Software Quality Journal, 11, 265–281, 2003

COCOMO-Based Effort Estimation for Iterative and Incremental Software Development

ODDUR BENEDIKTSSON
University of Iceland, Computer Science Division, Reykjaví, Iceland

DARREN DALCHER
Middlesex University, London N14 4YZ, UK

KARL REED
Department of Computer Science and Computer Engineering, La Trobe University, Australia

MARK WOODMAN
Middlesex University, London N14 4YZ, UK

Abstract. Incremental software development and delivery have been used in software projects in many ways for many years. Justifications for incremental approaches include risk amelioration, the management of evolving requirements, and end-user involvement. Incremental development, including iterative, incremental delivery, has become a norm in many sectors. However, there has been little work on modelling the effort in such development and hence a dearth of comparative analyses of cost models for incremental development/delivery. We attempt to rectify this by proposing a COCOMO-style effort model for incremental development/delivery and explore the relationship between effort and the number of increments, thereby providing new insights into the economic impact of incremental approaches to software projects.

Keywords: software effort estimation, incremental software development, software project management, COCOMO-based estimation, effort in increments

1. Introduction

The study of software processes is one of the most contentious areas in software engineering because it is inherently difficult to study software development in order to directly validate or refute the large number of processes that have been proposed. Furthermore, the process models that dominate what is taken to be conventional wisdom at any time have evolved substantially over the last forty years (e.g., as described in (Zahran, 1997)), making the study of any particular set of them time-dependent. In particular, the comparative study of the effort, and hence the cost, involved in different software processes is problematic. A number of mathematical models have been used with a view to predicting effort required for projects. Of these, Boehm's COCOMO models (Boehm, 1981, 2000) have become dominant.

In recent years incremental and iterative approaches to the production of software have become popular, however, we are unaware of comparative effort models which would assist with their planning and adoption. Incremental approaches encompass various ways of producing a sequence of parts of a system, while iterative approaches involve a diversity of ways of producing parts of a system, trying them out, and feed-

ing back user experience to the production of new or revised parts. Although notionally distinct (Goldberg, 1995) these approaches are very much interrelated: iteration depends on the delivery of parts of a system and incremental delivery is inherently "iterative," in that user feedback is used in producing later parts. Despite the increasing popularity of incremental approaches, there has been no systematic examination of their project estimating implications. All that is available is an example from (Boehm et al., 2000) of a comparative calculation of efforts. In principle, according to any standard estimating technique, the effort (and hence the cost) of adopting such an approach will be different from that of a monolithic development. This paper provides a general analysis using the COCOMO II model.

Assuming the model is applicable to the projects concerned, our analysis enables project planners to determine the benefits in terms of development effort of choosing either a monolithic project or either of the two incremental modes of production. More specifically, it offers a mechanism for reasoning about the relationship between the total effort and the number of increments, thereby enabling consideration of the "optimal" number of increments.

We proceed by establishing definitions that encompass the notions our model represents. Having established our vocabulary we explore these notions, beginning with relevant background and motivations for incremental and iterative software production and concluding with a statement of some interesting research questions in the area of the economics of incremental approaches. Next, we propose a simple variety of incremental delivery which, while not necessarily novel, focuses on a specific aspect of the process which allows it to be differentiated from other similar process models. Our particular form of incremental delivery is deliberately kept simple to demonstrate the feasibility of effort modelling and prediction for the class of incremental software processes. We then present an analysis based upon a simple estimating model common to a number of estimation techniques. This shows that there are indeed cases where the total effort will be greater, or lesser than, that for a monolithic project, assuming a continuous incremental model of development. We then examine the implications of a model in which a (substantial) core of a system is developed first, and a series of increments added to it. Finally, we discuss the implications of our findings for project planners, and suggest some future areas for research.

2. Incremental and iterative software development

2.1. Definitions

The most important differences in the process models we are concerned with are between incremental development and delivery. We adopt the definitions proposed by (Graham, 1989), thus:

- *Increment:* A self-contained functional unit of software with all supporting material such as requirements and design documentation, user manuals and training.
- *Incremental development:* The development of a system in a series of increments throughout the project timescale.

- *Incremental delivery:* The delivery of increments to the customer/users at intervals throughout the project timescale.

Arguably, incremental delivery is more important in as much as it is externally observable and can be monitored externally. Also, it implicitly obliges developers to collect and deal with feedback from the customer/users to inform the development of later increments. This is why there is confusion with "iterative development," which is actually about prioritizing a need to rework parts of a system, whereas incremental development and delivery are about prioritizing a need to partition a system so that the parts may be produced at different times or rates (Goldberg, 1995). Hence, we adopt Goldberg and Rubin's definition:

- *Iterative development:* A strategy for developing systems that allows for controlled reworking of part of a system to remove mistakes or make improvements based on user feedback.

Note that this term is badly named; it should, of course, be iterative delivery. However, leaving the misnomer allows us to distinguish the notion from the more prescriptive evolutionary delivery as described by (Graham, 1989). She uses that term for process models that involve each increment in its own whole lifecycle.

Whatever the particular version of incremental delivery/development and its priorities, from a project management point of view attention must be paid to the number and size of increments, their use over time and the effort needed to partition and administer increments.

Incremental software delivery entails the planning, development and release into actual use (service) of software products in increments, where each additional increment adds operational functionality, or capability, not available in previous releases. The underpinning assumption is that it is possible to isolate useable parts that can be developed, tested and implemented and delivered independently over time. When an increment is put into service, information on its efficacy or deficiencies can be gathered and used to guide the development of other parts, hence making the process iterative. Some projects warrant partitioning but do no need piece-wise release, use and feedback to ameliorate risk, improve quality, etc. We will distinguish between those cases where increments are developed and used within a project as a means of producing a whole by describing them as *endogenous*, and those where the increments are delivered to a customer, by describing them as *exogenous*.

2.2. *Brief history and rationale*

Incremental approaches emphasize phased development and gradual build-up of functionality by offering a series of linked mini-projects (Dalcher, 2002). The approach is underpinned by the assumption that it is possible to isolate meaningful subsets that can be developed, tested and implemented independently. Delivery of increments can thus be staggered as calendar time progresses.

As already stated, despite incremental delivery's current popularity, the approach is not new. Allusions to it can be found in quite early writing. Indeed, Bennington, in discussing the SAGE "mega-project" from the 1950s, contended that incremental

delivery would have been better suited to the project while Brooks commented on the need to plan to throw away one version of the production version (Bennington, 1995; Brooks, 1987). (Throw-away prototyping can be regarded as a form of incremental delivery.) Mills contended that the best way to produce software was to design a main program and to gradually develop the rest of the sub-programs level by level (Mills, 1971), with each module added progressively while paying attention to testing and integration (Mills, 1976). In 1980, Mills reported the successful completion of the US Navy LAMPS project using incremental development to deliver 7 million lines of code in 45 increments (Mills, 1980). The 200 person-year project was completed in four years with monthly deliveries completed on time and within budget.

Incremental (iterative) delivery has attracted significant attention in the last ten years. Unlike the classic "monolithic" or "big-bang" approaches that appear to be implied by the generally accepted version of the Waterfall model (compare, for example (Royce, 1970) and the description in standard software engineering text-books), an incremental approach is intended to create steadily enhanced versions of a system. Many variations of the approach are identifiable (Graham, 1989) but there are two main classes that appear to dominate. The first consists of a series of incremental deliveries, of small size, produced frequently throughout a project. The so-called agile process models, typified by Extreme Programming (Beck, 2000) and Rapid Application Development and the time-boxing notion (Stapleton, 1997), are modern examples of this approach to incremental delivery. In the second, one identifies some core functionality that could be developed and placed in service, and a series of subsequent "increments" which complete the project by adding functionality or otherwise improving the product's properties. The primary objectives for these strategies, which have been in use since the mid 1960s, is a combination of risk amelioration, rapid feedback, user involvement, reduced complexity and simplified management. Moreover, they also provides a means of dealing with P-type and E-type programs (Lehman, 2000, 2001) or with "Domain Dependency" (Giddings, 1984; Dalcher, 2002).

Incremental development has long been recognized as an effective way to get the user interested and actively involved in the development of the system in order to ensure a closer fit to real needs and a greater level of user satisfaction (Mills, 1976, 1980; Berrisford and Wetherbe, 1979; Brooks, 1987; Gilb, 1988). Indeed, the staggered release policy allows for learning and feedback to alter some of the customer requirements in subsequent versions thus incorporating user experience into the refined product. The gradual introduction provides time for the client to adjust to the system while also allowing for adjustment of expectations and responsiveness to change. Depending upon the nature of the project partitioning, and the accuracy of the initial determination of customer requirements, subsequent versions may incorporate those customer requirements that emerge from experience with prior increments. Putting this another way, problems arising in such cases are reduced in the context of exogenous incremental delivery, since the staggering of deliveries supports substantial learning and feedback within a project team which allows some of the difficulties arising from "E" and "near E" type projects to be dealt with. Equally importantly, the team will gain general expertise in the application domain, and be better able to both analyse the (evolving) client need, and implement it more efficiently (Podalsky, 1977).

Incremental development is also seen as one of the basic ways to enhance risk management and reduce the risk loss potential (Boehm, 1981; Boehm et al., 2000; Krutchen, 2000). Moreover, in keeping with the philosophy of risk reduction, the exposure to risk is limited to shorter time intervals (during which additional knowledge and confidence are gained, thereby reducing the inherent uncertainty (Dalcher, 2002)).

Technical benefits from the adoption of incremental development are likely to include earlier resolution of implementation problems and design errors, reduced rework, increased reliability and easier maintenance (Graham, 1992). From the client perspective, benefits may also encompass early (and on-going) delivery of functionality, enhanced confidence in the developers' ability to deliver the right system and greater involvement leading to enhanced familiarity. Additional benefits come from the improved mode of management. These were summarised in (Dalcher, 2002) and include the controlled release of funds (i.e., responsive cost management), early return on investment, improved loading profiles, smaller reliance on external and specialist personnel, stage-limited commitment, lower dependency on external events and deliverables, enhanced visibility, improved feedback, reduced uncertainty, better-informed decision making, better-understood trade-offs and the achievement of warranted performance levels. More crucially, perhaps, incremental approaches offer the potential for an economy of scale, not least in terms of addressing the inherent complexity.

2.3. Increments and iterations

Incremental approaches have been widely used as a within-project (endogenous) basis for compiler development, in the form of bootstrapping. In a more tightly coupled form, incremental development can be found as natural feature of actual software development in (Hess, 1996). While the form of incremental delivery may be indistinct, in that usable systems need not be delivered, the likes of extreme programming make use of a form of endogenous incremental delivery to obtain closure on the user's needs. In fact, detailed studies by Hess have shown that iteration is an integral part of software development (ibid).

We need to make it clear that we are discussing a particular scenario, which differs from the "software release" process associated with (large-scale) software products. Often, the life-cycle of a particular product is characterised by the delivery of additional functionality and the improvement of quality over a series of "releases." This phenomena has been identified in the early 1960s (see, for example, the successive releases of OS/360 (Lehman, 1969)). From this perspective, it seems clear that this is a fundamental characteristic of software development (Lehman, 2000, 2001). Incremental delivery can be considered a special case of this process, in which an intentionally incomplete but useful product core is delivered, and subsequently completed by the delivery of successive increments. In the classic "product release" cycle, the first release would be intended to be fully functional (complete), with only minor parts missing or requiring improvement. Meaningful adoption of this approach requires techniques for identifying an acceptable core, and "increments." The literature does not offer a lot of guidance here, however, techniques that may prove helpful include those described in operational testing strategies (Musa, 1993), and those implied in the Clean Room

process (Mills et al., 1987). Before concluding the general discussion of incremental development we re-evoke the distinctions introduced earlier in terms of how they address the notion of iterations and increments.

2.3.1. Iterative development In this case, we are assuming that a moderately large number of (roughly equal) increments are produced, and that there is no "major core" in the sense of some partial delivery of functionality. Extreme programming operates on the basis of delivering that functionality which can be delivered in fixed time (say six weeks), and then by augmenting it by fixed increments. The choice of initial delivery could be any functionality, no matter how small, that can be demonstrated to the client. Closely involving the client is expected to give extensive capacity to deal with requirements-creep (Capers Jones, 1998), and hence significantly reduce the probability of a project being delivered which fails to meet user expectations. In addition, a process such as this permits a truncated development cycle, and as such has capacity to meet time-to-market constraints.

However, there is evidence that these processes can have problems when scaled up where a defined objective of a large scale is being met (see (Boehm, 2002; Elssamadisy and Schalliol, 2002)). Moreover, "conceptual integrity" may also suffer, as there is little motivation to deal with scalability, extensibility, portability or reusability beyond what is called for in limited requirements (Booch, 1996).

2.3.2. Incremental delivery Conceptually, "incremental" delivery has the connotations of some definite process of production in which a client is provided with a useable system which meets a sub-set of functional and non-functional requirements as are understood at some point in time (Constantine and Yourdon, 1978). However, as remarked in our introduction, there has been a wide range of "incremental" development practices in use over the last forty years. In practice, we can identify two extremes in the "incremental" spectrum for classification purposes, but first we need to make a distinction relating to code production at the developer level. Software production can be regarded as series of "deliveries" of code considered by its developer to be ready for inclusion in some "build," i.e., executable integration of code-modules. We could (as seems to be done in the Time To Market literature (Cusumano and Yoffle, 1999)) take the view that the personal approaches used by the developers are not of interest. Hence, it may be that an individual group of developers may use an "incremental" approach, in that their coding practices resemble continuous maintenance. As already remarked, we call this "endogenous incremental delivery." The "incremental" aspects of the development at this level are not visible to the customers unless they are involved in the process (as they may be in an extreme programming project).

At the other end of the spectrum, the client may take delivery of series of versions of a complete version of the system, which realise major variations in the requirements, either functional or non-functional. This is the product release process adopted by software suppliers, and cannot be considered to be incremental delivery, since complete systems are being delivered, as we have already pointed out.

3. The relative economics of incremental processes

3.1. COCOMO and the BNEF

Whilst incremental approaches would appear to solve many problems and simplify some management tasks they also introduce new issues that need to be addressed by project managers and developers, particularly in terms of planning and controlling the effort. As stated earlier, there has been little work examining the impact of incremental delivery on project economics. The impact (in this case, negative) of the non-linearity on breaking a project up is shown by an example in (Boehm et al., 2000), while (Elbaum and Munson, 1998) examine the increase in complexity due to code-churn, as a project iterates, they do not consider the economic implications. Given that most estimating procedures are non-linear with size, being of the general COCOMO form, with exponents greater than unity, it is possible that there will be reductions in development effort due to the "incrementalisation" and possible reductions in complexity of the increments due to the increased knowledge of the development team.

Within these contexts, a number of questions need to be investigated. For example:

- How does the total development effort of incremental development relate to a similar, "integrated" development?
- In what way does the total effort depend on the number of increments?
- Is there an optimal number of increments?

We propose using the COCOMO II effort estimation model as the basis for exploring some of these questions by utilising the quantitative analytic framework for evaluating software technologies and their economic impacts (Clark et al., 1998; Royce, 1998).

A given development project has estimated the overall product size S and specified the overall effort adjustment factor A and the scale exponent E. The effort PM is a number computed by equation (1a). We will use this number as a normalising factor and name it the *Boehm Normalising Effort* (BNEF). COCOMO II (Boehm et al., 2000) has the general form

$$PM = A \times Size^E \times \prod_{i=1}^{n} EM_i, \tag{1a}$$

where the exponent, E is given by:

$$E = B + 0.01 \times \sum_{j=1}^{5} SF_j, \tag{1b}$$

where $B = 0.91$, and $0.0 < \sum_{j=1}^{5} SF_j < 3.16$, so that E is $0.91 < E < 1.226$. In what follows, we will work with a simplified version of the equations, in which E and the expression $A \times \prod_{i=1}^{n} EM_i$ is treated as a single variables. We use $a = A \times \prod_{i=1}^{n} EM_i$ and $y = PM$ and $S = Size$ in what follows so that we have:

$$y = aS^E. \tag{1c}$$

3.2. Incremental development

Now suppose that the development cycle is partitioned into work on n increments each of nominal size s_i and that:

$$S = \sum_i s_i. \tag{2}$$

The effective size x_i of each increment for effort estimation purposes is taken to be

$$x_i = (1 + c_i)s_i, \tag{3}$$

where the parameter c reflects the overhead in producing increments, which we term the *breakage* following Boehm's vocabulary (Boehm et al., 2000). The added work due to the breakage reflects the fact that as a new increment is added to a release then some glue-code will be need, some ideas and features may be abandoned, and some code may need to be thrown away. We talk about a breakage of 15% if c has a value of 0.15. In order to simplify the discussion, it is assumed that all the code needs to be written from start (so reuse is not taken into consideration).

The development effort y_i for an increment of size x_i is taken to be

$$y_i = a_i x_i^{E_i}. \tag{4}$$

Each additional increment has to be incorporated into the architecture defined for the whole project. The architecture for the solution needs to be envisioned initially to allow for the breakdown structure of the increments. This requires an upfront effort which is over and above the effort needed to develop each increment. The initial effort can be of considerable magnitude, especially if domain architecting and reuse considerations are taken into account. The initial effort is assumed to depend on *BNEF* as well as n, the number of increments.

The total *incremental development effort* is taken to be the sum of initial effort plus the effort of developing the n increments. We postulate that the total effort y_T can be expressed as:

$$y_T = d_n a S^E + \sum_i a_i x_i^{E_i}, \tag{5}$$

where the parameter d_n gives the fraction of the *BNEF* effort needed for the initial work. We talk about 10% initial effort needed if d_n has a value of 0.1.

Equations (3) and (5) give

$$y_T = d_n a S^E + \sum_i a_i (1 + c_i)^{E_i} s_i^{E_i}. \tag{6}$$

It is of interest to compute ratio r of the incremental effort and *BNEF*, i.e., y_T/y from equations (6) and (1c):

$$r = d_n + \sum_i (a_i/a)(1 + c_i)^{E_i} \left(s_i^{E_i}/S^E\right). \tag{7}$$

The ratio r can be named the *incremental effort ratio*.

3.3. *Equal size increments*

In order to get an indication of the effect of incremental development we make the following simplifying assumptions: The increments are all of equal size, i.e., $s_i = x/n$. The effort adjustment factors are all equal $a_i = a$ and the scale factors are all equal $E_i = E$. (Note that the so-called "time-boxed" incremental development is centred around constant time frame for each incremental delivery, say twenty working days. In this case we can make the approximation that the increments are of equal size and with the same development team throughout so that the adjustment and scale factors are constant.) We then get

$$r = d_n + n\left(\frac{1+c}{n}\right)^E. \tag{8}$$

The incremental effort ratio r is thus independent of a (the effort adjustment factor) as well as S (the product size).

3.4. *Initial core deliverable*

Here we assume that some substantial amount of core functionality is delivered initially, and a series of equal increments follow. That is, we assume that $s_1 = k \times S$ is the core deliverable and that $s_2 = \cdots = s_n$ are equal, and a fraction k of the functionality to be delivered in the core,

$$s_i = \frac{(1-k)S}{n-1}, \quad i = 2, \ldots, n. \tag{9}$$

Substituting in equation (6), we now designate the total effort (with an initial core delivery) as z_c and get

$$z_c = d_n a S^E + a\big[(1+c_1)s_1\big]^E + \sum_{i=2}^{n} a\big[(1+c_i)s_i\big]^E. \tag{10}$$

Substituting for s_i,

$$z_c = d_n a S^E + a(1+c_1)^E (kS)^E + \sum_{i=2}^{n} a\left[\frac{(1+c_i)(1-k)S}{n-1}\right]^E. \tag{11}$$

And, using our earlier assumptions regarding c_i being constant, we get

$$z_c = d_n a S^E + a(1+c)^E (kS)^E + (n-1)aS^E\left[\frac{(1+c)(1-k)}{n-1}\right]^E \tag{12}$$

and the incremental effort ratio for this case becomes:

$$r_c = d_n + (1+c)^E k^E + (n-1)\left[\frac{(1+c)(1-k)}{n-1}\right]^E. \tag{13}$$

Note that this reduces to equation (9) for the case $k = 1/n$.

3.5. *Impact of breakage and the scale factors*

Additional restrictions and assumptions need to be introduced in order to compute sample results: as we are talking about incremental development, we can restrict the number of increments, n to lie in a range between 2 and 100. Suppose that d_n is a linear function of n. For $n = 2$ it assumes the value e (say 5%) and for $n = 100$ it assumes the value f (say 15%). Then we can write d_n as:

$$d_n = (f - e)(n - 2)/98 + e. \tag{14}$$

In the original COCOMO model Boehm uses three classes of project context. They are termed organic, semidetached and embedded and relate respectively to the scale factors b of 1.05, 1.12 and 1.20 (Boehm, 1981) and correspond directly to a set of suitable values for E. We will use these values in the sample computations below as representative scale factors. Tables 1, 2, and 3 (in section 4.1) show the effort ratio r as computed with equation (8) for a range of n values and the three above stated scale factors. The initial effort factor d_n is fixed at 5% for $n = 2$ and 15% for $n = 100$, i.e., $e = 0.05$ and $f = 0.15$.

Values for a *breakage* (c) are difficult to find. Royce (1998) reports a re-work factor of between 0.05 and 0.15 in a study of Ada evolutionary projects. While his Rework Ratio is not identical in concept to our *breakage* we argue that it provides an indication of the maximum amount of inflation that should be expected/tolerated in such case. To ensure that worse-case analysis can be carried out, we include figures covering the full range between 5 and 30% for the breakage value c (i.e., $0.05 \leqslant c \leqslant 0.3$). Note that (Cusumano and Selby, 1995) reported that features may change by over 30% as a direct result of learning during a single iteration.

4. Decision processes for incremental processes

4.1. *Incremental delivery—the iterative approach*

In the incremental delivery/iterative development approach, Tables 1–3 yield the following results. If the project properties are such that E is at a maximum, the effort under an iterative process is significantly less than for the monolithic equivalent when the number of iterations is 20 or more as seen in Table 3. This is true for all chosen values of c. In fact, the number of iterations at which a saving occurs can be as few is in the range three to fifteen. Further, the savings, even for large values of the breakage

Table 1. The incremental effort ratio r for $E = 1.05$

$n\backslash c$	0.05	0.1	0.15	0.2	0.25	0.3
2	1.07	1.12	1.17	1.22	1.27	1.32
20	0.97	1.02	1.07	1.11	1.16	1.20
40	0.96	1.01	1.05	1.10	1.14	1.18
60	0.97	1.01	1.05	1.10	1.14	1.18
80	0.98	1.02	1.06	1.10	1.14	1.19
100	0.99	1.03	1.07	1.11	1.15	1.20

Table 2. The incremental effort ratio r for $E = 1.12$

$n\backslash c$	0.05	0.1	0.15	0.2	0.25	0.3
2	1.02	1.07	1.13	1.18	1.23	1.28
20	0.81	0.85	0.88	0.92	0.96	1.00
40	0.77	0.80	0.84	0.88	0.91	0.95
60	0.76	0.79	0.82	0.86	0.89	0.93
80	0.75	0.79	0.82	0.85	0.89	0.92
100	0.76	0.79	0.82	0.86	0.89	0.92

Table 3. The incremental effort ratio r for $E = 1.2$

$n\backslash c$	0.05	0.1	0.15	0.2	0.25	0.3
2	0.97	1.03	1.08	1.13	1.19	1.24
20	0.65	0.68	0.72	0.75	0.79	0.82
40	0.60	0.62	0.65	0.68	0.71	0.74
60	0.58	0.60	0.63	0.66	0.69	0.71
80	0.57	0.60	0.62	0.65	0.67	0.70
100	0.57	0.60	0.62	0.65	0.67	0.70

constant are substantial, equaling or exceeding 25%, irrespective of breakage. At the other end of the scale, for a value of E of 1.05, the achievable savings are minimal, and in fact, the iterative approach is likely to have a higher cost as seen in Table 1. As we point out in our conclusions, these results provide a basis for selecting projects for "iterative/non-iterative" development.

Tables 1–3 show that $r > 1$ for "small" n values and $r < 1$ for "large" n values indicating economy in small increments (large n). As the scale factor E increases the break-even point (where $r = 1$) sets in at lower n.

4.2. Incremental delivery—core + increments

Tables 4–8 repeat the calculations for incremental delivery involving an initial core deliverable followed by increments as presented in equation (13).

The results in this case are not dissimilar to these for pure iterative development. For $k = 0.1$, the savings vary by only a few percent from the purely iterative case. However, as the fraction in the core delivery rises, the table show that it is only possible to make a saving by keeping the breakage constant small, however, the project properties must be such that the exponent E is high. In the case where 30% of the functionality is delivered in the core, comparing Tables 3 and 7 ($E = 1.2$ in both cases), shows that the savings in the iterative case could exceed those in the incremental delivery by as much as 13%.

If $E = 1.05$, then any saving due to a reduction in effort, seems unobtainable. In fact, as Table 5 suggests, significant increases in effort occur (more than 20%).

The case where more than 50% of the functionality is delivered in the core is shown in Table 8. Here, for the maximum value of E must apply for any saving to be achieved, and then only if the expected breakage is less than 20%.

The wider implications of this are discussed in what follows.

Table 4. Incremental effort ratio r_c for $k = 0.1$, $E = 1.05$

$n\backslash c$	0.05	0.1	0.15	0.2	0.25	0.3
2	1.09	1.14	1.19	1.24	1.29	1.35
20	0.98	1.02	1.07	1.11	1.16	1.20
40	0.97	1.01	1.06	1.10	1.14	1.19
60	0.97	1.01	1.06	1.10	1.14	1.19
80	0.98	1.02	1.07	1.11	1.15	1.19
100	0.99	1.03	1.08	1.12	1.16	1.20

Table 5. Incremental effort ratio r_c for $k = 0.3$, $E = 1.05$

$n\backslash c$	0.05	0.1	0.15	0.2	0.25	0.3
2	1.07	1.12	1.17	1.22	1.28	1.33
20	0.99	1.04	1.08	1.13	1.18	1.22
40	0.99	1.03	1.08	1.12	1.17	1.21
60	1.00	1.04	1.09	1.13	1.18	1.22
80	1.01	1.05	1.10	1.14	1.19	1.23
100	1.02	1.07	1.11	1.15	1.20	1.24

Table 6. Incremental effort ratio r_c for $k = 0.1$, $E = 1.2$

$n\backslash c$	0.05	0.1	0.15	0.2	0.25	0.3
2	1.05	1.11	1.17	1.23	1.28	1.34
20	0.65	0.69	0.72	0.76	0.79	0.82
40	0.60	0.63	0.66	0.69	0.72	0.76
60	0.59	0.62	0.64	0.67	0.70	0.73
80	0.59	0.61	0.64	0.67	0.69	0.72
100	0.59	0.61	0.64	0.67	0.69	0.72

Table 7. Incremental effort ratio r_c for $k = 0.3$, $E = 1.2$

$n\backslash c$	0.05	0.1	0.15	0.2	0.25	0.3
2	0.99	1.05	1.10	1.15	1.21	1.27
20	0.70	0.74	0.77	0.81	0.85	0.89
40	0.67	0.70	0.74	0.77	0.81	0.84
60	0.66	0.70	0.73	0.76	0.79	0.83
80	0.67	0.70	0.73	0.76	0.79	0.83

Table 8. Incremental effort ratio r_c for $k = 0.6$, $E = 1.2$

$n\backslash c$	0.05	0.1	0.15	0.2	0.25	0.3
2	0.98	1.03	1.08	1.14	1.19	1.25
20	0.84	0.88	0.93	0.97	1.02	1.06
40	0.83	0.88	0.92	0.96	1.01	1.05
60	0.84	0.88	0.92	0.97	1.01	1.05
80	0.85	0.89	0.93	0.98	1.02	1.06
100	0.87	0.91	0.95	0.99	1.03	1.07

4.3. Comparison

The tables provide some interesting information which can be used be project planners, none of which is surprising given the form of the equations.

Firstly, the behaviour of the functions is dominated by the form of dn. However, using the equation concerned, we find that the ratios, independent of the mode of project (iterative of incremental), change little if there are more than 20 iterations. This is interesting, given Gilb's exhortation that iterations should add 2% to the result (Gilb, 1997) (in fact, not exceed 2% of the budget for the project).

Secondly, the influence of the exponent is over-powering. For example, with $b = 1.2$, it is still possible to achieve useful savings when the initial deliverable is as high as 60% of the predicted functionality. In addition, at 20 increments, a gain is still possible with an inflation factor as high has 20%.

However, the reader must be weary of treating E as an independent variable, under the control of the developers. The discussion refers to a relative saving, and does not consider the effect of forcibly increasing E. This will increase the BNEF directly. The overriding conclusion, however, is that project with large E, the more difficult ones, both forms of iteration offer relative savings provided breakage is controlled.

Thirdly, in practice, good design-planning is rewarded. Inflation factors above 20%, which could be regarded as an indication of sloppy planning at the increment level, preclude savings. This is NOT ameliorated by having large numbers of small, sloppily planned iterations, and may give some support to the XP fraternity who claim theirs is a disciplined approach after all.

5. Conclusion and discussion

In the original COCOMO model (Boehm, 1981) in "Component Level Estimation" a software project is divided into "components" (modules) that are sized separately. The over all project size (the sum of the size of the components) is used to compute the over all productivity that is used as nominal productivity to estimate the effort for the individual components. The gain in productivity in working with smaller components is not explicitly realised in these models. The model for estimating incremental development effort as presented in equation (6) above is different from the COCOMO models in that the economics of downsizing are realized in the sum of effort of the small increments. The initial effort term is however expected to start to penalise the incremental development if the increments become overly numerous. (An example of incremental effort computation is presented in the COCOMO II text (Boehm et al., 2000). Three increments are taken. The overall estimation method is along the lines presented in this paper but no analysis is performed on varying the number of increments.)

It is to be noted that we have looked at the effort side of the development in isolation. Other benefits of incremental development such as increased user participation, lessened risk of project failure, and benefits to customer of early delivery of parts of the system have not been taken into consideration.

In this paper, we have used two important but rather simple assumptions relating to the impact of iterative development. The first (linear over head), assumes that the

initial design overhead is linear with the number of iterations. The second is that the *breakage* can be treated as a constant, independent of the number of iterations. These are extremely simplistic, and the authors intend to investigate these issues further. However, we suggest that they are conditions of necessity for this approach to be viable. Our results suggest that, if these two conditions hold, then iterative development can yield significant gains, independent of the number of iterations. The challenge for project manager is to be able to either identify project types for which this is true, or, to manage a project so that they are!

Assuming that the breakage, c is even mildly dependant on the number of iterations (e.g., assume $O(n^{1/3})$), yields quite distinct minima. But only in those cases where E is large and k is small (for the evolutionary delivery). The question arises as to the impact of c not being independent of the number of iterations. There are many ways in which this could be modeled, based upon either communications over-heads or interference models. These might yield c as $O(n^2)$ or at best, perhaps $O(\log_2(n))$. Our conjecture is that this dependency would alter the results substantially. To test this, calculations were made assuming c was $O(n^{1/3})$, as a compromise. The results showed distinct minima, but only for the case where E was large, and k was small.

Parnas (1979) presents an approach to evolutionary delivery based upon virtual machine techniques, an approach found in the early days in embedded systems where memory was extremely limited. We could argue that the preliminary design cost in such a case may be quite high, if the team had no prior experience, but that expected breakage rate (c) may be quite low. Certainly, the number of iterations (releases) could be rather low.

We have already remarked that our simplifications are in fact saying something about desirable project properties required if iterative development is to yield lower costs. In fact, lower production cost will be irrelevant if the client is not satisfied, and rejects the project, or fails to award the developer repeat business. It is this in part, that the agile process community claims to be addressing.

The authors would like to add a note of caution however. The equations are quite sensitive to the exponent E in the COCOMO model. We have assumed that the values of E are the same in both cases. However, it is possible that the inherent difficulty (i.e., value of E, which can be considered to have this property), for an iterative project may be greater of lesser than that for a monolithic property, and be reflected in the value of E. In this case, if E were greater (for the iterative case), then the gain from a large number of small iterations would increase with n. In part, some of the authors are concerned that iterative/extreme development may be used as an excuse for not making a "proper" attempt to obtain relevant domain knowledge, and to compensate for the failure to foster large depths of experience within development teams, as happens in conventional engineering. One should not assume that the world only consists of "E-type" systems! However, even where the domain experience is sufficient to allow generalizations to be predicted, our calculations show that iterative/incremental delivery may yield savings in appropriate circumstances.

The work reported in this paper can be extended in a number of directions to explore a number of potential avenues. In particular, the authors are interested in addressing a number of emerging challenges that include:

- Extending the discussion to cover evolutionary development methods as well as addressing the new focus on growth of software and adaptive design;
- Developing an earned-value type of method to account for the "salvageable value" of a project, especially in relation to given increments;
- Evaluating the utility of adopting a long-term strategic perspective that stretches beyond single projects and enables;
- Increasing the scope of the work discussed here to assess the impact of agile development processes and propose agile estimation models;
- Providing a basis for reasoning about continuation and cancellation of projects (or deliverables) and conducting trade-offs between time, effort and increments in dynamic projects.

It is therefore hoped that the work put forward in this paper will serve as an opening for further discussion and investigation. The opening of a new perspective can also stimulate additional insights needed to bridge the gap in the analysis of the economic impact of non-conventional development approaches on the product, process and project.

References

Basili, V.R. and Turner, A.J. 1975. Iterative enhancement—a practical technique for software development, *IEEE Transactions on Software Engineering* SE-1-4: 390–396.

Beck, K. 2000. *Extreme Programming Explained*, Reading, MA, Addison-Wesley.

Bennington, H.D. 1995. Production of large computer programs, *Annals of the History of Computing* 5(4): 350–361.

Berrisford, T. and Wetherbe, J. 1979. Heuristic development: A redesign of systems design, *MIS Quarterly* 3(1): 11–19.

Boehm, B.W. 1981. *Software Engineering Economics*, Englewood Cliffs, NJ, Prentice-Hall.

Boehm, B.W., Abts, C., Brown, A.W., Chulani, S., Clark, B.K., Horowitz, E., Madachy, R., Reifer, D., and Steece, B. 2000. *Software Cost Estimation with COCOMO II*, Upper Saddle River, NJ, Prentice-Hall.

Boehm, B. 2002. Get ready for agile methods, with care, *IEEE-CS Computer* 35(1).

Booch, G. 1996. *Object Solutions: Managing the Object Oriented Project*, Reading, MA, Addison-Wesley.

Brooks, F.P. 1975. *The Mythical Man-Month*, Reading, MA, Addison-Wesley.

Brooks, F.P. 1987. No silver bullet: Essence and accidents of software engineering, *IEEE Computer* 20(4): 10–20.

Capers Jones, T. 1998. *Estimating Software Costs*, New York, McGraw-Hill.

Clark, B., Devnani-Chulani, S. and Boehm, B. 1998. Calibrating the COCOMO II post-architecture model, *Proceedings of ICSE 1998*, IEEE Press.

Constantine, L.L. and Yourdon, E. 1978. *Structured Design*, Englewood Cliffs, NJ, Prentice-Hall.

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

Cusumano, M.A. and Yoffle, D.B. 1999. Software development in Internet time, *IEEE Computer* 32(10): 60–69.

Dalcher, D. 2002. Life cycle design and management, In *Project Management Pathways: A Practitioner's Guide*, ed. M. Stevens, The Association for Project Management, High Wycombe, APM.

Elbaum, S.G. and Munson, J.C. 1998. Code churn: A measure for estimating the impact of code change, *Proc. IEEE-CS International Conference on Software Maintenance*, Bethesda, IEEE Press, pp. 24–33.

Elssamadisy, A. and Schalliol, G. 2002. Recognizing and responding to "Bad Smells" in extreme programming, *Proc. ICSE 2002*, Orlando, IEEE Press, pp. 617–622.

Giddings, R.V. 1984. Accommodating uncertainty in software design, *Communications of the ACM* 27(5): 428–434.

Gilb, T. 1997. *EVO: The Evolutionary Project Managers Handbook*, unpublished manuscript.

Gilb, T. 1988. *Principles of Software Engineering Management*, Wokingham, Addison-Wesley.

Goldberg, A. and Rubin, K. 1995. *Succeeding with Objects*, Reading, MA, Addison-Wesley.

Graham, D.R. 1989. Incremental development: Review of nonmonolithic life-cycle development methods, *Information and Software Technology* 31(1).

Graham, D.R. 1992. Incremental development and delivery for large software systems, *IEEE Computer* 25(11): 1–9.

Hess, W. 1996. Theory and practice of the software process—a field study and its implications for project management, *Proc. Software process Technology, 5th European Workshop*, WESPT 96, Lecture Notes in Computer Science, Vol. 1149, Berlin, Springer, pp. 241–256.

Krutchen, P. 2000. *The Rational Unified Process*, London, Longman.

Krzanik, L. 1988. Enactable models for quantitative evolutionary software processes, *Proc. the 4th International Software Process Workshop on Representing and Enacting the Software Process*, April 1988, ACM SIGSOFT Software Engineering Notes, Vol. 14, p. 4.

Lehman, M.M. 1969. The programming process, IBM Research Report RC2722M. IBM Research Centre, Yorktown Heights, September 1969, In *Program Evolution-Processes of Software Change*, eds. M.M. Lehman and L.A. Belady, London, Academic Press, 1985.

Lehman, M.M. 2000. Rules and tools for software evolution planning and management, *FEAST2000*, Imperial College, London, July, pp. 53–68.

Lehman, M.M. and Ramil, J.F. 2001. An approach to a theory of software evolution, *Proc. International Workshop on Principles of Software Evolution*—IWPSE 2001 (Keynote) Vienna.

Mills, H.D. 1971. Top-down programming in large systems, In *Debugging Techniques in Large Systems*, ed. R. Ruskin, Prentice-Hall, pp. 41–55.

Mills, H.D. 1976. Software Development, *IEEE Transactions on Software Engineering* SE-2(6): 265–273.

Mills, H.D. 1980. Incremental software development, *IBM Systems Journal* 19(4).

Mills, H.D., Dyer, M. and Linger, R.C. 1987. Cleanroom software engineering, *IEEE Software* 4(3): 19–24.

Musa, J.D. 1993. Operational profiles in software-reliability engineering, *IEEE Software* 10(2): 14–32.

Parnas, D.L. 1979. Designing software for EASE of extension and contraction, *IEEE Transactions on Software Engineering* SE-5(2): 128–138.

Podalsky, J.L. 1977. Horace builds a cycle, *Datamation*, November: 162–168.

Royce, W. 1998, *Software Project Management, A Unified Framework*, Reading, MA, Addison-Wesley.

Royce, W.E. 1990. TRW's Ada process model for incremental development of large software systems, *Proc. 12th International Conference on Software Engineering, ICSE 12*, IEEE Press, pp. 2–11.

Royce, W.W. 1970. Managing the development of large software systems, *Proc. IEEE WESCON 1970*, IEEE Press.

Royce, W.W. 1990. Pragmatic quality metrics for evolutionary software development models, *Proc. the ACM Conference on TRI-ADA '90*, Baltimore, pp. 551–563.

Stapleton, J. 1997. *DSDM Dynamic Systems Development Method*, Wokingham, Addison-Wesley.

Zahran, S. 1997. *Software Process Improvement*, Harlow, England, Addison-Wesley.

Oddur Benediktsson has been active in research in the field of software development methods for over thirty years. He has published a many papers, and participated in numerous national and international projects on standardisation and improved methods for software development. Oddur is a member of the programme committees and a regular participant in many international conferences. Dr. Benediktsson has worked as a consultant and been employed in the IT industry for some years. He is a Professor in Computer Science at University of Iceland where he has been instrumental in organising the curricula of computer science and software engineering.

Darren Dalcher is a Professor of Software Project Management at Middlesex University where he leads the Software Fornesics Centre (http://www.cs.mdx.ac.uk/research/SFC/), a specialised unit that focuses on systems failures, software pathology and project failures. He gained his Ph.D. in software engineering from King's College, University of London. In 1992, he founded and has continued as chairman of the Forensics Working Group of the IEEE Technical Committee on the Engineering of Computer-Based Systems, an international group of academic and industrial participants formed to share information and develop expertise in failure and recovery. Professor Dalcher is active in a number of international committees and steering groups. He is heavily involved in organising international conferences, and has delivered numerous keynote addresses and tutorials. He is the Editor-in-Chief of Software Process Improvement and Practice.

Karl Reed is a pioneer of software engineering education in Australia, and is recognised as national spokesperson on industry policy, advising State and Federal Governments both formally and informally. Associate Professor Reed held number of different positions in different institutions including Senior Visiting Fellow in the Faculty of Business at the Royal Melbourne Institute of Technology University. Associate Professor Reed is a Fellow and an Honorary Life Member of the Australian Computer Society, past chairman of the Victorian branch and Director of its Computer Systems and Software Engineering Board. He has also been a Distinguished Business Associate at Swinburne University of Technology. He was consultant editor to Australasian Computer World from 1978 to 1995. He is an Honorary Visiting Professor at the University of Middlesex, and has been a Guest Scientist at the Fraunhofer Institute for experimental Software Engineering. He was a Governor of the IEEE Computer Society for the statutory limit of two terms (1997–2000, 2000–2002), and is currently the Chair of the IEEE-CS' Technical Council on Software Engineering (2000–2002, 2002–2004).

Mark Woodman's research interests focus on complex software, particularly object-oriented systems. He also studies social and cultural aspects of software engineering. An author of many articles and books, and a consultant on several TV programmes, recent international articles focus on object technology and on process improvement. He is Middlesex University's principal investigator on a large European project on software components and process improvement. Professor Woodman has been heavily involved in international standards work as an ISO convenor and was a BSI panel chair. He was a member of the OOPSLA 99 programme committee and was co-programme chair of TOOLS Europe 2000. He has been involved in distance education for over twenty years and he has led projects for introducing computing to several thousand students in the UK and overseas and has led a team that won awards from the British Computer Society (BCS) and the Design Council. In 1999 he was a BCS judge for its IT Awards competition.