Teaching Software Engineering skills with the Leap Toolkit

Carleton A. Moore
Collaborative Software Development Laboratory
Information & Computer Sciences Department
University of Hawaii, Manoa
Honolulu, Hawaii 96822 USA
(808) 956-6920
cmoore@hawaii.edu

ABSTRACT
The Personal Software Process (PSP) teaches software developers many valuable software engineering techniques. Developers learn how to develop high quality software efficiently and how to accurately estimate the amount of effort it will take. To accomplish this the PSP forces the developer to follow a very strict development model, to manually record time, to defect and size data, and analyze their data. The PSP appears successful at improving developer performance during the training, yet there are questions concerning long-term adoption rates and the accuracy of PSP data.

This paper presents our experiences using the Leap toolkit, an automated tool to support personal developer improvement. The Leap toolkit incorporates ideas from the PSP and group review. It relaxes some of the constraints in the PSP and reduces process overhead. We are using the Leap toolkit in an advanced software engineering course at the University of Hawaii, Manoa.

Keywords
Software Developer Education, Process Improvement, Measurement, Personal Software Process

1 INTRODUCTION

Every software engineer occasionally wishes they got home from work sooner and spent less of their weekends at work. Software developers work very hard and very long, yet software is often delivered late, over budget, and full of defects. How can software engineers learn to produce high quality software more efficiently?

Software developers and managers have addressed the issue of software quality and development issues by focusing on the work product and the development organization. Work product solutions include practices such as testing and reviews. Testing and additionally reviews help improve the work product and reviews attempt to reduce the cost of development by finding and fixing the defect earlier in the development process.

Organization solutions focus on the software development organization and processes. There are hundreds of organization level solutions. Two widely followed organization-mode solutions are the Capability Maturity Model[8] and ISO 9000[6]. Both of these methods focus on the development organization and processes.

Traditional software engineering techniques such as requirements specifications, modularity, data abstraction, coupling and cohesion, PERT and GANTT charts, version control, and so forth are "best practice" techniques that provide developers tools for solving specific software engineering problems. Software developers should know how to use these "best practices" on their development problems. Not all practices are appropriate to every situation, however using the correct practice can save a project.

When a developer joins a software development organization they must learn the particular development processes for the organization. The developer has to learn what testing and review methods are used in the organization. They must learn how the organization manages the software development process.

In 1995, Watts Humphrey introduced the Personal Software Process in his book *A Discipline for Software Engineering*[4], a software development process and improvement process for individual software developers. The PSP incorporates many of the above best practices into a single method for software developer improvement.

After using the PSP for over two years we developed the Leap toolkit to overcome restrictions that we noticed in the PSP. The Leap toolkit relaxes many of the constraints in the PSP, and includes extensive automated support in order to simplify training and adoption of individual, empirically-based developer improvement.

Draft of paper submitted to ICSE 2000.

2 Personal Software Process (PSP)

The PSP teaches software developers techniques intended to support high-quality software development and how to improve estimate the amount of effort required to produce software. These two goals drive the entire PSP process.

Learning software development skills with the PSP

To teach developers how to use the PSP, Humphrey defines seven PSP processes (0, 0.1, 1.0, 1.1, 2.0, 2.1, 3.0). Each process has detailed scripts telling the user exactly how to perform the process. Figure 1 shows the seven levels. Each level builds upon the previous levels and introduce new software engineering concepts. Exercises at the end of each chapter ask the reader to use the knowledge from the chapter to improve their development skills. The book teaches powerful development techniques: design and code reviews, size and time estimation methods, and design templates. These techniques help the developer produce high quality products efficiently. As developers go through the book they develop 10 small software projects using the different PSP levels.

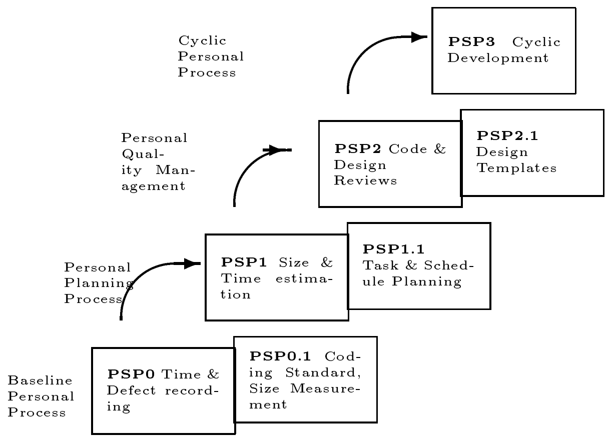

Cyclic Personal Process
PSP3 Cyclic Development
Personal Quality Management
PSP2 Code & Design Reviews
PSP2.1 Design Templates
Personal Planning Process
PSP1 Size & Time estimation
PSP1.1 Task & Schedule Planning
Baseline Personal Process
PSP0 Time & Defect recording
PSP0.1 Coding Standard, Size Measurement

Figure 1: PSP levels

In the Baseline Personal Process level the developer learns how to record their development process. They record all defects that they find in their source code, all the time it takes them to develop the source code and the number of lines of code in their source. From PSP 0.1 on the developer must use a very strict waterfall method of development. 100% of the code must be written before the developer can do their first compilation. This development method teaches the developer how to manage their development process. The next level of the PSP teaches the developer how to plan their projects.

In the Personal Planning Process level the developer learns how to estimate and plan future projects. The developer starts the planning phase by estimating the size of the next project. The PSP uses a Proxy Based Estimation (PROBE) technique to simplify size estimation. To estimate the number of LOC for the next project, the developer develops an initial design and counts the number of methods. Methods are the proxy for lines of code. The developer estimates the size of each method. Based upon historical project size data the average LOC for five different method sizes is calculated. The developer then counts up the number of methods of each size category and multiplies by the average LOC per method for that size category. The developer sums up all the LOC and gets the total estimated LOC for the project. Once the size of the project is estimated the developer can estimate the amount of time the project will take. The PSP uses a complicated time estimation technique. The PSP time estimation technique requires the developer to calculate two linear regressions. The first regression is based upon their planned size for each project vs the actual time spent developing the project. The second regression is between the actual size of each project and the actual time of taken for the project. Depending upon the correlation between the two size data sets, planned or actual size, and the actual time taken, the developer uses linear regression or historical average to calculate the effort for the next project. Based upon the estimated time and past direct hour calculation the developer can schedule the project. The next level of the PSP teaches the developer how to increase the quality of their source code.

In the Personal Quality Management level the developer learns how to efficiently find and remove defects from their code by using reviews. They also learn how to use design templates to help improve their design skills. Prior to the introduction of design and code reviews most of the defects are removed during the compile and test phases. Adding design and code review efficiently catches many defect before the compile phase. At this level in the PSP developers can plan their projects and efficiently remove defects. The next level teaches developer how to build larger software products.

In the Cyclic Development Process level developers learn how to break large project into smaller chunks. The PSP teaches developer how to do high-level design to find the cycles of development. Each cycle of the large project is considered a sub-project and the developer can use the PSP to manage them. To determine if the PSP is a powerful tool for improving an individual's software engineering skills, many researchers have evaluated the PSP.

Evaluations of the PSP

In a 1996 article, Watts Humphrey reported the results of 104 engineers taking the PSP course[5]. He states that the two goals of PSP were met. First, reported defects fell from an average of 116.4 defects per thousand lines of code (KLOC) for assignment 1 to 48.9 defects per KLOC for assignment 10. Second, the estimation accuracy of the students increased. For assignment 1 32.7% of the engineers' estimates were within 20% of their actual times. By assignment 10 49.0% of the engineer's estimates were within 20% of their actual times.

In 1996, Sherdil and Madhavji studied human-oriented improvement in the Software Process[9]. They used PSP as a basis for their studies. They found that subjects reduced their defect by 13% after project 6, when code reviews are introduced. They also found that their subjects reduced their size estimation error by more than 7% than expected.

Hayes and Over conducted an extensive study, with 298 engineers, of the PSP[3]. The results of the study were impressive. Over the projects completed, the median improvement in size estimation was a factor of 2.5. This means that 50% of the engineers reduced their size estimation error by a factor of 2.5. The median improvement in time estimation was 1.75. The median reduction in overall defect density was by a factor of 1.5. The engineers substantially reduced the percentage of defects surviving to later stages of development.

Anne Disney did her masters thesis on data quality issues in the PSP[2]. She found that in her sample of students who learned the PSP, the errors in their data were significant. These errors lead to incorrect insights into the students development practices. For example, in several cases the students' incorrect data indicated that they were over estimating their yield when in fact they were underestimating their yield.

Most of the errors that Disney found in the the students' PSP data were caused by the manual nature of the PSP. 66% of the errors were either calculation errors or transfer errors. The students had difficulty correctly manually calculating some of the values required and transferring the values between the many forms used in the PSP. This research and our experiences with using the PSP raised some important issues with the PSP.

Issues with the PSP
After using the PSP for over two years, we noticed several issues with the PSP:

- The PSP is designed for only for software development. It hardwires estimation, size measurement, and development processes. Since the PSP is designed around a set of hardcopy forms, the processes in PSP are intimately tied to the hardcopy forms. Modifying the forms or the processes are sufficiently difficult that the Humphrey strongly advises against doing so, at least until after finishing the course.

  The entire PSP process is focused on one type of work product, source code. If the software developer does not produce source code they cannot use the PSP. They must also modify most of the forms if they do not have a "compile" phase in their development process. Some of the key analyses calculate the number of defects before and after "compile". Without a "compile" phase these analyses are meaningless.

- The manual nature of the PSP introduces large amounts of overhead for the developer. It also reduces the benefits to the developer since it is difficult to analyze all the data. For each project the developer records their time, size and defects on log forms. At the end of the project they produce a postmortem report that summarizes the project. In higher levels of the PSP the developer must use over 10 different forms. Producing a report of the most costly defect type is a very time consuming process since the developer must pour through all of their defect logs and collate the data manually.

- The PSP collects all defects from the first project. This project introduces additional overhead to the developer's development process. They have to follow a new development process and record their effort. The addition of recording all their defects changes their development process significantly. Manually recording each defect on the defect recording log interrupts their thought processes.

- The PSP has no group support. The PSP is a personal process, however some of the most valuable insights about your development process may come from other developers that can see issues that you cannot. Coaches observe athletes perform and provide insight the athlete cannot get themselves, similarly in software development other observers can provide insights that the developer cannot see themselves.

These issues motivated us to begin designing an automated, empirically based, personal process improvement tool. Our goal is to reduce the collection and analysis overhead for the developer. This should improve the benefits to the developer and the long term adoption of empirically based process improvement. To pursue this work, we initiated Project LEAP, `<http://csdl.ics.hawaii.edu/Research/LEAP/LEAP.html>`, and began developing the Leap Toolkit.

3 Leap

Design Criteria
We hypothesized that improved support for software developer improvement would be obtained by attempting to satisfy four major design criteria: a light-weight process, empirical measurement, minimization of measurement dysfunction, and portability within and across organizations.

Criteria #1: Light-Weight
The first principle is that any tool or process used in software developer improvement should be light weight. This means that the tool or process should not impose too much overhead on the developer. Data collection should be easy to perform and should not add significant effort to the process. The processes that are used should not impose a burden on the developer. We do not want the developer to worry about the improvement effort while they are doing the development. They should be worrying about the development. Analyses and other work should also require as little effort by the developer as possible. The benefit of using the improvement processes should outweigh the cost of to the developer.

This principle implies that any improvement process must be automated as much as possible. A manual process requires too much overhead by the developer. The overhead of recording information by hand and manually doing the analyses will out weigh the benefits of the process. The PSP suffers from the problem of high overhead for data entry and analysis. It also suffers from high process overhead.

Criteria #2: Empirical
We believe in empirical data collection–that improvements should be based upon the developer's experiences. Leap supports the observe-evaluate-modify improvement cycle. Each modification is then tested by further observation to see if the change is actually an improvement or just a false start. By using looking at their development empirically the developer is able to judge for themselves what is best.

Criteria #3: Anti-measurement Dysfunction
Based upon our experiences with industrial software development and Richard Austin's book *Measuring and Managing Performance in Organizations*[1], we believe that any process improvement method should deal with the issue of measurement dysfunction. The empirical data collected could be misused. This issue is important since the development process is very interesting to people other than the developer. If there is measurement dysfunction then the data collected and analyses will not reflect reality. Any insights gained from this data and analyses will be faulty and may cause more problems than they solve.

Criteria #4: Portable
Software developers often change jobs and the tool support for their development improvement should be portable. They should be able to take their data and the tool support with them when they change organizations or jobs. A tool that supports developer improvement that cannot follow the developer as they move is not going to help those developers very much.

Based upon the four design principles we developed the Leap Toolkit, a Java application for software developer improvement. The Leap Toolkit combines the data recording and analysis ideas of the PSP with group review. The developer can record time, size, and defect data for their project just like in the PSP. They can also ask their co-workers to review their work product using the Leap Toolkit and share any defects the reviewers find. The Leap Toolkit also supports checklists and patterns.

Benefits of Leap
The Leap Toolkit provides many benefits and has allowed us to explore different aspects of personal software process improvement. Some of the benefits are fewer constraints, lower overhead and support for group review.

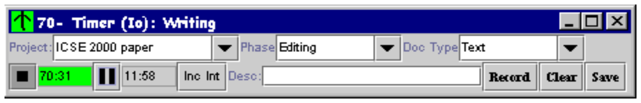
Relaxes many constraints in the PSP
The Leap toolkit allows the developer to define their own size definitions, development processes, defect types, defect severities and document types.

By allowing the developer to define their own hierarchical size definition the Leap toolkit can support different size measurements for the same work product. For example, a developer might define a object oriented size definition for Java that consists of packages that contain classes that contain methods that have lines of code. They might also define a size definition that just counts the number of expressions in the source code. Both of these size measures could be used for the same source code. The Leap Toolkit allows the developer to compare their projects in either of these size definitions.

The Leap toolkit allows the developer to define their own processes. They can use the PSP's software development process or use their own process. The developer can define processes for non-software development activities. For example, one writing process includes the following phases: Brain storming, Planning, Outlining, Writing and Editing.

Leap also relaxes the constraint that time, size and defects must always be recorded. In our Advanced Software Engineering classes we wait until the students have mastered time and size recording before we introduce defect recording. This reduces the cognitive overhead on the students. They become accustomed to collecting data about their development process without overly disrupting their process. Teaching the developer how to collect the data that they are interested in gives them more control over their process improvement efforts.

Low overhead


70- Timer (Io): Writing

Project: ICSE 2000 paper Phase: Editing Doc Type: Text

70:31 11:58 Inc Int Desc:

Figure 2: Time Recording Tool (Io) recording time for this paper. This screen shot illustrates an interactive time. During this session, 70 minutes of direct time has elapsed with almost 12 minutes of interrupt time.

Since the Leap Toolkit is an automated tool the amount of overhead for recording size, time and defect is greatly reduced. The developer does not have to take their hands off the keyboard and mouse to enter their data. Time recording is simplified by a single line time entry tool. Figure 2 shows the time entry tool Io recording time for this paper.

The Leap Toolkit also has a similar tool for defect collection. This tool simplifies the recording of defects and their fix times. Figure 3 shows the defect recording tool Mano.

As a part of his Master's thesis Joseph Dane developed LOCC[7] a grammar based size counting tool that produces size data that the Leap Toolkit can use. Currently, LOCC supports counting Java, C++ and plain ASCII text files. Using LOCC greatly simplifies the problem of determining the actual size of the current project.

All of these tools for data collection have reduced the developer's overhead so much that, in many cases, it now takes more effort to fake your data than it takes to collect accurate data. In her study of students' PSP data, Anne Disney found some suspicious data that possibly indicated that the students were making up their data. The Leap toolkit solves this problem–at least for situations where the data is faked due to time constraints.

The Leap Toolkit also allows the developer to rapidly analyze their data. In the manual PSP, comparing different time estimation techniques is very time consuming. Leap allows the developer to rapidly compare different time estimation models such as: linear regression, historical average, exponential regression, and power regression. The Leap Toolkit also allows the developer to quickly compare their planning values to their actual values. Figure 4 shows the Leap time estimation tool.

Support group review
To provide additional insight into a developer's process, the Leap Toolkit supports group review. Users can email their Leap data to each other. This allows reviewers to use the Leap Toolkit to record and share defect data. The Leap toolkit can load all the defects found by the different reviewers and then filter, and sort the data. The insights provided by other reviewers might be even more valuable than the defect detected by the developer.

The Leap Toolkits flexibility even allows reviewers to record data about their review process and improve their review techniques. We can use the toolkit to teach developers different review methods and compare the results of these different methods.

4 Strengths: Leap vs. PSP

The PSP's strong scripts tell the developer exactly what they have to do and exactly what data they must collect. There is very little ambiguity when using the PSP. The user just follows the scripts and does the analyses called for. The developer does not have to create their own goals, questions, and metrics: they are built into the PSP.

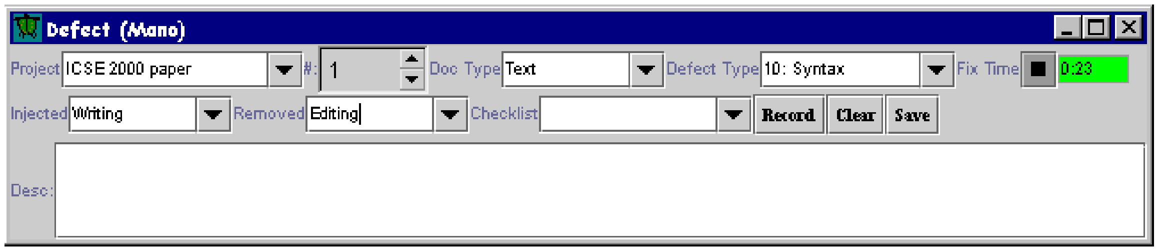
Leap builds upon many of the strengths of PSP. Automated tool support for data collection and analysis greatly reduces the user's overhead. By relaxing many of the constraints, Leap gains valuable flexibility. Developers can use the Leap toolkit to improve their development skills in a wide range of activities not possible with the PSP. Developers are able to experiment with different development techniques to find the one best suited to them.

5 Experiences with Leap

We started developing Leap in the summer of 1997. Our first release, 1.7.0, of the system, in November 1997, only supported recording and analyzing defect data. Since then we have made 25 public releases of the Leap toolkit. Currently, the Leap toolkit, version 5.8.2, consists of over 41,000 lines of Java code, over 2,000 methods, and over 275 classes in 13 packages. It is available for down-load from `<http://csdl.ics.hawaii.edu/Tools/LEAP/LEAP.html>`.

We have used the Leap toolkit to help teach advanced software engineering at the University of Hawaii. The Leap toolkit is currently being used by Dr. Philip Johnson in ICS 613: Advanced Software Engineering `<http://csdl.ics.hawaii.edu/~johnson/613f99/>`. Over the past 2 and a half years we have collected data on over 350 projects. The Leap toolkit has been used by several people working in industry not affiliated with the Collaborative Software Development Laboratory.

We are currently conducting an evaluation of the Leap toolkit by surveying and interviewing the students in ICS 613. We are also using the Leap toolkit to investigate the accuracy of different time estimation methods


Defect (Mano)

Project: ICSE 2000 paper #: 1 Doc Type: Text Defect Type: 10: Syntax Fix Time: 0:23

Injected: Writing Removed: Editing Checklist:

Desc:

Figure 3: Defect Recording Tool (Mano) recording a simple syntax error found in a draft of this paper.

based upon historical size and time data. The initial results of our evaluation will be available in January, 2000.

REFERENCES

[1] R. D. Austin. *Measuring and Managing Performance in Organizations.* Dorset House Publishing, 1996.

[2] A. M. Disney. Data quality problems in the Personal Software Process. M.S. thesis, University of Hawaii, August 1998.

[3] W. Hayes and J. W. Over. The Personal Software Process (PSP): An empirical study of the impact of PSP on individual engineers. Technical Report CMU/SEI-97-TR-001, Software Engineering Institute, Pittsburgh, PA., 1997.

[4] W. S. Humphrey. *A Discipline for Software Engineering.* Addison-Wesley, New York, 1995.

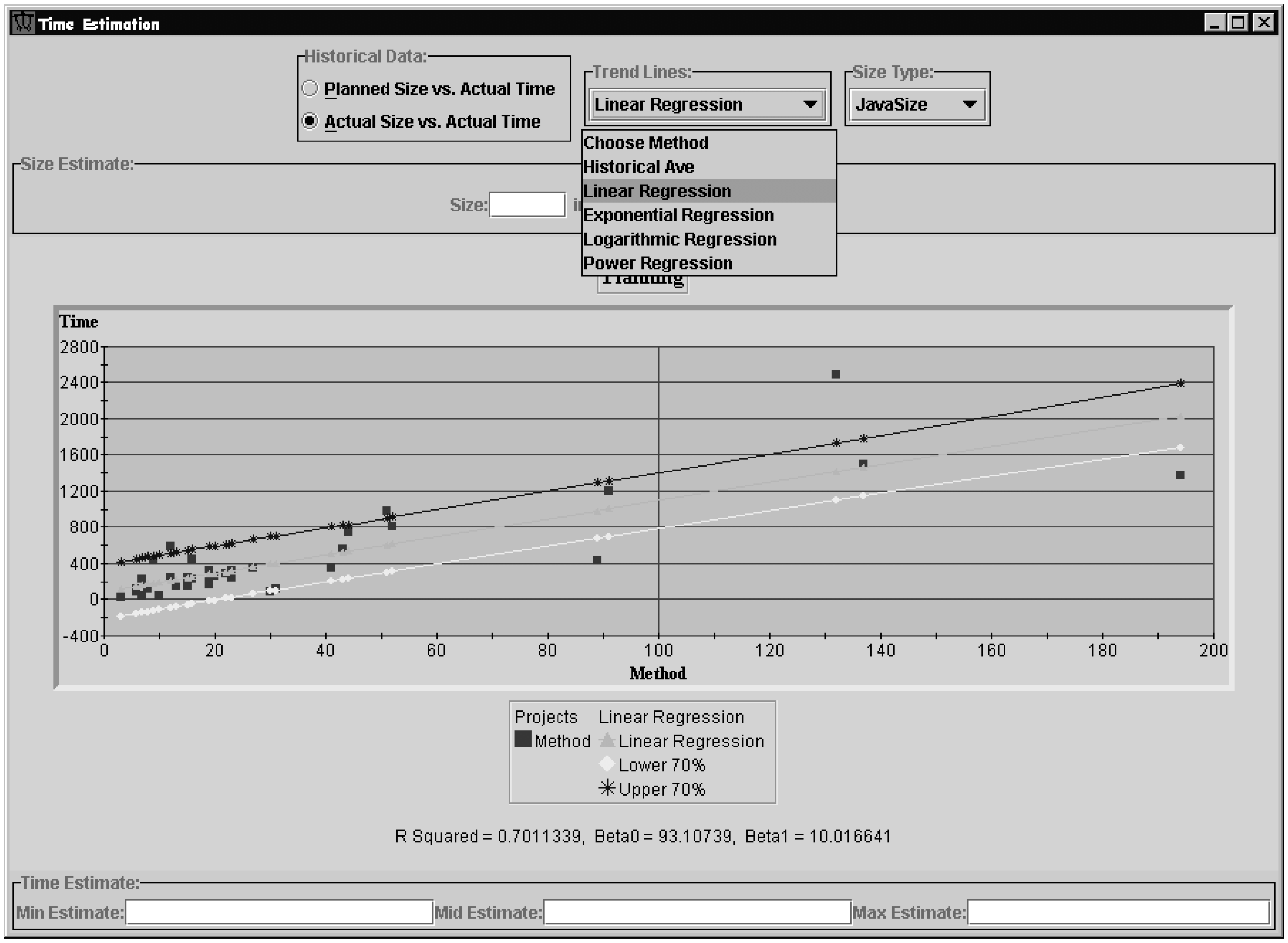
[5] W. S. Humphrey. Using a defined and measured Personal Software Process. *IEEE Software*, 13(3):77–88, May 1996.

[6] International Organization for Standardization. *ISO Standards Compendium - ISO 9000 Quality Management*, 7th edition, 1998.

[7] The locc system. http://csdl.ics.hawaii.edu/Tools/LOCC/LOCC.html.

[8] M. Paulk, C. Weber, B. Curtis, and M. B. Chrissis. *The Capability Maturity Model: Guidelines for Improving the Software Process.* Addison-Wesley, New York, 1995.

[9] K. Sherdil and N. H. Madhavji. Human-oriented improvement in the software process. In *Proceedings of the 5th European Workshop on Software Process Technology*, October 1996.



Historical Data: Planned Size vs. Actual Time [✓ Actual Size vs. Actual Time]

Trend Lines: Linear Regression

Size Type: JavaSize

Size Estimate:

Size:

R Squared = 0.7011339, Beta0 = 93.10739, Beta1 = 10.016641

Time Estimate:

Min Estimate: Mid Estimate: Max Estimate:

Figure 4: Time Estimation Tool with the author's Java development data. Showing the different time trend line options.