IJESRT

INTERNATIONAL JOURNAL OF ENGINEERING SCIENCES & RESEARCH TECHNOLOGY

Effectiveness of Software Development Process Using Programmer Ranker Algorithm in Pair Programming

Manisha Giri[*1], Saket Soni[2]

[*1,2]Department of Computer Science Engineering Chhatrapati Shivaji Institute of Technology, Durg India

manisha.giri1@gmail.com

Abstract

Pair programming is a style of programming in which two programmers work side-by-side at one computer, continuously collaborating on the same design, algorithm, code, or test. In industry, the practice of pair programming has been shown to improve product quality, improve team spirit, aid in knowledge management, and reduce product risk. In software industry, pair programming also improves associate's morale, helps associates to be more successful, and improves associates retention in an information technology major. Project efficiency of pairs in program design and implementation tasks is identified by using pair programming concept. Pair programming involves two developers simultaneously collaborating with each other on the same programming task to design and code a solution. Programming aptitude tests (PATs) have been shown to correlate with programming performance. In this paper we will measure time productivity using pair programming, in two important ways: One is elapsed time to complete the task and the other is the total effort/time of the programmers completing the task. Using Programmer Ranker Algorithm (PRA) we will generate pair and Rank will be provided to each pair of Junior, Senior of industry. After providing rank the best pair can be allocated to Embedded Software project type, Semi detached Software project type and Organic Software project type respectively.

Keywords: Pair programming, PAT, Collaborative programming, Team building, PRA.

Introduction

Each day, software applications grow larger and more complicated; perhaps, then, it is best for complexity of these application to be tacked by two humans at a time. Much of the increase in interest in Pair Programming is due to introduction of Extreme Programming (XP) [1].Pair programming is a software practice that involves a pair of programmers simultaneously collaborating with each other on the same programming effort [2], [3], [4], [5], [6]. One programmer controls the keyboard and implements the program. The other programmer watches, identifies defects, considers the direction of the work, and communicates with the customer/client. Sitting side by side at one computer, two colleagues collaborate on solving the problem, designing the algorithm and coding. In pair work, both partners actually perform each activity together in collaborative manner, making it possible to create and continuously review what is being created. Pairs regularly switch the driver and navigator roles and rotate their partners with other teams: This practice is thought to facilitate skills transfer and job rotation [7].

Many different variations of pair programming experiments have been reported but the results of these studies vary substantially (Williams 2000; Flor 1991; Nosek 1998; Nawrocki 2001; Hulkko 2005; Arisholm 2007; Ciolkowski 2002; Bellini 2005; Lui 2006; Lui 2008). This is mainly due to several consistent variables, which are difficult to control. Previous studies in pair programming have only addressed the basic understanding of the productivity of pairs and they have not addressed the variation in productivity between pairs of varying skills and experience, such as between novice–novice and expert–expert.

Several previous controlled experiments have validated the following quantitative benefits of pair programming over individual programming.

Significant improvements in functional correctness.

1. Various other measures of quality of the programs being developed.
2. Reduced duration (a measure of time to market), with only minor additional overhead in terms of total programmer hours (a measure of cost or effort)

3. Reduced the elapsed time and produced better software quality.

In this paper we have proposed Programmer Ranker Algorithm (PRA) for evaluating Programmer's Effort in Context of Pair Programming which will produced the best pairs from the individuals and then provide the ranking to the selected pairs by using Halstead Complexity Metrics. PRA is fundamentally different from the other researches in PP as it is using Programming Aptitude Test (PAT) and Software Metrics, the aim is to detect more defects and adjust implementation strategy just when code is written. PRA will contributes towards quality improvement, reliable and bug free software development.

The remainder of this paper is organized as follows: Section 2 provides a brief history of the use of pair programming. The section begins with a discussion with a classic Horse-Trading Problem (Mayer's Problem) done by Kim Man Lui. This exercise will prove that collaborative problem solving makes a difference when comparing pairs and individuals. In section 3 we will address the use of Programming Aptitude Tests (PAT) to evaluate pairs in program design. Section 4 defines a new measurement of time productivity. In section 5 we will discuss the Halstead Program Complexity Metrics which is being used for measuring programming skills of the pairs and provide final ranking. Section 6 identifies the problem in the existing system. Section 7 explains our approach of pair programming that is Programmer Ranker Algorithm (PRA). Section 8 provides the pair programming results. The final section provides concluding remarks and points some possible directions for future research.

Background

In 1998, Nosek conducted an experiment with five pairs and five individual professionals to solving a challenging problem. They were asked to write an UNIX script that performed a database consistency check. The programmers were well versed in UNIX script but had not performed that kind of task before. The controlled experiment showed that Pair Programming shortened the elapsed time and produced better software quality than individual programming. This creates evidence that collaboration improves the problem-solving process and produces more efficient code.

In an academic environment, the most cited study is probably that described in [8] in which 13 university students worked individually on a project and 28 choose to work in pairs. The finding showed that the code produced by the passed more automated test over four different programming exercises. This resulted that pair programming in software development yields better product in less time. The programmer feels happier, more confident.

Previous studies in pair programming have only addressed the basic understanding of the productivity of pairs and they have not addressed the variation in productivity between pairs of varying skills and experience, such as between novice–novice and expert–expert. In this paper we have proposed a quantitative method to develop a model for software development using Pair Programming and assessing effectiveness, variation in effectiveness between pairs of varying skills with respect to the coding phase of the software development.

Classic Horse Trading Problem: Understanding Pair VS Solo

A well-known problem called the Horse-Trading Problem by Maier can help us explore collaborative problem solving. We have replicated this experiment; the following summarizes the preparation and results.

The Horse-Trading Problem is a simple question as seen in Figure1.

> A man buys a horse for $60 and then sells it for $70. Later he buys the horse back for $80 and sells it again for $90. So, how much did the man earn?

Figure.1 Classic Horse-Trading Problem

The problem was part of a class activity intended to show the students the validity of collaborative problem solving. This experiment was carried out on students for learning purposes; therefore the experiment was not strictly monitored. Students were asked to solve the problem either alone or in pairs; they were also allowed to form their own groups with as many members as they referred (see Table 1).

Group No	1	2	3	4	5	6
Members/Group	10	10	4	2	3	1

Table 1: Group Distribution

Afterward, each group was handed a piece of paper with the problem printed on it. Groups were allowed to refer back to the question as needed throughout the problem solving process, and were able to use as much time as necessary to solve the problem. Finally, each group wrote down their respective answers and submitted it.

Most of the groups were able to work out a solution in around three minutes. The time needed to

determine a solution is not considered, as the difference in solution times may be large in terms of percentage, but overall the time is not significant as these differences are too short. More importantly, though, not all groups were able to correctly solve the problem by calculating the amount the man actually earns.

Result

The observations revealed significant improvements in the average percentage of accuracy when comparing a group with only one member to a group with more than three members. Groups 1 and 2 offer a strong statistical basis as evidence and such results are consistent with sociological research findings (Maier 1969) (see Figure 2). However, when the group size consists of five or more, the correction percentage dropped slightly. This result could be related to ergonomics as the classroom seats arranged in fixed rows. A group of two may be able to communicate side-by-side effectively, but for groups larger than three a round table is needed to facilitate communication and collaboration.

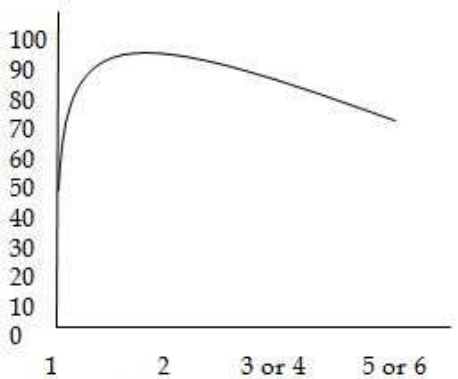


Figure.2 Accuracy graph

Relation between Programming Test and Performance

Programming aptitude tests (PATs) have been shown to be related to programming task related capabilities [26], [27], [28], [29], [30], [31], [32], [33], [34], [35], [36], [37]. The IBM Programmer Aptitude Test (IBM-PAT) is the best known test for measuring programming aptitude [34]. IBM-PAT does not contain tests for knowledge of specific language commands. The test includes a number of reasoning tasks: procedural problems, classification knowledge, deduction questions (also called serial questions), and mathematical reasoning.

PAT scores have been used by some organizations to prescreen programmer candidates for job interviews. McNamara and Hughes [27] conducted research involving 57 professional programmers, with an average age of 25 years, who have 15 months of relevant work experience. The study attempted to relate PAT scores with the following:

1. Actual job performance as a programmer,
2. Future programming potential,
3. System-analyst potential, and
4. Management potential, as graded by their supervisors.

Correlations of 0.40 ðp < 0:01Þ, 0.41 ðp < 0:01Þ, and 0.46 ðp < 0:01Þ were obtained between PAT and 1, 2, 3, respectively. For 4, it was 0.30 ðp < 0:05Þ. The correlation between PAT scores and system-analyst potential is found to be significantly higher than the others.

In addition to studies of PATs with professional programmers [26], [27], [28], [29], [30], [31], [32], studies of PATs with students have also been conducted [33], [34], [35], [36], [37]. Tukiainen and Mo¨nkko¨nen [36] have reexamined PATs to predict the abilities of student programmers. They used a PAT that was designed by Huoman in 1989 [37].

In a study in 2006, 30 computer science students were tested and results showed that scores of playing the Mastermind game, which also ignores computer language skills, correlated with in-class programming test scores at the 0.6 level [38].

In summary, most studies show that PAT scores are correlated with job performance in general but not necessarily with program design performance in particular. Most PATs do not require those being tested to have specific knowledge of any programming language or development environment. However, PATs measure performance on a number of reasoning tasks critically important to program design: procedural problems, classification knowledge, deduction questions, and mathematical reasoning.

Measuring Productivity

Basic COCOMO computes software development effort (and cost) as a function of program size. Program size is expressed in estimated thousands of source lines of code (SLOC). COCOMO applies to three classes of software projects:

Organic projects - "small" teams with "good" experience working with "less than rigid" requirements.

Semi-detached projects - "medium" teams with mixed experience working with a mix of rigid and less than rigid requirements.

Embedded projects - developed within a set of "tight" constraints. It is also combination of organic

and semi-detached projects.(hardware, software, operational, ...).
The basic COCOMO equations take the form:

Effort Applied (E) = ab (KLOC)bb [man-months]
Development Time (D) = cb (Effort Applied)db [months]
People required (P) = Effort Applied / Development Time [count]

where, KLOC is the estimated number of delivered lines (expressed in thousands) of code for project.
The coefficients ab, bb, cb and db are given in the following table:

Software project	ab	Bb	Cb	db
Organic	2.4	0.5	2.5	0.38
Semi-detached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

Table 2: Coefficients

From above table we can say that Embedded Software project type require more effort as compare to Semi-detached and Organic. Hence Embedded Software project should be allotted to high ranked pair.

Lui and Chan proposed the same COCOMO model in PP. Time productivity can be measured in two important ways: One is elapsed time to complete the task and the other is the total effort/time of the programmers completing the task. Both important measurements of time can be incorporated in a single measurement, that is, the Relative Effort Afforded by Pairs (REAP) [39]:

= (finish time of pair)*2 - (finish Time of Individual) / (finish Time of Individual) * 100

There are five cases for us to consider with REAP:

1. REAP < 0,
2. REAP = 0,
3. REAP is between 0 and 100,
4. REAP = 100, and
5. REAP > 100.

When REAP is negative, the total time of pair programmers is less than the time of the individual programmer, that is, pairs are actually more efficient than a single pair programmer and it is less costly to use pair programmers than individual programmers.

If REAP is zero, this is a break-even point, where the total time of pair programming is the same as individual programming, but pair programming halves the elapsed time required for individual programming. When REAP is greater than zero but is less than 100 percent, pairs require more total man hours to complete the task but are faster than individual programmers, that is, the elapsed time to complete is less for pairs than for individual programmers. This can be useful when the critical issue is time to market [4], [6]. As windows of opportunity and product life cycles have been shortening in recent years, premium pricing and higher sales levels that can accrue to early-mover companies can make it worthwhile for them to spend more on short-term development costs [32]. Pair programming provides an alternative to accelerate software programming beyond dividing up programming tasks. Two examples in this category are 1) the Nosek [4] results that equate to a REAP of 46 percent and 2) the Williams [6] results that equate to a REAP of 15 percent.

If REAP is around 100 percent, the elapsed time for pair programmers is almost the same time as in the individual programmer; therefore, pair programming doubles the total man hours as compared to individual programming. When REAP is greater than 100 percent, then the elapsed time for pair programming is longer than the time for an individual programmer. REAP can also be used in measurements for non programming related tasks. For example, in a controlled non programming experiment, Lazonder compared pairs of students against single students in Web search tasks and the REAP equates to 34.4 percent [33].

Halstead Complexity Metrics

Halstead complexity measures are software metrics introduced by Maurice Howard Halstead [56] as part of his treatise on establishing an empirical science of software development. Halstead makes the observation that metrics of the software should reflect the implementation or expression of algorithms in different languages, but be independent of their execution on a specific platform. These metrics are therefore computed statically from the code.

Halstead's goal was to identify measurable properties of software, and the relations between them. This is similar to the identification of measurable properties of matter (like the volume, mass, and pressure of a gas) and the relationships between them (such as the gas equation). Thus his metrics are actually not just complexity metrics.

Calculation

For a given problem, Let:

n1= the number of distinct operators
n2= the number of distinct operands
N1= the total number of operators
N2= the total number of operands

From these numbers, several measures can be calculated:
Program vocabulary: n = n1 +n2
Program length: N = N1 + N2
Calculated program length: N' = n1 log2 n1 + n2 log2 n2
Volume: V = N log2 n
Difficulty: D = n1/2 * N2 / n2
Effort: E = D * V

The required Programming Time (T) for a program P of effort E is defined as:

T = E / S = [n1*N2*N *log2 n / 2*n2*S]

where S is the Stroud number, defined as the number of elementary discriminations performed by the human brain per second. The S value for software scientists is set to 18 [Hamer 1982]. The unit of measurement of T is the second.

In 1967, psychologist John M. Stroud suggested that the human mind is capable of making a limited number of mental discrimination per second (Stroud Number), in the range of 5 to 20.
Number of delivered bugs : B = [E^(2/3) / 3000]or, more recently, B = V / 3000 is accepted.

Halstead Metrics: Example

```
void sort( int*a, intn )
 inti, j, t;
 {
     if ( n <2 )
         return;
     for( i=0 ; i <n-1; i++)
     {
       for( j=i+1 ; j <n ; j++)
       {
        if( a[i] > a[j])
        {
          t=a[i];
          a[i] =a[j];
          a[j]=t;
        }
      }
    }
 }
```

Ignore the function definition.

Operators Count				Operands Count	
3	<	3	{	1	0
5	=	3	}	2	1
1	>	1	+	1	2
2	-	2	++	6	a
9	;	2	if	8	i
4	(	1	int	7	j
4	)	1	return	3	n
6	[]			4	t

Table 3: Count Operators and Operands

Program Vocabulary: 24	Program Length: 80
Calculated Program Length : 89.64	Volume: 366.79
Difficulty: 36.42	Effort: 13358.49
Estimated Bug: 0.12	Programming Time: 0.0067

Table 4 Computed Halstead Metrics Values

Problem Identification

Each day, software applications grow larger and more complicated; perhaps, then, it is best for complexity of these application to be tacked by two humans at a time. Much of the increase in interest in Pair Programming is due to introduction of Extreme Programming (XP) [1].Pair programming is a software practice that involves a pair of programmers simultaneously collaborating with each other on the same programming effort [2], [3], [4], [5], [6].

Many different variations of pair programming experiments have been reported but the results of these studies vary substantially (Williams 2000; Flor 1991; Nosek 1998; Nawrocki 2001; Hulkko 2005; Arisholm 2007; Ciolkowski 2002; Bellini 2005; Lui 2006; Lui 2008). This is mainly due to several consistent variables, which are difficult to control. Previous studies in pair programming have only addressed the basic understanding of the productivity of pairs and they have not addressed the variation in productivity between pairs of varying skills and experience, such as between novice–novice and expert–expert.

Much work has been carried out on improving the efficiency of the pairs in Pair Programming. However, all these works suffer from finding efficient pairs. We aim to obtain an efficient algorithm which will produced the best pairs from the individuals and then provide the ranking to the selected pairs by using Average REAP and Halstead Complexity Metrics, the aim is to detect more defects and adjust implementation strategy just when code is written. PRA will contributes towards quality

improvement, reliable and bug free software development.

Our Approach

In this work, we have proposed a Programmer Ranker Algorithm (PRA) with Halstead Complexity Software Metrics to develop a model for improving the effectiveness of Software Development Process in Pair Programming. PRA is fundamentally different from all the previous algorithms and researches. As PRA uses Programming Aptitude Test (PAT) with Software Metrics, the aim is to detect more defects and adjust implementation strategy just when code is written. We have applied PP in the coding phase of software development. PP is not solely reserved to coding phase but can be applied to other phase of the process such as analysis and design.

As shown in the Figure 3, PRA algorithm takes input of REAPs of successful pairs and individuals and finds the ranking of pairs with respect to an individual using PAT performance and also finds pairs ranking by computing average reap and Halstead Complexity Metrics considering technical performance along with PAT performance.

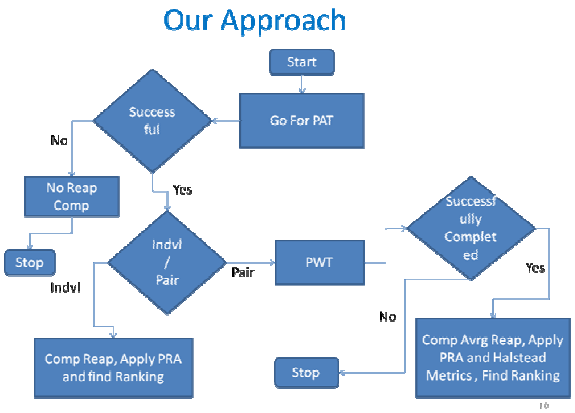
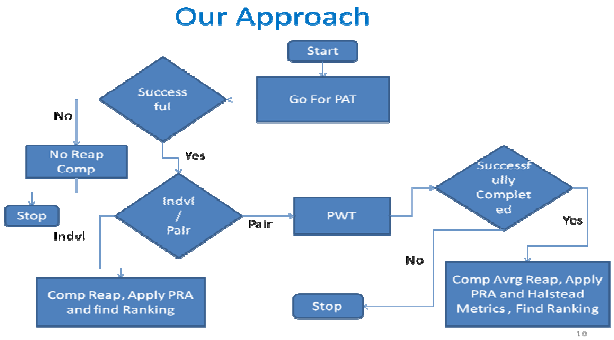


Figure.3 Our Approach



Programmer Ranker Algorithm(PRA)

Pair programming involves two developers simultaneously collaborating with each other on the same programming task to design and code a solution. Programming aptitude tests (PATs) have been shown to correlate with programming performance. In this paper we will measure time productivity using pair programming, in two important ways: One is elapsed time to complete the task and the other is the total effort/time of the programmers completing the task. Using Programmer Ranker Algorithm (PRA) we will generate pair and Rank will be provided to each pair of Junior, Senior of industry using Halstead Program Complexity Metrics considering PAT and PWT performance both. After providing rank the best pair is allocated to Embedded Software project type, Semi detached Software project type and Organic Software project type respectively.

We have proposed Programmer Ranker Algorithm (PRA), which improves the previous work done in context of pair programming (PP). It provides a quantitative method to find the best pairs from the given individuals. In PRA, REAP will be calculated considering elapsed time of successful individuals and pairs in PAT and the best pairs and their rank will be provided. The proposed method will be discussed in the following sub-sections.

In the proposed PRA best pairs will be found from the given list of individuals. But in the proposed algorithm following assumptions are made.

- Make sure that every programmer fully understands the concept of pair programming before trying to apply it.
- Try to describe the expected goals with pair programming before the work starts.
- Try to set up some kind of rules of how and when to pair up (i.e on random basis or on some domain expertise basis) before facing PAT, what roles there are, and what each pair member is expected to do.
- Let people work on their own if they feel they need to.

In this paper we are pairing up the individuals for PAT and PWT on random basis.

The process of finding best pairs and their ranking is divided into following parts.

Programming Aptitude Test (PAT).
Programming Written Test (PWT).
Programmer Ranker Algorithm (PRA).
Halstead Complexity Metrics.

Programming Aptitude Tests (PATs)

Programming aptitude tests (PATs) have been shown to be related to programming task related capabilities [26], [27], [28], [29], [30], [31], [32], [33], [34], [35], [36], [37]. The IBM Programmer Aptitude Test (IBM-PAT) is the best known test for measuring programming aptitude [34]. IBM-PAT does not contain tests for knowledge of specific language commands. In this paper we are also using PAT as a task to measure elapsed time of individuals and pairs. It is compulsory for every individual and pair to go through PAT. The pairs who have been successful in PAT will face PWT. In our PAT there will be different sets of questions which can be assigned by admin (project manager) to different individuals and pairs but the set should be same for a particular process. The sets will have four sections- Numerical, Reasoning, English and Technical. There will be sectional as well as aggregate cut off. The subjects (Individuals or Pairs) will have to clear the both cut offs to be considered for PRA as input.

Programming Written Test (PWT)

The subjects (only pairs) who have been successful in Programming Aptitude Test (PAT) will go for PWT. PWT is used to assess the style of coding of the subject (only Pairs). The programs of the selected best pairs generated through PRA will act as input for Halstead Program Complexity Metrics and then estimated number of bugs and their final ranking will be computed. In this paper we are computing Halstead Metrics for C and C++ language only.

Programmer Ranker Algorithm (PRA)

The PRA uses Relative Effort Afforded by Pairs (REAP) for finding best pairs and their ranking. We have proposed Programmer Ranker Algorithm in two parts- PRA (a) and PRA (b).The step-by-step procedure of PRA (a) and PRA (b) algorithm is described as follows.

Algorithm 1: Programmer Ranker Algorithm (PRA) (a)

```
Procedure Gen_Pair ()
 //indiTime -> Finish Time of Individual
 //p1Time   -> Pair-I Finish Time
 //p2Time   -> Pair-II Finish Time
 //p3Time   -> Pair-III Finish Time
1.        REAP1 = (((p1Time * 2) - indiTime) / indiTime) * 100;
2.        REAP2 = (((p2Time * 2) - indiTime) / indiTime) * 100;
3.        REAP3 = (((p3Time * 2) - indiTime) / indiTime) * 100;
4.    if (REAP1 < REAP2)
          {
            if (REAP1 < REAP3)
            {
              "The Pair One is Best compare to the others";
            }
       else
         {
           "The Pair three is Best compare to the others";
         }
     }
   else if (REAP1 > REAP2)
    {
       if (REAP2 < REAP3)
       {
         "The Pair two is Best compare to the others";
       }
       else
       {
         "The Pair three is Best compare to the others";
       }
     }
 End Gen_Pair
```

Algorithm 2: Programmer Ranker Algorithm (PRA) (b)

1) I1, I2, I3...In ; //Individuals successful in PAT
   T1, T2, T3...Tn ; // Elapsed Time of Individuals in PAT

2) P1, P2, P3....Pm ; // Pair successful in PAT and have PWT
   S1, S2, S3...Sm ; // Elapsed Time of Pairs in PAT

3) Calculate REAP of a Pair with all individuals
  a) For P1
    R1,1 = [(2 * S1 – T1)/ T1] * 100 ;
    R1,2 = [(2 * S1 – T2)/ T2] * 100 ;
    R1,3 = [(2 * S1 – T3)/ T3] * 100 ;
                .
                .
    R1,n = [(2 * S1 – Tn)/ Tn] * 100 ;

b) For P2
    R2, 1 = [(2 * S2 – T1)/ T1] * 100;
    R2, 2 = [(2 * S2 – T2)/ T2] * 100;
    R2, 3 = [(2 * S2 – T3)/ T3] * 100;
                .

.
    R2, n = [(2 * S2 – Tn)/ Tn] * 100;
.
.

. // similarly we will calculate REAP of other Pairs with all Individuals
.
.
c) For Pm
Rm, 1 = [(2 * Sm – T1)/ T1] * 100;
Rm, 2 = [(2 * Sm – T2)/ T2] * 100;
Rm, 3 = [(2 * Sm – T3)/ T3] * 100;
.
.
.
.
Rm, n = [(2 * Sm – Tn)/ Tn] * 100;
4) Calculate Average REAP of Each Pairs
AVG1 = (R1,1 + R1,2 + R1,3 ….+ R1,n) / n ;
AVG2 = (R2,1 + R2,2 + R2,3 ….+ R2,n) / n ;
AVG3 = (R3,1 + R3,2 + R3,3 ….+ R3,n) / n ;
.
.
.
AVGm = (Rm,1 + Rm,2 + Rm,3 ….+ Rm,n) / n ;
5) Sort the Average REAPS in Ascending Order.
6) Apply PRA (a) to pairs sorted by Average REAP in step (5) to find the required number of best pairs for which Halstead Program Complexity Metrics will be computed.
7) The pair having less estimated number of bugs calculated in step (5) will have higher rank than other pair having more number of estimated bugs.

Halstead Complexity Metrics

Halstead complexity measures are software metrics introduced by Maurice Howard Halstead [56] as part of his treatise on establishing an empirical science of software development. Halstead makes the observation that metrics of the software should reflect the implementation or expression of algorithms in different languages, but be independent of their execution on a specific platform. These metrics are therefore computed statically from the code.

Halstead's goal was to identify measurable properties of software, and the relations between them. This is similar to the identification of measurable properties of matter (like the volume, mass, and pressure of a gas) and the relationships between them (such as the gas equation). Thus his metrics are actually not just complexity metrics.

Calculation

For a given problem, Let:

n1= the number of distinct operators
n2= the number of distinct operands
N1= the total number of operators
N2= the total number of operands

From these numbers, several measures can be calculated:

Program vocabulary: n = n1 +n2
Program length: N = N1 + N2
Calculated program length: N' = n1 log2 n1 + n2 log2 n2
Volume: V = N log2 n
Difficulty: D = n1/2 * N2 / n2
Effort: E = D * V

The required Programming Time (T) for a program P of effort E is defined as:

T = E / S = [n1*N2*N *log2 n / 2*n2*S]

where S is the Stroud number, defined as the number of elementary discriminations performed by the human brain per second. The S value for software scientists is set to 18 [Hamer 1982]. The unit of measurement of T is the second.

In 1967, psychologist John M. Stroud suggested that the human mind is capable of making a limited number of mental discrimination per second (Stroud Number), in the range of 5 to 20.

Number of delivered bugs : B = [E^(2/3) / 3000]or, more recently, B = V / 3000 is accepted.

An Illustration

PAT Report Card:

User Id	User Type	PAT Set	PAT Status	Elapsed Time(min)
U0001	I	Set_01	Cleared	10.21
U0002	I	Set_01	Not Cleared	15.01
U0003	I	Set_01	Cleared	07.12
U0004	I	Set_01	Not Cleared	14.03
U0005	P	Set_01	Cleared	10.42
U0006	P	Set_01	Cleared	03.06
U0007	P	Set_01	Not Cleared	15.00
U0008	P	Set_01	Cleared	05.10

Figure.4 Report Card

PWT Details of Successful Pairs

U0006

```
//Sorting of an Array
void sort( int *a, int n )
{
  int i, j, t;
  if( n <2 )
  return;
 for( i=0 ; i <n-1; i++)
 {
   for( j=i+1 ; j <n ; j++)
    {
      if( a[i] > a[j])
      {
        t=a[i];
        a[i] =a[j];
        a[j]=t;
      }
    }
  }
}
```

U0008

```
/* Array sorting */
void array_sort(int *a, int n )
{
  int i, j, t, temp, q;
  if( n <2 )
  return;
 for( i=0 ; i <n-1; i++)
 {
   for( j=i+1 ; j <n ; j++)
    {
      if( a[i] > a[j])
      {
        t=a[i]; temp = t; q = temp; t=q;
        a[i] =a[j];
        a[j]=t;
      }
    }
  }
}
```

Figure.5 PWT Details

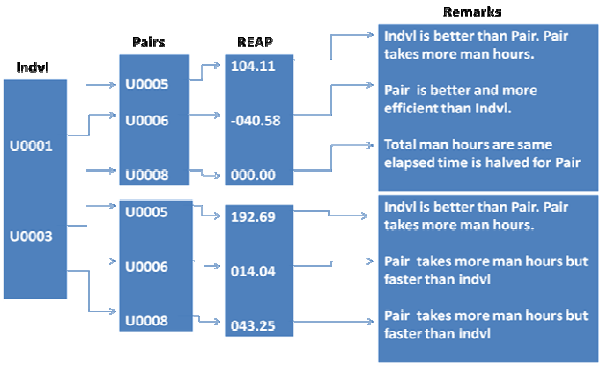


Figure.6 Pair vs Individual

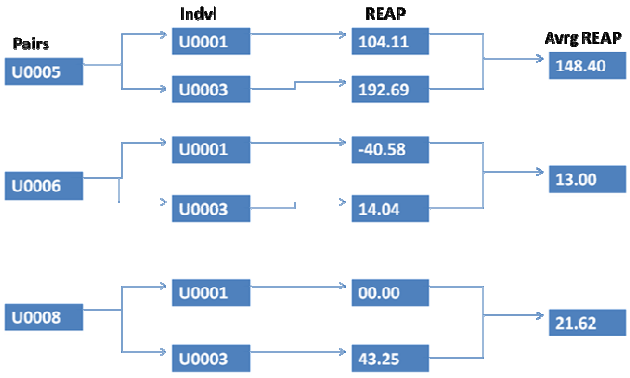


Figure.7 Average Reap

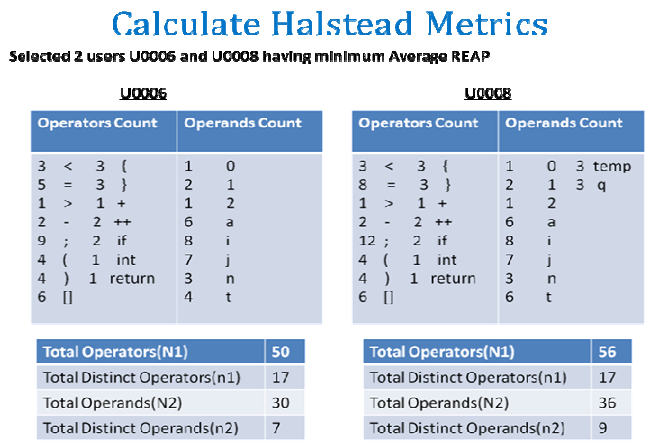


Figure.8 Operators and Operands Count

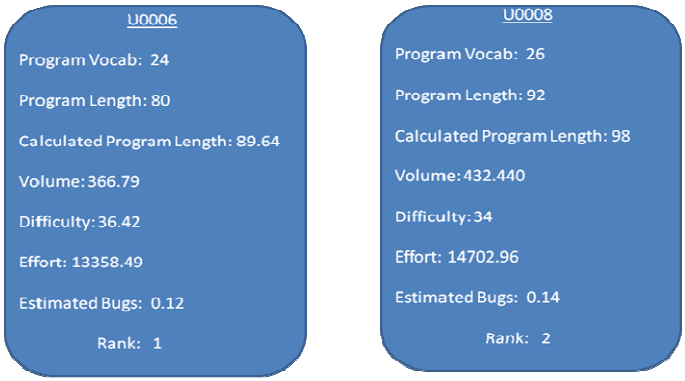


Figure.9 Pair Ranking

Experimental Analysis

Pair VS Individual

Fig 10 provides the comparison of pair programmers and individuals. It shows that the effort spent to develop the project can be reduced by pair programming. Programmer Ranker Algorithm (PRA) is used to generate pairs and the pairs generated by PRA can significantly reduce the Project development time and cost.

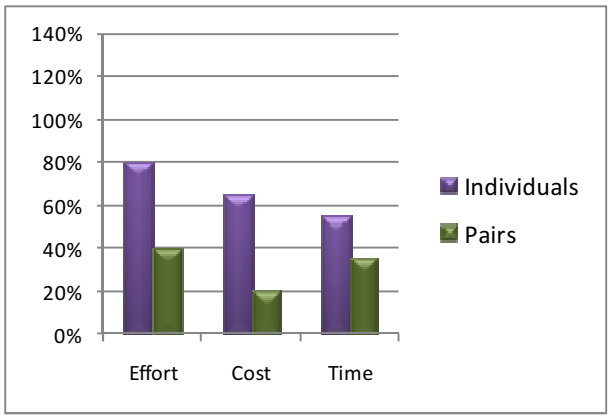


Figure.10 Comparison of pair programmers and individuals

Analysis and Discussion

Based on the above, we can conclude that PP is very effective in the Software Development Process and can be incorporated in the industry environment. In this work, we proposed a model for software development using pair programming suitable in industry environment. We will be able to compare Individual vs Pairs and generate efficient pairs in context of Pair Programming. Rank will be provided to each pair of Junior, Senior of industry.

We can work towards quality improvement, reliable and bug free software.

Effectiveness of Software Development Process in Pair Programming can be achieved. We can detect more defects, bugs and adjust implementation strategy just when code is written. The result shows that the PP is more effective with respect to fastness in completion, high quality, program size, defect identification speed and defect removal rate, number of rework done etc.

Conclusion

The primary contribution of this study is to provide an overview of Pair Programming and to demonstrate the use of Programming Aptitude Test in the aspect of pair generation or team building that facilitates to make pair of newly hired programmers in an industry.

In our study, we have pointed out the use of PAT as a measurement of productivity and to evaluate the performance of individuals and pairs in order to generate the correct pairs. Our study showed that junior individuals may lack the necessary skills to perform tasks with acceptable quality, in particular, on more complex systems. Junior pair programmers achieved a significant increase in correctness compared with the individuals and achieved approximately the same degree of correctness as senior individuals. Software testing is often viewed as requiring less skill than initial system development and is thus often allocated to the more junior staff. Our study concludes that, if juniors are assigned to complex tasks, they should perform the tasks in pairs.

Programmer Ranker Algorithm (PRA) will generate pair and Rank will be provided to each pair of Junior, Senior of industry. After providing rank the best pair is allocated to Embedded Software project type, Semi detached Software project type and Organic Software project type respectively. This will reduce the time and effort requires developing the Embedded Software project which will eventually reduce overall cost of software. In this work, we proposed a model for software development using pair programming suitable in industry environment. The result shows that the PP is more effective with respect to fastness in completion, high quality, program size, defect identification speed and defect removal rate, number of rework done etc. We can conclude that PP is very effective in the Software Development Process and can be incorporated in the industry environment.

Future Work

We have applied PP in the coding phase of software development. PP is not solely reserved to coding phase but can be applied to other phase of the process such as analysis and design.

Future study on PP should extend the scope of present study in two important ways. First our study suggests that the benefit of Pair Programming (PP) depends on programmer's expertise. Still our experimental task is relatively small and simple and our result might be therefore present a conservative estimates of benefits of Pair Programming. Future experiments should ideally, include larger systems and more complex task.

References

[1] Glenford J. Myers, "The Art of Software Testing". Second Edition.

[2] L.L. Constantine, Constantine on Peopleware. Yourdon Press, 1995.

[3] B. Kent, Extreme Programming Explained: Embrace Change. Addison-Wesley, 2000.

[4] J. Nosek, "The Case for Collaborative Programming," Comm. ACM, vol. 41, no. 3, pp. 105-108, 1998.

[5] L. Williams, R.R. Kessler, W. Cunningham, and R. Jeffries, "Strengthening the Case for Pair Programming," IEEE Software, vol. 17, no. 4, pp. 19-25, July/Aug. 2000.

[6] N.V. Flor, "Side-by-Side Collaboration: A Case Study," Int'l J. Human-Computer Studies, vol. 49, no. 3, pp. 201-222, 1998.

[7] L. Williams and R.R. Kessler, Pair Programming Illuminated. Addison-Wesley, 2003.

[8] Williams, L, et., al "Strengthening the case of pair programming" IEEE software 2000, 17(4),P.

[9] J. Nosek, "The Case for Collaborative Programming," Comm. ACM, vol. 41, no. 3, pp. 105-108, 1998.

[10] G. Keefer, "Extreme Programming Considered Harmful for Reliable Software," Proc. Sixth Conf. Quality Eng. in Software Technology, pp. 129-141, 2002.

[11] L. Williams, R.R. Kessler, W. Cunningham, and R. Jeffries, "Strengthening the Case for Pair Programming," IEEE Software, vol. 17, no. 4, pp. 19-25, July/Aug. 2000.

[12] A. Parrish, R. Smith, D. Hale, and J. Hale, "A Field Study of Developer Pairs: Productivity Impacts and Implications," IEEE. Software, vol. 21, no. 5, pp. 76-79, Sept./Oct. 2004.

[13] M. Ciolkowski and M. Schlemmer, "Experiences with a Case Study on Pair Programming," Proc. First Int'l Workshop Empirical Studies in Software Eng., 2002.

[14] J. Nawrocki, M. Jasin˜ ski, L. Olek, and B. Lange, "Pair Programming versus Side-by-Side Programming," Proc. 12th European Conf. Software Process Improvement, pp. 28-38, Nov. 2005.

[15] K.M. Lui and K.C.C. Chan, "Software Process Fusion: Uniting Pair Programming and Individual Programming Processes," Proc. Int'l Software Process Workshop and Int'l Workshop Software Process Simulation and Modeling, pp. 115-123, 2006.

[16] M.M. Mu¨ ller, "Two Controlled Experiments Concerning the Comparison of Pair Programming to Peer Review," J. Systems and Software, vol. 78, no. 2, pp. 166-179, 2005.

[17] M.M. Mu¨ ller, "Are Reviews an Alternative to Pair Programming," Empirical Software Eng., vol. 9, pp. 335-351, 2004.

[18] E. Arisholm, H. Gallis, T. Dyba, and D.I.K. Sjøberg, "Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise," IEEE Trans. Software Eng., vol. 33, no. 2, pp. 65-86, Feb. 2007.

[19] M.A. Poff, "Pair Programming to Facilitate the Training of Newly Hired Programmers," master's thesis, Florida Inst. of Technology,2003.

[20] H. Hulkko and P. Abrahamsson, "A Multiple Case Study on the Impact of Pair Programming on Product Quality," Proc. 27th Int'l Conf. Software Eng., pp. 495-504, 2005.

[21] K.M. Lui and K.C.C. Chan, "Pair Programming Productivity: Novice-Novice versus Expert-Expert," Int'l J. Human Computer Studies, vol. 64, pp. 915-925, 2006.

[22] M.M. Mu¨ ller and W.F. Tichy, "Case Study: Extreme Programming in a University Environment," Proc. 23rd Int'l Conf. Software Eng., pp. 537-544, 2001.

[23] "Timeline of Visual Basic," Wikipedia, http://en.wikipedia.org/, 2007.

[24] M.C. Daconta, Java for C/C+ Programmers. Wiley, 1996.

[25] "Visual Basic: Controversy," Wikipedia, http://en.wikipedia. org/, 2007.

[26] D.B. Mayer and A.W. Stalnaker, "Selection and Evaluation of Computer Personnel: The Research History of SIG/CPR," Proc. 23rd ACM Nat'l Conf., pp. 657-670, 1968.

[27] W.J. MeNamara and J.L. Hughes, "A Review of Research on the Selection of Computer Programmers," Personnel Psychology, vol. 14, pp. 39-51, 1961.

[28] G.P. Hollenbeck and W.J. McNamara, "CUCPAT and Programming Aptitude," Personnel Psychology, vol. 18, no. 1, pp. 101-106, 1965.

[29] A. Katz, "Prediction of Success in Automatic Data Processing Course," Technical Note 126,US Army Personnel Research Office, 1962.

[30] G.Y. Denelsky and M.G. McKee, "Prediction of Computer Programmer Training and Job Performance Using the AABP Test," Personnel Psychology, vol. 27, no. 1, pp. 129-137, 1974.

[31] J.M. Wolfe, "A New Look at Programming Aptitudes," Business Automation, vol. 17, pp. 36-45, 1970.

[32] J.M. Wolfe, "Perspectives on Testing for Programming Aptitude," Proc. 25th ACM/CSC-ER Ann. Conf., pp. 268-277, 1971.

[33] T.C. Oliver and W.K. Willis, "A Study of the Validity of the Programmer Aptitude Test," Educational and Psychological Measurement, vol. 23, pp. 823-825, 1963.

[34] R. Bauer, W.A. Mehrens, and J.F. Vinsonhaler, "Predicting Performance in a Computer Programming Course," Educational and Psychological Measurement, vol. 28, pp. 1159-1164, 1968.

[35] C.R. Bateman, "Predicting Performance in a Basic Computer Course," Proc. Fifth Ann. Meeting of the Am. Inst. for Decision Sciences, 1973.

[36] M. Tukiainen and E. Mo¨nkko¨nen, "Programming Aptitude Testing as a Prediction of Learning to Program," Proc. 14th Workshop Psychology of Programming Interest Group, pp. 45-57, 2002.

[37] J. Huoman, "Predicting Programming Aptitude," master's thesis, Dept. of Computer Science, Univ. of Joensuu, 1986.

[38] T. Lorenzen and H.L. Chang, "MasterMind: A Predictor of Computer Programming Aptitude," ACM SIGCSE Bull., vol. 38, no. 2, pp. 69-71, 2006.

[39] Kim Man Lui, Keith C.C. Chan, and John Teofil Nosek "The Effect of Pairs in Program Design Tasks" IEEE transactions on software engineering, VOL. 34, NO. 2, march/april 2008

[40] R.J. Calantone and C.A. Di Benedetto, "Performance and Time to Market: Accelerating Cycle Time with Overlapping

Stages," IEEE Trans. Eng. Management, vol. 47, no. 2, pp. 232-244, 2000.

[41] A.W. Lazonder, "Do Two Heads Search Better than One? Effects of Student Collaboration on Web Search Behaviour and Search Outcomes," British J. Educational Technology, vol. 36, no. 3, pp. 465- 475, 2005.

[42] R.G. Miller, Beyond ANOVA, Basics of Applied Statistics. John Wiley & Sons, 1986.

[43] A. Munzert, "Part IV: Computer IQ—Program Procedure," Test Your IQ, third ed., pp. 112-117, Random House, 1994.

[44] M. Snyder, Working with Microsoft Dynamics CRM 3.0. Microsoft Press, 2006.

[45] A. Wil van der, Workflow Management: Models, Methods, and Systems. MIT Press, 2002.

[46] T.R.G. Green, "Instructions and Descriptions: Some Cognitive Aspects of Programming and Similar Activities," Proc. Int'l Working Conf. Advanced Visual Interfaces, pp. 21-28, 2000.

[47] D. Jackson, Software Abstractions. MIT Press, 2006.

[48] N. Flor and E. Hutcheins, "Analyzing Distributed Cognition in Software Teams: A Case Study of Team Programming During Perfective Software Maintenance," Proc. Fourth Ann. Workshop Empirical Studies of Programmers, 1991.

[49] C.R. Holloman and H.W. Hendrick, "Problem Solving in Different Sized Groups," Personnel Psychology, vol. 24, no. 3, pp. 489-500, 1971.

[50] J. Puncochar and P.W. Fox, "Confidence in Individual and Group Decision Making: When 'Two Heads' Are Worse than One," J. Educational Psychology, vol. 96, pp. 582-591, 2004.

[51] L.A. Williams, "The Collaborative Software Process," PhD dissertation, Univ. of Utah, 2000.

[52] I.L. Janis, Groupthink, second ed. Houghton Mifflin, 1982.

[53] M.L. Pate, G.W. Wardlow, and D.M. Johnson, "Effects of Thinking Aloud Pair Problem Solving on the Troubleshooting Performance of Undergraduate Agriculture Students in a Power Technology Course," J. Agricultural Education, vol. 45, no. 4, pp. 1-11, 2004.

[54] A. Cockburn, Crystal Clear: A Human-Powered Methodology for Small Teams. Addison-Wesley, 2005.

Halstead, Maurice H. (1977). Elements of Software Science. Amsterdam: Elsevier North-Holland, Inc. ISBN 0-444-00205-7.