Tracking Use of the Eclipse Integrated Development Environment

Jamie Samdal
jsamdal@email.arizona.edu

I. Introduction

The primary goal of this research project was to develop a tool that would enable the analysis and further understanding of software processes, development environments, and their use by software developers. The secondary goal of this research project was to then use this tool in a proof of concept scientific experiment related to the area of computer science, in order to encourage empirical and repeatable scientific research projects in the future. The first goal was met, but the second goal was only partially met – this paper contains a set of detailed example research projects that could be completed with the tool developed and the benefits that they would provide each situation.

II. Motivation

There are a number of motivations behind this research project. There is a significant lack of experimental testing in the area of computer science. This is partially due to the belief that experimental testing is expensive and time consuming. By developing a tool that can be easily used and provides little to no additional overhead to normal working procedures, data can be collected easily. If the specific needs of a project are taken into account ahead of time, this data can be even more easily analyzed at the conclusion of its collection. Specifically in the area of software processes, it is doubtful that anyone is going to try to present a software process without at first implementing it and using it in a few projects, so this experimentation is going to be done at least on some small scale. However, many such experiments only collect subjective data in the form of surveys from their experimenters. The kind of tool that I have designed will collect objective, empirical data that can actually quantify the time spent on different phases of a project. Admittedly, this is a prototype tool, so it does not attempt to do everything. However, with more specific goals and a marginal amount of time and effort, this tool can be expanded to track data associated with various phases in a software process and provide comparisons to other processes.

One practical motivation for this project is that I am working on a parallel research project that is designing a new, global software process. Tools similar to these will be invaluable to identifying which types of projects or situations that this process works well for, determining which pieces of the process are more or less effective than current methods, and quantifying the time and/or cost savings associated with the use of this process. In particular, I am developing what is essentially a view in Eclipse that will manage project communication. My tool will be able to identify exactly how much time a software developer is using for communication, which is a key overhead in multi-site software projects. Traditionally, this kind of data has been collected in surveys or via estimations.

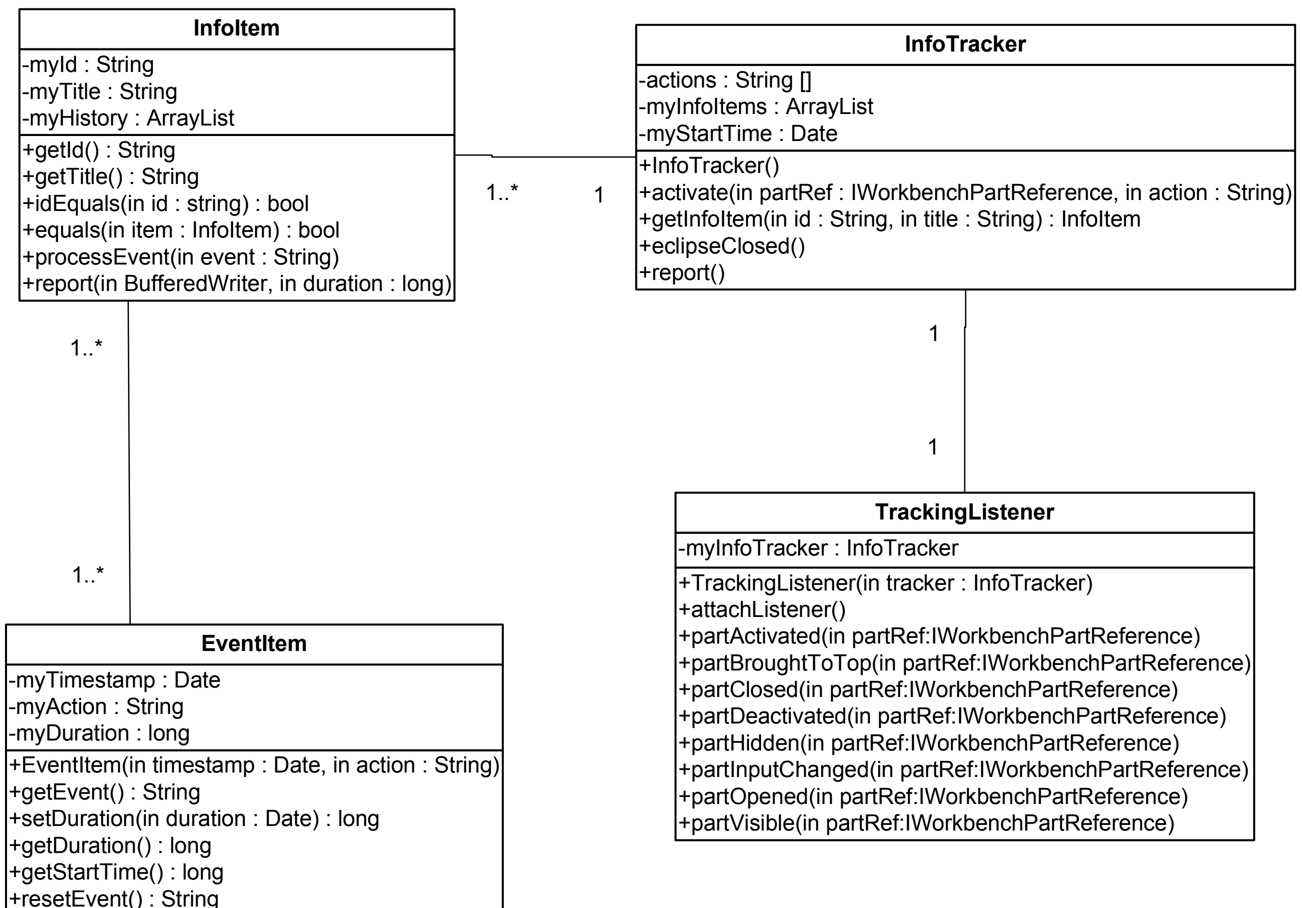
III. Tool Design

The Eclipse Development Platform was chosen as the base platform of study, primarily because it is open source and has a strong plugin development framework. In addition, this platform was designed for use by introductory computer science courses at

the University of Arizona, and therefore using Eclipse will facilitate using students in one of these courses as experimental test subjects. The intent of the plugin was to be able to track how long a particular Eclipse workbench item was either in view, active, or hidden from view. Workbench items include editor windows and any view associated with the Eclipse perspective in use. Once an `IPartListener` was attached to the relevant workbench items, the relevant interface events are tracked and an `InfoItem` object is created for that workbench item. These relevant events include opening, closing, hidden views becoming visible, activation and deactivation.

Every workbench item has an associated `InfoItem` that logs the timestamp and events as they occur. These are stored in an `ArrayList` in the order of their occurrence for later processing. The reasoning behind this design decision is that there may be a scientific experiment that requires knowledge of the entire scope of which views are visible at a particular time. While this prototype tool does not have an easy way to identify when two workbench items are open at the same time, by having event data stored in this manner and every event reported accordingly, it will be easier to either add to the tool or process the resulting report to obtain this information.

Every time the Eclipse Development Platform is closed, the resulting report data from this session is appended to an output file. Because the value of particular pieces of data will depend on the needs of each experiment, all the data collected during the session is reported: the starting and ending times for the session, each workbench item that was tracked, the events that occurred along with timestamps and durations, a summary of activity for each workbench item, and finally an identification of the most active item for that session.



The most difficult part of the tool development process was exploring the various plugin options available in Eclipse and determining which of these options would have access to the views that were visible to the user. While a view itself would have access to the primary workbench, and that workbench would have access to the other views, it would mean that a view would need to be open and operating to perform the tracking options that this tool required. An analogy of this situation would be that if a rat in a traditional scientific maze (intended to have the rat find the cheese reward via smell or other stimuli) were to have access to the map that showed where the cheese was. It added an element of the outside world into what should be a controlled experiment. For that reason, an action item was used instead. This is the equivalent of selecting an option from a drop down menu. This also was not ideal, but was the best of the various plugin options reached in the exploratory phase of development.

Attaching the `IPartListener` (which could hone in on the activation and deactivation of views and editors) required access to those views and editors. That access was obtained via the `PlatformUI`'s workbench:

```
IWorkbenchWindow w[] = PlatformUI.getWorkbench().getWorkbenchWindows();
```

It was then necessary to iterate through all of the workbench windows and all of the associated views and editors to those windows in order to attach a listener to them to track them. At this point an `InfoItem` was created for that window. Unfortunately, it was not as easy to determine when a new view was opened and create an `InfoItem` without constantly checking for new views and editors on any activation event. It is possible that this could be fixed if a different form of plugin were used (other than the drop down menu) that had additional access to views combined with an additional listener object. The number of various listener objects and various user interface components provided by the Eclipse Platform is high – which bodes well for other developers seeking to make similar tools, as the functionality is there, it will simply take time to find it.

The following is an example snippet of the results that the tool outputs for one session:

```
---------------------------------------------------
---------------------------------------------------
Start Date:Sun Apr 01 15:05:49 MST 2007          <-- Session Start and End Dates
End Date:Sun Apr 01 15:05:59 MST 2007
---------------------------------------------------
Package Explorer (org.eclipse.jdt.ui.PackageExplorer):   <-- View Name and Id
       Deactivated at Sun Apr 01 15:05:49 MST 2007 for 3422 ms
       Activated at Sun Apr 01 15:05:52 MST 2007 for 578 ms
       Hidden at Sun Apr 01 15:05:53 MST 2007 for 16 ms
       Deactivated at Sun Apr 01 15:05:53 MST 2007 for 422 ms
       Visible at Sun Apr 01 15:05:53 MST 2007 for 0 ms   <-- Activation Events
       Activated at Sun Apr 01 15:05:53 MST 2007 for 718 ms
       Deactivated at Sun Apr 01 15:05:54 MST 2007 for 5000 ms
       Closed at Sun Apr 01 15:05:59 MST 2007 for 0 ms
       -------
       Time Active = 1296 (0.12760929499803073%)
       Time Visible = 6296 (0.6199291059472233%)   <-- View Summary
       Time Hidden = 3860 (0.38007089405277666%)
----------------------
…
----------------------
Problems (org.eclipse.ui.views.ProblemView):
       Deactivated at Sun Apr 01 15:05:49 MST 2007 for 5938 ms
       Activated at Sun Apr 01 15:05:55 MST 2007 for 1172 ms
```

```
        Hidden at Sun Apr 01 15:05:56 MST 2007 for 15 ms
        Deactivated at Sun Apr 01 15:05:56 MST 2007 for 625 ms
        Visible at Sun Apr 01 15:05:57 MST 2007 for 0 ms
        Activated at Sun Apr 01 15:05:57 MST 2007 for 2406 ms
        Closed at Sun Apr 01 15:05:59 MST 2007 for 0 ms
        -------
        Time Active = 3578 (0.35230405671524223%)
        Time Visible = 3578 (0.35230405671524223%)
        Time Hidden = 6578 (0.6476959432847578%)
----------------------
…
----------------------
Most Active Item = Problems, with a duration of 3578.
```

Most Active Item this Session

While a full experiment was not run using the tool, some verification and unit testing was performed. Tests where different views and perspectives were switched between showed appropriate activation events in the correct order. Also, some basic numerical sanity checks were performed. For each session, the start date came before the end date and the next start date came after the previous end date. The total time that a view was open (the time visible added to the time hidden) was never larger than the total session time. In addition, the most active item always had the longest time active of all other workbench items.

IV. Examples of Experimental Study

The following are examples of experiments that could be conducted using the tool described above. The first is designed as an example to show how the tool could be used for generating understanding of people and their practices. The second is designed to show how this tool can be used to generate experimental results for publishing and validating research ideas. The third is designed to show how this tool can be used to evaluate additional plugins added to Eclipse.

Example Experiment #1: Analyzing Student Usage Data

Problem/Situation:

A professor in the computer science department wants to better understand the weekly cycles of the schedules of students in her class, in order to know when to schedule homework assignments and exams. The assumptions include that most students in a particular class are in similar core courses but may be in different elective courses, which will account for variations in sets of students. Also, there is the implicit assumption that schedules will be similar across different semesters, which is based on the fact that many courses are taught by the same professor year after year.

Research Questions:

Which day of the week do students in her class spend more time working on programming projects? How much total time per week do students in her class spend working on programming projects? These questions are not based on programming projects only in this course but across all of their courses that use the Eclipse development environment.

Hypotheses:

Students in her class spend more time working on programming projects on Sunday. (This is an educated guess, based on her bi-weekly project deadline of Monday at noon). She estimates that students spend between 10 and 15 hours per week in programming assignments. (This is also an estimated guess, based on her average bi-weekly project size being twenty hours of work).

Experimental Setup:

For this project, the experimental setup requires that all students use Eclipse for their class programming projects. The view tracking plugin will then be installed and remain for the entire semester. After every month, students will send in their view tracking result file for analysis.

Data Collection:

The total time spent per day on programming projects can be collected by subtracting the start and end times on each day.

Then, to determine the day of the week with the highest time spent on programming projects, all of the daily sums for each day of the week (ex. Mon, Tues) will be added together, and the largest can be calculated. In the result file, the day of the week is included at the top of each session data.

Finally, to determine the time spent per week on programming projects, each daily time spent (Sunday through Saturday) will be added together.

Data Analysis:

This data can be analyzed by looking at more than just the calculated results. In particular, if the results are different than expected, the results should be analyzed to determine why. With the weekly results calculated individually above, a plot of how those vary would be useful in understanding if there was an anomaly in weekly programming times. Comparing these weekly results with data on when assignments in other computer science courses were scheduled and when exams were scheduled might point out that students did not spend time on programming projects during weeks when exams in other courses were scheduled. Another interesting result would be if students spend significantly less time per week, because that may mean that the programming projects being assigned are too easy.

Conclusion:

Science concerns itself with seeking knowledge and understanding. This example experiment depicts a practical situation that would benefit from having knowledge rather than basing beliefs on assumptions. It also shows examples of relevant falsifiable hypotheses.

Example Experiment #2: Analyzing Process Improvements

Problem/Situation:

A research team has taken an existing agile development process and modified the communication procedures between team members. Unfortunately, there is a lack of

experimental validation for the existing process. This research team would like to compare the two processes.

Research Questions:

Does the new communication method improve overall development time?

Hypothesis:

The new communication method will improve overall development time by 10%. Despite the fact that the communication requirements are expected to add additional communication time, the communication is expected to improve code quality and thus save time in defect reduction.

Experimental Setup:

The experimental setup for this project requires that both the existing development process and the new development process be analyzed. This includes setting up a test project that is large enough to cover every step in the initial development process and require communication. Then, teams must be set up with similar backgrounds. For example, if two teams of three programmers are tested (one to test the initial development process and one to test the original process with the additional communication requirements), then each programmer on the first team should have a person on the other team with as close to their skill set as can be achieved. Should there be any large discrepancies in the two teams, then the experiment should be repeated with additional (and different) test projects and the team members varied amongst teams. This will help mitigate differences in team skill sets.

Next, one team is given instructions to execute the original development process and the other is instructed to follow the communication process developed by the research team. They will use the Eclipse development platform, including the plugin described above to complete each project. It is important that the two teams use different computers (or copies of Eclipse stored in different directories) because this prototype tool only creates and appends to a single output file.

Data Collection:

While few papers concerning software processes have empirical data to support them, they do tend to have surveys where the developers can give their feedback on the ease of use of the process. This project does not intend to take away from that data which is more subjective – it should be used in addition to it. Therefore, each team should be given a survey that collects subjective feedback that would help the research team improve their process if it does not meet the stated target in the hypothesis.

Total development time for each project can be collected by subtracting the end time from the start time of each development session and then adding all of the individual session values.

Data Analysis:

The results for this experiment will be two total development times that can be compared to evaluate the improvement that the communication procedure has on the original process. If the overall time increased with the additional communication

procedures, then that procedure probably isn't useful with the process provided. However, because a test project was selected, multiple teams could be set up with different skill sets to attempt the same project and perhaps determine if skill sets of programmers are a factor in whether or not this procedure is a success.

If overall time decreased with additional communication, the process was still so clearly defined that another research team could validate their experiment in order to pick out any anomalies or to verify the results.

Conclusion:

This is an example of where in a typical research project scientific experimentation can play a powerful role and lead to better understanding of a newly developed process or tool. Empirical data will show the strengths and weaknesses of a process so that it can be compared against other processes as well as be compared against itself when improvements are made.

Example Experiment #3: Analyzing Environment Improvements

Problem/Situation:

A company has decided to add a plugin to Eclipse that displays a list of all recent changes to a project artifact (files, executables, resources, documentation, etc). They are interested in obtaining usage data on how their developers are using this tool in conjunction with other Eclipse views, so that if significant switching time is occurring they can add additional functionality to their tool to minimize switching time.

Research Questions:

Do users tend to use a particular view before using this new Recent Changes view?

Hypothesis:

Users tend to use the Project Hierarchy view immediately before the Recent Changes view in order to determine which project artifact they want to see changes on.

Experimental Setup:

In this event, the tool developer would package this view tracking tool with their plugin and request that the results file be sent out after a few weeks of standard usage. Rather than set up a specific project that should be tested, in this case the goal is to determine how the tool is used in the real world.

Data Collection:

The various results files will be scanned for the relevant name of the Recent Changes view (likely, "Recent Changes"). It then lists a sequence of stages that this view was in. The timestamps at which it was Activated should be recorded. Next, the results file should be searched for other views that were Active just prior to that timestamp, or were Visible just prior to or during that timestamp. This can be done with a script file that checks the timestamps of each activity on a view and compares it to the timestamp of the next activity of a particular view. If the next timestamp of an Activated view is

within 100 ms of the Recent Changes activation, then there is likely a relation. Or, if the next timestamp of a Visible view is within 100 ms of the Recent Changes or the range of timestamps contains the timestamp of the Recent Changes activation, then there is likely a relations. All such relations should be recorded.

Data Analysis:

If there appears to be an even distribution for which views are related to each other views, then the hypothesis would be false. There would in fact be no clear relation between this tool and any other view in use by the user. However, another dramatic result would be if there was very little use of this tool at all – it would show that users are not actually using this tool to the extent expected.

Conclusion:

This example shows how empirical results can give important data that cannot necessarily be collected otherwise. A coworker can claim to be using the environment improvement tool extensively but they may not understand if their development time is increasing because of setups required for its use or they could be trying to be nice when they have decided not to use the tool provided. This empirical data can assist in identifying improvements to the tool, or determining if developing environment improvements is a valuable way to spend company time.

V. Further Research

Should another researcher be interested in further improving my tool, or creating additional supporting tools, I have identified a number of technical improvements that would benefit the computer science research community from a scientific perspective.

First, a technical issue with my tool is that the view tracking must be started manually. A valuable improvement would be to determine when all views have been instantiated by Eclipse and initiate tracking at that point, instead of via a manual selection. Due to the complexity of the Eclipse platform, I could not identify the point at which all views were created. This may be an issue with the style of plugin developed. The importance of this improvement is to prevent user error in users forgetting to turn on the plugin.

A second research initiative that would provide value to the research community seeking to use this tool in their research is to develop a separate tool to analyze the report data collected. Currently, the report data generated by the tool was meant to have quantity and quality, but this report does not make claims about the meaning of the data results. This means that researchers using this tool must develop their own analysis scripts to extract the individual data results that they need. An interactive tool that had the ability to display the various ways to view and interpret the data results would encourage scientists to use this tool, as an interactive report viewing tool would minimize the work necessary on their part to obtain value in their research results. In particular, I think the ability to step through events and get a snapshot of view states at a particular instant would be valuable to a number of research situations. Also, different collections of summary information presented together is more appropriate in a secondary tool than was in my reporting page, because for this tool to display a large amount of summary data would have caused redundancies in information displayed.

A third research initiative would be to add similar tracking devices to other tools that are typically used in software processes. Software processes are not always performed in isolation with a single system, and thus benefit is gained from tracking the entire process. Other tools that may benefit from such tracking include email clients, web browsers, instant messaging tools, and other development environments and tools. This tool tracks which views are open at a particular instance – a tool that would track which programs are open on a particular computer at a given instance, and tracks their activations and visibility would merely expand the benefits of this tool to a larger scale of analysis.

VI. Conclusion

In summary, this view tracking tool integrates closely with Eclipse to provide data on when views and editors are visible, activated, closed, hidden, etc. Extrapolating meaning from this data is experiment dependent, but the duration of a particular view's visibility or activity can be instrumental in understanding how software developers interact with the Eclipse tool, whether or not a software process is effective, and the sequence of events that a developer uses in their daily interactions with the Eclipse tool and the software project they are working with. Without such empirical data, claims about engineering improvements to processes or tools cannot be justified scientifically.

Although initial research into the Eclipse Development Platform was time consuming, the design and implementation of this tool was straightforward. Only four of the six classes used in the tool required significant development; the Eclispe plugin wizard generated the remaining two in order to hook unto the Eclipse Platform. Those four core classes only account for 249 lines of code (calculated by removing all blank spaces, lines with only curly braces, and lines with only comments). This tool is one-step on the path to ensuring that computer scientists have the ability to easily obtain scientific data about their own discipline and extrapolate meaning from that data. Science concerns itself with making claims about underlying causes for their empirical results, but without those empirical results it is difficult to make claims about underlying causes. The value obtained by a tool with such little development time after initial environment study demonstrates that this method of data collection and analysis should be considered more frequently in computer science research as a valid, valuable approach.

VII. Bibliography

Denning. "Is Computer Science a Science?" CACM 48(4):27-31. April 2005

Denning. "What is Experimental Computer Science?" CACM 23(10) 543-544. October 1980.

Eclipse Platform API Specification. Release 3.1. IBM Corp. 2005. Online. Available: http://help.eclipse.org/help31/nftopic/org.eclipse.platform.doc.isv/reference/api/index.html

Eclipse Platform Technical Overview. Object Technology International, Inc. February 2003. 1-20.

Glass. "A Structure-Based Critique of Contemporary Computing Research." Journal of Systems Software. 38:3-7. 1995.

Simon and Newell. "Computer Science as Empirical Inquiry." CACM 19(3):113-126. March 1976.

Simon, “The Sciences of the Artificial,” MIT Press.
Springgay, Dave. “Creating an Eclipse View.” November 2001. Online. Available: http://www.eclipse.org/articles/viewArticle/ViewArticle2.html
Proulx, Emmanuel. “Eclipse Plugins Exposed, Part 1: A First Glance.” O’Reilly On Java.com: The Independent Source for Enterprise Java. 1-2. February 2005. Online. Available: http://www.onjava.com/pub/a/onjava/2005/02/09/eclipse.html .