How Much Integrated Development Environments (IDEs) Improve Productivity?

Iyad Zayour, PhD
Computer Science
Lebanese University, Lebanon
izayour@ul.edu.lb

Hassan Hajjdiab
College of Engineering and Computer Science
Abu Dhabi University, Abu Dhabi
hassan.hajjdiab@adu.ac.ae

***Abstract*—Modern integrated development environments (IDE) such as Microsoft Visual Studio and Eclipse, and languages, such as Java and .Net, represent a far step ahead from the legacy development environments such as grep and Emacs. One would assume that the productivity of programmers has been dramatically improved. To get a more realistic assessment of improvements, after more than ten years on an investigation on programmer productivity, we recently embarked into an investigation of a group of programmers in a software company using all the modern features of Visual Studio. We found that indeed there were significant improvements. But, at the same time, other side effects made the overall improvement not so clear. The complexity of the development environment and its associated libraries and ready-made components represent a significant new source of loss of productivity that manifests the most in the cost of debugging and learning. A better understanding of the challenges associated with adopting new development technologies may rescue some of the gain in productivity.**

***Index Terms*—IDE, Productivity, empirical software engineering.**

I. Introduction

More than ten years ago, we embarked into an empirical investigation to help a telecommunication company reduce the cost of maintaining a large legacy software [1]. The development environment for that software was based on classical tools such as grep and Emacs and a Pascal like proprietary language. Today, with modern IDE tools such as Microsoft Visual Studio and Eclipse, and languages such as Java and .Net, one would assume that the productivity of programmers has been dramatically improved.

To get a more realistic assessment of improvements, we recently embarked into a similar investigation of a group of programmers in a software company using all the modern features of Visual Studio 2010 to develop a hospital information system. It is important to note that by IDE we do not reference only to the software system that integrates the text editor, the compiler and other tools, but to the whole environment that includes the language and its massive libraries and ready-made components like UI controls.

A. Back then

In our original investigation of the legacy system, the goal was to improve the productivity of programmers particularly those doing software maintenance, as this activity was considered to be highly inefficient by the target company. It was our conjecture that by studying and observing programmers doing software maintenance, we can identify the "inefficient" tasks. And then, we reduce inefficiency by developing a tool that addresses these tasks thus hoping to improve the overall productivity of the programmers. Among other things, the study revealed that programmers spend significant time doing keyword searches (see Table 1 for the list of most common tasks).

A more profound analysis, however, showed that the search problem was only a facade for a more complex and fundamental task: program comprehension. We can show this by exploring the typical scenario for solving a maintenance request. In order to solve the request, a programmer must:

1. Locate the code related to the maintenance request that is often described in term of program behavior description. The programmers must search for any clue (user interface strings and routine names) that relates the external program behavior to the code relevant to the maintenance problem.
2. Understand the code that generates the behavior so changes can be done. This code, however, is not typically stored in one contiguous passage of code, but rather in delocalized pieces of code that only gets connected during run time by branching and routine calls. This necessitate:
    a. Locating these delocalized pieces often by doing more searches.
    b. Understanding the code dynamics at run time by creating a mental simulation of execution synchronizing program

doi:10.4304/jsw.8.10.2425-2431

events with code to find what went wrong.

The essence of difficulties in such scenario does not reside in the effectiveness of searching mechanism, but rather in the inherent complexity of the code space that need to be understood. The mental reconstruction of execution paths can overload human cognitive abilities particularly in large systems with deeply nested relations such as routine call trees that often reach more than ten levels of nesting.

TABLE 1

TOP 5 ACTIVITIES IDENTIFIED DURING SOFTWARE MAINTENANCE OF THE LEGACY SYSTEM

1.1	Finding the next piece of code in a mentally simulated control flow (e.g. call flow/ call tree)
1.2	Finding the definition of a routine
1.3	Finding a starting point in code
1.4	Finding where an external behaviour is implemented in code
1.5	Finding the definition of a variable or a variable type

In short, although on the surface programmers spent significant time doing searching, yet improving this mechanism only removed "accidental" difficulties [9]. The "essential" difficulties faced by the programmers were due to the complexity of program comprehension.

B. More than 10 Years Later

With modern IDEs and modern programming languages, one can trace little similarity, at the surface, to the development experience of the legacy system. Yet, our thesis remains that, at a more profound level, the fundamental activities such as program comprehension tasks should be comparable.

To report on what really changed, what has been solved and improved and what has not, we investigated a company developing a hospital information system of medium size (about 320 forms/screens). The team developing this system is composed of 13 programmers of varying expertise levels, from six months to four years of experience in the development environment (IDE, languages, and class libraries). Activities included both developing new features and maintaining existing ones.

The programmers use all modern visual features of Visual Studio 2010, such as forms, reports, and database schema design. We were "embedded" with the development team repeating the same techniques used to investigate the legacy system such as shadowing, observing and interviewing the team. The study spanned three months in total.

II. RELATED WORK

Brooks [9] in his classical paper "No silver bullet" argues against high expectation from new technologies: "there is no single development, in either technology or in management technique that by itself promises even one order of magnitude improvement in productivity, in reliability, in simplicity."

The difficulties in program comprehension by programmers have been studied for long, and different cognitive models for program comprehension have been suggested. Von Mayrhauser and Vans surveyed this area in [3] and compared six cognitive models

The empirical study of the programmers work practices was one step ahead after cognitive models that mostly used controlled experiments. It seeks to understand how work occurs and suggest appropriate technologies for the workplace [Singer 97]. This can be done by "following and recording the work that people do" in their industrial setting.

More recently Latoza et. al. [6] used a series of surveys and interviews to capture the work habits of programmers. They found that programmers spend about half their time debugging, and the rest is spent writing new features and making their code maintainable.

The study of the effect of modern IDE on programming has not been so frequent. Ko et.al [5] claim to be the first to do so. They conducted an "in vitro" experiment designed to assess the impact of modern IDEs (Eclipse) on maintenance tasks. They found that programmers spent most of their time "reading and navigating" code. It wasn't clear how their experiment captured the effect of IDE given that predefined tasks were given to the subjects. We argue that controlled experiments are incapable of capturing the diversity of the development ecosystem.

Murphy et. al. [13] used a tool that generates traces representing the interaction history of programmers with IDE to study how Java programmers use the Eclipse IDE. The data collected represents the frequency of use in general of the different features, views and commands of Eclipse but without relation to the mental model or the high level intention of use by the programmers. They found, for example, that copy and paste are among the 10 top most used commands. We, on the other hand, ignored copy and paste as we considered them too low level perhaps at the same level with typing (programmers may copy and paste to save typing) and has no relevance to the essential difficulties experienced by programmers.

Recently, the "Debugging Canvas" [10], an add-on to Visual Studio that extends its debugger and provides visual representations of call relations among other dynamic information, represents a serious attempt to address the program comprehension difficulties during programming. So far, it is still a research tool and did not make it into official release of Visual Studio.

III. CHALLENGES OF THE STUDY

Like in any empirical study, we wanted to measure and compare specific variables, something that can be best done in a lab-like setting under controlled experiments. But a software development environment is more like an ecosystem with an infinite number of interdependent variables. The tools, the expertise and skill of programmers, the nature of projects, the development

process and many more variables, all affect the productivity of programmers.

A lab-like ("in vitro") experiment is too limited to capture the versatility of the development ecosystem. We advocate an "in vivo" approach where studies are performed in the native settings of the programmers and where observations are matured with prolonged observations and involvements that get synthesized into general high level conclusions.

Our approach falls under the ecological study of programmers, a term introduced by Shneiderman and Carroll [2]. It is a paradigm in empirical studies where the emphasis is to confront realistic software situations on their own terms where "researchers are direct participants in the definition and creation of new software artifacts."

Shneiderman and Carroll argue that such a realistic approach is more suitable to generating good results compared to research based on experiments, "because software design takes place in software shops not in psychological laboratories." Ecological studies must be "imperative, inductive, they seek to discover, not merely to confirm and disconfirm".

When one looks at a realistic environment in which software is developed, the human and social factors may be more important than the technological factors. In fact, we are looking at how new development technology changes the "life" of programmers during programming; in particular, how much it solved their difficulties and affected their productivity. The notion of difficulty is a human one; it is a phenomenon that is experienced in the brain of the beholder. In many cases, however, difficulties are not faced individually, but a rich social interaction within the development organization mainly to share expertise is involved [12].

We argue that the research approach most suitable to the study the work of programmers is more of qualitative research approaches like what is dominant in social sciences. In particular, our approach resembles what is known in social sciences as "participant observation" whose aim is "to gain a close and intimate familiarity with a given group of individuals and their practices through an intensive involvement with people in their cultural environment, usually over an extended period of time" [8].

IV. Details of the Study

The study spanned two phases. In the first, one system has been studied profoundly. In the second phase, we focused our efforts on investigating specific situations, based on the conclusions drawn from the first phase, in different projects hosted in the same target company.

A. Phase 1: System observed

The system observed in the first phase was a hospital information system (HIS) developed using 3-tiers architecture:

1. The user interface tier uses windows-form and the construction of these forms was mostly done using interactive visual tools that permits drag and drop, positioning and resizing of controls almost entirely visually using the mouse.
2. The business logic tier was developed using the C# .Net language and it was highly object oriented. The classes used, however, were mostly mapped to tables in the database. Class design tools were used to update the class graph visually.
3. The database was an SQL server, and SQL statements and stored procedures were used in the data layer to move data back and forth to the business layer. A visual database design tool was used to create and update the database schema.

B. Phase 1: Data collection

We started our study by embedding into the development team; we asked to be treated as trainees. The goal was to avoid any factors that alter the "normal" behavior of programmers so to increase the "ecological validity" [4].

Like in the legacy study, we used the technique of shadowing for studying how programmers spend their time in order to identify inefficient tasks. Shadowing involves sitting next to a programmer while actively working and recording his activities. Interruptions such as meeting and phone calls were excluded from recording unless they were related to the programming tasks such as asking a peer on how some functionality is implemented. We observed and recorded our observations on papers with time stamps. We did not intervene in the programming work as long as we understood what the programmer was doing and only asked short questions when we don't.

The familiarity of the authors with the IDE was helpful in securing fewer interruptions for the programmers. By the end of each day, we analyzed our logs looking for the proper of granularity of tasks to observe and encode, some tasks were classified as too low level (e.g. cut and paste) and were merged with higher order tasks and other tasks needed to be broken in more details tasks. The observable task in this study was meant to match a specific feature of the IDE or what the programmer most commonly answer when asked "what are you doing now?". We focused on identifying the most frequent tasks used by programmers that can affect productivity thus excluding writing code and creating new artifacts for new features, as it is a common conception to associate productivity with creating new artifacts.

After 15 days of observation we charted a rough map of the modern programmer landscape, we narrowed down our focus to significant activities, i.e. the activities that consume more than 5 % of working time. It was our choice to keep the measurement of the activities at this coarse level as we believe that this uncharted area requires more exploratory approach focusing on the big picture rather on the details and their statistical significance.

C. Phase 1: Results

Table 2 summarizes our finding that represents the top 5 most common tasks performed by programmers using IDE by order of frequency of usage.

TABLE 2

THE TOP 5 MOST USED FEATURES OF THE IDE

2.1	Intellisense
2.2	Navigating forms and controls to associated code
2.3	Go to definition / Navigate backward
2.4	Keyword search for code
2.5	Inserting and running to breakpoint, running and watching variable values

Task 2.1: Intellisense was very instrumental feature of IDE that was highly appreciated by programmers. Intellisense helps programmers find the proper naming of variable by suggesting name in auto-complete fashion. The suggestion of method parameters was also very convenient as it contains the proper order, names and types of the parameters. Most importantly, by providing visual cues of correct use of symbol names (e.g. variable and method names), it provides immediate feedback of syntax correctness.

Task 2.2: Very frequently, programmers find relevant code by finding a form, then locating a UI control on that form like a button, then clicking on this control to reveal associated code like the code that should run when that button is pressed.

Task 2.3: "Go to definition" IDE feature is used to bring the definition of a symbol (e.g. method, class and variable). Going to the definition of method was the most used. After getting to the definition, it was frequently followed going to another definition recursively (following a call path). These actions were almost always followed by a sequence of pressing the "back" navigation buttons to return to the original position.

Task 2.4: key word search is used to search for specific snippet of code for example to search for the text of an error message or part of a symbol name.

Task 2.5: Debugging with IDE is a frequent activity. Very often, programmers run debugging sessions that involve: setting a breakpoint, stepping after that breakpoint while watching the value of some variables and the result of the execution on the user interface.

D. Phase 2: Comparison

Compared with the legacy study, there were clear improvements at the accidental level of difficulties like with Intellisense (task 2.1) of the IDE that almost removed any loss of productivity due to syntax errors.

The search problem is highly reduced. Finding a starting point to locate code related to behavior (task 1.3) became much easier as the programmers use the UI artifacts to find relevant code (task 2.2.). In the legacy system everything was stored in code including the UI. With Visual Studio, the UI exists in separate visual artifacts (e.g. forms and reports) that can be designed, saved and viewed with little coding. These visual artifacts help in localizing relevant code as snippets of code can be associated with parts these artifacts. For example, instead of searching for what code executes when a button is pressed, now a programmer can find the form and double click the button to reveal the code that runs when the button is clicked (task 2.2).

The search to find delocalized piece of code (task 1.1, 1.2 and 1.5) is also largely replaced by semantics aware logical link navigation. Namely, the "Go to definition" (task 2.3) operation that brings the definition of a routine or a variable is frequently and efficiently used when comprehending the code and following control flow.

E. Phase 2: Settings

Phase 1 data and analysis shows that many of the accidental difficulties of the legacy environment have been well addressed, but what about the essential one like the difficulties of program comprehension?

The target company that we studied was a consulting company where the hospital Information system (HIS) was only one of many projects it was implementing. So, to evaluate the state of productivity under IDE in a holistic way, we conducted a series of semi-structured interviews with programmers focusing on their perception of productivity. The answers revealed a perception of non-consistent level of productivity: in most of the time productivity is acceptable but in many situations there are dramatic drops of productivity enough to reach "crisis" level.

So we decided to repeat our observations into a more exploratory form, looking into more projects and interviewing more peoples. Like in the legacy study, we looked for situations that were perceived as of low productivity.

Interviews and shallow observations helped us identify 17 programmers who perceived themselves in low productivity sessions. These programmers were working in 4 different projects all using Visual Studio 2010, all projects were 3-tiers projects but with two of them using ASP web interfaces instead of windows forms (desktop interfaces).

F. Phase 2: Results

We analyzed the common factors between the "low productivity" situations and we found that the existence one the following factors make it very likely for productivity to drop. The more factors exist in a combination, the worse the productivity gets until it reach "crisis" situation:

1. Experience in the development environment: as experience decreases productivity drops even faster. Programmers with less than 8 months of experience were perceived as having negative productivity: they take time from peers asking for help more than they produce. Even experienced programmers face a steep drop in productivity when they step outside their area of expertise. In one instance, controlling a camera by the software (for dermatologists) required 4 days of searching, learning, testing, and

debugging from an experienced programmer despite that it was supported by a COM component.

2. Stage of the project: projects starts with high productivity, UI screens, classes and database tables can be created very quickly using visual tool. But as the project grows in size and take shape enough to permit real data testing, software changes become the dominant task creating bugs that are hard to fix.
3. Complexity of software: The parts of software that are more complex coincide with significant drop in productivity. In particular we noted that it is the complexity of the business logic layers that mattes the most, the UI and data layers tends to have fixed contribution in the overall complexity.
4. Familiarity of code: The programmer who has to work on unfamiliar code tends to have a lower productivity.

During the "low-productivity" situations, programmers frequently switch between 3 moods: writing code, debugging and learning. All the moods were interwoven and can hardly be distinguished by an observer who is not familiar with the metal model of the programmer. Programmers, while actively programming, frequently pause and switch to learning. They may equally switch to "debugging/testing mode" to try and test what they wrote as it is convenient to do so with IDE, and when the unexpected remains, the test is repeated in debugging mode. A very common scenario is to: write, test, debug, learn, then write, test and debug for several iterations. Yet, the percentage of time spent on debugging and learning activities highly outweighs active programming.

Table 3, shows the top most 5 used tasks during "low productivity" situations. Note that these tasks are ranked not only by frequency of use but also by the total time spent on them.

TABLE 3

THE TOP 5 MOST USED IN LOW PRODUCTIVITY SITUATIONS

3.1	Stepping in debugging mode and watching data structures
3.2	Searching and following relation for finding the code relevant to a bug
3.3	Testing a fix by running the program and entering data
3.4	Searching the internet, the code base and asking peers for possible solutions to problems/bugs
3.5	Reading help and others to learn about the programming language features

G. Comparison & Analysis

Learning and searching for information about the development environment (tasks 3.4 , 3.5) is a new activity compared to the legacy system where the language and tools were totally mastered by programmers who rarely needed to learn anything about their environments. With Visual Studio, significant time is used for searching the Internet and local help and reading the found material. It is important to note that the costs of "learning while working" should also include the cost of programming in partial mastery of their environment as this "partial mastery" can result in subtle bugs, as we observed.

Debugging (tasks 3.1, 3.2, 3.3) can consume up to 80 percent of programmer time in certain situations. While it is hard to put an accurate general quantitative figure on debugging as it is hard to separate debugging from learning (learning by experimenting) and programming itself, what is very clear, is the level of programmers discontent vis-à-vis the cost of debugging.

Many bugs that may seem simple at first, show high resistance to resolution consuming significant time. Programmers report that, frequently, solutions for bugs are only found on the Internet in developer communities; such solutions or workarounds are adopted without in-depth understanding of problem causes. In many situations, it may be difficult to find solutions based on personal effort and persistence; there were not enough information or leads to pursue or hypotheses to test.

It seems that the essential difficulties identified in the legacy study persisted where program comprehension and in particular mapping code to program behavior (and vice versa) was a main cause of difficulties. The difficulties in debugging are directly related to the difficulties in program comprehension. A bug is, in fact, a failure of mapping code to behavior; that is, the programmer expected some behavior from the code he wrote but in actuality a different behavior is produced. Debugging, thus, is the attempt to perform the change in code to reproduce the intended behavior. The mapping requires following the code in the order that they will be execute dynamically. This is a core requirement in both studies. So how IDE performed compared to the legacy environment for this particular task?

As improvements we note:

1. The IDE facilitates navigation between different delocalised pieces of code that falls on the dynamic path (task 2.2.).
2. Breakpoint, stepping and watching variables facilitate this mapping between the code as they executed and the program behaviour. This replaces the probes (print statements to reveals the value of variables or to confirm the actual passage of execution over a particular point in code) used in the legacy environment that was less convenient to use and much more time consuming.

On the negative side, we note that the cost of debugging is increasing in proportion to the overall working effort, why simple bugs are harder to solve? Two explanations were commons to most of the answers we heard in interviews and the cases we observed:

1. Insufficient leads: Error messages that represent the only hints as to what went wrong inside the ready-made components are often too general and sometimes misleading. Pinpointing the source of errors can become very challenging; it is even sometimes hard to find the file that produced the error. We observed many cases of errors that turned to be caused by different causes than what the error message suggested. They actually diverted the programmer from the real cause of the problem and its location. Prolonged sessions of trial and error, testing different hypotheses and searching on the Internet for similar problems were needed to eventually solve the problem.
2. Obsolete debugging tools: The classical debugging tools (breakpoint, variable watch, stepping, call stack etc…), that still represent the only window of visibility into the dynamic domain of a program, provide a too-small window given the increased complexity of dynamic events. It is "like fixing a car engine while only looking through a straw", a programmer commented. Call stacks, for example, used to trace the call flow leading to the execution error are becoming less useful. Now a call stack is stuffed with calls that the programmer did not explicitly invite and sometimes they do not provide a way of inspecting their internals (see figure 1).

V. Overall Study Analysis

Modern IDEs successfully attack many of the accidental difficulties such as those related to syntax errors and searches. But when it comes to the most essential difficulty in software – program comprehension—they do not score as well. The complexity of the environment created additional burdens that manifest the most as increase in the cost of debugging.

While few questions the gain in productivity between primitive tools like VI editor to more sophisticated tool like Emacs, for example, the continuous increase in size and complexity of IDEs and in their massive libraries pose a question about the economy of scale: when bigger development systems stop to give better productivity?

In this study, we did not find that the gain in productivity in using Visual Studio is not so obvious. The gain in the initial phases of the project is offset by loss of productivity in the later phases when the cost of debugging taxes the productivity. This research contribution is to highlight the existence of serious side effects: the debugging and learning cost. We found that the use of more and more complex IDE is should not be taken for granted as a source of increased productivity. Shiny new technologies can unfold malicious demons and invincible bugs that poison the health of development and create prolonged malaise.

The frequency of new technologies arrival with visual Studio releasing a new version each two years sometimes with dramatic changes makes another question relevant: when it is most efficient to jump to a new technology? With more and more complex environments, the period to reach the status of being "expert" has been significantly prolonged; relinquishing existing experience to jump to a new technology may result in significant loss of productivity. Sometimes, it can turn to be an unforgivable mistake; anecdotal evidence from the company we studied suggested that groups that moved from Visual Basic 6 to Visual Studio 2002 faced disastrous consequences. The dramatic changes between the environments invalidated most of the prior existing experience base and visual studio 2002 was largely immature technology.

VI. Conclusion & Future Work

In this study, we showed that although IDE increased productivity at many accidental levels, the complexity of the environment created additional cost particularly at the level of debugging and learning. While in the long run there is no option but to shift to new technologies, this research raised the questions about how soon this shift should be done and to what level more sophisticated technologies would still yield better productivity given their side effects in debugging and learning.

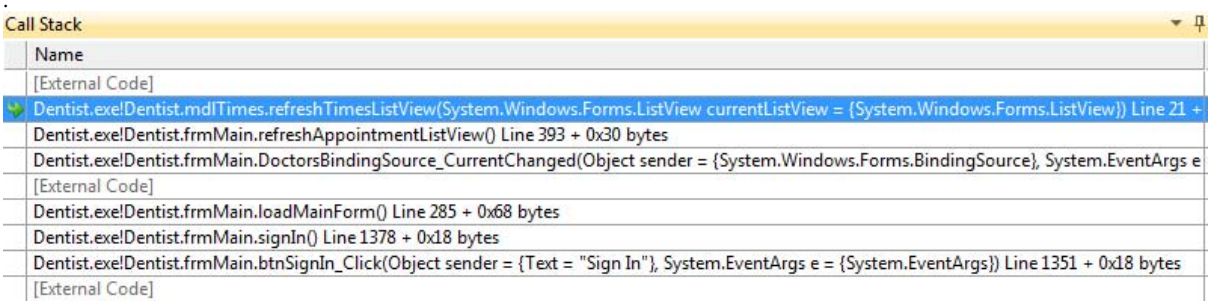

Figure 1: a snapshot of call stack in Visual Studio, note the "External Code" dimmed entries and the CurrentChanged that is a handler triggered indirectly by a BindingSource component

This qualitative and exploratory nature of this study achieved what qualitative studies aim at: to delve into the complexity of the problem rather than abstract it away and to produce rich and informative results [11]. Yet most of the results of this study remain in the theory and hypotheses generation phase, although grounded in data, much more empirical work is needed to establish these hypotheses to become more generalizable. Something we are planning for the future.

REFERENCES

[1] Zayour I., "Reverse Engineering: A Cognitive Approach, a Case Study and a Tool. Ph.D. dissertation", University of Ottawa, http://www.site.uottawa.ca/~tcl/gradtheses/, 2002

[2] Shneiderman B. and Carroll J., "Ecological studies of professional programmers", *Communication of the ACM*, Nov. (1988)

[3] Vonmayrhauser, A., & Vans, A.M. (1995*a*). Program comprehension during software maintenance and evolution. *Computer*, 28, 44-55.

[4] Halverson, C, et. Al. "Towards an Ecologically Valid Study of Programmer Behavior for Scientific Computing", SE-CSE 2008.

[5] Ko, A., et al., "Eliciting Design Requirements for Maintenance-Oriented IDEs – a Detailed Study of Corrective and Perfective Maintenance Tasks," in *Proc. ICSE'05*.

[6] LaToza, T. et al." Maintaining mental models: a study of developer work habits," Experience report in ICSE 2006.

[7] Singer, J., Lethbridge, T., Vinson, N., and Anquetil, N., "An Examination of Software Engineering Work Practices," in *Proc. CASCON'97*.

[8] http://en.wikipedia.org/wiki/Participant_observation

[9] F. Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering", *IEEE Computer*, vol. 20, 4 April (1987)

[10] Robert DeLine, Andrew Bragdon, Kael Rowan, Jens Jacobsen, Steven P. Reiss, "Debugger Canvas: Industrial experience with the code bubbles paradigm", ICSE 2012: 1064-1073

[11] Carolyn Seaman, "Qualitative Methods in Empirical Studies of Software Enginnering", *IEEE Transactions on Software Engineering*, Vol. 25, No. 4, July 1999.

[12] Gina D. Venolia, Robert DeLine, and Thomas LaToza, 'Software Development at Microsoft Observed: It's about people … working together', no. MSR-TR-2005-140, October 2005.

[13] Gail C. Murphy, Mik Kersten, and Leah Findlater , "How Are Java Software Developers Using the Eclipse IDE?", *IEEE Software*, July (2006).