower bound on UFP for *all possible* technologies usable for implementation of *all possible* applications that have the same counts for each of the five user function types.[18] The view of software leading to these features requires explanation; "measurement without an underlying theme can leave the experimentalist, the theorist, and the practitioner very confused" (Chillarege et al., 1992).

A major tenet of Albrecht and Gaffney (1983) is that function points have a high correlation with the eventual LOC. They suggest using "'function points' to estimate 'SLOC,' and then using 'SLOC' to estimate the work-effort." Jones (1991) considers backfiring a reliable method for converting LOC to function points. Converting function points to LOC and converting LOC to function points are not valid rescalings, in part because LOC is a continuous (unbounded) count. Furthermore, most such conversion factors vary according to the coding and project

[16] Just as a software entity is an abstraction (Brook, 1986), so, too, are CHANGE-POINTs.

[17] Albrecht and Gaffney (1983) refer to "counting the function points... changed by the development... project," but this can only mean changed user function types from which UFP is derived ["ChgA" and "ChgB" (Albrecht and Gaffney, 1983)]. Ellipsis when speaking of added or deleted function points, while problematic, is not nearly as dangerous as when speaking of changed function points.

[18] For example, for the function point calculator application in Jones (1991), valid UFP values are {18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 35}, *not* 34, and no value < 18 or > 35 is valid for any technology or application that keeps EI, EO, ILF, EIF, and EQ the same. Note also that the current standard UFP weights can be reduced to four {3, 4, 5, and 7—the unique simple (low) weights}, perhaps three {3, 4, and 5}, and just possibly to one {3}.

styles used in particular organizations and thus are not general. In addition, for mixed-language applications, backfiring requires adding LOC in different source languages, without adjustment [see Jones (1991), pp. 77-78].

Using the same function point count for different languages in *prediction models* [or "'formula' estimates" (Albrecht and Gaffney, 1983)] for effort might be misleading: if effort varies with source language, then using different source languages to develop the same application might imply *different* function point counts (due to different designs) should be used for the application in *some* of the languages as one way to eliminate or reverse the differences in effort. Rejecting this view forces one to accept that for *every* application, there is some function point count, f, for some least effort design in a particular language[19] such that f is the function point count for some least effort design for that application *in each* of all possible languages.[20] If there is no shared function point count among least effort designs for all possible languages for an application, then function points do *not* "stay constant regardless of the programming language used" (Jones, 1991) for purposes of comparison of least effort among languages. Thus, predictions of effort using a common function point count are not sufficient to make least-total-effort evaluations: for a specific application, least effort rankings might not correspond to function-point rankings—a language that does not have the smallest function point count might need the least effort, or the effort for design D in language L could be less than the effort for *all* designs that have a common function point count in *all* languages.

3.3 Issues and Questions

This section addresses a few issues and questions that may be on the minds of readers.

3.3.1 LOC not factored into count. Because LOC are not outputs of Design or Test, including LOC means we cannot use CHANGE-POINTs as a common output. Furthermore, current indications are that coding is only 10-20% of Implementation effort.

3.3.2 Different managers or developers, given the same implementation or enhancement task, will define different sets of AWI and thus generate different CHANGE-POINT counts. Exactly. "The hardest single part of building a software system is deciding precisely what to build" (Brooks, 1986). One hopes that software science can provide metrics useful for comparing such decisions. "Software construction is a *creative* process" (Brooks, 1986). "*Design and programming are human activities; forget that and all is lost*" (Stroustrup, 1991). Using total or net CHANGE-POINT count alone will not determine what to design; no mere count will suffice for that purpose, but prediction models or assessment formulas that use that count (or its components) may help. If we consider a case where different designs for the same system or application result in the same total and net CHANGE-POINT counts, then the selection of one design over the other will depend on other factors (e.g., elapsed-time target, total resources needed, usability, skills available, etc.).

3.3.3 How is "set of effects" different from "states" or "set of states"? States in software systems seem best related to an execution time view of programs. CHANGE-POINTs also relate to purely definitional program units. Even header files that only enable the use of convenient symbols require change management. Modification of such modules can have serious development and execution-time consequences. However, I do not think it is useful to talk about such a module as a "component [that] deals with a small number of cases" in the sense intended in Parnas (1985). The CHANGE-POINT approach integrates a construction-time (design, build, and test) view and an execution-time view of the software system. Development organizations must deal with the complexities of both the construction and execution contexts.

3.3.4 Complexity not factored into count. To the extent this is true and a problem, it is not unique to this approach. LOC counts, for example, do not factor in differences resulting from variations in complexity. Furthermore, because total and net CHANGE-POINT counts are not merely counts of unique modules, total and net CHANGE-POINT counts reflect complexity. A 100-LOC change might be more or less complex than one-hundred 1-LOC changes, but the latter are likely to be more difficult

[19] This assumes an optimal work breakdown structure for implementing that design. It is noteworthy that empirically derived cost prediction models may be plagued by less-than-optimal matches between designs and project work breakdown structure.

[20] This is not to say that there is no single design with the same function point count in each language, nor that there cannot be a collection of designs such that, although they may differ by language, their function point count is the same, nor that *all* least effort designs in *each* of all possible languages for an application must have the same function point count.

to manage, which could result in differences in productivity.

I do not know the "right" way to measure complexity, but some indicators might be as follows[21]:

- CP/Module (change density—structure)
- Modules/CP (change dispersal—structure)
- CP/AWI (change scope—structure)
- AWI/CP (change compactness—structure)
- CP/LOC (change concentration—implementation)
- LOC/CP (change spread—implementation)

Summing CP_{awi}, CP_{awi}/LOC_{awi}, or LOC_{awi}/CP_{awi} yields an indicator that increases or decreases when adding or deleting AWIs.

As noted earlier, a simple counting rule is desirable. Complex counting rules tend to be impractical and subjective. Setting up elaborate weighting procedures without factual knowledge about the influence of things factored in could be misleading and counterproductive.

3.3.5 External interfaces not factored into count. First, this is primarily a problem regarding modules that have multiple entry points or multiple types for a single argument. Second, external module references imply a count of at least two for the relevant AWI, so total and net CHANGE-POINT counts reflect such references. Module recompiles required because of control block modifications allow total and net CHANGE-POINT counts to reflect such interface changes.

3.3.6 Why not just use LOC? Each LOC is a change. LOC is implementation dependent and varies because of compiler facilities, skill level, etc.; again, it is not applicable to Design and Test.

3.3.7 CHANGE-POINT counts ignore the size/complexity of a change (e.g., a 10-LOC change is not equal to a 50-LOC change or a 100-LOC change). Neither "size" nor "complexity" of a change has a uniform relationship to LOC. A 1-LOC change can be complex because of all the things and relationships that one must know and keep in mind before or when making that change. A 10-LOC change can be "larger" than a 100-LOC change because the 10-LOC change spans 1,000 LOC, more than one module, and has substantial potential for system side effects, whereas the 100-LOC change is contiguous and has no potential for substantial system side effects.

Part of this problem is that we are used to sizing changes with LOC. Our reaction to a 10-LOC change is that it is a small change, easy and quickly done. LOC is only one dimension. Test impact, documentation impact, performance impact, service impact, etc., influence the size of a change. The size of a change is distinct from the count of its implementation LOC.

The issue seems to be that we don't now know the effects of all the things someone may think are important, so we should do nothing. Isn't it better to do something that will allow us to start finding out these effects (with data)? Otherwise, the issue must be whether total or net CHANGE-POINT counts are useful for productivity measurement, which is different, but permitted.

3.4 CHANGE-POINT Count: What Good Is It?

Several characteristics of a CHANGE-POINT count commend it for use in a productivity metric:

1. CHANGE-POINT counts are usable before product ship as an *actual* value (not something that is part actual and part projection). For example, once we have an established component level design baseline, the number of CHANGE-POINTs specified is a known value, not, as with lines of code, a number not yet determined. Depending on the tracking system, we can even get intermediate counts of completed CHANGE-POINTs based on the status of the component design documentation. Having a number that is a known value enables us to evaluate productivity as we go along, and thereby helps us take actions that will potentially affect the final productivity of the project.
2. CHANGE-POINT counts are probably more stable than LOC counts. We are all, no doubt, familiar with the fluctuations (growth) in LOC as projects progress through development phases. A fluctuating CHANGE-POINT count shows design instability or error.
3. Because CHANGE-POINTs relate to a logical organization of our work, CHANGE-POINT counts are usable even when process steps overlap. This is because process overlap, when successful, maintains a logical structure that allows us to track the status of an AWI. At any particular process step, we know the relationship of the

[21] "Module(s)" means either "total" or "new + modified + affected."

items being handled to their overall purpose (the AWI).

4. We know that change management is essential. Improvement in productivity using CHANGE-POINT counts reflects management efficiency, not just worker efficiency. The way we plan to handle change affects the potential for productivity improvement.
5. *Specified* CHANGE-POINTs are independent of implementation source language; the implementation language used for individual modules need not affect the CHANGE-POINT count.[22] This avoids the usual problems associated with different implementation languages, e.g., conversion of LOC counts in language x to LOC counts in language y.
6. CHANGE-POINT counts are usable for comparing different languages: selection of a language that allows a high-level design with a different CHANGE-POINT count could represent an opportunity for productivity or effectiveness improvement.
7. CHANGE-POINT counts are usable for periods of one year or less. Thus, periodic productivity assessments are possible; because of (3) above, these can include work in process.
8. Automation of counting of CHANGE-POINTs is possible. Version and modification control identifies new or modified LOC (by sequence number or other indicators). It is not a complicated matter to identify the AWI(s) associated with new, modified, affected, or deleted modules. A high-level design should indicate each module created, modified, affected, or deleted in support of each AWI; if this AWI information is in machine-readable form, then automation of counting CHANGE-POINTs is possible.

4. THE METRICS

This section discusses productivity, interference, and effectiveness metrics (the PIE matrix) for each development activity (Figure 9). Appendix 3 contains additional discussion of the terms "interference" and "effectiveness" and other terms used in this article. The focus here is on illustrating analysis that uses CHANGE-POINT counts for *assessment*, as opposed to *prediction*. The reason for this focus is that it is probably true that predictions of project productivity or effort will have large variances from actual results, unless effectiveness for Design, Implementation, and Test is at a high level.

4.1 The Data

Figures 10 and 11 present data collected for three releases of a product developed at STL (Ra, Rb, and Rc; Rc is the most recent release, Rb is its predecessor, and Ra is Rb's predecessor). Current data retention does not preserve all the data implied by the CHANGE-POINT approach, so the results shown in Figure 11 are incomplete. In particular, there is no AWI level data for Ra and Rb. In addition, the Rc values in Figure 11 use cld^{CP} (new, modified, and affected modules at the end of formal test) from Figure 10 as a surrogate for $CLDCP$ and CLD_bCP (this is equivalent to assuming no new, modified, or affected modules were added after design completion, i.e., $RCCP$ and $DCCP$ are zero). This is a best-case assumption (e.g., no implementation interference) and results in effectiveness values for Design that may be better than actual data would produce.

The cost data for Design used in Figure 11 are for effort through I1 (detailed design) and not I0 (component level design). This is because the Financial Accounting Standards Board's (FASB) rules treat cost through I1 as design, and the current data retention system meets this requirement. The result is that design productivity as shown may be less than actual data would produce, and implementation productivity as shown may be better than actual data would produce.

Cost data *by activity name* may not be comparable from one company to another, or even within the same company over time, because the *recorded* classification of departments and activities will vary. To deal with this problem, I have related cost to standard FASB accounting classifications in the following manner: as stated above, Des$ is per FASB; Test$ is formal test or performance amortized expense; Imp$ is amortized expense less Test$ and less expense associated with departments or functions that do not have responsibility for writing or designing source text (e.g., publications and assurance). Programming productivity analysis and measurement will have this type of ambiguity until establishment of FASB-like standards for classification of activities, and this will occur only when such standards appear to have importance to project success.

[22] Interlanguage communication in modern compilers makes mixing of languages a practical consideration. It is true that selection of a particular language might affect the product high-level design and thus the CHANGE-POINT count for a project; however, it is not true that a high-level design must be language specific (any module written in a high-level language *could* be written in assembler).

STAGE	*Data From*	PRODUCTIVITY METRICS	INTERFERENCE METRICS	EFFECTIVENESS METRICS
Design	Component Level Design	*CLD*CP/Des$ *Des*CP$_p$/Des$$_p$	*RCCP*/*CLD$_b$*CP	*DCCP*/*CLD$_b$*CP *CLD$_b$*CP/CP
Implementation	Unit Test	*CLD*CP/Imp$ *Imp*CP$_p$/Imp$$_p$	$1 - \frac{CLD_bCP}{DCCP+CLD_bCP}$	$\frac{DCCP+CLD_bCP}{CP}$
Test	Test and Service	*CLD*CP/Test$ *Test*CP$_p$/Test$$_p$	*FTCP*/CP	CP/(*SCP*+CP)

OVERALL: *CLD*CP/P$; PCP$_p$/P$$_p$

DEFINITIONS:

***CLD*CP:** Net CHANGE-POINT count from final **C**omponent **L**evel **D**esign.

CLD$_b$*CP:** Net CHANGE-POINT count from the ***base **C**omponent **L**evel **D**esign.

CP: *CLD*CP + *FT*CP.

***DC*CP:** Total CHANGE-POINT count for **D**esign **C**hanges after the base Component Level Design is established.

Des*CP$_p$:** Net CHANGE-POINT count for completed Component Level ***Design activities within a ***p***eriod.

Des$: Design total expense.

Des$$_p$: Incremental Design total expense between two ***p***eriods.

FT*CP:** Sum of fix module counts (during ***Formal ***T***est, count 1 for each new, modified, deleted, or affected module due to a software-defect repair).

Imp*CP$_p$:** Net CHANGE-POINT count for completed ***Implementation activities within a ***p***eriod.

Imp$: Implementation total expense.

Imp$$_p$: Incremental Implementation total expense between two ***p***eriods.

P$: Des$ + Imp$ + Test$.

P$$_p$: Des$$_p$ + Imp$$_p$ + Test$$_p$.

PCP$_p$: *Des*CP$_p$ + *Imp*CP$_p$ + *Test*CP$_p$.

RCCP*:** Total CHANGE-POINT count for ***Requirements **C**hanges after the requirements or objectives base is established.

SCP*:** Sum of fix module counts (after release, count 1 for each new, modified, deleted, or affected module due to a ***Service software-defect repair).

Test*CP$_p$:** Net CHANGE-POINT count for completed ***Test activities within a ***p***eriod.

Test$: Formal Test total expense.

Test$$_p$: Incremental Formal Test total expense between two periods.

Figure 9. The metrics.

4.2 Design

This section discusses the design metrics included in the PIE matrix.

4.2.1 Productivity metrics. *CLD*CP/Des$ is straightforward; it is simply the final number of outputs divided by the cost to produce them. During a project, cumulative *Des*CP$_p$/Des$$_p$ would let us see how design productivity is changing in a life cycle fashion. If we have life cycle history, we can compare the current project to history to see if the project is typical or shows expected changes, and attempt to investigate and understand any atypical or unexpected results. For an organization, we can sum *Des*CP$_p$ and sum Des$$_p$ for its various products for a period (year or month or quarter) and thus compute the high-level design productivity of the organization for that period.

Currently, there is a strong feeling (which I share) that much more needs to be done in the area of

Release C (Rc) AWI	LOC[a]	cld[CP]	ptm[CP]	ff[CP]	CP	CP/LOC[b]	LOC/CP[c]
1	1630	42741	371	1838	476	2,685	0.0628	15.9184
2	1632	1670	93	441	122	656	0.3928	2.5457
3	1633	13127	375	1687	392	2,454	0.1869	5.3492
4	1640	1743	66	335	137	538	0.3087	3.2398
5	1790	5067	304	390	380	1,074	0.2120	4.7179
6	1820	3160	259	445	449	1,153	0.3649	2.7407
7	1830	1742	148	179	133	460	0.2641	3.7870
8	1840	2310	14	13	57	84	0.0364	27.5000
9	1850	3284	72	149	232	453	0.1379	7.2494
10	1890	42741	957	2165	1391	4,513	0.1056	9.4706
11	1892	8658	83	190	38	311	0.0359	27.8392
12	1910	8155	278	772	433	1,483	0.1819	5.4990
13	1920	1239	59	258	177	494	0.3987	2.5081
14	1970	397	23	65	69	157	0.3955	2.5287
15	1980	424	56	122	163	341	0.8042	1.2434
16	1990	1136	519	1177	755	2,451	2.1576	0.4635
17	2000	3236	49	231	100	380	0.1174	8.5158
18	2010	25483	369	655	16	1,040	0.0408	24.5029
19	2030	1773	92	140	206	438	0.2470	4.0479
20	2040	1372	43	347	96	486	0.3542	2.8230
21	2080	3182	677	996	982	2,655	0.8344	1.1985
22	2090	70	7	50	14	71	1.0143	0.9859
23	2100	3095	4	9	2	15	0.0048	206.3333
24	2102	5143	16	12	9	37	0.0072	139.0000
25	2103	5143	141	311	147	599	0.1165	8.5860
26	2140	345	36	93	55	184	0.5333	1.8750
27	2150	44	2	14	3	19	0.4318	2.3158
28	2160	2028	261	594	303	1,158	0.5710	1.7513
29	2170	770	70	177	199	446	0.5792	1.7265
30	2180	8040	320	746	374	1,440	0.1791	5.5833
31	2190	3459	15	54	38	107	0.0309	32.3271
32	2192	3459	90	118	107	315	0.0911	10.9810
33	2200	2167	80	245	147	472	0.2178	4.5911
34	2220	1265	63	151	179	393	0.3107	3.2188
35	2261	1841	43	176	82	301	0.1635	6.1163
36	2310	6387	1450	2483	1740	5,673	0.8882	1.1259
37	2320	1187	28	14	7	49	0.0413	24.2245
38	2340	878	40	86	41	167	0.1902	5.2575
39	2342	1037	74	82	66	222	0.2141	4.6712
40	2380	4612	24	12	78	114	0.0247	40.4561
41	2430	3208	42	213	80	335	0.1044	9.5761
42	2490	2202	150	411	202	763	0.3465	2.8860
43	2500	1157	50	271	77	398	0.3440	2.9070
44	2501	1157	34	307	68	409	0.3535	2.8289
45	2550	3452	72	524	63	659	0.1909	5.2382
	Σ Complexity$_{awi}$						14.5887	688.2515

Summary							
Rc	234,786	8,019	19,748	10,885	38,652	0.1646	6.0744
Rb	Not Available	Not Available	2,306	2,790	5,096		
Ra	Not Available	Not Available	2,649	2,309	4,958		

Cost	*Release C*	*Release B*	*Release A*
Des$	8,825,000	3,437,000	8,264,000
Imp$	24,116,000	9,602,000	9,051,000
Test$	9,606,000	2,900,000	6,096,000
P$	42,547,000	15,939,000	23,411,000

Figure 10. The data. ([a] Total LOC for new, modified, and affected modules. [b] Change Concentration. [c] Change Spread.)

STAGE	*Data From*	PRODUCTIVITY METRICS	INTERFERENCE METRICS	EFFECTIVENESS METRICS
Design	Component Level Design	Rc: 0.0009 *Des*CP$_p$/Des$_p$	Rc: 0.0000	Rc: 0.0000 Rc: 0.2075
Implementation	Unit Test	Rc: 0.0003 *Imp*CP$_p$/Imp$_p$	Rc: 0.0000	Rc: 0.2075
Test	Test End and Service	Rc: 0.0008 *Test*CP$_p$/Test$_p$	Rc: 0.7925	Rc: ?.???? Rb: 0.8457^x Ra: 0.8071^x Ra: 0.7398^z

	Release C	Release B	Release A
***CLD*CP/P$**	0.0002		
Modules^t	5339	4438	4394
Modules^{nma}	3076		
Complexity			
Density^t	7.2396	1.1483	1.1284
Density^{nma}	12.5657		
Dispersal^t	0.1381		
Dispersal^{nma}	0.0796		
Scope	858.9333		
Compactness	0.0012		
SCP		930^x	1185^x 1744^z

Figure 11. The measures. (^{x}After 6 quarters. ^{z}After 11 quarters. ^{t} Total modules. ^{nma} New + modified + affected modules.)

design, and this will add expense as compared with the past. A likely result is that the productivity measurements for design will show decreasing productivity for a while (how long I cannot say), and this is GOODNESS: the expected net result should be improved effectiveness and/or reduced total cost. Often, the issue is not whether particular measurements go up or down, but whether they are where we expect them to be, and if not, what understanding we have of why they are not.

4.2.2 Interference metric. *RC*CP/*CLD*$_b$CP is essentially a requirements (in)stability indicator; it shows whether the requirements changed (were augmented or reduced).

4.2.3 Effectiveness metrics. *DC*CP/*CLD*$_b$CP (zero in Figure 11) is essentially a high-level design (in) stability indicator; it shows that a poor design was produced or that the requirements were incomplete or not well specified. *CLD*$_b$CP/CP is essentially a high-level design quality/completeness indicator. It is sensitive to effects from incomplete, unclear, or changing requirements, because it does not factor out the net CHANGE-POINT count for design changes (included in *CLD*CP) from CP. Design effectiveness, *CLD*$_b$CP/CP, in Figure 11 suggests the need for considerable work to improve the effectiveness of this activity for this product. Although it is not possible to pinpoint design as the key area for concern based on the data available, it does require examination.

4.3 Implementation

This section discusses the implementation metrics included in the PIE matrix.

4.3.1 Productivity metrics. *CLD*CP/Imp$ is straightforward; it is simply the final number of outputs divided by the cost to produce them. During a project, cumulative *Imp*CP$_p$/Imp$_p$ would let us see how implementation productivity is changing in a life cycle fashion. If we have life cycle history, we can compare the current project to history as we go along, to see if it is typical or shows expected changes, and attempt to investigate and understand any atypical or unexpected results. For an organization, we can sum *Imp*CP$_p$ and sum Imp$_p$ for its various

products for a period and thus compute implementation productivity for the period.

4.3.2 Interference metric. Growth in total CHANGE-POINT count relative to CLD_bCP represents incomplete or unstable high-level design and is an interference on implementation by high-level design.

4.3.3 Effectiveness metric. Implementation effectiveness in Figure 11 suggests this as an area for concern. The data do not allow us to target just design or just implementation as the key problem area; however, obviously, the development process for this product is quite dependent on test effectiveness being extremely high.

4.4 Formal Test

This section discusses the formal test metrics included in the PIE matrix.

4.4.1 Productivity metrics. *CLD*CP/Test$ is straightforward; it is simply the final number of outputs divided by the cost to produce them. During a project, cumulative *Test*CP_p/Test$$_p$ would let us see how formal test productivity is changing in a life cycle fashion. If we have life cycle history, we can compare the current project to history as we go along to see if it is typical or shows expected changes, and attempt to investigate and understand any atypical or unexpected results. For an organization, we can sum *Test*CP_p and sum Test$$_p$ for its various products for a period and thus compute test productivity for the period.

4.4.2 Interference metrics. Clearly, software defects interfere with test productivity.

4.4.2.1 Counting FTCP. *FT*CP (ptm^{CP} + ff^{CP} from Figure 10) includes forward fits[23] and therefore gives substantial weight to problems shipped in prior releases in the effectiveness measurements for design and implementation. Forward fits need separate tracking to allow isolation of their effect from other software defect repairs.

At least two options exist for counting *FT*CP: (1) count 1 for each new, modified, deleted, and affected module due to a software defect fix; (2) count 1 only for instances that increase or decrease the module set for an AWI (1 for each module added or deleted) where the reference module set changes with each added or deleted module.

Option (2) is the purist approach. I have selected option (1) for counting both *FT*CP and *S*CP (see Section 4.4.3); this avoids a need for additional effectiveness and interference metrics. Option (1) puts heavy emphasis on software defects in the effectiveness measurements for design and implementation and in the interference measurements for test. One can think of this counting approach, (1) above, as saying CP includes changes to CHANGE-POINT implementation as well as *additional* or *deleted* CHANGE-POINTs.[24]

4.4.3 Effectiveness metrics. CP/(*S*CP + CP) reflects growth in total CHANGE-POINT count after formal test completion. The theory is that test should validate function *as well as* find design and implementation errors. *S*CP is counted in the same manner as *FT*CP (see Section 4.4.2.1.); this gives significant weight to problems found after ship.

Reducing *S*CP by increasing *FT*CP, i.e., finding more software defects during formal test, will increase test effectiveness, but this will *decrease* implementation effectiveness and *decrease* design effectiveness (CLD_bCP/CP). For example, reducing *S*CP for Ra (see Figure 11) by a factor of 10 increases CP, thus reducing the effectiveness value for implementation.

It is important to note that CP/(*S*CP + CP) addresses *defect removal*, not *defect prevention*. This means that simple comparisons of test effectiveness can be misleading: avoiding high test interference is important, so considering *FT*CP/CP is necessary. This suggests that trade-offs between design, implementation, and test effectiveness are a necessary part of process improvement.

Without *CLD*CP data for Ra or Rb, it is difficult to reach any firm conclusions, but reviewing Figure 11 and using ptm^{CP} + ff^{CP} as CP suggests that test effectiveness did not improve significantly, because the computed values are worst case.

I do not have *S*CP data for Rc. However, we can expect that if test effectiveness is not at least 96%, then *S*CP for Rc will exceed *S*CP for Ra and Rb.

[23] A software defect repair included in an *enhancement* project (new release) because of a *service* software defect repair in a released product (predecessor release).

[24] For zero-defect software (i.e., *FT*CP = 0 and *S*CP = 0), both approaches result in the same CP count.

Table 1. Key Differences for CP, LOC, and FP

Attribute	CHANGE-POINTs (CP Measure)	Shipped, new + modified-LOC	Function Points
Continuous count	Yes	Yes	No
Bounded count	No	No	Yes
Scale	Absolute	Absolute	Views vary
Available before ship	Yes	No	Yes
Zero condition	Comment text only changes	No new or modified LOC	User function types unchanged (UFP is zero)
Imposes technology constraints	No	No, but redefinition of LOC may be necessary	Yes
Automated counting	Some manual effort required	Easy	Difficult

4.5 Overall Metrics

This section discusses metrics that encompass all the activities included in the PIE matrix.

4.5.1 Project productiveness. *CLD*CP/P$ is straightforward; it is simply the final number of outputs divided by the cost to produce them.

4.5.2 Periodic productivity. The theory here is that we can add *Des*CP_p, *Imp*CP_p, and *Test*CP_p for a given period. This is not double counting: for many projects in a given period, these represent different CHANGE-POINTs; even in a zero-interference world, these are additive from a process point of view. To help clarify the latter point, consider a message sent from point A to point B to point C. From an end-product point of view, one message went from A to C. From a process point of view, one message went from A to B, and another from B to C (two messages total); it makes no difference that B did not modify the message sent from A before sending it to C.

So, by totaling *Des*CP_p, *Imp*CP_p, and *Test*CP_p for the period (call this PCP_p: process net CHANGE-POINT count for the period), then dividing by $P\$_p$ for the period, we have *process* productivity for the period. Of course, there is leakage period to period, but over many periods this should smooth out.

5. CONCLUSION

This article has presented a conceptual approach to productivity measurement at a higher level than the individual development activity (Design, Implementation, and Test). It has described the concept of CHANGE-POINTs as a common output that permits both a combined and individual measurement of productivity for all three development activities. Admittedly, this metric does not encompass important activities, such as publications development and the many indirect activities required in a software development project. The focus is on the primary elements of direct expense. Inclusion of the other dimensions requires higher level conceptualizations.

Although this article has not attempted to define an implementation process or a set of procedures, clearly, any implementation of the CHANGE-POINT approach will need configuration management tools (for example, the IMS TU or the DB2 line item workbook).

The perspective provided by the PIE matrix should allow management to balance effectiveness with efficiency and emphasize one or the other at the appropriate times for the appropriate activity.

The CHANGE-POINT approach does not attempt to prescribe the design, implementation, or test processes or technology to use, and, in this sense, is process and technology independent; the CHANGE-POINT approach requires only that the project development process used identify where and when to perform the CHANGE-POINT counts for the design, implementation, and test techniques used. Table 1 is a summary of key differences between the CP, LOC, and FP measures.

I hope this article has presented enough regarding CHANGE-POINTs to encourage implementation and test of the concepts provided.

ACKNOWLEDGMENTS

Each member of the STL Productivity Task Force contributed to my thinking on this subject. Others who played a significant role include Joel D. Aron, Marilyn Bohl, Mance Drummond, Walt Fant, Janet Gregory, John Hardy, Ed Lassettre, Dr. Peter Lazarus, Baron McDonald, Horst Remus, and Vern Watts. I owe special thanks to Gary Davidson for supplying most of the data in Figures 10 and 11, and especially to Dr. Ursula Richter, whose criticism and encouragement have been invaluable.

REFERENCES

Albrecht, A. J., and Gaffney, J. E., Jr., Software Function, Source Lines of Code and Development Effort Prediction: A Software Science Validation, *IEEE Trans. Software Eng.* SE-9, 639–648 (1983).

Arthur, L. J., *Measuring Programmer Productivity and Software Quality*, John Wiley & Sons, 1985.

Bailey, J. W., and Basili, V. R., A Meta-Model for Software Development Resource Expenditures, in *Proceedings of the Fifth International Conference on Software Engineering*, 1981, pp. 107–116. [Reprinted in *Software State-of-the-Art: Selected Papers*. (T. DeMarco and T. Lister, eds.), Dorset House, 1990.]

Baker, A. L., Bieman, J. M., Fenton, N., Gustafson, D. A., Melton, A., and Whitty, R., A Philosophy for Software Measurement, *J. Syst. Software* 12, 227–281 (1990).

Brooks, F. P., No Silver Bullet: Essence and Accidents of Software Engineering, *Info. Proc.* (1986). [Reprinted in *Software State-of-the-Art: Selected Papers*. (T. DeMarco and T. Lister, eds.), Dorset House, 1990.]

Campbell, N. R., Measurement in *What is Science*? Dover Publications, 1921 [Reprinted in *The World of Mathematics* (J. R. Newman, ed.), Simon and Schuster, New York, 1956.]

Chillarege, R., Bhandari, I., Chaar, J., Halliday, M., Moebus, D., Ray, B., and Wong, M.-Y., Orthogonal Defect Classification—A Concept for In-Process Measurements, *IEEE Trans. Software Eng.* 18, 943–956 (1992).

DeMillo, R. A., and Lipton, R. J., Software project forecasting, in *Software Metrics: An Analysis and Evaluation* (A. Perlis, F. Sayward, and M. Shaw, eds.), The MIT Press, 1981.

Fenton, N. E., *Software Metrics: A Rigorous Approach*, Chapman & Hall, 1991.

Fenton, N., and Melton, A., Deriving Structurally Based Software Measures, *J. Syst. Software* 12, 177–187 (1990).

Glass, R. L., *Building Quality Software*, Prentice-Hall, 1992.

GPD Programming Development Process Handbook, TR 03.136–03.143, IBM.

Grady, R. B., and Caswell, D. L., *Software Metrics: Establishing A Company-Wide Program*, Prentice-Hall, 1987.

Heyel, C., *The Encyclopedia of Management*, 3rd edition, Van Nostrand Reinhold, 1982.

Jones, C., *Programming Productivity*, McGraw-Hill, 1986.

Jones, C., *Applied Software Measurement*, McGraw-Hill, 1991.

Kemerer, C. F., and Porter, B. S., Improving the Reliability of Function Point Measurement: An Empirical Study, *IEEE Trans. Software Eng.* 18, 1011–1024 (1992).

Nicholas, J. M., *Managing Business and Engineering Projects: Concepts and Implementation*, Prentice-Hall, 1990.

Packer, M. B., Measuring the Intangible in Productivity, *Technol. Rev.* 48–57 (1983).

Parnas, D. L., Software Aspects of Strategic Defense Systems, *Commun. ACM* 28, 1326–1335 (1985).

Programming Process Architecture, Version 2.1, 2nd edition, ZZ27-1989-1, IBM, 1986.

Radice, R. A., Roth, N. K., O'Hara, A. C., Jr., and Ciarfella, W. A., A Programming Process Architecture, *IBM Syst. J.* 24, 79–90 (1985).

Rask, R., Laamanen, P., and Lyytinen, K., Simulation and Comparison of Albrecht's Function Point and DeMarco's Function Bang Metrics in a CASE Environment, *IEEE Trans. Software Eng.* 19, 661–671 (1993).

Scudder, R. A., and Kucic, A. R., Productivity Measures for Information Systems. *Info. Manag.* 20, 343–354 (1991).

Stroustrup, B., *The C++ Programming Language*, 2nd edition, Addison-Wesley, 1991.

Symons, C. R., Function Point Analysis: Difficulties and Improvements, *IEEE Trans. Software Eng.* 14, 2–11 (1988). [Reprinted in *Software State-of-the-Art: Selected Papers* (T. DeMarco and T. Lister, eds.), Dorset House, 1990.]

Yu, W. D., Smith, D. P., and Huang, S. T., Software Productivity Measurements *AT & T Tech. J.* 69, 110–120 (1990).

APPENDIX 1: The Paradox of LOC

Assume (1) the hypothetical projects in Jones (1986) begin 1/1/1990; (2) the number of developers assigned to each task (requirements, design, coding, documentation, and integration/testing) for the Assembler project is no greater than the number assigned to the same task for the APL project; and (3) available time to work on the project is the same for developers performing the same tasks for each project. Given the seven-times ratio between person-months for coding and integration/testing for the APL versus the Assembler project, we can conclude that the calendar time to complete these two projects will differ. For simplicity, assume the APL project completes 12/31/1990; thus, the Assembler project takes longer than one year to complete. Measured by use of LOC produced, 1990 production for the APL project is 10,000 LOC (productivity is 125 LOC per person-month). Measured by use of LOC produced, 1990 production for the Assembler project is unknown, as is productivity, because the number of lines of source code produced is only specified as of project completion.

Given the assumptions above, we can conclude that the Assembler project, as of 12/31/1990, will have expended for each task, at most, the same number of person-months as needed for the same task for the APL project. Assuming the person-months expended are equal,[1A] any difference in 1990 production (and productivity) is because of differences in the number of lines of source code produced in 1990. The number of lines of source code produced in 1990 will be a function of "coding speed" (Jones, 1986). Therefore, any difference in 1990 production (and productivity) will not be be-

[1A] There is reason to believe that integration/testing might be less for the Assembler project, but this does not affect the fundamental conclusion.

cause of identical levels of fixed or inelastic costs (requirements, design, and documentation) as suggested by Jones (1986), but will be due to variable cost activity differences (coding or integration/testing), and, as Jones (1986) noted, "high-level languages actually do improve coding speed."[2A]

APPENDIX 2: Using Function Points in the PIE Matrix

Consider two projects: (1) our C compiler manufacturer has added a switch that will set up register linkage between modules, i.e., cause parameters to be passed via registers instead of storage, so for speed improvement we want to recompile our application using this new switch; (2) because of new tax laws, we must increase the top federal income tax rate and the salary limit for FICA withholding in our payroll application.

For both projects, UFP will be zero, so productivity will be zero. Use of the CHANGE-POINT approach for Project 1, even considered as only a test exercise, will generate nonzero CLD_bCP (and thus nonzero CLDCP). This is because to implement the AWI (expressed as "register linkage between modules" {the *result*} by recompiling the appropriate application modules using the new compiler switch {the *set of actions*}), there are many affected (recompiled) modules. Note that while physical source text changes are not required, nevertheless the set of effects needed in/from the program units does change: if a *called* module is recompiled with the new switch but a *calling* module is not recompiled with the new switch, then the software will not operate correctly. These changes would be more obvious if the example were an application written in assembler language, and thus the required linkages had to be coded by hand; or the compiler required a specific keyword in the function prototypes; or the compiler required the use of a #pragma statement in affected source modules.

Project 2 requires source text changes and will generate a nonzero CHANGE-POINT count, because the AWI (expressed as "update the payroll application to reflect the new tax laws") affects at least two definitions and perhaps several modules. (If we consider a case where the program is driven by tax rates in an external file, then, depending on the extent of testing of the original AWI, the relevant CHANGE-POINT count will be less than or equal to the CHANGE-POINT count for the original AWI.)

[2A]"The assembler example ... proceeded at ... 870 lines per month, while the ... APL [example] ... proceeded at ... 1,000 lines per month" (Jones, 1986).

APPENDIX 3: Glossary

Some of the terms in this paper are as follows.

Design. Product level design and component level design. This usage varies from *Programming Process Architecture* (1986) and Radice et al. (1985), but is the terminology of the STL Productivity Task Force.

Effectiveness. Extent to which requirements for an activity/area are met. An effectiveness metric attempts to quantify attainment of an area goal (e.g., "zero defects"). "Goodness" may mean an increase or decrease in the measurement, whichever is most convenient (and must be stated or self-evident). Ideally, when productivity is increasing, the effectiveness measurements should be going in the direction of "goodness." Alternatively, improvement in the effectiveness measurements *may* be combined with stable or decreasing productivity, but this should come about as a result of an explicit trade-off and not be "discovered." There are many facets to effectiveness, and in the PIE matrix I have chosen to identify metrics relating to quality/defects, i.e., reliability (Glass, 1992); there is no intent to exclude other possible metrics for effectiveness.

Formal test. See *Test.*

High-level design. See *Design.*

Implementation. Module level design through unit test. This usage varies from *Programming Process Architecture* (1986) and Radice et al. (1985), but is the terminology of the STL Productivity Task Force.

Interference. An effect due to *not* performing a task/activity with "zero defects." Normally, this affects the productivity of another group, but may also affect the performer's productivity. In general, the productivity measurement for the affected group would have been better if the affecting group had performed its task/activity with zero defects.

Productivity. The ratio between what is produced and what is consumed to produce it.

PTM. A program trouble memorandum is the means for reporting and recording defects found during formal test.

Test. The testing family in *Programming Process Architecture* (1986) and Radice et al. (1985).