AUTOMATIC DIFFICULTY DETECTION

Jason Carter

A dissertation submitted to the faculty at the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2014

Approved by:

Prasun Dewan

Fred Brooks

Diane Kelly

David Stotts

Wei Wang

ABSTRACT

Jason Carter: Automatic Difficulty Detection
(Under the direction of Prasun Dewan)

Previous work has suggested that the productivity of developers increases when they help each other and as distance increases, help is offered less. One way to make the amount of help independent of distance is to develop a system that automatically determines and communicates developers' difficulty. It is our thesis that automatic difficulty detection is possible and useful.

To provide evidence to support this thesis, we developed six novel components:

- programming-activity difficulty-detection
- multimodal difficulty-detection
- integrated workspace-difficulty awareness
- difficulty-level detection
- barrier detection
- reusable difficulty-detection framework

Programming-activity difficulty-detection mines developers' interactions. It is based on the insight that when developers are having difficulty their edit ratio decreases while other ratios such as the debug and navigation ratios increase. This component has a low false positive rate but a high false negative rate.

The high false negative rate limitation is addressed by multimodal difficulty-detection. This component mines both programmers' interactions and Kinect camera data. It is based on the insight that when developers are having difficulty, both edit ratios and postures often change.

Integrated workspace-difficulty awareness combines continuous knowledge of remote users' workspace with continuous knowledge of when developers are having difficulty. Two variations of this component are possible based on whether potential helpers can replay developers' screen recordings. One limitation of this component is that sometimes, potential helpers spend a large amount of time trying to determine if they can offer help.

Difficulty-level and barrier detection address this limitation. The former is based on the insight that when developers are having surmountable difficulties they tend to perform a cycle of editing and debugging their code; and when they are having insurmountable difficulties they tend to spend a large amount of time a) between actions and b) outside of the programming environment. Barrier detection infers two kinds of difficulties: incorrect output and design. This component is based the insight that when developers have incorrect output, their debug ratios increase; and when they have difficulty designing algorithms, they spend a large amount of time outside of the programming environment.

The reusable difficulty-detection framework uses standard design patterns to enable programming-activity difficulty-detection to be used in two programming environments, Eclipse and Visual Studio. These components have been validated using lab and/or field studies.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1. Introduction

This thesis is about collaborative software development. In such development, programmers interact with each other for a variety of reasons such as to get clarification on tasks, divide tasks, and resolve conflicts. A relatively unexplored form of such interaction is developers helping each other. When developers help each other, they stop their own work to help others. If this is true, helping teammates may decrease team productivity. However, three research results imply that helping teammates could actually improve team productivity.

The first result by Herbsleb et al. [27] found that the productivity of co-located teams, teams that are in the same building but sit in different cubicles, was higher than that of distributed teams, teams that are in geographically dispersed locations. The reason was that co-located teammates helped more than remote teammates. An earlier study by Herbsleb and Grinter suggests one reason for this phenomenon. They found that distributed developers are less comfortable asking each other for help because they interact with each other less than co-located developers [25]. If this is true, then as people get more opportunities to interact with each other, their productivity should increase.

Teasley et al. found that this indeed was the case. They found that radically co-located teams, teams in a single building that work in a war-room or bull-pen, were more productive than co-located teams because they saw and overheard each other's activity [55]. This enabled them to interject and provide help when they noticed that someone was having difficulty.

More concretely, if someone was having difficulty with some aspect of code, another developer in the war-room who is "walking by [and] seeing the activity over their shoulders, would stop to provide help." However, if developers are not nearby, sometimes they miss opportunities to help.

Conversely, constantly monitoring developers to see if they need help may increase their interactivity to an even greater degree. This is exactly what happens in pair programming where two programmers sit next to each other, with one programmer, called the driver, writing code, and the other programmer, called the navigator, offering help. Williams and Cockburn [9] found that student pair programming teams were more productive than solo programmers. They measured productivity using the following equation:

$$\frac{1}{task\ completion\ time + bug\ fix\ time}$$

where task completion time is the amount of time it took to complete assignments and bug fix time is the amount of time it took to fix bugs.

Together, these studies suggest that a) developers' productivity increases when they help each other and b) as distance increases, help is offered less. One way to increase the amount of help when teammates are not face-to-face is to make help independent of distance. To make help independent, we model the help-giving process. The first step in such a model is for potential helpers to become aware that developers need help (Figure 1.1). There are two ways they can become aware of teammates' difficulties. Developers can either explicitly ask for help, or teammates can detect or passively notice the need for help.

In some cases, asking for help is more efficient than passively noticing the need for help because askers do not have to wait for helpers to notice that they need help and helpers do not have to monitor teammates to determine if they need help. For example, in a classroom with hundreds of students and four teaching assistants, it is much more efficient for students to ask for help than for teaching assistants to notice that students need help. On the other hand, passively noticing the need for help is more efficient if developers delay asking for help. Two recent studies show that sometimes developers do indeed delay asking for help.

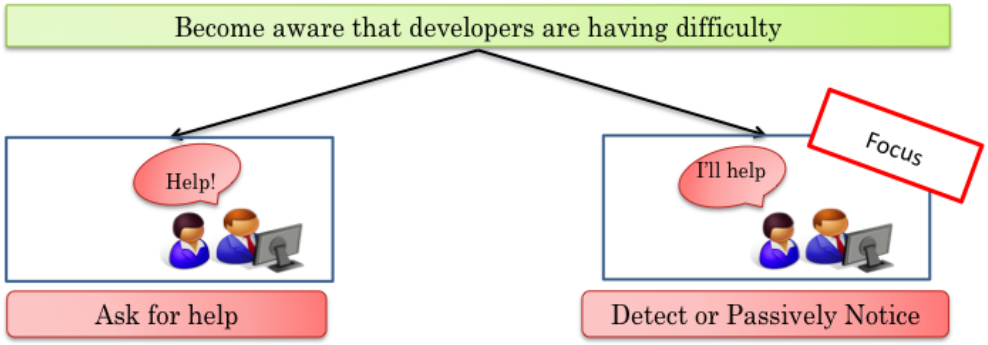


Figure 1.1: First step in help-giving model.

Begel and Simon found that students and new programmers are often late to use help while Latoza et al. found that programmers often exhaust other forms of help such as code or documentation before asking a teammate for help [3,39]. Thus, both requesting help and passively noticing the need for help are important. However, as mentioned earlier, as distance increases a) people are less willing to ask for help and b) fewer passive awareness scenarios are possible. The solution is to make help independent of distance. There are two ways to do so: reduce the need for developers to ask for help, and promote help.

Herbsleb et al. propose two solutions that reduce the need for developers to ask for help [25]. The first is to design software components to be as modular as possible and assign only

components that do not depend on each other to different locations. The second is to only split the development of only well-understood software components.

Social and technical solutions promote help. Social solutions enable developers and potential helpers to interact with each other without using special help-promotion tools. These solutions are important, because, as mentioned above, distributed developers sometimes delay asking for help. Herbsleb et al. proposed a social solution that addresses this problem [25]. Their solution is to bring distributed teammates who need to communicate together at the beginning of projects. These initial meetings enable distributed teammates to get to know each other. Technical solutions enable distributed developers and potential helpers to use special help-promotion tools to interact with each other. There are many technical solutions that promote both actively asking for help and passively noticing the need for help (Figure 1.2).

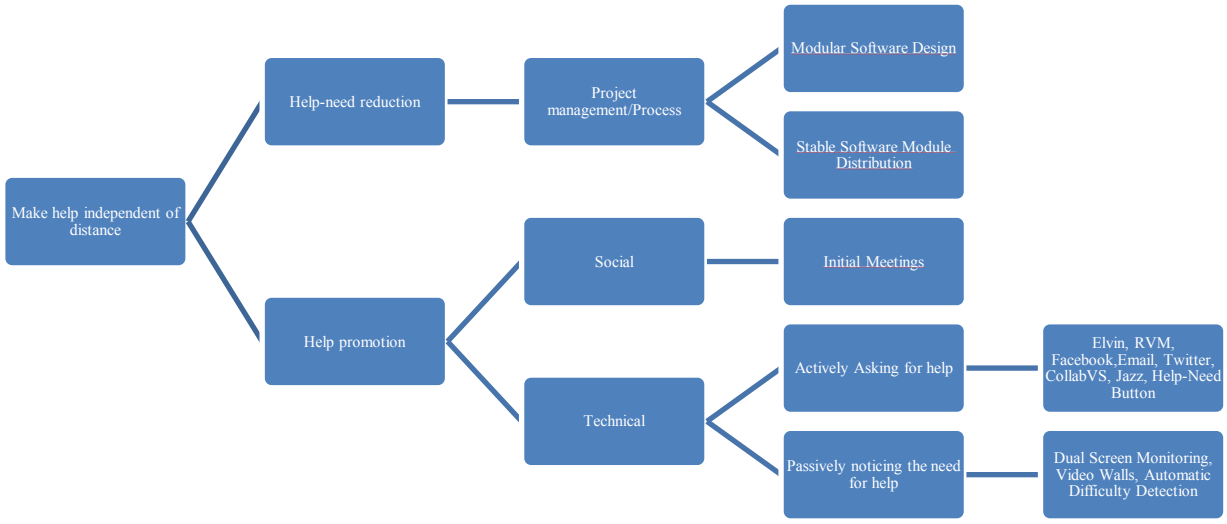


Figure 1.2: Help Independent of Distance.

One way to promote asking for help is to enable teammates who need help to use Elvin to post messages which are continuously shown to collaborators in a ticker tape [18]. These

messages indicate teammates' need for help. Herbsleb et al. also created a similar tool, Rear View Mirror (RVM), which provides distributed teammates two ways to ask for help [26]. Teammates can a) change a status message that is displayed to collaborators after some period of keyboard and mouse inactivity or use a more direct approach and b) send an instant message to collaborators. The features of these two early research tools, Elvin and Rear View Mirror, can also be found in modern state-of-the-practice tools. For example, programmers can send an email, tweet, instant message, or Facebook messages to potential helpers, or ask a question on a discussion forum.

These tools, including Elvin and Rear View Mirror, require developers who are interacting with a programming environment to switch to a separate tool to ask for help. A more lightweight approach is to provide the capabilities of the tools within the user interface of the programming environment. Both Jazz and CollabVS use this approach [8,24]. Jazz enables distributed developers to a) change a status message that is displayed to collaborators and b) send an instant message to each other within the Eclipse programming environment [8]. CollabVS enables distributed developers to send an instant message to each other within the Visual Studio programming environment [24].

Due to the large number of technical approaches that make asking for help independent of distance, as shown below in Figure 1.3, it was not clear if we could contribute to this area. Thus, making asking for help independent of distance is a high hanging fruit. The only thing we can imagine is to integrate a button into the user interface of a programming environment that when pressed changes a status field that is displayed to potential helpers. Therefore, developers only have to press a button instead of writing a status or instant message.

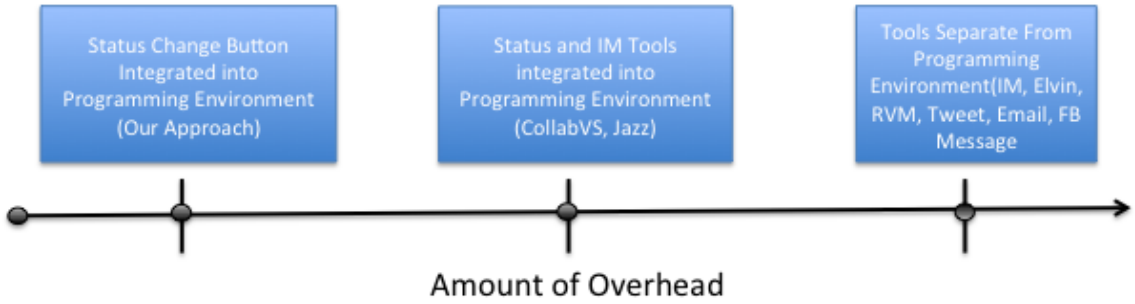


Figure 1.3: Relationship between tools that make asking for help independent of distance and the amount of overhead of each tool.

Two problems with this approach are a) developers who manually change their status are not likely to set it back, just as people forget to change their busy status in an IM tool and b) this approach alone is arguably not enough work for a thesis, even though it is the most lightweight when compared to other approaches. For all of these reasons, our focus is making passively noticing the need for help independent of distance rather than actively asking for help. Nonetheless, we have implemented a need help button into the Eclipse programming environment to enable distributed developers to ask for help. As explained later this button has to do less with promoting actively asking for help and more with passive awareness of help-need.

Given a physical coupling, one way to make passively noticing the need for help independent of distance is to virtually simulate a tighter physical coupling such as pair programming. This should give developers who are further apart, such as distributed teammates, the feeling of "being there" in the same room. One way to go towards being there is to have a potential helper watch two screens: one of a developer and the other of that developer's screen. They could use this information to determine when developers are having difficulty. However, this approach does not scale, because helpers can be aware of, at most, only a few developers.

A more scalable approach is to use video walls, walls that show the activities of remote teammates in one room, which simulate radical co-location [1]. However, as in radical co-location, teammates may miss opportunities to help each other.

A fundamental problem of both of these approaches is that they go towards "being there." "Being there" gives collaborators the feeling of being face-to-face. Hollan and Stornetta have argued that if collaboration technology is to be successful, it should go "beyond being there" by providing capabilities not available in face-to-face interaction [28]. In our case, this means creating a mechanism that detects when developers are having difficulty and communicates this to teammates, making teammates aware that developers need help.

1.2 Baseline Approaches

No work has addressed automatic difficulty detection, but there are three obvious approaches. These approaches are a) a randomized approach, which predicts one label 50% of the time and the other label 50% of the time, b) a modal approach, which always predicts the label that occurs most often, and c) a data distribution approach, which makes predictions based on the distribution of labels (e.g. making progress, having difficulty). In this dissertation, we compare the baseline approaches to the programming-activity difficulty-detection component, the multi-modal difficulty-detection component, and the difficulty-level and barrier detection components. More specifically, we compare the true positive rate and true negative rate of each baseline to the true positive rate and true negative rate of the programming-activity difficulty-detection component. The reason we do not compare the false positive and false negative rate is that these rates can be computed using the true positive and true negative rates.

1.3 Evaluation Metrics

In our work, the true positive rate identifies how often difficulty detection modules correctly predicted developers were having difficulty, the true negative rate identifies how often these modules correctly predicted developers were making progress, the false negative rate identifies how often these modules predict developers are making progress when they are actually having difficulty, and false positive rate indicates how often these modules predict developers are having difficulty when they are actually making progress. The following equations specify these metrics more precisely.

The true positive rate (TPR) is:

$$\frac{\#\ of\ true\ positives}{(\#\ of\ true\ positives + \#\ of\ false\ negatives)}$$

The true negative rate (TNR) is:

$$\frac{\#\ of\ true\ positives}{(\#\ of\ true\ positives + \#\ of\ false\ negatives)}$$

The false negative rate (FNR) is:

$$\frac{\#\ of\ false\ negatives}{(\#\ of\ false\ negatives + \#\ of\ true\ positives)}$$

The false positive rate (FPR) is:

$$\frac{\#\ of\ false\ positives}{(\#\ of\ false\ positives + \#\ of\ true\ negatives)}$$

1.4 Thesis

It is our thesis that automatic difficulty detection is possible and useful. In particular, our thesis verifies the following sub-theses:

I. *Programming Activity Difficulty Detection Sub-Theses (Sub-thesis I):* It is possible to develop an approach that a) uses developers' interactions with their programming environment to determine whether developers are having difficulty with their task and b) performs better than baseline measures.

II. *Implementation Sub-Theses (Sub-thesis II):* It is possible to develop a common set of difficulty detection modules for different programming environments that have significantly fewer lines of code than difficulty detection modules written specifically for each programming environment.

III. *Multimodal Difficulty Detection Sub-Theses (Sub-thesis III):* It is possible to develop an approach that a) combines programming activity and body posture recognition to predict when developers are having difficulty with their tasks and b) has greater accuracy and a lower false negative rate (predicting stuck) than existing approaches that only use programming activities to determine when developers are having difficulty with their tasks.

IV. *Context Awareness Sub-Theses (Sub-thesis IV):* Replaying the programming actions of developers who are stuck takes potential helpers longer to decide if they can offer help, but potential helpers prefer replaying programming actions to not having the ability to replay them.

V. *Difficulty Level and Barrier Detection Sub-Theses (Sub-thesis V):* It is possible to develop an approach that, using developers' interactions with their programming environment, a) automatically determines the barrier that is blocking programmers from making progress, b) automatically determines the level of difficulty

programmers are having with their tasks, and c) performs better than baseline measures.

VI. *Field Study Sub-theses (Sub-theses VI):* It is possible to build a difficulty detection tool that is successfully used to offer help to students.

1.5 Definition of Having Difficulty

As having difficulty is a human characteristic it is difficult to define and measure. A variety of prior work in detecting human characteristics such as frustration and interruptibility has also faced this problem. Despite this obstacle, prior work has been successful in predicting human characteristics. In particular, Fogarty et al. faced this issue while developing a tool that uses developers' interactions with the programming environment to determine if they are interruptible. Their solution was to randomly interrupt users to determine how interruptible they were. We cannot use this approach, as it is likely that no random interruption would find a developer is having difficulty – the results of our studies show that having difficulty is an exceptional event. Therefore, an alternative approach is to allow participants to report when they are having difficulty. Kapoor et al. use this approach to allow students to indicate their frustration level.

In this thesis, we use a variation and extension of Kapoor's approach, which is to allow a) developers to correct an incorrect difficulty status and b) observers' to report when developers are having difficulty. Having difficulty is developers' or observers' perceptions that developers are making slower than normal progress. We use both of these perceptions as ground truth. The argument for using developers' perceptions is that they are programming and perhaps are in the best position to know whether they are having difficulty. Observers cannot read the minds of developers to actually determine if they are having difficulty. On the other hand, people tend to

underestimate their problems [52]. This means that developers may not always admit when they are having difficulty, which is the reason for using observers. In some studies, we only use developers as ground truth, in some, we only use observers as ground truth, and in some, we use both. In field studies, we only use developers as ground truth because there were no screen recordings for observers to view. However, in lab studies, there are screen recordings; therefore, we use both observers and developers. There were some lab studies where we only used observers. More specifically, in the difficulty level and barrier detection lab studies, we only used observers because developers do not indicate their difficulty level or barrier even though we provided tools that allowed them to indicate this information.

1.6 Summary

In this chapter, we introduced motivations and techniques for increasing the amount of help in software development. Each of these techniques had benefits and limitations, which led to our focus, automatic difficulty detection. It is our thesis that automatic difficulty detection is possible and useful. To provide evidence to support this thesis, we decompose it into several sub-theses and provide evidence for each sub-thesis in the following chapters. The chapters are organized as follows. Chapter 2 describes in greater detail, the motivation and techniques for increasing the amount of help in software development. Chapter 3 provides evidence for sub-theses I and II. Chapter 4 provides evidence for sub-thesis III. Chapter 5 provides evidence for sub-thesis IV, V, and VI. Finally, Chapter 6 presents conclusions and future work.

Chapter 2. Comparison with Related Work

2.1 Overview

In this chapter, we present previous techniques and motivations to promote help. These motivations and techniques come from a combination of computer science fields: collaborative software engineering, data mining, human-computer interaction, and computer science education.

The rest of this chapter is organized as follows. Next, we present previous work that provides the motivation for encouraging programmers to help each other. Following this, we discuss techniques to promote help. Then, we present work on making developers aware that their teammates need help. Finally, we end with a summary.

2.2 Motivation for Encouraging Programmers to Help Each Other

A variety of previous work shows that increasing the amount of help in a software development team also increases the productivity of the team.

Herbsleb et al. [27] conducted a field study to compare the productivity of co-located teams (teams that are in the same building but sit in different cubicles) and distributed teams (teams that are in geographically dispersed locations). Productivity was measured as the amount of time it takes to fix bugs, update software, and add new functionality. To measure this time, they used data from a change management system and a survey. The data from the change management system was taken from modification requests, which are requests for new

functionality, bug fixes, or software updates. A modification request has a record of the date of the request, the date a change was made to the code, the requestor. They created two measures using the date of the request and the dates a change was made to the request. The first measure, work interval, was the difference between the date of the first code change and the date of the last code change. This measure was an approximation of how much time it took to complete the work. The second measure, full interval, was the difference between the date the request was made and the date of the last change. The full interval includes the work interval and the time taken to assign the work and for developers to start the work. The data from the survey was participants' approximation of the number of times their work was delayed in the last month and the average duration in days of these delays. Participants gave approximations for both participants' local and remote teammates. Figure 2.1 shows the results from the modification data.

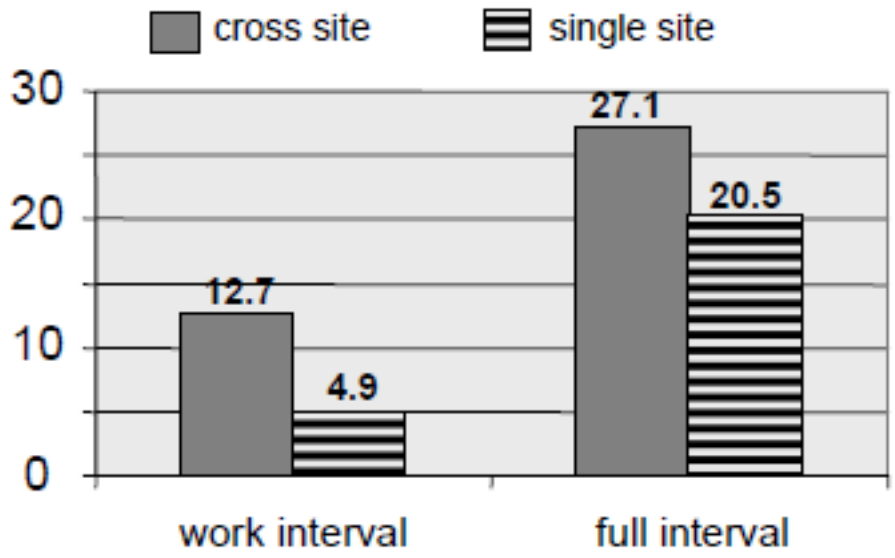


Figure 2.1: Comparative statistics on productivity measures taken from [27].

This figure shows that the work interval modification requests that involved distributed teams took 2.5 times as long to complete than modification requests that only involved co-located teams. Their survey data also support these findings. On average, participants reported

fewer delays when working with distributed teammates, but the length of these delays were on average 1.5 days longer. These results show that co-located teams were more productive than distributed teams. To determine which factors caused delays in productivity, they surveyed members of distributed and co-located teams and found that several breakdowns made it difficult to find co-workers, get timely information about plan changes, have clearly formed plans, agree about plans, be clear about assigned tasks, and have co-workers provide help beyond the call of duty. However, the perception of received help was the only factor that correlated with productivity. An earlier study by Herbsleb and Grinter suggests one reason developers had this perception. They found that distributed developers are less comfortable asking each other for help because they interact with each other less than co-located developers [25]. If this is true, then as people get more opportunities to interact with each other, their productivity should increase.

Teasley et al. found that this indeed was the case. They conducted a field study to compare the productivity of six radically co-located teams, teams that are in single building that work in a war-room or bull-pen (Figure 2.2), to teams at the same company who were spread out in different cubicles (co-located teams) [55]. Productivity was measured as function points per staff month and cycle time. A function point is a standard software metric that measures the size of a software project by adding the weighted sum of inputs, outputs, logical files, queries, and interfaces. A staff month is the average available work hours in a month per person. Cycle time is the number of months from start of the project to the time when the project is completed. They normalized the cycle time for the size of the projects: the number of months per 1000 function points. The paper did not explicitly indicate whether co-located teams and radically co-located teams were the same size.

Figure 2.2: One of the war-rooms taken from [55].

Table 2.1 shows the productivity results for both the radically co-located and co-located teams (company's baseline). Radically co-located teams produced twice as many function points per staff month as co-located teams and their cycle time was one-third lower than the company baseline. These results show that radically co-located teams were more productive than co-located teams. One reason was that radically co-located teammates saw and overheard each other's activity [55], which enabled them to interject and provide help when they noticed that someone was having difficulty. More concretely, if someone was having difficulty with some aspect of code, another developer in the war-room that is "walking by [and] seeing the activity over their shoulders, would stop to provide help." However, if developers are not nearby, they may miss opportunities to help. Conversely, constantly monitoring developers to see if they need help may increase their interactivity to an even greater degree.

Table 2.1: Comparative statistics on productivity measures taken from [55].

	Radically co-located teams	**Co-located teams (company baseline)**
Function points per staff month (Higher is better)	29.49	14.35
Cycle Time (lower is better)	7.64	19.47

This is exactly what happens in pair programming where two programmers sit next to each other, with one programmer, called the *driver*, writing code, and the other programmer, called the *navigator*, offering help. Cockburn and Williams conducted a lab study to determine whether student pair programming teams were more productive than solo student programmers [9]. Eighteen students programmed alone while 28 students completed their assignments with a partner. They measured productivity across three assignments using the following equation:

$$\frac{1}{\text{assignment completion time} + \text{bug fix time}} (1)$$

Their results show that after taking into account "jelling" time, the time taken for pairs to get used to each other, pairs spent 15% more time on their programs than individuals. However, pairs' resulting code had 15% fewer bugs. The percentage of bugs was equal to the percentage of the instructor's test cases students' code passed. Figure 2.3 shows the percentage of the instructor's post-development test cases the students' code passed for each program. The 15% increase in assignment completion time is recovered in the reduction of bugs. More specifically, assuming certain values for the amount of bugs programmers inject per line of code and the amount of time spent to fix these bugs, the study claims that pairs are more productive than solo

programmers. The study also claims that pair programmers learned different coding strategies from each other and the design of their code improved. Williams and Cockburn report that "pairs often find that seemingly "impossible" problems become easy or even quick, or at least possible, to solve when they work together [9]".

Together, these studies provide some evidence that a) developers' productivity increases when they help each other and b) as distance increases, help is offered less. These results can be taken as a given of our research. One way to address this problem is to make help independent of distance. There are two ways to do so: reduce the need for help and promote help.

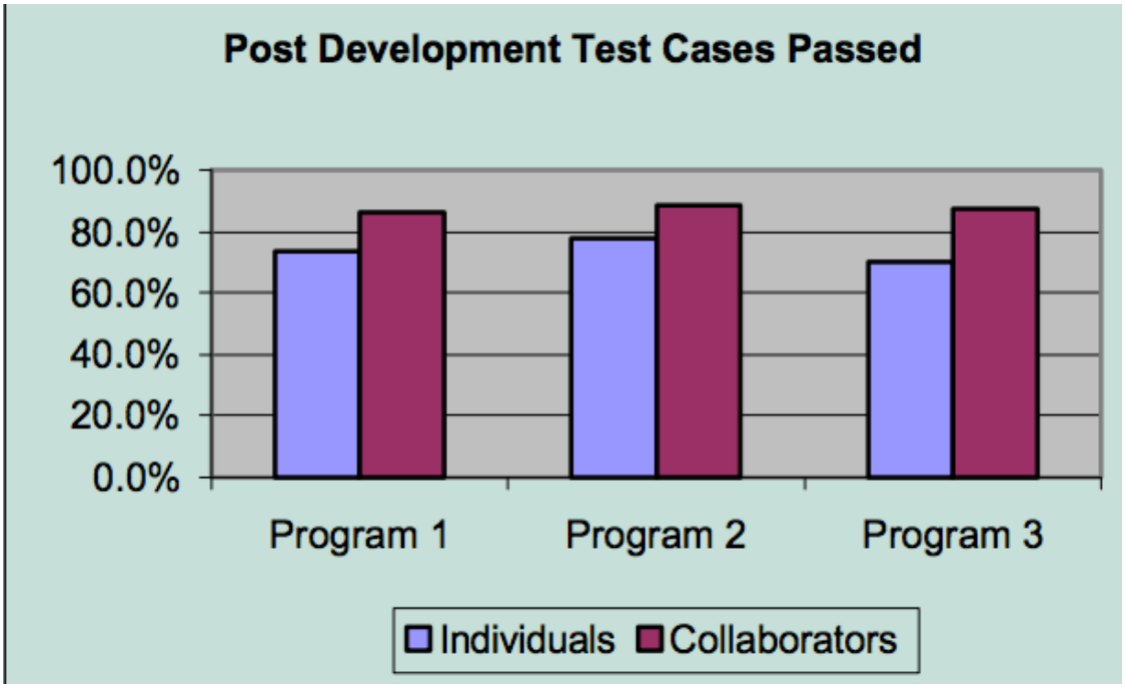


Figure 2.3: The percentage of post-development test cases solo and pair programmers passed on three assignments.

2.3 Techniques to Reduce the Need for Help

Project management/process solutions reduce the need for developers to get help by structuring how software is designed and developed. Herbsleb et al. propose two such solutions [25]. The first is to design software components to be as modular as possible and assign only

components that do not depend on each other to different geographical locations. The second is to only split the development of only well-understood software components among teams.

Social and technical solutions promote help. Social solutions enable developers and potential helpers to interact with each other without using special help-promotion tools. These solutions are important because, as mentioned above, distributed developers are less comfortable asking for help than co-located developers. Herbsleb et al. proposed a social solution that overcomes this problem [25]. Their solution was to bring distributed teammates who need to communicate together at the beginning of projects. These initial meetings enable distributed teammates to get to know each other, which could make them more comfortable with each other. Technical solutions enable distributed developers and potential helpers to use special help-promotion tools to interact with each other. There are many technical solutions that promote both actively asking for help and passively noticing the need for help.

Figure 2.4: A screenshot of the ticker tape taken from [10].

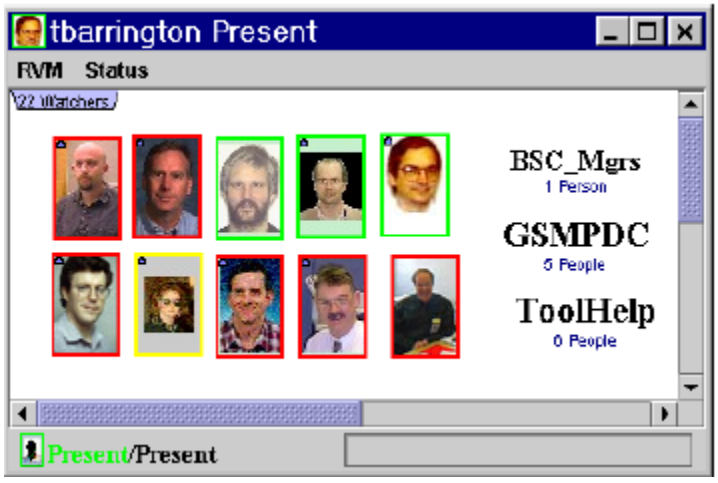


Figure 2.5:A screen shot of the Rear View Mirror Tool taken from [16].

2.4 Techniques to Promote Actively Asking Help

One way to promote actively asking for help is to enable teammates to use a ticker tape, implemented in Elvin, to post messages that are continuously shown to collaborators [18].In their work, a ticker tape, shown in Figure 2.4, is a resizable rectangular window that displays colored messages that scroll from right to left. Teammates can use a ticker tape to write messages that indicate a need for help. Herbsleb et al. also created a similar tool shown in Figure 2.5, Rear View Mirror (RVM), which provides distributed teammates two ways to ask for help [26]. Teammates can a) change a status message that is displayed to collaborators after some period of keyboard and mouse inactivity or b) send an instant message to collaborators. The features of these two early research tools, Elvin and Rear View Mirror, can also be found in modern state of the practice tools. For example, programmers can send an email, tweet, instant message, or Facebook message to potential helpers, or ask a question on question and answer sites such as Stack Overflow.

These tools, including Elvin and Rear View Mirror, require developers who are interacting with a programming environment to switch to a separate tool to ask for help. A more lightweight approach is to provide the capabilities of the tools within the user interface of the programming environment. Both Jazz and CollabVS use this approach. Jazz enables distributed developers to a) change a status message that is displayed to collaborators and b) send an instant message to each other within the Eclipse programming environment [8]. CollabVS enables distributed developers to send an instant message to each other within the Visual Studio programming environment [24].

However, there are several apparent problems with each of the previously mentioned approaches. As mentioned before a study investigating the coordination and communication

breakdowns that occur in distributed software teams found that developers are less comfortable asking remote rather than co-located software developers for help [25]. This lack of trust occurs because distributed software teams do not interact with each other as often as co-located teams. This result is consistent with previous work [13] which found that subjects doing side-by-side programming were willing and found it socially acceptable to interrupt a partner to ask for help, while in radically-co-located programming, they were afraid of disturbing the same partner. Moreover, studies show students and new programmers are late to use help [3,12] and programmers often exhaust other forms of help before contacting a teammate. Even those who are willing to manually change their status are likely to not set it back, just as people forget to change their busy status in an IM tool or turn off the "call steward" light in a plane. One way to address these problems is to have teammates passively notice the need for help.

2.5 Techniques to Promote Passively Noticing the Need for Help

By *passive*, we mean that developers do not have to take explicit steps to communicate that they need help, but observers monitor developers to determine if they need help. One way to make passively noticing the need for help independent of distance is to enable a pair of developers to passively monitor the progress of each other, using local [44] or distributed side-by-side programming [13]. Another approach is to use passive awareness mechanisms. Several passive awareness mechanisms have been developed to enable collaborators to become aware of a teammate's activities. These mechanisms can be classified into two categories: syntactic and semantic. Such mechanisms provide information to remote collaborators that co-located collaborators could observe by sitting next to their teammates.

Mechanisms in the syntactic category do not make inferences about the data they capture. Mechanisms in the semantic category make inferences about the data they capture. The

PortHoles [17] system developed by Dourish and Bly is in the syntactic category. PortHoles periodically communicates images of users to their collaborators with the goal of increasing increase awareness in distributed teams. One participant inferred interruptibility of his teammates by looking to see if they were talking to someone, while another participant inferred progress of a distributed student by noticing that he worked many late nights. Being aware of the teammate's presence led them to become aware of the progress on his dissertation.

One kind of passive awareness is workspace awareness [23], which is knowledge of another person's interactions with a shared workspace. Gutwin and Greenberg argue that this information is useful for several types of activities: mixed-focus collaboration -- collaboration where a distributed team member switches between both group and individual work-- simplifying communication, coordination, anticipation of collaborators' actions, and helping collaborators with their tasks. Workspace awareness supports distributed mixed-focus collaboration because it enables collaborators who are working individually to monitor and keep track of the rest of the group's activity. Similarly, it simplifies communication because collaborators can view information about a task without the need to explicitly ask teammates for that information. Collaborators monitoring the activities of others can also use this information to anticipate when teammates will have a need and fulfill it. Finally, collaborators can use information about teammates' tasks to determine if they should offer help and the type of help that is required.

One way to provide information about developers' tasks is to share a) programming artifacts such as current files being edited and b) programming activities such as debugging. CollabVS [24] provides collaborators with this information, which could be used with other related project information to help determine if a teammate is stuck. For example, [24] gives a

scenario in which Bob, on seeing Alice stuck on debugging a particular class, deduces she could use help, and offers it.

Providing virtual channels that give distributed users the feeling of "being there" in a single location is an important goal of CSCW. However, Hollan and Stornetta have argued that if CSCW is to be truly successful, it should go "beyond being there" by providing capabilities not available in face-to-face interaction [28]. Dewan [16] surveyed several software engineering tools, which provide "beyond being there" capability to collaborators. For example, systems by Schummer and Haake, Dewan and Hegde, and Sarma et al. automatically try to determine if the activities of members of a team conflict and make collaborators aware of these conflicts [15,48,49]. One such capability not provided by the previously mentioned systems [15,48,49], is to infer when developers need help. Previous work has [4] argued that inferring when developers need help could increase useful group awareness among large development teams, and enable new programmers to get help from their mentors [5,12].

2.6 Automatic Detection of Difficulty

Previous research has taken a step toward developing mechanisms that explicitly or implicitly infer developers' progress status. A status is a semantic inference of interest to collaborators about the state of developers such as having difficulty, being interruptible, or the level of interruptibility. Each work uses a different technique to make this inference. First, we present a technique that uses non-standard equipment, second, one that uses logs from developers' interactions with a version control system, newsgroup, and wiki; third, one that monitors programming environment activities; fourth intelligent tutoring systems, and finally one that monitors changes in code.

2.6.1 Mining Interactions with Non-Standard Equipment

Kapoor et al. [35] use classification, a data mining technique, to infer when kids are frustrated. The goal of classification is to associate labels (in their case frustrated or not frustrated) with new input data based on known labels of preexisting data. The input data are sets of attributes or features that are in some way correlated with the labels. In the case of the labels, frustrated and not frustrated, an attribute could be the amount of pressure put on a mouse. A general approach to classification consists of four steps. First, training data, preexisting data with known labels, is gathered. Next, training data are used as input to an algorithm to create a function or model that identifies the relationship between attributes and labels. Then, new attributes are used as input to the model, which outputs the predicted values of labels. Finally, the performance of the model is evaluated. There are several ways to measure the performance of models. One such measure is accuracy, the number of correct predictions divided by the total number of predictions.

Kapoor et al. [35] used this general approach to predict when kids were frustrated. Their intuition was that frustration would correlate with changes in body posture, facial expressions, pressure applied to a mouse, and skin conductance. To test their intuition, they collected these attributes from cameras, posture seating chairs, pressure mice, and wireless Bluetooth skin conductance tests while 24 middle school students solved a Tower of Hanoi problem. To create training data, participants clicked on an "I'm frustrated" button. When participants did not click the "I'm frustrated" button, they were labeled as not frustrated.

Table 2.2: Example of attributes and labels.

Amount of Pressure on Mouse (Attributes)	Labels
.9 N	I'm frustrated
.1 N	I'm not frustrated
1.0 N	I'm frustrated
.2 N	I'm not frustrated
.3 N	I'm not frustrated

They trained several classification algorithms to output models. One such algorithm is Support Vector Machines (SVMs), which find a hyper-plane, a plane in n-dimensional space, which best separates records into groups. To illustrate, let us assume that Table 2.2 shows an example of attributes taken from the equipment with mouse pressure data as an attribute and "I'm frustrated" or "I'm not frustrated" as a label. Figure 2.6 shows a hyper-plane based on Table 2.2 that separates the mouse pressure data. Data on one side of the hyper-plane represents the "I'm Frustrated" label and data on the other side represents the "I'm Not Frustrated" label. Many hyper-planes could be drawn to separate the data, but the best one represents the largest separation between groups of data and has the greatest distance between the nearest data point on each side of the hyper-plane.

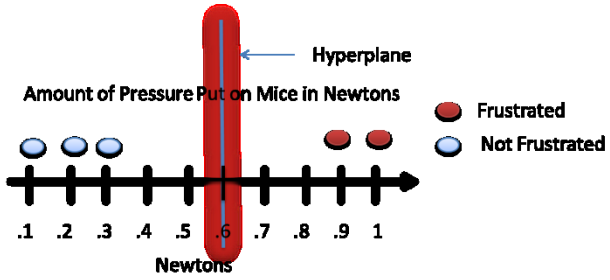


Figure 2.6: Hyper plane that separates mouse pressure data into two groups.

To train SVMs, they aggregated data from all participants except the participant whose label they were trying to predict. The exclusion was meant to test if SVMs trained by one set of participants could be used to predict the label of another. This approach was used to predict the label of each participant. To determine the accuracy of the model's predictions, they summed the number of times the algorithm predicted the correct label and divided the sum by the total number of participants. Their results show that SVMs algorithm was accurate 70% of the time. A problem with their approach is the overhead of using non-standard equipment.

2.6.2 Mining Interactions with Components

An alternative approach to infer difficulty is to log developers' interaction with some component of their systems. Liu and Stroulia take an important step in this direction. They developed a tool that monitors students' interactions with CVS and newsgroups to calculate the workloads and work-status of students [40]. Several CVS operations such as adding a file, checking out a file, removing a file, and modifying a file were used to determine students' workloads. In particular, the number of file modifications was used to compare the workloads of two student groups to infer which group preformed the most work. This information could potentially be used to determine if students were having difficulty, but this awareness would be provided, not when they had the difficulty, but later, when they checked in the files or posted to newsgroups. Developers may struggle for a long time before they take these actions, and for certain problems, would not expect a response from the Internet. Providing earlier awareness requires mining interactions with the programming environment.

2.6.3 Mining Interactions with Programming Environments

Previous work has explored, to some degree, the idea of mining developers' interactions with their programming environment. Fogarty et al. mined developers' interactions with their

programming environment and used classification to determine whether they were interruptible [20]. Their intuition was that a) the longer it took developers to respond to an interruption, the less interruptible they were and b) the specific actions developers perform right before they were interrupted would correlate with being interruptible. To test their intuition, they created an Eclipse plug-in to log developers' actions and randomly interrupted developers while they were performing maintenance tasks. Interruptibility was measured using the following equation:

$$\text{time users' acknowledged a notification} - \text{time a notification appeared on users' screen} \quad (2)$$

They used this information along with developers' actions to create training data (labels and attributes). To create labels, they clustered the amount of time it took for developers to respond to an interruption using the expectation–maximization (EM) algorithm as implemented in the WEKA toolkit [59]. Given a set of values (the time taken to respond to an interruption) and a number of clusters to produce, the algorithm computes the mean and standard deviation of each cluster and the probability that each value belongs to a cluster. They experimented with the number of clusters and decided to use three clusters because the standard deviation of one cluster was large when using two clusters. This large standard deviation indicated that some values in that cluster were significantly further away from the mean than other values. Therefore, they increased the number of clusters to three with the hope that those values that were significantly further away from the mean would form their own cluster. This was indeed the case. They found that three clusters represented a better division of the data and that four clusters offered no improvement over three. The three clusters represented the labels: interruptible, engaged, and deeply engaged. Developers who were interruptible/engaged/deeply engaged responded on average within 2.2/6.9/43 seconds to an interruption.

To determine what actions indicate that developers are interruptible, engaged, or deeply engaged, they analyzed the logs of developers' actions to compute attributes. The goal of their analysis was to a) find actions that occurred within some time prior to an interruption and b) determine how frequent these actions occurred within that same time. They used these actions and their frequency as attributes. Some of their attributes are the number of edit events that occurred 15 seconds prior to an interruption, whether the Eclipse programming environment lost focus 5 seconds prior to an interruption, and whether developers stopped typing 15 to 20 seconds prior to an interruption. These attributes were combined with their labels to form training data. The labels engaged and interruptible were combined to form one label: engaged/interruptible.

They input this training data into the naïve Bayes classification algorithm, which computes the conditional probability for all possible values of labels and uses this probability to predict labels. The predicted label is the one with the highest probability. For example, in their case, the algorithm computed the probability that a label is engaged/interruptible and deeply engaged given that developers made no edits, one edit, or multiple edits 15 seconds prior to an interruption. To train the naïve Bayes algorithm, they used a standard technique, ten-fold cross-validation, which executes 10 trials of model construction and splits the logged data so that 90% of the data are used to train the algorithm and 10% of the data are used to test it. The accuracy of the model is computed during each test phase and averaged to determine the final accuracy of the model. Their results show that the naïve Bayes algorithm was accurate 71.8% of the time and had 14.7% false negatives and 13.5% false positives. Fogarty et al. did not explore the use of the interruptibility status in the context of collaborative software development, the subject of this thesis.

However, several tools [31,41,42,46] have explored logging programming activities and computing metrics using these activities. One such metric is the time spent programming, which could be used to infer when developers have difficulty. For example, if student programmers are expected to finish an assignment in ten hours, but one student is taking longer to complete the assignment, one can infer that the student is having difficulty. There are several ways to compute the time spent programming.

2.6.4 Computing Time Spent Using Programming Environment Events

McKeogh and Exton developed Eclipse Watcher [41], an Eclipse plug-in that logs students' programming activities such as editing (insertions and deletions) code and navigating between files, the timestamp of these programming activities, the line number of a specific edit, and the file being edited. They used this logged data to compute the time spent editing on each line of code and the time spent editing and navigating. The goal of their work was to determine if these metrics could be used to determine the lines of code and files that are causing developers to have difficulty. This information could be used to find complex code and refactor it.

To compute the time students spent editing and navigating, the timestamp of the previous programming activity is subtracted from the timestamp of the current programming activity. A problem with this approach is that periods of inactivity could lead to inaccurate calculations of time spent. To overcome this problem, the timestamp of the current programming activity is disregarded if it occurs more than five minutes after the timestamp of the previous programming activity. This threshold was chosen arbitrarily. The time spent programming equals:

$$\sum_{i=1}^{n-1} (x_{i+1} - x_i), if\ (x_{i+1} - x_i) > y\ mins \quad (3)$$

where n is the number of programming activities, x is the timestamp of a programming activity, x_i is the i^{th} timestamp of a programming activity, x_{i+1} is the $i^{th} + 1$ timestamp of a programming activity, and y is the number of minutes. To illustrate, consider the example log we created shown in Table 2.3 and Equation 3. The total time spent editing and navigating is 12 minutes.

Table 2.3: Our example of a programming activity log captured by Eclipse Watcher [41].

Id	Timestamp	Programming Activity	Line Number	File Name
1	3:40 pm	Editing	**1**	Test.java
2	3:41 pm	Navigating	**n/a**	**n/a**
3	3:50 pm	Navigating	**n/a**	**n/a**
4	3:51 pm	Editing	**1**	Test.java
5	3:52 pm	Editing	**1**	Test.java
6	3:53 pm	Editing	**2**	Test.java
7	3:56 pm	Editing	**2**	Test.java
8	3:59 pm	Navigating	**n/a**	**n/a**
9	4:10 pm	Editing	**2**	Test.java
10	4:13 pm	Editing	**2**	Test.java

This paper did not give any indication of how to compute the time spent editing on each line of code in a file, but it provided a table, Table 3 taken from [41], which shows the amount of time a student spent editing on each line of code in a file. Rows represent a file and columns represent five lines of code. Based on Table 2.4, we assume that time spent on each line of code is computed by a) modifying Equation 2 to only subtract the timestamps of edit events if both occur in the same file and on the same line, b) summing the computed time of edits with the same file name and line number, and c) summing the result of the previous step for every five lines of code. For example, using Table 2.3, if we perform steps (a) and (b), the total time spent editing code on line 1 is 2 minutes and on line 2 is 6 minutes. If we perform step (c), the total time editing lines 1 through 5 is 8 minutes.

Table 2.4: Amount of time editing each line of code [41]. Rows represent a file and columns represent five lines of code.

	1	6	11	16	21	26	31
DoodlePad.java	0.040367			0.120317		1.1026	0.132817
DrawPad.java	0.081517		0.527083	0.060933	1.059633	1.17085	0.014067
OptionPanel.java	0.707017			0.346617	1.085667	1.2758	0.24895
ColorChanger.java	1.13305			0.038283	3.116117	3.9883	0.124217
JColorChooser.class	1.920833						

To evaluate their tool, they logged three student programmers with varying levels of experience while working on a maintenance task. Their preliminary results show that their tool could be used to explain the navigation behavior of the students. More specifically, the student with the most experience spent the least amount of time navigating code. One possible reason is that the most experienced student understood the code faster than the inexperienced students.

Norris et al. [46] also used time spent to understand student programmers' difficulty, but instead of focusing only on students' navigation behavior, they focused on students' compilation

and edit behavior. To realize this goal they created ClockIt [46], an extension of the BlueJ programming environment that a) monitors compilations, edits (insertions and deletions), and the number of lines of code and comments written, b) computes the number and percentage of failed compilations, time spent on an assignment, and project growth, and c) graphically displays this information to students and instructors through a web interface. This paper gave no indication of how time spent is computed, but we assume they use Equation (2). Project growth is computed daily using the following equation:

$$Project\ growth = \frac{\#\ of\ lines\ of\ code + \#\ of\ lines\ of\ comments}{day} \quad (4)$$

To determine the usefulness of these metrics, they logged and computed metrics for 75 students working on lab assignments. However, they only reported the results of three of those students. Their results show that a) the student with the least amount of time spent writing code wrote the least amount of code and got the worst grade of the three students and b) two of the students spent nearly the same amount of time writing code, but the student with the best grade wrote less code. The student with the best grade encountered fewer compiler errors and did not encounter the same types of compiler errors as the other two students. These metrics could be used to answer questions common to instructors teaching a CS1 course. Examples of questions are: a) how much time are students spending on their assignments? and b) what types of errors do students make?

Retina [42], a related tool, provides preliminary answers to these questions. It monitors students' compilations and also their run time errors to a) estimate the amount of time students spent on assignments, b) predict how much time students would take on assignments, c) compute the total number of compilations, d) compute the total number of compilation errors, e) compute

the percentage of successful compilations, and f) compare these metrics for an individual student against the class's average of these metrics.

Time spent is estimated by grouping compilation events that occur within a fixed time, e.g. 30 minutes, into individual programming sessions and summing the time spent on each individual session for an assignment. The time spent programming for a session can be computed using Equation 3 if we set y equal to a fixed time. Thus, the amount of time spent programming for all sessions equals:

$$\sum_{i=1}^{n}(x_i) \quad (5)$$

where n is the number of sessions, x is the time spent programming computed for a session using Equation 1, and x_i is the i^{th} session. The example log we created in Tables 2.5a, 2.5b, and 2.5c show that a) a student had two programming sessions for an assignment , b) the same student spent 80 minutes programming during the first session (Table 2.5a) and 25 minutes during the second (Table 2.5b), and c) the sum of both programming sessions for an assignment is 105 minutes (Table 2.5c).

Table 2.5a: Student's programming activity log for session 1.

Programming Activity	Timestamp
compilation event	3:40 PM
compilation event	3:45 PM
compilation event	3:54 PM
compilation event	4:00 PM
compilation event	4:15 PM
compilation event	4:38 PM
compilation event	4:50 PM
compilation event	5:45 PM
Total Time Spent:	80 minutes

Table 2.5b: Student's programming activity log for session 2.

Programming Activity	Timestamp
compilation event	5:49 PM
compilation event	6:04 PM
compilation event	6:13 PM
compilation event	6:14 PM
compilation event	6:55 PM
Total Time Spent:	25 minutes

Table 2.5c: The sum of the time spent for sessions 1 and 2 on an assignment.

Time spent Session 1	80 minutes
Time spent Session 2	25 minutes
Total Time	105 minutes

Retina estimates the amount of time it would take a student to finish an assignment based on how much time (a) students in a previous offering of a course took on their assignments and (b) students in a current offering of the course took on previous assignments. Let S_c denote all students in the current semester, S_p all students in the previous semester, j the student whose time is being estimated, and n the assignment for which time is being estimated. The estimated amount of time it would take j to finish n is computed by a) averaging the amount of time S_c took on previous assignments, b) ranking j's average time spent on previous assignments with respect to S_c, c) averaging and ranking the amount of time S_p spent on all assignments, and d) finding the time it took similarly-ranked S_p to complete n. For example, a student with Id of 2 in Table 2.6a a) ranked 2nd in terms of average time spent on all previous assignments, b) spent a similar average time on assignments as a student with an Id of 3 in a previous semester (Table 2.6b), and c) is predicted to spend approximately 107 minutes on an assignment, which is the same time that a student with an Id of 3 spent on that assignment (Table 2.6c).

To evaluate Retina, they a) logged 48 students doing class assignments and b) asked a small number of students and three instructors to comment on the usefulness of the tool. Students liked using the tool because it helped them compare themselves to their classmates and

determine that their classmates also struggled with the material. Instructors used to tool to a) determine the types of compilation errors that caused students to have difficulty, b) anticipate the questions students would ask, and c) tailor their help based on students' compilation errors. They focused on compiler errors because student programmers often struggle with getting their programs to compile correctly [37].

Table 2.6a: Average time students spent on all assignments.

Current Semester	
Student Id	Average time spent on previous assignments
1	165 minutes
2	175 minutes
3	177 minutes
4	179 minutes

Table 2.6b: Average time students taking a previous offering of a course spent on all assignments.

Previous Semester	
Student Id	Average time spent on previous assignments
1	150 minutes
2	165 minutes
3	172 minutes
4	185 minutes

Table 2.6c: The time students spent on an assignment in a previous offering of the course.

Previous Semester	
Student Id	Time spent on assignment in previous semester
1	95 minutes
2	105 minutes
3	107 minutes
4	113 minutes

2.6.5 Automatically Determining Compiler Error Difficulties

One reason students struggle is compiler errors can be cryptic making them difficult to understand. To determine the amount of difficulty students have with compiler errors, Jadud created the Error Quotient (EQ) metric [32]. The intuition behind this metric is that students in introductory programming courses spend the majority of their time editing and compiling code to

make their programs syntactically correct. The EQ metric could be used to determine when students are having difficulty. It is computed using the following algorithm:

Given a session of compilations, e_1 through e_n:

(1) Pair consecutive compilations (e_1, e_2), (e_3, e_4), (e_5, e_6).....(e_{n-1}, e_n)

(2) Assign a numerical penalty to pairs with compilation errors

   a. Assign a penalty of 8 if both pairs have a compilation error

   b. Assign an additional penalty of 3 if both pairs have the same compilation error

(3) Divide the score assigned to each pair by 11 (the maximum value possible for each pair)

(4) Sum the scores for each pair of compilation

(5) Divide the sum by the total number of pairs

They determined numerical values for penalties by experimenting with multiple numerical values. A perfect error quotient is 0.0, which means students were able to fix syntax errors as they had them. Conversely, an error quotient of 1.0 implies that each time a student compiled code, the compilation ended in the same syntax error. To determine how well the EQ metric indicated the amount of difficulty students had with their assignments they, a) extended the BlueJ programming environment to log compilations and compilation errors, b) logged 96 students working on homework assignments, c) computed the EQ metric for each student, and d) determined if there was a correlation between students' EQ metric and the average grade of homework assignments and exams. Their results show that there is a correlation between the EQ metric and the average grade of homework assignments and exams.

2.6.5 Difficulties with Logic Errors

After student programmers get their program syntactically correct, they may also have problems getting their programs to output the correct information. We refer to these types of problems as *logic errors*. One reason students struggle with logic errors is that they do not give a warning like compiler errors. Therefore, students have to a) detect that the output is incorrect, b) find the code that is responsible for the error, c) understand the cause of the error, and d) modify the code to remove the error. This process is debugging and can often be difficult and time-consuming [58]. One way to ease the difficulty and reduce the amount of time students and even experienced developers spend debugging is to use debugging tools.

2.6.6 Using Breakpoint Debuggers to Overcome Logic Errors

One such tool, breakpoint debuggers, enables developers to a) pause a program's execution at specified lines, b) inspect the value of variables of the paused program, and c) step through the program's execution. Breakpoint debuggers are a standard part of modern programming environments, but novice student programmers may hesitate to use them or may not be aware that they exist. More importantly, these tools require developers to guess which line of a program is causing an error. If developers guess incorrectly, the tools provide them with incorrect information. One way to reduce this speculation is to help developers find the line(s) of code that is causing an error, show developers the line(s) of code that is causing an error, explain the reason for the error, and fix the error.

Ko et al. take a step towards this direction. They created a tool, the WhyLine, which enables developers to ask "why" or "why not" questions about a program's output and identify the parts of code responsible for the output [43]. Their results showed that participants who used the WhyLine completed more debugging tasks and took less time on their tasks than participants

who used a breakpoint debugger. One possible reason is that the WhyLine helped participants find the code that was responsible for an error. However, some participants still struggled with understanding the reason for their errors and thus, had trouble fixing them.

One way to overcome this problem is to anticipate the problems students will have and offer hints to students about why their programs are incorrect and how to fix them. This is exactly how Intelligent Tutoring Systems (ITS) helps students.

2.6.7 Intelligent Tutoring Systems

One example of an ITS is the LISP Tutor [2], which helps students who write code in the LISP programming language overcome compiler and logic errors. Since we have discussed an approach to detect compiler errors, we do not consider them from now on. To help students overcome logic errors, the tutor must model students' progress. It does this by monitoring students' individual keystrokes and determining whether students' input will produce the correct output. The tutor monitors individual keystrokes using a structured editor that a) only enables one line of code to be entered at a time, b) automatically balances parenthesis, and c) provides placeholders that structure methods into segments. These segments are: a) the parameters of a method, b) initialization of variables, and c) the body of the method.

To determine whether students' input will produce the correct output, the tutor must know a) what the correct output is and b) whether students have deviated from it. The LISP tutor does this by giving students small tasks to solve such as combining two lists into a single list and checking students' input against a set of rules. There are two types of rules: buggy rules, which represent the logic errors that students may make while solving a problem and correct rules, which represent the correct output. For example, if the goal is for a student to combine two lists, a correct rule would be: "if the goal is to combine $list_1$ and $list_2$ into a single list, then use the

function append", while a buggy rule would be "if the goal is to combine $list_1$ and $list_2$ into a single list, then use the function add, add is the incorrect function name, should use the function append."

If students' input follows a buggy rule, the tutor determines the exact error and offers help to students. It offers two types of help. First, it gives students hints in natural language. It constructs hints using the buggy rules and the segment of the code where students are editing. For example, if the task is to define a method that takes two numbers as parameters and returns the sum of those numbers, and students write a method that returns the difference of two numbers, the tutor would suggest that students use addition instead of subtraction. Second, after students make a certain number of errors, usually two, the tutoring module gives students the correct code piece of code, which enables them to continue solving the problem instead of staying stuck and giving up. One problem with Intelligent Tutoring Systems is that they constrain students' input, which limits their ability to explore and try different ways of solving a problem.

2.6.8 Mining Code Changes

This limitation is overcome to some extent by Piech et al. who enable students to use the Eclipse programming environment, a non-structured editor, to invoke methods that are defined by instructors and use statistical models to automatically determine the amount of difficulty students had with their assignments [47].

The goal of their work was to automatically monitor students' progress through assignments and determine where they are having difficulty. Their intuition was that a) incremental submissions of students' programs can be automatically assigned a label such as

"the student has just started", b) students transitions from label to label could be graphically modeled to show how students transition through assignments, and c) the amount of difficulty students had on assignments was based on the number of code submissions it took before they moved to the next label.

To test their intuition, they created an Eclipse plug-in to log students' code when they saved or compiled their program. Since labels were not predefined, they needed a way to a) automatically convert incremental code submissions into labels and b) predict the likelihood that students will transition to the next label. To create labels, they clustered 2000 code submissions from different students using the K-Mediods algorithm. Given n versions of code and k (the initial number of clusters to produce), they asked the algorithm to partition the code into clusters based on the median distance between versions of code. In their case, n was 2000 and k was 26. Since there is no well-known measure for determining similarity between two pieces of code, they created three measures.

The first measure, Bag of Words Difference, uses histograms to represent the frequency that key words appear in two versions of code and the Euclidean distance between two histograms as a measure of difference between the two versions of code. The second measure, Application Program Interface (API) Call Dissimilarity, is computed by a) executing students' programs to capture the sequence of API calls, b) finding sequences of API calls that do not match using the Needleman-Wunsch global DNA alignment algorithm [45], and c) using the number of sequences that do not match as the difference between two programs. The last measure, Abstract Syntax Tree (AST) Change Severity, is computed by first creating AST representations of two programs. An abstract syntax tree is a tree representation of programs where each tree node contains programming syntax. The term "abstract" means some

programming syntax is excluded from the tree such as parentheses. The next step is to determine the Evolizer change severity score, which is the minimum number of changes needed to transform an AST from one program to the AST of another program. Finally, they use the Evolizer change severity score to determine the difference between two programs.

To evaluate each measure, they a) selected 90 pairs of programs where each pair was from the same student, b) computed each measure for each pair of programs, c) recruited five experts to label each pair of programs as either similar or different based on style and functional rules given to them, and d) compared the experts' assessment to each measure. The API Call Dissimilarity and the AST Change Severity metric preformed best. Therefore, they created a weighted sum of both metrics. We refer to the weighted sum of these metrics as code distance because they determine the amount of difference between two versions of code. They used the K-Mediods algorithm to partition the code submissions clusters based on the code distance. A manual inspection of the clusters confirmed that code submissions that were clustered together were similar in functionality and intuitively made sense.

To graphically model students' transitions they used Hidden Markov Models (HMMs), which is a probabilistic finite state machine shown in Figure 2.7. The term "hidden" means that states or labels, are not explicitly labeled, but are inferred using data that correlates with states. In their case, data is incremental code submissions. Each label is a node in the finite state machine and the HMM provides the probability of transitioning from one label to the next and A computes the probability that a code submission is a label given X. The final step is to determine the amount of difficulty students had using the graphical model.

To find patterns, they compared students' transitions through the various labels in the HMM by clustering their paths using the K-Means algorithm. Given the transitions through the

various labels and k (the number of clusters to produce), they asked the algorithm to partition the sequences into clusters based on the average probability that one student's path could be produced by another student's HMM and vice versa. They found that there were several clusters where students submitted several versions of code, but remained in the same label. The number of times students remained in the same label indicated the amount of difficulty students had while programming. Interestingly, once students were having difficulty, there was a high probability that they would continue being stuck.

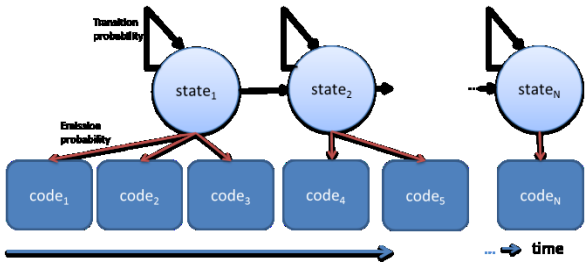


Figure 2.7: Hidden Markov Model of state transitions for a student. "Code" nodes represent a version of a student's code at a particular time and "State" nodes represent the high-level label the student is in at that same time. N represents the number of states and versions of code for a student.

2.7 Notification of Status

So far, we have discussed mechanisms that explicitly or implicitly infer developers' progress status. One question left unanswered is, how do we display these inferences to collaborators? One way to display inferences is to use standard notifications; however, notifications may be disruptive if they occur too frequently. One way to address disruptions is to enable observers to poll for a developer's status through an IM status or newsfeed during their activity breakpoints [30] to learn about status changes. The Jazz [8] and CollabVS [24] programming environments embody this approach. Both systems continuously update a view of collaborators' programming activities. The difficulty status of a teammate could be added to this view, similar to an IM status, to enable collaborators to view the status. In both the CollabVS

[24] and Jazz [8] programming environments, the view is located in the user interface of the programming environment. Alternatively, this information could be presented in a separate tool. For example, an update to the Jazz programming environment [21] provides newsfeeds, which are implemented in dashboards, a separate tool, to enable developers to passively monitor collaborators' programming activities.

Yet another alternative is a manager polling for status during a "walk around". Management by physically walking around is a well-known practice devised at HP by Dave Packard and Bill Hewlett. Sharma et al. [50] create a virtual analog of this technique to enable such management of remote offices. Their system, Virtual Office, is a 3D environment that mimics the layout of a physical office and supports audio and text chat, screen sharing, and navigation. To make the system even more realistic, only users that are within a certain distance of each other can communicate. Sharma et al. [50] envision several uses of this tool such as enabling employees who are at home to meet with collaborators who are either physically or virtually at the office and enable managers to remotely manage virtual offices. Combining a virtual office with difficulty status predictions could enable a manager visiting a worker to see, in addition to other information, the current difficulty status, and an aggregation of the status values of the worker computed since the last walk around.

These techniques inform collaborators of developers' progress status. To our knowledge, no work in the literature explores the use of developers' progress status. One possible use is teammates helping developers when they become aware that developers are having difficulty. However, before they offer to help, collaborators must first determine if they can indeed help. To answer this question, collaborators need information about teammates' context [57].

2.8 Context Awareness

One approach to share programming context is screen awareness, continuous knowledge of remote developers' screens. Teammates can use this information to see what is blocking developers from completing their tasks. Previous work has explored, to some degree, the idea of screen awareness. Tee et al. implemented screen awareness in the Community Bar tool as a sidebar that contained thumbnails of remote users' screens [56]. Collaborators used these thumbnails to monitor each other and when a change in a teammate's thumbnail indicated a potentially interesting event, they expanded the thumbnail to show that teammate's full screen. Experience with the Community Bar found that observers used the tool to determine remote users' availability, and to monitor how much progress a co-author, using track changes, was making on a shared document. In particular, the same thumbnail image for a period of time implied that the associated user was not available; and the degree of progress of a user was determined by how much tracked text in the document had the user's color. However, screen awareness alone may not provide enough information for potential helpers to determine if they can offer help, because it only shows the current information on developers' screens. Previous work found that teammates needed to know what developers have done (so far) to try to solve their problem [57]. We can model this problem as the classic latecomer problem, which generally occurs in scheduled meetings where individuals invited to a meeting join late and need to catch up on information they missed.

The most recent work on this topic, by Junuzovic and colleagues, records audio of meetings, video of the participants' faces, and a shared workspace (presentations) and transcribes the audio recording so that latecomers can replay this information to catch up [34]. The goal of their work was to determine whether replaying audio, video, shared workspace, and transcript

was better than only replaying audio. To achieve this goal, they compared the amount of information (facts, explanations, and the identity of the speaker in the meeting) that latecomers could recall after replaying audio only and different combinations of audio, video, transcript, and shared workspace. Their results show that participants who replayed audio, video, shared workspace, and transcript and audio and workspace combinations recalled more information than audio alone and any other combination. This suggests that replaying audio, video, transcript, and workspace actions are useful in helping latecomers get up to date.

However, their work did not show that replaying workspace actions, transcripts, or video alone was useful. Therefore, their results do not apply to developers who work alone because there is no video or transcript to replay. The reason is that developers work alone until they get stuck and the only information available to replay is developers' screens.

Replaying developers' screen activities provides syntactic awareness to potential helpers, which means they are not provided with inferences about developers' context. Therefore, potential helpers must manually look at developers' screens to determine context, which could cause them to spend a large amount of time trying to determine if they can offer help. This means that sometimes potential helpers waste a significant amount of their time. This problem can be addressed by inferring not only developers' progress status, but also their programming context.

2.9 Programming Context

Two types of programming context that may be useful to potential helpers are developers' difficulty level and the barriers that cause them to have difficulty. Ko et. al categorized barriers student programmers faced based on explicit help requests. To do this they

a) asked students to report what they were stuck on, how they became stuck, and how they tried to overcome being stuck and b) found similarities in this reported data. Two of the authors independently classified each barrier. The six barriers they identified are: not being able to design algorithms, unable to combine Application Programming Interfaces (APIs), not understanding compiler or runtime errors, unable to find documentation for APIs, and unable to find tools within the programming environment [37]. They showed that for most barrier kinds, about half of its instances were insurmountable, thereby suggesting two levels of difficulties (*insurmountable* and *surmountable*) [37].

Difficulties where students explicitly asked a collaborator, in their case a teaching assistant, for help were labeled insurmountable. When students asked for help, as mentioned previously, the helpers asked students how they became stuck and found that students had earlier difficulties. These earlier difficulties were labeled *surmountable* because students overcame them using other forms of help such as code examples or guessing the solution. When students' guesses were incorrect or they had trouble adapting these examples to fit their code, they were faced with insurmountable difficulties. For example, students had to design an alarm clock that could be set to ring at a certain time, but they had difficulty trying to get their program to keep time. This difficulty occurred because they were not aware of the timer API. To overcome this surmountable difficulty, they used the code of other students as examples, but adapting this code caused an insurmountable difficulty.

2.10 Multi-level Classification of Developers' Status

Previous work has also explored multi-level classification of developers' status. Fogarty et al. [19] determined levels of interruptibility by randomly interrupting participants to rank their interruptibility on a scale from 1 (most interruptible) to 5 (least interruptible). Analogous work

by Iqbal and Bailey [30] determined levels of breakpoints by using external observers/coders to view samples of participants' videos and identify breakpoints and their types. Horvitz et al. [29] provide a novel way to address the degree of difficulty issue. They asked users to quantify their willingness to receive a notification based on how much money (scaled from 0 to 1) they would be willing to give to not receive a notification. A similar scheme could be used for quantifying difficulty – how much money would one be willing to pay to make the difficulty go away.

2.11 Summary

A significant amount of previous work has motivated the idea of increasing the amount of help in software development. These motivations arguably show that the closer developers are the easier it is for developers to become aware that teammates need help. More important, previous work suggests that an increase in help also provides an increase in productivity. Given this motivation, in this chapter, we describe how techniques stemming from prior work that implicitly or explicitly infer developers' progress status. This chapter provides a basis for our work and an avenue for extending it.

Chapter 3: Programming Activity Difficulty Detection and Implementation

3.1 Introduction

In the previous chapter, we present related work that infers frustration using non-standard equipment. In this chapter, we describe an approach that logs developers' interactions with standard equipment such as programming environments and inputs these actions into a difficulty detection module that predicts whether developers are having difficulty or making progress (Figure 3.1) would be useful because it does not have the overhead associated with using non-standard equipment, reduces privacy concerns, and provides earlier awareness than non-standard equipment. The use of programming environments overcomes all the problems mentioned above because developers routinely use some form of a programming environment such as editors or debuggers. In this chapter, we develop a programming-activity difficulty-detection component that embodies such an approach.

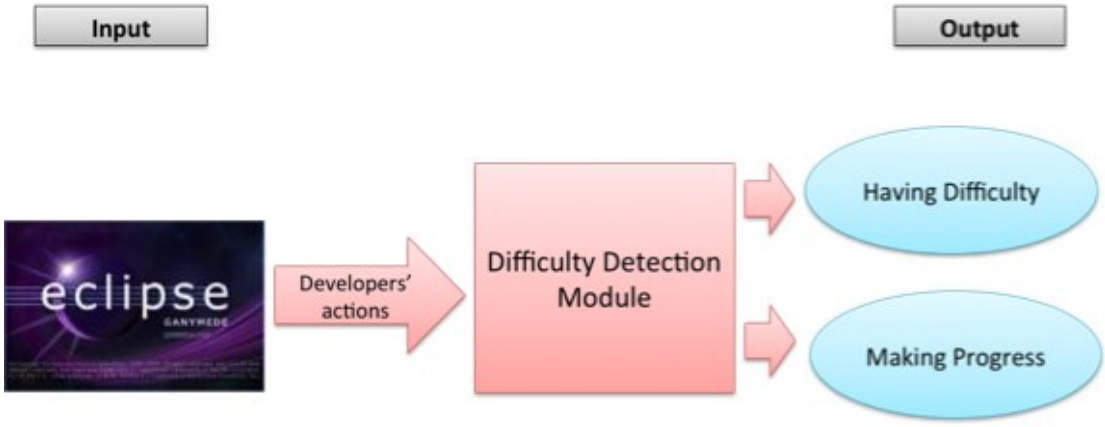


Figure 3.1: Block diagram of our difficulty detection module that takes as input developers' actions an outputs a prediction as to whether developers are having difficulty or making progress.

To show how well this approach works in practice, we needed to implement a difficulty detection module into a programming environment. We implemented a difficulty detection module into Eclipse because it is a popular programming environment at UNC-Chapel Hill. However, half of the developers that were willing to participate in our studies used Visual Studio and refused to use Eclipse. Therefore, we implemented a difficulty detection module in both programming environments, which increased programming time and effort because as the difficulty detection algorithm changed; code had to be changed in both the Eclipse and Visual Studio implementations. Existing difficulty detection systems also faced this problem and their solution was to avoid logging interactions with the programming environment. In this chapter, we address this problem by developing a reusable difficulty detection framework, which is a common set of difficulty detection modules for the Visual Studio, and Eclipse programming environments.

The rest of this chapter is organized as follows. First, we describe how we use developers' interactions to predict whether developers were having difficulty or making progress. Second, we describe experiments conducted with this approach. Third, we describe how we implemented the common set of difficulty detection modules. Following this, we describe experiments we conducted with the common set of difficulty detection modules. Finally, we present limitations and a brief summary.

3.2 Issues, Approach, and Evaluation

As mentioned above, previous work has explored automatic difficulty detection. A problem with this research, described above, is the overhead of using non-standard equipment. An alternative approach would be to determine developers' difficulty status by logging their interaction with some component of the system. As mentioned above, an important step in this

direction is taken in [40], which describes a tool that monitors students' interactions with CVS and newsgroups to calculate the workloads and work-statuses of students. This information could potentially be used to determine if students were having difficulty, but this awareness would be provided, not when they had the difficulty, but later, when they checked in the files or posted to newsgroups. Developers may struggle for a long time before they take these actions, and for certain problems, would not expect a response from the Internet. Providing earlier awareness, as in the two scenarios above, requires logging interactions with the programming environment.

Previous work has explored, to some degree, the idea of mining developers' interactions with their programming environment. Fogarty et al. [20] show that it is possible to develop a tool that uses developers' interactions with the programming environment to determine if they are interruptible. The challenge for us was to train and evaluate a system that tries to determine if someone is having difficulty. Fogarty et al. also faced this problem, and their solution was to randomly interrupt users to determine how interruptible they were. We cannot use this approach, as it is likely that no random interruption would find a developer is having difficulty - by definition having difficulty is an exceptional event.

We built on the approach taken by Kapoor et al. [35]. The participants indicated their frustration level by clicking on "I'm frustrated" or "I need help" buttons. These buttons are useful only for the training phase. Even in this phase, it may be useful to run an initial naïve algorithm that actively guesses the progress status, whose predictions can be corrected by the developers. The reason is that developers are more apt to correct a guessed status than to remember to press the buttons to indicate their status. This is the first approach we took, and Figure 3.2 shows the user-interface for correcting the status. The "Eureka" button was intended

to capture those situations in which developers did not realize they had been having an unusual problem until they had solved it. However, none of our subjects used it. The "Notifications Enabled" button enabled developers to determine if they received status change notifications.

Figure 3.2: Buttons developers press to indicate their status.

Creating a naïve algorithm for the training phase requires some top down thinking about how having difficulty could be inferred. The basic intuition is to monitor progress of developers, and when this progress is less than some threshold, indicate that they are having difficulty. Progress is related to productivity but is also fundamentally different. It is measured while programmers are writing code, while productivity is usually measured after programmers have done so. There are several measures for productivity such as time to market. However, little work has been done on measuring progress.

The only one we found was one done by Kersten and Murphy [36], which provides a tool for automatically showing to developers items related to their tasks, thereby reducing the need to manually navigate to these items. They measure the success of their tool by determining how the tool changes developers' edit ratio, which is the ratio of number of editing commands to the number of navigation commands. Instead of using this metric to evaluate the performance of an algorithm, as in [36], we used it as an input to the design of our naïve algorithm for determining status changes. If the edit ratio and number of debugs is less than a low threshold, the algorithm notifies the developers that they are having difficulty. If a correction is made, the threshold is increased. As in [36] , we have extended an Eclipse plug-in [51] to log developers' interaction with it and compute the edit ratio. We logged three freshmen doing class assignments, and three

graduate students doing class and research assignments. We equated being stuck and needing help with having difficulty.

All but one programmer pressed the "Stuck" button to indicate lack of progress. The naïve algorithm did not predict the progress status well. Our next step was to explore the logs and corrections to derive a better algorithm.

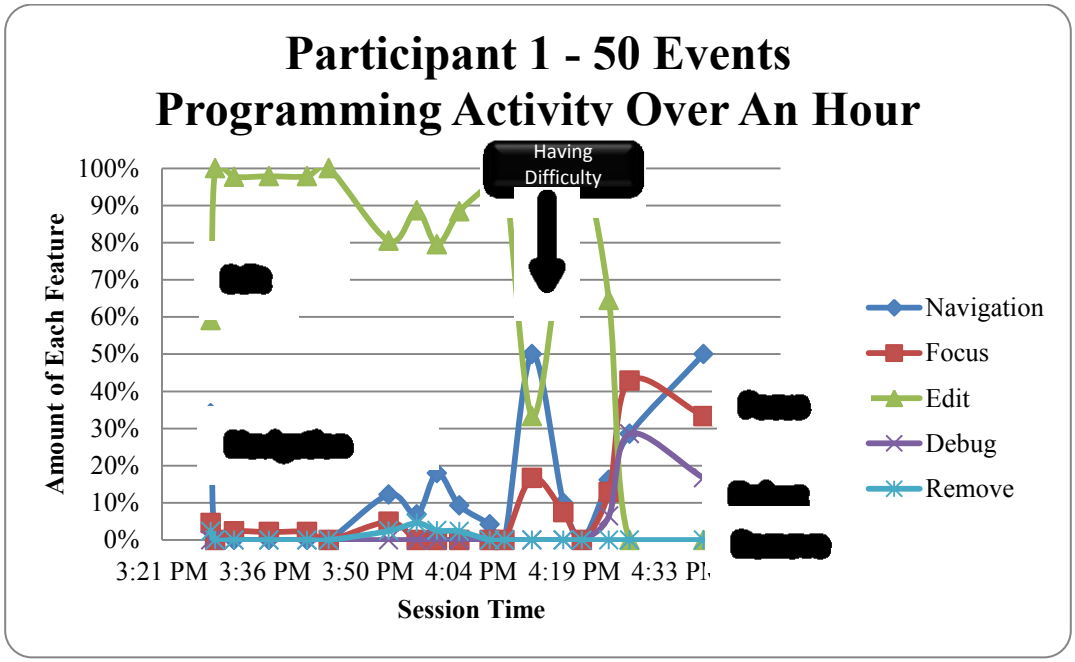


Figure 3.3: Participant 1's programming activity over an hour.

3.3 Deriving Mining Algorithm

We analyzed the logs to determine if there are patterns that occur when developers indicate they are having difficulty. To determine the patterns, we must determine values known as features that change when programmers are making progress and having difficulty.

A manual inspection of the logs showed that, consistent with the assumption of the naïve algorithm, the frequency of certain edit commands decreased when developers were having difficulty. Depending on the developer, the frequency of execution of other commands increased.

Based on these data, we grouped the commands into five categories (Table 3.1):

- Navigation
- Edit (text insertion/deletion)
- Remove (methods and/or classes)
- Debug
- Focus

We calculated, for different segments of the log, the ratio of the occurrences of each category of commands in that segment to the total number of commands in the segment as percentage, and used these percentages as features over which patterns are identified.

Table 3.1: Programming action categories and their explanations.

Programming Action Category	**Explanation**
Navigation	User switches from one file to another file.
Edit	User inserts or deletes text.
Remove	User removes entire methods or classes.
Debug	User explicitly debugs code using the debugger.
Focus	User switches between Eclipse and other open windows.

As programmers work at different rates, the log was segmented based on the number of events executed instead of time. The size of these segments is an important issue - if the size is too large, then both kinds of patterns might occur in a single segment, and if it is too small, there might not be sufficient information to determine whether developers were having difficulty. To illustrate, it is undesirable to have segment size that is one or the size of the complete log. After experimenting with several values of it, we found a segment size of 50 to be the best.

To determine how indicative the features are of programmers' behavior we graphed the programming behavior of all six programmers. In each graph, the x-axis is session time and y-

axis is the percent for each feature. Figures 3.3 and 3.4 are portions of the graphs created for participant 1 and 2, respectively, illustrating both commonalities and differences in the behavior of programmers. In both cases, when the programmers indicated they were having difficulty, the edit percentages decreased and other percentages increased.

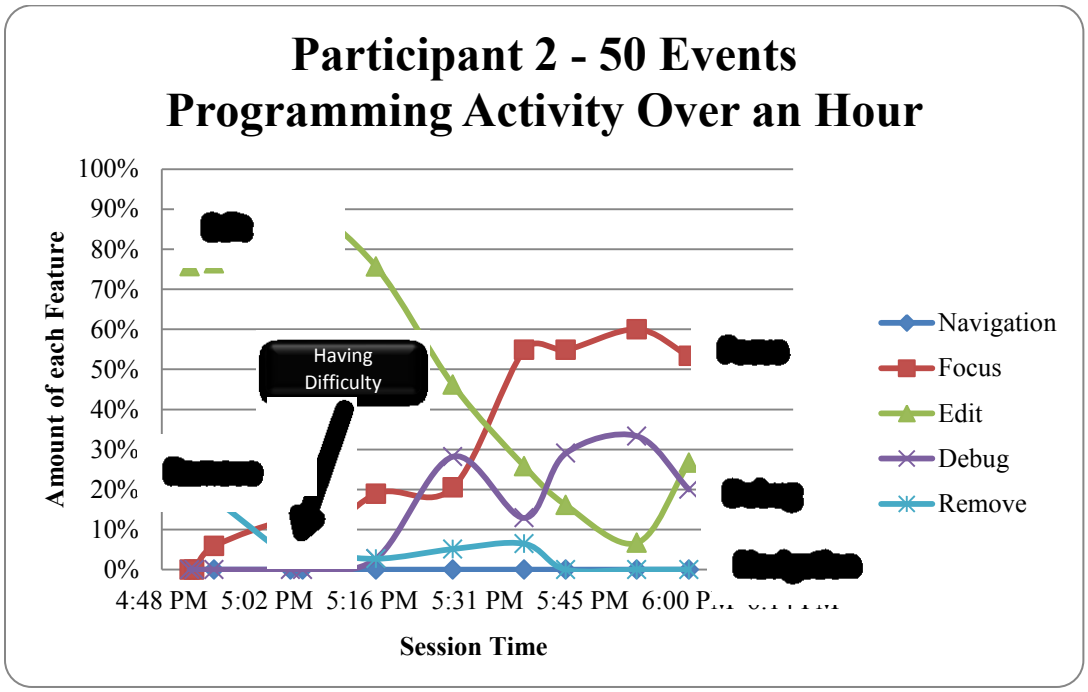


Figure 3.4: Participant 2's programming activity over an hour.

When participant 1(2) was stuck, the navigation (debug and focus) percentage increased. Participant 2's edit (debug and focus) percentages continued to decrease (increase) for a while after he indicated he was stuck, which was not true in the case of participant 1. This seems to indicate that participant 1 was quicker in detecting, or at least informing the system, that he was having difficulty. Thus, the two graphs validate our feature choice.

There are several standard ways to build a general model. In particular, we tried the naïve Bayes model as it is the one used in [20] for predicting the interruptibility status. Interruptibility and progress seem to be related as they both indicate the status of developers. More interestingly, there may be a correlation between the two – the more progress developers

are making, the less interruptible they might be.

On the other hand, there is also reason to believe that progress and interruptibility statuses are fundamentally different because having difficulty is a rare event. In our experiments, developers indicated they were stuck only for 76 of the 2288 (3%) total segments (Figure 3.5). This leads to the class imbalance problem, which occurs when trying to detect a rare but important event such as having difficulty. The accuracy of traditional classification algorithms are biased towards the more common event, making progress, and will not recognize the rare event, having difficulty. The SMOTE [7] algorithm implemented in the WEKA toolkit [59] overcomes this problem. It replicates rare data, having difficulty, until that data are equal to the more common data, making progress. Therefore, we used this scheme, which converted the 76 rare records to 1216 replicated ones.

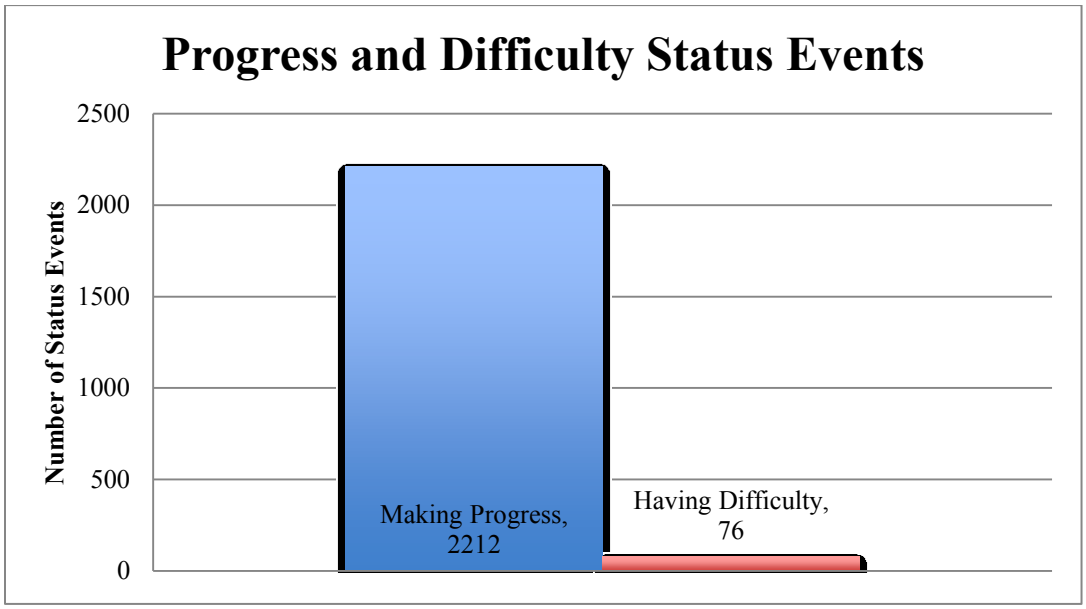


Figure 3.5: The number of progress and difficulty status events.

3.4 Mining Algorithm Results

The replicated data of all developers were combined and used as input to three standard

algorithms to build statistical models. The performance of each model is shown as a confusion matrix. The rows represent the actual times developers were stuck and the columns represented the values predicted by the model. The confusion matrix in Table 3.2 shows the results of the naïve Bayes algorithm. The accuracy of this algorithm is 73%, the true positive rate is 63%, the true negative rate is 77%, the false negative rate is 37%, and the false positive rate is 23%. These results show that the algorithm missed 37% of the time when developers were having difficulty and 23% of the time when they were making progress. The decision tree algorithm, described in [59], gave better results (Table 3.3).

Table 3.2: Confusion matrix for naïve Bayes algorithm with the smote algorithm applied.

	Predicted Stuck	Predicted Progress
Actual Stuck	769 (True positives)	447 (False negatives)
Actual Progress	504 (False positives)	1708 (True negatives)

Table 3.3: Confusion matrix for decision tree algorithm with the smote algorithm applied.

	Predicted Stuck	Predicted Progress
Actual Stuck	1101 (True positives)	115 (False negatives)
Actual Progress	158 (False positives)	2054 (True negatives)

It correctly predicted making progress 92% (true negative rate) of the time and having difficulty 91% (true positive rate) of the time. By themselves, these numbers are not very

impressive, because simply guessing that the developer is always making progress would have been correct 97% of the time, but would never correctly predict when developers were having difficulty. More interestingly, our scheme identified 91% of the having-difficulty statuses.

To determine these numbers, we used a standard technique, known as k-fold cross validation, which executes 10 trials of model construction, and splits the logged data so that 90% of the data are used to train the algorithm and 10% of the data are used to test it.

There were times when the model was early or late in its prediction, which is consistent with the fact that developers differed in how quickly they indicated they were stuck. These were counted as incorrect predictions.

We also tried an algorithm, Classification by Clustering [59], which is designed to identify rare events without replicating records. The confusion matrix in Table 3.4 shows the results of this algorithm. The accuracy of this algorithm is 65%, the true positive rate is 79%, the true negative rate is 65%, the false negative rate is 21%, and the false positive rate is 35%. These results show that the algorithm identified when developers were having difficulty 79% of the time (true positive rate).

Table 3.4: Confusion matrix for classification via clustering algorithm.

	Predicted Stuck	Predicted Progress
Actual Stuck	60 (True positives)	16 (False negatives)
Actual Progress	766 (False positives)	1446 (True negatives)

3.4.1 Comparison of Mining Algorithm Results to Baselines

These results show that the decision algorithm correctly identified more times when developers were having difficulty and making progress than the naïve Bayes and classification via clustering algorithm. To provide evidence to support our *Programming Activity Difficulty Detection* sub-thesis, we compare the results of the decision algorithm to the baselines we described in section 3.3. To compute each baseline we a) use the same data given to the decision tree algorithm and b) assume that the distribution of the actual having difficulty and making progress statuses follow the distribution of the having difficulty and making progress statuses for each baseline.

To compute the random baseline, we assume that a) half of the actual having difficulty statuses were predicted as having difficulty, b) half of the actual having difficulty statuses were predicted as progress, c) half of the actual progress statuses were predicted as progress, and d) half of the actual progress statuses were predicted as having difficulty. Figure 3.6 shows the distribution of data for the random baseline. To compute the modal baseline, we assume that a) none of the actual having difficulty statuses was predicted as having difficulty, b) all of the actual having difficulty statuses were predicted as progress, c) none of the actual progress statuses was predicted as having difficulty, and d) all of the actual progress statuses were predicted as progress. Figure 3.7 shows the distribution of data for the modal baseline. To compute the last baseline, data distribution, we use the distribution of the actual having difficulty and progress statuses, which is 35% and 65% respectively. More specifically, we assume that a) 35% of the actual having difficulty statuses were predicted as having difficulty, b) 65% of the actual stuck having difficulty were predicted as progress, c) 65% of the actual progress statuses were predicted as progress, and d) 35% of the actual progress statuses were predicted as having

difficulty. Figure 3.8 shows the distribution of data for the data distribution baseline. We use this approach to create baselines in chapter 5.

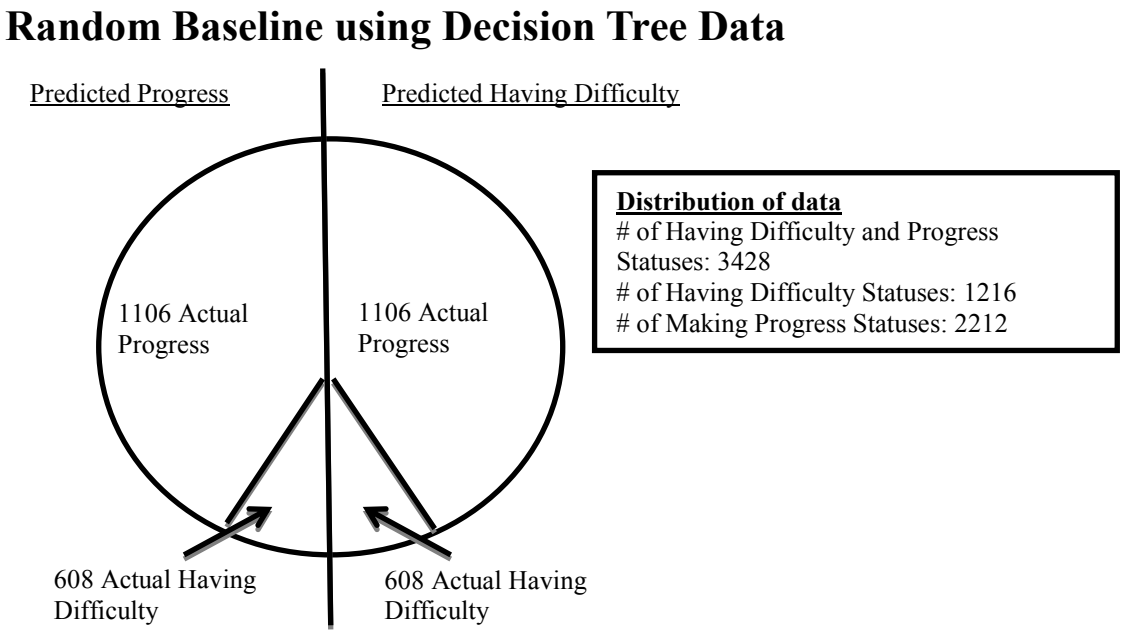


Figure 3.6: The distribution of data for the random baseline (decision tree data).

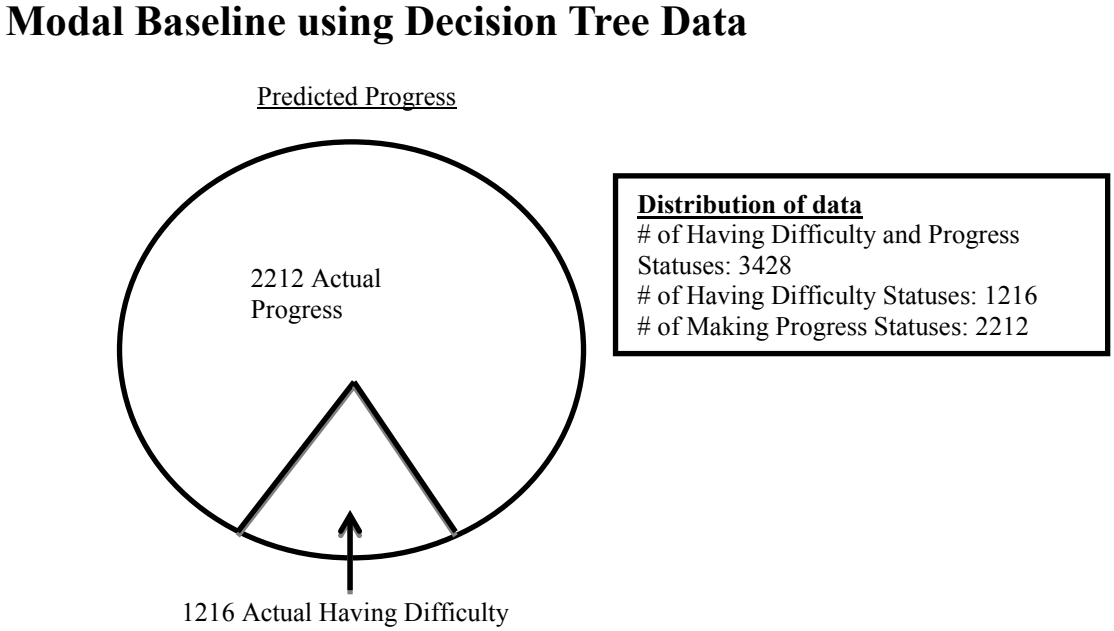


Figure 3.7: The distribution of data for the modal baseline (decision tree data).

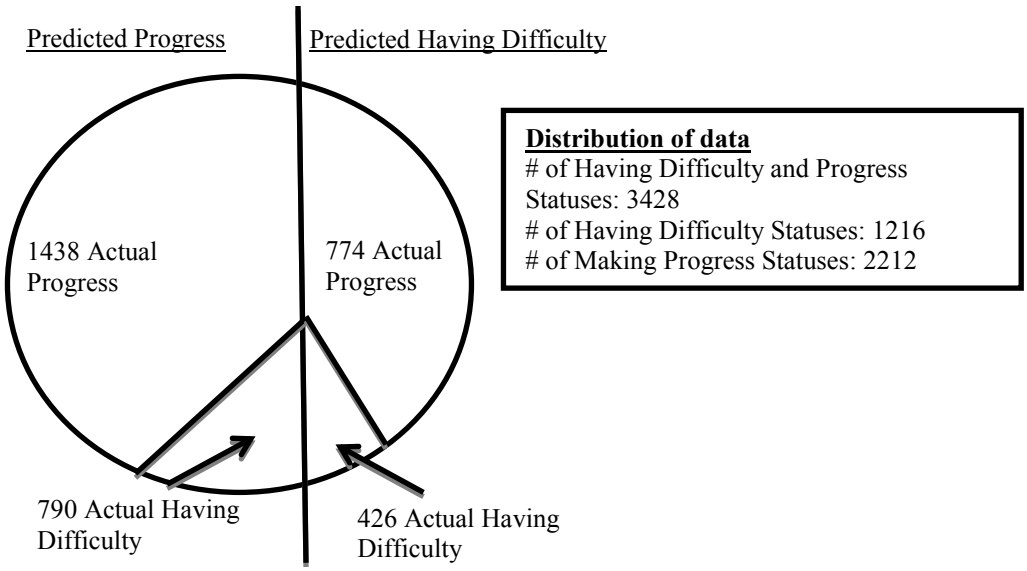


Figure 3.8: The distribution of data for the data distribution baseline (decision tree data).

Similar to the decision tree algorithm, the results of each baseline are shown as confusion matrices. Table 3.5 shows the results for the random baseline. The accuracy of this baseline is 50%, the true positive rate is 50%, the true negative rate is 50%, the false negative rate is 50%, and the false positive rate is 50%. This baseline identifies 50% of the time when developers are making progress and 50% of the time when they are having difficulty. To determine whether there is a significant statistical difference between the performance of the random and data distribution baselines and the decision tree algorithm, we use the binomial test. We do not use a significance test to compare the modal baseline to our approach because it never identifies when developers are having difficulty (TPR=0%), which means the true positive rate is clearly different. The binomial test determines whether there is a significant difference between an observed experimental value, the results from the decision tree algorithm, and a fixed value, the results of each baseline. The decision tree algorithm performs significantly better than the random baseline (TPR=91% vs. TPR=50%, $p < .001$) (TNR=93% vs. TNR=50%, $p < .001$).

Table 3.5: Confusion matrix for random baseline.

	Predicted Stuck	Predicted Progress
Actual Stuck	608 (True positives)	608 (False negatives)
Actual Progress	1106 (False positives)	1106 (True negatives)

Table 3.6 shows the results for the modal baseline. The accuracy of this baseline is 65%, the true positive rate is 0%, the true negative rate is 100%, the false negative rate is 100%, and the false positive rate is 0%. This baseline always identifies when developers are making progress, but never identifies when they are having difficulty. As mentioned above, we do not use a significance test to compare the modal baseline to our approach. The true positive rate (91%) of the decision tree algorithm is better than true positive rate (0%) of the modal baseline.

Table 3.6: Confusion matrix for modal baseline.

	Predicted Stuck	Predicted Progress
Actual Stuck	0 (True positives)	1216 (False negatives)
Actual Progress	0 (False positives)	2212 (True negatives)

Table 3.7 shows the results for the data distribution baseline. The accuracy of this baseline is 65%, the true positive rate is 35%, the true negative rate is 65%, the false negative rate is 35% and the false positive rate is 35%. This baseline identifies 65% of the time when developers are making progress, but only identifies 35% of the time when they are having

difficulty. The decision tree algorithm performs significantly better than the data distribution baseline (TPR=91% vs. TPR=50%, $p < .001$) (TNR=93% vs. TNR=50%, $p < .001$).

Table 3.7: Confusion matrix for data distribution baseline.

	Predicted Stuck	Predicted Progress
Actual Stuck	426 (True positives)	790 (False negatives)
Actual Progress	774 (False positives)	1428 (True negatives)

3.4.2 Discussion

Our results are promising because it a) recognizes with high accuracy when student programmers are having difficulty even though having difficulty is a rare event and b) performs better than the random, modal, and data distribution baselines. These results provide partial evidence to support sub-thesis I, which we restate here.

Programming Activity Difficulty Detection Sub-Theses (Sub-thesis I):

It is possible to develop an approach that a) uses developers' interactions with their programming environment to determine whether developers are having difficulty with their task and b) performs better than baseline measures.

Iterative Approach

Our exploration of a programming-activity difficulty-detection algorithm yielded an iterative process, which consists of the following steps:

1. Develop an initial naïve algorithm for predicting the having difficulty status.

2. Implement the algorithm in one or more programming environments.

3. Ask selected developers in lab and/or field experiments to correct the predictions made by the current algorithm.

4. Analyze the logs to refine the set of features.

5. Input these features to existing selected log-mining algorithms.

6. If none of these algorithms makes a significant improvement, stop.

7. Make the algorithm that gives the best results the current algorithm.

We have carried out the first iteration of the process, but our work so far leaves several important questions unanswered.

1. Is it possible to develop a common set of extensible prediction modules for different programming environments?

2. Is it possible for the modules to have no impact on the response times perceived by the developers?

3. How well does the previous algorithm work when industrial programmers use it?

4. Is it better to train the modules using logs of the individual developer whose status is predicted, or some group of programmers that excludes him/her?

5. What is the correlation between the perceptions of the developers and their observers regarding whether the developers are having difficulty?

6. If these perceptions differ, how well can the predictions made by a tool correlate with the perceptions of human observers?

3.5 Initial Evaluation and Adaptations

To determine how well our programming-activity difficult- detection algorithm works in practice, we took two additional implementation and evaluation steps. (1) We incorporated the algorithm in both the Eclipse and Visual Studio programming environments. (2) Some members of our research group, and one industrial software developer, used the Eclipse and Visual Studio implementations for their daily work. We gained important lessons from these steps.

The industrial developer complained about frequent false positives while building a new product – a workflow system. In particular, when he started a new session, the tool gave a relatively high number of false positives because of the navigations performed to build the working set of files. He also needed more time to determine if the predicted change of status was correct, and, thus, often was not sure about his status.

My advisor identified two additional problems. The cost of processing incremental input events was noticeable, and sometimes intolerable, on his 3-year old laptop. Moreover, even when the tool accurately predicted he was having difficulty, seeing the status message hurt his ego, as he felt that the change in progress was caused by the difficulty of the problem rather than lack of appropriate skills!

A final problem had to do with the implementation architecture: the Visual Studio and Eclipse implementations performed the same functions, but did not share code. Therefore, when a change was made to the code in the Eclipse implementation, the code in Visual Studio had to also change. Put in another way, there would need to be a different implementation of the tool

per programming environment, which increases programming time and effort. In particular, the Eclipse implementation had 11,000 lines of code and the Visual Studio implementation had 9,096.

We took several steps to address these problems. To address the "hurt ego" issue, we changed the status message from "Having Difficulty" to "Slow Progress." In addition, we enabled developers to customize the message so that my advisor could, for instance, report it as "Complex Programming."

To address the false positives faced by the industrial programmer, we developed a label aggregation technique that complemented the event aggregation technique. As before, we computed the status every 50 events. However, we notified the developer every 250 events – the value reported was the dominant status in the last five segments.

Together, the two aggregation techniques take into account the fact that the status of a developer does not change instantaneously. In addition, we added an "indeterminate" status value to capture the fact that developers need time to decide if they are stuck. At startup, before 250 events were input, the tool reported the indeterminate value. We also enabled the developer to correct a predicted status to indeterminate.

Table 3.8: Field Study of Industrial Software Developer.

Status	Guessed	# Corrected	Accuracy
Difficulty	17	2	88%
Making Progress	69	7	89%
Indeterminate	2	0	100%

Table 3.8 shows that the changes resulted in a high accuracy for the industrial developer. However, the table shows that the aggregation scheme results in a large number of false negatives. In particular, it missed 7 of the 22 cases when the developer was having difficulty. To develop a more accurate scheme, we gathered more data points through a user study.

3.6 Reusable Difficulty Detection Framework

Before this step can be taken, it was important to address the performance and implementation overhead of the Eclipse and Visual Studio implementations. Liu and Stroulia also faced this problem and their solution was to avoid logging interactions with the programming environment. Instead, they logged student programmers' interactions with version control systems, wikis, and newsgroups to calculate their workloads and work statuses [40]. This information could be used to determine when students are having difficulty. The authors of this work admit that a drawback of this approach is the additional work involved in using wikis and newsgroups for programming tasks. Even more important, potential helpers would not become aware of students' difficulties until they checked in certain files or posted to newsgroups.

Students may struggle a long time before they take either of these actions. Although the authors of [40] were aware of at least the first drawback of their approach, they decided to use it because "many students have a preferred programming environment and establishing a common one would be a challenge."

One way to overcome this problem is to create a common set of difficulty detection modules for mainstream programming environments. This approach reduces the amount of programming time and effort. A reusable architecture is crucial for this research because of its iterative nature. We were able to apply certain standard design patterns and existing libraries to address the reuse issue. To address the performance issue, we offloaded event processing to a separate process that worked asynchronously from the programming environment.

Figure 3.9 shows the architecture. Naturally, a separate module is needed per programming environment to intercept its events. In addition, a separate module is needed per programming environment to display the current status, which is done by using a Google talk plug-in. Thus, in our implementation we use two different event-interception and status-display modules – one pair for Eclipse, and one for Visual Studio.

An event-interception module asynchronously forwards the events to a separate process, which makes the predictions. As the process was written in C#, serialized events could be sent directly from Visual Studio to this process. Java events, on the other hand, require conversion, and we were able to use standard (WOX and IKVM) libraries to do so.

Consider now the modules in the predicting process. Events are received by the "communication director" of the system, the mediator, which mediates between a pipeline of other modules. The mediator gives the received event to the first module in the pipeline. In

addition, it receives output from each of these modules and feeds it as input the next module, if such a module exists.

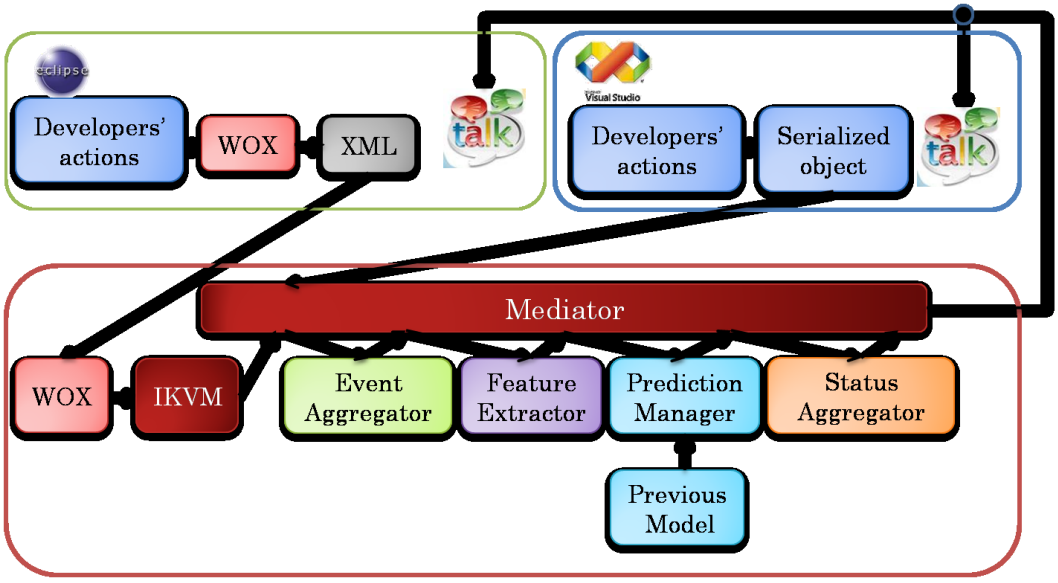


Figure 3.9: System Architecture.

The first module to receive input from the mediator is the event aggregator module. This module aggregates 50 events and passes these events to the mediator. The mediator passes these events to the feature extractor module, which computes the ratios that are used to predict a status. The feature extractor passes the ratios to the mediator, and the mediator gives these ratios to the prediction manager. The prediction manager includes the decision tree algorithm (used in section 3.4), which uses previous data and the ratios to predict a status. This status is passed to the status aggregator, which aggregates each status and gives a final prediction to the mediator. The mediator delivers this status to the status displayer of the appropriate programming environment.

The benefit of using the mediator pattern is that it enables modules to be loosely coupled so that any change in the flow of communication would not require a change to a module. For example, if the status manager had to be omitted, the mediator would have to change. However, the other modules in the system would stay the same.

The iterative nature of this research requires the ability to easily change also the behavior of each of the individual modules in this pipeline. We used the standard Strategy pattern to achieve this goal. We give below specific uses for it in our context by considering each of the phases in the pipeline, and showing that multiple algorithms could be used in each phase.

1. Event aggregator: There are at least two algorithms that can be run to aggregate events. The current algorithm uses discrete, independent chunks of 50 events. An alternate option is to use a gradual sliding window approach similar to the approach used in TCP/IP. The code below shows the use of the strategy pattern to easily switch between the two, assuming both are implemented:

   ```
   EventAggregator ea = new EventAggregator();
   ea.setEventAggregationStrategy(new SlidingWindow());
   ea.setEventAggregationStrategy(new DiscreteChunks())
   ```

2. Feature extractor: It currently extracts features based on the number of events. For example, the edit ratio it computes is the number of edits divided by the total sum of all actions including editing. It would also be useful extract features based on time such as editing time/total time. Another useful feature that was observed while watching developers solve problems is the number of exceptions per run.
3. Prediction manager: It currently uses two machine learning algorithms, decision tree and classification via clustering, to predict developers' status. In the future, we plan to test other classification or clustering algorithms, and perhaps build our own algorithm.
4. Status manager: There are at least two ways to aggregate statuses. Currently, it aggregates five statuses and takes the most dominant status. This algorithm is similar to aggregating events in discrete chunks. Another approach is to use a sliding window, which corresponds to using a sliding window to aggregate events.

Our experience with the new architecture showed that (a) as expected, when multiple strategy objects were implemented for a stage, it was indeed trivial to replace one with the other, (b) the asynchronous processing did not result in perceptible delays in user-response times, and (c) the number of lines of code to implement the architecture, 4,643 is significantly less than the number of lines of code to implement difficulty detection modules written specifically for Visual Studio and Eclipse. These results provide evidence to support sub-thesis II, which we restate here.

Implementation Sub-thesis (Sub-thesis II): *It is possible to develop a common set of difficulty detection modules for different programming environments that have significantly fewer lines of code than difficulty detection modules written specifically for each programming environment.*

We were now ready to do a controlled user study to evaluate the adapted algorithm and investigate additional adaptations based on this study.

3.7 User and Coding Study

In a controlled user study, the problems must be chosen carefully. Our previous work, section 3.4, found that having difficulty is a rare event. Thus, we must try to ensure that developers face difficulty in the small amount of time available (1-4 hours) for a lab study, and yet do not find the problems impossible.

We used problems from the Mid-Atlantic ACM programming competition. These problems are attractive because they have varying difficulty. We piloted several problems to find problems that were difficult but not impossible to solve by the subjects. Based on these pilots, we settled on the problems shown in Table 3.9. The table characterizes the difficulty of each problem by showing the number of teams that solved the problem, the total number of teams, and the fraction of teams that solved the problem. The number of teams that solved the problem determined the difficulty level of each problem. For example, 100% of teams that

attempted the Simple Question of Chemistry problem solved it, while only 16% of teams that attempted the Balanced Budget Initiative Problem solved it.

Five industrial and nine student programmers participated in the study. Participants were instructed to correct an incorrect prediction by the system using status-correction buttons discussed in section 3.4 (Figure 3.5). By measuring how often the developers corrected their status, we could, as in section 3.4, measure the accuracy of our approach with respect to the perceptions of the developers.

However, there is a question as to whether participants would accurately report their status, given the hurt ego problem faced by my advisor. Moreover, it is useful to compare the tool's predictions about a developer's status with that of a third party manually observing the developer. Therefore, two independent coders and I observed participants' programming activities and made an independent determination of their status. To enable coders to independently and asynchronously observe participants' programming activities, we used Microsoft Live Meeting® to record the participants' screens. Live Meeting® also enabled me to observe remote sessions. In fact, Tang et al. [54] argued that screen recording is an effective and unobtrusive technique when subjects do not feel it invades their privacy.

Table 3.9: ACM problems from Mid-Atlantic contest.

Year	Problem Title	# of teams that solved problem	# of teams	% correct
2006	Shrew-ology	43	138	31%
2004	Balanced Budget Initiative	23	142	16%
2002	A Simple Question of Chemistry	124	124	100%

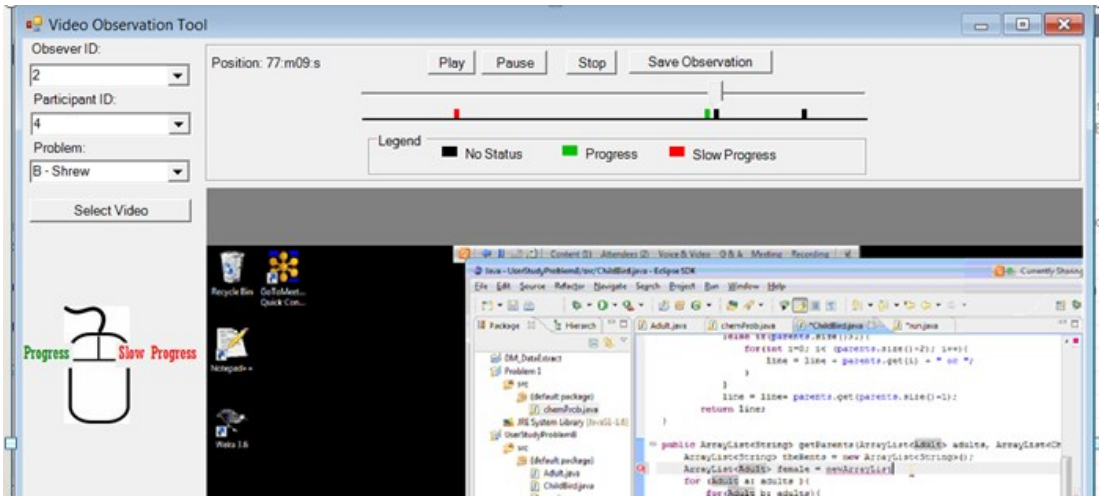


Figure 3.10: Video Coding Tool

We obtained participants' consent to record their screens. We recorded 40 hours and 44 minutes of video. To relieve coders from watching hours of video, we created a video observation tool, shown in Figure 3.10. This video tool shows all segments where the participant, system, or myself (while observing the experiments and later when randomly sampling the video), indicated the participant was having difficulty or not sure of their status (indeterminate). As it turned out, in our study, there was one indeterminate segment (indicated by a participant). We shall refer to these segments as “stuck” segments.

As there were few such segments, we asked the coders to classify each of these segments. It was not reasonable, however, to ask them to classify all of the other segments, which would have involved watching over forty hours of video. We could use a statistical sampling approach to reduce the number, but because having difficulty is a rare event, we would have had to sample the vast majority of segments to capture the false negatives.

Therefore, we used the following, somewhat arbitrary approach to choose the "making progress" segments. We randomly chose these segments, and made the number of randomly sampled points about the same as the number of having difficulty or indeterminate segments. If there were fewer than three having difficulty or indeterminate segments, we randomly sampled three segments. We shall refer to the randomly sampled segments as "random segments."

Each segment was two minutes of video. Coders were not aware of the status of each segment and had to classify the segment as making progress or slow progress. They were shown the video that corresponded to a particular participant and problem. If there were any segments for the coder to classify, they were shown on a line below the track bar. The segments on the line corresponded with the particular point in the video the coder needed to classify.

To classify segments, coders right clicked on the segment to label it as "slow progress" (the message displayed for "having difficulty"), and left clicked to label it "making progress." An image of a mouse was provided to remind coders what each mouse button meant, and a legend was also provided to help coders remember that a black segment meant the segment was unlabeled, a red segment meant slow progress, and a green segment meant making progress. Two coders and I classified 26 stuck segments and 36 random segments.

3.8 User and Coding Study Results

After the user study and coding phases were complete, we were able to answer the following questions: What is the correlation between (a) predictions of the two coders; (b)

developers' and coders' perception of status, (c) predictions of the tool and the developers' perception of the status, and (d) predictions of the tool and the coders' perception of the status? As we see below, the answers depended on whether the segment involved was one of the "stuck" segments or random segments.

Table 3.10 shows that coders agreed 88% of the time with each other on stuck segments, and 83% of the time on random segments, and overall they agreed 85% of the time. To determine the level of agreement within the stuck (random) segments we counted the number of times observers agreed with each other and divided that by the total number of stuck (random) segments observed.

The confusion matrix in Table 3.11 shows the results of using coders as ground truth. The accuracy of this system based on coders' perceptions is 79%, the true positive rate is 63%, the true negative rate is 100%, the false negative rate is 37%, and the false positive rate is 0%. To understand the high false negative rate, we asked coders their reason for classifying developers as having difficulty during random segments. Coders seemed to take the inactivity of developers as having difficulty. More specifically, we noticed that coders seemed to have a difficult time classifying participants when they were idle and apparently thinking. The tool uses developers' actions to predict their status and does not take into account think times or when developers are idle. Therefore, we consider the fifteen random segments as "making progress" when computing the accuracy of the tool.

So what did the participants themselves feel about their status in case of random segments? Table 3.13 provides an answer to this question. The accuracy of this system based on developers' perceptions is 97%, the true positive rate is 76%, the true negative rate is 100%, the false negative rate is 24%, and the false positive rate is 0%. By definition, participants agreed

completely with the predicted status for random segments, as these were the segments that were classified by the tool, participant, and me as "making progress" segments.

Consider now the non-random or "stuck segments." Again, these are the segments classified by either me, the participant, or the tool as "having difficulty." These segments tell a very different story. Table 3.12 shows the agreement of the coders with the tool, the author, and the participants for these segments. Interestingly, coders agreed with the tool 100% of the time that participants were stuck. Perhaps even more interestingly, participants never corrected a "having difficulty" status predicted by the tool.

In four of these segments, participants corrected the "making progress" prediction of the tool. Three of those times, participants indicated they were having difficulty, and one of those times participants indicated that they were not sure of their status (indeterminate.) In nine of these segments, I classified the "making progress" prediction of the tool as actually "having difficulty." The coders agreed with seven of these observations (77%). Coders agreed with the participant 75% of the time. The coders disagreed with the participant who indicated indeterminate as the status. I also reviewed this disagreement and agreed with the coders that the participant was indeed having difficulty.

Several (preliminary) conclusions can be drawn from these results. What is perhaps most remarkable is that when the tool predicts programmers are having difficulty, all three types of humans involved in making the prediction – the participants, the coders, and I, also think they are having difficulty. Thus, the tool does not seem to give a false positive, which is a very strong result and a significant improvement over the decision tree algorithm results in section 3.6.

Table 3.10: Observer's agreement with each other.

Segment Type	# of Agreements	# of Observations	% Agreement
Stuck segments	23	26	88%
Random segments	30	36	83%
Total	53	62	85%

Table 3.11: Confusion matrix for programming environment component using observers as ground truth.

	Predicted Stuck	Predicted Progress
Actual Stuck	26 (True positives)	15 (False negatives)
Actual Progress	0 (False positives)	21 (True negatives)

Table 3.12: Coders' agreement with the tool, me, and participants (stuck segments).

Entity	# of Agreements	# of Observations	% Agreement
Tool	13	13	100%
Me	7	9	77%
Participant	3	4	75%
Total	23	26	88%

Moreover, if we take the participants' perceptions as ground truth, the tool also gives negligible false negatives – only four segments out of 1222 segments in the entire study were corrected. On the other hand, if we take the coders' agreements as ground truth, the results are not so good, and it seems, based on our sampling, the tool missed half of the positives (stuck status).

Table 3.13: Confusion matrix for programming environment component using developers as ground truth.

	Predicted Stuck	Predicted Progress
Actual Stuck	13 (True positives)	4 (False negatives)
Actual Progress	0 (False positives)	1205 (True negatives)

There are two ways to interpret these data. The first relies on the viewpoint of the participants rather than the coders. The argument for doing so is that the observers could not read the mind of the participants, and were probably looking only at idle times to deduce the developer status. Idle times, alone, are not sufficient to distinguish between thinking and having difficulty. Our tool, on the other hand, keeps track of and computes a larger number of factors, such as the navigation, edit, and focus ratios, and thus agrees more with the participants. In fact, when asked about the accuracy of the tool, participants commented that they were happy with it (Table 4). The numbers shown in the table are represented by the following two comments: *"I think it worked pretty well; It's non-intrusive, and only pops up with information when the status changes." " It knew when I was having issues cause it switched to slow progress and when I was flyin doing all the class design it said progress."*

The other interpretation relies on the observers (coders and me) rather than the participants. The rationale for doing so is that participants tend to underreport [52]. The false negatives of the tool can be explained by two factors:

1. The tool uses developers' actions to predict their status, and does not take into account idle times, which should probably be considered in a future algorithm.
2. The training set consisted of data from the six student programmers logged section 3.4, who used the tool during normal "field work" consisting of assignments and research projects. The behavior of these programmers was different in some ways from those of several of the programmers in this lab study. The first group primarily used the Internet to look for help when they were having difficulty. The participants in this study did not use the Internet often because of the type of tasks and duration of this study. The only times they used the Internet was to remember syntax or look at the Java or .NET API. Moreover, the two groups solved different types of problems, and the group in this study also included industrial programmers. One piece of objective data seems to indicate that the type of programmer may be a factor in automatic status prediction. For three student participants, the automatic predictions were completely in agreement with the perceptions of the coders, when the coders agreed.

Even under this interpretation, our tool seems useful because of the zero false-positive rates. It seems that if a choice has to be made between low false positives and negatives, the former is more desirable, as it does not unnecessarily waste the time of the developers and those who offer help. Missing some "having difficulty" statuses is no worse than the current practice of not having any automatic predictions. Our tool did give several positives (thirteen), which were all correct under this interpretation. Thus, if it is considered desirable to automatically let others

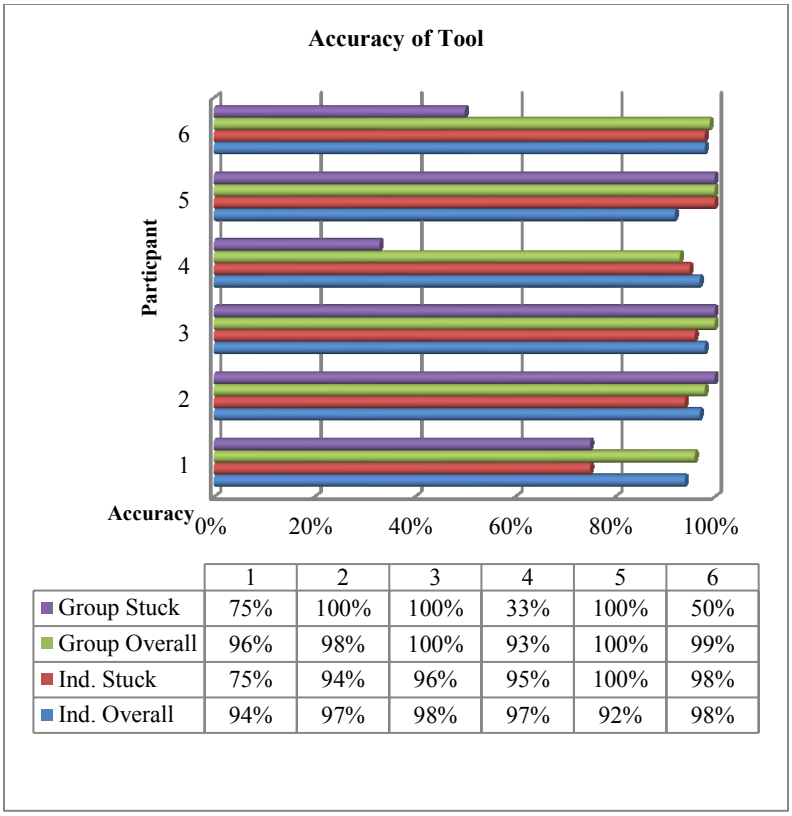
know about developers' difficulties – an assumption of this research based on previous work - then it seems better to use our tool than not use it.

Naturally, it is attractive to try to reduce the false negative rate (under the second interpretation) without increasing the false positive rate. One way to do so is train the system using the observers' conclusions rather than developer corrections (assuming the former are true). Moreover, the accuracy can be further improved if the training data involved the same exercises as the ones used in the testing phase. We could build either a group model, in which the data of multiple developers is aggregated during the training phase, or an individual model, where no aggregation is done. (The approach described so far was also a group model, but in it, the training group was smaller and solved different problems) Therefore, we decided to, next, explore these directions.

3.9 Predicting Observer Status

To build the individual and our group models, we assumed the following ground truth. All segments classified by the participants as stuck, were indeed stuck segments. Participants implicitly classify segments as stuck when they do not correct a stuck prediction of the tool. They explicitly classify them as stuck when they correct a "making progress" segment as "slow progress."

Of the remaining segments, if the two coders and I classified a segment as stuck, then it was also a stuck segment, regardless of how the participant classified it. All other segments were making progress.



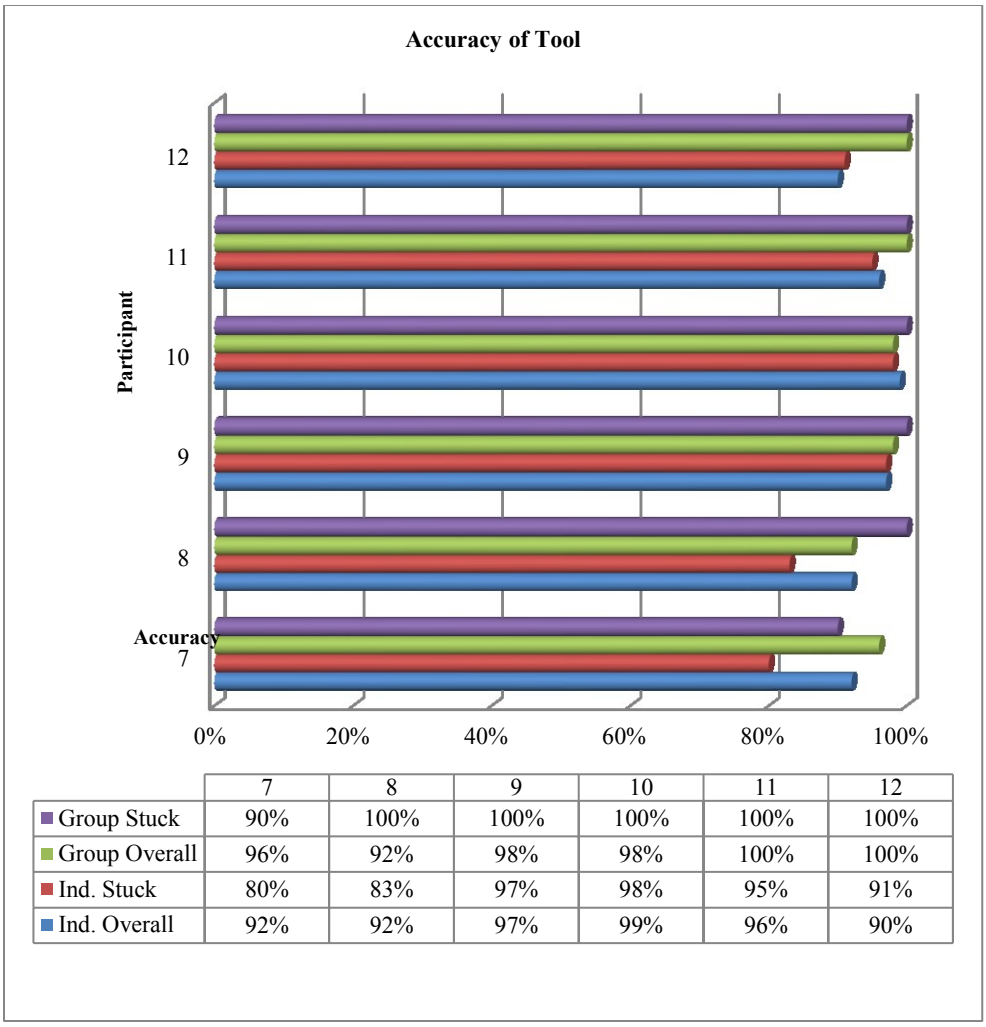
	1	2	3	4	5	6
Group Stuck	75%	100%	100%	33%	100%	50%
Group Overall	96%	98%	100%	93%	100%	99%
Ind. Stuck	75%	94%	96%	95%	100%	98%
Ind. Overall	94%	97%	98%	97%	92%	98%

Figure 3.11a: Accuracy of tool (participants 1-6)

To build and evaluate the individual model, we used a standard technique, known as cross validation, which executes 10 trials of model construction, and splits the data so that 90% of the data are used to train the algorithm and 10% of the data are used to test it. In some of the participant's training sets, the number of “making progress” segments vastly outnumbered the number of “having difficulty” segments, resulting in low accuracy in predicting the "having difficulty" segments. This is an example of the class imbalance problem in classification algorithms, wherein the accuracy of predicting an event can decrease as the frequency of a rare but important event decreases. The SMOTE [7] algorithm implemented in the WEKA toolkit [59] overcomes this problem by replicating rare data records until that data are equal to the more common data.

Therefore we used this scheme in the data sets of those participants who experienced the class imbalance problem. In our case, we used an accuracy threshold of 90% to determine if a

participant experienced this problem, which was the accuracy of our previous approach. The accuracy of the model without SMOTE was 66% or less for participants who had difficulty 20% or less of the time. For participants who had difficulty more than 20% of the time, the accuracy of the model without SMOTE was 94% or more. Thus, according to our threshold, participants who had difficulty less than 20% of the time faced the class imbalance problem. For these participants, we used SMOTE to replicate the "having difficulty" segments. In the case of the remaining participants," having difficulty" was either less or about as frequent as "making progress." Thus, there was never a need to use SMOTE to replicate the "making progress" segments. Three of the twelve participants faced so much difficulty that they did not complete two of the three exercises.



	7	8	9	10	11	12
Group Stuck	90%	100%	100%	100%	100%	100%
Group Overall	96%	92%	98%	98%	100%	100%
Ind. Stuck	80%	83%	97%	98%	95%	91%
Ind. Overall	92%	92%	97%	99%	96%	90%

Figure 3.11b. Accuracy of tool (Participants 7-12)

To build the model for a particular individual, we used that individual's data as both the training and test set. To build the group model, we aggregated the data from all of our participants except data from the participant whose status we were trying to automatically predict. The exclusion was meant to test if a tool trained by one set of developers could be used to predict the status of another. We used the group data to predict the status of each individual.

The group data set did not suffer from the class imbalance problem because some of the participants had difficulty just as much as they were making progress. As mentioned before, even

those who made relatively smooth progress experienced some difficulty. The decision tree algorithm [59] was used to build both the individual and group models.

Figures 3.11a and 3.11b show the accuracy of the tool. We considered four accuracies: (a) group stuck: the accuracy of the group model when predicting having difficulty, (b) individual stuck: the accuracy of the individual model when predicting having difficulty, (c) group overall: the accuracy of the group model when predicting both making progress and having difficulty, and (d) individual overall: the accuracy of the individual model when predicting both making progress and having difficulty. The accuracies are shown for all but two participants. These two participants were not included because their data was not collected correctly.

We expected each individual's model to be more accurate than the group model, but surprisingly, the group model was more accurate in predicting both “having difficulty” and “making progress” than the individual model. This unintuitive result is likely because the group model has more data than the individual model. It is possible that with more training, the individual model would perform better.

Even then, it may not be the preferable approach because participants, probably, would not like training the tool. In fact, during the debrief one participant commented that pressing buttons "*stopped my flow of thought"* and another participant felt that pressing buttons *"sort of broke my concentration.”*

We asked participants if they preferred to speak their status because this could help reduce breaking their concentration (Table 3.14). Participants did not like this feature either, and felt it would be disruptive to those around them.

Table 3.14: Survey Questions and Results (Scale: 1 = Strongly Disagree to 7 = Strongly agree).

	Survey Question	Mean	Median	STDDEV
Q1	I felt that the tool was accurate.	6	6	.95
Q2	I would prefer to use a speech interface (speaking your status) instead of pressing buttons to correct the status.	2.83	3	1.53

There were two participants whose accuracy was 50% or below. We examined these cases and determined that the tool believed these participants were making progress while human observers believed the participants were stuck. In each case, the participants were performing significant edits, which indicated to the tool that they were making progress. However, these edits involved a large number of deletions. This kind of activity suggests that, when extracting features, editing actions should be split into two categories: insertion and deletion of text.

3.9.1 Comparison of Group Model Results to Baselines

The evaluations above show that it is possible to increase the agreement between a tool and a set of observers by (a) keeping the exercises the same in the training and evaluation set, and (b) using the judgments of these observers in the training set. Additional iterations are required to determine if (a) a tool trained using one set of exercises can be used to predict the status for another set of tasks, and (b) judgments of one set of observers can be used to agree with the judgments of another set of observers. To provide more evidence to support our *Programming Activity Difficulty Detection* sub-thesis, we compare the results of the group model based on both observers' and developers' perceptions to the baselines described in section 3.3. The reason is the group model outperformed the individual model.

We use the same approach described in section 3.6.1 to compute each baseline. Table 3.15 shows the results for the random baseline. The accuracy of this baseline is 50%, the true positive rate is 50%, the true negative rate is 50%, the false negative rate is 50%, and the false positive rate is 50%. This baseline identifies 50% of the time when developers are making progress and 50% of the time when they are having difficulty. As mentioned above, we use the binomial test to determine if there is a significant statistical difference between the random and data distribution baselines and the group model that uses observers' perceptions as ground truth. The group model that uses observers' perceptions as ground truth performs significantly better than the random baseline (TPR=76% vs. TPR=50%, $p < .02$) (TNR=100% vs. TNR=50%, $p < .001$).

Table 3.15: Confusion matrix for random baseline (observers' data group model).

	Predicted Stuck	Predicted Progress
Actual Stuck	38 (True positives)	38 (False negatives)
Actual Progress	1106 (False positives)	1106 (True negatives)

Table 3.16 shows the results for the modal baseline. The accuracy of this baseline is 97%, the true positive rate is 0%, the true negative rate is 100%, the false negative rate is 100%, and the false positive rate is 0%. This baseline always identifies when developers are making progress, but never identifies when they are having difficulty. The true positive rate (76%) of the group model that uses observers' perceptions as ground truth is better than true positive rate (0%) of the modal baseline.

Table 3.16: Confusion matrix for modal baseline (observers' data group model).

	Predicted Stuck	Predicted Progress
Actual Stuck	0 (True positives)	76 (False negatives)
Actual Progress	0 (False positives)	2212 (True negatives)

Table 3.17 shows the results for the data distribution baseline. The accuracy of this baseline is 58%, the true positive rate is 4%, the true negative rate is 97%, the false negative rate is 96%, and the false positive rate is 3%. This baseline identifies 97% of the time when developers are making progress, but only 4% of the time when they are having difficulty. There is no significant difference between the true positive rate based on observers' perceptions as ground truth and the data distribution baseline (TPR=76% vs. TPR=4%, $p > .56$). There is a significant difference between the true negative rate of the group model that uses observers' perceptions as ground truth and the data distribution baseline (TNR=100% vs. TNR=58%, $p < .001$).

Table 3.17: Confusion matrix for data distribution baseline (observers' data group model).

	Predicted Stuck	Predicted Progress
Actual Stuck	2 (True positives)	74 (False negatives)
Actual Progress	66 (False positives)	2146 (True negatives)

These results show that the group model based on observers' perception performs better than the baseline measures. To determine whether the group model based on developers' perceptions performs better than baseline measures, as before, we compute each baseline using the model's data.

Table 3.18 shows the results for the random baseline. The accuracy of this baseline is 50%, the true positive rate is 53%, the true negative rate is 50%, the false negative rate is 47%, and the false positive rate is 50%. This baseline identifies 50% of the time when developers are making progress and 53% of the time when they are having difficulty. As mentioned above, we use the binomial test to determine if there is a significant statistical difference between the baselines and the group data model that uses developers' perceptions as ground truth. The group model based on developers' perceptions performs significantly better than the random baseline (TPR=76% vs. TPR=50%, $p < .004$) (TNR=93% vs. TNR=50%, $p < .001$).

Table 3.18: Confusion matrix for random baseline (developers' data group model).

	Predicted Stuck	Predicted Progress
Actual Stuck	9 (True positives)	8 (False negatives)
Actual Progress	603 (False positives)	602 (True negatives)

Table 3.18 shows the results for the modal baseline. The accuracy of this baseline is 97%, the true positive rate is 0%, the true negative rate is 100%, the false negative rate is 100%, and the false positive rate is 0%. This baseline always identifies when developers are making progress, but never identifies when they are having difficulty. The true positive rate (76%) of the

group model based on developers' perceptions as ground truth is better than true positive rate (0%) of the modal baseline.

Table 3.18: Confusion matrix for modal baseline (developers' data group model).

	Predicted Stuck	Predicted Progress
Actual Stuck	0 (True positives)	17 (False negatives)
Actual Progress	0 (False positives)	1205 (True negatives)

Table 3.19 shows the results for the data distribution baseline. The accuracy of this baseline is 94%, the true positive rate is 6%, the true negative rate is 96%, the false negative rate is 94%, and the false positive rate is 4%. This baseline identifies 96% of the time when developers are making progress, but only 6% of the time when they are having difficulty. The group model that uses developers' perceptions as ground truth performs significantly better than the data distribution baseline (TPR=76% vs. TPR=6%, $p < .001$) (TNR=100% vs. TNR=.93%, $p < .001$).

Table 3.19: Confusion matrix for data distribution baseline (developers' data group model).

	Predicted Stuck	Predicted Progress
Actual Stuck	1 (True positives)	16 (False negatives)
Actual Progress	54 (False positives)	1168 (True negatives)

3.10 Privacy

So far, we have assumed that letting others know about difficulties of others is good. This assumption is probably true when the observers are mentors/advisors, as suggested in [3]. However it is possible to have observers who judge programmers without actually helping them. These judgers can use information about developers being stuck repeatedly in a negative manner, which could cause programmers to lose respect in their team. Even when observers can be trusted, the developers may want more time to investigate their problems. There are several ways to solve this problem. One approach is to block judgers, a feature readily available in Google Talk and other IM clients. The problem with this approach is that blocked judgers can realize that they are blocked, which could cause them to become hostile. Therefore, a superior approach is to enable programmers to decide which status they want to report. Figure 6 shows a preliminary scheme we have implemented to support this feature, which is also used by developers to train the system. This interface reports two statuses – the true status and the reported status. Buttons are provided to change both statuses.

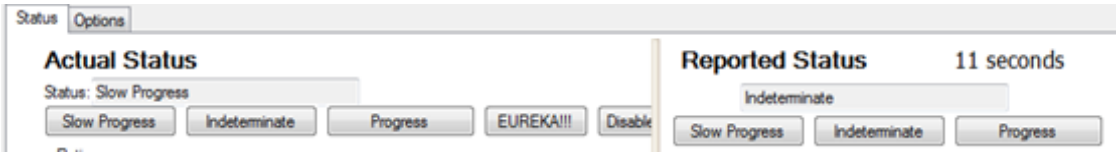


Figure 3.12: Training user interface that show actual versus report status.

The buttons that change the true status are used to train the system and the buttons that change the reported status determine what others on their buddy lists see. The true status field is automatically copied to the reported status field after a certain time lag. During this time, developers can manually disable the copying. Assuming that having difficulty is indeed a rare event, this user-interface does not impose substantial overhead.

We have not formally evaluated these privacy controls, but we have gotten some initial feedback from those who have used them. Users would indeed like to customize not only what status is reported, but also when it is reported, and to whom it is reported. Thus, this scheme must be extended to control the nature and timing of reported status for different classes of observers such as (a) human observers and tools, (b) a team member sitting on the next seat, radically co-located, and distributed, (c) a close friend, mentor, and boss, and (d) team members who have and do not have the expertise to help solve a problem.

Such elaborate customization could make the overhead required to use the tool high. Future versions of this scheme must enable for setting user-specific defaults. For example, the number of IM messages with team members can be used to identify close friends; organization charts can be used to find mentors and bosses; location information can be used to find the physical distance between developers and various observers; and the difficulty each team member has with different pieces of a project can be used to find expertise. In addition, the tool can adapt how developers morph the reported status. For instance, if they always report the indeterminate status to their boss, then the tool could ask them if they wish to set this value automatically for this observer.

3.11 Limitations

A central limitation in this chapter is the size of both studies.

Size of studies: Both studies in this chapter had a limited sample size, which could limit the generalizability of the studies. The first study had six developers and the second study had 14 developers. The number of participants in the second study is typical for studying software developers.

Developer experience: Developers in the first study were students, which could limit the findings to education. Developers with more industry experience may perform different programming actions when they are having difficulty. The second study partially addresses this limitation by using five industrial participants. However, as mentioned above, this study also had a limited sample size (14 developers).

Maintenance tasks: In both of our studies, developers implemented programs from scratch, which could mean our results may not apply to maintenance tasks. Maintenance tasks can be expected to have more navigation for the same difficulty degree.

Difficulty-Detection Algorithm Performance: A limitation of the difficulty detection algorithm is that it has a high false negative rate. This limitation can be addressed in a number of ways. One way is to determine the most accurate machine learning algorithm. In this chapter, we tried several machine learning algorithms and showed that the decision tree algorithm performed the best. Another way is to investigate different approaches that address the class imbalance problem. We use the SMOTE algorithm, which is a form of oversampling, increasing the size of the minority class (in our case, having difficulty). There are additional approaches such as undersampling, decreasing the size of the majority class (in our case, making progress) and cost-sensitive learning. In a two-class problem, as in our case, cost sensitive learning assigns a cost to both classes (making progress and having difficulty). Difficulty detection modules can

use these cost to a) change the distribution of the data set according to the costs or b) only predict the high-cost class when the module is confident about the prediction. It would also be useful to investigate different log segmentation methods such as segmenting the log based on time or using a sliding window approach. An alternative is to investigate additional features such as idle time, the amount of pressure on a mouse or keyboard, or features that can be computed from non-standard equipment such as body posture. The performance of the difficulty detection algorithm is addressed in the next chapter.

Ground Truth: Another limitation is using developers' as ground truth. The argument for using developers' perceptions is that they are programming and should know whether they are having difficulty. However, people tend to underestimate their problems, which could mean developers may not always admit when they are having difficulty. Given that having difficulty is a rare event and that developers may not admit when they are having difficulty, we could miss collecting a large amount of data. To reduce the chance of missing a large amount of data, our study involved three observers. The first observer, I, watched participants while they were programming and labeled times when I thought they were having difficulty. Studies that rely on the experimenter to label data for difficulty detection modules are subject to experimenter's bias. To reduce this bias, we recruited two additional observers who were not associated with the experiment to blindly label the making progress and having difficulty moments indicated by the first observer and participants.

Privacy: We did not formally evaluate privacy controls, but we have gotten some initial feedback on users' preferences.

3.12 Summary

To summarize, we have described our programming-activity difficulty-detection component. This component overcomes the overhead and privacy limitations of previous

approaches by logging developers' interactions with programming environments and inputting these actions into a difficulty detection module that predicts whether developers are having difficulty or making progress. To evaluate this component, we describe several performance metrics, define three baselines, and conduct a small field study (6 participants). Our results show that the component performs better than baseline measures.

We also evaluate this component in a lab study (14 participants) with more participants than our field study (6 participants). Our results show that the framework performs better than baseline measures when using the perceptions of observers and developers as ground truth. These results combined with the results from our field study provide evidence to support sub-thesis I, which we restate here.

Programming Activity Difficulty Detection Sub-Theses (Sub-thesis I):

It is possible to develop an approach that a) uses developers' interactions with their programming environment to determine whether developers are having difficulty with their task and b) performs better than baseline measures.

To determine how well programming-activity difficulty-detection works in practice, we develop the reusable difficulty detection framework, which uses standard design patterns, Mediator and Strategy, to enable the component to be used in two programming environments. Our results show that the number of lines of code to implement the reusable difficulty detection framework, 4,643 is significantly less than the number of lines of code to implement difficulty detection modules written specifically for Visual Studio (9,096) and Eclipse (11,000). These results provide evidence to support sub-thesis II, which we restate here.

Implementation Sub-Theses (Sub-thesis II): It is possible to develop a common set of difficulty detection modules for different programming environments that have significantly fewer lines of code than difficulty detection modules written specifically for each programming environment

Chapter 4: Multimodal Difficulty Detection

4.1 Introduction

Our evaluation of the programming activity-detection component, in the previous chapter shows that the component gives a high false negative rate. A high false negative rate is an important issue because developers who are having difficulty may not get help when they need it. Thus, they may waste a significant amount of time having difficulty.

To address this limitation, we follow the iterative process developed in the previous chapter. In particular, we start at step 4 in the iterative process, which is refining our set of features. We refine the programming activity feature set based on suggestions in the previous chapter. Specifically, we implement two new features, insertion and deletion ratios, and evaluate these features with our other programming activity features mentioned in the previous chapter. Next, we combine the refined programming activity features with features from non-standard equipment. As mentioned before, a limitation of non-standard equipment is the overhead of using it. However, some developers may be willing to use it.

As mentioned above, previous work has used many types of non-standard equipment such as posture seating chairs, wireless Bluetooth skin conductance tests, pressure mice, and video cameras for predicting frustration. In particular, Kapoor et al. showed that posture, captured by posture seating chairs was the most predictive feature. Based on this result, we decided to use non-standard equipment that captured developers' postures to predict whether they were having difficulty. Using posture to predict developers' difficulty status is novel because to our knowledge, no other work has used it to predict difficulty. However, we did not

use posture seating chairs because we did not have access to them and more important, they are not widely available. If we use non-standard equipment that is more widely available our work can be applied to a greater number of domains.

A particularly interesting form of non-standard equipment that is a) novel for difficulty detection because to our knowledge, it has not been used in previous work, b) more widely available than posture seating chairs, and c) captures data about users' posture is the Microsoft Kinect camera. Another interesting form of non-standard equipment that is novel and was made available to us is the Creative® Interactive Gesture camera. We use both of these cameras as non-standard equipment.

Given that our small field study and lab study in the previous chapter only use programming environments, we need to conduct a new lab study with the Microsoft Kinect camera and Creative® Interactive Gesture camera. In this chapter, we describe a lab study and evaluation we conducted using both programming environments and non-standard equipment.

The rest of this chapter is organized as follows. First, we describe the new user study. Second, we present results of using two additional features, insertion and deletion ratio to determine when programmers are having difficulty. Third, we discuss the results from using body posture as a feature. Fourth, we discuss the results from combining both body posture and programming activity features. Fifth, we discuss limitations of our work. Finally, we end with a brief summary.

4.2 User Study

To provide evidence to support our ***Multimodal Difficulty Detection Sub-theses,*** we performed a controlled lab study. As in the previous chapter, we had to choose our tasks carefully because having difficulty is a rare event. Therefore, we must ensure developers face difficulty in the small amount of time available (1-2 hours) for a lab study, and yet do not find

the problems impossible.

After experimenting with different tasks, we chose the tasks that required participants to use the AWT/SWT toolkit. To ensure that developers would face difficulty during the study, but the tasks would not be too difficult to solve, we performed several pilot studies. Based on these studies, we settled on the tasks shown in Table 4.1.

Table 4.1: Participants' tasks.

Tasks
Create a program that visually represents a car with a red body and two black tires.
Enable the user to use arrow keys to move around the car in any direction (up, forward, left, and right) by 10 pixel decrements/increments.
Enable the user to make the car a bus by clicking anywhere on the screen. A bus has an extra body that should be colored black. It should be positioned directly on top of the car. When the extra body is on top of the car, it should move with the rest of it.
Enable the user to make the bus a car by pressing the 'r' key. The extra body should be removed
Enable the user to scale up the car/bus 2X, each time they press the 'm' key
Enable the user to scale down the car/bus 2X, each time they press the 's' key
Draw a transparent square (not a rectangle) with yellow borders. The car/bus should be inside the square.
Do not allow the car/bus to go outside of the square (when moving and resizing the vehicle).

Ten student programmers participated in the study. They were given an hour and a half to complete their tasks and were free to use the Internet. We logged participants' programming activities using our difficulty detection tool. This tool also predicted whether students were having difficulty or making progress during the study. Participants were instructed to correct an

incorrect prediction by the system using status-correction buttons shown in Figure 4.1. Additionally, participants could ask for help, by pressing the "I am asking someone for help" button. After participants pressed this button, they were instructed to discuss their issue with me. Help was given in the form of URLs to API documentation or code examples. By measuring how often the developers corrected their status, we could, as in the previous chapter, measure the accuracy of our approach with respect to the perceptions of the developers.

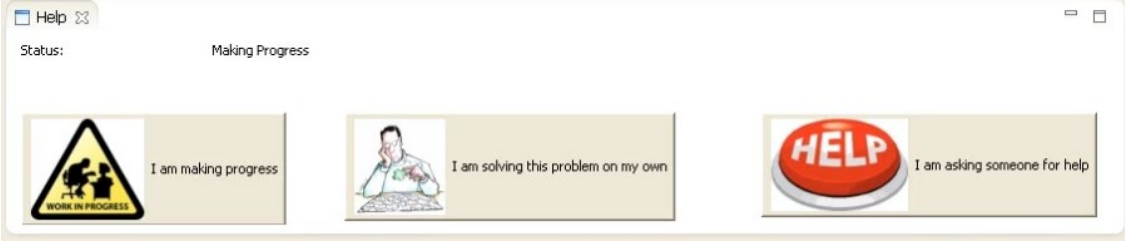


Figure 4.1: Status correction and indication buttons.

However, as in the previous chapter, there is a question as to whether participants would accurately report their status. Shrauger and Osberg [52] found that participants tend to underreport their problems. Therefore, I observed participants' programming activities and made an independent determination of their status. We used Cisco WebEx Web Conferencing® to observe participants' programming activities and Camtasia Studio® to record participants' screens. The screen recordings were used during the debrief to show participants portions of the screen recording where I indicated they were having difficulty and to enable them to confirm or deny. Their confirmations, button corrections, and explicit help requests were used as ground truth.

4.3 Programming Activity Results

There were 814 predictions during the study. To evaluate, the programming activity algorithm we use the metrics in Chapter 3 (accuracy, true positive rate, true negative rate, false

positive rate, and false negative rate). The confusion matrix, shown in Table 4.2, shows the results.

Table 4.2: Confusion matrix for initial programming activity algorithm.

	Predicted Stuck	Predicted Progress
Actual Stuck	11 (True positives)	44 (False negatives)
Actual Progress	0 (False positives)	759 (True negatives)

Participants had difficulty 55 times during the study and never corrected the tool when it predicted that they were having difficulty (11 times). Thus, the false positive rate is 0%. However, the false negative rate is 80%, which means, the tool missed 80% (44/55) of the times developers were having difficulty. More specifically, participants corrected the tool's making progress predictions 47% (26/55) of the time and agreed with me that they were having difficulty 33% (18/55) of the time. These results are consistent with difficulty detection results in Chapter 3. Naturally, it is attractive to try to reduce the false negative rate without increasing the false positive rate. In Chapter 3, we gave two possible reasons that the tool missed more than half the times that participants in their experiments had difficulty.

First, participants who were having difficulty made a significant number of edits, which indicated to the tool that they were making progress. However, these edits involved a large number of deletions. This kind of activity suggests that when computing features, editing actions should be split into two categories: insertion and deletion of text.

During our study, I also observed that participants made a large number of deletions

when they were having difficulty. Therefore, we analyzed data for all participants to confirm these observations. To show our findings, we graph the results for one participant. In the graph (Figure 4.1), the x-axis is the session time and the y-axis is the percentage for the edit, insertion, and deletion ratios. Figure 4.2 shows that participant 1's edit ratio is high even though he is having difficulty and when the edit ratio is split into insertion and deletion ratios that deletions make up the majority of the edit ratio. This analysis led us to a) split the edit ratio into insertion and deletion ratios and b) re-evaluate the programming activity algorithm with the new features. To re-evaluate the programming activity algorithm, we used k-fold cross validation, as described in the previous chapter, where k is equal to 10.

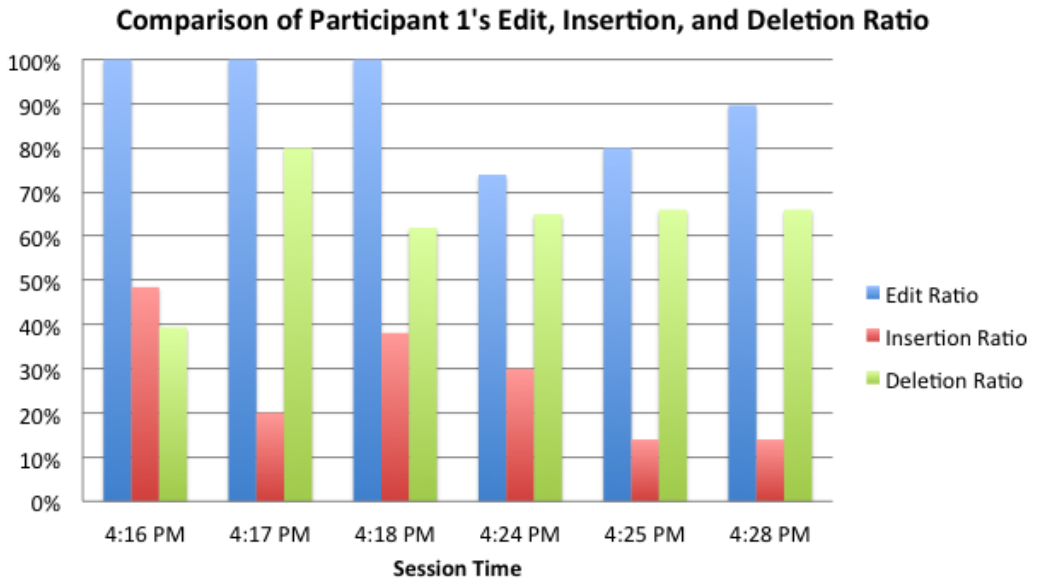


Figure 4.2: A comparison of participant 1's edit, insertion, and deletion ratio.

Table 4.3, the improved programming activity algorithm confusion matrix, shows our results. When compared to the original programming activity algorithm, the false negative rate decreased from 80% to 27%, the false positive rate increased slightly from 0% to 3%, and the accuracy of the algorithm increased from 20% to 72% (40/55). These results are a significant improvement over the results from the previous algorithm. However, the improved algorithm

still missed 27% of the time that participants had difficulty.

Table 4.3: Confusion matrix for improved programming activity algorithm.

	Predicted Stuck	Predicted Progress
Actual Stuck	40 (True positives)	15 (False negatives)
Actual Progress	20 False positives)	739 (True negatives)

The second reason we gave for the tool missing more than half the times participants had difficulty is that the tool does not consider idle time, which may be important. Idle time may indicate that participants are not sure of how to proceed on their tasks.

A problem with idle time is that it is difficult to distinguish between different types of idle time. For example, developers may be idle because they are thinking, taking a break, or truly having difficulty. In our case, because we performed a lab study with a time limit and developers did not take a noticeable break. However, we would still have to distinguish between developers thinking and having difficulty.

4.4 Tracking Body Posture

An alternative approach is to use non-standard equipment to determine difficulty. Previous work has used this approach to determine developers' emotional state such as frustration or confusion. Our goal is to use non-standard equipment to determine when developers are having difficulty. To meet this goal, we also logged participants' images and postures using the Creative® Interactive Gesture Camera and the Microsoft Kinect camera**.** We first describe the results from the Creative® Interactive Gesture Camera and second we describe the results from the Microsoft Kinect Camera.

4.4.1 Creative® Interactive Gesture Camera

Figure 4.3 shows the placement of the Creative® Interactive Gesture Camera in our

experimental setup. This camera captured facial positions, shown in Figure 4.4 at 30 fps. The white square is shown when the camera has captured the facial positions. White is the default color of the square, but we changed it to red.

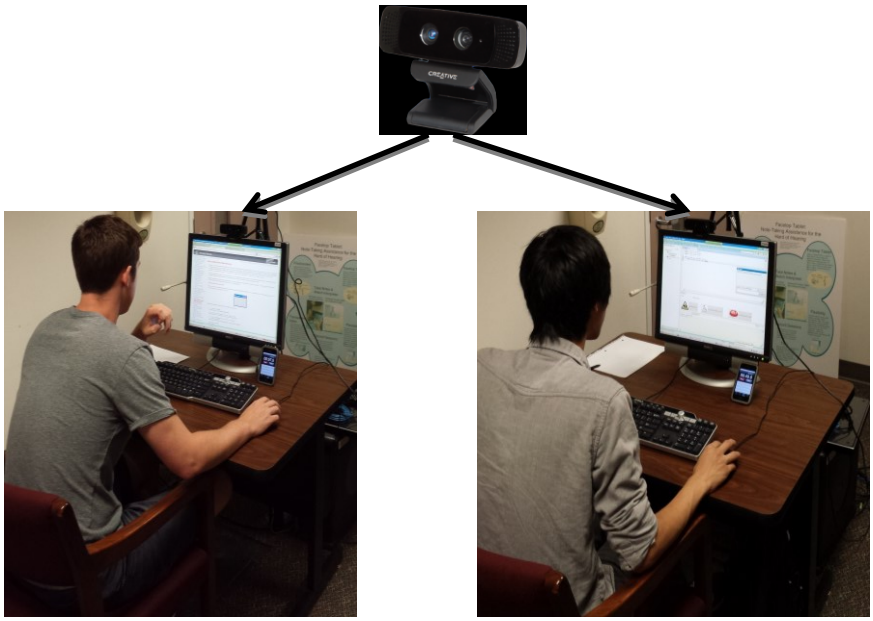


Figure 4.3: Experimental setup that shows placement of the Creative® Interactive Gesture Camera.

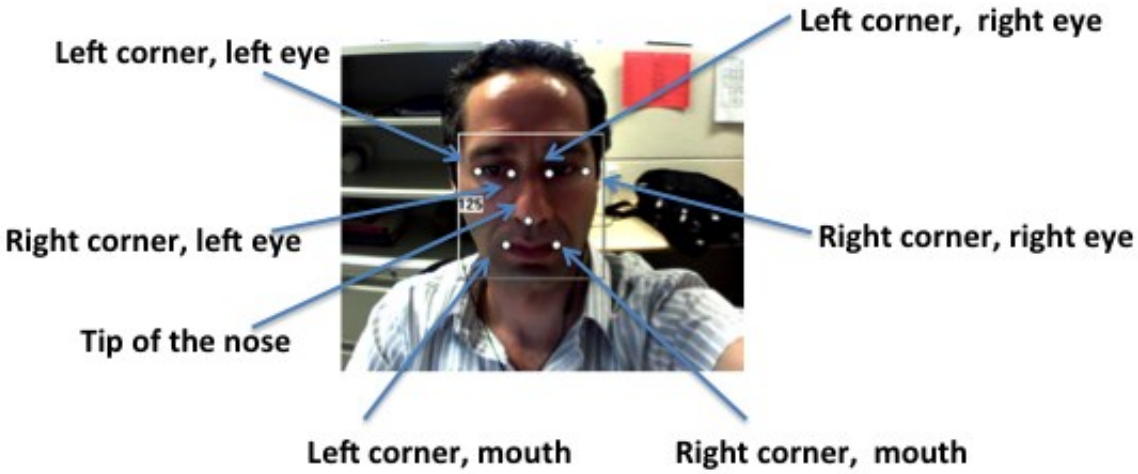


Figure 4.4: Facial positions measured by the Creative® Interactive Gesture Camera.

There were a few times, shown in Figure 4.5, where the camera did not capture the facial positions correctly. In the left picture, the participant leans in very close to the camera. In the middle picture, the participant puts his hand on his mouth. In the right picture, the camera

recognizes the participant's hand as her face. Despite these limitations, we were able to capture most facial positions correctly.

Figure 4.5: Examples of when the camera did not capture facial positions.

Given the facial positions, we computed posture as mouth distance, right eye distance, and left eye distance. Mouth/right eye/left eye distance is the difference between the left and right corners of the mouth/right eye/left eye. The intuition behind these metrics is that the bigger the mouth, right eye, and left eye distance, the closer participants are to the camera and the smaller the mouth, right eye, and left eye distance, the further away participants are from the camera. D'Mello and Grasser have shown that body lean correlated with students' affective states such as boredom or delight [10]. More specifically, when participants are confused or frustrated, they tend to lean forward and when they are bored they tend to lean back. In our case, the hope is that body lean would correlate with participants who are having difficulty. Therefore, we used these distance measures to capture whether participants were leaning forward, normal, or leaning backward. By normal, we mean participants are not leaning forward or leaning back.

To convert mouth, right eye, and left eye distances to body lean (leaning forward, normal, leaning back) we averaged each participants' distance measure per minute and clustered each participants' data individually using the K-means clustering algorithm. Given the distance values and k, the number of clusters to produce, the algorithm partitions values into clusters based on the average Euclidean distance between them. We examined the algorithms' output with two, three, and four clusters. We decided to present the output with three clusters because

a) as mentioned above, previous work has shown that leaning back, normal, and leaning forward correlated with affective states, b) the standard deviation of one cluster was large when using two clusters, and c) four clusters offered no improvement over three.

To understand the clustered data, we looked for examples of participants' postures in the images taken by the camera. Figure 4.6 shows examples of each type of posture from three participants.

Figure 4.6: Examples of participants' leaning back, normal, and leaning forward postures (Creative® Interactive Gesture camera).

We used each developer's posture per minute as features. Before we could feed this data into a machine learning algorithm, we had to determine, as we did in the previous chapter, whether we would use individual models where training data for a developer is only used for that developer or a group model that developers do not need to train. Our intuition was to use an individual model because no developers may be in the same posture when they are having difficulty. However, we chose to try both individual and group models to evaluate which model would perform better.

First, we built individual models for each developer, which consisted of feeding each

developer's postures per minute into the decision tree algorithm and using 10-fold cross validation to evaluate it. The confusion matrix in Table 4.3 shows the results of the individual models. The accuracy of this model is 57%, the true positive rate is 82%, the true negative rate is 31%, the false negative rate is 18%, and the false positive rate is 69%. These results show that the model missed 18% of the time when developers were having difficulty and 69% of the time when developers were making progress.

Table 4.3: Confusion matrix for Creative® Interactive Gesture camera (individual model).

	Predicted Stuck	Predicted Progress
Actual Stuck	378 (True positives)	81 (False negatives)
Actual Progress	306 (False positives)	135(True negatives)

Next, we built a group model. To do this, we labeled all postures that were leaning back or leaning forward as "NOT NORMAL" and all normal postures were labeled as "NORMAL." We fed this information into a decision tree algorithm. The confusion matrix in Table 4.4 shows results of the group model. The accuracy of this model is 23%, the true positive rate is 33%, the true negative rate is 12%, the false negative rate is 67%, and the false positive rate is 88%. When compared to the individual model, the accuracy decreased from 57% to 23%, the true positive rate decreased from 82% to 33%, the true negative rate decreased from 31% to 12%, the false negative rate increased from 18% to 67% and the false positive rate increased from 69% to 88%. These results show that the model missed 67% of the time when developers were having difficulty and 88% of the time when developers were making progress.

The individual model has a high false negative rate and the group model has a high false negative and false positive rate. None of these results is good. One possible reason is that mouth

distance, left eye distance, and right eye distance, are not good measures of body lean. Another reason could be the limitations of the Creative® Interactive Gesture camera mentioned above. One way to overcome these limitations is to use a camera that better measures body lean. One such camera is the Kinect, which measures users' physical distance from the camera in meters.

Table 4.4: Confusion matrix for Creative® Interactive Gesture camera (group model).

	Predicted Stuck	Predicted Progress
Actual Stuck	153 (True positives)	306 (False negatives)
Actual Progress	387 (False positives)	54 (True negatives)

4.4.2 Microsoft Kinect Camera

Figure 4.7 shows the placement of the Microsoft Kinect camera in our experimental setup. This camera captures the x, y, and z coordinates of 20 joints shown in Figure 4.8 at 5 fps. These joints represent the human body. The Kinect camera could not capture all of the joints because the desk that developers sat at occluded their lower body. This limitation made it difficult for the camera to capture any of the lower body joints. To support users who are sitting at a desk, the Kinect also has a seated mode, which is designed to track users who sit, and does not attempt to capture joints below the hip. Figure 4.9 shows the difference between the standing and seated modes. We used the seated mode to capture participants' joints in our study.

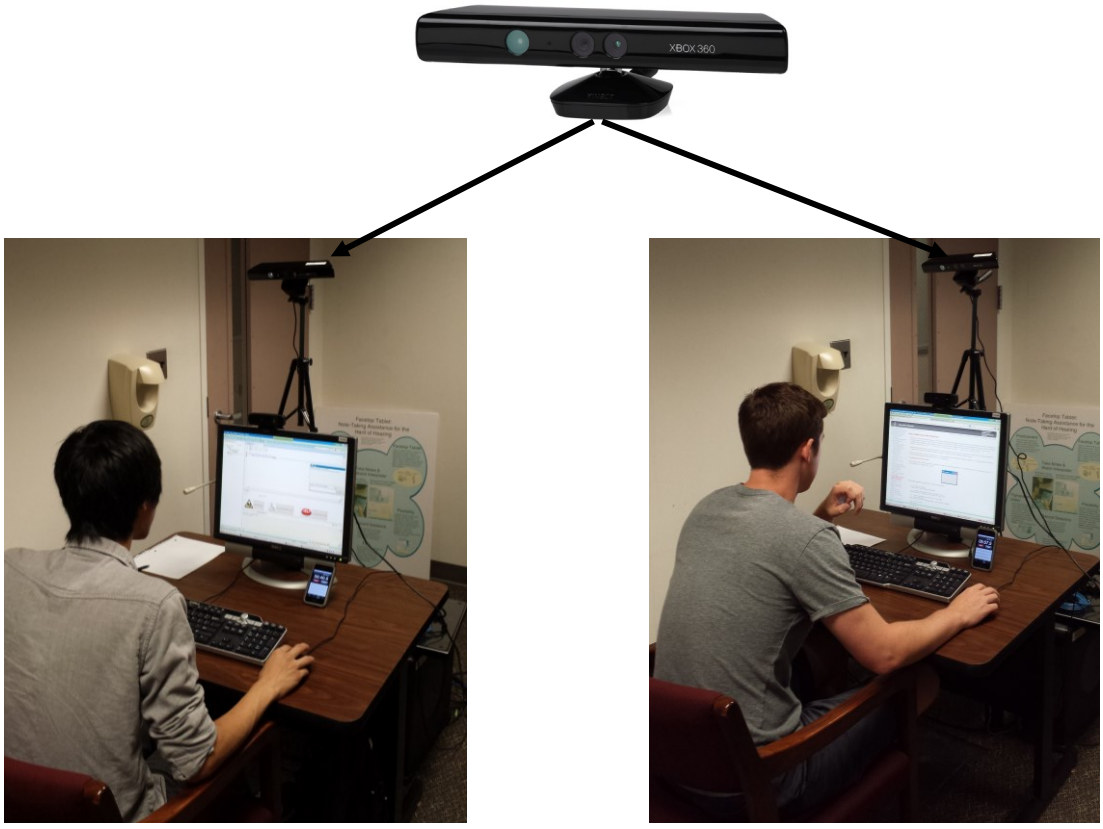


Figure 4.7: Experimental setup that shows placement of the Microsoft Kinect camera.

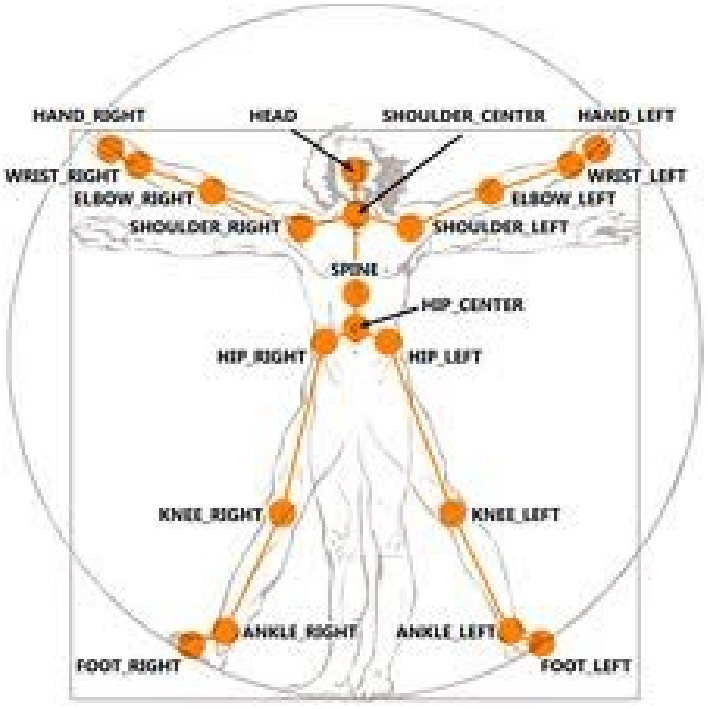


Figure 4.8: The 20 joints captured by the Kinect camera.

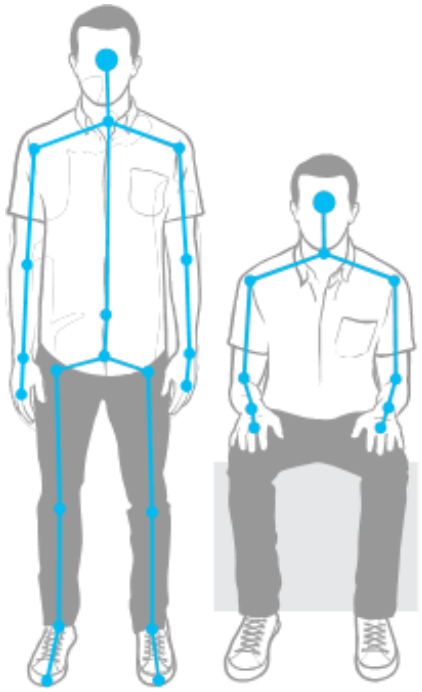


Figure 4.9: Differences between joints in standing and seat mode.

There were times where the Kinect camera did not capture the arm and shoulder joints well, but it did capture the head joint correctly the majority of the time. Therefore, we used the distance from the head joint to the Kinect camera to measure body lean. This distance was the z coordinate of the head joint in meters. To convert the head joint distance to body lean, we used the same process as the Creative® Interactive Gesture camera (using K-means to cluster the data). To better understand the clustered data, we looked for examples of participants' postures in the images taken by the camera. Figure 4.10 shows examples of each type of posture from three participants. As with the Creative® Interactive Gesture camera data, we used each developer's posture per minute as features and built individual models for each developer and group models.

Figure 4.10: Examples of participants' leaning back, normal, and leaning forward postures (Kinect camera).

The confusion matrix in Table 4.5 shows the results of the individual model. The accuracy of this model is 74%, the true positive rate is 73%, the true negative rate is 75%, the false negative rate is 27%, and the false positive rate is 25%. When compared to the individual model from the Creative® Interactive Gesture camera, the accuracy increased from 57% to 74%, the true positive rate decreased from 82% to 75%, the true negative rate increased from 31% to 75%, the false positive rate decreased from 69% to 28%, and the false negative rate increased from 16% to 30%. These results show that the Kinect camera individual model has a lower false positive rate and a slightly higher false negative rate than the Creative® Interactive Gesture

camera model. The latter is more desirable because it does not unnecessarily waste the time of the developers and those who offer help.

Table 4.5: Confusion matrix for Kinect camera (individual model).

	Predicted Stuck	Predicted Progress
Actual Stuck	333 (True positives)	126 (False negatives)
Actual Progress	108 (False positives)	333 (True negatives)

We also evaluated a group model and used the same technique mentioned in the previous section to create it. Table 4.6 shows the results of the group model. The accuracy of this algorithm is 62%, the true positive rate is 69%, the true negative rate is 55%, the false negative rate is 31%, and the false positive rate is 45%. None of these results is good when compared to the programming activity approach. One possible reason is that frustration is not equal to difficulty. More specifically, some people are calm and do not change posture when in difficulty, while some people are fidgety and change posture when making progress. So was this a wasted effort?

Alone, the body posture approach does not work well, but we combine group data from the programming activity and the body posture approaches to determine if this combination will give greater accuracy and fewer false negatives than the programming activity approach. More specifically, we will use the Kinect camera for the body posture approach because it gives better results than the Creative® Interactive Gesture camera and combine it with the improved programming activity approach because it gives better results than the original programming activity approach.

Table 4.6: Confusion matrix for Kinect camera (group model).

	Predicted Stuck	Predicted Progress
Actual Stuck	315 (True positives)	144 (False negatives)
Actual Progress	198 (False positives)	243 (True negatives)

4.5 Combining Body Posture and Programming Activity Tracking

One way to combine the posture and programming activity approaches is to use the features from both approaches together. In the individual approaches, the two kinds of features were computed at different moments - every 50 events for programming activity and every minute for body posture. In the combined approach, we still computed the programming environment features every 50 events, and we computed the body posture feature only when there was a programming activity feature. We fed these features as input to a decision tree algorithm. To evaluate this algorithm, we use k-fold cross validation, where k is equal to 10. The confusion matrix in Table 4.7 shows our results. The accuracy of this algorithm is 95%, the true positive rate is 93%, the true negative rate is 95%, the false negative rate is 7%, and the false positive rate is 5%. More specifically, the accuracy increased from 72% to 95%, the false negative rate decreased from 27% to 4%, and the false positive rate slightly increased from 3% to 4%.

Table 4.7: Confusion matrix for improved programming activity algorithm and posture.

	Predicted Stuck	Predicted Progress
Actual Stuck	53 (True positives)	4 (False negatives)
Actual Progress	35 (False positives)	724 (True negatives)

4.6 Limitations

There are three limitations that affect the performance of posture detection modules: equipment, posture, and choice of machine learning algorithm.

Equipment Limitation: The Creative® Interactive Gesture camera/Kinect camera had difficulty capturing participants' facial/body positions correctly. There are many reasons for this difficulty such as lighting conditions and device limitations.

Posture Limitation: A limitation of using posture in our work is that it can negatively affect the performance of difficulty detection modules. One way to address this limitation is to use other standard equipment such as keyboards or mice to detect difficulty. For example, Kapoor et al. showed that pressure on a mouse could be used to detect frustration [35]. Similarly, the amount of pressure put on each key of a keyboard may also be used. Despite this limitation, our posture difficulty detection module gave good results.

Choice of machine learning algorithm: In this chapter, we used the decision tree algorithm to build a posture model. It may be possible that an alternative model would give better performance. One such model is a hidden Markov model (HMM). HMMs, described in chapter 2, section 2.6.8, are probabilistic finite state machines. Each label (having difficulty or making progress) is a node in the finite state machine and the HMM provides a) the probability of transitioning from one label to the next and b) the probability that a given posture is a label.

To determine whether HMM performs better than the decision tree algorithm, we used a variation of HMM, HMM -based sequencer classifier, as implemented in the Accord.NET Framework [6]. The HMM-based sequencer classifier is trained individually on sequences of having difficulty and making progress postures; thus creating two classifiers. After each classifier is individually trained, both classifiers are given test data (sequences of postures with no label). The classifiers compute the probability that the test data belongs to it. The classifier

that outputs the highest probability is used to determine the label for the test data. After changing several parameters of the classifier such as the length of sequences and transition probabilities, we could only achieve an accuracy of 44%, a true positive rate of 57%, a true negative rate of 31%, a false negative rate of 43%, and a false positive rate of 69%. The decision tree gave better results.

4.7 Summary

To summarize, we a) split the edit ratio into insertion and deletion ratios, b) added body posture as a feature, and c) evaluated the programming activity algorithm with this new feature set. Our results show that the programming activity algorithm that uses the refined feature set gives less false negatives than the programming activity algorithm developed in Chapter 3. Given the success of these results, we combine the new programming activity feature set with body posture to create multimodal difficulty detection. These results provide evidence to support sub-thesis III, which we restate here:

Multimodal Difficulty Detection Sub-theses (Sub-thesis III): *It is possible to develop an approach that a) combines programming activity and body posture recognition to predict when developers are having difficulty with their tasks and b) has greater accuracy and a lower false negative rate (predicting stuck) than existing approaches that only use programming activities to determine when developers are having difficulty with their tasks.*

Chapter 5. Help Promotion

5.1 Introduction

In Chapter 3, we presented a) an approach that uses programming activity features to determine when developers were having difficulty, b) a common set of difficulty detection modules for two programming environments, and c) results that show the difficulty detection modules perform better than baseline measures. In the following chapter, we a) refined the programming activity feature set, b) combined the refined programming activity feature set with a body posture feature set, and c) presented results that show this combined approach has greater accuracy and a lower false negative rate than an approach that uses the refined programming activity feature set.

One question left unanswered, so far is: what are the potential benefits of difficulty awareness? Three potential benefits are estimating an individual's progress, clustering problems according to their difficulty, and teammates helping each other when they become aware of developers' difficulties. In this chapter, we focus on help promotion, the last benefit, for two reasons. First, we motivated help promotion in the introduction chapter and second and more important, to our knowledge, no work in the literature explores how manual or automatic difficulty detection can be used for help promotion.

To explore how difficulty detection can be used for help promotion, we perform two lab studies and a field study. In the first lab study, we investigate how teammates determine if they

can or should offer help. To make this determination, they need more information about teammates' context [57]. One way to provide context is screen awareness, continuous knowledge of remote users' screens with help awareness. The idea of screen awareness is not new. Tee et al. implemented screen awareness as a sidebar that contained thumbnails of remote users' screens [56]. It is not clear if this semantic awareness is manifested visually in a thumbnail. More important, having difficulty is a rare event. Therefore, observers would have to continually monitor remote developers' screens looking for when developers are stuck, essentially searching for a "needle in a hay stack".

This problem can be addressed by our programming-activity detection-component, described in chapter 3, which infers users' help status from their interaction with the programming environment and communicates this inference as help awareness to others. Observers can use such awareness to offer help when developers are stuck. Combining screen awareness with help awareness enables observers to view collaborators' screens only when they need help. This novel combination is our integrated workspace-difficulty awareness component. In a field study of this component, students were successfully offered help in a CS1 class.

The rest of this chapter is organized as follows. We first describe and evaluate integrated workspace-difficulty awareness. Next, we describe a classroom field study conducted with this component. Finally, we end a brief summary.

5.2 Context Awareness

As mentioned above, before teammates offer help, they must first determine if they can or should help. To make this determination, teammates need more information about developers' context, which is the second step in our help-giving model.

One way for teammates to make this determination is to use screen awareness, continuous knowledge of developers' screens. Teammates can use this information to see what

is blocking developers from completing their tasks. Tee et al. implemented screen awareness as a sidebar that contained thumbnails of remote users' screens [56]. Collaborators used these thumbnails to monitor each other, and when a change in a teammate's thumbnail indicated a potentially interesting event, they expanded the thumbnail to show that teammate's full screen. Sometimes, screen awareness alone does not provide enough information for potential helpers to determine if they can or should offer help because it shows only the current information on developers' screens and previous work found that teammates needed to know what developers have done to try to solve their problem [57]. We can model this problem as the classic latecomer problem, which generally occurs in scheduled meetings where individuals invited to a meeting join late and need to catch up.

The most recent work on this topic, by Junuzovic and colleagues, records audio of meetings, video of the participants' faces, and a shared workspace (presentations) and transcribes the audio recording so that latecomers can replay this information to catch up [34]. The goal of their work was to determine whether replaying audio, video, shared workspace, and transcript was better than only replaying audio. To achieve this goal, they compared the amount of information (facts, explanations, and the identity of the speaker in the meeting) that latecomers could recall after replaying audio only and different combinations of audio, video, transcript, and shared workspace. Their results show that participants who replayed the combinations of audio, video, workspace actions, and transcript and audio and workspace actions combinations recalled more information than audio alone and any other combination. This suggests that replaying audio, video, transcript, and workspace actions are all useful in helping latecomers get up to date.

However, their work did not show that replaying workspace actions, transcripts, or video alone was useful. Therefore, their results do not apply to our latecomer scenario because there is

no video or transcript to replay. The reason is that developers work alone until they get stuck and the only information available to replay is developers' screens.

5.2.1 Semi-Structured interviews

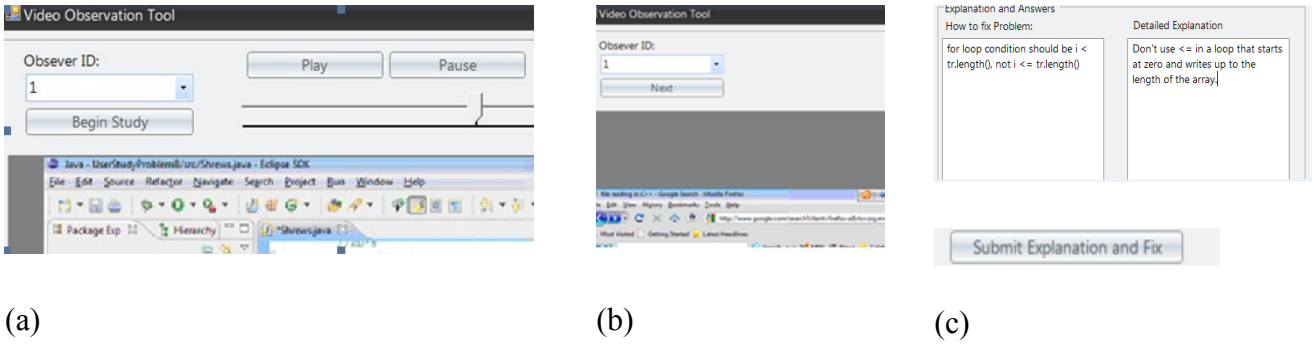
To test our intuition about the usefulness of simple screen-help awareness, viewing developers' screen without the ability to pause and replay, and buffered screen-help awareness, replaying developers' screens, in promoting help giving, we focused on the intern/mentor scenario and interviewed eight subjects - four mentor/intern pairs - in a large organization. The subjects had professional experience ranging from 3 months to 20 years. Some of the subjects were not employed as programmers, but programming played a major part of their job role. To determine the usefulness of simple and buffered screen-help awareness, we asked participants if they could give examples of simple and buffered screen-help awareness would have been useful in previous help giving interactions. To give participants a concrete idea of how simple and buffered screen-help awareness would be implemented, we showed them a prototype that replays the audio and video of meetings [34].

All subjects liked the general idea of simple and buffered screen-help awareness. For example, one intern reported that he liked that his mentor would be able to watch over his shoulder when he needed help. Similarly, one mentor reported that the tool would eliminate the overhead of scheduling multiple project status meetings with interns.

5.2.2 User Study

Encouraged by the results of the interviews, we decided to perform studies that evaluated simple and buffered screen-help awareness. A field study of simple and buffered screen-help awareness would give the most reliable evaluation, but requires robust implementations and long-term usage of these implementations. Therefore, we decided to do a lab study to motivate such implementations and usage.

There are many ways to simulate developers and observers in a lab study. One approach is to use "live" developers and observers, wherein the latter monitor developer activities as they are performed. This was the approach used in [11] in a game playing activity designed to support help giving. However, this approach has several problems when applied to traditional software development. First, as getting stuck is a rare event, this means an observer must wait or perform some other activity irrelevant to our study until the next stuck point. Enabling a subject to monitor the activities of multiple developers can ameliorate this problem, but the observers and developers must be scheduled at the same time. More important, the study must involve multiple developers and subjects to reduce the impact of subject and observer idiosyncrasies, and ways must be found to make them familiar with the developers' tasks and give the observers some expertise so they can play a potential helper (mentor/teacher) role.



(a) (b) (c)

Figure 5.1: "Stuck Point" Video Observation Tool simulating two modes: (a) buffered and (b) simple with (c) answer interface.

An alternative simulation approach – the one we followed – is to perform a two phase study in which subjects play the role of developers in the first phase and observers in the second phase. In the first phase, developers' screens are recorded and the stuck points marked. In the second phase, observers are shown screen recordings of other subjects using simulations of the simple and buffered screen-help awareness. All developers are given the same problems in the first phase. As a result, when they play the role of observers in the second phase, they are

familiar with the problems and have some expertise to solve them. As mentioned below, our developers used different languages and created different solutions for the same problems. Moreover, several months elapsed between the two phases. Thus, arguably, this approach simulates the teaching assistant/student mentor/protégé scenario where the former is helping the latter with a similar problem they have solved before.

Yet another issue is what it means for observers to determine if they can help. A reliable solution is for them to actually develop a working solution, but without the code base it is not possible for them to do so, and more important, this approach is overkill as our help giving model assumes that after making this determination, they communicate or work with the developer to help them get unstuck. Therefore, we asked them to instead outline the solution, and used two coders and I to check if it is correct. A problem with using only this approach in a lab study is that sometimes the observers lacked certain knowledge (such as the file API of a programming language with which they were unfamiliar) to solve the problem. Rather than have them search for this information, which is unrealistic if they were real experts, we asked them to also describe the problem the developers were having. If the coders found this description to be correct, then, we assumed that with the right knowledge, they were in a position to help. We built a special tool for these coders' evaluation (Figure 5.2).

We used problems from the Mid-Atlantic ACM programming competition as in chapter 3. In the first phase, fourteen participants solved the three ACM programming problems while using the programming-activity difficulty-detection component. We recorded more than 40 hours of developer activities. I identified 16 stuck points in these recordings based on the inferences made by the help awareness mechanism and manual monitoring of developers' actions, which were confirmed by two coders.

The second phase involved ten of the fourteen initial subjects who were able to participate again. To simulate screen and help awareness, we created a stuck point video observation tool shown in Figure 5.1. This tool only shows a five minute video segment of each developer's stuck point. We chose five minutes segments after doing some pilot studies because participants were able to determine the fix to a problem, on average, within five minutes.

Each participant viewed the stuck segments under one of two conditions: either without rewind and pause (simple) or with rewind and pause (buffered). Participants did not view their own stuck points. For each participant, we assigned conditions to tasks randomly. We trained participants using one of the 16 stuck points, leaving 15 total stuck points. Two observers did not view 11 stuck points because of time constraints.

Participants were given an unlimited amount of time to describe a problem and determine how to fix it. The help determination time was calculated based on the time users pressed a button to view the video minus the time users pressed a button to submit the fix and explanation (Figure 5.1).

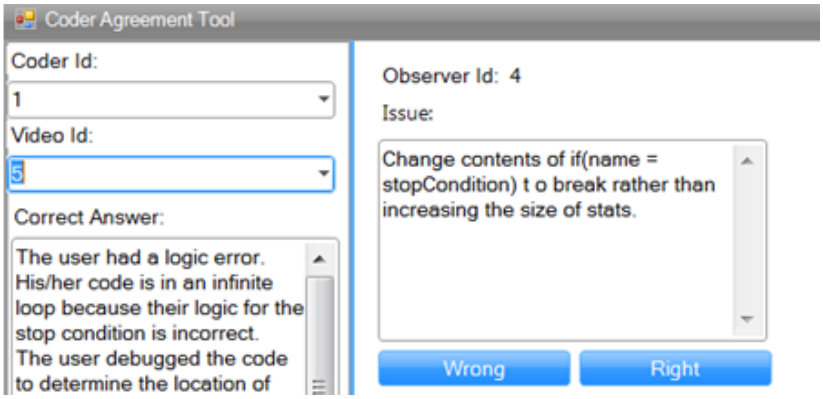


Figure 5.2: Coder Agreement Tool.

5.2.3 Metrics and Study Results

We computed several metrics to evaluate the effectiveness of the two mechanisms. We differentiated between temporary and permanent stuck points, based on whether the developers

were able to recover or not from the stuck point. We determined the success rate of the observers for all temporary and permanent stuck points. From Coder 1's viewpoint, observers were correct 56% (127) of the time on (total) stuck points, while from Coder 2's viewpoint; observers were correct 51% (127) of the time. The coders agreed 95% of the time. We computed Cohen's Kappa coefficient to take into account the agreement occurring chance (k=0.84). According to previous work ($k > .60$) is reliable and ($k > .75$) is excellent [38].

To better understand the impact of these results, we compared the amount of time developers spent overcoming difficulties to the average amount of time observers spent trying to solve developers' difficulties. On average, developers spent 16 minutes and 30 seconds trying to overcome their problems. Experts spent who solved developers difficulties spent 8 minutes and 46 seconds solving developers' problems. To determine whether there is a significance difference between the amount of time both developers and observers spent, we used an independent two means t-test. There was a significant difference between the amount of time both developers and observers spent ($t = 3.104$, $p<0.05$).

On average, observers spent 20% more time on each stuck point in the buffered condition. One of the reasons participants spent more time in the buffered condition is because they wanted to make sure they were giving the right advice. In fact, in the debriefing, a participant commented, "*I used rewind to rewatch some portion* [of the video] *and make sure I was correct*."

There was no significant difference in the success rate between the buffered and simple conditions. However, subjects overwhelmingly preferred using rewind and pause (Table 5.1). One participant made an unsolicited comment, after the study, stating that, "*when I couldn't pause or rewind it was frustrating because sometimes I couldn't remember exactly what*

happened." Another participant commented that, "*by rewinding I could see what they did to get them into trouble in the first place*.

Table 5.1: Survey Questions and Results (Scale: 1 = Strongly Disagree to 7 = Strongly Agree).

	Survey Question	**Mean**	**Median**	**STD. DEV.**
Q1	I preferred to use rewind/pause than not using it (just watching the video).	5.63	5.5	.74
Q2	I was confident in my answers when using rewind/pause.	6.38	7	1.06

We logged participants' actions with the stuck point video observation tool and found that on average, a participant used rewind 10 times and pause 14 times. Thus, from these results, we can conclude that a) combining screen and help awareness is a promising direction, motivating implementation of the two simulated mechanisms and field studies of them on bigger problems and b) they provide evidence to support sub-thesis IV, which we restate here.

Context Awareness Sub-Theses (Sub-thesis IV): Replaying the programming actions of developers who are stuck takes potential helpers significantly longer to decide if help can be

offered, but potential helpers prefer replaying programming actions to not having the ability to replay them.

5.3 Classroom Field Study

So far, all studies of help promotion have been in the lab. In this section, we explore help promotion in a field study. This exploration raises several intriguing design, implementation, privacy, usability, and usefulness questions: *(a) Usability:* Is it possible to build a usable environment that enables instructors to push help to students in a programming course? *(b) Privacy:* Would students have privacy concerns regarding such a tool? (c) *Helpability:* Would students who used the tool feel they were effectively helped? (d) *Learnability:* Students may learn more if they fix problems without assistance. What, if any, is the relationship between the amount of help students get and their final grade? (e) *Context:* What kind of context can be automatically given to potential helper to enable them to determine if they can or should help?

These questions are related. For example, it does not matter how much a programmer can be assisted if the programmer has privacy concerns preventing use of the tool, or the help interferes with learning. In the rest of the chapter, we answer these questions.

5.3.1 Field Studies in Education

As part of the UNC computer science department's Ph.D. requirement, I taught a course on object-oriented programming, which was taken by 35 students. We made use of this opportunity to do a two-part field study on education application of difficulty-detection. The main component of the study involved actual usage of our difficulty detection and context awareness tool. The other part was a "pre-tool" study to motivate the educational use.

5.3.2 Help vs. Grade

In educational settings, students may learn more if they fix problems without assistance. *What, if any, is the relationship between the amount of help students get and the amount they learn?*

To answer this question, we determined the number of times the students in the course were given help. Students were observed during office hours and email for six weeks, and special help sessions for two weeks. Help sessions were different from office hours because students could receive help in a small group (3 people) as opposed to a potentially large group in office hours. These help sessions were offered because a) some students did not feel comfortable asking questions during office hours or class, and b) a few of these students sent the instructor emails asking for one-on-one help. More than half of class (21 out of 35) met the instructor during office hours and asked questions through email (21 out of 35) while a little less than half (17 out of 35) of the class attended help sessions.

During office hours and help sessions, with students' permission, we recorded audio with a voice recorder, and later, immediately after office hours and help sessions, transcribed the audio. These recordings together with email logs enabled us to get the help data we needed. Through email, help sessions, and office hours, 27 students received help 190 times.

Help was given only after students attempted to solve problems and failed. To determine the correlation between the amount of help students received and the degree to which they learned the subject, we determined the relationship between the sum of the number of times students received help (in email, office hours, and help sessions) and their letter grade converted to a numerical value in the range 1..10 with a uniform step size of 1: A = 11, A- = 10, B+ = 9,…, D = 2, and F = 1.

Figure 5.2 shows that, for the vast majority of students, as the amount of help received increases, the final grade of the student tends to increase. We say "for the vast majority" because this pattern does not hold true for 29% of the students (10 out of 35). However, the remaining students performed just as well or better than some who received less help, which is consistent with the intuition that the stellar students would require little help. As the figure shows, several students who received an "A" also received help.

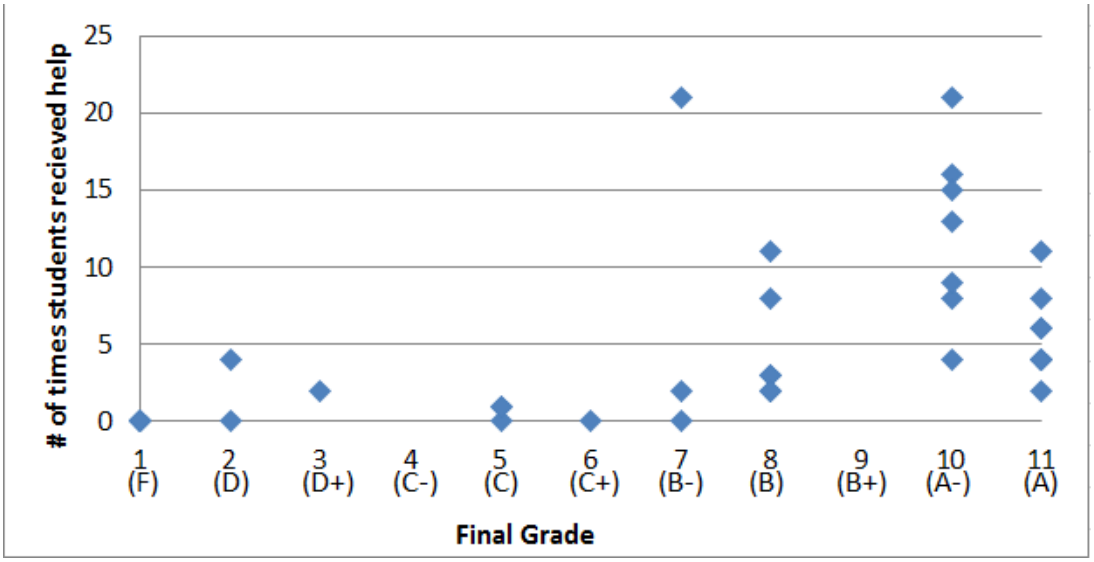


Figure 5.3: Grades vs. amount of help received.

As is common in such data, the cause and effect are not separated, and two possible conclusions are: (1) students with higher grades ask for and receive more help, or (2) students who ask for and receive more help get higher grades. Either way, the data show the importance of providing help, even to stellar students. In section 3, we showed that help giving can lead to time saving. Here, we show that for the vast majority of students, it does not have an adverse effect on learning, and may even have a positive impact if help giving was the cause and the grade was the effect in Figure 5.3.

5.3.3 Screen vs. Model Sharing

The next question: *is it possible to build a usable and useful collaborative tool in an educational setting*?

As mentioned earlier, such a tool must enable potential helpers to share the developers' programming context to determine if and how they should offer help. Based on our study of integrated screen awareness, we combined our difficulty-detection mechanism with a custom screen-sharing component.

We expected that most students (in the course taught by me) would install the screen-sharing help tool for two reasons. First, students would receive extra help. Second, participants who used the screen sharing feature in Community Bar were comfortable with enabling co-workers to view their complete screen [56].

Surprisingly, none of the 35 students was willing to use a tool that shared their screens. When the instructor asked for the reason, most of them stated that they perform activities other than programming, and would not want that information to be shared. For example, one student remarked: *"I don't want you looking at my Facebook page."* Even after the instructor mentioned that the tool would only record the Eclipse window, students were hesitant and would not install it.

This experience forced us to take a step back and consider the ways in which a programming context can be shared between two users. The context can be shared at multiple levels of abstraction such as the frame-buffer, windows, toolkit widget, and model; and the level of the ideal shared abstraction goes down with the coupling or divergence between the actions of the collaborators [6, 9]. Our screen-sharing help tool essentially provided window sharing. Based on the privacy concerns of the students, we decided to replace window sharing with Eclipse model sharing. We refer to this version of the tool as the Eclipse Helper.

Figures 5.4 and 5.5 show the student and instructor view, respectively, of Eclipse Helper. The student view contains buttons that enable students to manually indicate their status. The instructor view displays a) a notification when a student has difficulty (predicted automatically or indicated manually by the student), b) a status button that changes from green to red when a student has difficulty and back to green when the student is making progress, and c) a View Project button that opens up the instructors' programming environment and displays students' edits (model) in an editor window.

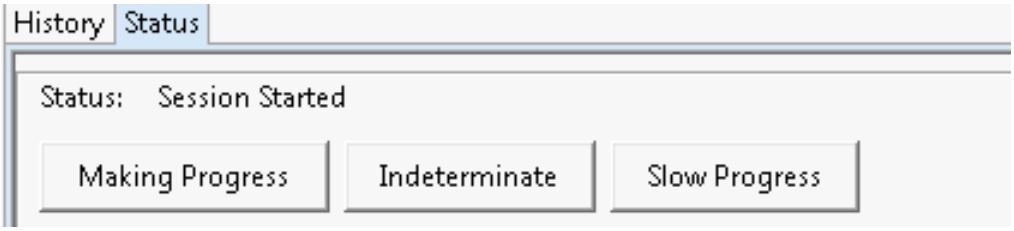


Figure 5.4: Student's view.

This time we had much more success with adoption - the majority of students (30 out of 35) installed Eclipse Helper. Thus, our work shows, surprisingly, that the level of sharing is a function of not only the coupling between the tasks of the collaborators but also the perceived privacy risks of the users.

5.3.4 Distributed Tool Use

Eclipse Helper gave students a fourth avenue for receiving help - they could now receive distributed assistance when their status showed that they were facing difficulty. Students used the tool for the last four of nine assignments.

For the first monitored assignment, students were required to manually indicate their status to ask for help, because the instructor was afraid the difficulty detection tool would give false positives. However, only one student pressed the 'slow progress' button to indicate a need

for help. This was surprising because some of the students did not perform well on the assignment.

When the instructor asked for the reason, the most common responses were: (a) they did not want to bother the instructor, (b) they were not sure if the instructor was available to help, and (c) when they were in difficulty they did not remember to press a button because they were trying to solve their problem.

One way to address this problem is for instructors to announce their availability and willingness to help, as we did with the first monitored assignment. However, this approach resulted in students asking for help with problems they were expected to solve alone. Therefore, we turned on automatic difficulty detection in the tool, and the instructor stopped announcing when he would be available for help. We wondered if students would be unnerved with help seeming to "come from nowhere" at unanticipated times, but the students who were helped indicated that this was not a problem, as they knew it was possible to receive help when they were having difficulty.

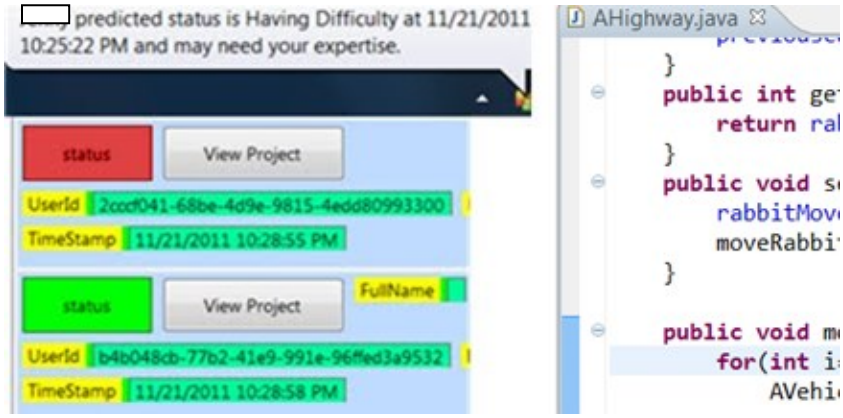


Figure 5.5: Instructor's view.

If the instructor decided that help should be offered, he entered into an email discussion with the student. Usually, the first message in the email exchange had the subject of "Help" and contained information specific to the students' difficulty. The type of help offered was in either

the form of references to background material or a description of how specific errors could be fixed.

To illustrate this process, in one case, after watching a student attempt to fix compiler errors for several minutes, the instructor sent an email to the student asking if she needed help. Several minutes later the student responded saying,

"Yes! I am having a few problems. I'm getting an error message and it won't recognize that I'm using methods from the AVehicle class."

The instructor sent an email asking her to:

"review how to write a method that takes parameters and how to call a method with parameters."

The student emailed back a few minutes later saying:

"thanks that fixed my problem. I struggled with that for over an hour."

Several students were appreciative of not only the specific help offered but also the instructor's willingness to help and a tool that enabled such help to be offered, as illustrated by the following response to a help inquiry:

"Cool to see that the helper is working! Thanks for asking me what is wrong!"

Even though students had a positive experience and appreciated the help, there was one case where a student, having trouble with conditionals, could not overcome the difficulty. To help the student, the instructor sent the following message:

"Are you having issues with your if statements in your while loop?"

The student responded to the email two hours later saying,

"I am having issues with my while loop and if statements. I'm not exactly sure what I am doing wrong. It only lets me enter commands once and after that it doesn't print anything."

After several email exchanges, the student did not solve the problem. One reason was that she had difficulty explaining her problem, and would have preferred if the instructor would have been able to see her screen so that she could point to it and explain the problem. Nonetheless, she still appreciated the attempt to help her.

Before the last assignment of the semester, we surveyed students to see if they would be more willing to install the screen sharing tool. Half (20 out of 35) of the students in the class were now willing to install the tool. The students gave two reasons for changing their minds. First, they said that they trusted the instructor more now than at the beginning of the semester. Second, students who were helped with Eclipse Helper indicated that it should be much easier to point at something on their screen to explain their problem than trying to explain it through email. After the last assignment, the most difficult one, we surveyed students again to see if they would be more willing to install the screen sharing tool. Almost all (31 out of 35) of the students in the class were now willing, which provides some evidence for our intuition that window sharing is a useful abstraction in a help session.

The instructor was able to offer help 9 times to 8 different students over the last three assignments. There were some instances where Eclipse Helper did not predict that students had difficulty, but the next day students came to office hours for help. However, each time the tool predicted a student was in difficulty and the student was asked if they needed help, the student answered in the affirmative. This result is consistent with our previous lab studies of the tool, which also showed the lack of false positives but the presence of false negatives. Thus, this tool augments but does not replace existing avenues for help such as email and help sessions.

5.4 Summary

To summarize, we have described integrated workspace-difficulty awareness. It combines continuous knowledge of remote users' screens, with difficulty detection, which

enables potential helpers to provide help; the moment developers are having difficulty. Two variations of this component are possible based on whether potential helpers can replay developers' screen recordings. A lab study of this component shows that potential helpers prefer replaying the programming actions of developers who are stuck, but replaying these actions takes them longer to decide if they can offer help (*Context Awareness Sub-Theses (Sub-thesis IV).* In a field study of this component, students were successfully offered help in a CS1 class. In particular, this thesis provides evidence to support our *Field Study Sub-theses (Sub-theses VI):* It is possible to build a difficulty detection tool that is successfully used to offer help to students.

Chapter 6. Difficulty Level and Barrier Detection

In the previous chapter, we show that replaying the programming actions of developers who are stuck takes potential helpers significantly longer to decide if they can offer help, but potential helpers prefer replaying the programming actions to viewing the programming actions. Replaying developers' programming actions provides syntactic awareness to potential helpers, which means they are not provided with inferences about developers' context. Therefore, potential helpers must manually look at developers' screens to determine context, which causes them to spend a large amount of time trying to determine if they can offer help. This means that potential helpers may waste a significant amount of their time. In this chapter, we address this problem, by inferring not only developers' progress status, but also their programming context.

Two types of programming context that may be useful to potential helpers are developers' difficulty level and the barriers that cause them to have difficulty. Ko et al. categorized barriers student programmers faced based on explicit help requests and manually determined their difficulty level [37]. These barriers are:

- not being able to design algorithms
- unable to combine Application Programming Interfaces (APIs)
- not understanding compiler or runtime errors, unable to find documentation for APIs
- unable to find tools within the programming environment.

They showed that for most barrier kinds, about half of its instances were insurmountable, thereby suggesting two levels of difficulties (insurmountable and surmountable). As mentioned above, both barriers and difficulty levels can provide useful context to potential helpers. For example, instructors should be able to help students on all barriers, but they may not want to help them on certain ones that students are expected to overcome alone. On the other hand, in industry, where the goal is to maximize productivity, potential helpers can help developers on insurmountable problems.

The difficulty-level detection component tracks sequences of interactive commands executed by the programmers to determine whether their difficulties are surmountable or insurmountable. The barrier detection component tracks both the ratios of various interactive commands executed by the programmer and the rates at which these commands are executed to distinguish between difficulties involving incorrect output from those involving design problems.

The rest of this chapter is organized as follows. We describe and evaluate difficulty-level and barrier detection and end with a brief summary.

6.1 Barrier Detection

As mentioned above, Ko et al. [37] have identified several barriers. Our own data involving the students we monitored in help sessions, suggested a simpler classification scheme: algorithm design issues and difficulty with correcting incorrect output. However, by the time the difficulty studies were done in the course, issues with using this tool has been ironed out. Therefore, we decided to determine if it was possible to automatically distinguish between design and incorrect output barriers.

Our recordings of the help sessions provided us with the data required to make this distinction. As before, we used coders to derive this information. To enable this process, we

developed a variation of the video observation tool of Figure 3.13 (chapter 3), which shows all segments where participants asked for help, and enables observers to identify the barriers the participants face. The coders agreed on 44 out of 50 difficulty points (k=0.79). 66% of these were classified as design barriers and 34% as incorrect output.

Now that we had ground truth, we had to identify appropriate features to automatically detect the barriers. Based on our observations of the recordings around difficulty points, we found the following: When programmers had incorrect output, the frequency of debug commands increased and the frequency of edit commands decreased. When they had design problems, they spent a large amount of time outside of the programming environment.

The feature set of our difficulty detection tool had been deliberately chosen to ignore wall-clock time. The reason was that we wanted to prevent our mechanism from classifying idle phases as difficult ones. Based on the observations above, it seemed we now had to consider the passage of time. We envisioned a two-phase difficulty detection scheme in which, first our previous time-independent *detection features* are used to determine difficulties, and then a new set of *classification features* is used to identify the barrier.

We included all of the previous detection features in the second classification set. In addition, we added features measuring the rate of interaction with the programming environment. The result was the following set of features. An asterisk indicates the previous detection features.

Classification Features

(1) **Insertion ratio* = # of insertions / # of total events.

(2) **Deletion ratio* = # of deletions/ # of total events.

(3) **Navigation ratio* = # of navigations/ # of total events.

(4) **Debug ratio* = # of debugs / # of total events.

(5) **Focus ratio* = # of focus changes/ # of total events.

(6) *Mean time between events* = total time / # of total events.

(7) *Mean insertion time* = total insertion time/# of insertion events.

(8) *Mean deletion time* = total deletion time / # of deletion events.

(9) *Mean focus time* = total focus time/# of focus events.

(10) *Mean navigation time* = total navigation time/ # of navigation events.

(11) *Mean debug time* = total debug time / # of debug events.

All of these times were measured in milliseconds. As before, we divided a log into 50-command segments, and computed these features independently for each segment.

To determine how indicative the detection and classification features are of programmers' behavior we graphed the programming behavior of six programmers. In each graph, the x-axis is session time and y-axis is the percent or time (in milliseconds) for each feature. Figure 6.1 shows portions of the graphs created for participant 1 and 2, respectively, illustrating commonalities in the behavior of the programmers when they are having difficulty correcting incorrect output. In both cases, participants' debug percentages increased, and the edit (insertion and deletion) percentages decreased. Figure 6.2 shows commonalities in the behavior of participant 2 and 4 when they are having algorithm design issues. In both cases, the participants spent a large amount of time outside of the programming environment, which is indicated by the mean focus time. In particular, participant 3 (4) spent 120 (350) seconds outside of the programming environment. Thus, the four graphs validate our feature choice.

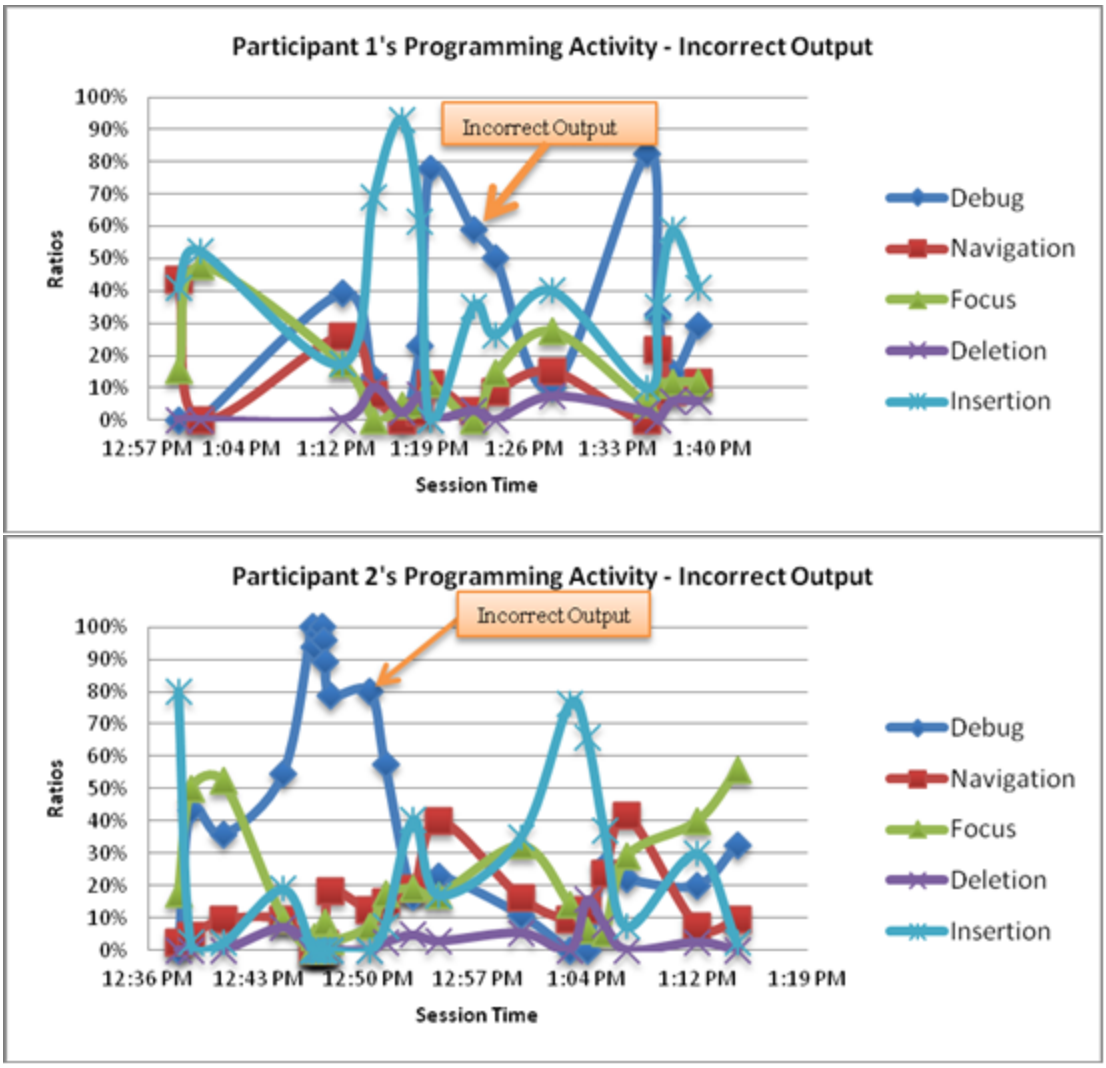


Figure 6.1: Programming activities when participants are correcting incorrect output.

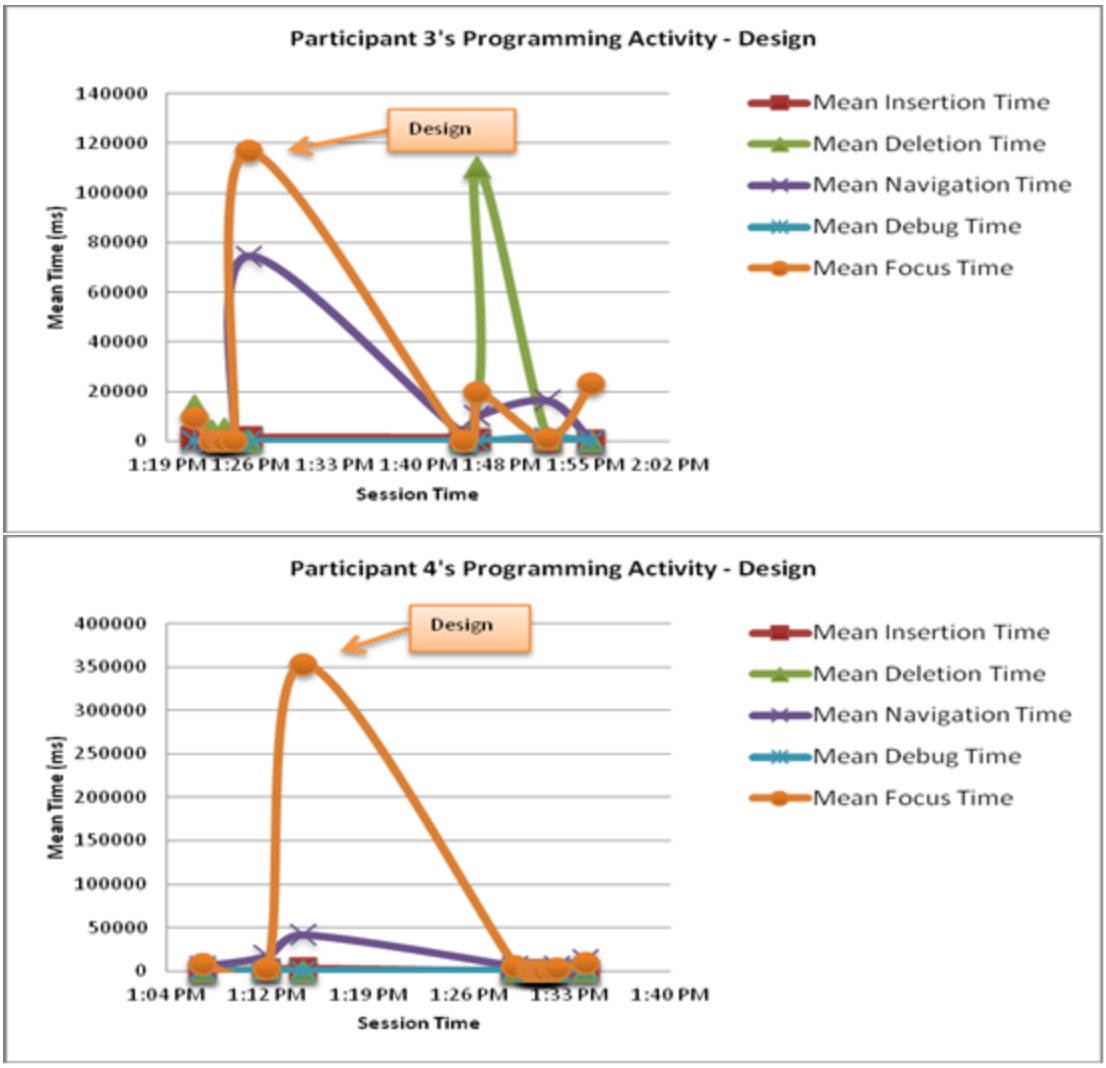


Figure 6.2: Programming activities when participants are having issues designing algorithms.

We fed to our decision tree algorithm the features of (a) each segment during which the programmer had explicitly indicated difficulty, which we refer to as an explicit segment, and (b) each segment that preceded an explicit segment and occurred within two minutes of the explicit segment, which we refer to as an implicit segment. The reason for (b) is that, on average, coders took two minutes to determine the barrier, and we assumed an algorithm would need the same amount of information.

We used a standard technique known as cross validation, which executes 10 trials of model construction, and splits the logged data so that 90% of the data are used for training and

10% for evaluation. We used a group model, as in chapter 3, in which the data of multiple programmers was aggregated during the training phase.

Results

The confusion matrix of Table 6.1 shows the results of the using the decision tree algorithm on the group model. The positive is Incorrect output and Design is the negative. The accuracy of this algorithm is 82%, the true positive rate is 73%, the true negative rate is 86%, the false negative rate is 27%, and the false positive rate is 14%. These results show that the algorithm correctly classified 25 of the 29 (86%) design barriers, and 11 of the 15 (73%) incorrect output barriers. To provide evidence to support part (a) of sub-thesis V, we compare the results of the decision tree algorithm to the baselines described in chapter 3. We use the same approach described in chapter 3 to compute each baseline.

Table 6.1: Barrier Confusion Matrix for Help Sessions.

	Predicted Incorrect Output	Predicted Design
Actual Incorrect Output	11 (True positives)	4 (False negatives)
Actual Design	4 (False positives)	25 (True negatives)

Table 6.2 shows the results for the random baseline. The accuracy of this baseline is 53%, the true positive rate is 53%, the true negative rate is 52%, the false negative rate is 47%, and the false positive rate is 48%. This baseline correctly identifies 53% of incorrect output barriers and 52% of the design barriers. As in chapter 3, we use the binomial test to determine if there is a significant statistical difference between each baseline and the results of our decision

tree algorithm. The decision tree algorithm performs significantly better than the random baseline (TPR=73% vs. TPR=53%, $p < .05$) (TNR=86% vs. TNR=52%, $p < .001$).

Table 6.2: Confusion matrix for random baseline.

	Predicted Incorrect Output	Predicted Design
Actual Incorrect Output	8 (True positives)	7 (False negatives)
Actual Design	14 (False positives)	15 (True negatives)

Table 6.3 shows the results for the modal baseline. The accuracy of this baseline is 66%, the true positive rate is 0%, the true negative rate is 100%, the false negative rate is 100%, and the false positive rate is 0%. This baseline correctly identifies all of the design barriers, but never identifies the incorrect output barriers. As mentioned in chapter 3, we do not use a significance test to compare the modal baseline to our approach. The true positive rate (73%) is better than the true positive rate (0%) of the modal baseline.

Table 6.3: Confusion matrix for modal baseline.

	Predicted Incorrect Output	Predicted Design
Actual Incorrect Output	0 (True positives)	15 (False negatives)
Actual Design	0 (False positives)	29 (True negatives)

Table 6.4 shows the results for the data distribution baseline. The accuracy of this baseline is 55%, the true positive rate is 33%, the true negative rate is 66%, the false negative rate is 67%, and the false positive rate is 34%. This baseline identifies 66% of the design barriers, but only 33% of the incorrect output barriers. The decision tree algorithm performs significantly better than the data distribution baseline (TPR=73% vs. TPR=33%, $p < .001$) (TNR=86% vs. TNR=66%, $p < .001$).

Table 6.4: Confusion matrix for data distribution baseline.

	Predicted Incorrect Output	Predicted Design
Actual Incorrect Output	5 (True positives)	10 (False negatives)
Actual Design	10 (False positives)	19 (True negatives)

6.2 Difficulty Level Detection

Before we could detect whether students' difficulties were surmountable or insurmountable, we had to first define these terms. A surmountable difficulty is one in which the observers' perception is that developers are having difficulty, but developer are later able to overcome the programming barrier without help. They overcome the difficulty by increasing the available resources or reducing the required resources such as looking at documentation or code samples. An insurmountable difficulty is one in which the observers' perception is that developers are having difficulty, but developers are unable to overcome the difficulty in the time given to them.

To determine whether students' difficulties were surmountable or insurmountable, we analyzed the video screen recordings from the Toolkit study in Chapter 4. This analysis occurred in three steps. First, we determined participants' barriers using the method in section 6.1. Coders agreed on 87% (48/55) difficulty points (k=0.70). Second, we went to the moment in the video where participants indicated or confirmed they were having difficulty and scanned backward in the video to find the moment when participants started having difficulty. These moments were identified when participants had an explicit error such as a compiler error or exception, searched the web, executed their programs, or spent a large amount of time outside of their programming environment. For example, if developers had incorrect output, we scanned backward to where they first started executing their program to test the behavior they were modifying or implementing in their current task. Third, we scanned forward in the video to find whether participants overcame their difficulty or switched tasks. These moments were identified based on no longer seeing the programming barrier or seeing participants start to implement a different task. For example, if participants had difficulty with incorrect output, we scanned forward in the video until we saw correct output.

We found that some barriers had the same start and end time. Therefore, we grouped these barriers together and counted them as one barrier. If participants overcame these barriers, we labeled them as surmountable. If participants could not overcome the barriers, switched tasks, or the user study ended before participants overcame the barrier, we labeled these barriers as insurmountable. There were 21 surmountable difficulties and 10 insurmountable difficulties. A coder and I performed these analyses together on approximately 45 hours of videos.

Now that we had ground truth, we had to identify appropriate features to detect the difficulty levels. Based on our observations of the recordings around difficulty points, we found the following: Developers who had surmountable difficulties tended to perform a cycle of editing and debugging their code, whereas those with insurmountable difficulties tended to spend a large amount of time between actions and outside of the programming environment. Given these observations, we now had to consider the order/sequence of programming actions. As before with barrier detection, we envisioned a two-phase difficulty detection scheme in which, first, our previous time-independent *detection features* are used to determine difficulties, and then the order/sequence of programming actions is used to identify the difficulty level. The detection features are shown in Table 3.1. The programming actions we used can be found in Table 6.5. Programming actions marked with an asterisk are new programming actions, while the others are ones we have used before.

Table 6.5: Programming actions and their explanations (an asterisk denotes new programming actions).

Programming Action	**Explanation**	**Single Character Representation**
*Content Assist	User invokes intellisense.	A
Insertion	User inserts a line of text.	B
Deletion	User deletes a line of text.	C
*Exception	User has an exception.	D
*Wait	User pauses for 1 second between programming actions.	E
Run	User executes the program.	F
*Compilation/Save Code	User saves the program, which also compiles the program.	G
Navigation	User switches from one file to another.	H
Gain/Lose Focus	User switches from or to the Eclipse programming environment	I
Break Point	User creates a break point.	J
Debug	User invokes the debugger.	K
*Url Hit	User goes to a url in the browser.	L

To predict whether developers' difficulties are surmountable or insurmountable, we fed the sequences of programming actions into the k-nearest neighbor algorithm. The intuition behind using this algorithm is that it makes predictions using the labels of training data, the sequences of programming actions excluding the sequence we are trying to predict, which are closest or more similar to the testing data, the sequence we are trying to predict. More specifically, given an input sequence and k the algorithm a) determines the similarity of k sequences to the input sequence and b) classifies the input sequence as the most frequently occurring label in k sequences.

To measure similarity between two sequences of programming actions, the k-nearest neighbor algorithm needs a real number that reflects the degree to which the sequences are similar. Therefore, we could not use common distance metrics such as the Euclidean distance, because these metrics take numeric values. Given that our input is two sequences of programming actions, we need a metric that can measure the similarity of two string values.

Two such metrics are the Hamming distance and Levenshtein distance. Hamming distance only works with strings of the same length. The Levenshtein distance can determine the similarity of variable length strings. The Levenshtein distance is the minimum number of edits needed to change one string into another string. Before we could use this distance metric with the k-nearest neighbor algorithm, we need to determine a value for k.

Choosing the value of k is still an open research problem. However, the value of k is also important because this value affects the algorithm's performance. We experimented with different values of k. This experiment consisted of five steps. First, we converted each programming activity into a single character, shown in the single character column in Table 6.5. Second, we divided the log, a sequence of programming actions, into segments.

One can segment the log based on the number of events, as we did in chapter 3. More specifically, after experimenting with different segment sizes, we found a segment size of 50 to be the best. Instead of experimenting, a bottom up approach, we did some top down thinking about how to determine the segment size. The basic intuition, used by Piech et al. [47], is to segment the log such that programming actions in each segment represent a unit of work. In our case, a unit of work is a group of related programming actions. We found the longest common subsequence between surmountable and insurmountable sequences.

Our intuition behind using the longest common subsequence is that the neighbors found by the k nearest neighbor algorithm would be a subsequence of the longest common subsequence. We implemented this algorithm and found that the longest common surmountable/insurmountable subsequence was content assist, insertion of text, deletion of text, content assist, content assist, insertion of text, insertion of text. We segmented the log every seven programming actions, the length of the longest common sequence, and every save. There were 295 surmountable sequences and 121 insurmountable sequences split by save; and 24142 surmountable sequences and 8245 insurmountable sequences split by the length of the longest common subsequence. In total, there were 416 surmountable and insurmountable sequences split by save and 32387 surmountable and insurmountable sequences split by the length of the longest common subsequence.

Now that we have two ways to segment the programming activity logs, the third step is to choose an odd value for k to avoid ties when determining the most frequently occurring label. The initial value for k was 1. Fourth, we used 10-fold cross validation to compute the performance metrics. Finally, we repeated step two, but increase the value of k by 2, and step three until we find the optimal value of k.

Results

Figure 6.3 show graphs where the x-axis is the values of k and the y-axis is the performance metric when segments are split by saves. The value of each performance metric no longer increases or decreases after k equals 25. Therefore, we chose 25 as a value for k. The confusion matrix of Table 6.6 shows the results when k is equal to 25. The positive is surmountable and insurmountable is the negative. The accuracy of the k nearest algorithm is 80%, the true positive rate is 96%, the true negative rate is 63%, the false negative rate is 4%, and the false positive rate is 37%. These results show that the algorithm correctly identified 96% of the surmountable difficulties and 63% of the insurmountable difficulties.

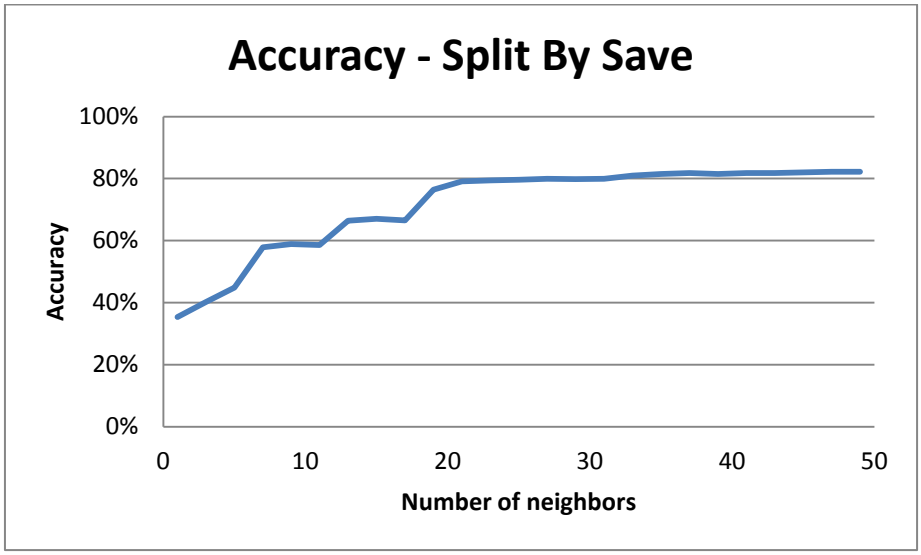


Figure 6.3a: The accuracy of the k nearest neighbor algorithm per number of neighbors when segments are split by save.

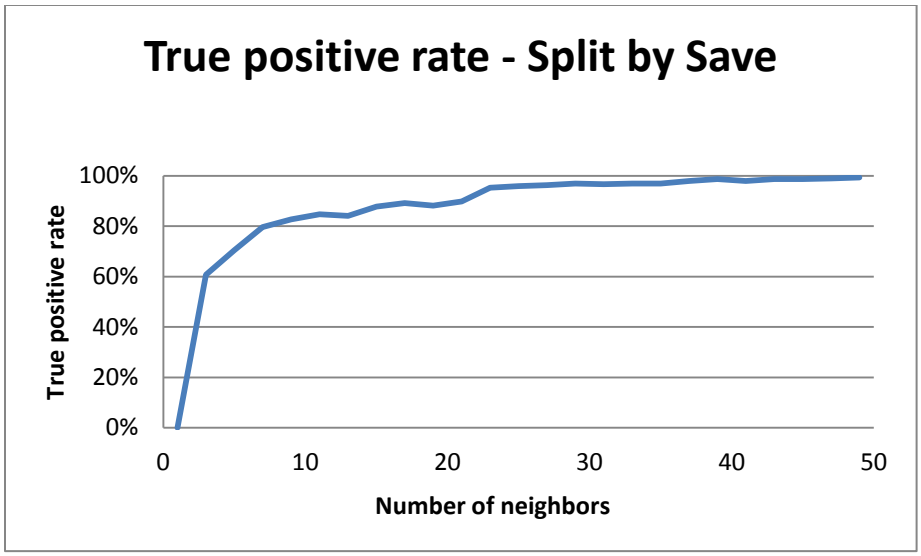


Figure 6.3b: The true positive rate of the k nearest neighbor algorithm per number of neighbors when segments are split by save.

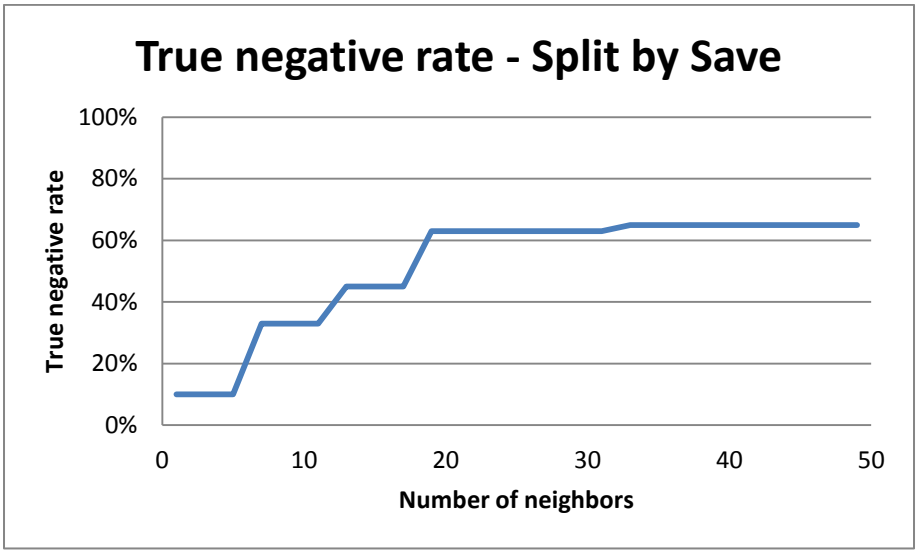


Figure 6.3c: The true negative rate of the k nearest neighbor algorithm per number of neighbors when segments are split by save.

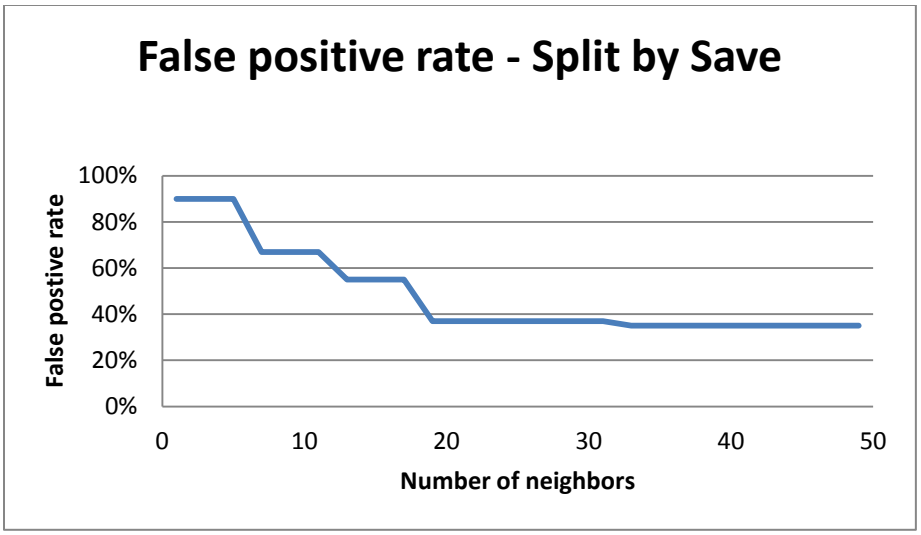


Figure 6.3d: The false postive rate of the k nearest neighbor algorithm per number of neighbors when segments are split by save.

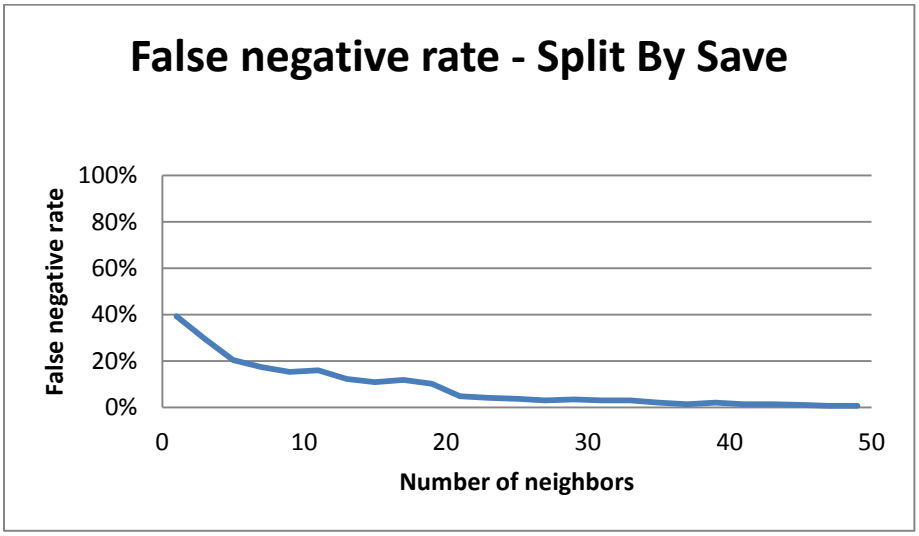


Figure 6.3e: The false negative of the k nearest neighbor algorithm per number of neighbors when segments are split by save.

Table 6.6: Confusion matrix for k-nearest neighbor algorithm when k is 25 and segments are split by save.

	Predicted Surmountable	Predicted Insurmountable
Actual Surmountable	281 (True positives)	14 (False negatives)
Actual Insurmountable	45 (False positives)	76 (True negatives)

Figure 6.4 shows graphs where the x-axis is the values of k and the y-axis is the performance metric when segments are split by the length longest common subsequence. As with segments that are split by save, the value of each performance metric no longer increases or decreases after k equals a certain value. In this case, the value is 29. The confusion matrix of Table 6.7 shows the results when k is equal to 29. The accuracy of the k nearest algorithm is 77%, the true positive rate is 93%, the true negative rate is 3%, the false negative rate is 7%, and the false positive rate is 97%. These results show that the split by longest common subsequence algorithm correctly identified 93% of the surmountable difficulties, but only 3% of the insurmountable difficulties.

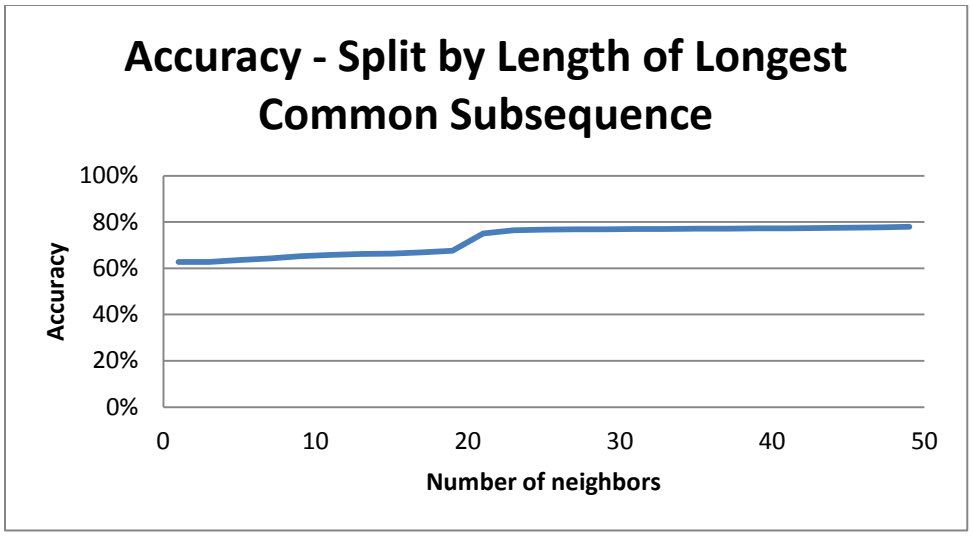


Figure 6.4a: The accuracy of the k nearest neighbor algorithm per number of neighbors when segments are split by the length of the longest common subsequence.

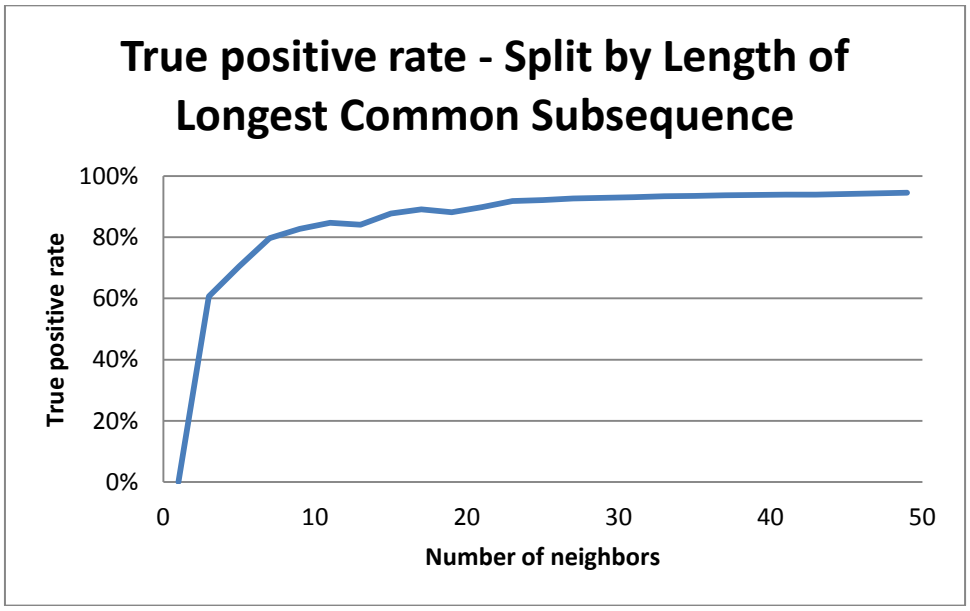


Figure 6.4b: The true positive rate of the k nearest neighbor algorithm per number of neighbors when segments are split by the length of the longest common subsequence.

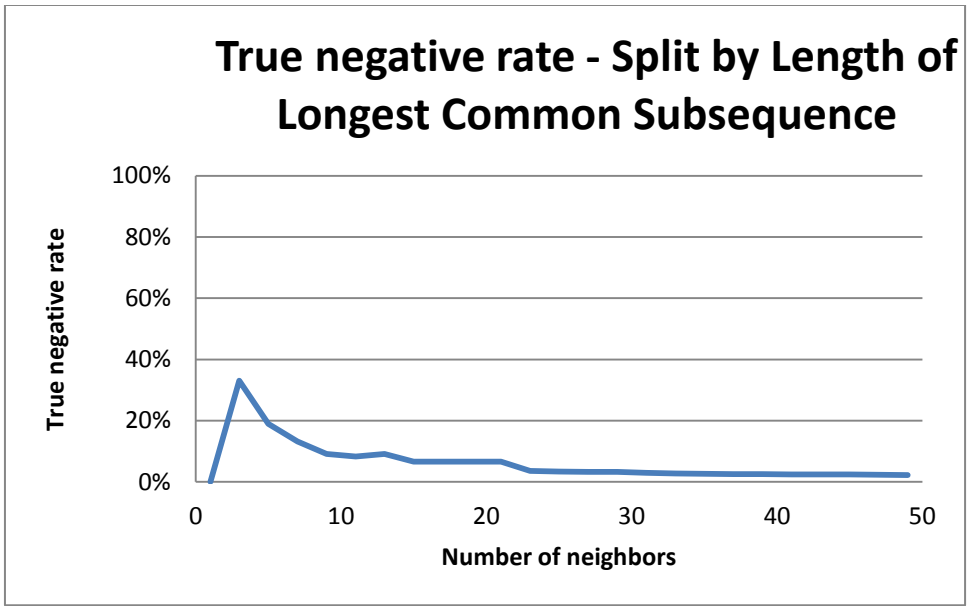


Figure 6.4c: The true negative rate of the k nearest neighbor algorithm per number of neighbors when segments are split by the length of the longest common subsequence.

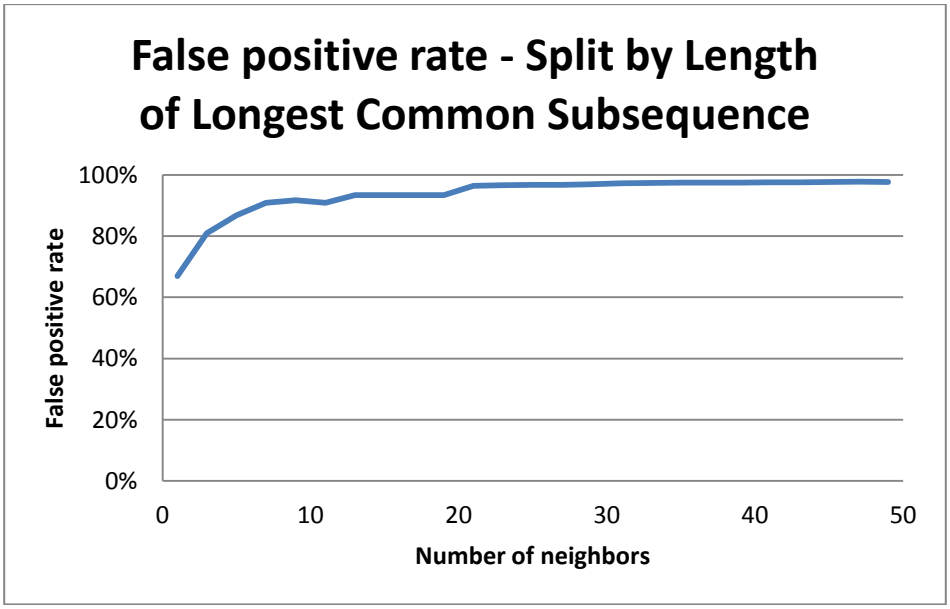


Figure 6.4d: The false positive rate of the k nearest neighbor algorithm per number of neighbors when segments are split by the length of the longest common subsequence.

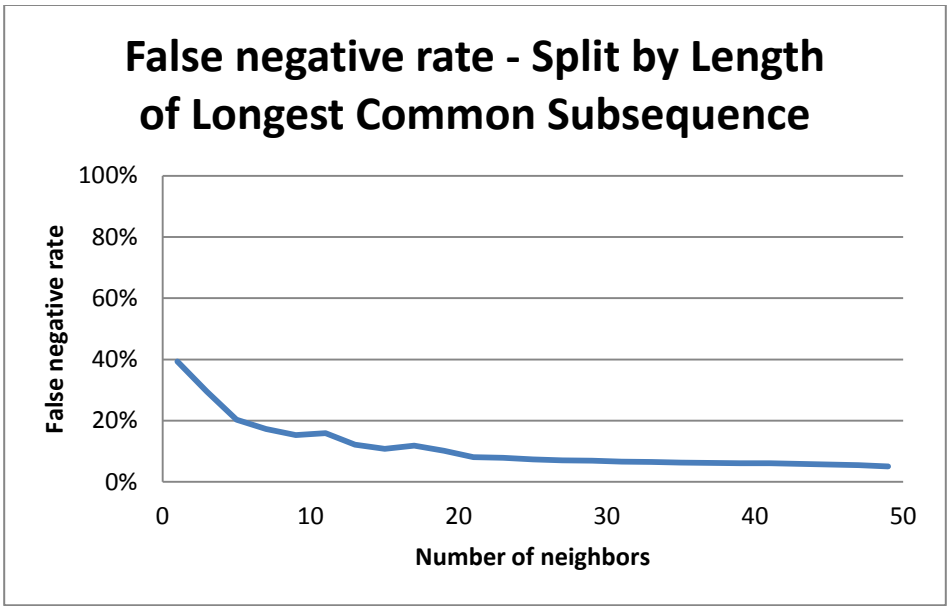


Figure 6.4e: The false negative rate of the k nearest neighbor algorithm per number of neighbors when segments are split by the length of the longest common subsequence.

Table 6.7: Confusion matrix for k-nearest neighbor algorithm when k is 29 and segments are split by the length of the longest common subsequence.

	Predicted Surmountable	Predicted Insurmountable
Actual Surmountable	22470 (True positives)	1672 (False negatives)
Actual Insurmountable	7997 (False positives)	248 (True negatives)

To provide evidence to support part (b) of sub-thesis V, we compare the results of the k nearest neighbor algorithm when k is 25 and segments are split by saves to the baselines described in chapter 3. We use the same approach described in chapter 3 to compute each baseline.

Table 6.8 shows the results for the random baseline. The accuracy of this baseline is 50%, the true positive rate is 50%, the true negative rate is 50%, the false negative rate is 50%,

and the false positive rate is 48%. This baseline correctly identifies 50% of the surmountable barriers and 52% of the insurmountable barriers. As in chapter 3, we use the binomial test to determine if there is a significant statistical difference between each baseline and the results of our decision tree algorithm. The k-nearest neighbor algorithm (with sequences split by save) performs significantly better than the random baseline (TPR=95% vs. TPR=50%, $p < .001$) (TNR=63% vs. TNR=50%, $p < .001$).

Table 6.8: Confusion matrix for random baseline (split by save).

	Predicted Surmountable	Predicted Insurmountable
Actual Surmountable	148 (True positives)	147 (False negatives)
Actual Insurmountable	60 (False positives)	61 (True negatives)

Table 6.9 shows the results for the modal baseline. The accuracy of this baseline is 71%, the true positive rate is 0%, the true negative rate is 100%, the false negative rate is 100%, and the false positive rate is 0%. This baseline correctly identifies all of the surmountable barriers, but never identifies the insurmountable barriers. As in chapter 3, we do not use a significance test to compare the modal baseline to our approach. The true positive rate (95%) of the k-nearest neighbor algorithm is better than true positive rate (0%) of the modal baseline.

Table 6.9: Confusion matrix for modal baseline (split by save).

	Predicted Surmountable	Predicted Insurmountable
Actual Surmountable	295 (True positives)	0 (False negatives)
Actual Insurmountable	121 (False positives)	0 (True negatives)

Table 6.10 shows the results for the data distribution baseline. The accuracy of this baseline is 41%, the true positive rate is 29%, the true negative rate is 71%, the false negative rate is 71%, and the false positive rate is 29%. This baseline identifies 71% of the insurmountable barriers, but only 29% of the surmountable barriers. The true positive rate of the k-nearest neighbor algorithm (with sequences split by save) is significantly better than the data distribution baseline (TPR=95% vs. TPR=29%, $p < .001$). There is not a significant difference between the true negative rate of the k-nearest neighbor algorithm (with sequences split by save) and the data distribution baseline (TNR=63% vs. TNR=71%, $p < .001$).

Table 6.10: Confusion matrix for data distribution baseline (split by save).

	Predicted Surmountable	Predicted Insurmountable
Actual Surmountable	86 (True positives)	209 (False negatives)
Actual Insurmountable	35 (False positives)	86 (True negatives)

So far, we have discussed how teammates can become aware that developers' need help of a certain level and how they can gain context to determine if they can help developers. If they decide that they can help, the last step in our model is for them to collaborate with developers to provide help. We do not provide special solutions to help developers collaborate because they can use existing mechanisms such as online-meeting software or meet with each other face-to-face.

6.3 Limitations

Effectiveness of difficulty level and barrier detection: We motivated difficulty level and barrier detection, but did not address the effectiveness.

Barrier Detection: As in the related work chapter, previous work has identified five barriers: not being able to design algorithms, unable to combine Application Programming Interfaces (APIs), not understanding compiler or runtime errors, unable to find documentation for APIs, and unable to find tools within the programming environment. In this chapter, we only infer two: incorrect output and design. It may be possible to detect a broader range of barriers such as API. We attempted to detect API difficulties, but could not distinguish them from design difficulties. The reason was that in both design and API difficulties, participants tended to spend a large amount of time outside of the programming environment. Additional features are needed to distinguish between API and design difficulties.

6.4 Summary

To summarize, we have described and evaluated difficulty-level and barrier detection. Our evaluation of these components show that they perform better than baseline measures. These results provide evidence to support sub-thesis V, which we restate below.

Difficulty Level and Barrier Detection Sub-Theses (Sub-thesis V): It is possible to develop an approach that, using developers' interactions with their programming environment, a)

automatically determines the barrier that is blocking programmers from making progress, b) automatically determines the level of difficulty programmers are having with their tasks, and c) performs better than baseline measures.

Chapter 7. Conclusions and Future Work

In addressing the goal of making help independent of distance, this thesis makes several innovations.

Automatic difficulty detection as a first class research area: It is the first work to show that automatic difficulty detection is possible and useful.

Motivation for automatic difficulty detection: It surveys previous work that suggests a) that the closer developers are to each other the easier it is for them to become aware that teammates need help and b) an increase in help provides an increase in productivity. This thesis also identifies several techniques to increase the amount of help in software development. Manual approaches enable developers to indicate their need for help. Automatic approaches infer when developers are having difficulty. To provide this automatic ability, the strengths and limitations of several mining techniques are explored.

Components of automatic difficulty detection: This thesis describes and evaluates six novel components: basic programming-activity difficulty-detection, multimodal difficulty-detection, integrated workspace-difficulty awareness, difficulty-level detection, barrier detection, and reusable difficulty-detection framework.

Basic programming activity difficulty detection: Programming-activity difficulty-detection builds on the work of previous difficulty detection approaches by logging developers' interactions with programming environments and inputting these actions into a difficulty detection module that predicts whether developers are having difficulty or making progress. It is

based on the insight that when developers are having difficulty their edit ratio decreases while other ratios such as the debug and navigation ratios increase. This thesis provides evidence to support the *Programming Activity Difficulty Detection Sub-Theses (Sub-thesis I)* that it is possible to develop an approach that a) uses developers' interactions with their programming environment to determine whether developers are having difficulty with their task and b) performs better than baseline measures. A limitation of this component is a high false negative rate.

Multimodal difficulty detection: Multimodal difficulty-detection addresses this limitation. It determines whether developers are having difficulty by using their body posture and interactions with their programming environment. It is based on the insight that when developers are having difficulty, both command ratios and postures can change. This thesis provides evidence to support our *Multimodal Difficulty Detection Sub-Theses (Sub-thesis III):* It is possible to develop an approach that a) combines programming activity and body posture recognition to predict when developers are having difficulty with their tasks and b) has greater accuracy and a lower false negative rate (predicting stuck) than existing approaches that only use programming activities to determine when developers are having difficulty with their tasks.

Integrated workspace-difficulty awareness: Integrated workspace awareness shows that difficulty detection is useful. It combines previous workspace awareness techniques with difficulty detection. In particular, it combines screen awareness, continuous knowledge of remote users' screens, with difficulty detection, which enables potential helpers to provide help, the moment developers are having difficulty. Two variations of this component are possible based on whether potential helpers can replay developers' screen recordings. A lab study of this component shows that potential helpers prefer replaying the programming actions of developers

who are stuck, but replaying these actions takes them longer to decide if they can offer help (*Context Awareness Sub-Theses (Sub-thesis IV).* In a field study of this component, students were successfully offered help in a CS1 class. In particular, this thesis provides evidence to support our *Field Study Sub-theses (Sub-theses VI): It is possible to build a difficulty detection tool that is successfully used to offer help to students.* One limitation of this component is that potential helpers may spend a large amount of time trying to determine if they can offer help.

Difficulty-level and barrier detection: To address the time-spent issue potential helpers face while trying to determine if they can offer help, this thesis develops difficulty-level and barrier detection. Both components automatically provide potential helpers with context. The former is based on the insight that when developers are having surmountable difficulties they tend to perform a cycle of editing and debugging their code; and when they are having insurmountable difficulties they tend to spend a large amount of time a) between actions and b) outside of the programming environment. Barrier detection predicts determines whether developers' are having difficulty with incorrect output or designing algorithms. This component is based the insight that when developers have incorrect output, their debug ratios increase; and when they have difficulty designing algorithms, they spend a large amount of time outside of the programming environment. This thesis provides evidence to support our *Difficulty Level and Barrier Detection Sub-Theses (Sub-thesis V): It is possible to develop an approach that, using developers' interactions with their programming environment a) automatically determines the barrier that is blocking programmers from making progress, b) automatically determines the level of difficulty programmers are having with their tasks, and c) performs better than baseline measures.*

To show how well difficulty-detection works in practice, we implemented a difficulty detection module into the Eclipse and Visual Studio programming environments. Both implementations increased programming time and effort, because as the difficulty detection algorithm changed, code had to be changed in both the Eclipse and Visual Studio implementations.

Reusable difficulty-detection framework: The reusable difficulty-detection framework addresses the cost of multiple implementations. It uses standard design patterns to enable programming-activity difficulty-detection to be used in two programming environments, Eclipse and Visual Studio. This thesis provides evidence to support our *Implementation Sub-Theses (Sub-thesis II):* It is possible to develop a common set of difficulty detection modules for different programming environments that have significantly fewer lines of code than difficulty detection modules written specifically for each programming environment. In particular, the Eclipse implementation had 11,000 lines of code and the Visual Studio implementation had 9,096. The number of lines of code to implement the framework is 4,643, which is significantly less than the number of lines of code to implement difficulty detection modules written specifically for Visual Studio and Eclipse.

This work suggests several new directions for future work.

Help and productivity study: Previous work has suggested that the productivity of developers increase when they help each other and as distance increases, help is offered less. However, none of these studies directly compares productivity with help and without help. This comparison could provide even more evidence to demonstrate the relationship between help and productivity. The data from our study in section 5.2.2 could be used to more directly compare productivity with help and without help.

Accuracy increase: It would be useful to increase the accuracy of the basic programming-activity difficulty detection component. One way would be to investigate additional features such as idle time, the amount of pressure on a mouse or keyboard, or features that can be computed from non-standard equipment such as the facial expressions. An alternative is to investigate different approaches that address the class imbalance problem. As in chapter 3, we use the SMOTE algorithm, which is a form of oversampling. There are additional approaches such as undersampling and cost-sensitive learning. It would also be useful to investigate different log segmentation methods such as segmenting the log based on time or using a sliding window approach. These approaches may lead to greater performance.

Other applications of difficulty detection: As in chapter 5, help promotion is only one potential application of difficulty detection. It would be useful to explore other applications such as (a) informing developers in difficulty about actions of others who earlier overcame similar difficulties so that they can take similar actions; (b) enabling those who carry out (educational or industrial) assignments to anticipate the kind of difficulties they would encounter and thus be better prepared for the assignment; and (c) providing those (e.g. supervisors, mentors, instructors) who assign tasks an understanding of the inherent difficulty level of the task, which can lead to redefinition or better explanation of the assignment. It may be possible to address application (a) by using collaborative filtering techniques where, the actions of multiple users are used to infer information about a user.

Manual help: In addition to investigating other applications of difficulty detection, it would be useful to explore whether barrier detection can be applied to situations in which help is requested explicitly. In particular, it may be possible and useful to automatically associate a question in a question and answer site, such as Piazza or Stack Overflow, with the detected

barrier and browsable screen recordings of the problem. This approach could enable potential helpers to better determine whether they can or should offer help.

Effectiveness of difficulty level and barrier detection: Additionally, we did not address the effectiveness of automatic barrier and difficulty level detection. It would be useful to perform both lab and field studies to compare a) the amount of time it takes potential helpers to determine whether they can help and b) the quality of their judgments with and without automatic barrier and difficulty detection.

Barrier detection use: This thesis shows that it is possible to detect incorrect output and design barriers. It would be useful to detect a broader range of barriers – in particular API barriers. It would also be useful to explore whether barrier kinds can be used to triage questions in a large class and assign them to appropriate instructors. More specifically, it would enable instructors to answer incorrect output questions first, which may require quick fixes. Similarly, assign design problems to a professor, who may be less likely to blurt out the complete answer, and an incorrect output problem to a TA, who may be quicker with debugging tools.

Help promotion environment: Also, it would be interesting to extend our help promotion environment described in chapter 5. It may be possible to enable smooth transitions between screen and model sharing, perhaps using the ideas in flexible coupling [14,22].

Maintenance tasks: In all of our studies, developers implemented programs from scratch. Maintenance tasks can be expected to have more navigation for the same difficulty degree. It would be useful to explore if it is possible to develop a single mechanism for detecting difficulty in both development of new software and maintenance of existing software.

Privacy: As in chapter 3, we did not formally evaluate privacy controls, but we have gotten some initial feedback from those who have used them. Users would indeed like to customize not

only what status is reported, but also when it is reported, and to whom it is reported. Thus, this scheme must be extended to control the nature and timing of reported status for different classes of observers such as (a) human observers and tools, (b) a team member sitting on the next seat, radically co-located, and distributed, (c) a close friend, mentor, and boss, and (d) team members who have and do not have the expertise to help solve a problem.

Applicability of results: Finally, another important future work direction is to determine whether our results apply to (a) a broader range of software engineering courses, (b) industrial training and mentoring, and (c) teams of peer programmers in educational and industrial settings.

This thesis provides a basis and motivation for carrying out these future research directions.

APPENDIX A: LAB STUDY TASKS (CHAPTER 3)

Problem H: Shrew-ology

Dr. Montgomery Moreau has been observing a population of Northern Madagascar Pie-bald Shrews in the wild for many years. He has made careful observations of all the shrews in the area, noting their distinctive physical characteristics and naming each one.

He has made a list of significant physical characteristics (e.g., brown fur, red eyes, white feet, prominent incisor teeth, etc.) and taken note of which if these appear to be dominant (if either parent has this characteristic, their children will have it) or recessive (the children have this characteristic only if both parents have it).

Unfortunately, his funding from the International Zoological Institute expired and he was forced to leave the area for several months until he could obtain a new grant. During that time a new generation was born and began to mature. Upon returning, Dr. Moreau hopes to resume his work, starting by determining the likely parentage of the each member of the new generation.

Input

The first line of input will containing a sequence of 1 to 80 consecutive ’D’ and ’R’ characters describing a list of physical characteristics, indicating whether each is dominant or recessive.

After this line will follow several lines, each describing a single adult shrew. Each shrew is described by a name of 1-32 non-blank characters terminated by a blank space, then a single M or F character indicating the gender of the animal, another blank space, then a list of consecutive 0 or 1 characters, describing the animal. A 1 indicates that the animal possesses that physical characteristic, a 0 indicates that it does not. The list of adults is terminated by a line containing only the string “***”.

This is followed by one or more lines describing juvenile animals. These contain a name and description, each formatted identically to those for the adults, separated by a blank space. The list of juveniles is terminated by a line containing only the string "***".

Output

For each juvenile animal, print a single line consisting of the animal's name, the string " by ", then a (possibly empty) list of all possible parents for that animal. A set of parents should be printed as the name of the mother, a hyphen, then the name of the father. If the animal has multiple pairs of possible parents, these pairs should be printed in alphabetic (lexicographic) order first by the mother's name, then by the father's name among pairs where the mother is the same. Each pair should be printed separated by the string " or ".

Example

Input:

```
RDDR
Speedy M 0101
Jumper F 0101
Slowpoke M 1101
Terror F 1100
Shadow F 1001
***
Frisky 0101
Sleepy 1101
***
```

Output:

```
Frisky by Jumper-Slowpoke or Jumper-Speedy or Shadow-Speedy
Sleepy by Shadow-Slowpoke
```

Problem H: Balanced Budget Initiative

After bouncing 10 checks last month, you feel compelled to do something about your financial management. Your bank has started providing you with your statement online, and you believe

that this is the opportunity to get your account in order by making sure you have the money to cover the checks you write.

Your bank provides you with a monthly statement that lists your starting balance, each transaction, and final balance. Your task is to compare the statement with the transactions from your checkbook register over the same time interval. You will identify transactions that appear in only the statement or register, as well as incorrect amounts recorded in the register (naturally the bank's statement is always correct) and math mistakes in your register.

Input

The bank statement appears first. It begins and ends with lines of the form:

balance <X>

with the first line indicating the starting balance and the second line indicating the final balance. In between the balances is the list of transactions, one per line, in the form:

{check|deposit} <N> <X>

Where N is the integer check or deposit number (the same check or deposit number will only appear once, although the same number can apply to both a check and deposit), and X is the amount of the transaction.

Following the final balance the register entries appear. The first line of the register is the starting balance

<X>

Following are pairs of lines, with the next transaction appearing followed by the balance you

calculated by hand after entering the transaction.

{check | deposit} <N> <X>

<X>

The pairs repeat until the end of the input file. For all input numbers and intermediates, |X| < 1000000. All dollar amounts are given to the

penny (0.01).

Output

For ease correcting your register, the output for each transaction occurs in the order it appears in the register. Each register entry receives exactly one line in the output.

If the register entry is entirely correct, meaning that it is found in the statement for the same amount, the math in the register is correct, and it is not a duplicate entry for a transaction previously found in the register, then output the line

{check|deposit} <N> is correct

However, if the transaction is not entirely correct, you will output a single line beginning with the transaction type and number, and one or more of the following mistakes, whitespace separated, in this order:

- **is not in statement** the transaction type and number do not occur in the statement
- **repeated transaction** the transaction has occurred previously in the register
- **incorrect amount** the register amount is different than the statement amount

- **math uses correct value** the math uses the value from the statement, although the actual transaction amount is recorded incorrectly in the register. This can only appear if **incorrect amount** is also displayed.

- **mathmistake**theregisterbalanceafterthetransactionmatchesneitherthestatementamount for the transaction, nor the register entry for the transaction (if different than the statement amount) Following the line for the final entry in the register, a listing of all transactions missing from

the register will be printed. These items may be printed in any order, one per line:

missed {check|deposit} <N>

Problem A: A Simple Question of Chemistry

Your chemistry lab instructor is a very enthusiastic graduate student who clearly has forgotten what their undergraduate Chemistry 101 lab experience was like. Your instructor has come up with the brilliant idea that you will monitor the temperature of your mixture every minute for the entire lab. You will then plot the rate of change for the entire duration of the lab.

Being a promising computer scientist, you know you can automate part of this procedure, so you are writing a program you can run on your laptop during chemistry labs. (Laptops are only occasionally dissolved by the chemicals used in such labs.) You will write a program that will let you enter in each temperature as you observe it. The program will then calculate the difference between this temperature and the previous one, and print out the difference. Then you can feed this input into a simple graphing program and finish your plot before you leave the chemistry lab.

Input

The input is a series of temperatures, one per line, ranging from -10 to 200. The temperatures may be specified up to two decimal places. After the final observation, the number 999 will indicate the end of the input data stream. All data sets will have at least two temperature observations.

Output

Your program should output a series of differences between each temperature and the previous temperature. There is one fewer difference observed than the number of temperature observations (output nothing for the first temperature). Differences are always output to two decimal points, with no leading zeroes (except for the ones place for a number less than 1, such

as 0.01) or spaces.

After the final output, print a line with “End of Output”

Example

Input:

10.0
12.05
30.25
20
999
Output:

2.05
18.20
-10.25
End of Output

APPENDIX B: FIELD STUDY TASK (CHAPTER 5)

Comp 110-003 - Assignment 9: MVC and Animation

In this assignment, you will strengthen your knowledge of the MVC programming paradigm and get practice with animations. As in the previous two assignments, you will continue to improve your structured object programming skills.

Start this assignment with all of your code from the previous assignment!

Part 1: Turning Rectangle, Oval, and Highway into observables

Extend your Rectangle and Oval classes from the previous assignments by making them an observable that follows the Java Beans observable pattern. There are three steps that you must do:

- 1) Define a PropertyChangeListener history
- 2) Implement the standard addPropertyChangeListener method in AHighway
- 3) After you change a value of one of the Rectangle/Oval properties, notify all observers

For help on completing these steps, read the "Variations in Observer/Observable Communication" section in the MVC chapter notes. You can find additional examples in the AnAnimatingShuttleLocation discussion in the Array chapter notes.

Recall that the object that is actually changed is the one that must notify the observers. For example, when a car location is modified, it should be Rectangle and Oval instances making up the car that notify observers.

You do not have to implement PropertyChangeEvent and PropertyChangeListener. Instead, you can need to import them from the java.beans package. To do so, include the following two lines in every class that uses PropertyChangeEvent and PropertyChangeListener instances:

import java.beans.PropertyChangeListener; import java.beans.PropertyChangeEvent;

Part 2: Moving the rabbit in all directions

Extend your AHighway by adding a moveRabbitLeft, moveRabbitRight, moveRabbitUp, and moveRabbitDown operations, which move the rabbit left, right, up, and down, respectively, by RabbitMoveDistance (defined in previous assignment). This should not take long because you have already defined moveRabbit in the previous assignment, which moved the rabbit vertically by MoveRabbitDistance. As was the case with moveRabbit in the previous assignment, each one of the new move operations must also check if the rabbit is colliding with any of the cars and update the RabbitStatus property accordingly.

If in the previous assignment, you moved the cars or the rabbit whenever the corresponding move distance properties changed, you will have to correct your assignment. The rabbit and the cars need to move when you or a user (which can be another object) calls moveAllCar or moveRabbit(Left/Right/Up/Down).

Also, if in the previous assignment, you used the move distance property values as offsets, you will need to correct your assignment. These properties stored by how much the cars and the rabbit should move, not to what location they should move.

Part 3: AConsoleHighwayController

Implement a console-based controller for AHighway, called AConsoleHighwayController. It should implement the following interface:

public interface HighwayController { public void setModel(Highway model); public void processInput();

} The processInput command accepts user input from the console. Assume the user can enter the

following commands only (hence, no erroneous input checking is required):

. 1) ‘a’ to move the rabbit left

. 2) ‘d’ to move the rabbit right

. 3) ‘w’ to move the rabbit up

. 4) ‘s’ to move the rabbit down

. 5) ‘q’ to stop entering commands

Each command must be followed by Enter. This is a somewhat awkward input interface, but it makes the code you have to write a little less complex. So to move the rabbit up twice, the user would type first w, then Enter, then w again, and then Enter again.

Part 4: Animating AHighway

Define two editable integer properties, AnimationDistance and AnimationPause, in AHighway. Then implement a startGame operation in AHighway that has an infinite loop that does the following:

. 1) Move the cars by AnimationStepSize to the left

. 2) Sleep for AnimationPauseTime time

NOTE: After a while, all of the cars on the highway will move off the screen. This behavior is acceptable. Part 5: AHighwayDriver

Implement your main method in AHigwayDriver. The method is responsible for:

. 1) instantiating AHighway (the model)

. 2) instantiating AConsoleBasedHighwayController (the controller)

. 3) Connecting the model and the controller

. 4) Calling bus.uigen.ObjectEdtitor.edit to display the highway

AHighwayDriver also sets up the highway for the game as follows:

. 1) Sets MoveCarDistance to 10

. 2) Sets PreviousCarDistance to 150

. 3) Sets MoveRabbitDistance to 35

. 4) Sets AnimationStepSize to 10

. 5) Sets AnimationPauseTime to 30

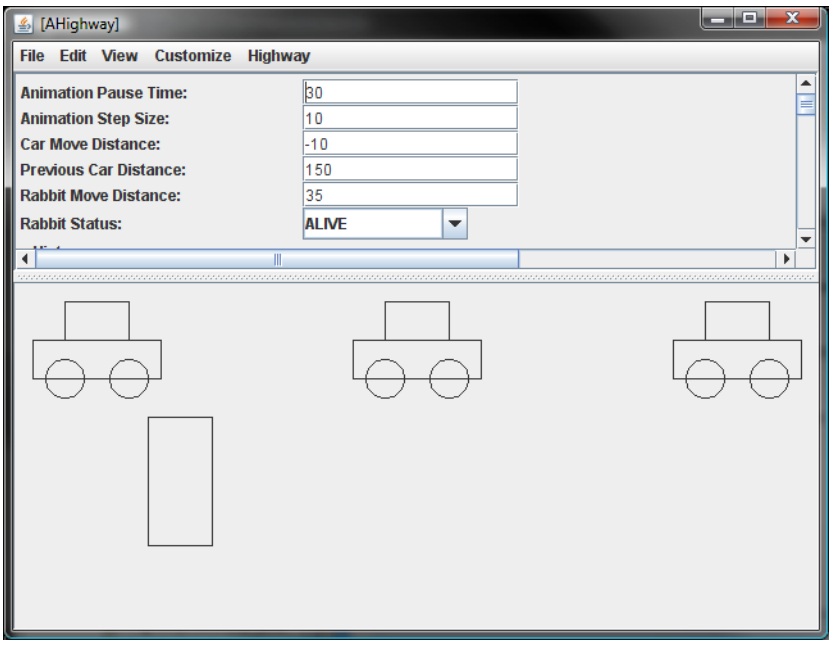
. 6) Adds 30 cars to the highway, each of which is of width 100 and height 30

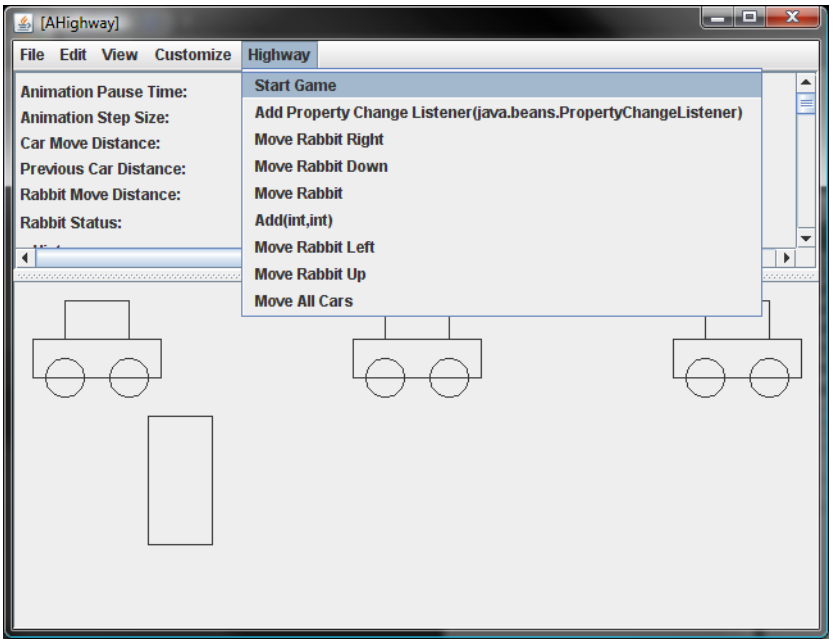
Playing the Game

1) The user first runs AHighwayDriver. The following screen should appear.

2) Once the highway is displayed in an ObjectEditor window, the user invokes the StartGame operation on the Highway as shown below.

3) Then the user enters commands to move the rabbit.





Finishing the Game

There are two ways to finish the game.

1) Rabbit crosses the road without getting hit: In this case, you need to display a "Congratulations! You live to play another day" message in a JOptionPane.

2) Rabbit gets hit: In this case, you need to display a "Splat! Please try again" message in a JOptionPane.

When a finishing condition is reached, you do not have to stop the cars from moving. The reason for the latter is that in the real-world, cars on a highway will not stop because of a rabbit. Also, assume that the user will not enter any more commands once the finish message is displayed. In other words, you do not have to write any code to explicitly prevent the user from entering anything but a non 'q' command.

Bonus

For the past few months, the bunny world has been hopping. A rumor has been spreading that the legend of Bravehops will be fulfilled on Dec 5, 2007. For those of you who have not heard of Bravehops, it is a legend of a rabbit who escapes the Land Below the Highway and reaches the world beyond. Rabbits from the farthest reaches of the Land Below the Highway have gathered

in the grassy area just inside the border of their world in anticipation of the fulfillment of the legend. If Bravehops manages to reach the other side, they will all stand up and clap their ears. But if Bravehops does not reach the other side, they will quietly leave without ever being seen.

Your bonus, should you choose to complete it, is to draw the spectators once Bravehops crosses the highway. The number of spectators you need to draw the n^{th} number in the Fibonacci sequence. The value of n is stored in an editable integer property of AHighway, called FibonacciNumber. You must use recursion to calculate the number of spectators from the FibonacciNumber value. The n^{th} number in the Fibonacci sequence is calculated as follows:

$F(1) = 1$ $F(2) = 1$ $F(n) = F(n-1) + F(n-2)$

if $n = 1$ if $n = 2$ if $n > 2$

While the exact locations of and the separation between the spectators is not important, an outside observer should be able to count the exact number of spectators. In other words, do not draw them all in the same location because it will appear to an outside observer that there is only one spectator regardless of how many you actually draw

APPENDIX C: PARTICIPANT DIFFICULTIES

1.1 Lab Study Difficulties (Chapter 3)

Participant Id	Problem	Explanation of Error
4	B	Logic error: in the for loop the user had greater than or equal to, the user changes the condition to > than, and later changes it to less (<) than. The user set breakpoints to find the logic error. After a cycle of editing and debugging the user determined the cause of the logic error and fix it.
5	B	User had a logic error. The user set a breakpoint where they believed the issue existed. The user debugged the code several times before determining there was an error with a for statement. . for(int pos = 0; pos < MAX_SIZE; i++) The i++ in the for statement should be pos++.
5	C	User had a logic error in their if statement. The user set a breakpoint and debugged the code to determine why their logic was incorrect. The user changed !true to true in an if statement. If(!inRegister to if(inRegister
5	C	The user had a logic error in an if/else branch. The user ran the code and determined that the output was incorrect. After debugging the code, the user determined that an else statement should be an else if statement. The user also determined that they forgot a variable assignment.
6	A	User knew which libraries to use for reading in a file, but she did not understand how to use them. The user looked up on the web how to read input from a file. The user copied and pasted example code and modified the code. The user could not get the example code to read input.
6	B	User searched the web for a data structure that did not enable duplicate items. The user changed data structures multiple times, but could not find the data structure the user was looking for.
6	C	User got a NullPointer exception when reading input in from a file. The user looked at the BufferedReader class in the Java API. More specifically the user looked at the readLine method in BufferedReader. The user checked for null: inLine.equals(null) || inLine.equalsIgnore(null)
6	C	User could not get the correct output. The user debugged the code to determine where the logic error existed. The user believed he/she found the logic error and cut/pasted an if else statement and changed the condition in the if statement. The user tested their error and still got incorrect output. The user proceeded to continue debugging and editing the code where the user believed there was an error.
6	C	User got a NumberFormat exception, changed Integer.parseInt to Double.parseDouble. The user believed that this line was causing the exception. User ran the code again and still got the same exception. User changed another line of code, the index of a substring. This fixed the user's NumberFormat Exception, but the user ran into a NullPointer.
9	B	Used the List API incorrectly list[i] is not correct for adding something to a list, changed list[i] to list.add after looking at the code completion drop down list
10	A	User had a NumberFormatException. The user was trying to parse a string into an int. The string was a floating point number e.g. "4.01", hence the NumberFormatException. The user parsed the string to a float and got rid of the NumberFormatException.

10	B	The user had a logic error. The user was reading input from a file. The user wanted to put two different types of input into two separate arrays. The input was read correctly into one array, but the input was not read into the second array. This lead to a NullReference Exception
10	B	The user had a Null Reference Exception
11	A	User compiled the program and got several compiler errors. The user read the compiler errors and could not figure out the problem. The user was also using an unfamiliar API and assumed the errors were caused by the API call. Therefore, the user looked up the API using Google and looked at example code to ensure the API was being used correctly. The user made changes to the API call, but undone the changes because the user was sure the API was used correctly. The user commented out the API call to see if the program would compile with no errors. The user immediately noticed the error after the program compiled with no errors and he could focus on the code without the API call.
11	B	The user had a logic error. The user debugged the code to determine the location of the logic error. After the user found the logic error, the user edited the code, but still could not get the correct output. The user tested the edited code, but the logic was still incorrect. The user debugged the code to determine why the edits did not produce the correct output. The user added additional code and tested the additional code to ensure the output was correct.
14	C	User had several compiler errors after compiling the program. The user was trying to read input in from a file. The user looked at examples on the web of how to read input from a file and copied the examples verbatim. After looking on the web and comparing examples, the user was able to fix his syntax error
14	C	The user modified example code from the web that read input from a file. The user had several compiler errors after compiling the program. The user could not figure out what errors were introduced after the modifications to the example code. The user looked for new examples and compared the modified example code to the new example code to fix the compiler errors.
14	C	User had a syntax error, cannot covert parameter 1 from substring to substring[]. The user modified the line where the compiler told the user the error occurred. The user's modification introduced more compiler errors.

1.2 Field Study Difficulties (Chapter 5)

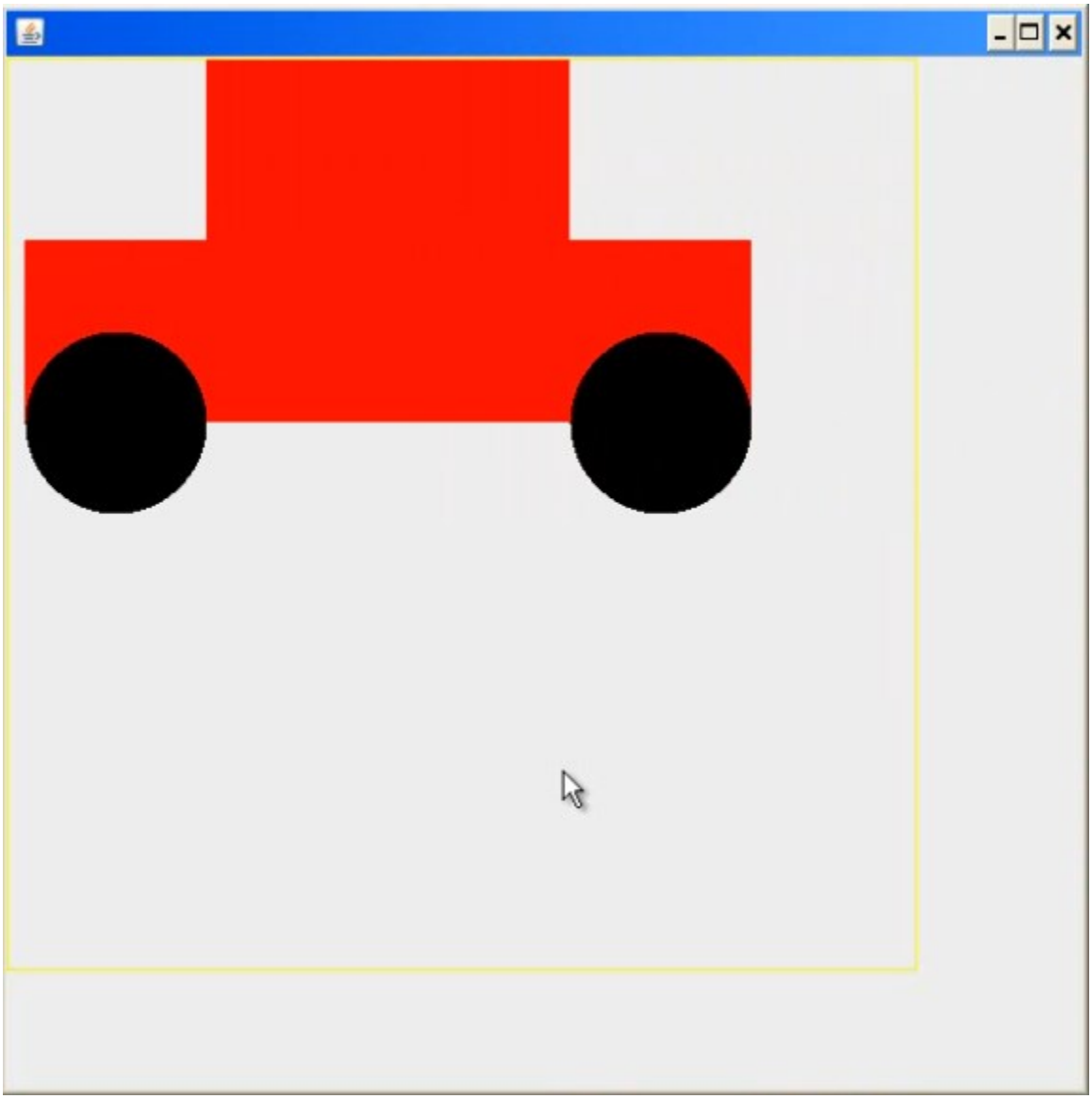
What is student trying to do?	What is causing the difficulty?
create a new type of Point given an x and y value	They don't understand how to create the new type
create a new type of Point given an x and y value	They don't understand how to create the new type
create a new type of Point given an x and y value	They don't understand how to create the new type
create a new type of Point given an x and y value	They don't understand how to create the new type
create a getter and setter for Rectangle (object) unsure what to return in her getter Unsure how to get code to do what she needs it to do	The student does not understand how to create an object type ex. Rectange myRectangle or what to return in a getter
create a new type of Point given an x and y value	They don't understand how to create the new type
create a new type of Point given an x and y value	They don't understand how to create the new type
create a new type of Rectangle	not sure of where to put the code to create the Rectangle
draw two Rectangles on the screen	the student had two incorrect assignment statements (had initialWidth = width and initialHeight =height, should have had weight = initialWidth, height=initialHeight the student did not get the correct output
pass variables into the AVehicle constructor to run the program (trying to test program)	the student passed in the incorrect number of parameters and the incorrect types for the parameters into the constructor, student has a syntax error but does not understand how to fix it
create a new Rectangle	using an interface instead of a class to create an object in AVechicle, student has a syntax error but does not understand how to fix it
create a constructor that takes two Point and two Oval parameters student did not understand the syntax error message	the student imported java.Oval instead of the Oval interface they created and Eclipse was suggesting to create a new constructor in the AVechicle class, this suggestion would not have helped the student student was not sure of how import java.Oval got there, see if i can figure this out from the logs
make the Rectangle cabin in the AVechicle class centered on the Rectangle body in the AVehicle constructor	the student's output is incorrect, the cabin is not centered over the body because the user is changing the X coordinate in the body constructor instead of the cabin's constructor
use a scale factor to increase the height and width of the Rectangle cabin in the setScaleFactor method	the student is getting a syntax error that says the width and height variables are not recognized in the setScaleFactor method, the student is trying to use the height and width parameters in the AVechicle constructor in the setScaleFactor method, the student should either create height and width variables in the setScaleFactor method or create instance variables for height and width and set their values

	equal to the height and width variables in the AVehicle constructor, then the student can use the instance variables height and width in the constructor
implement the Vehicle interface in the AVehicle class	the student is getting a syntax error that is telling him he has to implement every method in the interface, the student has written the method headers but did not include curly braces around each method
scale the car	look at the logs
scale the car	look at the logs
scale the car	look at the logs
scale the car	look at the logs
scale the car	look at the logs
draw the cabin and the body	the student is testing the program but the location of the body and cabin are not the location she manually entered in the ARecantgle constructors for cabin and body, the problem was that the ACartesianPoint class was not completely implemented, the student copied and pasted the incorrect ACartesianPoint class from the notes
create a static int property that is increased each time a new AVehicle object is created	the student did not understand what the static keyword does for a method or a variable
create a static int property that is increased each time a new AVehicle object is created	the student did not understand what the static keyword does for a method or a variable
return the ARectangle cabin object using a getter and create an instance of that object in the AVehicle constructor	the student created The ARectangle cabin variable in the constructor and is trying to reference that variable in the getter method getCabin(), the student should create an instance variable named cabin and create the instance in the constructor and return the instance in the getter
implement the Vehicle interface in the AVehicle class	had a syntax error because he created a method without curly braces {} had: public Point getLocation() return location; should have had: public Point getLocation() { return location; } all of his methods in the class were like this
draw an Oval shape it was only drawing points	i am not sure, i rewrote his Oval constructor to take two ints as x and y coordinates instead of a point
create a Rectangle object named cabin	syntax error - she was using width1 varible in the getCabin, getBody, getFrontTire, getBackTire methods, but the width1 variable is a paramter in the AVechicle Constructor, she also had width and height instance variables, she should have been using the width instance variable
create an Oval Constructor that takes a point, a width, and a height	did not understand the syntax error message the student imported java.Oval instead of the Oval interface they created and Eclipse was suggesting to create a new constructor in the AVechicle class, this suggestion would not have helped the student student was not sure of how import java.Oval got there, see if i can figure this out from the logs
get object editor to display the	had everything correct, but for the method getBody he had Body and

vehicle	for the methods getCabin he had Cabin, for getFrontTire he had FrontTire, for getBackTire he had BackTire, object editor enabled him to call the methods, but did not display the car, after creating getters, his code worked
trying to get object editor to display two rectangles	Had a void return type on constructors she also had syntax errors, look for those in the logs
scale the vehicle	look at the logs
scale the vehicle	look at the logs
scale the vehicle	look at the logs
scale the vehicle	look at the logs
test the program to see if the if statements work	the student is using == with strings instead of .equals() if(command == "move") //do stuff else if(command == "scale")
get the Vehicle to display with a different width and height when she enters in a new width and height from the console	passed in width and height into the AVehicle constructor, but did not use the width and height variables when creating the Rectangles that represent the body and cabin of the vehicle
Get the program to recognize when she enters in a command (move, scale), the program runs and when she enters "move", it skips her if statements	student has a semi-colon after each if statement if(command.equals("move"));
create a tire	syntax errors: private Oval createTires" saying that the constructor AnOval(int, int, int, int) student is calling the constructor and passing it 4 integers, but the AnOval constructor takes a Point and two ints
create a tire	syntax errors: private Oval createTires" saying that the constructor AnOval(int, int, int, int) student is calling the constructor and passing it 4 integers, but the AnOval constructor takes a Point and two ints
create the body and cabin	private Oval createBodyandCabin saying that the constructor Rectangle(int, int, int, int), but the Rectangle constructor takes a Point location, and two ints
create the body and cabin	private Oval createBodyandCabin saying that the constructor Rectangle(int, int, int, int), but the Rectangle constructor takes a Point location, and two ints
test the program to see if the if statements work	the student is using == with strings instead of .equals() if(command == "move") //do stuff else if(command == "scale")
enable a user to enter commands until the type quit	the student did not understand how while statements worked, the student had the while keyword where if statements should have been
display the vehicle and print information to the console	the student was calling ObjectEditor.edit(object), in the while loop, the student did not know how to create a Vehicle object and change the properties of the object
make the while loop stop when the user enters quit	the student created a method readAction which reads input from the console, the student called while(readAction()) and in the if statements he also had if(readAction(), the student was reading input in the while and if statements, the student should just read input once in the while statement
draw tires as ovals, the ovals were being drawn as points	the student had getInitWidth and getInitHeight in her oval class instead of getWidth and getHeight, Object Editor expects getWidth and getHeight
get the while loop to work	the student doesn't understand while loops and also had some syntax errors

move the vehicle offSetX and offSety	the student had two incorrect assignment statements (had y = offSetY and x =offSetX, should have had offSetX= x, offSetY=y the student did not get the correct output, when the student tried to set the offSetX and offSEtY values via ObjectEditor, the values always changed back to 0
take the values a user entered for offSetX and offSetY and enter those values into the offsetX and offSetYvalues for a car	the student is not sure how to set teh value, the student did not know that he could call car.setOffSetX and car.setOffSetY
the student is trying to print out information about the car	the student got a nullpointer exception, because the student was not setting the values of cabin, body, frontTire, or backTire, in the constructor, to solve the problem, i inserted ObjectEditor.edit and the name of the AVehicle variable, ObjectEditor calls the getters, getCabin, getBody, etc. and in those methods is where the students create new instances of cabin, body, etc. the student says that everything just comes out zero, the student also tried to debug, but said he doesn't fully understand the debugger

APPENDIX D: EXAMPLE CORRECT OUTPUT FOR TASKS (CHAPTER 4)



REFERENCES

1. Abel, M. Experiences in Exploratory Distributed Organization. Lawrence Erlbaum (1990), 489–510.

2. Anderson, J. and Reiser, B. The LISP tutor: it approaches the effectiveness of a human tutor. (1985), 159–175.

3. Begel, A. and Simon, B. Novice software developers, all over again. *Proceedings of the Fourth international Workshop on Computing Education Research*, ACM (2008), 3–14.
4. Begel, A. Help, I Need Somebody! *In the CSCW Workshop: Supporting the Social Side of Large-Scale Software Development*, (2006).

5. Berlin, L.M. and Jeffries, R. Consultants and apprentices: observations about learning and collaborative problem solving. (2008).

6. C. de Souza. *The Accord.net Framework*. 2012.

7. Chawla, N.V. et al. Smote: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research 16. Smote: Synthetic minority over-sampling technique*, (2002).

8. Cheng, L.-T., Hupfer, S., Ross, S., and Patterson, J. Jazzing up Eclipse with collaborative tools. (2003), 45–49.

9. Cockburn, A. and Williams, L. *The Costs and Benefits of Pair Programming*. Addison Wesley, 2001.

10. D'Mello, S. and Graesser, A. Automatic Detection of Learner's Affect From Gross Body Language. *Appl. Artif. Intell. 23*, 2 (2009), 123–150.

11. Dabbish, L. and Kraut, R.E. Controlling interruptions: awareness displays and social motivation for coordination. In *Proceedings of the 2004 ACM conference on Computer supported cooperative work*. ACM Press, Chicago, Illinois, USA, 2004, 182–191.

12. Dagenais, B., Ossher, H., Bellamy, R., Robillard, M., and de Vries, J. Moving into a new software project landscape. 275–284.

13. Dewan, P., Agrawal, P., Shroff, G., and Hegde, R. Experiments in Distributed Side-by-Side Software Development. ICST/IEEE (2009).

14. Dewan, P. and Choudhary, R. Coupling the User Interfaces of a Multiuser Program. *ACM Transactions on Computer Human Interaction 2*, 1 (1995), 1–39 %! Coupling the User Interfaces of a Multiuser Program.
15. Dewan, P. and Hegde, R. Semi-Synchronous Conflict Detection and Resolution in Asynchronous Software Development. Springer (2007), 159–178.

16. Dewan, P. Towards and Beyond Being There in Distributed Collaborative Software Development. In *Collaborative Software Engineering*. Springer-Verlag, 2010.

17. Dourish, P. and Bly, S. Portholes: Supporting Awareness in a Distributed Work Group. (1992), 541–547.

18. Elvin, S.C., Fitzpatrick, G., Mansfield, T., et al. *Instrumenting and Augmenting the Workaday World with a Generic Notification Service called Elvin*. 1999.

19. Fogarty, J., Hudson, S.E., Atkeson, C.G., et al. Predicting human interruptibility with sensors. *ACM Trans. Comput.-Hum. Interact. 12*, 1 (2005), 119–146.

20. Fogarty, J., Ko, A.J., Aung, H., Golden, E., Tang, K., and Hudson, S. Examining Task Engagement in Sensor-Based Statistical Models of Human Interruptibility. *In Proceedings of the SIGCHI conference on Human factors in computing systems*, (2005), 331–340.

21. Frost, R. Jazz and the Eclipse way of collaboration. *IEEE Software*, (2007), 114–117.

22. Grundy, J. Engineering Component-based, User-Configurable Collaborative Editing Systems. Kluwer Academic Publishers (1998), 111–128. Engineering Component–based, User–Configurable Collaborative Editing Systems.

23. Gutwin, C. and Greenberg, S. A Descriptive Framework of Workspace Awareness for Real-Time Groupware. *CSCW 11(3):*, (2002), 411–446.

24. Hegde, R. and Dewan, P. Connecting Programming Environments to Support Ad-Hoc Collaboration. IEEE/ACM (2008).

25. Herbsleb, J. and Grinter, R.E. Splitting the Organization and Integrating the Code: Conway's Law Revisited. Proceedings of International Conference on Software Engineering. *In Proceedings of the 2000 ACM conference on Computer supported cooperative work*, (1999), 85–99.

26. Herbsleb, J.D., Atkins, D.L., Boyer, D.G., Handel, M., and Finholt, T.A. Introducing instant messaging and chat in the workplace. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM (2002), 171–178.

27. Herbsleb, J.D., Mockus, A., Finholt, T.A., and Grinter, R.E. Distance, dependencies, and delay in a global collaboration. *In Proceedings of the 2000 ACM conference on Computer supported cooperative work*, (2000), 319–328.

28. Hollan, J. and Stornetta, S. Beyond Being There. (1992), 119–126.

29. Horvitz, E., Apacible, J., and Koch, P. BusyBody: Creating and Fielding Personalized Models of the Cost of Interruption. (2004).

30. Iqbal, S. and Bailey, B. Understanding and Developing Models for Detecting and Differentiating Breakpoints during Interactive Tasks. *In Proceedings of the SIGCHI conference on Human factors in computing systems*, (2007), 697–706.

31. Jadud, M. A first look at novice compilation behavior using BlueJ. (2005), 25–40.

32. Jadud, M.C. Methods and tools for exploring novice compilation behaviour. *Proceedings of the second international workshop on Computing education research*, ACM (2006), 73–84.

33. Junuzovic, S., Dewan, P., and Rui, Y. Read, write, and navigation awareness in realistic multi-view collaborations. IEEE (2007), 494–503.

34. Junuzovic, S., Inkpen, K., Hegde, R., Zhang, Z., Tang, J., and Brooks, C. What did i miss?: in-meeting review using multimodal accelerated instant replay (air) conferencing. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM (2011), 513–522.

35. Kapoor, A., Burleson, W., and Picard, R.W. Automatic prediction of frustration. *Int. J. Hum.-Comput. Stud. 65*, 8 (2007), 724–736.

36. Kersten, M.M.G.C. Mylar: A degree-of-interest model for IDEs. *Proceedings of the 4th international conference on Aspect-oriented software development*, (2005), 159–168.

37. Ko, A.J., Myers, B.A., and Aung, H.H. Six Learning Barriers in End-User Programming Systems. *Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing*, IEEE Computer Society (2004), 199–206.

38. Landis, J.R. and Koch, G.G. The measurement of observer agreement for categorized data. (1977), 159–174.

39. LaToza, T.D. and R., D. Maintaining mental models: a study of developer work habits. *Proc. ICSE*, IEEE (2006), 492–501.

40. Liu, Y.S.E. A Lightweight Project-Management Environment for Small Novice Teams. *In Proceedings of 3rd International Workshop on Adoption-Centric Software Engineering*, (2003), 42–48.

41. McKeogh, J. and Exton, D.C. Eclipse plug-in to monitor the programmer behavior. (2004).

42. Murphy, C., Kaiser, G., Loveland, K., and Hasan, S. Retina: Helping Students and Instructors Based on Observed Programming Activities. (2009), 178–182.

43. Myers, A.J.K. and Brad, A. Debugging reinvented: asking and answering why and why not questions about program behavior. ACM %! Debugging reinvented: asking and answering why and why not questions about program behavior (2008).

44. Nawrocki, J.R., Jasinski, M., Olek, £., and Lange, B. Pair Programming vs. Side-by-Side Programming. In *Software Process Improvement*. Springer Berlin / Heidelberg, 2005, 28–38.

45. Needleman, S. and Wunsch, C. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology.*, (1970).

46. Norris, C., Barry, J., Fenwick Jr., B., Reid, K., and Rountree, J. ClockIt: Collection quantitative data on how beginning software developers really work. (2008).

47. Piech, C., Sahami, M., Koller, D., Cooper, S., and Blikstein, P. Modeling how students learn to program. *In Proceedings of the 43rd ACM technical symposium on Computer Science Education*, (2012), 153–160.

48. Sarma, A., G, B., and Hoek, A. van der. Towards Supporting Awareness of Indirect Conflicts across Software Configuration Management Workspaces. (2007), 94–103.

49. Schummer, T. and Haake, J.M. Supporting distributed software development by modes of collaboration. (2001), 79–98.

50. Sharma, G., Shroff, G., and Dewan, P. Workplace Collaboration in a 3D Virtual Office. *IEEE International Symposium on Virtual Reality Innovations*, (2011).

51. Sharon, Y. Eclipseye-spying on eclipse. 2007.

52. Shrauger, J.S. and Osberg, T.M. The Relative Accuracy of Self-Predictions and Judgments by Others in Psychological Assessment. *Psychological Bulletin 90*, 2 (1981), 322–351.

53. Staiano, J., Menéndez, M., Battocchi, A., De Angeli, A., and Sebe, N. UX_Mate: from facial expressions to UX evaluation. *Proceedings of the Designing Interactive Systems Conference*, ACM (2012), 741–750.

54. Tang, J.C., Liu, S.B., Muller, M., Lin, J., and Drews, C. Unobtrusive but invasive: using screen recording to collect field data on computer-mediated interaction. *In CSCW '06: Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, (2006), 479–482.

55. Teasley, S., Covi, L., Krishnan, M.S., and Olson, J.S. How does radical collocation help a team succeed? *Proceedings of the 2000 ACM conference on Computer supported cooperative work*, ACM (2000), 339–346.

56. Tee, K., Greenberg, S., and Gutwin, C. Providing artifact awareness to a distributed group through screen sharing. *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, ACM (2006), 99–108.

57. Twidale, M.B. Over the Shoulder Learning: Supporting Brief Informal Learning. *Comput. Supported Coop. Work 14*, 6 (2005), 505–547.

58. Vessey, I. Expertise in debugging computer programs: an analysis of the content of verbal protocols. *IEEE Trans. Syst. Man Cybern. 16*, 5 (1986), 621–637.

59. Witten, I.H.F.E. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations, 1999.