Software Domains in Incremental Development Productivity Decline

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

Celia Chen
Computer Science Department
University of Southern California
941 W. 37th Pl, SAL 339
qianqiac@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 339
boehm@usc.edu

ABSTRACT
This research paper expands on a previously introduced phenomenon called Incremental Development Productivity Decline (IDPD) [1] that is presumed to be present in all incremental software projects to some extent.

Incremental models are now being used by many organizations in order to reduce development risks. Incremental development has become the most common method of software development. Therefore its characteristics inevitably influence the productivity of projects. Based on their observed IDPD, incrementally developed projects are split into several major IDPD categories.

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 has been collected and analyzed, indicating the degree of IDPD associated with each category. Several hypotheses have undergone preliminary evaluations regarding the existence, stability and category-dependence of IDPD with encouraging results. Further data collection and hypothesis testing is underway.

Categories and Subject Descriptors
D.2.9 [Software Engineering]: Management – productivity, life cycle, cost estimation.

General Terms
Management, Measurement, Economics, Human Factors, Standardization.

Keywords
Software engineering; incremental development; productivity decline; statistics;

1. INTRODUCTION
IDPD (Incremental Development Productivity Decline) is the phenomenon of an overall productivity decline occurring over the course of several increments. The IDPD between two increments is the percentage by which the productivity in new SLOC/PM (where “PM” means person-month) is diminished between them. (Example: If the productivity of a given increment is 80% of a given previous one, this is called an IDPD of 20%.)

The overall IDPD of a project is the decrease in productivity between its first and last increments (whether or not the project is still ongoing).

IDPD research is concerned with the question of how much new SLOC output an organization can expect for a new increment.

There are currently three hypotheses being pursued:
1. That there is IDPD.
2. That within the same project, two IDPD levels may differ by more than 10%.
3. That IDPD differs between project domains.

(The second hypothesis is concerned with IDPD levels not being constant. Two IDPD levels are defined as substantially the same if they do not differ by more than 10%.)

This paper focuses on the third hypothesis.

1.1 Productivity
For the purposes of IDPD research, productivity is measured in new SLOC/PM. While SLOC are not an ideal measure of software productivity because they are not very useful in improving it [2], much useful work is not concerned with the creation or testing of code and different management perspectives should be taken into account when measuring productivity [3]. IDPD research uses them as a measure because they are subjective and difficult to measure. In addition, code churn, which includes debug code churn, and refactoring code churn, is more associated with defect density than productivity [4] [5].

An IBM study case used “function points” to measure productivity [6]. 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. But the problem was that he never discussed how these weights were determined or was not}

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. (Depending on the chosen publication rights for the paper, ACM might send a text that slightly differs to the first author.)

ICSSP'14, May 26–28, 2014, Nanjing, China
Copyright 2014 ACM 978-1-4503-2754-1/14/05... $15.00.
In TRW's software cost estimation program, Boehm found that “software cost drivers is effectively the inverse of productivity drivers” [7]. 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. Vosburg and associates [8] did one of the most substantial studies of large-scale software productivity. 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.

1.2 INCREMENTAL DEVELOPMENT

In order to come to a useful definition of incremental development, it helps to look at the definition of “increment”. By definition, an increment is an improvement evolving additions to, modifications of, and deletions of existing increments’ code [14]. 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 [23]. Additionally, whether or not a project will be ended at a certain point can be hard to know a priori.

The definition of incremental development for IDPD purposes is therefore any development effort that is:
- achieved in more than one step,
- with each new step adding new functionality, and
- where each new step builds on the previous ones such that it would not be able to stand alone without them.

2. STATUS OF IDPD RESEARCH

The concept has been introduced with some first observations [1]. It has been found to be supported by Lehman’s Laws of Software Evolution [9].

Research remains to be done on several areas of interest:
- Whether the overall or average (weighted) IDPD varies significantly between different project domains,
- Whether a model of IDPD can be found that allows to predict it using parametric or artificial intelligence methods, and
- Whether the choice of a specific life cycle model or other management policy influences can mitigate IDPD.

2.1 Theoretical foundations of IDPD

The decline in productivity seems to be due to factors such as previous-increment breakage and usage feedback, increased integration and testing effort. Other factors may be that rising complexity of the overall body of code and architectural drift and erosion make it more difficult to add code to the project.

While data shows IDPD to be observable, it does not explain the phenomenon. Motivations of the research have been based on the theoretical foundations laid out in this section.

2.1.1 Lehman’s Laws

Lehman’s Laws of Software Evolution effectively state that software evolution is a predictable process with invariances and that in order to preserve quality, responsible organizations will need to perform regular and organized maintenance on their existing software and mental maintenance on their training. Therefore, in the case of incremental development, existing older increments will need to have that maintenance performed on them, which will decrease the productivity of newer ones [9].

2.1.2 Architectural Decay

A common phenomenon in software systems is a discrepancy between the prescriptive and descriptive architecture. That is, their implemented architecture is different from the architecture as it was originally planned. This happens due to practicalities in programming about by constraints such as deadlines or shortcomings in hardware in software that need to be worked around.

This effect is not limited to the first release of a system, but also accumulates over time. There are architectural and code “smells” (architectural decisions that negatively impact design quality [10]), which are violations of best architectural and coding practices, respectively. They are not bugs because they do not negatively impact the functionality of a system. They are not visible to the end user in themselves. They do, however impact the internal workings of the system and its maintainability [10]. This is because they add complexity which will either complicate any further work on the system that builds on the current implementation or they will have to be eradicated, which also adds work to maintenance.

While a correlation between the amount and extent of smells in a system and its maintainability remains elusive (some smells may even be beneficial [11]), architectural decay and smells have been a motivating factor for this research.

2.1.3 Maintenance Peak Phenomena

While there is always a basic level of necessary maintenance due to factors such as technological progress, discovery of bugs and others, there are special 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 mid-1990s).

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).
While the occurrence of such special situations is not regular and predictable, they occur with such frequency and take long enough to make it so historically there has always been one going on at any given point in time over the last several decades. Their impact is so high as to make it necessary to always make provisions for them.

2.2 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.

2.2.1 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 project, 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.

2.2.2 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 or the user decide which installed version is used by a given application (this is true for the Java Virtual Machine and DirectX).

2.2.3 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 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.

<table>
<thead>
<tr>
<th>Table 1. Presumed IDPD Intensity by Category</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
</tr>
<tr>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>Non-Deployable Support Software</td>
</tr>
<tr>
<td>Firmware</td>
</tr>
<tr>
<td>Platform Software</td>
</tr>
<tr>
<td>Application Software</td>
</tr>
<tr>
<td>Infrastructure Software</td>
</tr>
</tbody>
</table>
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.

2.2.4 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.

2.2.5 Firmware
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/UEFI of an IBM-compatible computer enables the computer to load its operating system).

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

The external requirements for firmware tend to be rigid, but limited in scope, because the fulfillment of more complex 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 complex APIs.

New increments of upgradeable firmware need to provide the same interface to drivers, but have no restrictions on the hardware end.

2.3 Research Challenges
Productivity is linked to effort in the sense that a higher effort lowers productivity [12]. Software effort estimation is not exact and the productivity of an increment is subject to many influences. Few data points cover the full set of possible influences.

2.4 Research Approach
The principal research question to be addressed by this paper is:

Does IDPD vary enough between different categories of software projects to warrant making the category of a project an input parameter for IDPD prediction?

This raises the question what type of taxonomies could be used to categorize software projects regarding IDPD. For this paper, projects have been categorized by their level of abstraction from the hardware or the underlying layers.

In ascending order of abstraction, the categories are firmware, platform software, infrastructure software and applications.

(Non-deployable support software does not fit exclusively in any layer, but the only statement about it is that it has no increments, so this does not break the categorization by layers.)

2.4.1 Domain Hypothesis

As outlined further above, this paper focuses on the third IDPD hypothesis, which concerns project domains. It is reproduced here.

Domain Hypothesis: For different software development domains, the average decline in productivity over the course of a project varies significantly.

The corresponding null hypothesis is shown below.

Null hypothesis: IDPD does not vary significantly across different software development domains.

2.4.1.1 Data collection
Data has been collected from various sources that fall into the three main categories of software industry, controlled experiments, and open source. Table 2 lists the characteristics of the three different types of data sources. (UCC stands for “Unified Code Count” [13])

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 projects that satisfy all of the following criteria are 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

The core attributes of each data point are 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.

The size metrics are collected by using code counting tools to compare differentials between two baselines of the source pro-
gram. 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.

2.4.1.2 Data Collection Challenges

Major issues found in the collected data received from others have included:

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

The issue of gathering quality data has been addressed through the collection of data from controlled experiments and open source projects.

2.4.2 Controlled Experiments

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 range 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.

2.4.3 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.

2.4.4 Data Analysis

Several data points have been analyzed for IDPD and the average variation across software domains. The data used for this study are mostly in the application, infrastructure and platform domains.

Our goal is to validate the domain Hypothesis to test the IDPD percentage difference among the application, infrastructure and platform domains.

3. STATISTICAL METHODS

Statistical analysis was performed in order to prove the hypothesis one way or another. 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 4). For example the difference between the average IDPD decline in application domain and the average 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.

The Modified Thompson Tau test is used to determine if outliers exists in the data set. It uses a data set’s standard deviation and average to provide a statistically determined rejection zone:

\[ Rejection\ Region = \frac{t_{\alpha/2}n - 1}{\sqrt{n(n - 2 + t_{\alpha/2}^2)}} \]

where \( t_{\alpha/2} \) is the critical value from the t-test distribution, and n is the sample size.

To determine whether a value is an outlier, the value is compared to the rejection region value. If it is greater than the rejection region value, the data point is an outlier. Otherwise, the data point is not.

4. Data Analysis Results

The significance level is set to 0.05, which equals to a confidence level of 95%. All p-values are much bigger than the predetermined significance level (p < 0.05). This indicates that the observed result would be highly likely under the null hypothesis, which means the change of productivity from build-to-build doesn’t follow patterns. Based on t-stat values and confidence values with N-1 degrees of freedom, there is a 95% level of confidence that the null hypothesis cannot be rejected. Therefore the mean slope does not differ from the target based on the sample evidence.
Lastly, the average IDPD percentages among different domains were compared. The third hypothesis states that the average IDPD decline differs across different domains.

One-way ANOVA was used to test for average IDPD differences among three domains. Table 6 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 average IDPD differs across the three different domains.

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
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 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.

5.1.3 Staff sizes and effort in open source software
Most open source software projects do not conduct systematic collection of effort and related data (e.g. COCOMO II cost drivers). Therefore, for the purposes of this research, it has been assumed that for a limited range of increments within a minor version of projects that have been going on for many years (e.g. PHP 5.3), the staff size and the applied effort of the staff members remained either constant or did not change significantly.

5.2 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 fail-
ure. 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 multiple projects.

6. CONCLUSION
This paper has described the analysis of IDPD variation across software domains. Commercial and open source software incremental projects were studied. Across the different domains, IDPD has been measured within and between groups. The results were found to show with statistical significance that average IDPD differs across different domains. Between the platform, infrastructure and applications domains, the average IDPD factor is 5%, 18% and 9% respectively.

This confirms the third hypothesis on the basis of existing data with regard to these three domains. Further work will have to be done on firmware. (Non-deployable software does not warrant any further consideration due to the lack of increments.)

Work is already in progress to add more projects for evaluation. Additionally, the impact of COCOMO II drivers on IDPD is being researched as well.

7. FUTURE WORK
While initial data analysis progress has been made, it would be desirable to obtain more data to further the 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.

A much desired outcome of the 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.

With a fresh look at IDPD domains, we can analyze the effect of IDPD on new sets of domains other than separating projects into domains by their position in the hierarchy. Envisioned domain taxonomies include project size, project time frame, functional domain or which section of the software industry a project is most closely related to.

In addition, the impact of COCOMO II drivers on IDPD needs to be looked into. There are 22 COCOMO II cost drivers and each of these drivers can potentially influence IDPD. The challenge here is to find projects with sufficiently retrievable or determinable background information such that COCOMO II drivers can either be determined or captured by interviewing the participants. Our approach is to run a controlled experiment with graduate students and provide surveys before and after each increment to be able to capture and measure the effect.
8. References


4. Nagappan, Nachiappan. Use of Relative Code Churn Measures to Predict System Defect Density. In ICSE ’05 (St. Louis, 2005), ACM.


