Identifying Wasted Effort in the Field via Developer Interaction Data

Gergő Balogh*, Gábor Antal*, Árpád Beszédes*, László Vidács†, Tibor Gyimóthy* and Ádám Zoltán Végh‡
*Department of Software Engineering, University of Szeged, Hungary
†MTA-SZTE Research Group on Artificial Intelligence, University of Szeged, Hungary
‡AENSys Informatics Ltd., Szeged, Hungary
{geryxyz, antalg, beszedes, lac, gyimothy}@inf.u-szeged.hu, adam.vegh@aensys.hu

***Abstract*—During software projects, several parts of the source code are usually re-written due to imperfect solutions before the code is released. This wasted effort is of central interest to the project management to assure on-time delivery. Although the amount of thrown-away code can be measured from version control systems, stakeholders are more interested in productivity dynamics that reflect the constant change in a software project. In this paper we present a field study of measuring the productivity of a medium-sized J2EE project. We propose a productivity analysis method where productivity is expressed through dynamic profiles – the so-called Micro-Productivity Profiles (MPPs). They can be used to characterize various constituents of software projects such as components, phases and teams. We collected detailed traces of developers' actions using an Eclipse IDE plug-in for seven months of software development throughout two milestones. We present and evaluate profiles of two important axes of the development process: by milestone and by application layers. MPPs can be an aid to take project control actions and help in planning future projects. Based on the experiments, project stakeholders identified several points to improve the development process. It is also acknowledged, that profiles show additional information compared to a naive diff-based approach.**

I. Introduction

On-time delivery determines the daily life of software companies, hence development productivity is a key point to research and improve. The roots of software productivity management and measurement go back to traditional industrial production processes. However, since software development is a process of always creating something new, the usual approaches to productivity measurement – like the ratio of produced uniform units per day – are not always applicable [20].

The history of software productivity measurement goes back to the '70s, and some observations of influence factors of productivity still hold – like the previous experience with programming language or amount of user participation [24]. However, productivity literature is mainly centered around the notion of productivity influence factors and is less concerned with productivity change dynamics. Several (prediction) models have been set up on these factors like development team characteristics, software reuse, technology/tools and methods, etc. For a review on these factors we refer to [20].

Most of the approaches deal with *macro-productivity*, i.e., they try to reason about the (future) characteristics of a system's development process by considering coarse granularity data available at higher levels of observation (such as projects and tasks) [13]. These approaches try to use the highest possible number of projects to reach generally applicable observations if possible [9]. On the other hand, *micro-productivity* refers to approaches where fine granularity data are collected and used from lower levels of operation in the projects (that is, elementary changes made to the system) [19]. The target of our study is in the middle of the two approaches: it is built up on fine grained productivity data to model productivity as Micro Productivity Profiles (MPP), but enables to reason about varying levels of software development productivity observation. We can use MPPs to capture the amount of *wasted effort* as the different observed amounts of work products produced (such as changed lines) on short and long term. For example, we can observe that a developer is able to modify x lines in an hour of work on average, but when totalling for a day we get y lines, where $y < 8 \cdot x$. This difference is what we call "wasted effort", because many intermediate changes will be cancelled over a longer period of time due to the inability to work perfectly. A major issue with productivity measurement is that such wasted effort may distort the measurement making it difficult to interpret.

Projects would clearly benefit from measuring (and thus controlling) wasted effort, however this is not simple. One of the challenges is the approximation of development time which is able to distinguish between types of developer activities. In our approach, we use developer interaction data for approximation of wasted effort. First, raw interaction data are collected within the IDE and then Micro Productivity Profiles are computed to aid reasoning about productivity, and hence the predicted amount of wasted effort. Profiles show additional information about productivity dynamics in software projects compared, for example, to a naive diff-based approach. For instance, using MPP-s it becomes visible what is the amount of changes after which the long-term observable productivity settles, and this could be a hint to project management about what is the "natural" development cycle characteristic of the project and team at hand.

We applied the proposed method in the field during the development of a middle-sized J2EE project to aid the project management with detailed productivity information. This paper presents the following main contributions:

- We propose (division based) micro-productivity profiles as a modification frequency based representation of

978-1-4673-7532-0/15 © 2015 IEEE

development productivity, which can be used to express productivity levels at different granularities.
- We evaluate a real-life development project where productivity is analyzed with respect to milestones and application layers.

The paper is organized as follows. We outline our motivation, introduce the subject project and the way we obtain developer interaction data in the following two sections. Productivity measurement method is detailed in Section IV. Experiment results are presented and evaluated in Section V, while we discuss the results in Section VI. Related work is presented in Section VII. Conclusions with future possibilities are outlined in the last section.

II. Motivation

The maintenance of software systems can be more and more difficult as their size and complexity increases during the development. The changing requirements can easily make hours and days of work unnecessary, because the result of many former modifications does not appear in the final version of the source code. Similarly, temporarily accepted but imperfect coding solutions cause extra coding work at a later point of development.

Our partner experienced the obstructive nature of thrown-away code during the development of a medium-sized, Java Enterprise Edition based home security system. The motivation of this paper is the investigation of a seven-month period in the development of this web application, where we measured the productivity of the work contributed by 17 developers. The management reported that parts of the specification were very inaccurate, which caused many iterations of modifications related to the same components of the system. These parts of the application are more difficult to maintain because the frequent changes and corrections decreased the quality of the source code. For the prevention of future maintenance problems, we applied productivity-based measurement and analysis to detect specific phases of the development process and parts of the system, which need more accurate specifications and design or any kind of intervention from the management. In this paper we report our experiences in measuring and analyzing productivity of the project.

III. Field Experiment

A. Subject System

We investigated the development of a medium-sized web application, which is based on Java Enterprise Edition and Seam 2.3 platforms, and it contains approximately 2200 classes and around 119k logical lines of code. The application is a part of a home security system developed by AENSys Informatics Ltd., which is responsible for the management of various security sensors installed at the end user's apartment, and handling security alerts sent by the sensors.

The development of the application was carried out iteratively with some agile elements, so the project managers wanted to see the effects of the changing requirements to the productivity of the development in some measurable way to refine the further iterations of the project. The technical leaders of the project were interested in the productivity of different application layers to see the sensitivity of the layers related to changes in the application. The architecture of the system is divided into the following five layers:

- User interface layer: it contains the implementation of composite user interface components and general, complex operations related to the user interface.
- Business logic layer: it is responsible for the management of complex business processes and transactions. This layer establishes connection between the persistence layer and the user interface.
- Integration layer: it is responsible for the communication with external systems and sensors.
- Utility classes: this layer provides general, common functionality used by many other components and layers.
- Persistence layer: it contains the entities to be managed in the system, and the high level implementation of database operations related to the entities.

B. Measured Development Phases

We investigated a seven-month period (from 3 April 2013 to 7 November 2013) in the early stage of the development. This period consisted of three main development phases:

- Phase 1 ("customer UI"): development of user interfaces for customer users. It ended with Milestone 1 on 3 June 2013.
- Phase 2 ("provider UI"): development of user interfaces for service provider users. It ended with Milestone 2 on 1 September 2013.
- Phase 3 ("Release"): development tasks related to the preparation for the first release of the application.

During the investigated period 17 developers worked on the project: 8 developers with at least 4 years of development experience, 5 developers with 2-3 years of experience and 4 junior developers with less than 2 years of experience. All developers committed their work to SVN version control system at least once a day, therefore approximately 2200 revisions were created by the developers.

Figure 1 shows an overview about the measured properties of the project. Productivity data were collected from all three phases. We identified the endpoints of each phase by an SVN revision. Unfortunately we had to ignore the last one (labelled as first release). Some developers did not use the productivity measurement tool properly, so too few events were collected from their work. Most of the data loss occurred in the third phase, which caused that we could not collect enough productivity data to analyze that phase properly.

C. Productivity Measurement Process

Our productivity measurement method relies on development data including various developer actions in IDE, file modifications and time logs. In order to accumulate important project information, detailed traces are logged in the IDE. Figure 2 depicts our productivity measurement process. In the beginning of the development process, the project manager

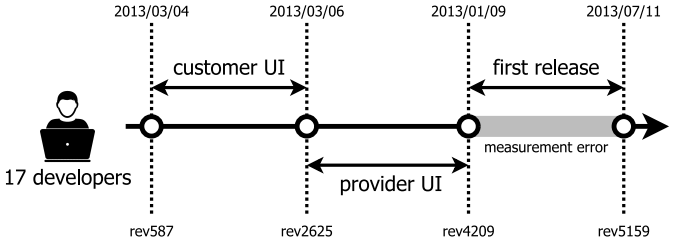


Fig. 1. Overview of the history of the measured project

defines the tasks of the project on the productivity data collector server. The developers work with the Eclipse IDE with the productivity plug-in included, which monitors the detected activities and uploads the collected events and data to the server. The developers commit their source code modifications to the SVN version control server. An internally developed productivity data analysis toolkit processes and analyzes the collected events, and calculates the real development time for files in the project. A source code analyzer toolkit analyzes the source code of revisions and compares them to each other to find modifications between them. Using these two data sets, productivity information can be calculated for the project.

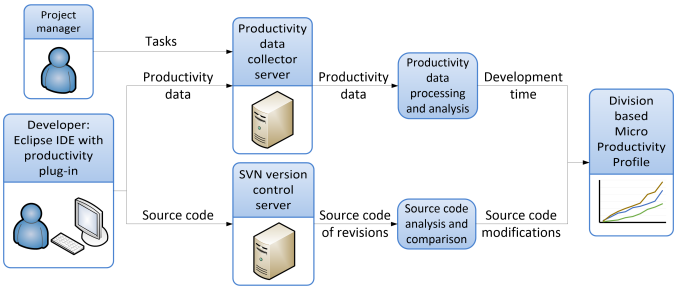


Fig. 2. Measurement architecture

During the experiment developers used the productivity measurement plug-in [1], which monitored the following types of events and characteristics of the development:

- File events (opening, closing, creating, deleting, saving, switching).
- Project events (creating, deleting, opening, closing).
- Events related to the user interface (editors, views, perspectives, dialogs, windows, etc.) in the IDE.
- Code execution events (starting, stopping, debugging, profiling).
- Code editing events (cut, copy, paste).
- Keystroke and shortcut events from the keyboard.
- Detecting idle time intervals and interruptions. After a predefined time limit is exceeded without any interactions with the IDE, an idle time interval is detected, and a special file event is raised, indicating that the actually opened file is left unchanged by the developer. After another interaction is performed with the IDE, the developer can select that he/she worked on that file or not.
- The actual task of developer. The plug-in can download a predefined list of tasks for the project, and the developer can select his/her actual task and switch between tasks. The task selector view of the productivity measurement plug-in is shown in Figure 3.
- Every time a Java source file is saved, the structure of the source file and some code quality metrics are logged.

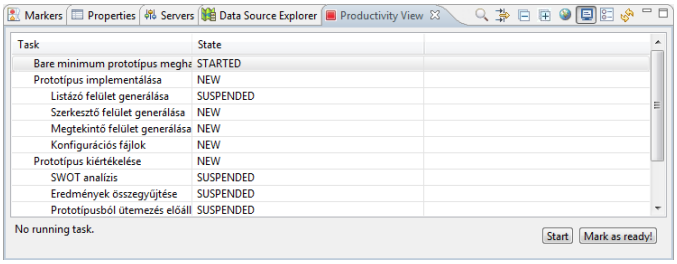


Fig. 3. Task selector view of the productivity measurement toolkit

The collected productivity data can be used to calculate the net development time of files in the project, by iterating over the file events for each developer in the ascending order of event timestamps. An example of a file event log entry in JSON format can be seen in Listing 1. If the actually examined file event is an open, switch or save event related to the file to be investigated, its timestamp is the starting point of a time interval which is relevant for the net development time of the file. The ending point of the interval is the timestamp of the next file event in the list. The net development time of the file can be calculated by adding the lengths of all of these relevant time intervals.

```
{
  "registrationDate":"2013-06-05 15:14:54.446",
  "file":"/TestProject/src/model/Person.java",
  "project":"TestProject",
  "type":"SAVE",
  "developer":"developer1"
}
```

Listing 1. Example of a file event log entry in JSON format

D. Experiment Aims

To reason about project properties and help stakeholders, raw productivity data must be processed. To analyze development productivity data and reason about wasted effort, we use the so-called Division based Micro-Productivity Profiles (MPPD for short), which will be introduced in detail in the next section. Using these productivity profiles we seek answers to the following questions:

RQ 1 What differences are indicated by productivity measurement about wasted effort in various parts of the development process? More precisely:

1) What kind of relations can be observed between the MPPD curves of various application layers?
2) Can we use the MPPD to show the changes in the productivity during various phases of development?

RQ 2 Which additional aspects of development productivity can be revealed with the aid of the MPPD against a naive diff-based approach?

IV. Research Method

In this section we give details on two key concepts of our productivity measurement method: we introduce how modification effort is expressed and present micro-productivity profiles both in informal and formal ways.

A. Measuring Developer Productivity with Modification Effort

The first crucial component of measuring the overall developer productivity is to define a comparable measure of the effort spent on various modifications. We model Modification Effort during software development as the ratio of profit (program code) and time spent to produce it [2]. To express the profit, the natural metric is the produced lines of code. Instead of counting the changed lines, we calculate profit using the count of high level modifications, like method creation or deletion. This adds an abstraction level to make difference between code constructs, which require different effort but were written in the same number of lines. Doing so this metric provides a more detailed view about the various modifications in the source code, than the CLOC tool or other the traditional metrics based on the changed lines of code [17].

```
1  class IntSet
2  {
3    protected double FindGreater( double limit )
4    {
5      for (int _i = 0; _i < Items.Count(); _i++)
6      {
7        double _current = Items[ _i ];
8        if ( _current > limit )
9        {
10         return _current;
11       }
12     }
13   }
14 }
```

Listing 2. Previous version (1)

```
1  class IntSet
2  {
3    protected int FindGreater( double limit )
4    {
5      for (int _i = 0; _i < Items.Count(); _i++)
6      {
7        int _current = Items[ _i ];
8        if ( _current > limit )
9        {
10         return _i;
11       }
12     }
13   }
14 }
```

Listing 3. Current version (2)

Modification Effort is a number to express the average amount of performed modification during a unit of time. Let us consider the following example. Code example in Listing 2 will be used to illustrate the measures for expressing programmer productivity.

The modified code in Listing 3 includes two changes over the previous version occurring in three separate lines. The first change refers to a "return type change" in line 3, while the second one is a "method implementation change" in line 7 and 10. For the purpose of illustration let us assume that it takes 8 minutes for the programmer to implement both modifications together.

Based on these values the modification effort can be calculated by taking ratio of the sum of the modification and the net development time:

$$\frac{1 \text{ return type ch.} + 1 \text{ method imp. ch.}}{8 \text{ min}} = 0.25$$

Notice that it is different from the naive method, which only count the changed lines. We choose to use modification effort, because during the implementation developers consider methods and classes as logical units and not individual lines of source code.

During our experiments the modifications of following source code entities were collected:

- Attributes, for example `int model.Person.age`
- Methods, like `int model.Person.getAge( )`
- Classes, consider the `model.Person` as an example
- Interfaces, for example `java.lang.Iterable<T>`

We count the existential (creation and deletion) and any other changes of these elements. The fully qualified name is used to identify an item. We note, that the sum can be weighted according to the type of the modifications. In this study we did not added weights to modification types to reduce the bias of inaccurate weight vector assigned as determining proper weights requires further research. In practice it means that we count any modifications considering these four entities of the source code.

B. Division based Micro-Productivity Profile

The central concept in our work is the Division based Micro-Productivity Profile (MPPD for short), which measures the frequency of changes in productivity at various granularity levels. To understand the basic concept consider the following scenario. Let us suppose we are able to measure the productivity of the developer, i.e. the ratio of produced output and required effort. The measured productivity depends on the sample size: productivity measured on the whole development considers only final program code, while measurements on weekly samples consider thrown away program code as well. Thus, repeating productivity measurement with various sample size lets us estimate the wasted effort, i.e. where developers modified the same code again. Informally, plotting these numbers as a curve is what we call productivity profile.

Figure 4 shows the history of the source code, with its revisions. We used a division based approach instead of the approach with gradually growing the sampling window [18], [19]. The figure illustrates both sampling methods: our division based method (at the bottom) and the related window based

one (on the top). The window based method uses windows with various size to swipe along the history. This let them capture the wasted effort independently from the frequency of the commits, but there are always a part of the history which can not be measured, due the window extends over the last revision.

To measure the neglected parts of the history we introduced another technique for sampling the changes. To calculate the initial value of the MPPD for the whole history (zero division points), we compared the first and last revisions of the system, i.e. the range is divided into one single part (P(0,0)) with no intermediate points. The modifications were aggregated into the Modification Effort metric. After that the algorithm moves on to the next value for one, when we take one division point in the middle of the range. It divides the history into two parts, P(0,1) and P(1,1). In this iteration we compare each division points with the subsequent ones – i.e. rev0-rev3, rev3-rev6 – and compute the Modification Effort for these pairs. The value of the MPPD is the sum of these values. As we continue with two, three, four or more divisions, the range will be cut into three, four, five or more parts and the productivity will be the sum of more and more pairs. Notice that this method depends on the frequency of the revisions, however in this particular case the distribution of the commits allows us to use it without any serious side effects.

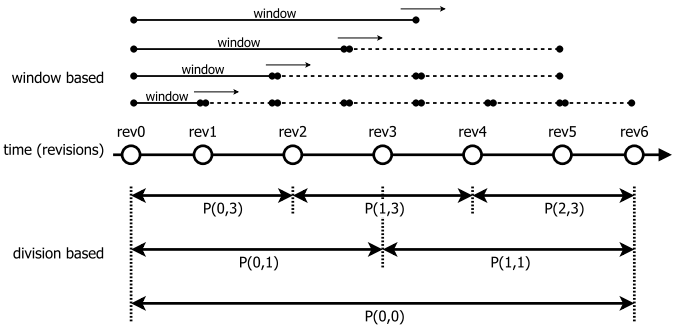


Fig. 4. Equal division of revisions

A resulting curve (Figure 5) shows the superfluous effort spent by developers during the implementation. In an ideal case these would be zero and the MPPD would be a flat line. In the real life these values are affected by the incomplete specifications and requirements which are changing over time. The steepness of the MPPD curve can be interpreted as the ratio at which the developers re-modify the same code again. Using these profiles instead the naive approach where only the most fine- and coarse-gained divisions were compared, shows not just the amount but the distribution (the frequency) of the wasted effort.

While Figure 5 illustrates the underlying concept of MPPD, Figure 6 shows a concrete example of the curve itself. The measured productivity values are represented in the right, vertical axis. As previously stated, these values increase for higher number of divisions. There are nine distinct points each for every sum of equal distance divisions parts.

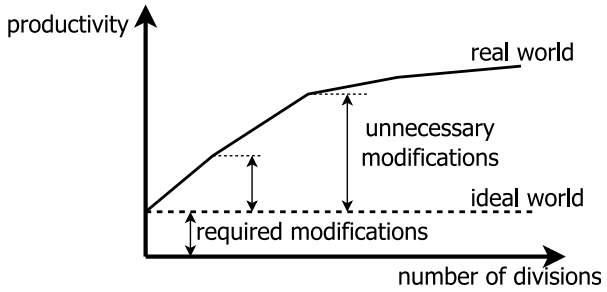


Fig. 5. The underlying concepts of MPPD

Figure 6 shows a concrete curve based on repository commits in the subject project. Besides the MPPD curve itself, the figure compares time-based and revision-based division of the project history. On the bar-chart at the top we displayed the average number of SVN additions, deletions and modifications. At the bottom part one can inspect the median and the average elapsed time between the division points. Both the number of SVN changes and elapsed times approximate a hyperbolic function as it is expected from a gradually increasing division. These facts confirm that our revision based approach provides approximately equal divisions as dividing the development phase based on elapsed time.

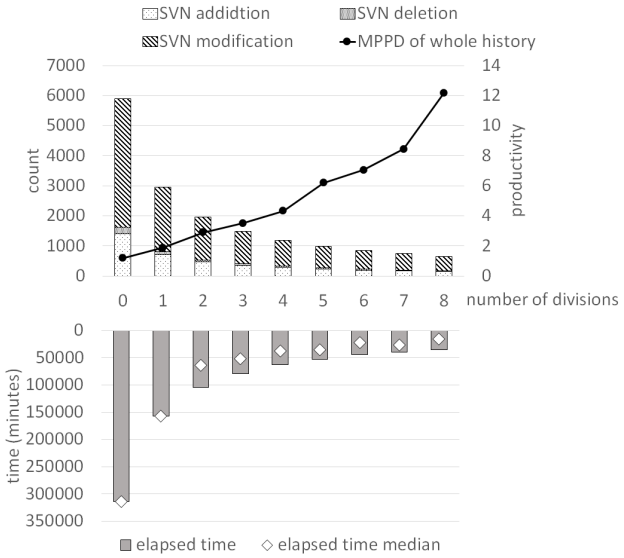


Fig. 6. Overview of MPPD over the whole history and its statistics

C. Formal Definitions

In this section we specify MPPD and the kind of data used to compute it in a more formal way.

Definition 1. *For a given software system we define* $R = \langle r_0, \ldots, r_n \rangle$ *to be the* ***ordered set of revisions*** *of the source code.*

During the experiment various modification was collected to grasp the effort spent by developers.

Definition 2. *A **modification** m is any difference between any two revisions, $m \in \text{diff}(r_i, r_j)$ where $i < j$. We assign one from a predefined set of types to each modification, based on the affected source-code element and its affected property if any, $t(m) \in T$.*

Definition 3. *$\delta_t(r_i, r_j) \in \mathbb{N}$ is the **count of modifications of type** t, between the revisions r_i, r_j. In other words $\delta_t(r_i, r_j) = |M|$ where $M \subseteq \text{diff}(r_i, r_j)$ and $m \in M, t(m) = t$. Furthermore $\Delta(r_i, r_j) \in \mathbb{N}^n$ is a vector over natural number contains the counts of all predefined modification types between the revisions r_i, r_j.*

An equal distance division was used to determine points of comparisons.

Definition 4. *We define the **equal distance divisions** for an ordered set as a list of indexes:*

$$j = \left\lfloor i \cdot \frac{n}{d+1} \right\rfloor$$

Where $n \in \mathbb{N}$ is the number of revisions, $i = 0, \ldots, d+1$ is the index of parts and $d \in \mathbb{N}$ is a predefined number of divisions. $R_i^{(d)}$ is also used to simplify further definitions, which is the i^{th} revision of the equal distance division with d dividing point.

Definition 5. *Productivity $P_i^{(d)}$ for a given equal distance division is*

$$P_i^{(d)} = \frac{\Delta(R_i^{(d)}, R_{i+1}^{(d)})}{t_{developer}}$$

where $t_{developer}$ is the net development time between $R_i^{(d)}$ and $R_{i+1}^{(d)}$.

Definition 6. *The **division based micro-productivity profile** is defined as a function over natural numbers, $\text{mppd} : \mathbb{Z} \to \mathbb{Q}$. It assigns the sum of all productivity values for a given equal distance division:*

$$\text{mppd}(x) = \sum_{i=0}^{x+1} P_i^{(x)}$$

Notice that in a perfect world the mppd is a constant function, $\text{mppd}(i) = \text{mppd}(0)$; however in real life software development it is always increasing ($\text{mppd}(i) \leq \text{mppd}(i+1)$) because of re-written code. Productivity values may incorporate wasted effort, so a higher $P_i^{(d)}$ value does not necessarily mean better overall productivity.

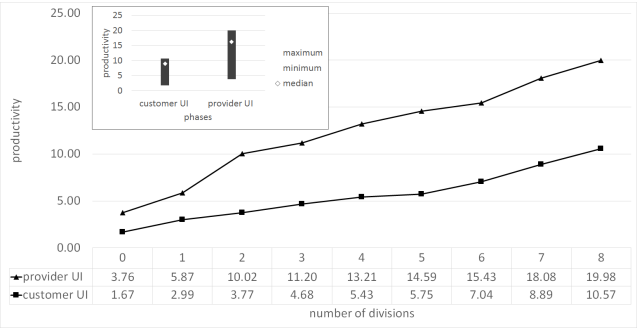
V. Evaluation

Using the measurement architecture presented above, we monitored the development activities of the developers in the presented project, and examined the productivity data of the developer team using the MPPD profiles produced by our analysis tool-chain. We calculated MPPD profiles for the following examination aspects: comparison of profiles of different development phases and different application layers, examining profiles of the developer team during the whole 7-month period of the project.

We present our findings grouped according to the research questions in the following sections.

A. RQ1 – Division based Micro-Productivity Profile for the Characterization of the Development Process

1) Productivity over Development Phases: We investigated productivity over two phases of the development. During these phases two main components were implemented: the customer user interface in the first, and the provider user interface in the second. The MPPD-s calculated for different development phases are shown in Figure 7. The collected productivity data and the calculated MPPD curves show different shapes. The developers create more modifications during the implementation of provider UI hence it has higher MPPD values. Furthermore there is a slight increase in its steepness which denotes that there are more unnecessary modifications (i.e. possibly wasted effort) during this phase than the previous one.



	0	1	2	3	4	5	6	7	8
provider UI	3.76	5.87	10.02	11.20	13.21	14.59	15.43	18.08	19.98
customer UI	1.67	2.99	3.77	4.68	5.43	5.75	7.04	8.89	10.57

Fig. 7. MPPD over development phases

These differences can be explained by the fact that there was more rigid specification for the customer UI than the provider UI as reported by the project manager. This means that the developers of provider UI had to discover the possibilities considering the technical details of the implementation. By doing this they produce more code, and change more components. They also need to adapt the existing solutions to the new requirements which results in more rewritten parts of the source code and more unnecessary modifications. In this particular case it means that managers should rearrange they resources and provide a more detailed specification for the provider UI. The slightly higher steepness of this curve shows a manageable amount of wasted effort, but we suggest that it should address to prevent the further grow.

2) Productivity over Application Layers: Figure 8 shows the MPPD-s for the development productivity of the developer team related to the five layers of the application. The MPPD of the utility layer has very high steepness, therefore differences between MPPD-s of the other four layers are not clearly visible. For this purpose, Figure 9 shows their differences without the utility layer.

The higher productivity values near the right hand side of the curve and steepness in the MPPD of the utility layer can be explained by the fact that this layer has to provide the most reusable solutions for the most general problems. Its

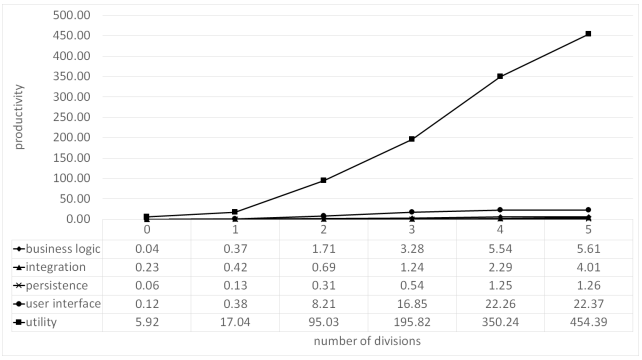


Fig. 8. MPPD over application layers (all)

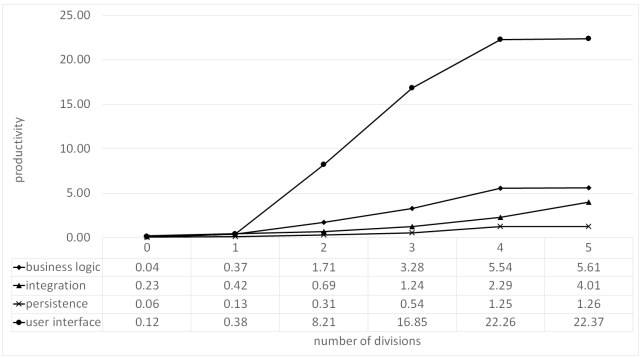


Fig. 9. MPPD over application layers (exluding utility)

components should be easily usable from any of the other layers, therefore the requirements related to the interface of this layer changes very often. This may result in frequent modifications in the source code of the layer, in addition, many changes do not appear in the final revision of the application. The developers also verified that most of the unnecessary modifications were related to utility classes. However we do not suggest that these modifications are strictly wasted effort and developers should stop writing utility classes, but they have to be aware of the nature of this layer and try to reduce the amount of rewritten code. It can be achieved by careful planning of the common functionalities and inspecting the feature specification of other layers.

The user interface layer in this context contains only the Java implementation of general, composite components and operations used by the web pages of the application. This layer also has to provide general solutions for different types of pages, and the developers also stated that several components in this layer needed many modifications to follow the changing requirements. This fact explains that the MPPD of the user interface layer has the second highest steepness.

Some slight increase can be observed in the MPPD of the business logic layer which can be originated from the changing requirements of the service provider related functions. The MPPD curves of the other two layers are quite flat, which can be verified by the facts, that the persistence layer has been well designed, and the integration layer depended only on the fixed interfaces of the external systems and sensors to be integrated. Therefore these layers did not need significant number of modifications after the implementation.

B. RQ2 – Division based Micro-Productivity Profile versus Naive Diff-based Approaches

There are several approaches to measure the unnecessary work of developers. The most simple and naive methods use some kind of historical data about the development to calculate the differences between the number of changes. For example one can measure the amount of changes of code in every single step of the implementation, then subtract the number of changes between the first and last state of the system. The underlying concepts of these type of algorithms are independent of the method of change detection.

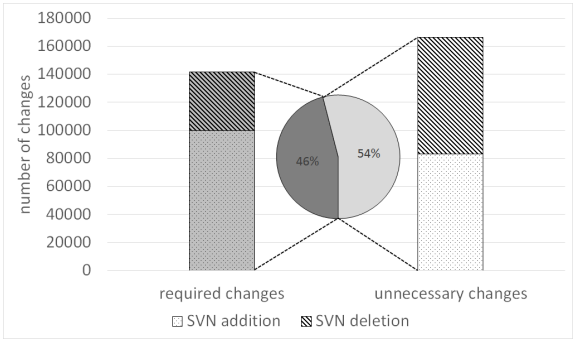


Fig. 10. Estimation of unnecessary changes from SVN logs

We implemented this naive diff-based concept using the deletions and additions from the SVN log to estimate the amount of unnecessary changes as shown in Figure 10. In practice the precision of these methods highly depends on the unit of the measurement. As stated before we use the modification effort metric to express the productivity. It has more descriptive power than the simple LOC based counterparts as reported in by Balogh et al. [2]. While these approaches are able to capture the total amount of unnecessary work, they fail to give any insights about the processes that generate these superfluous changes. Those methods, which are able to give useful help for the participants to improve the processes are more successful in practice. The MPPD curves provide details about subtle productivity changes over time. To assign precise meaning to the shape of these curves requires further analysis, but some practical suggestion can be concluded already. These hints concern mainly the development process and provide help for the managers. For example the shape of the MPPD can be used to plan the time of various activities during the project, like code reviews and milestones (see discussion below).

The main difference between the two methods is that the naive diff-based concept gives a very inaccurate approximation for the development time of the changes. The time elapsed between two commits of the same developer is necessarily much greater than the real development time of the changes by the developer. The intervals collected from SVN logs often contain parts which are not related to working time (nights,

weekends, holidays etc.). These can be approximated by daily working time of developers. But there are further problems caused by other parts of the working time, which are not related to the implementation of the software: meetings, activities related to documentation or time spent on other parallel projects etc. Our approach collects events related to interactions with the IDE to give a more accurate approximation for the real development time of the changes in the software.

VI. Discussion

A. Benefits and Drawbacks of the Approach

Considerations shown in the previous section reveal that MPPD curves provide a new viewpoint for the development process and can be used to help managers plan the monitoring of future developments. In this section we discuss benefits and drawbacks of the method and highlight future possibilities for project leaders opened by MPPD measurements.

One benefit of MPPD curves is that they show not just how much effort was spent or wasted, but the time periods when these more likely to happen, i.e. the frequency of productivity changes. To utilize this we distinguished two types of regions during the analysis of the MPPD curve in Figure 11, which represents the frequency of the productivity changes over the whole project. There are plateaus where the steepness of the function is closer to zero and ramps where it is higher. Suppose that the project managers plan to monitor the efficiency of the further development, but they have limited resources and would like to minimize the number of code review sessions. Our suggestion is to place these sessions either before or after each ramp, and do not measure quality more than once inside a plateau. If a review is placed after each ramp, the suggested frequencies can be calculated as the lengths of intervals created by the given numbers of division points at the end of the ramps. In this particular case, the whole measured period of the project was 218 days long, therefore the best frequencies of reviews are $\frac{218}{2+1} \approx 72$, $\frac{218}{5+1} \approx 36$ and $\frac{218}{8+1} \approx 24$ days. These are close to the one-month period, which means the manager needs to hold a review session once every month. The MPPD curve shows that increasing the frequency will introduce measurement where productivity value do not change significantly, i.e. measuring more than once per plateau. On the other hand, if they decrease the period there will be left over phases when the developers make many unnecessary modifications, which will be visible for the management only in a later phase of the project.

Another benefit of the productivity measurement is that the results can be used to improve the efficiency of the developer team. In the future, we would like to examine the MPPD curves of different developers, which can be used to determine whether a developer can work effectively with a given technology or on a given application layer. Therefore, developers can be educated more effectively, and they can be assigned to tasks that are the most suitable for them.

Project managers and technical leaders of AENSys Informatics Ltd. examined the MPPD curves, and they concluded that their development model should be changed in the future.

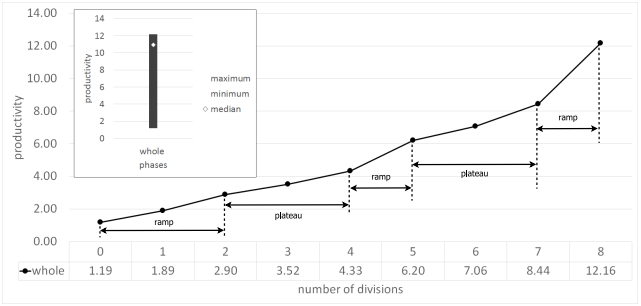


Fig. 11. MPPD over the whole range

Based on the measurement results, the new development model will have the following properties:

- A testable application prototype should be created as soon as possible in every iteration. The customer can make decisions more easily with the testing of a working application prototype, than with the examination of documents with a lot of mockups and flow charts.
- It should concentrate on the detailed elaboration of short-term tasks, and the appropriate selection of developers for the critical application layers (e.g., utility layer) to avoid unnecessary code modifications.
- It should allow the dynamic reallocation of resources to neutralize the differences between MPPD-s of different application layers.
- It should use a scheduling with one-month periods, as it was suggested earlier.

We need to discuss drawbacks of the method as well. The success of the productivity measurement depends on the active and proper use of the Eclipse plug-in by the developers. The amount of extra effort due to measurements is critical. The plug-in is designed to be seamlessly incorporated into the daily work of developers. The developers were required to maintain one extra information: the task they are actually working on. Although the task information is only a small plus, we experienced that some programmers did not use the plug-in properly, which caused a significant data loss in the third development phase. Based on the analysis of the experiences of the team after the project, the reason for improper use fo the tool was not the high amount of extra effort, but insufficient motivation by the management and also some technical issues.

Despite the careful design, interaction-based measurements in general require additional effort compared to solutions solely based on version control systems. On the other hand, the ease of information extraction may be a trap in this latter case. Rough approximations in the base data – like time estimation based on commit timestamps – may provide uncertain results. Interaction data contains more accurate and detailed information. The plug-in collects per user and per task data as well, which enables low level evaluation of the work in progress. This is invaluable when the stakeholders aim to make evidence-based decisions.

B. Lessons Learned

In the following we present a brief take away message of our experiences.

- Interaction based productivity measurement can be (almost) transparent. A primary expectation was that the experiment must not hinder the work of the developers. The IDE plugin required small extra effort and significant part of the collected data could be finally used.
- The vertical position of the profile enables the comparison of productivity and wasted effort. In this particular case project managers noticed that the curve of provider UI is always above customer UI, so they can rearrange the resources in the future.
- Identified wasted effort does not always originate from bad design or planning, but from the nature of the task at hand. As Figure 8 clearly shows, the steepness of the MPPD related to the utility application layer is much higher then the other ones. In this particular case utility classes were continuously rewritten to reflect the needs of evolving features in other layers. We found these changes much likely to be updates and not reimplementation of existing features.
- The additional time related information, which are encoded into the shape of the MPPD can be used to plan the project. Especial the frequency of the milestone and larger code review sessions can be adjusted by considering the position of plateaus and ramps, introduced in previous section and shown on Figure 11. By doing so managers have more chances to capture wasted effort with low additional overhead.

C. Threats to Validity

The presented experiment is conducted on a single project, thus it is appropriate for introducing the advantages and usefulness of the proposed method, and not for modeling productivity or for drawing general conclusions on productivity factors for other systems. Likewise, the measurement strongly relied on the fact that the investigated application was developed in Java programming language with Eclipse IDE.

As already mentioned, we experienced a significant data loss in the third development phase. To overcome this threat, we measured the ratio of data loss alongside productivity. There are two major factors of data corruption in this research. First is the improper measurement of source code modifications, the second is the missing information about the net development time, caused by improper use of the Eclipse plug-in by the developers. We are able to eliminate the first of these by using a sophisticated static code analyzer. On the other hand there was 6% of data corruption caused by the second kind. These are concentrated to the third phase of the project. This seemingly low error level render the calculated profile useless so we had to drop these data altogether from the analysis.

The used method is highly sensitive of the predefined modification types and their weights. The current measurement used the trivial unit weight function, however this may blur some aspect of the development process (e.g., addition is more complex than deletion). The modification detection component of the model was designed for object oriented languages, hence it can not be applied in the case of systems with other paradigm. However, we believe that with necessary modifications the concept can be easily adapted to other paradigms as well.

Another internal threat is the sensitivity of the MPPD curves to the homogeneity of measurement points in time. To eliminate this dependency we plan to introduce a new sampling algorithm over the history of the source code.

VII. Related Work

The area of cost and productivity estimation is constantly a frequent topic in software engineering literature. Productivity research is mainly centered around productivity influence factors. Traditional factor-based models for measurement and prediction include Putnam's SLIM, Albrecht's FP method of estimation, the COnstructive COst MOdel (COCOMO and COCOMO II) [3]. One may distinguish technical and soft factors that influence productivity [23]. We refer the interested reader to the survey of Trendowicz and Münch [20].

An effort estimation model was introduced by Mockus et al. [11], which predicts the amount and the distribution of maintenance effort over time. Several change oriented factors were included like change type, status, size, rate of size and complexity. It is found that the elapsed time collected from the version control system is not an appropriate indicator of effort. We overcome this inaccuracy by collecting net development time directly from the IDE.

Product metrics are often used for prediction of maintenance effort, such as the the well-known group of object-oriented product metrics has been proposed by Chidamber-Kemerer [4]. Soft factors like developer's expertise has already been investigated in the fault prediction area. Mockus and Weiss [10] found that change diffusion and developer's expertise were essential to predict failures. Apart from measuring productivity, it is also studied how to increase productivity of developers through motivation factors [22].

Our approach aims at micro-productivity, where fine granularity data are collected and used from lower levels of operation in the projects. Donzelli [5] used data from a real project to show that using a combination of different maintenance practices is needed to maximize maintenance performance. Junio et al. [8] applied the k-means clustering algorithm for partitioning and grouping the maintenance requests. By their PASM process the grouping of maintenance requests helps to improve productivity.

To measure the productive effort, several development monitoring tools can be used. They collect data about the actual task of the developer, time to be spent on a task, lines of code or bug details (e.g. Jasmine [15], Dashboard [14], PSP [16], Hackystat [7]). Others can be used for the monitoring of activities and interactions on the user interface of the development environment. Most of these IDE usage monitoring tools were developed for the Eclipse IDE. A usage monitoring tool was included in the Eclipse IDE itself until version 3.7, which is called Usage Data Collector (UDC) [6]. It could

capture events related to perspectives, views, menus, toolbars, and editors. The collected data were uploaded to servers hosted by the Eclipse Foundation. The Mylar Monitor [12] can also collect many types of user interface events and commands in the Eclipse IDE. The CodingSpectator plug-in can monitor the usage of refactoring commands in Eclipse, but its extended version, the CodingTracker plug-in can capture events related to files, Version Control System interactions, and application runs [21]. Our productivity plug-in combines the advantages of both types of monitoring tools, since it handles higher level tasks and provides detailed logs of developer activities.

VIII. Conclusions and Future Plans

In this paper we presented a productivity measurement method applied in an ongoing development project with 17 developers for a 7 months period. We propose *division based micro-productivity profiles*, a novel approach to model and measure productivity at fine grained level to reason about productivity at various granularity views of the project. We used micro-productivity profiles for in depth productivity analysis of project milestones and application layers. We showed the advantages of using profile curves over traditional, diff based measures that provide only one final value about the investigated phase of development.

Our experiment showed that MPPD curves are capable to encapsulate and present different aspects of the amount of productivity and its changes over the development process. Besides detecting the amount and the frequency of wasted effort, these profiles are able to show the amount of required effort for the whole project or just a specific application layer. These properties let the management fine tune the schedule of the project and reassign resources to the most sensitive tasks.

Productivity profiles could be used for analyzing other aspects of the development process as well. For example, MPPD curves can be computed for each developer working on a project, to compare their productivity. In the future, we plan to apply the proposed method on developer data to analyze per developer behavior and to investigate the effect of soft factors on productivity.

References

[1] Gábor Antal, Ádám Zoltán Végh, and Vilmos Bilicki. A methodology for measuring software development productivity using Eclipse IDE. In *Proceedings of the 9th International Conference on Applied Informatics (ICAI 2014)*, 2015. Accepted.

[2] Gergő Balogh, Ádám Zoltán Végh, and Árpád Beszédes. Prediction of Software Development Modification Effort Enhanced by a Genetic Algorithm. *SSBSE Fast Abstrackt track*, pages 1–6, 2012.

[3] Barry W Boehm. *Software Engineering Economics*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[4] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[5] Paolo Donzelli. Tailoring the software maintenance process to better support complex systems evolution projects. *Journal of Software Maintenance*, 15(1):27–40, January 2003.

[6] Eclipse Usage Data Collector. http://www.eclipse.org/org/usagedata/. Accessed: 2015-03-30.

[7] P.M. Johnson, Hongbing Kou, J. Agustin, C. Chan, C. Moore, J. Miglani, Shenyan Zhen, and W.E.J. Doane. Beyond the personal software process: Metrics collection and analysis for the differently disciplined. In *25th International Conference on Software Engineering (ICSE 2003)*, pages 641–646, May 2003.

[8] Gladston Aparecido Junio, Marcelo Nassau Malta, Humberto de Almeida Mossri, Humberto T Marques-Neto, and Marco Tulio Valente. On the Benefits of Planning and Grouping Software Maintenance Requests. In *15th European Conference on Software Maintenance and Reengineering (CSMR 2011)*, CSMR '11, pages 55–64, Washington, DC, USA, 2011. IEEE Computer Society.

[9] Robert Lagerström, LivMarcks von Würtemberg, Hannes Holm, and Oscar Luczak. Identifying factors affecting software development cost and productivity. *Software Quality Journal*, 20(2):395–417, 2012.

[10] A Mockus and D M Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, 2000.

[11] Audris Mockus, David M Weiss, and Ping Zhang. Understanding and Predicting Effort in Software Projects. In *In 2003 International Conference on Software Engineering*, pages 274–284. ACM Press, 2002.

[12] G.C. Murphy, M. Kersten, and L. Findlater. How are Java software developers using the Eclipse IDE? *Software, IEEE*, 23(4):76–83, July 2006.

[13] R. Premraj, M. Shepperd, B. Kitchenham, and P. Forselius. An empirical analysis of software productivity over time. In *Software Metrics, 2005. 11th IEEE International Symposium*, pages 10 pp.–37, Sept 2005.

[14] Process Dashboard homepage. http://www.processdash.com/, 2015.

[15] Hyunil Shin, Ho-Jin Choi, and Jongmoon Baik. Jasmine: a PSP supporting tool. In *Proceedings of the 2007 international conference on Software process*, ICSP'07, pages 73–83, Berlin, Heidelberg, 2007. Springer-Verlag.

[16] Raymund Sison, David Diaz, Eliska Lam, Dennis Navarro, and Jessica Navarro. Personal Software Process (PSP) Assistant. In *Proceedings of the 12th Asia-Pacific Software Engineering Conference*, pages 687–696, Washington, DC, USA, 2005. IEEE Computer Society.

[17] Gabriella Tóth, Ádám Zoltán Végh, Árpád Beszédes, and Tibor Gyimóthy. Adding Process Metrics to Enhance Modification Complexity Prediction. In *Proceedings of the 19th IEEE International Conference on Program Comprehension (ICPC'11)*, pages 201–204. Ieee, June 2011.

[18] Gabriella Tóth, Ádám Zoltán Végh, Árpád Beszédes, Lajos Schrettner, Tamás Gergely, and Tibor Gyimóthy. Adjusting Effort Estimation Using Micro-Productivity Profiles. In *12th Symposium on Programming Languages and Software Tools (SPLST'11)*, pages 207–218, 2011.

[19] Gabriella Tóth, Ádám Zoltán Végh, Árpád Beszédes, Lajos Schrettner, Tamás Gergely, and Tibor Gyimóthy. Adjusting effort estimation using Micro-Productivity Profiles. *Proceedings of the Estonian Academy of Sciences*, 62(1):71–80, 2013.

[20] Adam Trendowicz and Jürgen Münch. Chapter 6: Factors Influencing Software Development Productivity – State-of-the-Art and Industrial Experiences. In *Social networking and the web*, volume 77 of *Advances in Computers*, pages 185 – 241. Elsevier, 2009.

[21] Mohsen Vakilian, Nicholas Chen, Stas Negara, Balaji Ambresh Rajkumar, Roshanak Zilouchian Moghaddam, and Ralph E. Johnson. The need for richer refactoring usage data. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU '11, pages 31–38, New York, NY, USA, 2011. ACM.

[22] J.M. Verner, M.A. Babar, N. Cerpa, T. Hall, and S. Beecham. Factors that motivate software engineering teams: A four country empirical study. *Journal of Systems and Software*, 92(0):115 – 127, 2014.

[23] Stefan Wagner and Melanie Ruhe. A systematic review of productivity factors in software development. Technical report, Technische Universität München, 2008.

[24] C. E. Walston and C. P. Felix. A method of programming measurement and estimation. *IBM Syst. J.*, 16(1):54–73, March 1977.