Productivity Trends in Incremental and Iterative Software Development

Thomas Tan
Center for System and Software Engineering, University of Southern California
941 W. 37th Place SAL 334
Los Angeles, CA, USA
thomast@usc.edu

Qi Li
Center for System and Software Engineering, University of Southern California
941 W. 37th Place SAL 330
Los Angeles, CA, USA
qli1@usc.edu

Barry Boehm
Center for System and Software Engineering, University of Southern California
941 W. 37th Place SAL 328
Los Angeles, CA, USA
boehm@usc.edu

Ye Yang
Institute of Software Chinese Academy of Sciences
4# South Fourth Street, Zhong Guan Cun, Beijing, China
ye@itechs.iscas.ac.cn

Mei He
Institute of Software Chinese Academy of Sciences
Graduate University of Chinese Academy of Sciences
4# South Fourth Street, Zhong Guan Cun, Beijing, China
hemei@itechs.iscas.ac.cn

Ramin Moazeni
Center for System and Software Engineering, University of Southern California
941 W. 37th Place SAL 334
Los Angeles, CA, USA
moazeni@usc.edu

Abstract

In an investigating study to trace the productivity changes of a commercial software project, which uses incremental and iterative development model, we've found evidence that attributes such as staffing stability, design compatibility/ adaptability to incremental and iterative development, and integration and testing would have significant impact on modifying the productivity outcome – either positive or negative. In this report, we will present an empirical analysis to review, evaluate, and discuss these influential attributes in regard of their correlations with productivity in incremental and iterative software development. We also hope that our approach and results would contribute to initiate more research in this subject area.

1. Introduction

The advancement in hardware technology demands an equal or better improvement in producing complicated software in order to fully utilize the potential in this digital age. More software has been produced in the past decade than in years before and much more will need to be done in the future. With more expected capabilities from every application to be produced, a simple system could have thousands lines of source code at its initial release. This increase in size, however, is often inversely proportional to the time we need it to hit the market. Thus, the traditional waterfall-like process with its slow and rigorous flow of production has fallen off from engineers' favorite lists. Incremental and iterative development (IID) models are now being used by many organizations to reduce development risks while trying to deliver the product on time. [22]

As we do for all other engineering processes, we wish to evaluate the effectiveness and efficiency of the process to find ways to improve both engineering practices and management strategies. A common parameter to evaluate production is to measure productivity [19]. In an investigation effort to study the productivity trend of a commercial software project, which has been developed using incremental and iterative development model, we have found evidences that may contribute to modify productivity outcomes from build to build. This report will serve to discuss our study results and hopefully lead to more future studies in this subject area.

The rest of the report is organized as follows: we will briefly introduce our assumptions and approach for conducting this study. In section 3, we will summarize our effort of the literature review that sets

up a foundation for our data analysis. In section 4, we will quickly describe the steps to normalize the project data to produce a working data set. Following section 4, we will present out data analysis and its results in section 5. Then, we will explain the implications based on our observations from the analysis in a discussion section. Lastly, we will revisit our purpose and summarize our findings in section 7.

2. Assumption and Approach

Studying software productivity, we understand that there are numerous existing works from many different perspectives. Some present basic concepts, some introduce new metrics and measurement, and some evaluate existing arguments, therefore, in order for us to have a thorough and effective analysis, we find that it is necessary to set up our own assumptions as the basis to ensure that we are delivering in a clean and clear fashion. We make the following assumptions in defining the relationship of size, effort, and productivity for this study:

- To calculate productivity, we will use the following formula, where Size is measured in equivalent source lines of code (ESLOC) and Effort is measured in Person-Hours or Person Month as discussed in [20]. Equivalent source lines of code counts include adjustment factors for reuse and requirements volatility.

$$productivity = \frac{size}{effort} \qquad \text{(EQ-1)}$$

- For effort, we use the typical form for development effort [21], as shown below, in order to include possible adjustment due to diseconomies of scale. The factor A in the equation is the product of a project's cost driver multipliers.

$$effort = A * size^{E} \qquad \text{(EQ-2)}$$

- Combines the above two definitions, we can derive a productivity formula as below:

$$productivity = \frac{size}{A * size^{E}} \qquad \text{(EQ-3)}$$

  In this formula, we can see that productivity is simply a function of size with adjustment from exponent factor E, which evaluates the diseconomies (or economies) of scale magnitude, and factor A, which collects the cost driver ratings. In our analysis, staffing drivers may affect the productivity outcome through the calculation of cost driver A. Architecting risk resolution (RESL), such as architecting action, may affect through its influence of the diseconomies (factor E).

Since this study is based on data with uniform size and effort definitions, we will narrow our efforts to investigate causes and effects on the effort turbulence in an incremental and iterative development setting. Various methods in measuring and evaluating size will not be in the scope of this work.

Having presented our assumptions, we have also determined steps in order to fully understand what we need to analyze about software productivity. The following process outlines our approach for this study:

1) Literature Reviews: In this phase, we determined key characteristics of incremental and iterative development and what factors may modify the effort outcome in such development model. The product of this phase has been a list of potential influential factors that have higher chance to impact on effort and productivity.
2) Data Collection and Consolidation: In this phase, we gathered some relevant project data and cross checked its validity and consistency. The consolidated data set is our main working set for the data analysis.
3) Data Analysis: After the data consolidation, we performed correlations between our candidate factors and the project data, and tested a hypothesis that, all other thing being equal; productivity will not decline from increment to increment.

3. Literature Review

In the literature review, we have studied more than two dozen sources covering both software productivity and incremental and iterative development. In this section, we will provide a brief overview of some of these studies and our summary of potential drivers that may influence productivity in incremental and iterative development.

We begin with classic definition on software productivity. Boehm defines software productivity as outputs produced by the process over inputs consumed by the process [9]. A list of productivity drivers have been identified by many researchers: these drivers include resource constraints, requirement volatility, use of software tools, program complexity, customer and user involvement, personnel experience and capabilities, etc.

Scacchi [16] divided these drivers into three attribute lists: software development environment, software system product, and project staff attributes. He also described the desired conditions for each attribute in order to achieve high productivity.

Similar to Scacchi's analysis, Anseimo and Ledgard discussed the issues of software productivity from the perspectives of production environment, software products, and project process [1]. They summed up a list of influential factors that include independence between software modules, understandability, visibility, and abstraction of software architecture and implementation, and flexibility of the software production process.

In [20], Boehm et al. introduced the approach to estimate for incremental development. They described the three strategies for using incremental development and the tradeoff of each strategy. They also identified the effort and schedule distribution percentage for IID, showed the way to calculate adjusted size for increments, and provided step-by-step procedures to estimate effort, size (adjusted size for increments), and schedule by demonstrating the approach on a sample project in IID. From their discussion, we've learned that the Adaptation Adjustment Modifier (AAM) [20], which is based on the percent design modified (DM), percent code modified (CM), percent integration and test modified (IM), the assessment and assimilation (AA), the Software Understanding (SU), and Programmer Unfamiliarity (UNFM), can modify the size of an increment to accommodate reuse and integration from one build to another. We've also found traces of influential drivers such as the added integration effort due to code breakage and retesting and increment overlaps that are affected by uncertainties, which are characterized by COCOMO drivers such as Architecture and Risk Resolution (RESL), Required Reliability (RELY), Product Complexity (CPLX), and Platform Volatility (PVOL) [20].

Based on Boehm's discussion about estimate for incremental development, Benediktsson and Dalcher conducted a series of studies [4-6] estimating IID production effort and size using expanded calculations derived from Basic COCOMO [21] and COCOMOII [20]. In their reports, they noted that relative productivity, which is the productivity for IID, is inversely proportional to the scale factor exponents—meaning if a project is developed under single increment setting and it is thought to be productive, but it may not be the case if the same project is developed with IID. They also added an analysis that relative productivity is highest when the number of increment is set to 1 when scale factor exponent is fixed. Graham also observed similar trend and suggests that design integrity for IID, requirement specification for increments, and additional testing will contribute extra effort to lower productivity [10]. Moreover, both Mohagheghi [14] and Royce [15] mentioned the importance of design for increments in order to maintain overall advantages of using IID. In addition, they also discussed the challenges of integration from build to build and the uncertainty due to integration in productivity outcome.

Among other notable works, Thadhani's study [17] illustrates a unique relationship between response time, programmer skills, and program complexity, whereas Krishnan, et al., described an empirical results to link productivity and quality in terms of personnel capability, usage of tools, software process, product size, and higher front-end investments [12]. In addition to these analyses on influential factors, we've also read through other works that focus on productivity measurements such as [11], which introduces a measurement method using multiple size measures, and [3], which discusses how to apply function points on productivity measurement.

The following table summarizes the list of productivity drivers that we've found from our reviewed studies. We've also rated each driver with a relevancy degree to incremental and iterative development that we believe would help us narrow down our target space for this study.

Table 1: Productivity Drivers

Productivity Drivers	Relevant to IID	References
Software Design	Yes	[1, 4-6, 10, 12, 13, 15, 20]
Personnel Capabilities and Experiences	Yes	[7, 12, 16-18, 20]
Software Process and Increment Overlapping	Yes	[1, 12, 16, 20]
Resource Constraints (such as time, memory, hardware, etc.)	Maybe	[2, 6, 7, 12, 16-18]
Environment Parameters (such as tools, platforms, language support, etc.)	Maybe	[1, 2, 7, 12, 16, 18]
Software Integration and Testing	Yes	[4-6, 10, 14, 15, 20]
Software Reuse	Yes	[7, 16]
Software Complexity	Yes	[1, 7, 16-18, 20]
Requirement Volatility	Yes	[1, 7, 10, 13, 14, 16, 18, 20]

Factors such as resource constraints and environment parameters are applied in a broad sense to affect productivity. These factors seem to have indifferent effects on all software projects regardless of their development process models. In our current data set, these factors also remain constant from build to build, therefore, we decided to exclude them from our study. Moreover, due to lack of support from our current data set, we are unable to find solid results from Software Reuse, Software Complexity, and Requirement Volatility, so we have also decided to

push these factors to future study. For this report, we will look at the following effort drivers and see how they are represented in the project data and hopefully learn more about their trends:

1. Personnel Capabilities and Experiences: Effort data related to staffing
2. Software Design in IID: Effort data related to requirement and design phase
3. Software Integration and Testing: Effort data related to testing phase

4. Data Collection and Consolidation

The subject project of our study is a medium size commercial software application that provides facilitation capabilities to improve software development process for small and medium software organizations. It is called the software Quality Management Platform (QMP). Quality management functionalities such as process definition, project planning, data reporting, measurement and analysis, defects tracking, etc. are incrementally developed as the main features of the system over a period of 6 years and more features are scheduled to be inserted as years to come. The increments were originally determined as maintenance projects to the first build but later analysis shows some casual characteristics of incremental and iterative development. Although the overall project process does not 100% fall into the category of IID – longer increment duration and no clear-cut IID plan at the beginning, we see enough similarities to work with it and treat it as an IID process and hopefully learn more about IID especially in term of productivity.

In order to find evidence to match our targets, we first normalized our data set so they are consistent and easy to work with. To do that, we performed two sets of operations to "clean" the data to fit to our need:

1) First, we went through all the release documents, both official documentations and unofficial reports: from these reports, we are able to cross check all the size, effort, defects, and intangible cost drivers ratings – the project team did keep a list of cost drivers rating matching COCOMOII cost drivers [20].
2) After the first round of operations, we found 11 sets of increment data, however, some of them are questionable in term of data consistency – some size and effort data does not match from report to report. Therefore, we went back to the development team to dig more information in order to get our working data set.

By the end of the data consolidation phase, we are ready for the data analysis. In this working data set, we have six distinguishable increments. Each increment has duration of 6 to 11 months and a full set of efforts records for engineering phases such as requirements, design, coding, and testing. Each increment has a final deliverable following the passing of the acceptance test with official release documents and reports. There may be multiple builds in one increment, but we will only use the final delivered build as the basis for product size. We've also noted that an average of 20 staff was maintained by the project team to ensure consistent support of the development.

5. Productivity Analysis

For our data analysis, we studied three major areas and investigated the causes and effects for each in order to achieve our goals. These three major areas are staffing stability, the productivity trend, and phase effort distribution. In the analysis, since we will primarily use the final build in each increment as benchmarks to study, we will refer each increment with its final builder number, e.g., build 1, build 2, and so on.

5.1 Staffing Stability

In the staffing stability area, we have examined evidences about staffing turnover, personnel capabilities change, and team experience growth to promote the positive contribution of staffing stability. In order to evaluate such attributes, we used COCOMO II cost drivers described in [20] as the basis of our analysis. The following two tables summarize the attributes' ratings based on team members' and organization experts' rating.

Table 2: Staffing & Personnel Ratings

Build	TEAM	PCON	ACAP	PCAP
1	NOM	HI	NOM	NOM
2	HI	HI	NOM	NOM
3	HI	HI	NOM	NOM
4	HI	HI	NOM	NOM
5	VHI	LO	NOM	NOM
6	VHI	LO	HI	NOM

Table 3: Personnel Experience Ratings

Build	APEX	LTEX	PLEX
1	NOM	HI	VLO
2	HI	HI	LO
3	HI	HI	NOM
4	NOM	HI	HI
5	NOM	HI	HI
6	NOM	HI	HI

The first thing we looked at was staffing continuity (PCON). This cost driver tells us the turnover rate of staff during the span of the project life-cycle. Based on the given ratings, we can see that staffing level was very stable until build 5, when the several developers left for other jobs. At the same time, since build 5 was a large build, there were additional developers joined the team and thus created a bigger turnover rate. This also carried on to the next build. However, we are told that the core analysts and developers in the team remain in the project for all builds and there were always enough talents in the team to keep this project rolling. Consequently, there was no significant drop in personnel capabilities and experiences. As we predicted, notable improvements occur in Team Cohesion (TEAM) and Personnel Experiences – especially language (LTEX) and platform (PLEX) experiences. This is a typical growth trend in IID because of the learning and confidence gaining when bigger project is broken down to smaller increments. Other drivers such as analyst capabilities (ACAP) and programmer capabilities (PCAP) seem to have little impact by staying basically the same.

Overall, we believe that staffing is stable and the communication and experience growth of the team is expected.

5.2 Productivity Trend

Based on our assumptions, the two parameters we need to calculate productivity are size and effort. In this project, deliverable size is measured in Thousands Source Lines of Code (KSLOC) and the effort is represented in Person-Hours. The following table shows this basic information as well as the calculated productivity measured in source lines of code per Person-Hour. We've also calculated the productivity variance from the previous build to show the amount of changes from build to build. Note that the size in Table 4 refers to the equivalent new size of the build.

Table 4: Productivity by Build

Build	Size (KSLOC)	Effort (PH)	Productivity (SLOC/PH)	Productivity Variance
1	69.12	7195	9.61	0.0%
2	64	9080	7.05	-26.6%
3	19	6647	2.86	-59.4%
4	39.2	8787.5	4.46	56.1%
5	207	31684.5	6.53	46.5%
6	65	16215.1	4.01	-38.6%

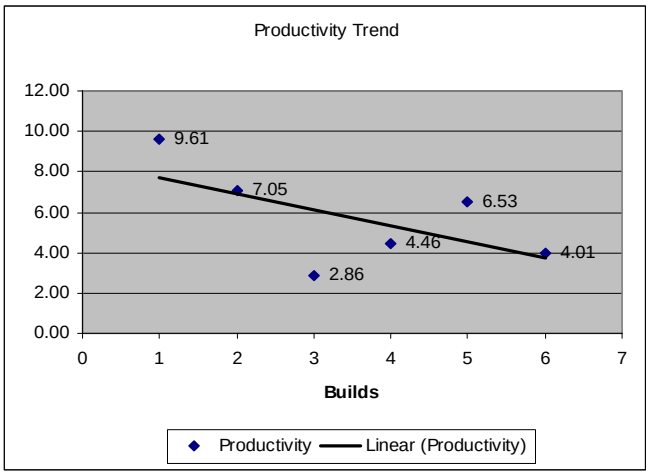

Figure 1: Productivity Trend

The actual trend line for productivity is well displayed in figure 1. The slope of the trend line is -0.76 SLOC/PH per build. Across the five builds, this corresponds to a 14% average decline in productivity per build. This is smaller than the 20% Incremental Development Productivity Decline (IDPD) factor for a large defense program reported in [23], most likely because the project is considerably smaller in system size and complexity. It is also consistent with the Lehman-Belady "laws" of program evolution involving complexity growth, continuing change, and invariant work rate [24].

According to the project progress reports and the final release documentations for these builds, we've found the following interesting facts:

1) After the first two builds, the organization was undergoing a process maturity upgrade and this project was one of the pilot programs to improve their process maturity level – this fact is also captured by the PMAT cost driver ratings as recorded in their release documents. Extra training and workshops were administered, and additional efforts for preparation of new documents and process products were required.
2) In build 3, the planned progress was not realized due to increased difficulties to add new enhancements to build 2. A significant number of design and implementation incompatibilities were found. We can observe a significant drop in deliverable size as the first two builds have steady output size whereas build 3 is just about 30% of build 2.
3) As the results of build 3's productivity drop, the project team decided to resolve future integration issues to perform a major reconstruction of the architecture to accommodate both foreseeable new features and possible enhancements in build 4, which they began with more effort on

requirements analysis and well-planned design for incremental and iterative development.

4) The reconstruction effort payoff is the most in build 5, which included 20 new capabilities that almost doubled the product deliverable size.
5) However, the productivity dropped again in build 6. It seems that the increase in size recreated the same integration issues as the team had encountered after build 2: with more components and modules, it is difficult to insert or modify without careful planning that possible redesign for future builds may be necessary to neutralize the productivity decline.

Based on these findings, we can see that with the project size increases, integrating components and modules causes notable problems, and in order to resolve the issues, the project team made decisive efforts on re-architecting which paid off in the following increments.

5.3 Phase Effort Distribution

To seek more evidence from the data, we've also studied the phase effort distribution in order to 1) verify the correctness of the productivity trend; 2) learn additional information that we may miss from looking at the productivity trend. The following tables show a compilation of the phase effort data.

Table 5: Phase Effort

Build	Rqt	Design	Impl	Test	Total
1	953	1185	3694	1363	7195
2	1240	1480	4400	1960	9080
3	923	1108	3231	1385	6647
4	1651	1886	2476.5	2774	8787.5
5	803.5	2086	12003	15466	30358.5
6	1760	1113	5554.5	7787.6	16215.1

Table 6: Phase Effort Percentage

Build	Rqt	Design	Impl	Test
1	13.2%	16.5%	51.3%	18.9%
2	13.7%	16.3%	48.5%	21.6%
3	13.9%	16.7%	48.6%	20.8%
4	18.8%	21.5%	28.2%	31.6%
5	2.6%	6.9%	39.5%	50.9%
6	10.9%	6.9%	34.3%	48.0%

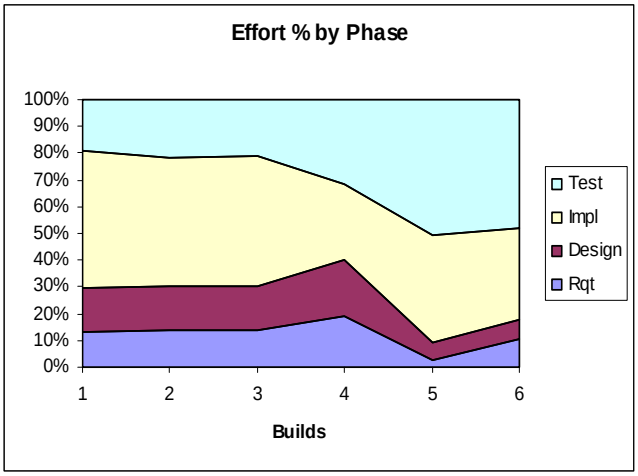

Figure 2: Phase Effort Percentage

Based on the phase effort percentage distribution, we can clearly see that requirement and design effort were steady until build 4, where the major reconstruction occurred. Then the effort ratio dropped dramatically in the next two builds, leaving most of these builds' effort dedicated to coding and testing. As we discussed early about findings from the project progress report, this effort ratio change was due to a major re-architecting action at the beginning of build 4, when productivity started to increase as a result.

In addition to the changes in requirements and design effort ratios, we also observed that testing phase effort is increasing: testing effort starts at 18.9% of the development effort to as high as 50.9% of the development effort in build 5. This increase is especially more significant after the re-architecting in build 4. From partial requirements data, we learned that in build 4 and 5, there were no additional level of services or hard-to-do requirements but mostly straightforward new features. Putting that with the testing reports for these builds, we can say that most of these extra testing efforts were dedicated to test the aftermath of inserting new components or modules along with fixing bugs that were discovered when inserting new components or modules.

Another discovery we made from digging up the reports and surveying the project team members is that by build 5, an independent sub-team of developers was formed to dedicate to testing: these testers were assigned to generate test plans, perform integration and regression tests as necessary, and finalize the deliverables with acceptance tests. The addition shows a clear realization by the project management to insert more man power in order to accommodate increase in testing efforts.

Putting these two trends together, we can observe another notable characteristic: if there is a steady allocation of efforts to every phase in the

development, such as build 1 to build 3 when a similar percentage of efforts was allocated to requirement, design, coding, and testing, the productivity will drop (from 9.61 SLOC/PH to 2.86 SLOC/PH). We also see this trend a little bit from build 5 to build 6. On the other hand, productivity may rise after major allocation of efforts to requirement and design, but we have not found supportive evidence from our data analysis to be able to draw a range to demonstrate how long this re-architecting lasts in effect.

6. Discussion

The main advantage of risk-driven incremental and iterative development is spreading and mitigating risks in small steps (meaning small loss). In many cases, the risky items are usually those that are not familiar of, therefore, many practitioners would prefer to start with something they can relate or know of and gradually build around that simple subset until finally reach full functionalities. This approach allows them to gain enough knowledge and experience with the production in order to move on to the next increment which they will face something a little more challenging. This process model also ensures early realization of business values as well as other project requirements with quick deliveries for users and clients to react upon [5, 13]. However, as the user base grows, there is more concern with reliability, which is hard to retrofit into an easiest-first code base.

Productivity of an incremental development process is not exactly the same as measuring productivity in a conventional waterfall like setting. From our productivity analysis, we learn three things:

1) Staff stability helps to improve team cohesion and personnel experience in term of driver ratings
2) Re-architecting helps to maintain productivity but the effect may not last long
3) Integration becomes more difficult from build to build – as more components and modules are added to the system

As we mentioned earlier, incremental and iterative development process exploits the fact that a bigger project is breaking into smaller pieces and engineers can start a simple build and work around this initial set while gaining experience and knowledge to ensure deliveries. This learning advantage from build to build can help the engineers to learn more about the system requirements and architecture as well as languages, tools, and platforms used to develop the system, thus reduce effort in future increments while maintaining a stable staffing level. This may also resolve risks early and thus save effort due to risk assessment and analysis in future builds. This reduction of effort may have positive effects that increase productivity from build to build or at least maintain a constant productivity rate throughout the development lifecycle. From our analysis, we see a stable staffing level for this project which leads to improvements in both team cohesion and personnel experience in term of driver ratings. Based on [20], these two cost drivers may contribute slight productivity increase such that they can bring down the overall effort as they get higher in ratings.

However, this gain in productivity can be easily negated by the increase in integration difficulties. We observe this fact from our analysis that as more components and modules are inserted into the system, additional effort is required. Combined with our literature reviews, we identify two sources that most likely to contribute to this additional effort: 1) deficiency in design for integration and code breakage; 2) the necessity of more testing and fixing in every level to ensure that the additions would work correctly with all other existing components or modules.

The design deficiency may be the results of requirement volatility or incompatible easiest-first design of earlier increments, which usually means that the original designer did not understand design for incremental and iterative development or did not design enough for multiple increments. This type of deficiency may cause severe rework on both the design and the actual implementation. According to our data results, this project experienced a 59% drop of productivity from build 2 to build 3 due to this type of deficiency and thus triggered the necessary re-architecting in build 4. It is clear that careful architecting or re-architecting can prevent or cure such design problem and have positive effect on productivity by easing some integration difficulties. Due to limited data resources, we are not sure of the extent of influence due to this design deficiency or how long the re-architecting effort last in effect. It may depend on the adaptability of the architecture to incremental and iterative development and the capabilities of the analysts and designers to make modifications on the design. Theoretically, we believe a well-designed architecture that covers most foreseeable evolutionary requirements and have prepared proper connectors for future integration as well as risk mitigation plans if there will be changes to any of the existing or evolutionary requirements will have smaller productivity decrease effect. On the other hand, if not carefully planned and designed, extra effort may be stacked on to future increments exponentially due to the diseconomies of scale as introduced by Boehm [21].

The code breakage is almost a certain thing in incremental and iterative development. Any additional sources line of code inserted in the original code base will need to be recompiled and tested. These modifications will require additional efforts for understanding previously developed code, analyzing the effects of the insertion, and testing for possible breakage after insertion. For the code breakage part, staff capabilities and experience will play a dominate role such that if the same people developing the previous build is working on this new build, the best he/she can do is to minimize the effort to understand and analyze the code, however, extra effort of implementation level testing will still cost extra that may equalize the gain from staff experience. As the result, we believe that in most cases of incremental and iterative development, with the staffing experience and team cohesion likely to improve, the code breakage effect should become smaller and smaller, thus decrease additional effort due to code breakage and increase productivity.

Last but not least important, the impact of extra testing and fixing seems to be inevitable in well-organized development processes; that is, in order to ensure deliverable quality, the project team will have to perform unit and regression test after every insertion or modification to system components or module. The effect of this rigorous practice is well represented in our data analysis that the overall testing and fixing effort percentage rise as the project become larger and larger with more and more capabilities developed. In addition to the extra effort for regression tests, integration tests also add negative contributions as deliverables grow larger. Each new component or module may have a number of inter-dependencies to other components or modules, each of these inter-dependencies will have to be tested for its correctness and performance to ensure overall system functionalities. Overall, testing and fixing effort, including both regression and integration testings, grow significantly as the number of total components or modules increase. And when problems are found, the rework can involve the full body of software, thus requires more additional attention.

7. Conclusion

As pointed out by Boehm [8], software projects are growing larger in size and become more and more complicated to fulfill our need in every possible way while delivery requires faster turnaround time. Incremental and iterative development serves well in this regard for its user-oriented approach and risk-driven process. In our attempt to understand and evaluate productivity in such setting, we conducted a research study to find evidences to support that factors such as staff capabilities and experience, software architecture, and build integration would have impacts to the productivity trend from build to build. Our analysis results help us to realize the following points to respond to our goals as stated in section 3:

- Staff capabilities and experience will be an influential factor such that not only it helps to gain confidence and resolve some experience related risks, it has significant impacts on reducing implementation level code breakage and possibly faster resolution for design deficiency if such design problem occurs.
- Software architecture will have a driving effect to alter productivity such that a well-planned and organized architecture that complies with incremental and iterative development will most likely to increase productivity. On the other hand, a deficient architecture will result major decline in productivity that requires re-architecting to bring the project back.
- Other than design deficiency and natural code breakage for integrating new or modified components or modules, additional testing and fixing efforts are required in incremental and iterative development. Such extra efforts are inevitable and the only way to balance it is to have more effective architecture and more capable/experienced staff.

Although the results are very positive that we have found linkage to prove the impacts of our goal factors, there are still much more to improve our results: One of our follow up study will be focused on normalizing new size by product effort multipliers and effort by personnel effort multipliers as discussed in COCOMO II [20]; another interesting experiment we can do is to apply CodeCount™ with "Diff" function on each build to find a new equivalent size with integration rework and compare that with actual size, we can then use the results to adjust the calculation parameters as discussed in Appendix B of [20]. In addition to our planned future researches, we also wish to call for participation to include more projects in order to solidify our findings. And we also hope that we have demostrated some methods in conducting this type of research study to continuously provide new discoveries in the research area of productivity in incremental and iterative software development.

As a final remark, our hypothesis, that productivity will not decline from increment to increment, is rejected based on our current results, which show a clear decreasing trend of productivity. The effects of staffing, design, and testing help in modifying the outcome of productivity based on our

data analysis. With each contributing different amount and direction – either positive or negative, it is important to balance these attributes in order to have a consistent production output as evaluated by productivity.

8. Acknowledgement

This work is supported by the National Natural Science Foundation of China under Grant Nos. 60573082, 60873072, and 60803023; the National Hi-Tech R&D Plan of China under Grant Nos. 2006AA01Z182 and 2007AA010303; the National Basic Research Program (973 program) under Grant No. 2007CB310802.

9. References

[1] Anseimo, Donald. Henry Ledgard. “Measuring Productivity in the Software Industry”, Communication of the ACM. Vol.46, No.1, November 2003. Page 121-125.

[2] Banker, Rajiv. Srikant M. Datar, Chris F. Kemerer. “A Model to Evaluate Variables Impacting the Productivity of Software Maintenance Projects”, *Management Science*. Vol. 37, No 1. January 1991. Page 1-18.

[3] Behrens, C.A., “Measuring the Productivity of Computer Systems Development Activities with Function Points”, IEEE Transactions on Software Engineering, Vol.9, Issue 6, 1983. Page 648-652.

[4] Benediktsson, O., D. Dalcher, K. Reed, M. Woodman, “COCOMO-Based Effort Estimation for Iterative and Incremental Software Development”, *Software Quality Journal*, November 2003. Page 265-281.

[5] Benediktsson, Oddur. Darren Dalcher. “Effort Estimation in Incremental Software Development”, IEE-Proceedings – Software, Vol.150, Issue 6, December 2003. Page 351-357.

[6] Benediktsson, O., D. Dalcher, “Estimating Size in Incremental Software Development Projects”, IEE Proceedings – Software, Institution of Engineering and Technology, 152 (6). Page 253-259.

[7] Boehm. B., M.H. Penedo, E.D. Stuckle, R.D. Williams, A.B. Pyster, “A Software Development Environment for Improving Productivity”, *Computer*, Vol.17, Issue 6, June 1984. Page 30-44.

[8] Boehm. B., “A View of 20th and 21st Century Software Engineering”, Proceedings of the 28th International Conference on Software Engineering, 2006. Page 12-29.

[9] Boehm, B.W. “Improve Software Productivity”, *Computer*, Vol. 20, Issue 9, September 1987. Page 43-57.

[10] Graham, D.R., “Incremental Development and Delivery for Large Software Systems”, Colloquium on Software Prototyping and Evolutionary Development, IEE, 11 November 1992. Page 2/1-2/9.

[11] Kitchenham, Barbara. Emilia Mendes. “Software Productivity Measurement Using Multiple Size Measures”, IEEE Transactions on Software Engineering. Vol. 30, No. 12, December 2004. Page 1023-1035.

[12] Krishnan, M.S. C. H. Kriebel, Sunder Kekre, Tridas Mukhopadhyay. “An Empirical Analysis of Productivity and Quality in Software Products”, *Management Science*. Vol. 26, No.6, June 2000.

[13] Larman, Craig. Victor R. Basili. “Iterative and Incremental Development: A Brief History”, *Computer*, Vol.36, No.6. June 2003. Page 47-56.

[14] Mohagheghi, P., B. Anda, R. Conradi, “Effort Estimation of Use Cases for Incremental Large-Scale Software Development”, Proceedings of the 27th International Conference on Software Engineering, 2005. Page 303-311.

[15] Royce, W. “TRW’s Ada Process Model for Incremental Development of Large Software Systems”, Proceedings of the 12th International Conference on Software Engineering, 1990. Page 2-11.

[16] Scacchi, Walt. “Understanding Software Productivity”, Advances in Software Engineering and Knowledge Engineering, Vol.4, 1995. Page 37-70.

[17] Thadhani, A.J., “Factors Affecting Programmer Productivity During Application Development”, *IBM Systems Journal*. Vol.23. No.1. 1984. Page 19-35.

[18] Vosburg, J., B. Curtis, R. Wolverton, B. Albert, H. Malec, S.Hoben and Y.Liu, “Productivity Factors and Programming Environments”, Proceedings of the 7th international conference on Software engineering, 1984. Page 143-152.

[19] Wikipedia. “Iterative and Incremental Development”, http://en.wikipedia.org/wiki/Iteartive_and_incremental_devleopment

[20] Boehm, B., et al., *Software Cost Estimation with COCOMO II*, Prentice Hall, New Jersey. 2000.

[21] Boehm, B., *Software Engineering Economics*, Prentice Hall, 1981.

[22] Larman, C., *Agile and Iterative Development: a Manager’s Guide*, Addison- Wesley Professional, 2003.

[23] Boehm, B., “Future Challenges for Systems and Software Cost Estimation”, Proceedings of the 42nd Annual DoD Cost Analysis Symposium, Feburary 2009.

[24] Lehman, M, L. Belady, *Program Evolution: Process of Software Change*, Academic Press, 1985.