Incremental Development Productivity Decline

Ramin Moazeni
Computer Science Department
University of Southern California
941 W. 37th Pl, SAL 339
moazeni@usc.edu

Daniel Link
Computer Science Department
University of Southern California
941 W. 37th Pl, SAL 339
dlink@usc.edu

Barry Boehm
Computer Science Department
University of Southern California
941 W. 37th Pl, SAL 326
boehm@usc.edu

ABSTRACT

Incremental models are now being used by many organizations in order to reduce development risks while trying to deliver the product on time. It has become the most common method of software development with characteristics that influences the productivity of projects.

This paper introduces a phenomenon called Incremental Development Productivity Decline (IDPD) that is presumed to be present in all incremental software projects to some extent.

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 kinds of projects.

Based on their coherence and other common characteristics, incrementally developed projects are split into several major categories.

Following this, several major projects are used as case studies in order to find out whether IDPD can be proven to exist.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management – *productivity, life cycle, cost estimation*.

General Terms

Management, Measurement, Economics, Human Factors, Standardization.

Keywords

Incremental Development, Productivity, Software Development, Software Industry.

1. INTRODUCTION

Economics of scale do not apply to software increments as they do to increased unit numbers of hardware. While hardware benefits from an increase in units, the "unit cost of later 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." [8]

PROMISE '13, October 9, 2013 Baltimore, USA

Copyright 2013 ACM 978-1-4503-2016-0/13/10… $15.00..

The following example of a large defense software system is given in [8] (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, rework)
- 318% change in requirements across all builds

An IDPD factor (i.e. the reduction in productivity between two subsequent builds) of 19% over the course of four increments will result in a reduction of productivity to about 53% of the original one [8]. IDPD factors of 14% (with variations, in a smaller software system) and similar have been found for large commercial software applications, such as Microsoft's Word for Windows [14], Windows Vista and others [12][8].

2. INCREMENTAL DEVELOPMENT

2.1 The concept

In order to understand what incremental development is, it helps to first define the opposite, monolithic development. This is development that is comprised of only one step and one release at the end of the development effort.

It may initially seem that the simplest and best way to define incremental development is the opposite of monolithic development, such as "any development effort that is done in more than one step and has more than one released build".

However, this definition is not sufficient because it would capture cases that are not fundamentally different from monolithic development, such as a sequence of efforts that add up to form a collection of products that have no dependencies on each other.

We therefore define incremental development for our purposes as any development effort that is done in more than one step, has more than one released build and where each step builds on the previous ones in that it would not be able to stand alone without these pre-existing steps.

The focus of the research is on incremental development as a type of development that "reflects the characteristic of building upon and enhancing the previous versions of systems, rather than developing whole new systems each time" [3].

This excludes development efforts that may be labeled "incremental", but whose increments don't have any significant dependency on previous increments. Those types of projects may just as well be regarded and predicted as collections of individual smaller projects.

Also excluded are cases of black-box reuse, in which the code produced up to and including some increments becomes immutable. This is because in those cases one major characteristic of incremental development, the ability to rework all previous increments, is not fully present. Such partially incremental projects are outside of the scope of this research.

2.2 Productivity

The conventional productivity equation is

Equation 1. Conventional Productivity

```
Productivity = Output/Input
```

While this is simple to understand, different metrics have been used to measure the input and output in terms for software development.

Since we are looking at projects that may take years to finish in some cases, we felt it made more sense to look at the input in terms of time spent by the developers than to look at the cost, since the cost may fluctuate significantly over that time due to economic developments.

While there is a host of different measures for the output (SLOC, Equivalent Source Lines of Code aka ESLOC and function points to name a few), the available data from the observations limits which metrics can be used. In the case of the case studies presented here, this has been the new logical SLOC for each increment.

Our equation for the productivity in a given increment is therefore

Equation 2. Software Productivity

```
Productivity = New SLOC/time
```

(Since comparisons between case studies are made on a normalized scale, SLOC can be replaced by KSLOC, which stands for kiloSLOC, and time by any unit of time as needed.)

Each increment adds to the body of code existing for the project.

When each increment depends on the increments before it, the amount of code that increment has to be compatible to increases. Additionally, having to depend on previous increments, the developers will uncover and have to fix previously unknown errors in the existing code. They will also have to rework parts of it.

This results in a productivity decline in two ways:

- Planning the development of the new increment will take longer.
- The developers' time is spent in reworking and fixing the errors of the existing code.

Depending on the characteristics of a given project, the productivity may rise again after several increments with declining productivity. This may due to productivity gains from learning how to deal with the system (e.g. through figuring out the architecture), from later increments having less complex requirements or because later increments may not depend on all earlier ones.

There are laws that have been observed to govern the behavior of software projects that mechanize human or societal activities and which become part of the world they model [17][18].

This applies to some, but not all programs are observed here.

The consequences of applying the laws to incrementally developed projects are as follows:

Continuing Change: If an incrementally developed project takes more than a very short amount of time to develop, then changes in the environment dictate that by the time the later increments are developed, the previous ones have to be reworked.

Increasing Complexity: In an incrementally developed project, work has to be done to either maintain or reduce the complexity of the program. Otherwise the complexity will add to the workload. (In both cases, the complexity will cause a need to do work.)

Fundamental Law of Program Evolution: Incrementally developed projects are self-regulating and follow statistically determinable trends and invariances.

Conservation of Organizational Stability (Invariant Work Rate): Work on existing increments takes away productivity from the currently developed one.

Conservation of Familiarity (Perceived Complexity): Everyone working on an incrementally developed project must maintain mastery of its content and behavior. The bigger the project gets, the more work 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 work will have to go into this and less will be able to go into new development.

Continuing Growth: The functional content of existing increments will need to be increased in order to maintain user satisfaction.

Declining Quality: In order to keep up their quality levels, all increments will have to be maintained and adapted to operational environment changes, which means that less work can go into the currently developed one.

Feedback System: In order to improve the project over time, feedback has to be taken into account, which goes into the workload.

3. INCREMENTAL DEVELOPMENT PRODUCTIVITY DECLINE (IDPD)

Incremental Development Productivity Decline (IDPD) is a phenomenon in which there is an overall decline in productivity of the increments. 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.

The criterion is whether the productivity has a significant correlation with a negative trend function. This function can be linear, logarithmic or exponential.

In order to be considered an increment for the purposes of IDPD evaluation, 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. In such cases we consider it part of the previous increment.

3.1 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 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% [20]. Large-scale commercial software projects, such as Word for Windows, Windows Vista and others have had similar experiences [12][14].

The table below shows experience-based IDPD effort drivers and their ranges per increment (based on the table in [8]):

Table 1. IDPD Effort Drivers [8]

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.

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 effort increase by 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 effort increase per increment would be 23%.

Over a significant number of builds, the difference will be striking. The following figure shows how a seemingly small difference in the effort increase can stretch out a project to twice as many builds. [8]

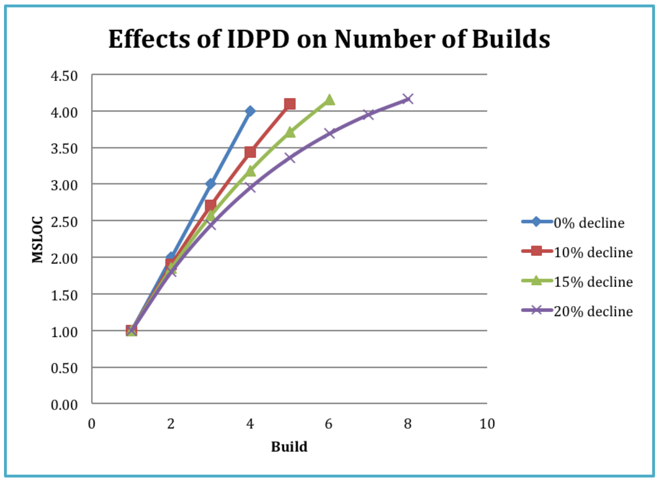

Figure 1. Effects of IDPD on Number of Builds

Thus, it is important to understand the IDPD factor and its influence when doing incremental or evolutionary development. Ongoing research 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 would be very helpful in improving the understanding of the IDPD factor.

3.2 Categories of Projects (IDPD Types)

Before we look at different categories of projects and their typical IDPD-related relevance, we need to preface this 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 author(s) of the software.

3.2.1 Internal, Non-Deployable

Non-deployable code is code that either cannot be deployed by 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 project, but not a part of its 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 subtypes 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 not developed in meaningful increments. (While it would be possible to increment "Hello World" in several steps by having the first increment only output "Hello", the first increment would already have the full functionality.)

3.2.2 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.

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 decide which installed version they are most compatible to (this is true for the Java Virtual Machine and DirectX).

3.2.3 *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 (Commercial Off-The-Shelf) 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:

3.2.3.1 *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.

3.2.3.2 *After the first release*

From this point on, increments will continue, and there are three ways in which they manifest themselves:

3.2.3.2.1 *Internal builds*

These are increments that are neither published nor deployed.

3.2.3.2.2 *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.

3.2.3.2.3 *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.

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

3.2.4 *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.

3.2.5 *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 enables 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

Type	Creates Documents	Compatibility needs	Has API	Scope	Has Versions	IDPD
Internal, Non-Deployable	-	-	-	-	-	-
Firmware	-	-	-	-	+	O
Platform Software	-	-	+	O	+	O
Application Software	+	+/O	O	+	+	+
Infrastructure Software	-	+/O	+	+	+	+

Internal, non-deployable software is software that is written ad hoc, incidental to the life cycle of other software, not released to customers and has no meaningful increments. An example is a simple one-shot shell script having few or no control structures in order to sequentially run or loop other software for demonstration purposes. Another example would be throwaway code to test a compiler, such as “Hello World”. Such software can easily be reproduced on the spot.

Due to generally being undocumented and proprietary to manufacturers of devices, firmware generally has no particular compatibility needs between versions beyond being able to perform the same functions. APIs are generally not supported. To the extent that different versions build on each other, IDPD may occur.

Platform software stands for hardware drivers running under operating systems. These may have APIs, and their scope tends to be bigger than that of firmware.

Infrastructure software is designed to not interact with the user directly, but to provide facilities and services for other software to run. Examples include operating systems, virtual machines and operating system extensions through libraries with APIs as well as common operating environments.

Infrastructure and Application Software are likely to have a major decline because of the expectations of users described above.

4. RESEARCH APPROACH

What we are trying to find out is:

How does the productivity behave over the course of a project? Are there patterns to it? What are the factors that influence it?

4.1 Decline Hypothesis

Decline hypothesis: In incrementally developed software projects that have coherence and dependency between their increments, productivity declines 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.

Null hypothesis: In incrementally developed software projects that have coherence and dependency between their increments, productivity does not follow any trend.

4.1.1 Data Collection for the case studies

As part of this research, we collected data on software engineering projects from private software industry, aerospace and defense contractors. This data would be used to validate IDPD. This includes performing behavioral analysis on the collected data, validation, and hypothesis test.

As with most software engineering research, the extent and depth of the data collection have their limits. For this research, this has mostly been the case when the data came from defense projects, where access to more detailed information often hits barriers [21]. Therefore, most of the data collected for this study have been solicited from private software industry sources.

As part of data collection effort for this study, we have interviewed programmers, held workshops and described our model and core data attributes of the projects such actual effort, actual size, and rating levels of the COCOMO II cost drivers. Effort is collected in person-hour and converted into person-month using the standard COCOMO model that defines 152 hours per person-month.

The size metrics are collected by using code counting tools such as USC’s Unified Codecount (UCC)[1] based on the logical SLOC definition, adopted from the Software Engineering Institute of CMU [22].

4.1.2 Data Collection Criteria

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.

Only projects that satisfy all of the following criteria are collected:

- Start and end dates are clear
- Has at least three 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
- Well-substantiated reuse factors for COTS and modified code
- Source code counted with tool that is aware of logical vs. physical SLOC

4.1.3 Data Collection Challenges

Data conditioning is a major challenge in the software data collection and analysis process. There can be many problems and misunderstandings that can make the data unreliable, biased and unusable. This data should not be used in the calibration process as it can introduce poor results.

Here is a summary of issues found during our data analysis:

1. Inadequate information on modified code (size provided)
2. Size measured inconsistently
3. Missing effort data
4. Missing schedule data
5. Inadequate information on size change or growth
6. Inadequate information on average staffing or peak staffing
7. Inadequate information on personnel experience
8. Inaccurate effort data in multi-build components
9. Replicated duration (start and end dates) across components
10. Inadequate information on schedule compression
11. No quality data
12. Low number of observations
13. History of data is unknown

4.2 Data Analysis

Several data points have been analyzed for their IDPD effect and we evaluate several ways of finding a model that fits our observed productivity. The data used for this study are mostly in the application and infrastructure domains.

[1] http://csse.usc.edu/ucc_wp/

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.

4.2.1 Exclusions based on logical reasons

While 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 would mean that the SLOC in the product would be diminished.

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 possible.

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.

4.2.2 Case studies

4.2.2.1 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. [20]

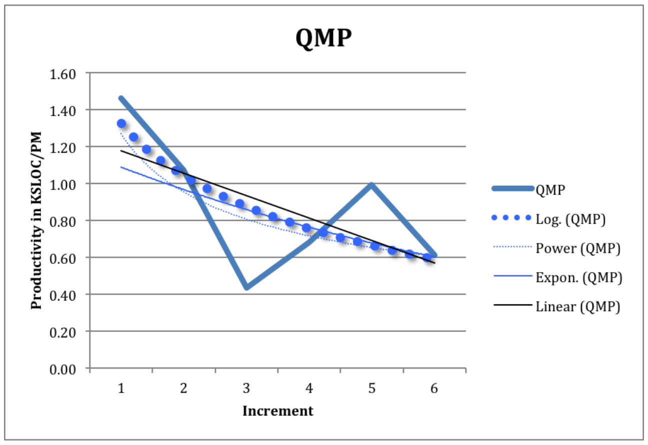

Figure 2. QMP Productivity across Increments

This is 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 [20].

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.)

4.2.2.2 Component management data mining projects

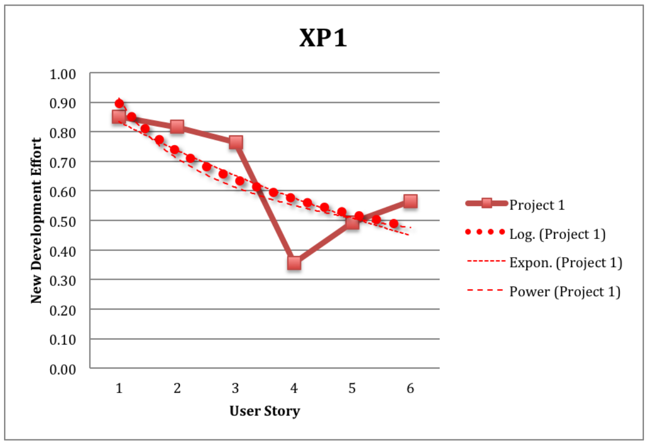

Figure 3. XP 1 Productivity across Increments

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 [19].

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.

"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 non-trivial 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."[10]

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. [1]

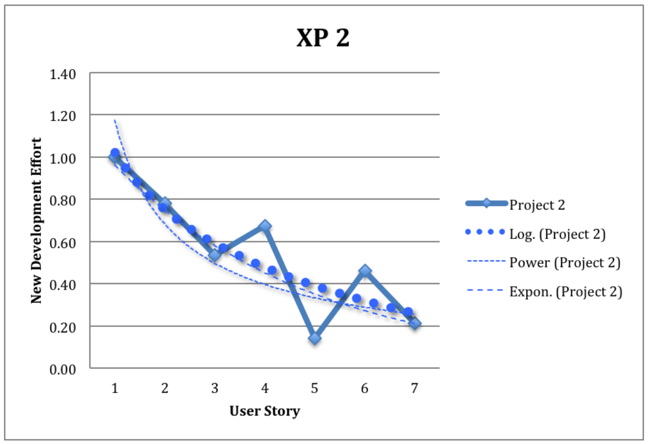

Figure 4. XP 2 Productivity across Increments

For project 2, the logarithmic trend line does best at 0.77, followed by the exponential one at 0.59 and the power line at 0.57.

4.2.3 Statistical Significance

The following figure shows the normalized productivity of all our case studies over their increments, with 1.00 being the productivity of the initial increment:

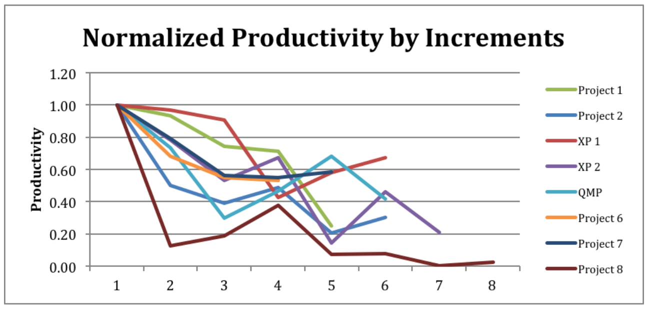

Figure 5. Normalized Productivity by Increments

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%.

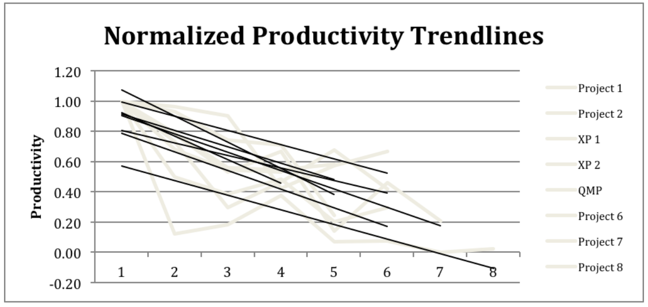

Figure 6. Normalized Productivity Trend Lines

This graph 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.

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.

Table 3. Regression and Correlation

Project	P-Value	Correlation Coefficient
Project 1	0.024	-0.93
Project 2	0.044	-0.82
XP 1	0.086	-0.75
XP 2	0.014	-0.86
QMP	0.057	-0.87
Project 6	0.201	-0.61
Project 7	0.085	-0.92
Project 8	0.151	-0.71

Interpretation: All projects save for Project 6 and Project 8 have good p-values and correlation.

Table 4. Regression Weighted by Size

Project	P-Value	R-Squared
Project 1	0.024	85.70%
Project 2	0.015	81%
XP 1	0.086	56.20%
XP 2	0.014	73.50%
QMP	0.067	72.40%
Project 6	0.141	45.60%
Project 7	0.085	83.70%
Project 8	0.019	69.70%

When the increments are weighted by size, the p-value of Project 8 improves considerably (from 0.151 to 0.019, effectively from unacceptable to outstanding) and that of Project 6 improves from 0.201 to 0.141, which is still not acceptable, but better than before.

5. THREATS TO VALIDITY

5.1 Internal validity considerations

5.1.1 *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.

While this is a concern, the productivity of all examined projects still decreases between the second and last increments, though to a smaller degree.

5.1.2 *Time reporting*

Two of the projects, XP1 and XP2 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 their expected, there is a temptation to overstate the time so 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 fictional 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 at the same time and need to attribute parts of their time to different projects.

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.

5.2 External validity considerations

Student projects XP1 and XP2 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.

6. OUTLOOK

The amount of data points that we have analyzed is limited. It would be desirable to obtain more data for analysis. Project data from projects that are at the same time substantial, fulfill the IDPD criteria, have sufficient background information and participants that can be interviewed a long time after the fact is a scarce resource.

Other researchers will be encouraged to related cases.

Overall we think that the way to go is to find a mathematical model for incremental development cost prediction that, for a given project, takes into account:

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

The model will also take into account that factors may be interrelated.
Once 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.

Where available, other measures of productivity than just the customer-oriented newly added SLOC per increment will be considered to broaden the base of our research. This will include modified and deleted SLOC and possibly more.

Work is already in progress to add open source infrastructure and application projects to the body of project data being evaluated. Additionally, the impact of Lehman's Laws of Software Evolution on IDPD is being researched as well.

7. CONCLUSIONS

This paper has described our analysis of the trends of productivity in incremental projects. The study was based on eight major industrial, incrementally developed software projects. All of them show an overall decline in productivity over their course. Most of them 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%.

The decline in productivity is due to factors such as previous-increment breakage and usage feedback, increased integration and testing effort.

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.

After evaluating how different kinds of trend functions (linear, logarithmic or exponential) fit the observed cases of IDPD, we found 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 they 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.

Statistical analysis of the 8 studied projects shows that 6 of them are statistically significantly aligned with the model. When the increments are weighted by size, we observed a significant improvement in p-value of one of the remaining projects (from 0.151 to 0.019) while the p-value of the other improved slightly from 0.201 to 0.141.

With the evidence obtained in this study, we plan on moving to the next step of analyzing IDPD for different domains and work toward implementing an IDPD model.

8. REFERENCES

[1] Alshayeb, M., & Li, W. An empirical study of relationships among extreme programming engineering activities. Information and Software Technology, 48 (11), 1068-1072.

[2] Aronson, E., Wilson, T. D., Akert, R. M., & Fehr, B. (2007). Social psychology. (4 ed.). Toronto, ON: Pearson Education.

[3] Avison, D. E., & Fitzgerald, G. (2003). Where Now for Development Methodologies? Communications of the ACM, 46 (1), 79-82.

[4] Boehm, B. (2012). RT 6–Software Intensive Systems Data Quality and Estimation Research in Support of Future Defense Cost Analysis. MONTH. Tavel, P. 2007. *Modeling and Simulation Design*. AK Peters Ltd., Natick, MA.

[5] Boehm, B. W. (1984, 1). Software Engineering Economics. Software Engineering, IEEE Transactions on , SE-10 (1), pp. 4-21.Forman, G. 2003. An extensive empirical study of feature selection metrics for text classification. *J. Mach. Learn. Res.* 3 (Mar. 2003), 1289-1305.

[6] Boehm, B., & Lane, J. A. (2010). DoD Systems Engineering and Management Implications for Evolutionary Acquisition of Major Defense Systems. DoD Systems Engineering Research Center.

[7] Boehm, B., Abts, C., Brown, A. W., Chulani, S., Clark, B. K., Horowitz, E., et al. (2000). COCOMO II: Estimating for Incremental Development. In B. Boehm, C. Abts, A. W. Brown, S. Chulani, B. K. Clark, E. Horowitz, et al., Software Cost Estimation with COCOMO II (pp. 328-336). Upper Saddle River: Prentice Hall.

[8] Boehm, B., Clark, B., Tan, T., Madachy, R., & Rosa, W. (n.d.). Future Software Sizing Metrics and Estimation Challenges.

[9] Boehm B., Software Engineering Economics. Englewood Cliffs, NJ, Prentice-Hall, 1981

[10] Boehm, B., Turner, R., Balancing agility and discipline: a guide for the perplexed, Addison-Wesley Professional, 2004

[11] Defense Cost and Resource Center. (2012, 6 5). Understanding the Software Resource Data Report (SRDR) Requirements. Retrieved 2 24, 2013, from Defense Cost and Resource Center: http://dcarc.cape.osd.mil/Files/Training/CSDR_Training/DCARC%20Training%20X.%20SRDR%20102012.pdf

[12] A. Elssamadisy, and G. Schalliol, "Recognizing and Responding to 'Bad Smells' in Extreme Programming", Proceedings, ICSE 2002, pp. 617-622.

[13] Galorath Incorporated. (2011). Product Brief SEER for Software: Cost, Schedule, Risk, Reliability. Retrieved 12 1, 2012, from galorath.com: http://www.galorath.com/DirectContent/SEERforSoftware2.pdf

[14] G. Gill, and M. Iansiti, "Microsoft Corporation: Office Business Unit," Harvard Business School Case Study 691-033, 1994.

[15] Lamba, S. K. (2012). Productivity Measurement During Incremental Development of Software System. International Journal of Engineering and Innovative Technology, 2 (1), 105-108.

[16] Lane, J. A., & Bohn, T. (2010). Using SysML to Evolve Systems of Systems. INCOSE.

[17] Lehman, M. M. (1980). Programs, Life Cycles, and Laws of Software Evolution. Proceedings of the IEEE. 68, pp. 1060-1076. IEEE.

[18] Lehman, M. M., Ramil, J. F., Wernick, P. D., Perry, D. E., & Turski, W. M. (1997). Metrics and Laws of Software Evolution - The Nineties View. IEEE.

[19] Li, W., & Alshayeb, M. (2001). An Empirical Study of Extreme Programming Process. Proceedings, 17th Intl. COCOMO/Software Cost Modeling Forum. USC-CSE.

[20] Tan, T., Li, Q., Boehm, B., Yang, Y., He, M., & Moazeni, R. (2009). Productivity Trends in Incremental and Iterative Software Development. 3rd International Symposium on Empirical Software Engineering and Measurement. Los Angeles, CA, USA.

[21] Valerdi, R., Davidz, H., "Empirical Research in Systems Engineering: Challenges and Opportunities of a New Frontier," Systems Engineering, 12(2), 169-181, 2009.

[22] Park R.E., Software Size Measurement: A Framework for Counting Source Statements, CMU/SEI-92-TR-11, Sept. 1992