Incremental Development Productivity Decline

by

Ramin Moazeni

A Dissertation Presented to the
FACULTY OF THE USC GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
Doctor of Philosophy
(Computer Science)

August 2014

Copyright 2014 Ramin Moazeni
DEDICATION

To my parents, Nahid and Mahmoud

To my grandfather, Ali who I lost this year

And to my wife, Laleh
ACKNOWLEDGMENTS

I would like to greatly thank everyone who was a part of my journey. First, I would like to thank my advisor, Dr. Barry Boehm for guiding and advising me through the research journey. He was the main inspiration for me to pursue the Ph.D. degree. His vision and great advices on selecting the right topic and taking the correct approach helped me through all these years.

I am grateful to my dissertation committee members, Dr. Aiichiro Nakano, and Dr. Stan Settles, for the additional guidance and suggestions in completing my dissertation. Also, I would like to extend my thanks to the professors that were part of the Ph.D. qualifying exam committee, Dr. William GJ Halfond and Dr. Richard Selby, for comments and feedback on my topic proposal.

I would like to thank all my friends at USC Computer Science department and members of the Center for Systems and Software Engineering Lab for their valuable friendship which made my studies joyful. I would also like to specially thank my colleague Daniel Link that our working together has not only inspired me, but has helped me finish my dissertation in a fun way. He is a true friend I can really rely and build on.

I wish to express gratitude to my family, my parents Nahid and Mahmoud, for their great deal of support throughout my PhD studies. They’ve taught me about hard work and self-respect, about persistence and about how to be independent. They both were great role models of resilience, strength and character.

Last but not least, I adore the love and support from my dear and loving wife, Laleh, for sincere encouragement and inspiration throughout my research work and lifting me uphill this phase of life. She was always there for me through all the difficult
times. Without her it would have been unquestionably much harder to finish a PhD while having a fruitful and challenging career.
TABLE OF CONTENTS

1 Introduction 1

1.1 Background 1

1.2 Central Hypotheses 3

1.3 Iterative and Incremental Development 4

1.3.1 Incremental Development Definition 4

1.3.2 Incremental Development Types 5

1.4 Definitions 8

1.4.1 Increments and builds 8

1.4.2 Productivity 8

1.4.3 IDPD 10

1.4.4 IDPD factor 11

1.4.5 Categories of projects (IDPD types) 12

2 Background and Related Work 19

2.1 Major Cost Estimation Models 19

2.1.1 Cost Estimation Considerations 19

2.1.2 COCOMO II Cost Drivers and Scale Factors 22

2.2 Cost Model for Maintenance and Reuse 24

2.3 Laws of Software Evolution 26

2.3.1 Lehman's System Types 26

2.3.2 Lehman's Laws of Software Evolution 27

2.3.3 Discussion of the individual Laws of Software Evolution 29

3 Research Approach 39

3.1 Behavioral Analysis 39
3.2  Data Collection and Analysis 39
  3.2.1  Controlled Experiment 42
  3.2.2  Open Source Projects 43
  3.2.3  Data Collection Challenges 44
3.3  Contextual Analysis 45
3.4  Statistical Analysis 46
  3.4.1  Linear Correlation 46
  3.4.2  Tailed Pair T-Test 46
  3.4.3  One-Way ANOVA 48
4  Analysis and Representation of Data 49
  4.1  Summary of Data Collection and Selection 49
  4.2  Productivity Decline: Hypothesis 1 51
    4.2.1  Case Study 1: Quality Management Platform 52
    4.2.2  Case Study 2: XP1 and XP2 54
    4.2.3  Statistical Significance 56
  4.3  Build-to-Build Behavior: Hypothesis 2 61
  4.4  Domain Variation: Hypothesis 3 63
  4.5  COCOMO II and IDPD 65
    4.5.1  Cost Estimation 65
  4.6  IDPD and COCOMO II 67
    4.6.1  COCOMO II cost drivers and scale factors 67
    4.6.2  How IDPD influences parameters 71
    4.6.3  Potential New Cost Estimation Parameters 71
      Time between increments 72
      Defect number and complexity in increments 73
LIST OF TABLES

Table 1. IDPD Effort Drivers (Madachy R., Boehm, Clark, Tan, & Rosa, 2011) .......... 11
Table 2: IDPD intensity by category .............................................................................. 18
Table 3: Effect of COCOMO II effort multipliers on IDPD ........................................... 23
Table 4: Effect of COCOMO II Scale Factors on IDPD ................................................. 24
Table 5: Core collected data attributes ......................................................................... 41
Table 6: Summary of data sources ................................................................................ 44
Table 7: Incremental project domains ........................................................................... 45
Table 8: COCOMO II Driver ratings ............................................................................. 53
Table 9. Correlation coefficient (with p-value) between build-to-build productivity and number increments .................................................................................................................. 57
Table 10. Confidence Value, t-stat value, and p-value of 2 tailed pair t-test .................... 62
Table 11. One-way ANOVA results for Hypothesis 3 .................................................... 64
Table 12. COCOMO II Cost Drivers and Scale Factors expected to have a decrease in effort ........................................................................................................................................ 66
Table 13. COCOMO II Cost Drivers and Scale Factors expected to have no specific trend in effort ........................................................................................................................................ 68
Table 14. P-value of weighted multiple regressions between cost drivers and productivity ....................................................................................................................................... 69
Table 15. P-value (<0.08) of regression between cost drivers and productivity for 4 DR (directed research) projects ........................................................................................................... 70
Table 16. Weighted regression between cost drivers and productivity (only for cost drivers with correlation> 0.7) .......................................................................................... 71
Table 17. Correlation of Code Size Metrics to Increments .............................................. 74
Table 18. Comparison of cost estimation models............................................................. 77
LIST OF FIGURES

Figure 1. Incremental development types ................................................................. 7
Figure 2. Effect of IDPD on number of builds .......................................................... 13
Figure 3. Productivity for all projects on a logarithmic scale ................................. 50
Figure 4. Average SOLC of projects .................................................................... 50
Figure 5. Histograms showing number of increments across projects .................. 51
Figure 6: QMP productivity across increments ...................................................... 54
Figure 7. XP1 productivity across increments ....................................................... 55
Figure 8. XP 2 Productivity across Increments ...................................................... 56
Figure 9. Normalized productivity by increments for 8 projects ......................... 58
Figure 10. Normalized productivity trend lines ...................................................... 58
Figure 11. Overview of productivity trends across projects ................................. 60
Figure 12. P-value of 2 tailed t-test across projects .............................................. 63
Figure 13: Averages of Average IDPD................................................................. 64
Figure 14: IDPD Distribution over Domains ......................................................... 64
Figure 15: Average IDPD over Domains ............................................................... 64
Figure 16. Overlap across Builds ......................................................................... 82
Figure 17. RUP with MBASE Anchor Point ........................................................... 83
Figure 18. Overlap across builds with carry forward ........................................... 84
Figure 19: COINCOMO Element Relationship diagram ........................................ 90
Figure 20: COINCOMO Component estimation .................................................. 91
Figure 21: COPSEMO calculation ...................................................................... 91
Figure 22: COINCOMO support for multibuild estimation ................................. 92
ABSTRACT

Software production is on the critical path of increasingly many program abilities to deliver effective operational capabilities. This is due to the number, complexity, independence, interdependence, and software-intensiveness of their success-critical components and interfaces. The estimation parameters and knowledge bases of current software estimation tools are generally good for stable, standalone, single increment development. However, they do not fully account for the degrees of program and software dynamism, incrementality, coordination, complexity, and integration. These phenomena tend to decrease software productivity relative to the cost model estimates made for the individual software components and for the overall systems, but it is difficult to estimate by how much.

Incremental software development generally involves either adding, modifying, or deleting parts of the code in the previous increments. This means that if a useful system is to be built, the maintenance that will have to go into previous increments will take away productivity from the later ones.

This research tests hypotheses about a phenomenon called Incremental Development Productivity Decline (IDPD) that may be more or less present in incremental software projects of various categories.

Incremental models are now being used by many organizations in order to reduce development risks while trying to deliver releases of the product on time. It has become the most common method of software development with characteristics that influence the productivity of projects.
Different ways of measuring productivity are presented and evaluated in order to come to a definition or set of definitions that is suitable to these categories of projects. Data from several sources has been collected and analyzed, and hypotheses tested about the degree of IDPD and its variation by increment and category. The results indicated the existence of an IDPD phenomenon, that its magnitude varies by application category, but that it tended to vary from increment to increment.
1 Introduction

1.1 Background

There have been indications on individual projects that economies of scale do not apply to software increments as they do to increased unit numbers of hardware. While hardware productivity benefits from an increase in units, the “unit cost of later software increments tend to increase, due to previous-increment breakage and usage feedback, and due to increased integration and test effort. Thus, using hardware-driven or traditional software-driven estimation methods for later increments will lead to underestimates and over-runs in both cost and schedule” (Madachy R., Boehm, Clark, Tan, & Rosa, 2011).

The following example of a large defense software system is given in (Madachy R., Boehm, Clark, Tan, & Rosa, 2011) (SLOC stands for “Source Lines of Code):

- 5 builds, 7 years, $100M cost
- Build 1 productivity over 300 SLOC/person-month
- Build 5 productivity less than 150 SLOC/person-month (including build 1-4 breakage, integration, and rework)
- 318% change in requirements across all builds

A constant IDPD factor (i.e. the reduction in productivity between two subsequent builds) of 20% over the course of four increments would result in a reduction of productivity to about 48% of the original one (Madachy R., Boehm, Clark, Tan, & Rosa, 2011). At the time that (Madachy R., Boehm, Clark, Tan, & Rosa, 2011) was written, there were indications of similar IDPD phenomena, but very little data. For example, IDPD factors of 14% (with variations, in a smaller software
system) had been found. Large schedule slippages had been found for later increments of large commercial software applications, such as Microsoft’s Word for Windows (Gill & Iansiti, 1994), Windows Vista and others (Elssamadisy & Schalliol, 2002) (Madachy R., Boehm, Clark, Tan, & Rosa, 2011), but specific data on the quantities were not available.

Intuitively, an IDPD for software increments seems reasonable. Each increment adds to the body of the existing code for the project. When each increment depends on the previous increments, the amount of code that increment has to be compatible with and integrated with increases. Additionally, being dependent on previous increments, the developers will uncover previously unknown errors in the existing code and have to fix them. They will also have to rework parts of it, all on the budgets available for the current increment.

On the other hand, though, depending on the characteristics of a given project, the productivity may rise after several increments with declining productivity. This productivity gain might happen for several reasons: learning how to deal with the system (e.g. through figuring out the architecture); from later increments having less complex requirements; or later increments that may not depend on all parts of earlier ones.

Given the issues and considerations about incremental development, it is instructive to look at a typical incremental development project and compare productivity-related situations that will arise if the increments are developed in parallel, sequentially, or with overlapping timeframes. The first consideration relevant to this is the degree of dependence between increments. Conceivably, a given increment can depend on any other given increment: not at all; partially; or completely.

The earliest part at which work on an increment can start depends on how far along the previous ones need to be. From a technological standpoint if an increment has no dependencies, it
can be started on immediately. There will still need to be considerations on the amount of resources that can be committed to increments that are developed in parallel. The tradeoff is between the amounts of resources being put into development versus the time the development will take. Where dependencies exist, increments have to wait until the previous ones reach to the point that allows them to go forward. The choice then lies in whether the next increment should be started at that point or later. The longer the time span covered by the development, the higher the cost and risk of reworking older increments.

The objective of this research has been to gather sufficient data to be able to test hypotheses about the existence of an IDPD phenomenon, about its magnitude, about its consistency from increment to increment.

### 1.2 Central Hypotheses

In order to describe our central hypotheses, we need to understand what productivity is. The conventional productivity equation is Output per Input. While this is simple to say, different metrics have been used to measure this in terms for software development. There is a host of different output metrics (SLOC, Equivalent Source Lines of Code aka ESLOC, function points, and business value to name a few), but the available data from the observations limits which metrics can be used. Typically, in software engineering, productivity is measured in SLOC per person-hour. Formal definition of productivity is presented in 1.4.3.

The principal research question to be addressed by this research is:

*How does the productivity behave over the course of an incremental project? Are there patterns to it? What are the factors that influence it? Does its behavior vary by class of software?*
To address the above questions, the following null hypotheses have been formulated to test whether they can be rejected.

**Productivity Decline Hypothesis:** In incrementally developed software projects that have coherence and dependency between their increments, productivity doesn’t decline over their course. Coherence and dependency between increments means that the result of the project is not merely an accumulation of different pieces of software that might just as well be developed concurrently without a loss in overall productivity. An example for such a case would be a set of tools that are developed independently of each other and sold together.

**Build-to-Build Behavior Hypothesis:** The rate of productivity decline from increment to increment is relatively constant. Although some projects and “Laws” (Lehman M., 1980) (Lehman, Ramil, Wernick, & Perry, 1997) suggest that there is a statistically invariant percentage of productivity decline across increments, this may not be the case in general.

**Domain Hypothesis:** For different domains (IDPD types), the average decline in productivity over the course of a project doesn’t vary significantly.

### 1.3 Iterative and Incremental Development

#### 1.3.1 Incremental Development Definition

In order to come to a useful definition of incremental development, it helps to look at the definition of “increment”: for software, a generally accepted definition of an increment is an improvement evolving additions to, modifications of, and deletions of existing increments’ code (Nguyen, 2010).

Incremental development therefore has to be a process by which each successive release adds value to the product, albeit in different ways. While at first it may look promising to make a
distinction between iterative development as a process oriented toward a specific end goal that merely perfects the product through repetition of a sub-process cycles and incremental development as one that is more open-ended and adds functionality at each step, that distinction is not tenable when it is considered that the goal-oriented iterative process also leads to the addition of functionality at every step (Larman, 2004). Additionally, whether or not a project will be ended at a certain point can be hard to know a priori.

This dissertation focuses on the cases of incremental development for which the increments are:

- done in more than one step,
- where each new step adds new functionality, and
- where there is coherence and dependency the new step and the previous steps

Additionally, in order to be considered an increment for this IDPD research, an increment will have to contribute a significant amount of new functionality and must add a significant amount of size over the previous one, i.e. it must not be less than a tenth of the size of the previous one and it must not be just a bug fix of the previous one. Otherwise it is considered part of the previous increment.

### 1.3.2 Incremental Development Types

Four models of incremental development have been identified (Boehm & Lane, DoD Systems Engineering and Management Implications for Evolutionary Acquisition of Major Defense Systems, 2010) (SEBoK). For the purposes of this research, two of them will be combined into one.
**Pre-specified Sequential model**

In this model, development is split up so initial operational capability can be achieved early. This is then followed by several pre-planned product improvements. The evolution of the project is limited by the pre-specified plan.

**Evolutionary Sequential model**

The second strategy is the Evolutionary Sequential model. It rapidly leads to an initial operational capability, which is upgraded based on operational experience. Agile software development fits this model. Any defects of a build will be fixed in the next one.

Boehm and Lane also identify a model called the Evolutionary Overlapped model. It covers the special case of deferring the next increment until critical enablers such as desired new technologies, anticipated new commercial product capabilities, or needed funding become available or mature enough to be added. As far as this research is concerned, this is regarded as a mere variant of the Evolutionary Sequential model in which the time interval between increments is unknown or harder to plan for.

**Evolutionary Concurrent model**

In the Evolutionary Concurrent model, several increments are worked on at the same time in a staggered way: For example, while part of the team develops one increment, another part already scopes and architects the next one. This mode of operations can be extended to any number of increments.

Figure 1 shows the workflow of the four identified incremental types (Boehm & Lane, 2010) (SEBoK):
Time phasing terms: Scoping; Architecting; Developing; Producing; Operating (SADPO)

Pre-specified Sequential: SA; DPO1; DPO2; DPO3; …

Evolutionary Sequential: SADPO1; SADPO2; SADPO3; …

Evolutionary Concurrent: SA; D1 ; PO1…

SA2; D2 ; PO2…

SA3; D3; PO3 …

Pre-specified Sequential

Evolutionary Sequential

Evolutionary Concurrent

Figure 1. Incremental development types
1.4 Definitions

1.4.1 Increments and builds

To understand our empirical considerations here, it is important to know the definitions of “increment” and “build” use in this research. For simplicity, the definitions exclude deleted code, which was usually insignificant in the projects analyzed.

- A build is a released version of the software that is comprised of all the code that has been written up to the point the build was released.
- An increment is only the code that has been written for a given increment. This includes new code and adapted/reworked/fixed code from previous increments.
- A build (at least beyond the third one) therefore contains the code from several previous increments, however it may be dispersed over the build.

1.4.2 Productivity

Productivity is defined as the ratio between “the amount of goods or services produced” and “the labor or expense that goes into producing them”. Similarly, in software projects, productivity can be measured as the ratio between “the amount of software” or code produced to “the labor and expense of producing it” (Boehm B., 1981). However, this simple approach doesn’t work too well in real world scenarios. There are so much more factors that affect productivity measures and prediction. Boehm (Boehm B., Improving Software Productivity, 1987) defined productivity as “the ratio of output units produced per unit of input effort”, for example, it could be lines of source codes versus person-month effort. Scacchi (Hurley, 1995) (Scacchi, 1991) pointed out that productivity is comparable when the outputs and inputs are of the same kind, i.e. “a software
development effort with productivity 2X is twice as productive as another effort whose productivity is X”. In the IBM federal study case, Walston and Felix (Walston & Felix, 1977) measured productivity in terms of number of lines of code produced per person-hour, and in another IBM study that was done by Albrecht (Albrecht A., 1979) (Albrecht & Gaffney, 1983) he used “function points” to measure productivity. A function point is defined as “a composite measure of a number of program attributes that are multiplied by weighting factors then added together”. In Albrecht’s approach, productivity wasn’t simply just a ratio, but he introduced various attributes, which could potentially have effects on productivity measurement, such as file accesses, function calls, etc. In TRW’s software cost estimation program, Boehm (Boehm B., 1981) found that “software cost drivers are effectively the inverse of productivity drivers”. The whole idea is that drivers that yield a higher cost will conversely yield a lower productivity. For example, when the staff capacity is high and the product complexity is low, it will lead to high productivity and low cost software production. Vosburgh and associates did one of the most substantial studies of large-scale software productivity (Vosburgh, 1984). They defined and divided productivity drivers according to “the ability of a software project manager to control them”. They identified two types of factors: product-related factors that are not usually controllable by a project manager and production process related factors that are controllable by managers. Within each type, they also identified more detailed and specific constraints. Overall, software productivity cannot be understood nor measured simply as a ratio of the amount of produced codes for some unit of time. It requires a more systematic and statistical analysis of various production measures, as well as their inter-relationships.
The conventional productivity formula is:

\[ Productivity = \frac{Output}{Input} \]

In the case of software development, we generally define productivity as:

\[ Productivity = \frac{SLOC}{Effort} \]

(SLOC can be replaced by KSLOC as needed.)

The problem with measuring productivity in this way is that it regards all SLOC that are produced as an equal measure of productivity, without differentiating whether a given amount of SLOC is a desirable product or just a byproduct of the development team’s effort. For example, SLOC that are produced in order to fix errors, refactor existing code, generated by and IDE or from duplicating or cutting and pasting code all count as productive based on this formula. Also for software maintenance, Nguyen (Nguyen, 2010) found that deleting code often required a good deal of effort. Since any data collection didn’t find much deleted code, I adopt SLOC added in the new increment as the definition of productivity as SLOC/Effort in this research.

1.4.3 IDPD

Incremental development productivity decline is a phenomenon in which there is an overall decline in productivity of the increments as measured in section 4. It should be clarified that for there to be IDPD, an overall decline includes cases where the decline between increments is not constant and where there can even be a rise in productivity between two given increments. In such cases, an average productivity decline can be derived from a negative data fitting trendline.
1.4.4 IDPD factor

The IDPD factor is the percentage by which productivity decreases between two given increments. Since it describes a decrease, it is positive if the productivity decreases and negative if the productivity increases between the two increments.

A build-to-build IDPD factor of about 16% will cause a reduction from the initial productivity to 50% during the fifth new build. That such declines can happen in actual projects is shown by a project that forms the basis of one our case studies, which had an IDPD factor of 14% (Tan, 2009). Large-scale commercial software projects, such as Word for Windows, Windows Vista and others have had similar experiences (Elssamadisy & Schalliol, 2002) (Gill & Iansiti, 1994). The table below shows experience-based IDPD effort drivers and their ranges for a four-build system (based on the table in (Madachy R., Boehm, Clark, Tan, & Rosa, 2011)):

| Table 1. IDPD Effort Drivers (Madachy R., Boehm, Clark, Tan, & Rosa, 2011) |
|---------------------------------|-----------------|
| Personnel experience gain (variation is due to different turnover rates) | -5% to -20% |
| Breakage, maintenance of full code base | 20-40% |
| Diseconomies of scale in development, integration | 10-25% |
| Requirements volatility, user requests | 10-25% |

Note that personnel experience gain decreases IDPD, while the other drivers listed in the remaining three rows increase it. Basically, work needs to be done on the code in the previous increments. But needs to be done on the budget for the current increment.
The best case would be 20% more effort per increment when the increasing drivers are minimized and the decreasing one maximized (20% + 10% + 10% - 20%). This would amount to an productivity decline of 6% per increment for a four-build system.

The worst case would have the decreasing driver minimized and the increasing one maximized, which would add up to 85% more effort (40% + 25% + 25% - 5%); for a 4-build system, the productivity decline per increment would be 23%.

The difference between 6% and 23% may not seem large, but over a significant number of builds, the difference will be striking. Figure 2 shows how a seemingly small difference in the productivity decline can stretch out a project to twice as many builds and a factor of 2 schedule overruns, for a four million SLOC system able to produce one million SLOC in its first increment (Madachy R., Boehm, Clark, Tan, & Rosa, 2011). Thus, it is important to understand the IDPD factor and its influence when doing incremental or evolutionary development. Testing of hypotheses indicates that the magnitude of the IDPD factor may vary by type of application (infrastructure software having higher IDPDs since it tends to be tightly coupled and touches everything; applications software having lower IDPDs if it is architected to be loosely coupled), or by recency of the build (older builds may be more stable). Further data collection and analysis will be very helpful in improving the understanding of the IDPD factor.

1.4.5 Categories of projects (IDPD types)

Before the different categories of projects and their typical IDPD-related relevance are considered, it needs to be said that these are only the typical cases and that ultimately any software project, no matter how big or small, can be done with or without increments. Similarly, any
software that would normally fit one category can be treated like a member of another. The ultimate decision lies with the organization conducting the development.

![Figure 2. Effect of IDPD on number of builds](image)

**Non-Deployable Support Software**

Non-deployable code is code that either cannot be deployed due to its nature or which its author(s) do not intend to deploy. Typically this is code that is developed ad hoc as auxiliary related to one or more given projects, but not a part of its deliverables. For turnkey software to be maintained elsewhere, test and other support software should be part of the deliverables. Examples include configuration scripts, XML-based configuration settings, compiler scripts, compiler tests (“Hello World”), tests needed for conditional breakpoints, repository commit scripts and small internal one-off projects such as scripts to run software at trade shows. Additionally, while it may
be theoretically possible to deploy some subcategories of this code, the deployed code would have no value to any project stakeholder outside of the author’s organization.

Due to its small size and its ad hoc character, this kind of code is generally not developed in increments. Even if it would be done, the increments would not be of any meaningful size that would allow inferences about major projects.

**Infrastructure Software**

Infrastructure software is software that is designed to not interact with the user directly, but to provide facilities and services for other software to run. Examples include operating systems (e.g. Mac OS, Windows, Linux), virtual machines (e.g. the Java Virtual Machine or the CLR in Microsoft .NET) and operating system extensions through libraries with APIs (such as DirectX in Windows) as well as common operating environments. This software can also be more specialized than the examples given here, such as software providing email, conferencing, geolocation, or weather geolocation services.

Since the need of users to continue using a given software product that relies on specific infrastructure software tends to often outlast one or more of that infrastructure software’s major releases, the compatibility needs of infrastructure software as desired by the customers tend to be high.

While new, incompatible versions of infrastructure software are released from time to time alongside new operating systems or separately, it is often possible to have several versions of the same infrastructure software installed and have applications or the user decide which installed version is used by a given application (this is true for the Java Virtual Machine and DirectX).
**Application Software**

Application software is the software general users interact with directly in most cases and which therefore represents the computer experience for them. Examples of incrementally developed COTS products include web browsers (MS Internet Explorer, Google Chrome, Mozilla Firefox, Apple Safari) and software that is part of office suites (e.g. Word as part of MS Office), but also custom-built applications that are created for specialized usage.

Of major relevance for the compatibility needs of a given application is whether or not it is designed to read and write documents that were created by it or other applications. If so, the application will require the documents to be compatible with the current and historic data formats of itself and those other applications.

Any application will have requirements regarding infrastructure software because it will at least require an operating system to run on. Therefore the customers of such software tend to have two main compatibility needs: Operating system compatibility (will a given version of the application still work on a new operating system release?) and document compatibility (will a new version of the application still be able to work with documents that were created with previous versions?)

The functionality of COTS applications tends to be incremented in different ways over different phases, with the following pattern being typical:

*Phase 1: Toward the first release*

Before the first release, functionality is built up over several non-public unreleased increments. At some point before the release, the makers of the software may allow external beta testers to try out the software and give feedback. Beta licenses typically contain clauses that
document formats are subject to change at any time and that compatibility is not guaranteed. Since no customer external to the maker of the application has used the application to create value yet or at least has no reasonable expectation of compatibility between the pre-release versions and the finished product, the compatibility needs of the application are nil at this point.

Phase 2: After the first release

From this point on, increments will continue, and there are three ways in which they are presented (or not) to the outside world:

- Internal builds: These are increments that are neither published nor deployed.
- Bug fixes and minor versions: These are increments that are generally delivered to customers as free updates. They tend to be named “bug fixes”, “updates” or “service packs”. Compatibility needs are high because the customers regard versions with the same major release number or name as one and the same.
- Major releases: These are new full releases of the software that are presented as new versions, which typically require customers to buy a new license or renew their existing one. Compatibility needs tend to be intermediate in that it has been established over time that major releases are the typical point where a change in the document format of an application is to be expected. This is generally accepted as long as previous document formats can be read or written to (albeit with some loss of document features when compared to the new version). Infrastructure compatibility tends to be changed as well in major releases. In most cases, previous releases are supported for a while and the users of the old versions are given an upgrade path to the new version.
Some applications do not only require infrastructure software to be installed, but also offer access to their own services through a programmable interface (API). If that is the case, it adds additional compatibility requirements that are at about the same level as the ones that apply to documents. The primary IDPD driver for application will be the degree of coupling between existing and new-increment applications. The closer the coupling, the higher the breakage in the existing application and higher IDPD.

Therefore overall the compatibility needs of applications are high, but with some flexibility at periodical intervals.

**Platform Software**

These are hardware drivers for operating systems. Such drivers interact with the hardware on one side (e.g. over memory-mapped I/O and hardware interrupts) and with the operating system on the other side using an interface specified by the operating system. They do not store documents and have no other APIs.

Since they can be arbitrarily rewritten as long as they provide the same functionality to the outside, their compatibility requirements to previous versions are low.

**Firmware (Single Build)**

Firmware is code that either provides all functionality for a device (such as the firmware of a remote control) or that enables a device to load other code (such as the BIOS of an IBM-compatible computer enable the computer to load its operating system).

Firmware can further be broken into upgradeable and non-upgradable code.

The external requirements for firmware tend to be rigid, but limited in scope, because the fulfillment of greater requirements, if any, will tend to be left to one or more of the operating
systems whose loading the firmware enables. Firmware needs to support specific hardware on one end and specific functionality on the other. It does not save documents or provide APIs.

The compatibility needs of upgradable firmware are the same as that of drivers.

Table 2: IDPD intensity by category

<table>
<thead>
<tr>
<th>Type</th>
<th>Creates Documents</th>
<th>Has API</th>
<th>Compatibility needs</th>
<th>Scope</th>
<th>IDPD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-Deployable</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Firmware</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>Platform Software</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>O</td>
<td>0</td>
</tr>
<tr>
<td>Infrastructure Software</td>
<td>-</td>
<td>+</td>
<td>+/-0</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Application Software</td>
<td>+</td>
<td>O</td>
<td>+/-0</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

The IDPD intensity can be explained as follows: non-deployable support software has no meaningful increments. Firmware and Platform Software don’t need to be compatible to previous versions, but may build on them. Infrastructure and Application Software are likely to have a major decline because of the expectations of users described in Table 2 ( - : Very Low, O : Low, + : High).
2 Background and Related Work

2.1 Major Cost Estimation Models

2.1.1 Cost Estimation Considerations

Organizations planning sizable incrementally developed projects will be interested in estimating the cost of those increments. This applies to projects that have their number of increments or requirements determined and set in advance (like custom-built applications) as well as to open-ended projects (e.g. operating systems).

Conventional cost estimation methods do not fit incremental development well because they have no provisions for incremental development. The COCOMO II cost estimation model, for example, uses cost drivers that remain constant over the course of the whole project. Because of their staying the same, they are unable to explain the changes in productivity over the course of an incrementally developed project. Therefore this model in its current form is unsuitable for the estimation of the cost of incremental development.

Two different points of views for cost predictions are possible regarding incremental projects and may be combined:

1. Trying to determine the cost of all increments before the beginning of the project, and

2. Trying to determine the cost of the next increment or remaining increments while the project is ongoing.

The first type allows us to estimate the full cost of a project and therefore to decide whether to go ahead with it in light of the cost. However, the prediction will be less precise because it cannot allow for changes in the project that arise during its course.
The second type has the advantage of allowing us to take the history of the project into account and to therefore make more precise predictions. This history cannot be known before the project starts. On the other hand, by its nature, it does not help in deciding whether it will make financial sense to go ahead with the project.

In light of the characteristics of these two approaches, it appears to make the most sense to combine them so a reasonably precise cost estimate can be made before the start of the project, but the estimate can be refined while the project is ongoing, and action can be taken where necessary in order to prevent cost getting out of hand.

For an a-priori prognosis of how the productivity of a project will develop, it would be helpful to find an average percentage factor by which software productivity declines from one increment to the next (IDPD factor). This factor may vary for different categories of projects.

In order to clearly define the scope of the research, some definitions and categorizations are needed. They follow in the next several sections of this chapter.

The general types of possible cost estimation models that have been identified (Boehm B., 1981) and could be applied to non-incremental development can also be applied to incremental development:

1. Algorithmic (also known as parametric)
2. Expert Judgment
3. Analogy
4. Parkinson
5. Price-to-Win
6. Top-Down
7. Bottom-Up

Out of these, the only one that makes sense to base a verifiable, predictive model of IDPD cost estimation is the first. This is for the following reasons:

Expert Judgment relies on the presence of experts, who may not always be available. Analogy relies on the existence of comparable projects. Parkinson (expand the work to fit the budget and resources) is not project-oriented (and not seriously predictive). Price-to-Win is also neither project-oriented nor predictive. Top-Down and Bottom-Up are lateral to our model: They can be used to refine the model further in a given case and to make the prediction more precise, but are themselves not models.

A parametric model has the advantage that it is predictive, does not depend on the availability of resources outside the project at hand (save for the ability to set the parameters reasonably) and that its accuracy and quality can be objectively measured on the basis of existing project data.

There are several different parametric cost estimation models, with different sets of parameters that don’t all map to each other. They include (STSC, 2010):

- True S (formerly Price-S)
- SEER for Software
- Sage
- REVIC
- SLIM

While it would be scientifically viable to base an IDPD model on parameters from each of these models or a synthesis of all of them, the scope of this research dictated that only one be
chosen. The choice fell on COCOMO II not because it is of higher quality than the others, but because of familiarity and the fact that if an IDPD model could be based on COCOMO II parameters, it should be possible to map it to the other models. Another aspect is that most of the collected parameters for projects that formed the basis of our case studies were for the COCOMO II model.

**2.1.2 COCOMO II Cost Drivers and Scale Factors**

It may be helpful to consider how cost drivers and scale factors behave over the course of an incremental project when looked at per increment. While cost drivers and scale factors may certainly develop in different ways depending on the specifics of a given project, it still holds some interest to attempt to predict their trends over the course of an incremental project that is managed and executed by rational actors and does not undergo a lot of upheaval. This will at the least allow an informed discussion about expectations as to whether overall there will be a decrease of productivity as well as which parameters, be they cost drivers or a scale factors, will most likely experience it.

Note that an expected trend of “None” doesn’t mean that there should not be any trend, just that it can go either way depending on the situation of the project.

While Personnel Continuity should not be influenced by the attributes of incremental development, it is very influential on other projects, which means that a company that wants to be productive in incremental development should make efforts to retain their workforce, at least during the course of a given project.
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Expected IDPD trend</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>RELY</td>
<td>Required reliability</td>
<td>Increase</td>
<td>May increase for safety/security critical systems</td>
</tr>
<tr>
<td>DATA</td>
<td>How much test data used per line of code</td>
<td>Increase</td>
<td>May increase for big-data analytics</td>
</tr>
<tr>
<td>CPLX</td>
<td>Complexity of the project</td>
<td>Increase</td>
<td>May increase with need to become part of system of systems or to support increasing demands</td>
</tr>
<tr>
<td>RUSE</td>
<td>Instances of reuse</td>
<td>Increase</td>
<td>Reused components may need further capabilities</td>
</tr>
<tr>
<td>DOCU</td>
<td>Amount of documentation</td>
<td>None</td>
<td>Maybe some documentation growth</td>
</tr>
<tr>
<td>TIME</td>
<td>Percentage of CPU time used</td>
<td>None</td>
<td>Only relevant to very large projects; will increase then</td>
</tr>
<tr>
<td>STOR</td>
<td>Percentage of storage space used</td>
<td>None</td>
<td>Only relevant to very large projects; will increase then</td>
</tr>
<tr>
<td>PVOL</td>
<td>Volatility of platform used for development</td>
<td>Decrease</td>
<td>In the beginning, platform hasn’t solidified yet</td>
</tr>
<tr>
<td>ACAP</td>
<td>Analyst capability</td>
<td>Decrease</td>
<td>Will rise if personnel continuity good</td>
</tr>
<tr>
<td>PCAP</td>
<td>Programmer capability</td>
<td>Decrease</td>
<td>Will rise if personnel continuity good</td>
</tr>
<tr>
<td>PCON</td>
<td>Personnel continuity</td>
<td>Increase</td>
<td>High turnover reduces productivity plus affecting experience factors</td>
</tr>
<tr>
<td>APEX</td>
<td>Applications experience</td>
<td>Decrease</td>
<td>Will rise if personnel continuity good</td>
</tr>
<tr>
<td>PLEX</td>
<td>Platform experience</td>
<td>Decrease</td>
<td>Will rise if personnel continuity good</td>
</tr>
<tr>
<td>LTEX</td>
<td>Language and tools experience</td>
<td>Decrease</td>
<td>Will rise if personnel continuity good</td>
</tr>
<tr>
<td>TOOL</td>
<td>Quality of development tools used</td>
<td>Decrease</td>
<td>Better tools will be acquired over time</td>
</tr>
<tr>
<td>SITE</td>
<td>Collocation and communication</td>
<td>Decrease</td>
<td>If continuity is good, teams grow closer</td>
</tr>
<tr>
<td>SCED</td>
<td>Percentage of needed time available</td>
<td>Increase</td>
<td>Schedule compression becomes more expensive as system size grows</td>
</tr>
</tbody>
</table>
Table 4: Effect of COCOMO II Scale Factors on IDPD

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Expected IDPD trend</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>PREC</td>
<td>Precedentedness of the project</td>
<td>Decrease</td>
<td>Project becomes more familiar</td>
</tr>
<tr>
<td>FLEX</td>
<td>Procedural flexibility</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>RESL</td>
<td>Architectural, risk resolution</td>
<td>Increase</td>
<td>Weak architecture increases rework</td>
</tr>
<tr>
<td>TEAM</td>
<td>Quality of team interactions</td>
<td>Decrease</td>
<td>If continuity is good, customer-developer teams grow closer</td>
</tr>
<tr>
<td>PMAT</td>
<td>Process maturity</td>
<td>Decrease</td>
<td>CMMI level increases should reduce rework</td>
</tr>
</tbody>
</table>

All this means that in general, numerous factors can affect the effort involved in creating a new increment. As these can vary by increment, they can cause productivity declines and increases from increment to increment, but overall IDPD effects should be evident.

2.2 Cost Model for Maintenance and Reuse

Maintenance can refer to upkeep effort that has to be expended on

- The codebase,
- Its documentation, and
- Training of personnel and users.

Work on the codebase can consist of

- Adding code,
- Modifying code,
- Deleting code.
There are different models for counting the code that is being added, modified and deleted. One option is of course to fully count the SLOC.

For cases of reuse (for code that is treated as a black box) and adaptation of code (white box), in some common models (such as COCOMO II (Boehm, et al., 2000)) the effort is calculated by converting the code size to equivalent SLOC (ESLOC) based on how much effort will have to go into the integration of that code.

COCOMO II has a nonlinear reuse model that takes into account the understanding of the software (SU), the degree of assessment and assimilation necessary to integrate fully reused software modules and the unfamiliarity of the programmers with the software (UNFM). Additionally, the percentages of the existing software’s design, code and previous integration effort that would be modified are taken into account as DM, CM and IM parameters.

For maintenance, COCOMO II uses the same model if the amount of the code base changed is less than or equal to 20%. Above that, a maintenance model is used that takes into account code size, SU and UNFM. Documentation will have to be updated as needed. Training will have to be done to the degree needed in order to keep up productivity.

While there is always a basic level of necessary maintenance due to factors such as technological progress, discovery of bugs and others, there are situations where peaks occur. These have included Y2K (addressed by fixing the date format and handling of years) and Sarbanes-Oxley (changes in accounting standards).

Maintenance has also been necessary in many cases in order to adapt to 64-bit addressing (transition from 32 to 64 bit largely took place in the first decade of the 2000s) and to adapt to paradigm changes, such as network security (awareness of which has started in the late 1960’s as
the Rand Corp., and reached critical mass with the formation of CMU-SEI Computer Emergency Response Team (CERT) in the late 1980s). Other occasions have included migration from mainframe to desktop computers (1970s/80s) command line oriented operations to GUI-focused ones (1980s/90s) and from desktop to mobile computing (ongoing, started in the early 2000s).

2.3 Laws of Software Evolution

The increasing pace of change in software-intensive systems has increasingly shifted software engineering processes from single-pass, fixed-target development to a multiple-pass, evolutionary development process. Among other changes, this has changed the scope of software evolution to include not only relatively small maintenance changes to large operational systems, but also evolutionary development of similar-sized increments. This raises questions about the applicability of traditional principles or laws of software evolution to current and emerging forms of evolutionary development.

Understanding software evolution improves the ability to create processes for effective and consistent software development. Software evolution is defined as the process of going through the development, deployment and maintenance of software systems.

2.3.1 Lehman’s System Types

Lehman (Lehman & Belady, 1985) categorizes all software systems to fit one of three types: S, P or E. This classification was derived from a series of empirical observations Lehman and Belady made regarding large system development.
S-Type Systems

S-type or static-type systems are based on static, formal and verifiable sets of specifications with easy to understand solutions, such as firmware incorporated in computer hardware. These simple systems are generally not subject to evolution and therefore Lehman’s laws don’t apply.

P-Type Systems

P-type systems, or practical-type systems, can have their requirements defined precisely and formally, but the solution is not immediately apparent and its acceptability is dependent on the environment in which the program is being used. The key difference to S-type is that those are written to adhere to a static specification, but P-type programs are iteratively developed in order to improve their effectiveness.

E-Type Systems

The final type of system proposed by Lehman is the E-type, or embedded-type. E-type programs are defined as all programs that ‘operate in or address a problem or activity of the real world’. Their evolution is a direct consequence and reflection of ongoing changes in a dynamic real world. Lehman’s laws only describe the behavior of these projects, which mechanize human or societal activities and become part of the world they model (Lehman M., 1980) (Lehman, Ramil, Wernick, & Perry, 1997).

2.3.2 Lehman’s Laws of Software Evolution

Lehman's laws describe how E-type software projects evolve over their releases. The original study Lehman and Belady conducted studied IBM software development of OS/360 in the 1970s. After this study, additional papers were written which expanded and amended the set of laws.
For non-trivial incremental development, the earlier increments will exist for some time before the project is finished (if there even is an end planned – some projects have continued to evolve). They will therefore either evolve or at least have a considerable age when the later increments are built.

This applies to some, but not all programs observed here. These laws have been reviewed in the context of several major open source software projects, with some found to hold and others not (Xie, Guowu, Chen, & Neamtiu, 2009).

The following is a discussion about the individual Laws of Software Evolution as stated by Lehman in (Lehman M., 1980) (Lehman, Ramil, Wernick, & Perry, 1997).

Some notes and caveats for the discussion are:

- The effort attributed to an increment also includes effort spent on previous increments during the time the new increment is developed.

- Usefulness is regarded as an indispensable part of what makes the quality of a piece of software sufficient. The authors posit that while other attributes than usefulness are part of the quality of software, developing a useless piece of software is pointless.

- There is no distinction made or intended between the terms “program” and “system”. Reflecting the evolution of Software Engineering toward Systems and Software Engineering, Lehman’s focus has shifted from software to systems over the 17 years that passed between the publications of (Nguyen, 2010) and (Lehman, Ramil, Wernick, & Perry, 1997), but the laws are intended for both, although the data support is based on software.

A “Reasoning” section is provided when a law’s statement is in need of explanation.
2.3.3 Discussion of the individual Laws of Software Evolution

First law: Continuing Change

Statement

A program that is used and that as an implementation of its specification reflects some other reality, undergoes continual change or becomes progressively less useful. The change or decay process continues until it is judged more cost effective to replace the system with a recreated version (Lehman M., 1980). Note that this “law” does not apply to many embedded software programs, which usefully control thermostats, carburetors, or most elevators because those programs tend to have highly stable requirements.

Application to IDPD

Incremental development means that increments build on each other. After each increment, there is a fully functional and tested program (or system – Lehman talks of “programs” in (Lehman, Ramil, Wernick, & Perry, 1997) and “systems” in (Lehman, Ramil, Wernick, & Perry, 1997) reflecting the widening scope of software engineering, the terms will therefore be used interchangeably for the duration of this discussion).

That the program is used and reflects reality just means that it is an E-type system. The extensions to systems also excludes S-type systems such as roads, bridges, and piers.

The consequence of this law for the parts of the evolving systems to which it applies is that from the point in time that any increment gets released, the quality of the code base will decay unless effort goes into changing or replacing it. The change mentioned in the law can be any type of maintenance or even design change. It will result in the addition, deletion or modification of code. It is possible not to put any effort into this part of the existing codebase, but this will result in a
reduction of quality and is therefore not a reasonable course of action for a rational actor. While no effort would be incurred in this case, money would be effectively spent due to the reduction of the quality of the overall system. Over time, the utility of the system would settle at a low point or degrade all the way to zero.

Assuming that the effort that can go toward a given increment is largely invariant (as stated by the Fourth Law further below), this means that a part of the effort for that increment will have to be directed toward keeping the existing code base useful. Together with the fact that the code base is growing over time as increments are added over time, this means that the part of the effort devoted to maintaining the quality of the code base will steadily increase and that the part that can be used to develop new code will steadily decrease. Even assuming that all other parameters stay the same (i.e. the productivity as expressed in SLOC per person-hour is the same for later increments), this will result in a lower new code output per time interval, therefore reducing the productivity in terms of new SLOC per person-hour.

The only situation in which this effort could be negligible is if the increments are developed in rapid succession or in cases where the reduction in quality is minimal.

Summary

The productivity in later increments will decline due to the rework and maintenance that has to be done on the earlier ones. However, little quantitative information has been available, except for generic information about overall effort breakdowns (Lientz, & Swanson, 1980), domain-dependent maintenance staffing levels (Boehm B., 1981), and recalibration of software maintenance effort estimation models (Nguyen, 2010). The validity of this “law” will be tested by
hypotheses 1: i.e., if IDPD=0, then no effort was needed to maintain the earlier increments quality and value.

Second Law: Increasing Complexity

Statement

As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless effort is applied to maintain or reduce it (Larman, 2004).

Application to IDPD

As has been stated in the discussion of the First Law, if usefulness is to be retained, the existing code base will have to be constantly changed (i.e. maintained) as the system evolves. The Second Law reflects the fact that with all maintenance, there are two layers of effort that need to be addressed if the quality of the whole system is to be kept equal:

1. The coding work itself

2. Integration of the coding work done into the system in terms of code integration, documentation, adaptation of the design and rework of its other sections.

The second kind of effort is addressed in this, the Second Law. As with the First Law, there is the option of not spending the effort, but the outcome will – again – be a loss of quality because the individual parts of the system will become less and less integrated, creating “rough patches” that are the results of adding code that is not well integrated, be it to fix deficiencies or to add functionality.

As in the case of the First Law, increments will require progressively more effort due to this, reducing the productivity from increment to increment.
If no effort is being directed toward reducing the complexity, the complexity will still add to the effort of the later increments by way of making their development more complex in turn because the more complex a previous increment becomes, the more complex it will become for the later increments to integrate with the previous ones.

**Summary**

Changing previous increments adds complexity, which in turn adds to the effort needed for the later increments. Again, though, quantification of this effect has been scarce and difficult. Again the validity of this “law” will be tested by hypotheses 1.

**Third Law: Fundamental Law of Program Evolution**

**Statement**

Program evolution is subject to a dynamics which makes the programming process, and hence measures of global project and system attributes, self-regulating with statistically determinable trends and invariances (Larman, 2004).

**Reasoning**

This law is a generalization of the fourth and fifth laws (Larman, 2004). As with the other laws, laws 3-5 have large major exceptions, such as the effects of Y2K and Sarbanes-Oxley, and numerous individual exceptions, such as the effects of mergers and acquisitions on corporate infrastructure software.

**Application to IDPD**

This means that any change or variance in one system attribute will also be relevant for all others and that given enough information, the interdependencies of variables in a system should be predictable.
More specifically, the Fourth and Fifth Law state that organizations gravitate toward stability and that organizational dynamics and the need to maintain familiarity with the system impose upper limits on the output (that is, if that output is supposed to have any acceptable level of quality and the growth of the system is to be maintainable).

A fitting metaphor is therefore that of a cover that gets pulled in different directions. For IDPD, this means that effort spent on one part of the system will directly influence how much effort can be spent on another. The same applies for different features of the system.

One possible example is that time spent on making the system more secure would take away from the time spent on improving the user interface. (This is because the law of Conservation of Organizational Stability says that the work rate per increment is invariant, see below.) Another interpretation of the Third Law is that the evolution of a program is a predictable process where similar parameters should yield similar results. This means that the decline in productivity over the increments – IDPD – should also be predictable.

**Summary**

The amount of effort to be spent on any given increment is limited and there are interdependencies between effort parameters. Productivity is often predictable based on parameters (Nguyen, 2010). The validity of this “law” will be tested by hypotheses 2: i.e. statistical invariance implies that IDPD will be relatively constant across increments.

**Fourth Law: Conservation of Organizational Stability (Invariant Work Rate)**

**Statement**

During the active life of a program the global activity rate in a programming project is statistically invariant (Larman, 2004).
**Reasoning**

Organizations are striving for stability and there is a point where “resource saturation” is reached and more resources will not make a difference (Larman, 2004). Frequently, organizational inertia leaves a maintenance activity with a relatively fixed budget or staffing level over the years.

**Application to IDPD**

General software engineering experience contradicts at least the literal meaning of this law: The activity rate can very well vary over the course of a project. This is evident because it is always possible to reduce the level of activity from the previous one (unless the initial one is already nil, but then there would be no actual programming project and the law would not apply). The law therefore needs to be interpreted.

A useful interpretation of this law is that there is an upper limit to the resources that can reasonably be committed to and the effort that can go into the development of a given increment. Beyond that limit, the effect of committing more resources or effort will either be little to no benefit or even detrimental. (This is in line with the well-known insight that “adding manpower to a late software project makes it later” (Brooks, 1975).

Therefore, according to this law, there is no way for an organization to address the maintenance and integration effort that needs to go into existing increments by simply expending more resources in the way of hiring additional programmers. It is inevitable that this effort will take away from the activities that can be done on the later ones, reducing the productivity achievable for the new increment.
Summary

Beyond a certain upper limit, adding more resources or effort does not benefit the system in a meaningful way. Again, there is exceptions, such as Y2K. Again, statistical invariance will be tested by hypotheses 2.

Fifth Law: Conservation of Familiarity (Perceived Complexity)

Statement

As an E-type system evolves all associated with it, developers, sales personnel, users, for example, must maintain mastery of its content and behavior to achieve satisfactory evolution. Excessive growth diminishes that mastery. Hence the average incremental growth remains invariant as the system evolves (Lehman M., 1980).

Application to IDPD

Everyone working on an incrementally developed project must maintain mastery of its content and behavior. The bigger the project gets, the more effort will have to be spent on this. Therefore, since the size of the project will grow over the course of its development, more and more effort will have to go into this and less will be able to go into new development.

If it is considered that the work rate is invariant, then there is an upper bound to the amount of effort that can go into any increment. Since the increment to be developed needs to make use of the existing increments, these will have to be maintained. The law of Conservation of Familiarity states that there is such a thing as “mental maintenance” which needs to be performed on the minds of the people working with the existing increments because otherwise, regardless of their quality, there is nobody who can integrate them.
Summary

Even if an increment may have solidified to the point that it needs little or no maintenance anymore, effort will have to go into the mental states of the people working with it. Excessive growth may be attempted for an increment, but the mastery of the system will have to keep up with the increments, so either the growth will not be able to take place or more training will have to be done after the increment, which reduces that increment’s productivity and evens it out.

Sixth Law: Continuing Growth

Statement

The functional content of E-type systems must be continually increased to maintain user satisfaction over their lifetime (Lehman M. , 1980).

Application to IDPD

All systems evaluated here are E-type systems. The systems are fully tested and functional after each increment. The lifetime of each increment runs until the overall system is decommissioned.

Therefore the functional content of every increment must be continually increased over the lifetime of the overall system if user satisfaction is to be achieved (which can reasonably be assumed to be the goal of the development of any system). The effort on this will decrease the productivity of the current increment because it is done during the same time, and has only the budget for the current increment.
Seventh Law: Declining Quality

Statement

The quality of E-type systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes (Lehman M., 1980).

Application to IDPD

This “law” looks like a weaker phrasing of the one about Continuing Change that adds nothing of value: It’s not relevant for software development and evolution whether the quality of an E-type system “appears” to be declining. The decisive question is whether it actually does. If it does not, then whether or not the systems are maintained and adapted is irrelevant. This law is therefore only of relevance if that section is changed to “will decline unless”. If it is considered additionally that E-type systems respond to the environment by definition and that quality and being satisfactory are the same attributed, then this is the law about Continuing Change restated with different words. The considerations about its application to IDPD are therefore the same as the ones regarding Continuing Change.

Hypotheses 1 will test this over the long run. Hypotheses 2 will test “continuality”: there may be increments that focus more on improving quality rather than functionality.

Eighth Law: Feedback System

Statement

E-type evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable base (Lehman M., 1980).
Application to IDPD

A significant amount of parameters will have to be controlled in order to make incremental development a success. The productivity of a given increment and its quality is relevant to all following increments.

Lehman E-type systems don’t exist in a vacuum. In order to stay relevant and useful to their environment, they have to react to the feedback from that environment, which they influence in turn. Reacting to the environment causes maintenance effort on the existing increments, which reduces the productivity of the later ones.

This further supports that effort going into the quality of earlier increments will improve the later increments but also take away productivity from them.

Summary

This “law” just appears to be a restatement of the definition of an E-type system. There may be different levels of interaction with the surrounding world that increase or decrease the required feedback work and thus IDPD. This will be tested by hypotheses 3.
3 Research Approach

3.1 Behavioral Analysis

In order to arrive at a statistical model of IDPD, the attributes of increments as well as the parameters of the projects, their personnel and their environments have to be collected. Therefore, for each increment of projects that had available data, the following has been collected:

- The amount of code added, modified and deleted for each increment in SLOC
- The dates of their increments (where available)
- COCOMO II cost driver and scale factor levels (to the extent that individual drivers and factors were available)

The collected environmental data (cost drivers, scale factors) is then put in relation to the quantitative data of the increments themselves (SLOC, dates) and compared via such methods as correlation and regression analysis.

3.2 Data Collection and Analysis

Due to the diverse nature of incremental software projects, the data collection process involves establishing and applying consistent criteria for the inclusion of completed incremental projects. Data has been collected from various sources that fall into the three main categories of

- Software industry
- Controlled experiments
- Open source
As with most software engineering research, opportunities for data collection are limited (Valerdi & Davidz, 2009) due to the size and length of most software engineering projects, and the fact that many are developed for proprietary or military applications. Knowing this, software engineering project data was still solicited from industry affiliates supporting this research. Due to the diverse nature of incremental software projects, the data collection process involves establishing and applying consistent criteria for inclusion of completed incremental projects.

Only data from projects that satisfy all of the following criteria is collected:

- Starting and ending dates are clear
- Has at least two increments of significant capability that have been integrated with other software (from other sources) and shown to work in operationally-relevant situations
- Has well-substantiated sizing and effort data by increment
- Less than an order of magnitude difference in size or effort per increment
- Realistic reuse factors for COTS and modified code
- Uniformly code-counted source code
- Effort pertaining just to increment deliverables

As part of data collection process, I interviewed programmers, held workshops and described the model and core data attributes of the projects such as actual effort, actual size, and rating levels of the COCOMO II cost drivers. Effort is collected in person-hour and converted into person-month. Since definition of person-month might differ per organization (by 10% to 20%), the standard COCOMO model that defines 152 hours per person-month is used (Boehm, et al., 2000). The size metrics are collected by using code counting tools such as USC’s Unified Code count.
(UCC)\textsuperscript{1} to compare differentials between two baselines of the source program. The size metrics are based on the logical SLOC definition, adopted from the Software Engineering Institute (Park, 1992) and adopted into the definition checklist for source statement counts (Boehm, et al., 2000). This checklist defines an executable statement as one logical SLOC, while non-executable statements such as blanks and comments are excluded.

### Table 5: Core collected data attributes

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product</td>
<td>Name of the software product</td>
</tr>
<tr>
<td>Build</td>
<td>Build number. A software product that has multiple builds (increments)</td>
</tr>
<tr>
<td>Effort</td>
<td>Total time in person-hours for a given increment</td>
</tr>
<tr>
<td>New SLOC</td>
<td>SLOC count of new modules</td>
</tr>
<tr>
<td>Reused SLOC</td>
<td>SLOC count of reused modules</td>
</tr>
<tr>
<td>COTS SLOC</td>
<td>SLOC count of COTS modules</td>
</tr>
<tr>
<td>CM</td>
<td>The percentage of modified code</td>
</tr>
<tr>
<td>DM</td>
<td>The percentage of design modified</td>
</tr>
<tr>
<td>IM</td>
<td>The percentage of implementation and test needed for the reused/COTS modules</td>
</tr>
<tr>
<td>SU</td>
<td>Software Understanding</td>
</tr>
<tr>
<td>UNFM</td>
<td>Programmer Unfamiliarity</td>
</tr>
<tr>
<td>Cost drivers</td>
<td>Rating levels for 22 cost drivers</td>
</tr>
</tbody>
</table>

\textsuperscript{1} http://csse.usc.edu/ucc_wp/
3.2.1 Controlled Experiment

In order to have better control over data collection and project parameters, a controlled experiment was conducted with four simulated application software projects that were given to 21 graduate students of Computer Science. The amount of new requirements and personnel composition of the projects and related teams were changed throughout the project at irregular intervals. The number of requirements ranged between zero and more than the teams could reasonably be expected to finish. In some cases, impossible requirements were given.

It turned out that in the experiment with graduate students it was impossible to reasonably influence some cost drivers:

- TIME (Execution Time Constraint, the share of available execution time being used)
- STOR (Main Storage Constraint, the share of available storage being used)
- RELY (Required Software Reliability, the impact of a software failure)
- RUSE (Developed for Reusability, the extent to which the software is intended to be reused)

The lack of meaningful CPU and storage limitations in current hardware configurations made TIME and STOR meaningless for the small projects assigned. Since the subjects knew that their software was not going to be used in real-world situations, RELY was not going to be a modifiable influence. Similarly, due to the small scale of the projects, influencing RUSE was not practical either.

Some personnel-related cost drivers can only change slowly and in most cases within a limited ranges per person. These include ACAP, PCAP, APEX, PLEX and LTEX (Analyst Capability, Programmer Capability, Applications Experience, Platform Experience and Language
and Tool experience). They therefore had to be changed indirectly by moving personnel between teams.

Data was collected from the students before the formation of teams, after each week on their projects and at the end of the semester.

Internal validity considerations concerned mainly the veracity of the submitted time sheets, while external validity considerations were the small scope of the projects, lack of acceptance testing, different motivation of the students than that of industry programmers and the influence of IDE use on code size.

3.2.2 Open Source Projects

Collection of cost estimation parameters for open source projects would have to happen after the fact since open source organization usually do not collect this data over the course of a project. In the cases of the open source projects that were used for this research, cost estimation parameters were not collected. The focus was on observing whether the productivity in new SLOC per person-month was decreasing. While fluctuations in cost estimation parameters may explain some of the productivity variations that were observed, it was assumed that over the course of the minor versions of open source projects that SLOC data was collected here, these parameters would not vary significantly. The selection of open source projects was based on the stability of projects (stable release schedule, patch updates, etc.) and their sponsors (i.e. corporate open source). In such cases, it was assumed stable and constant staff size across increments and the effort was calculated based on the man-days between the releases.
3.2.3 Data Collection Challenges

Data conditioning is a major challenge in the software data collection and analysis process. Major issues found in the collected data received from commercial sources have included (Boehm B., Future Challenges for Systems and Software Cost Estimation, 2009):

1. Inaccurate, inadequate or missing information on modified code (size provided), size change or growth, average staffing or peak staffing, personnel experience, schedule, and effort
2. Size measured inconsistently (different tools for different increments)
3. Replicated duration (start and end dates) across all increments
4. Low number of increments (less than 3)
5. History of data is unknown

It should be obvious that any one of these issues can severely reduce the quality of the data to the level of being unsuitable for research. This issue can be addressed through the collection of data from controlled experiments and open source projects. Table 6, summarizes data sources for commercial, controlled experiment and open source software:

**Table 6: Summary of data sources**

<table>
<thead>
<tr>
<th></th>
<th>Typical KSLOC</th>
<th>SLOC collection</th>
<th>Parameter collection</th>
<th>Main issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software Industry</td>
<td>&gt;100</td>
<td>Various tools, unverifiable</td>
<td>Unverifiable</td>
<td>Secrecy</td>
</tr>
<tr>
<td>Controlled experiments</td>
<td>&lt;5</td>
<td>UCC, source available</td>
<td>Authors</td>
<td>Individuals have great influence</td>
</tr>
<tr>
<td>Open source</td>
<td>Any</td>
<td>UCC, source available</td>
<td>Authors</td>
<td>Hard to collect parameters</td>
</tr>
</tbody>
</table>
In order to collect cost estimation parameters for open source projects, this needs to happen after the fact since open source organizations usually do not collect this data during the course of a project, particularly for effort expended by contributors.

### 3.3 Contextual Analysis

While a fully detailed statistical model of IDPD has been elusive, there may be different classes of incremental projects that exhibit different patterns or levels of IDPD due to their different characteristics.

One approach is to separate projects into domains by their position in the hierarchy between applications, as shown in Table 7, on the top and firmware on the bottom, with some consideration given to support software (Refer to 1.4.5 detailed explanation of project domains). Envisioned domain taxonomies include project size, project time frame, functional domain or which part of the software industry a project is most closely related to.

<table>
<thead>
<tr>
<th>Table 7: Incremental project domains</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
<th>Expected IDPD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Software</td>
<td>Applications that may build on infrastructure software</td>
<td>+ (High)</td>
</tr>
<tr>
<td>Infrastructure Software</td>
<td>Software that provides services to other software</td>
<td>++ (Higher)</td>
</tr>
<tr>
<td>Platform Software</td>
<td>Drivers that are integrated with hardware</td>
<td>0 (Medium)</td>
</tr>
<tr>
<td>Firmware (Single Build)</td>
<td>Simple cases of firmware that cannot be updated</td>
<td>0 (Medium)</td>
</tr>
<tr>
<td>Support Software</td>
<td>Non-deployable throw-away code</td>
<td>n/a</td>
</tr>
</tbody>
</table>
3.4 Statistical Analysis

3.4.1 Linear Correlation

Linear correlation is used to measure the strength of the linear association between two paired sets of data. By computing the correlation coefficient, which is a number between -1 and 1, the degree of the relationship between two paired sets of data can be determined. The closer to 1 it is, the more confidence there is of a positive linear correlation and the closer to -1 it is, the more confidence there is of a negative linear correlation. When the correlation coefficient is zero or close to zero, there is no evidence of any relationship. Corresponding p-values are looked up in the standard statistical tables based on the size of data set and its correlation coefficient. The p-value is a number between 0 and 1, which represents the probability that this data would have arisen if the null hypothesis were true.

3.4.2 Tailed Pair T-Test

A t-test is used to compare the means from two sets of data in order to test the probability to accept the null hypothesis. When the two sample sets of data are dependent, a paired t-test is used. Since the difference between the means is unknown and the first mean can be different from the second in either direction, a two-tailed t-test is performed. Confidence intervals express the range within which a result for the whole population would occur for a particular proportion of times an experiment or test was repeated among a sample of the population. It is computed from the following formula:

\[
\text{Confidence Interval} = [\text{mean} - \text{Confidence Value}, \text{mean} + \text{Confidence Value}]
\]
A confidence level refers to the percentage of all possible samples that can be counted as reliable. For example, if a significance level of 0.05 is used, the 2-tailed test uses half of the significance level to test the significance in one direction and the other half in the other direction. That means a significance level of 0.025 in each tail of the normal distribution of the test statistics.

The null hypothesis can be either accepted or rejected based on the significance level of the results. There are two methods that are used to determine the significance of the results. One is to compare the p-value of the sample with the predetermined significance. If p-value is less than the threshold, then it indicated strong evidence against the null hypothesis. Otherwise, the null hypothesis cannot be rejected. The other way is to compare the absolute value of t-stat with confidence value at the n-1 degrees of freedom. Confidence value is computed from a normal distribution with confidence level, standard deviation and the number of observations in the given sample.

T-stat is computed from the following formula:

$$ t_{stat} = \frac{M - \mu_0}{SD/\sqrt{N}} $$

M is the sample mean and $\mu_0$ is the hypothetical mean if the null hypothesis were true. SD is the standard deviation and N is the number of subjects in the sample.

For a 2-tailed t-test, the null hypothesis can be rejected if your sample t value is more extreme than the critical t value in either positive or negative direction. If $|t_{stat}| < \text{Confidence Value}$, then the null hypothesis cannot be rejected at the given confidence level, which means that the probability of getting get a confidence value at least as extreme as the t-stat value is greater than the given confidence level, if the null hypothesis were true.
3.4.3 One-Way ANOVA

One-way ANOVA is a way to test the equality of three or more means at one time by using variances. In this case, the question that needs to be answered is whether the IDPD of the three categories differs in a statistically significant way. This can be ascertained using one-way ANOVA, which is a way to determine whether the means differ significantly. An overall F test is performed to determine if there is any significant difference existing among any of the means. It is calculated by the division of between-groups variance and within-groups variance. Between Groups variance is the explained variance that is due to the independent variable, the difference among the different categories (referring to the categories in table 7). For example the difference between the overall IDPD decline in application domain and the overall IDPD decline in platform domain would represent explained variance. Within Groups variance is the variance within individual groups, variance that is not due to the independent variable. For example, the difference between the overall IDPD decline for one project in application domain and another project in the same group would represent the within groups variance.

F-ratio is the ratio of two sample variances and is computed to determine the p-value. If p-value is well below the predefined significance level, then it can be concluded that the groups are statistically significantly different from one another.
4 **Analysis and Representation of Data**

This section summarizes the key data analyses and their results conducted in building and testing the IDPD hypothesis. Section 4.1 provides an overview of the data selection and normalization results that defines the baseline data sets for the data analyses. Sections 4.2, 4.3 and 4.4 reports on the data analyzed for productivity decline, build-to-build behavior and domain hypotheses respectively. Section 4.5 reports on the threats to validity.

4.1 **Summary of Data Collection and Selection**

Data selection and normalization are completed before most data analyses are started. A set of project data points, where collected from industry affiliates, open source projects and controlled experiments. As discussed in Section 3.4, simple and straight forward browsing through data points helped us eliminate most defective (missing effort, size, or important data definitions such as counting method or domain information) and duplicated data points.

Error! Not a valid bookmark self-reference. plots a summary of data and the productivity for each project that was used in this study. Due to the large range of values, data was presented on a logarithmic scale. Figure 4 provides a histogram showing the distribution of projects by size.
Figure 3. Productivity for all projects on a logarithmic scale

Figure 4. Average SOLC of projects
4.2 Productivity Decline: Hypothesis 1

The Productivity Decline Hypothesis states, in incrementally developed software projects that have coherence and dependency between their increments, productivity declines over their course.

Several data points have been analyzed for their IDPD effect and we evaluate several ways of finding a model that fits our observed productivity declines. The data used for this study are mostly in the application and infrastructure domains. Our goal is to find an approximation function that fits the data reasonably well while still having some generality and predictive quality. While some types of trend line can be excluded purely due to their shape not being a good fit in general for purely logical reasons, others need to be evaluated on goodness of fit metrics.

Although a linear trend line could conceivably turn out to be the best fit over a stretch or even all the observed data during the course of a project, its predictive quality is inevitably negated.
when there is a productivity decline. This is because a decreasing trend line will cross the x-axis. After that point, a linear trend line would therefore predict that productivity is negative, which wouldn't be a credible prospect for investment in incremental development.

That alone would not necessarily mean that there’s a problem, because one limitation of our productivity model is that it uses SLOC as a measure of output. In many cases a product can actually become better when its SLOC are reduced, such as when repetitious code becomes converted into a subroutine or when code is optimized. Such optimization work is undoubtedly productive, though it would not be productive as far as the productivity model is concerned. Therefore it would be possible to have several iterations where SLOC is reduced. However, the ultimate problem with a linear trend line and its predictive quality would be that when a project continues long enough, inevitably its overall SLOC would fall below zero SLOC, which is not feasible.

Another exclusion situation arises with polynomial approximations. Here, with a high enough order of the polynomial, any number of observations can be approximated fully in a way that makes the coefficient of determination equal to 1. As with the previous case of the linear line, the problem is the predictive quality: The polynomial would simply predict that the productivity would continue to increase or decrease depending on what way the curve went for the last increment, with possibly grotesquely “wild” results.

4.2.1 Case Study 1: Quality Management Platform

The Quality Management Platform (QMP) is a medium-size commercial software application that serves to improve the software development process for small and medium-sized organizations. Functionalities such as process definition, project planning, data reporting,
measurement and analysis, defects tracking etc. were incrementally developed as its main features over six years. More features are planned and scheduled to be added in the future. The increments were originally regarded as maintenance projects for the first build, but later re-categorized as incremental development due to showing characteristics of incremental development (Tan, 2009).

Figure 6 shows the productivity over six increments. As tends to be the case in all productivity graphs so far, a logarithmic trend line (R-squared of 0.53) fits the data better than a linear one (R-squared of 0.37), though neither fits it particularly well since the R-squared is not very close to 1. This is because of the increase in productivity in increments 4 and 5, followed by another decrease in increment 6.

The authors of the study on the project explain the productivity changes in the fourth and following increment mostly using COCOMO II cost drivers (Tan, 2009) as shown in Table 8.

<table>
<thead>
<tr>
<th>Build</th>
<th>TEAM</th>
<th>PCON</th>
<th>ACAP</th>
<th>PCAP</th>
<th>APEX</th>
<th>LTEX</th>
<th>PLEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build 1</td>
<td>NOM</td>
<td>HI</td>
<td>NOM</td>
<td>NOM</td>
<td>NOM</td>
<td>HI</td>
<td>VLO</td>
</tr>
<tr>
<td>Build 2</td>
<td>HI</td>
<td>HI</td>
<td>NOM</td>
<td>NOM</td>
<td>HI</td>
<td>HI</td>
<td>LO</td>
</tr>
<tr>
<td>Build 3</td>
<td>HI</td>
<td>HI</td>
<td>NOM</td>
<td>NOM</td>
<td>HI</td>
<td>HI</td>
<td>NOM</td>
</tr>
<tr>
<td>Build 4</td>
<td>HI</td>
<td>HI</td>
<td>NOM</td>
<td>NOM</td>
<td>NOM</td>
<td>HI</td>
<td>HI</td>
</tr>
<tr>
<td>Build 5</td>
<td>VHI</td>
<td>LO</td>
<td>NOM</td>
<td>NOM</td>
<td>NOM</td>
<td>HI</td>
<td>HI</td>
</tr>
<tr>
<td>Build 6</td>
<td>VHI</td>
<td>LO</td>
<td>HI</td>
<td>NOM</td>
<td>NOM</td>
<td>HI</td>
<td>HI</td>
</tr>
</tbody>
</table>

The conclusion from this case study of QMP is that while IDPD happens when cost drivers are largely constant, a large improvement in those drivers can offset IDPD and lead to increases in productivity, at least over a few increments. Once the improvements have been fully realized, IDPD will set in again. This is what we’re seeing in the sixth increment here. (Even if we assume that the
cost drivers all improve up to COCOMO II’s optimal values, they can eventually not be improved upon anymore, so IDPD will set in again.)

4.2.2 Case Study 2: XP1 and XP2

Projects XP 1 and XP 2 were two commercial web-based client-server systems for data mining developed in Java using a process similar to Extreme Programming (XP). The programmers were graduate students and detailed logs of activities were kept (Alshayeb & Li, 2006). Figure 7 and Figure 8 show the productivity over different increments for projects XP1 and XP2 respectively.

Judging by the coefficient of determination, R-squared, a logarithmic trend line is the best here at 0.60, followed by a power one at 0.50 and an exponential one at 0.46.

Figure 6: QMP productivity across increments
“No data was available on the number of changes and the effort per change, but the percentage of total story development effort by story number for Project 1 shows an average increase from one story to the next of about 6% per story number for refactoring effort and about 4% percent per story for error fix effort. The corresponding figures for project 2 are 5% and 3%. These are nontrivial rates of increase, and while clearly not as good as the anecdotal experiences of agile experts, they are more likely representative of mainstream XP experience. The small decrease in rates from Project 1 to Project 2 indicates there was a small, but not dominant, XP learning curve effect” (Bohem & Turner, 2003).

Refactoring and error fixing efforts increased over time. The new design effort seems to be the dominating effort and the error fixing effort is the smallest of all efforts. The new design effort was negatively correlated with refactoring and error fixing. There does not appear to be any correlation between refactoring and error fixing efforts (Alshayeb & Li, 2006).
4.2.3 Statistical Significance

Productivity Decline Hypothesis is tested by linear correlation model to assess the relationship between build-to-build productivity and number of increments. The result shown in table 9 indicates a negative correlation between build-to-build productivity and number of increments. If increment increases, productivity falls. In addition, probability values (p <0.1) for each project show a strong negative relationship between build-to-build productivity and number of increments. Thus these results are compatible with Lehman “laws” 1 on Continuing Change and 2 on Increasing Complexity.

Figure 9 illustrates the normalized productivity of eight of our case studies over their increments, with 1.00 being the productivity of the initial increment. While the productivity changes per increment vary greatly and in some cases a lot of productivity can be gained back between increments, no project has been able to keep its productivity at the level of the initial increment. For
any given observation, the loss of productivity (IDPD) from the beginning of the project divided by the number of increments up to that point ranges from 4% to 91%.

Table 9. Correlation coefficient (with p-value) between build-to-build productivity and number increments

<table>
<thead>
<tr>
<th>Project Names</th>
<th>Correlation Coefficient</th>
<th>P – value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-0.9265</td>
<td>0.0120</td>
</tr>
<tr>
<td>2</td>
<td>-0.8243</td>
<td>0.0219</td>
</tr>
<tr>
<td>3</td>
<td>-0.7500</td>
<td>0.0430</td>
</tr>
<tr>
<td>4</td>
<td>-0.8575</td>
<td>0.0069</td>
</tr>
<tr>
<td>5</td>
<td>-0.8675</td>
<td>0.0282</td>
</tr>
<tr>
<td>7</td>
<td>-0.9152</td>
<td>0.0425</td>
</tr>
<tr>
<td>8</td>
<td>-0.7097</td>
<td>0.0242</td>
</tr>
<tr>
<td>Apache 1.3</td>
<td>-0.5813</td>
<td>0.0012</td>
</tr>
<tr>
<td>Apache 2.0</td>
<td>-0.6938</td>
<td>0.0002</td>
</tr>
<tr>
<td>Perl 5.004</td>
<td>-0.7756</td>
<td>0.0616</td>
</tr>
<tr>
<td>Perl 5.005</td>
<td>-0.9473</td>
<td>0.0264</td>
</tr>
<tr>
<td>GFS Kernel</td>
<td>-0.5372</td>
<td>0.0028</td>
</tr>
<tr>
<td>php 4.3</td>
<td>-0.7424</td>
<td>0.0110</td>
</tr>
</tbody>
</table>

Figure 10 shows the slopes of the productivity of the case studies between the first and last increments. The IDPD factors vary between 6.44% and 37.20% per increment, with an arithmetic average of 16.79% and the median being 14.11%. The range of the slopes for straight lines from the first to the last increment is narrower: The slopes vary between 5.49% and 15.09%, with an average of 10.07% and a median of 10.50%.
It is easy to argue that there is IDPD in projects that have a certain degree of coherence between increments. One real-world case study in which cost drivers stayed largely constant showed a near perfect alignment with a trend line using a power formula.
Figure 11. Overview of productivity trends across projects
Another case study shown was not aligned well with a trend line, but had increases with productivity that could be explained with an improvement in project parameters, such as COCOMO II cost drivers.

All the projects selected for this study are shown in Figure 11. For each project, the data points represent normalized productivity value for a specific build. The linear trend line is shown for each project across different builds.

### 4.3 Build-to-Build Behavior: Hypothesis 2

Build-to-Build behavior hypothesis states the rate of productivity decline from increment to increment is relatively constant. Although some projects and “Laws” (Lehman M., 1980) (Lehman, Ramil, Wernick, & Perry, 1997) suggest that there is a statistically invariant percentage of productivity decline across increments, this may not be the case in general.

Build-to-Build behavior Hypothesis was tested by 2-tailed pair t-test model to assess the relationship between build-to-build slope and overall slope. Results are shown in table 10. Typically, a null hypothesis states that there is no actual relationship between variables. Since the hypothesis here declares that IDPD could decline, rise or stay the same, in order to form a statistically correct test, the hypothesis and its null hypothesis was reversed for the testing purpose. In other words, the hypothesis is that IDPD is constant across all increments and the null hypothesis is that the build-to-build IDPD could decline, rise or stay the same. T-tests were performed on the sample data to examine whether or not the null hypothesis could be rejected.

The significance level $\alpha$ is set to 0.05, which corresponds with a confidence level of 95%. All the $p$-values (in Table 10 and Figure 12) are much bigger than the predetermined significance level.
level \((p < 0.1)\). It indicates that the null hypothesis cannot be rejected under the observed result. Therefore, based on the sample data, it cannot be concluded that the change of productivity from build-to-build remains constant, which means the productivity could decline, rise or stay the same from one increment to the next.

### Table 10. Confidence Value, t-stat value, and p-value of 2 tailed pair t-test

<table>
<thead>
<tr>
<th>Project Name</th>
<th>Confidence Value (with (\alpha = 0.05))</th>
<th>T-stat (absolute value)</th>
<th>P-value (2 tailed probability)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.1920</td>
<td>0.1589</td>
<td>0.8835</td>
</tr>
<tr>
<td>2</td>
<td>0.2246</td>
<td>0.1514</td>
<td>0.8875</td>
</tr>
<tr>
<td>4</td>
<td>0.2459</td>
<td>0.0816</td>
<td>0.9355</td>
</tr>
<tr>
<td>5</td>
<td>0.1317</td>
<td>0.0539</td>
<td>0.9578</td>
</tr>
<tr>
<td>7</td>
<td>0.1726</td>
<td>0.0229</td>
<td>0.9874</td>
</tr>
<tr>
<td>Sendmail</td>
<td>0.3911</td>
<td>0.2150</td>
<td>0.8384</td>
</tr>
<tr>
<td>axc 100</td>
<td>12.2599</td>
<td>0.0147</td>
<td>0.9884</td>
</tr>
<tr>
<td>perl 5.004</td>
<td>18.0668</td>
<td>0.2712</td>
<td>0.8038</td>
</tr>
<tr>
<td>perl 5.005</td>
<td>27.3963</td>
<td>10.7037</td>
<td>0.9631</td>
</tr>
<tr>
<td>perl 5.17</td>
<td>22.4554</td>
<td>0.3767</td>
<td>0.7151</td>
</tr>
<tr>
<td>GFS Kernel</td>
<td>9.4272</td>
<td>0.0224</td>
<td>0.9823</td>
</tr>
<tr>
<td>php 5.3</td>
<td>13.006</td>
<td>0.0726</td>
<td>0.9428</td>
</tr>
<tr>
<td>php 5.4</td>
<td>5.0253</td>
<td>0.2089</td>
<td>0.8377</td>
</tr>
<tr>
<td>php 4.3</td>
<td>9.526</td>
<td>0.023</td>
<td>0.9823</td>
</tr>
<tr>
<td>php 4.4</td>
<td>3.0079</td>
<td>0.1158</td>
<td>0.9134</td>
</tr>
</tbody>
</table>

Additionally t-stat values and confidence values with \(n-1\) degrees of freedom are computed for comparison. Since the absolute value of t-stat is less than the confidence value, there is a level of confidence greater than 95\% that the null hypothesis cannot be rejected, and that Lehman “laws” 3 and 4 claiming productivity-decline invariance are not supported by the data.
4.4 Domain Variation: Hypothesis 3

The domain variation hypothesis states, for different domains (IDPD types), the average decline in productivity over the course of a project varies significantly. It was tested using the one-way ANOVA model to test the IDPD percentage difference among the application, infrastructure and platform domains. In this research, the IDPD percentage difference among the application, infrastructure and platform domains were examined.

As shown in Figure 13, the average of overall IDPD decline is 18% for Infrastructure domain, 9% for application domain, and 5% for platform domain. The distribution of IDPD across software domains is shown in Figure 14 and Figure 15.
One-way ANOVA was used to test for average IDPD differences among three domains. Table 11 shows the output of the one-way ANOVA analysis and whether we have a statistically
significant difference among our group means. Average IDPD for different domains differed significantly across the three sizes, $F(2, 20) = 8.7453, p = .002 (p < 0.05)$. The significance level is 0.002, which is below 0.05. Therefore, there is a statistically significant difference in the mean average IDPD among different domains. Taken together, these results suggest that overall IDPD differs across the three different domains. At this level, though, the results extend Lehman “law” in identifying domains with differing IDPD percentages.

4.5 COCOMO II and IDPD

4.5.1 Cost Estimation

Organizations planning sizable incrementally developed projects will be interested in estimating the cost of those increments. This applies to projects that have their number of increments or requirements determined and set in advance (like custom-built applications) as well as to open-ended projects (e.g. operating systems).

The types of possible cost estimation models that have been identified and could be applied to non-incremental development can also be applied to incremental development include Algorithmic (also known as parametric), Expert Judgment, Analogy, Parkinson, Price-to-Win, Top-Down and Bottom-Up.

Out of these, a parametric model has the advantage that it is predictive, does not depend on the availability of resources outside the project (save for the ability to set the parameters reasonably) and that its accuracy and quality can be objectively measured on the basis of existing project data. Though not all parameters of all models can be mapped or converted to each other, the large majority of them can be (Madachy & Boehm, 2008). For this reason, it becomes feasible to evaluate data collected using any model with any other model.
Table 12. COCOMO II Cost Drivers and Scale Factors expected to have a decrease in effort

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Comments/Special cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>PREC</td>
<td>Project precedence</td>
<td>Project becomes more familiar</td>
</tr>
<tr>
<td>TEAM</td>
<td>Team interactions</td>
<td>If continuity is good, customer-developer teams grow closer</td>
</tr>
<tr>
<td>PMAT</td>
<td>Process maturity</td>
<td>CMMI level increases should reduce rework</td>
</tr>
<tr>
<td>RUSE</td>
<td>Instances of reuse</td>
<td>Reused components may need further capabilities. Later increments will tend to see less reuse with the exception of systems that are built top to bottom</td>
</tr>
<tr>
<td>DOCU</td>
<td>Amount of documentation</td>
<td>Foundational increments need the most and best documentation.</td>
</tr>
<tr>
<td>PVOL</td>
<td>Development platform volatility</td>
<td>In the beginning, platform has not solidified yet</td>
</tr>
<tr>
<td>ACAP</td>
<td>Analyst capability</td>
<td>Analysts will gain capabilities if personnel continuity good</td>
</tr>
<tr>
<td>PCAP</td>
<td>Programmer capability</td>
<td>Programmers will gain capabilities if personnel continuity good</td>
</tr>
<tr>
<td>TOOL</td>
<td>Quality of development tools</td>
<td>Better tools will be acquired over time</td>
</tr>
<tr>
<td>SITE</td>
<td>Collocation and communication</td>
<td>If continuity good, teams grow closer and communications improve</td>
</tr>
</tbody>
</table>

While it would therefore be scientifically viable to base an IDPD model on parameters from each of these models or a synthesis of all of them, the scope of this document dictated that only one be chosen. The choice fell on COCOMO II not because it is of higher quality than the others, but because data collected using its parameters was most readily available and the fact that if an IDPD model could be based on COCOMO II parameters, it should be possible to map it to the other models.
It may be helpful to consider how cost drivers and scale factors behave over the course of an incremental project. Collected industry data suggests cost drivers to be variable across increments. In contrast to this, the incremental development model for COCOMO II cost estimation model assumes that the cost drivers remain constant across the increments of the whole project (Boehm & Lane, 2010).

Constant cost drivers are unable to explain the changes in productivity over the course of an incrementally developed project. Therefore, this model in its current form is not equipped for the estimation of the cost of incremental development.

### 4.6 IDPD and COCOMO II

Adapting COCOMO II to incremental development in order to predict IDPD also presents an opportunity to re-evaluate some parameters for the current time. While some parameters have held up over time, some may not have a major influence anymore nowadays (TIME and STOR come to mind, defined in Table 13).

#### 4.6.1 COCOMO II cost drivers and scale factors

Table 13 and Table 14 list expectations of whether the effort factor related to a given COCOMO II cost driver is expected to increase, decrease or have no expectation of any trend. The reason why the tables look at the expectation for effort multipliers instead of cost driver or scale factor levels is in order to make the tables more readable for readers unfamiliar with the individual drivers or factors: While for some of those a higher level can mean more effort, for some others it means less effort. Note that an expectation of no specific effort does not mean that there should not be any trend, just that it can go either way depending on the situation of the project.
### Table 13. COCOMO II Cost Drivers and Scale Factors expected to have no specific trend in effort

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Comments/Special cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLEX</td>
<td>Procedural flexibility</td>
<td></td>
</tr>
<tr>
<td>RESL</td>
<td>Architectural risk resolution</td>
<td>Architecture may require increasing amounts of rework; but familiarity with architecture grows</td>
</tr>
<tr>
<td>RELY</td>
<td>Required reliability</td>
<td>May increase for safety/security critical systems</td>
</tr>
<tr>
<td>DATA</td>
<td>Test data per line of code</td>
<td>May increase for big-data analytics</td>
</tr>
<tr>
<td>CPLX</td>
<td>Complexity</td>
<td>May increase with need to become part of system of systems or to support increasing demands</td>
</tr>
<tr>
<td>TIME</td>
<td>Percentage of CPU time used</td>
<td>Only relevant to projects with finite CPU resources or that share their execution time with others, such as in a mainframe environment; will increase then, but such projects are exceptions nowadays</td>
</tr>
<tr>
<td>STOR</td>
<td>Percentage of storage space used</td>
<td>Only relevant to projects with hard upper limits on storage or that share their storage with others; will increase then, but such projects are exceptions nowadays</td>
</tr>
<tr>
<td>PCON</td>
<td>Personnel continuity</td>
<td>High turnover reduces productivity because new team members need to be trained and affects experience related factors. Therefore, this is a pivotal cost driver for IDPD which depends on how the project is managed.</td>
</tr>
<tr>
<td>APEX</td>
<td>Applications experience</td>
<td>Applications experience will increase if personnel continuity good</td>
</tr>
<tr>
<td>PLEX</td>
<td>Platform experience</td>
<td>Platform experience will increase if personnel continuity good</td>
</tr>
<tr>
<td>LTEX</td>
<td>Language and tools experience</td>
<td>Language and tools experience will increase if personnel continuity good</td>
</tr>
<tr>
<td>SCED</td>
<td>Percentage of needed time available</td>
<td>Schedule compression becomes more expensive as system size grows</td>
</tr>
</tbody>
</table>
That the expectation for all effort factors is to go in no specific direction or to decrease suggests that in a general case, a well-managed organization which is able to keep turnover under control should experience a gradual decrease in the effort that is expressed in the existing COCOMO II cost drivers when applied to individual increments. This is because the organization will grow more familiar with the project and the developers will become more experienced. This means that in the case of a well-managed organization and in non-exceptional situations, IDPD cannot be captured or predicted using existing COCOMO II drivers applied to increments.

Table 14. P-value of weighted multiple regressions between cost drivers and productivity

<table>
<thead>
<tr>
<th>Project Drivers</th>
<th>DR-1</th>
<th>DR-2</th>
<th>DR-3</th>
<th>DR-4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product Factors</td>
<td>0.961</td>
<td>0.945</td>
<td>0.986</td>
<td>0.001</td>
</tr>
<tr>
<td>Platform Factors</td>
<td>0</td>
<td>0.789</td>
<td>0.762</td>
<td>0.139</td>
</tr>
<tr>
<td>Personnel Factors</td>
<td>0.025</td>
<td>0.029</td>
<td>0.137</td>
<td>0.169</td>
</tr>
<tr>
<td>Project Factors</td>
<td>0.136</td>
<td>0.425</td>
<td>0</td>
<td>0.02</td>
</tr>
<tr>
<td>Scale Factors</td>
<td>0.249</td>
<td>0.42</td>
<td>0.651</td>
<td>0.014</td>
</tr>
</tbody>
</table>

While Personnel Continuity (PCON) is a parameter that cannot be directly set by an organization, it is very influential on other parameters. This means that an organization that wants to be productive in iterative and incremental development should make efforts to retain its workforce, at least as far as the team of a given project is concerned.

In order to see which cost factors may have a higher influence on productivity and to increase the accuracy of predicting the IDPD factors in the next build, the relationship between individual cost drivers and productivity was evaluated by examining the p-value of regression between cost drivers and productivity.
Since there are 22 cost drivers and scale factors in total, only the ones with p-values within the significant range were reported (< 0.08) (see Table 15). In these 4 directed research (DR) projects, TIME is the factor that predicts the productivity the best. Then we have STOR, PCAP and SITE (Collocation), followed by ACAP, RELY, PREC, TEAM and PMAT. A weighted regression was conducted on all the cost drivers versus productivity (see Table 16).

**Table 15. P-value (<0.08) of regression between cost drivers and productivity for 4 DR (directed research) projects**

<table>
<thead>
<tr>
<th>Project Driver</th>
<th>DR-1</th>
<th>DR-2</th>
<th>DR-3</th>
<th>DR-4</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME</td>
<td>0.0022</td>
<td>0.08</td>
<td>0.08</td>
<td>/</td>
</tr>
<tr>
<td>STOR</td>
<td>0</td>
<td>0.04</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>ACAP</td>
<td>/</td>
<td>0.06</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>PCAP</td>
<td>/</td>
<td>0.06</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>RELY</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>0.07</td>
</tr>
<tr>
<td>SITE (Collocation)</td>
<td>0.08</td>
<td>0.07</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>PREC</td>
<td>/</td>
<td>0.01</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>TEAM</td>
<td>0.0031</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>PMAT</td>
<td>0.05</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
</tbody>
</table>

The results are still not consistent throughout the four DR projects. APEX seems to be the factor that predicts the productivity the best among all the factors. The ratings of APEX represent the level of applications experience of the project team developing the software system. Multiple regressions between groups of cost drivers and productivity, weighted by increment sizes, were also performed (Table 14). In order to find better predictions for productivity, the added lines of codes were used as the weight of the regression. The results show that personnel factors and project factors seem to be the best factors to predict productivity within the 4 projects. Weighted regression between cost drivers and productivity (only for cost drivers with correlation > 0.7)
Table 16. Weighted regression between cost drivers and productivity (only for cost drivers with correlation > 0.7)

<table>
<thead>
<tr>
<th>Driver</th>
<th>Project</th>
<th>DR1</th>
<th>DR2</th>
<th>DR3</th>
<th>DR4</th>
</tr>
</thead>
<tbody>
<tr>
<td>PREC</td>
<td>0.05</td>
<td>0.37</td>
<td>0.93</td>
<td>0.01</td>
<td></td>
</tr>
<tr>
<td>CPLX</td>
<td>0.34</td>
<td>0.55</td>
<td>0.94</td>
<td>0.45</td>
<td></td>
</tr>
<tr>
<td>RUSE</td>
<td>0.02</td>
<td>0.25</td>
<td>0.10</td>
<td>0.94</td>
<td></td>
</tr>
<tr>
<td>TIME</td>
<td>0.38</td>
<td>0.47</td>
<td>0.61</td>
<td>0.72</td>
<td></td>
</tr>
<tr>
<td>STOR</td>
<td>0.00</td>
<td>0.94</td>
<td>0.29</td>
<td>0.02</td>
<td></td>
</tr>
<tr>
<td>LTEX</td>
<td>0.00</td>
<td>0.62</td>
<td>0.76</td>
<td>0.70</td>
<td></td>
</tr>
<tr>
<td>SITE (Communication)</td>
<td>0.40</td>
<td>0.94</td>
<td>0.00</td>
<td>0.79</td>
<td></td>
</tr>
</tbody>
</table>

4.6.2 How IDPD influences parameters

In a project whose increment schedules have been planned beforehand, the size of IDPD will influence the SCED driver in subsequent increments in cases where one increment takes longer than expected. This can cause a chain reaction of schedule compressions if adjustments to the schedule aren’t made.

In situations where team members are judged by their productivity, a loss in productivity can cause an increase in personnel fluctuation (captured by PCON), which will weigh negatively on other cost drivers as pointed out further above.

4.6.3 Potential New Cost Estimation Parameters

Some of the main differences between monolithic and incremental projects are: 1) The existence of previous increments that need to be maintained, 2) The passing of time between releases, 3) That a large body of code (when compared to the current increment) needs to be integrated with the current increment, 4) that the previous increments may have a number of known
defects, 5) that those increments may be more or less complex, and 6) that later increments may depend on previous ones to a lower or higher degree.

The first of these is trivially true from the second increment on with regard to the existence of previous ones. It has been established that previous increments need to be maintained (Moazeni, Link, & Boehm, 2013). The remaining attributes of individual increments do not correspond with COCOMO II cost drivers. Since they are all measurable, albeit to different degrees, they hold interest as potential cost drivers.

**Time between increments**

The time passing between two increments causes the following phenomena (Moazeni, Link, & Boehm, 2013):

- Quality/usefulness of the software will slowly decrease all the way to zero because a system at rest will not keep pace with technological advances.
- Knowledge of the existing increments diminishes by personnel fluctuation or details being forgotten in the organization.
- New “must have” features need to be added to existing increments due to customer or market requests.
- Rework may need to be done due to changing paradigms (e.g. desktop to mobile).

Since all of these factors are exasperated the more time passes, more time between increments should cause the IDPD between them to increase. Increment productivity in terms of new SLOC over effort from several open-source projects has been found to be aligned with the time difference to the previous increment in a statistically significant way.
**Defect number and complexity in increments**

Defects of the previous increments will need to be fixed in those increments or compensated for in newer ones.

While defect complexity may not be easy to measure, it should be possible to at least categorize them into three categories:

1. Easy (e.g. typo or likely off-by-one)
2. Intermediate (requires fixing some complex interactions, but root cause is known)
3. Difficult (e.g. unknown root causes make system display occasional off-nominal behavior)

**Comparative size of current increment**

Most projects that data has been collected on have shown a decline in productivity between the first and last increment. The body of code will generally rise over the course of an incremental project. It is therefore of interest to relate the amount of existing code to the productivity of a new increment.

Intuition would say that the more code is pre-existing, the more work will have to be done on maintenance and integration. In addition to that, a growing body of code will either become increasingly complex, which causes the effort to master it to grow or work to become necessary to reduce the complexity (Moazeni, Link, & Boehm, 2013).

Increment productivity in terms of new SLOC over effort from several open-source projects has been found to be aligned with the size of the previous increment in a statistically significant way (Table 17).
Coherence of the project

An influence on the complexity of increments is to what extent these increments need to make calls to functions of other increments.

The measures here could lie in the number of calls to other increments, the complexity of these calls, the amount of required data flow over time between increments and the complexity of the increments that are being called (cyclomatic complexity and other measures).

Evaluated new Cost Drivers

Four potential parameters that would extend COCOMO II for increments have been evaluated. These were \textbf{RCPLX} (Complexity of Increment I+1 relative to complexity of existing software), \textbf{IPCON} (Staff turnover from previous increment’s staff), \textbf{PRELY} (Previous increments’ required reliability) and \textbf{IRESL} (Architecture and risk resolution (RESL) rating of increment I+1 relative to RESL of 1..I).

Initial data collected by the software industry did not show any good correlation with IDPD.

<table>
<thead>
<tr>
<th>Project</th>
<th>Correlation</th>
<th>P-value</th>
<th>Correlation</th>
<th>P-value</th>
<th>Correlation</th>
<th>P-value</th>
<th>Correlation</th>
<th>P-value</th>
<th>Correlation</th>
<th>P-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>common-daemon</td>
<td>-0.67</td>
<td>0.0086</td>
<td>0.17</td>
<td>0.2987</td>
<td>0.08</td>
<td>0.4024</td>
<td>-0.16</td>
<td>0.3008</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tomcat 6.0</td>
<td>-0.465</td>
<td>0.0245</td>
<td>0.4663</td>
<td>0.0245</td>
<td>-0.2806</td>
<td>0.1302</td>
<td>-0.4195</td>
<td>0.0326</td>
<td></td>
<td></td>
</tr>
<tr>
<td>php-5.4</td>
<td>0.1028</td>
<td>0.3726</td>
<td>0.0384</td>
<td>0.4484</td>
<td>0.107</td>
<td>0.3603</td>
<td>-0.0068</td>
<td>0.4859</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sendmail 8.14</td>
<td>0.5548</td>
<td>0.1684</td>
<td>0.5614</td>
<td>0.1631</td>
<td>-0.1356</td>
<td>0.4112</td>
<td>0.5181</td>
<td>0.1158</td>
<td></td>
<td></td>
</tr>
<tr>
<td>php-5.3</td>
<td>-0.6464</td>
<td>0.0003</td>
<td>0.698</td>
<td>0.0001</td>
<td>-0.1264</td>
<td>0.2724</td>
<td>-0.7045</td>
<td>0.0001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gsf-kernel</td>
<td>-0.5906</td>
<td>0.0015</td>
<td>0.5905</td>
<td>0.0015</td>
<td>0.1345</td>
<td>0.2772</td>
<td>-0.6184</td>
<td>0.0005</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmirror 0.1</td>
<td>-0.3215</td>
<td>0.1225</td>
<td>0.2063</td>
<td>0.2263</td>
<td>0.3132</td>
<td>0.1304</td>
<td>-0.2302</td>
<td>0.1873</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmirror 2.6.9</td>
<td>-0.3935</td>
<td>0.4111</td>
<td>0.0073</td>
<td>0.4776</td>
<td>0.2264</td>
<td>0.0953</td>
<td>-0.0405</td>
<td>0.4084</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cluster 3.1</td>
<td>0.3863</td>
<td>0.2223</td>
<td>-0.7339</td>
<td>0.0498</td>
<td>-0.8014</td>
<td>0.028</td>
<td>0.7578</td>
<td>0.0143</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Ten projects have been evaluated using measures of effort and size. The best ANOVA p-values have been found when relating new lines to productivity.

4.7 Threats to Validity

4.7.1 Internal validity considerations

First increment peculiarities

The productivity of the first increment of any software project can be atypical due to a number of reasons. These include that exploration may occur and that IDEs (Integrated Development Environments) being used may generate a significant amount of the code that is only edited or configured in later increments. Similarly, newly added source code may be based on templates the development team is reusing from other projects. Two further reasons for lower incremental productivity are that it is doing much of the requirements engineering and architecting for the later increments; and that it may end with a good deal of partially-developed code not counted in the incremental release (this may also be the case for later releases). While these are concerns, the productivity of all examined projects still decreases between the second and last increments, though to a smaller degree.

Time reporting

Some of our projects had university students in their development teams. The accuracy of time logs by the students is somewhat questionable. Depending on the amount of credits they are aiming for, students have to work a certain amount of hours per week. In cases where the students did not actually work as long as expected, there is a temptation to overstate the time so that it appears they are doing the work that is expected of them.
Another aspect is that some students may be late in filling out the time sheets, and, when reminded of them, fill them out with poorly-remembered hours. The threat to the internal validity is somewhat mitigated by the interest in embellishing their hours being common to all student projects and all parts of the projects, so that statements about whether productivity is increasing or decreasing remain valid. Similar concerns can apply to professional programmers occasionally.

The accuracy of the time logs as submitted by members of development teams can be questionable when team members work on several projects or several increments (testing increment 1, developing increment 2, designing increment 3) at the same time and need to attribute parts of their time to different projects or increments. The threat is mitigated for professional and student developers by the likelihood of distortions being common to all parts of the project, which will not significantly affect the study of the development of their productivity.

4.7.2 External validity considerations

Student projects have relevant threats to their external validity due to the motivation of the students. The motivation of the students is different than that of people working in the software industry in that the students are typically facing less existential risks for failure. A professional programmer may get laid off for bad performance and face significant material losses. A student may face a bad grade in one course.

Students in project courses are generally working part-time, and may have productivity peaks and valleys due to exams or big homework assignments in other courses. This can also happen to professional developers working on multiple projects.
5 Incremental Development Cost Estimation

Several major cost estimation models are used throughout the software industry. Some are open to the public while others are closed and can only be used as a commercial product. If they would address incremental development, this would obviously have an impact on this research. What follows is Table 18 with the names of the models and a sourced statement on whether they have provisions for incremental development.

Table 18. Comparison of cost estimation models

<table>
<thead>
<tr>
<th>Model</th>
<th>Incremental development provisions</th>
</tr>
</thead>
<tbody>
<tr>
<td>COCOMO II (Incremental COCOMO)</td>
<td>Yes (Boehm, et al., 2000)</td>
</tr>
<tr>
<td>SEER for Software</td>
<td>No</td>
</tr>
<tr>
<td>PRICE H</td>
<td>unknown</td>
</tr>
<tr>
<td>SLIM</td>
<td>No</td>
</tr>
<tr>
<td>SAGE</td>
<td>unknown</td>
</tr>
</tbody>
</table>

Conventional cost estimation methods do not fit incremental development well because they have no provisions for incremental development. COCOMO II model has an extension called Incremental COCOMO that has provisions for Incremental development. In the following section the Incremental COCOMO II model is described followed by the estimation steps that are part of the Incremental COCOMO model. This research invalidates the Incremental COCOMO extension in its current form, therefore, the Constructive Incremental Cost Model (COINCOMO) and its tool will be introduced.
5.1 Incremental COCOMO II

Incremental COCOMO was defined as an alternative to the traditional Waterfall model of the software development process. This model deals with cost estimation of software projects with a series of concurrent increments yielding an intermediate product. Increments have the following MBASE/RUP phases:

- Product Design (PD)
- Detailed Design (DD)
- Code and Unit Test (CUT)
- Integration and Test (IT)

At a high level, the estimation for incremental development involves:

- Use COCOMO II model with scale factor and cost driver ratings
- The percentage of change in software size from all previous increments (Adaptation Adjustment Modifier, AAM)
- The starting milestone for each increment after the first
- Any delay time for starting an increment
- The distribution of effort and schedule for each phase

5.1.1 Incremental COCOMO Estimation Steps

1. Compute the size for each increment and the total size

   Adjust the size of the succeeding increments after the first increment due to reusing the software product from all the previous increments.
\[
\text{Adj. Size}_i = \text{Size}_i + \left( AAM \times \sum_{j=1}^{i-1} \text{Size}_j \right)
\]

AAM is applied to the size of all previous increments. AAM is based on the percent design modified (DM), percent code modified (CM), the integration and test modified (IM), the assessment and assimilation (AA), software understanding (SU) and programmer unfamiliarity (UNFM).

\[
\text{AAF} = (0.4 \times DM) + (0.3 \times CM) + (0.3 \times IM)
\]

\[
AAM = \frac{[AA + AAF(1 + ((0.02 \times SU \times UNFM)))]}{100}, \text{ for } AAF \leq 50
\]

\[
AAM = \frac{[AA + AAF + (SU \times UNFM)]}{100}, \text{ for } AAF \geq 50
\]

2. Determine the total effort and schedule for the total single product

Calculate total effort and schedule using the COCOMO II model and distribute the estimates for MBASE/RUP phases.

3. Determine increment 1’s estimated effort and schedule

Use the COCOMO II effort calculation formula and calculate the effort using COCOMO II parameters, size and phase distribution.

4. Determine the added integration effort

The added integration effort is calculated by subtracting the full product effort and the sum of estimated effort for all of the independently estimated increments.

5. Distribute the added integration effort across remaining increments

6. Determine all remaining increments’ effort and schedule
Use COCOMO II model to estimate the total effort for the remaining increments. Use the increments’ adjusted size.

7. Distribute each increments’ effort and schedule over the phases

5.1.2 Incremental Development Estimation Considerations

The COCOMO II cost estimation model, uses cost drivers that remain constant over the course of the whole project. Because of their staying the same, they are unable to explain the changes in productivity over the course of an incrementally developed project. Therefore this model in its current form is unsuitable for the estimation of the cost of incremental development.

5.2 Constructive Incremental Cost Model

Build-to-Build behavior hypothesis states that the rate of productivity decline from increment to increment is not constant which means that productivity may decline, rise or stay the same. In a multi-build incremental development model, this means that COCOMO cost drivers (Effort Multipliers and Scale Factors) should be variable across increments. This contradicts with the incremental development model in Appendix B of the COCOMO II book (Boehm, et al., 2000) which assumes that the cost drivers remain constant across the increments. To overcome this, a new tool called COINCOMO (COstructive INcremental COst MOdel) is developed to enable the cost drivers to be variable across increments.

5.2.1 Software Engineering Reasons for Multi-Build, Overlapping RUP Cycles

The selection of multiple builds with overlapping MBASE/RUP cycles was done for sensible systems/software engineering reasons. The primary one for multiple builds was based on productivity in the face of risks, in particular for unprecedented System of Systems. An all or
nothing approach to the software development (also known as a "big-bang" approach) does not make sense. With multiple builds, the program has a chance to re-scope either pieces of software or whole systems at the decision points based on Feasibility Evidence. It also provides the ability to generate and demonstrate (in an earth-based test environment) some of the system capabilities earlier than is possible in a big-bang approach, and those capabilities should involve many of the "risky" bits, like communications with the constellation of spacecraft modules and network management of the constellation.

Employing overlapping, concurrent builds for the different systems was also selected for sensible systems/software engineering reasons. Assuming there is not a software development resource restriction, and that developers have appropriate "application" domain experience, then getting the teams to work concurrently is the smartest thing to do from an overall software development effort perspective: the efforts of the teams are additive and not based on the total Source Lines-of-Code (SLOC) count being used on any given system. From a systems and software engineering perspective, producing builds that are fully systems-tested, even if parts of the build may use simulation and/or emulation, means that both the software-to-system-integration teams and systems-test teams can start to work earlier than in a big-bang approach. The systems engineers and software architects can practice the concurrent "Agile Architecting and Stabilized Development" approach, for systems with high concerns for safety or security or requirements that are unstable or emerging. The early integration and test experience will also increase the acquirer’s confidence about the capabilities and architecture of the system.
5.3 Incremental Development Cost Estimation using COINCOMO

The original COCOMO I model used the waterfall model for its anchor points: the cost and schedule estimates covered the effort between Software Requirements Review (SRR) and Initial [software] Operational Capability (IOC). That model, which dates back to the 1970s (Royce, 1970) has been recognized as an appropriate subset of the possible process models under the appropriate conditions) (Barry & Port, 1999). Current terminology for anchor points, as used in COCOMO II, requires a more general approach (Bohem B., 1996).

Figure 16. Overlap across Builds
The co-evolution of the RUP in the late 1990s led to the obvious alignment of the anchor points (and their assessments) as gatekeepers to the RUP “phases”. The gatekeepers are Inception Readiness Review (IRR) for Inception phase; Life Cycle Objectives (LCO) for Elaboration; Life Cycle Architecture (LCA) for Construction; and IOC, with a Transition Readiness Review (TRR), for Transition; and a Product Release Review (PRR) to formalize the end of the ”Transition”. These anchor points, super-imposed on the RUP activities diagram, are shown in Figure 17. Also shown in Figure 17 via shading is the area from LCO to IOC, i.e., the Elaboration and Construction phases, and the covered activities that are included in the total effort that a COCOMO II estimation covers. Builds made using the COCOMO II model follow either the “once-through” or “incremental” development strategies, as described in Appendix B of the IEEE’s J-STD-016-1995 (IEEE Standard Association, 1995). In both cases, there is only one system integration and software qualification

Figure 17. RUP with MBASE Anchor Point
test, which are two of the primary activities taking place during the Transition phase. These two development strategies are in contrast to the “evolutionary” strategy that develops a system in builds, where the first build incorporates part of the planned capabilities, the next build adds more capabilities, and so on, until the system is complete. If the builds follow the “evolutionary” strategy, including their integration with the system to be deployed, they are often fully tested and delivered to the acquirer, in which case all the phases of RUP are repeated for each build. However, how the phases overlap is not depicted in RUP; Figure 16 shows the possible overlaps across the builds.

![Figure 18. Overlap across builds with carry forward](image-url)
5.3.1 COINCOMO Cost Estimation Steps

Bellow are different steps in estimation:

1. Compute schedule and effort for each Increment

Use the COCOMO model to calculate schedule and effort for each build. COCOMO calculates effort and schedule for the Elaboration and Construction phases of a build with new code and code carried forward (Figure 18) from the previous build treated as re-used code with re-use parameters.

Recall from COCOMO reuse model that reuse is influenced by Adaptation Adjustment Modifier (AAM) which is based on % Design Modified (DM), % Code Modified (CM), % Integration and test Modified (IM), assessment and assimilation (AA), Software Understanding (SU) and programmer unfamiliarity with software (UNFM).

\[
AAF = (0.4 \times DM) + (0.3 \times CM) + (0.3 \times IM)
\]

\[
AAM = \frac{[AA + AAF(1 + ((0.02 \times SU \times UNFM)))]}{100}, \text{ for } AAF \leq 50
\]

\[
AAM = \frac{[AA + AAF + (SU \times UNFM)]}{100}, \text{ for } AAF \geq 50
\]

To calculate the size for the next increment, the sum of previous build’s increments are treated as reused and adjusted with AAM. This is then carried forward from the previous build and added with the increment to be developed.

The amounts and re-use factors of the carried code change with the number of times the code is carried: these factors influence the "Equivalent SLOC", or ESLOC, for the carried code which in-turn impacts the effort and schedule.

2. Distribute the effort and schedule across phases
Use the Constructive Phased Schedule and Effort Model (COPSEMO) to distribute effort and schedule across the inception, elaboration, construction and transition phases. The COPSEMO model, separates the man- power loading between Elaboration and Construction, and adds additional effort and schedule for the Inception and Transition phases.

3. Determine resulting cumulative effort and schedule for each increment

The total effort is additive, meaning it is the sum of effort across all the phases of the increments. When calculating the total schedule in this approach, only the parts up to an overlap are counted.
6 The COINCOMO Tool

COINCOMO (CONstructive INcremental COst MOdel) is a screen-oriented, interactive software package that assists in budgetary planning and schedule estimation of a software development project. Through the flexibility of COINCOMO, a software project manager (or team leader) can develop a model (or multiple models) of projects in order to identify potential problems in resources, personnel, budgets, and schedules both before and while the potential software package is being developed.

The current version of COINCOMO, version 2.0, is a multi-platform software cost estimation application, which in one tool supports both the cost estimation models of COCOMO and COPSEMO (Constructive Phase and Schedule Model). Future iterations will be extended to include other cost estimation models, and will be extensible to support such models.

The tool will allow users to do cost estimation of systems with multiple subsystems, where each subsystem could have multiple builds. The tool will also support estimation of individual and multiple builds of a single subsystem; where estimates across builds take into account concurrent development of overlapping MBASE/RUP phases (Boehm, et al., 2000):

- Inception
- Elaboration
- Construction
- Transition

The full COCOMO II model includes three stages.

1. Supports estimation of prototyping or applications composition efforts.
2. Supports estimation in the Early Design stage of a project, when less is known about the project’s cost drivers.

3. Supports estimation in the Post-Architecture stage of a project. COINCOMO 2.0 implements stage 3 formulas to estimate the effort, schedule, and cost required to develop a software product. It also provides the breakdown of effort and schedule into software life-cycle phases and activities from both the Waterfall model and the MBASE/RUP Model (Boehm, et al., 2000).

The COINCOMO 2.0 implements stage 3 formulas to estimate the effort, schedule, and cost required to develop a software product. It also provides the breakdown of effort and schedule into software life-cycle phases and activities from both the Waterfall model and the MBASE/RUP Model. The MBASE/RUP model is fully described in *Software Cost Estimation with COCOMO II*.

6.1 COINCOMO Terminology

Before launching into the discussion of COINCOMO, one must understand the terminology used by COINCOMO. A *System* is the basic aggregate unit used by COINCOMO to represent a real-life software development project. A single part of the system is represented by a Sub System in COINCOMO. Thus, a single and independent functional part of a real-life system may be represented by a sub-system. Sub-systems are further broken down into Components. For example, different builds of the same part of a project can be represented in COINCOMO as components. Components are further broken down into Subcomponents. These are the basic atomic units used by COINCOMO to describe a software development activity. In a real-life scenario, for one build, new code development and maintenance of previously developed code for a given component could qualify as two separate sub-components.
6.2 The COINCOMO Architecture

In order to meet resource contention and portability capability goals and requirements, a client-server separation of the User Interface (UI), import/export, and format translation from the model calculations and database access logic was used. A Java based client UI and JDBC-compatible server-based RDBMS was used to ensure portability among the required platforms. The RDBMS-centric design inherently supports extensibility, as methods for extending 3NF schemas are well known. In addition, the design supports multiple open-source database vendors for flexibility. An XML-based formatting system allows translation of estimation data to and from multiple formats, as well as direct data transfer between databases.

The element relationship diagram in Figure 19 shows the primary entities involved with the COINCOMO 2.0 system. The user interacts with the system by providing inputs based on the use of multiple cost models. The cost models are database-driven and interact with the users input via the database. Outputs are put back to the database. Then, the outputs are retrieved and displayed back to the user.
Figure 19: COINCOMO Element Relationship diagram

Figure 20 accommodates for the new System, Subsystem, Component (on the left side) and multiple subcomponents (on the right side). The COCOMO estimation is shown at the bottom of the figure.

As shown in Figure 21, the COPSEMO calculation shows the effort and schedule distribution across the MBASE/RUP phases along with a graphical representation of phased schedule and effort. The spinners shown in Figure 21, allows adjusting the effort and schedule percentage for the MBASE/RUP phases.
Figure 20: COINCOMO Component estimation

Figure 21: COPSEMO calculation
The COINCOMO tool support cost estimation of multibuild software, Figure 16, and allow adjusting the overlapping across the builds.

Figure 22: COINCOMO support for multibuild estimation
7 Conclusion and Future Work

7.1 General Conclusions

The research presented in this dissertation, defined Incremental Development Productivity Decline (IDPD), formulated hypothesis (Decline hypothesis, Build-to-Build behavior hypothesis and Domain hypothesis) and based the industry, open source and controlled experiment data, confirmed or rejected the hypothesis. In summary, most of the analyzed data points are well aligned with negative trend lines. The average arithmetic decline is about 16.79 and the slopes vary between 5.49% and 15.09%, with an average of 10.07% and a median of 10.50%. Average IDPD across domains are 18% for infrastructure domain, 9% for application and 5% for platform software domain. Finally, the build-to build behavior hypothesis confirmed IDPD to be different across the builds in a software projects. This hypothesis disproved Lehman Laws 3 and 4 on statistical invariance.

After evaluating how different kinds of trend functions (linear, logarithmic or exponential) fit the observed cases of IDPD, it was concluded that there is not one single type of negative trend function that fits best in all cases, but that different cases had a better correlation with different types of trend functions. However, we found that on average the trend lines were best aligned with a logarithmic trend function. This is in line with an IDPD factor determining the ratio of productivity between subsequent increments, and that factor having a limited variance.

Challenges included that there was no way to retroactively validate and verify the collected data and that backstories of the projects were often either nonexistent outright or not deep.
Additionally, the collected metrics were not always the same and the code counting methods differed between different projects.

As part of this research, individual COCOMO II parameters were evaluated with regard to their development over the course of increments and how they influence IDPD. The reverse is also done. In light of data collected in the controlled experiment projects and the analysis that was presented in this dissertation, the following conclusions are made:

- Even when conventional COCOMO II drivers are applied to the increments of such projects and made variable over them, they are unable to explain the cases where a well-managed organization (meaning one which manages personnel education and turnover well) experiences IDPD. Therefore, either new parameters need to complement or replace the existing ones in the model or the prediction has to take place outside the model.

- A controlled experiment with students showed inconclusive results.

- Promising new parameters with statistically significant predictive capabilities in several open-source projects include the size of the current increment and the time between increments and the time between the current and the first increment as well as the time between current and the last increment.

- Data should be collected using parameters that include defect and complexity analysis of code.

### 7.2 Summary of Contributions

The main contributions of this research are as follows:

- Confirmed nontrivial existence of IDPD (Hypothesis 1)

- Rejected (confirmed null hypothesis) build-to-build constancy of IDPD (Hypothesis 2).
  
  Lehman “Laws” 3 and 4 on statistical invariance not confirmed
• Confirmed IDPD variation by domain (Hypothesis 3)

• Developed COINCOMO cost estimation model supporting cost driver variation by increment

This dissertation described the COINCOMO cost estimation model and its tool for incremental development estimation. In a study that was done on existing cost estimation models, the majority of them turned out to be not supporting incremental development. The COCOMO cost estimation model that was selected as the model for this research supported an incremental model through an extension. However, as described in this dissertation, the model was invalidated based on the results of hypothesis 2.

In addition, the study of COCOMO II parameters and IDPD discussed how existing parameters can be applied to IDPD and proposed new cost parameters.

7.3 Future Work

A much desired outcome of this research would be to construct a mathematical model for incremental development cost prediction that, for a given project, takes into account:

• Its IDPD domain,
• The COCOMO II (or other major cost estimation model) cost driver ratings applicable for its individual increments,
• New parametric cost drivers specific to IDPD,
• Other factors to be determined.

Once such a cost estimation model for IDPD has been found and sufficiently verified, it may be used to extend major cost estimation models such as COCOMO II.

To achieve a mathematical model, future work should focus on employing the following methods:
• Collected data needs to be statistically evaluated further under more aspects in order to increase chances of finding patterns that predict productivity.

• Project data needs to be collected and evaluated regarding the parameters proposed in this paper.

• Additional potential parameters should be evaluated.
BIBLIOGRAPHY


COPSEMO, Constructive Phase and Schedule Model, Retrieved from: http://csse.usc.edu/csse/research/COPSEMO/
Appendix A: The COINCOMO User Interface

COINCOMO 2.0 is a standalone software system intended for a single user. The software is user-interactive in that it attempts to interface well with a user’s needs, using extensive mouse interaction wherever possible. In order to use COINCOMO efficiently, the user must become familiar with the user interface (UI) of COINCOMO.

Figure A-1 shows the initial screen presented by COINCOMO to a user once he/she has connected to the database.

![Initial screen after connecting to database](image)

**Figure A-1. Initial screen after connecting to database**

As depicted, the COINCOMO UI is divided into three major areas:
1) **Main Menu Bar** – This area contains the menu selection of the main functions of COINCOMO. These selections are File, Parameters, and Help. The menu selections File and Parameters are discussed in sections 4 and 5 of this manual respectively. The menu selection Help is used to display the version and copyright information about COINCOMO, and open up HTML version of this User Manual.

2) **Project Browser** – This area is used to view and access a project. COINCOMO 2.0 can display only one project (system) at a time. The Project Browser displays the entire project, along with all of its sub-systems and components, in a hierarchical or tree fashion. An example of such a display is shown in Figure A-2. The Project Browser also contains, at the top, a Project Toolbar with image buttons that allow the addition or removal of sub-systems and components to a project. Note that these context-sensitive actions can also be invoked directly in the Project Browser by right-clicking on a given unit and selecting the appropriate option from the resulting pop-up menu. These context-sensitive pop-up menus also allow the user to rename a unit. Two examples of this behavior are shown in A-3.

3) **Unit Detail Area** – This area shows detailed information about a particular unit (System, Sub-System, or Component) selected in the Project Browser. There are two tabs in this area:

   a. **Overview** – displays relevant information about the unit selected in the Project Browser; shows the Summary Report for a system (refer A-4), Phase-Effort-Schedule for a sub-system (refer Figure A-5) and the Component Level Estimating Form (CLEF) for a component (refer Figure A-6), respectively

   b. **COPSEMO** – displays COPSEMO calculations for a component (refer Figure A-7); this tab is disabled for systems and sub-systems.
We now examine the Component Level Estimating Form (CLEF) in more detail, since this is where most of the user interaction takes place. A typical CLEF display for a component is shown in Figure A-8.

The important parts of the CLEF, as marked in Figure A-8, are discussed below:
1) **Add Subcomponent** – This button is used to add a sub component to the currently selected component in the Project Browser.

2) **Delete Subcomponent** – This button is used to remove a sub component to the currently selected component in the Project Browser.

3) **Aggregate Subcomponent Selection** – This drop-down menu can be used to select all, deselect all or toggle the selection of the various sub components for the currently selected component in the Project Browser (refer Figure A-9).

![Figure A-9](image)

4) **Scale Factors** – This button displays the Scale Factor Dialog Box as shown in Figure A-10.

![Figure A-10](image)
Dialog boxes such as these are common throughout COINCOMO 2.0. To make changes to a given factor, the user needs to click on the corresponding Rating button. This displays a pop-up menu, displaying the values available for the factor (refer Figure A-11). To adjust further, one can use the Increment button, which also displays a pop-up menu with preset options of 0%, 25%, 50%, and 75% (refer Figure A-11). After making the desired adjustments, the user should click the Apply button to propagate the changes to the currently selected component or sub component. To reset all factors to their original values, one can click on the Reset button. To exit the dialog box, the user should click on Close.

5) **Schedule** – This button displays the Schedule Dialog Box as shown in Figure A-12.
6) **Subcomponent Selection Column** – This column is reserved for identifying and selecting a sub component. Selection is denoted by a tick mark (√) that appears in this column. Only one sub component can be selected at a time using this column.

7) **Subcomponent Name Column** – This column is used to house the name of each sub component located in the CLEF. Clicking on the name of a sub component displays an input dialog box as shown in Figure A-13. This dialog box can be used to rename a newly added or existing sub component. To propagate the change in the sub component, the user should enter the desired name and click on *OK*. By default, the addition of a new sub component creates a sub component with the name *(SubComponentX)*, where X is an auto-incremented number.

8) **Subcomponent Size Column** – This column is used to house the SLOC of each sub component located in the CLEF. Clicking in this column corresponding to a sub component spawns a dialog box as shown in Figure A-14. The value for SLOC can be computed in one of three ways. One way is to enter the value directly in the SLOC field as shown in Figure A-14. Another way is to use the function point model as shown in Figure A-15. Finally, the
Adaptation Adjustment Factor method can be used for the computation of the SLOC as shown in Figure A-16. The language of implementation of each sub component is initially unspecified, but may be set using this dialog box. Upon completion of SLOC sizing input, click on Apply and then Close.
Figure A-15
9) **Labor Rate Column** - This column specifies the amount of money (per month) which a developer working on a given sub component would be paid. The labor rate for a particular sub component can be edited by clicking on its corresponding Labor Rate column. This
spawns a dialog box as shown in Figure A-18. One can enter the new labor rate here, and click on OK to affect the change. The range on labor rate is between $0 and $99,999.

![Figure A-18](image)

10) **Subcomponent Effort Adjustment Factor (EAF) Column** – This column displays the cost drivers for a specific sub component. By clicking on this field, a dialog box appears as shown in Figure 3-19.

![Figure A-19](image)

As shown in Figure A-19, the cost drivers are divided into five groups: Product, Platform, Personnel, Project, and User. By default, the ratings of all cost drivers are “NOM”, and their
percentage increments (also referred to as inter-cost driver values) are set at 0%. The user can manipulate the value of each of these cost drivers individually by changing the corresponding rating and % increment values. The final rating of a cost driver is calculated using this formula for the interpolation.

\[
\text{Final rating} = (\text{Next cost driver rating} - \text{Current cost driver rating}) \times \text{Current inter-cost driver} / 100
\]

As individual cost driver ratings are changed, the total product of the cost drivers also changes. When all cost drivers have been modified, one can click on Apply and then Close to view the final EAF, which is displayed in the EAF column.

COINCOMO currently supports only the Post Architecture model of software development. The Post Architecture model applies once the software architecture has been formulated. This is in contrast to the Early Design model, which is supposed to be used at the earliest phase of a software project. In terms of the COCOMO paradigm used by the COCOMO II suite of estimation tools, the Early Design model differs from the Post Architecture model in its use of Effort Adjustment Factors. The Early Design model considers only seven pre-defined effort adjustment factors, whereas the Post Architecture Model makes use of seventeen pre-defined effort adjustment factors, sixteen of which are shown in Figure A-19.

**Estimation Results Area**

This area displays the effort, schedule, cost and staff estimates calculated by COINCOMO for a given component. These statistics are: the total SLOC count for the entire component (SLOC), total hours per month (PM/M), the total effort in person-months (PM), the total schedule for project completion in months (M), the total productivity (PROD), the total project cost (COST), the total project cost per instruction (INST), the total staff requirement to complete the project (Staff), and the risk associated with the project (Risk).
Status Bar

This area at the bottom of the CLEF displays various status and error messages to the user.

Subcomponent Implementation Language Column

This column indicates the development language for a given sub component. By default, no language is selected for a sub component. This can be changed while sizing the sub component, as shown in Figure A-16, where a drop-down box for implementation languages is present.

Subcomponent Nominal Development Effort (NOM Effort DEV) Column

This column holds the most likely effort estimate for a given sub component without incorporating the Effort Adjustment Factors (EAF).

Subcomponent Estimated Development Effort (EST Effort DEV) Column

This column holds the most likely effort estimate for a given sub component obtained by incorporating the Effort Adjustment Factor (EAF).

Subcomponent Productivity (PROD) Column

This column contains the calculated result of the sub component’s individual SLOC divided by the sub component’s most likely effort estimate.

Subcomponent Cost (COST) Column

This column contains the most likely estimate of the development cost for a particular sub component.

Subcomponent Instruction Cost (INST COST) Column

This column contains the most likely cost per instruction for a given sub component. This number is calculated from the Cost/SLOC for each sub component.

Subcomponent Staff Requirement (Staff) Column
This column houses the most likely estimate for the number of full-time software developers (FSWP) that would be needed to complete a given sub component in the estimated development time.

Appendix B: Working with COINCOMO

This section discusses the main menu options of COINCOMO 2.0 in more detail. COINCOMO 2.0 is available in three editions: Desktop Edition, Database Edition, and Unified Edition. Depending on the edition you obtained, some menu options might not be available/accessible to you.

COINCOMO 2.0 has two modes of operation: Desktop Mode and Database Mode. Desktop Edition only operates under Desktop Mode, and Database Edition only operates under Database Mode; only Unified Edition allows you to switch between the two modes.

File Menu

The expanded File menu from COINCOMO 2.0 is shown in Figure B-1 for Database Edition and Figure B-2 for Desktop Edition.
Connect (Database Mode Only)

This menu selection is used to connect and retrieve project data from the COINCOMO 2.0 database pertaining to a user. To connect to the database, one must click on Connect. A login screen then appears, as demonstrated in previous section, and also shown in Figure B-3. The user should enter his/her credentials to connect to the database. Once the user is connected to the database, the message “Connected to Database” appears on the Status Bar at the bottom of the screen. COINCOMO 2.0 requires a running database server to operate for Database Edition or in Database Mode for Unified Edition. Therefore, this selection will result in an error if the database server is
not running, or if there is a communication failure between the COINCOMO tool and the database system.

Figure B-3

**Disconnect (Database Mode Only)**

This menu selection is used to disconnect from the COINCOMO 2.0 database. To disconnect from the database, one must simply click on **Disconnect**. Once the user is disconnected from the database, the message “Disconnected from Database” appears on the Status Bar at the bottom of the screen. Before disconnecting from the database, the current state of the project will be saved in the database, and the next time the user retrieves the project, this state will be restored. Hence, adequate care must be taken before disconnecting from the database so as not to save the current project with improper data values.
New Project

This menu selection is used to create a new project. By default, the newly created project is named “(SystemX)”, where X is an auto-incremented number. It can be renamed using context menus in the Project Browser.

View Projects (Database Mode Only)

This menu selection is used to view the list of existing projects in the COINCOMO 2.0 database. Clicking on this selection brings up the dialog box shown in Figure B-4. From this dialog box, the user can choose the desired project, click on Load and then on Close to display the contents of that project in the Project Browser.

This same dialog box also contains the option to delete a project. Again, one simply needs to choose a project and click on Delete in order to delete a project. Note that deleting a project will permanently delete a project and all its contents from the COINCOMO 2.0 database. Hence, extreme caution is advised while using this option, unless the database system is configured (with regular backups) to overcome the problems caused by an unwanted deletion.
Save Project (Desktop Mode Only)

This menu selection will allow you to save the current project to a file in COINCOMO 2.0’s own file format for later editing, as shown in Figure B-5. Once you save the current project to a file, any subsequent Save menu selection will also save to the same file. If your current project is opened through a COINCOMO 2.0 file, this menu selection will also save to that file.
Save Project As

This menu selection allows you to save the current project in COINCOMO 2.0’s own file format in a different file name locally. For Database Mode, this menu selection allows you to actually save the current project to a local file so you can do offline editing in Desktop Mode later, and then use the local file to synchronize with COINCOMO 2.0 database.

Export As CSV/HTML/XML

Although COINCOMO 2.0 stores all the project-related estimates and calculated values in either its own file format or in COINCOMO 2.0 databases, in the real world, such estimates lend themselves to further processing and presentation if they are in commonly used file formats.
COINCOMO 2.0 has a data export feature, which allows it to generate such files. This feature is available through the Export menu selection is shown in Figure B-6.

As shown, data related to an entire project can be exported in CSV (Comma-Separated Values), HTML or XML format. It must be noted that this export feature operates on the current project. Once a particular export format is selected, a save dialog box pops up to allow the user to save the files in a desired location. An example of this behavior is shown in Figure B-7.
While the choice of HTML and XML formats creates only one file per project, an export in the CSV format creates two different files, named “(SystemX).csv” and “(SystemX)’s Multibuild Report.csv”, where (SystemX) is the file name you specified. While the first file deals with the complete estimates for the entire project, the second deals with estimates in relation to multiple builds within the project, if there are any.

Exit:

This menu selection can be used to exit from COINCOMO 2.0. COINCOMO 2.0 in Database Mode automatically save any changes to the database immediately; however, to ensure data integrity, it is recommended that users first disconnect from the database using the Disconnect option, and then exit COINCOMO using the Exit option. For COINCOMO 2.0 in Desktop Mode, this menu selection will detect if your current project has been saved or not, and prompt you for the option to save if necessary.
Parameters Menu

The expanded *Parameters* menu from COINCOMO 2.0 is shown in Figure B-8. This menu is available when an available component has been selected. Options within this menu allow control over the model used by COINCOMO 2.0 to generate the effort, schedule and cost estimates.

![Figure B-8](image)

Effort Adjustment Factors

This menu selection presents a dialog box that allows the user to manipulate the default multiplier values of the various cost drivers used by COINCOMO for a particular rating. Screens with sample values are shown in Figure B-9 through Figure B-13. To affect a change, one must simply change a multiplier value, and click on Apply. To set all ratings to their default multiplier values, the Reset button can be used.
### Figure B-9

![Table](effort_adjustment_factors.png)

<table>
<thead>
<tr>
<th></th>
<th>VLO</th>
<th>LO</th>
<th>NOM</th>
<th>HI</th>
<th>VHI</th>
<th>XHI</th>
</tr>
</thead>
<tbody>
<tr>
<td>RELY</td>
<td>0.82</td>
<td>0.92</td>
<td>1.0</td>
<td>1.1</td>
<td>1.26</td>
<td>0.0</td>
</tr>
<tr>
<td>DATA</td>
<td>0.0</td>
<td>0.9</td>
<td>1.0</td>
<td>1.14</td>
<td>1.28</td>
<td>0.0</td>
</tr>
<tr>
<td>DOCU</td>
<td>0.81</td>
<td>0.91</td>
<td>1.0</td>
<td>1.11</td>
<td>1.23</td>
<td>0.0</td>
</tr>
<tr>
<td>CPLX</td>
<td>0.73</td>
<td>0.87</td>
<td>1.0</td>
<td>1.17</td>
<td>1.34</td>
<td>1.74</td>
</tr>
<tr>
<td>RUSE</td>
<td>0.0</td>
<td>0.95</td>
<td>1.0</td>
<td>1.07</td>
<td>1.15</td>
<td>1.24</td>
</tr>
</tbody>
</table>

### Figure B-10

![Table](effort_adjustment_factors.png)

<table>
<thead>
<tr>
<th></th>
<th>VLO</th>
<th>LO</th>
<th>NOM</th>
<th>HI</th>
<th>VHI</th>
<th>XHI</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME</td>
<td>0.0</td>
<td>0.0</td>
<td>1.0</td>
<td>1.11</td>
<td>1.29</td>
<td>1.63</td>
</tr>
<tr>
<td>STOR</td>
<td>0.0</td>
<td>0.0</td>
<td>1.0</td>
<td>1.05</td>
<td>1.17</td>
<td>1.46</td>
</tr>
<tr>
<td>PVOL</td>
<td>0.0</td>
<td>0.87</td>
<td>1.0</td>
<td>1.15</td>
<td>1.3</td>
<td>0.0</td>
</tr>
</tbody>
</table>
**Figure B- 11**

![Effort Adjustment Factors - MyFirstComponent](image1)

**Figure B- 12**

![Effort Adjustment Factors - MyFirstComponent](image2)
Scale Factors

This menu selection presents a dialog box (refer to Figure B-14) that allows the user to manipulate the default values of the various scale factors used by COINCOMO. To affect a change, one can change a value, and click on Apply. To set all ratings to their default multiplier values, the Reset button can be used.
Equation Editor

This menu selection allows the user to change the basic effort and schedule equations used by COINCOMO 2.0. The menu selection spawns the dialog box shown in Figure B-15, which allows modification of these equations. To make the changes effective, the user must click on Apply after making the necessary changes. Reset can be used to set all ratings to their default values.
Function Points

This menu selection spawns the dialog box shown in Figure B-16. It allows the user to modify the default values of the Function Point Factors, which are used to calculate the SLOC size if the Function Point method is used. Again, clicking on Apply will cause any changes to be effective, while clicking on Reset set all values to default.
Person Month

This menu selection spawns the dialog box shown in Figure B-17, and allows the user to enter a new value for the hours included in one person-month. The default value for Person Month is 152, which means that there are 152 hours in one person-month. To change this value, the user can enter the new value in the dialog box (e.g., 160), and click on OK. The valid range of values is between 120 and 184 only.
Mode Menu

For COINCOMO 2.0 Unified Edition, this expanded menu is available to allow the user to switch between Desktop Mode (Desktop Edition functionality) and Database Mode (Database Edition functionality). If you are running in Desktop Mode (by default) and want to switch to Database Mode or vice-versa, a message will pop up asking if you want to switch to the other mode if you have project(s) opened currently. Confirming the switch will result in closing of the projects(s).

Figure B-18