On the Productivity of Agile Software Practices: An Industrial Case Study

Frank Maurer & Sebastien Martel

University of Calgary
Department of Computer Science
Calgary, Alberta, Canada, T2N 1N4
{Maurer, smartel}@cpsc.ucalgary.ca

Abstract. In this paper, we present a case study comparing the productivity resulting from applying some agile practices (from Extreme Programming) with a more conventional OO software development approach. We show four productivity metrics gathered from a small software company before and after adopting agile practices. The data shows strong productivity gains.

1. Introduction

Agile methods [1, 2, 8, 13] arguably are some of the most promising approaches to software development nowadays and their fast adoption rate in industry is surprising. Extreme programming (XP) [4, 5, 8] is the most visible of these methods. XP is gaining momentum because of its promise of increased development productivity and higher code quality. A short summary of XP is available in [10].

In this paper, we report on an industrial case study that focuses on one of the XP claims: development productivity. We gathered industrial data from one company. Bitonic Solutions Inc. (http://www.bitonic.ca/frontpage/index.html) is based in Calgary, Canada. Bitonic has 9 software developers and is developing software in the area of e-Business applications. Its focus is on applying advanced server-side Java technologies for solving their customer's problems. It was using ad-hoc OO software development practices (use cases, UML design, etc.) before adopting agile practices. The data collected represents a view on transitioning from an ad-hoc OO process to a process inspired by agile engineering practices.

The paper is organized as follows: In Section 2, we briefly explains the development process used by Bitonic before adopting some XP practices and then describes their current process in more detail. The next section contains the empirical data that we gathered. In Section 4, we discuss the empirical results. The last section summarizes the paper and provides an outlook on future work.

2. Bitonic Development Process

Originally, Bitonic followed an ad-hoc OO methodology. Requirements analysis was partially based on use cases. The design process was using standard UML

diagrams. Implementation was done using Java 2 Enterprise Edition technologies. Their customers provided quality assurance. Unfortunately, we do not have much information about their original process: it was rather ad-hoc and created quite a bit of documentation (which often in the end did not provide much value to the development effort). Bitonic always interacted closely with the customer representatives. They did not produce any sizeable amount of test drivers before switching to XP.

Bitonic saw XP as a way to provide better value to their customers and adopted some XP practices in February/March 2001. Besides incorporating some of the XP practices, the company context stayed more or less stable over the timeframe providing the data for our investigation: the data was gathered from work for the same client on the same product and the team was stable over the time when agile practices were introduced.

In this section, we will examine the XP practices used by Bitonic and the reasons why they deviate from the XP process described by Beck [4].

Customer Involvement. Two key employees of the client's company represent the customer. They provide requirements and decide on priorities for the next release. In addition, the client provides one dedicated QA person, 3 representatives helping out in areas that require very detailed knowledge and 9 users of the system that help ensure that functionality that they request gets properly implemented.

The customer representatives are not co-located with the development team but are in close proximity in the city so that they are accessible fast when telecommunication is not sufficient to resolve issues. Instant messaging is seen as "fairly key to the efficient communication" to sort out smaller issues.

At the beginning of their migration to agile practices, Bitonic and the customer representatives had decided that anyone on the team could call anyone at anytime. This approach consumed too much time, since the client would constantly be in communication with the programmers, often reporting problems twice. Now, each programmer is allowed to contact any of the client's personnel whereas only the two key employees can call two designated programmers.

Estimation and Metrics. User stories are placed in a specific iteration immediately or during the next meeting with the customer representatives. After gathering estimates from the programmer, user stories are brought to the next meeting to be discussed with the client. When user stories are created to close to the meeting date, it is impossible to provide estimation for them in time for the next meeting. As a result the two designated programmer end up providing an estimate for a task that they might not be assigned too. This has proven to be very problematic. It is planned that the person responsible for a task should estimate the effort for it.

The team is currently not using any charts or diagrams to show the current status of the iteration. Instead the project manager keeps track of the progress and steers the team accordingly. The justification is that according to Bitonic the programmers are not interested in this information but just in programming.

Documentation. Even though Bitonic has drastically reduced the amount of analysis and design documentation since adopting XP, they recognize the need for a certain amount of documentation. The documentation is used to get approval from the client executive committee for implementing new features and major tasks (e.g. major redesigns etc.).

Currently the team provides limited design documentation for all tasks that are estimated to last longer than 8 billable hours. The value "8h" was derived from experience as Bitonic believe that they save development effort by roughly documenting more complex features. Bitonic spends about 1h on documentation for 12h of development time.

Testing, Integration and Development Cycle. Since the adoption of agile practices, the team has changed their testing practice to coincide more closely with the XP methodology. They are using JUnit to aid in their unit testing task. Sometimes, tests are still done after the production code because time to the next release was approaching fast and the team wanted to ensure that the feature will be present in the next release. Additional quality assurance is provided by the customer company on the same level as it was before Bitonic adopted XP practices.

As for code integration, they use the technique suggested in 4: use one integration machine and if the new code breaks the current tests suite then it will be fixed immediately.

One iteration is 1 week in length. The code is deployed approximately each week on a test system, and each month on the live system (code released to customer).

Pair Programming. Pair programming is only done for major task. Bitonic estimates that 50% of the production code is written using pair programming. The fear is that pair programming on smaller task, for example small bug fixes, will reduce productivity. The team members that used pair programming liked it.

The lower level of pair programming (compared with standard XP practices) has caused the creation of programmers that are very knowledgeable in specific parts of the system. A (planned) future increased use of pair programming will help ease this problem and improve the understanding of the entire system for each programmer.

Contractual Issues. The company has a service license agreement (SLA) with the customer guaranteeing X amount of hours. The numbers of hours X are calculated using 80% utilization of a 40-hour workweek instead of yesterday weather [5] since not enough historical data was available at the time of the contract signing. Some overtime was necessary in the beginning until the team became a good working unit. According to Bitonic, the team now does not work more than what they are paid for.

Beside the contract, Bitonic goes a long way to create trust with the client. The client even has access to internal data, including time sheets and bug tracking system.

Tool Support. Bitonic is currently using four separate tools to coordinate and manage their development effort. Bugzilla is used to keep track of any project related issues. This includes bugs found, requests for new features, and enhancements to existing functionality. In a sense, Bugzilla replaces user story index cards in XP or the product/sprint backlog in Scrum. Time Tracker is an application developed in-house, which is used for keeping track of the effort spent for the client's projects. MS Project's Gantt charts are used to communicate visually the status of the current iteration. The open-source CVS is used for version management.

3. Empirical results

In this section, we report on the empirical findings of the case study. We will start by defining the metrics that we gathered. Then we introduce the average results of the study followed by graphs for each metric that we collected. A discussion of the empirical results can be found in the last subsection.

The goal of the paper is to gather initial evidence on the effects of agile practices on the productivity of software development. We do this by comparing the productivity of the two different development processes used by Bitonic. The first process – labeled "Pre XP" – was a n ad-hoc OO development process as explained in Section 2. The second is based on extreme programming practices and is labeled XP in this paper. [1]

3.1 Metrics definition

Productivity is defined in classical textbooks (e.g. [6, p. 408]) as

$$(1) \quad productivity = \frac{size}{effort}$$

While this formula is easily defined, measuring size and effort objectively and in a way that is useful for the practitioner is not trivial. A good size measure should reflect the amount of functionality provided to the customer. Effort should measure the amount of work for developing the new functionality.

Concretely, while we were collecting several size metrics (see below) as part of the case study, effort was measured by the number of hours billed in a given time period. According to Bitonic, the hours billed closely reflect the actual amount of work done. The number of hours billed per month is 3.4% higher in the XP phase than in the pre-XP phase.

We decided to use billable hours because this is a hard metric of what the customer paid for the effort of the development team. The billable hours were determined based on accounting data from Bitonic and were the basis for invoicing the client.

There is quite a bit of discussion on problems related to various size metrics in the literature (e.g. [6]). To stabilize the empirical results, we gathered several different size metrics. Concretely, we collected the following metrics from Bitonic's version control system and their bug tracking system:

---

[1] As Bitonic has not adopted all XP practices, labeling their process as XP is not absolutely correct. Nevertheless, we use the label "XP" in the paper for brevity's sake.

1. Number of new lines of code in a given release
2. Number of new methods in a given release
3. Number of new classes in a given release
4. Number of bugs fixed in a given release
5. Number of features added (and bugs fixed) in a given release

The size metrics were collected based on a release of the software provided to the customers at a given date. Release dates were approximately one month apart.

Number of new lines of code in a given release (NLOC): Lines of codes (LOC) were counted for two separate entities: the java source files (JLOC) and code for html templates (HLOC). The HTML code from the templates is used in conjunction with Java Servlets to provide the equivalent functionality of Java Server Pages (JSP). Since the development of Servlets also includes HTML code developed for the html templates we include those lines of code in our total. The HLOC for the HTML templates were determined by counting the carriage returns in the files. HLOC includes comment lines. The lines of Java source code JLOC for a given release were computed using the JavaNCSS [7] application. The way JavaNCSS computes the lines of code is roughly equivalent to counting ';' and '{' characters in Java source files. The number of new lines of code (NLOC) was determined by subtracting the LOC from the last release from the LOC of the current release.

$$NLOC = (JLOC + HLOC)_{new} - (JLOC + HLOC)_{old} \quad (2)$$

Number of new methods in a given release (#methods) and *number of new classes in a given release* (#classes): The number of classes and methods in a given release was determined using JavaNCSS.

$$\#methods = \#methods_{new} - \#methods_{old} \quad (3)$$

$$\#classes = \#classes_{new} - \#classeds_{old} \quad (4)$$

Number of fixed bugs in a given release (#bugs) and *number of new features in a given release* (#features): These numbers were obtained by counting the newly closed entries in Bitonic's Bugzilla database for the timeframe of a release.

Based on these five base metrics, we defined four productivity metrics for our study.

$$p_1 = \frac{NLOC}{effort} \quad (5)$$

$$p_3 = \frac{\#classes}{effort} \quad (7)$$

$$p_2 = \frac{\#methods}{effort} \quad (6)$$

$$p_4 = \frac{\#bugs + \#features}{effort} \quad (8)$$

The next subsection presents the average results for these productivity metrics. After that we show the graphs for these metrics.

3.2 Productivity averages

Table 1 summarizes the metrics we have collected for our case study. It shows productivity averages for the pre-XP timeframe and for the XP timeframe as well as

the percentage change. The percentage change was determined by the following formula

$$\%_change = \frac{Average_XP_Value}{Average_Pre_XP_Value} - 1 \quad (9)$$

In total, we collected data from 16 releases. The first nine releases were developed following the Pre-XP process and are the basis for the first row in the table. The last 5 releases followed the XP approach and are used to calculate the numbers in the second row. Two intermediate releases were omitted for calculating the averages as they are based on a transitional phase where part of the team already adopted XP while another part still relied on the Pre-XP process.

	$\frac{NLOC}{effort}$	$\frac{\#methods}{effort}$	$\frac{\#classes}{effort}$	$\frac{\#bugs + \#features}{effort}$
Ave Pre-XP	10.3	0.36	0.05	0.073
Average XP	17.0	1.45	0.21	0.069
% Change	**65.0**	**302.1**	**282.6**	**-4.96**

Table 1 Productivity averages

Overall, p_1-p_3 indicate strong productivity increases after adopting XP. p_4 shows a productivity decrease after adopting agile practices. A detailed analysis and an interpretation of the data can be found in Section 0.

The following table shows the average LOC metrics per release for the Pre-XP and the XP phases. From the table, we can conclude that the work shifted more towards the development of Java source code.

	JLOC	HLOC	NLOC
Ave Pre-XP	4237.2	3267.67	7504.9
Average XP	10608.0	2194.40	12802.4
% Change	**150.4%**	**-32.8%**	**70.6%**

Table 2 Average LOC metrics per release

Table 3 shows the average sizes per closed entry in the bug tracking database for Pre-XP and XP phases. For example, the second column shows that in the Pre-XP phase, for closing a bug the developers were writing 140.18 NLOC in average while in the XP phase, they usually needed 245.26 NLOC (an 73.6% increase). In Section 4.2, we need this data to interpret the results of the empirical study.

	$\frac{NLOC}{\#bugs + \#features}$	$\frac{\#methods}{\#bugs + \#features}$	$\frac{\#classes}{\#bugs + \#features}$
Average Pre-XP	141.31	4.95	0.74
Average XP	245.26	20.93	2.97
% Change	**73.6%**	**323.0%**	**302.6%**

Table 3 Average size per bug in database

3.3 Charts

The following charts show the productivity metrics p_1-p_4 over the timeframe of the investigation (05/2000 – 09/2001). The shaded areas in the charts indicate which values were omitted when we collected that averages above.

Figure 1 shows the development of p_1 over time. The linear trend line indicates an increase of new lines of code per billable hour. Most individual values of the XP timeframe are higher than the average of the Pre-XP phase.

In Figure 2, p_2 is displayed. The linear trend line shows a steep angle – fitting to the 4-fold average productivity increase. With the exception of one data point, all XP values are way above the Pre-XP values.

Figure 3 presents the graph for p_3. In also contains a steep linear trend line. The last chart in Figure 4 shows the p_4 metric over time.

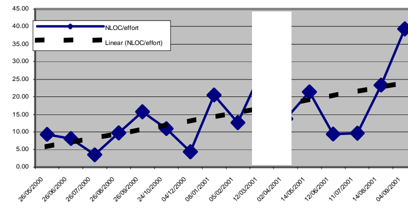

Figure 1: NLOC/effort

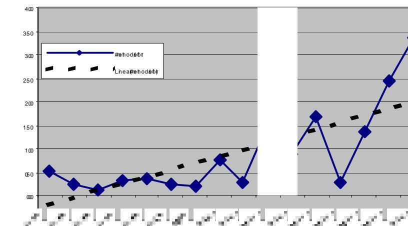
Figure 2: #methods/effort

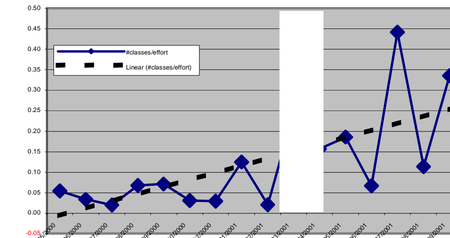

Figure 3: #classes/effort

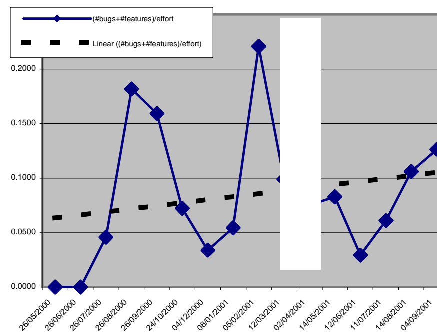

Figure 4: (#bugs=#features)/effort

4. DISCUSSION OF DATA

In this section, we interpret the empirical findings of the study and put it into the business context. In addition, we discuss issues resulting from the process how the empirical data was gathered.

4.1 Data issues

The productivity metrics p_1-p_3 rely on objective data as the measuring process was automated.

We collected LOC[2] as it is a standard size metric in industry. There are lots of issues related to LOC as a size measure including

- LOC does not directly represent progress towards the customer requirements
- LOC is not an OO measure
- LOC numbers depend on programming style of the developer

NLOC basically combines the HLOC metric for HTML code (including comments) and the JLOC metric for Java Code (excluding comments) into one metric. One argument against NLOC is that the effort for writing one line of HLOC

[2] As mentioned earlier, we concretely collected JLOC and HLOC and computed NLOC.

code is different from one line of JLOC code. We decided to use NLOC anyway as we are not interested in the LOC numbers by themselves but in the relative change from the Pre-XP phase to the XP phase. For this, we only needed to ensure that the numbers for both phases are collected in the same way.

NLOC counts the number of *additional* lines of code in the code base from one release to the next. This is slightly different from *new* lines of code: if one line of code is deleted and two new ones are added, NLOC will result in 1 NLOC line. As a result, NLOC may underestimate the numbers of new lines of code as the deleted lines reduce it.

One of the key practices of agile methods used by Bitonic is refactoring. Refactoring often results in reductions to the LOC size of the code base. This reduction is often coupled with an increased number of methods in the code base. This concurs with the findings of our study as the average percentage increase in #methods/effort is much higher than NLOC/effort.

We also gathered basic OO metrics (#classes, #methods) to reflect the OO paradigm in our productivity data and to check if we would get completely different results compared with NLOC.

While NLOC, #classes, and #methods can be automatically measured, p_4 relies on manual counting. In principal, this could have been automated. Hence, we would argue that p_4 also relies on objective data.

4.2 Data interpretation[3]

Switching to XP has – so far – paid off for Bitonic and their customer. The productivity metrics p_1-p_3 show strong gains for the XP-based development process. Increasing the productivity by 65%-302% (depending on the metric used) is a big step forward. Using these results as a guide, one would recommend adopting XP for software development teams working in a similar context as Bitonic (small teams, close customer relationships, server-side Java development).

One could argue that the increase in NLOC, #methods, and #classes and the corresponding productivity metrics could be attributed to the development of automated test drivers in the XP phase. Test driver code is often simpler and easier to write than production code. Hence, the amount of code that can be developed in one billable hour should increase. It is unlikely that the test drivers are the only reason for the productivity gains as the amount of test code constitutes only about 30% of the total number of classes and about 15% of the total LOC.

The results for p_4 is more difficult to interpret. p_4 is the productivity metric in our study that most closely reflects the major question from SE practitioners: how much user relevant functionality can I get for a given effort? The result for p_4 looks very disappointing for XP. The average shows a slight productivity *decrease* (-4.96%). In our opinion, this number is misleading. It probably results from a change of the average "difficulty" or average "complexity" to close a bug/feature in Bugzilla. The development efforts for adding two different features or for fixing two different bugs may be vastly diverse. In fact, the efforts can be orders of magnitude apart. Table 3 shows a 1.75-fold increase in NLOC for adding a feature/fixing a bug in the XP phase: while in Pre-XP 141 NLOC were in average sufficient to add a feature or fix a bug, now 245 NLOC are required. Looking on #methods and #classes also supports this interpretation. When comparing Pre-XP and XP numbers, both values show about a 4-fold increase in size for adding one feature/fixing one bug. Assuming that

---

[3] As we are trying to make sense of the collected data, the interpretation is somewhat subjective and may be biased.

program size is somehow correlated with problem difficulty, the numbers from Table 3 show that during the XP timeframe the team simply tackled more complex problems[4]. p_4 on the other hand assumes that the average problem complexity stays the same as each added feature/fixed bug counts only 1 (independent of its difficulty). Loosing less than 5% productivity when the average task difficulty is increased 2-4 fold may actually be seen as an increase in productivity.

Obviously, other interpretations of the p_4 result are possible: One way to argue is that adopting agile practices did reduce the productivity of the team – but that would contradict the impression of Bitonic's president who confirmed our initial interpretation: "I would strongly agree with the conclusion here, by implementing XP we have spent a lot less time implementing features that were misunderstood or improperly communicated between the developers and the client. The code produced today contains a lot more punch than previously." Another argument centers around a possible Hawthorne effect. Unfortunately, the data that we have does not allow to evaluate this.

4.3 Limitations and issues

The case study presented in this paper provides a data for empirically validating claims made by the agile community on productivity increases. As the sample data covers only one software development team dealing with one customer, the generalizability of the findings is limited.

Also, the number of data points underlying each graph is small. In total, we collected data from 16 releases, used 9 points for calculating the Pre-XP averages and 5 points for determining XP averages. 2 points were omitted as the developers did use both approaches in these releases.

The spiky nature of the chart in Figure 4 needs more investigation. The development effort per release is not constant in the project. In some releases, the billable hours are very high as the customer requested more work to get things done. This resulted in some overtime work for the developers. This overspending resulted in a reduced effort for the next release as the contract is based on an average hours per month. An initial observation is that the productivity usually goes up when the actual billable hours are low. We see two explanations for this:

1. Agile method stress sustainable workweeks and claim that only fresh people are highly productive. This could be a reason for productivity increases when actual hours are low.
2. For contractual reasons, Bitonic needs to average out the billable hours over several releases. The high productivity values may result from not billing the customer for effort that was actually spent on the project. In this case, more hours would go into the project than we saw from the accounting data. According to their president, this was not the case (they know that "some" extra work may be done "but not much").

5. Conclusion and future work

In this paper, we presented the results of an industrial case study on the productivity of agile practices compared to a more conventional OO development process. Overall, adopting agile practices provided strong productivity gains to the company under investigation. We believe that these gains could be attributed to the

[4] Bitonic confirmed that they are now working on more complex issues.

new practices as the development context stayed stable: the team did not fluctuate during the last year, the customer is the same, the project is the same, technologies that are used for implementing the system are the same. Hence, our study provides an initial data point indicating that some of the claims concerning productivity of agile methods may actually become empirically validated in the future.

Lots of work still needs to be done. First, more data points need to be gathered to provide evidence that agile methods are *generally* a highly productive approach to software development in small, co-located teams.

Second, we are planning to monitor the Bitonic process over a much longer period of time to determine if the current gains will result in overly high maintenance costs some time down the road. Our current hypothesis is that increased maintenance costs will not occur because the available data indicates that the project is already switching to a more maintenance-focused stage.

Another task on our desk is to analyze the Bitonic data concerning software quality: how did adopting XP practices affect software quality? As the team currently was spending more effort on bug fixes than in the past: when were these bugs introduced? In the Pre-XP phase or in XP? If the later: why weren't they caught by testing?

6. Acknowledgements

We would like to thank Marcos Lopez, the president of Bitonic, and his whole team for supporting the research underlying this paper with their time & insights.

Part of the research was sponsored by the Alberta Software Engineering Research Consortium (www.aserc.ab.ca) and by NSERC (www.nserc.ca).

7. References

1. Mike Beedle, Ken Schwaber: Agile Software Development with SCRUM, Prentice Hall 2001.
2. Alistair Cockburn: Agile Software Development, Addison Wesley, 2002.
3. Allistair Cockburn, Laurie Williams. The Costs and Benefits of Pair Programming. Extreme Programming Examined. Massachusetts: Addison Wesley. 2001, 223-243.
4. Kent Beck. Extreme Programming Explained: Embrace Change. Reading, Massachusetts: Addison Wesley Longman, Inc., 2000.
5. Kent.Beck, Martin Fowler. Planning Extreme Programming. Reading, Massachusetts: Addison Wesley Longman, Inc., 2001.
6. Norman E. Fenton, Shari Lawrence Pfleeger: Software Metrics – A Rigorous & Practical Approach, PWS Publishing Company, Boston USA, 1996.
7. JavaNCSS – A Source Measurement Suite for Java, http://www.kclee.com/clemens/java/javancss/ (Last visited March 2002)
8. James A. Highsmith III: Adaptive Software Development: A Collaborative Approach to Managing Complex Systems, Dorset House Publishing 2000.
9. Ron Jeffries, Ann Anderson, Chet Hendrickson. Extreme Programming Installed. Reading, Massachusetts: Addison Wesley Longman, Inc., 2001.
10. Frank Maurer, Sebastien Martel: Extreme Programming: Rapid Development for Web-Based Applications, IEEE Internet Computing, January/February 2002 (Vol. 6, No. 1)
11. James Newkirk, Robert C. Martin: Extreme Programming in Practice, Addison-Wesley, 2001.
12. Roger S. Pressman: Software Engineering: A Practitioner's Approach, Fourth Edition, McGraw-Hill 1996.
13. Jennifer Stapleton: DSDM Dynamic Systems Development Method, Addison Wesley 1997.