Journal of Computer Sciences

Original Research Paper

Optimising Pair Programming in a Scholastic Framework: A Design Pattern Perspective

Smitha Rajagopal, Katiganere Siddaramappa Hareesha and Poornima Panduranga Kundapur

Department of Computer Applications, MIT, Manipal, India

Article history
Received: 22-06-2016
Revised: 10-01-2017
Accepted: 15-05-2017

Corresponding Author:
Poornima Panduranga Kundapur
Associate Professor,
Department of Computer Applications, MIT, Manipal, India
Email: poornima.girish@manipal.edu

Abstract: Pair programming is one of the widely used practices of Extreme Programming (XP). XP is a software development process which intends to enhance quality of software code in order to cater to the ever increasing demands of customers looking for IT solutions. Pair programming promotes team building, raises confidence among individuals and eventually results in richer and a better codebase. As an attempt to compare solo and pair programming, group of post graduate students were assigned the task to implement four out of 23 design patterns (pertaining to real world scenarios) in their lab sessions. In this study, a systematic investigation on pairing by contemplating a pair programming scenario from a design pattern perspective has been presented. Results were obtained using JavaNCSS tool by considering software code metrics which indicated that pair programming can be beneficial in a scholastic framework.

Keywords: Extreme Programming, Maintainability Index, Cyclomatic Complexity, Data Abstraction Coupling, JavaNCSS, Facade, Observer, Mediator

Introduction

Pair programming is a software development technique in which two individuals collaborate and work at the same workstation as a pair. Typically, in a pair programming scenario, individuals play two crucial roles as that of driver and navigator. Driver codes, navigator observes. They swap their roles quite often and are termed as continuous brainstorming partners. The role essayed by both driver and navigator is pivotal.

The idea of proposing a pair programming pedagogy is greatly influenced by the immense popularity it has gained over recent years in a software industry framework. With critical deadlines to meet, customer demands to be fulfilled and a constant desire to outshine their competitors, software professionals adopt pair programming in every possible situation (Lewis, 2011; Dogs and Klimmer, 2004).

This pair programming activity was conducted for post graduate students of Computer Applications (a three year course spanning across six semesters). These students, who would be future professionals, need to get accustomed to the functioning within the conventional software industry. In this context, an attempt was made to apply pair programming approach in a scholastic framework augmented by a methodical study as explained in the following segment. This paper explores the essence of pair programming in a scholastic framework by emphasizing upon quantitative evidence in terms of code metric assessment using a tool called Non Commented Source code Statements (JavaNCSS) and comparing its results with solo programming. Software code metrics like: Maintainability Index (MI), Cyclomatic Complexity (CC) and Data Abstraction Coupling (DAC) were considered. This work describes the actual conduction of pair programming by considering three different lab courses (Service Oriented Architecture, Free and open source and Design patterns) at postgraduate level in order to implement this pedagogy.

Furthermore, this paper aims to investigate pair programming paradigm from a design pattern viewpoint wherein complex codebases of design patterns like Flyweight, Facade and Mediator were examined. Design patterns are well-proven solutions for solving specific problems and its advantage being programming language independent which certainly, in most cases leads to more flexible, reusable and maintainable codebase.

Literature Review

Programming was considered a solitary activity till Kent Beck introduced Extreme Programming (XP) and listed pair programming as one of its twelve

Science Publications

© 2017 Smitha Rajagopal, Katiganere Siddaramappa Hareesha and Poornima Panduranga Kundapur. This open access article is distributed under a Creative Commons Attribution (CC-BY) 3.0 license.

practices mentioned by Lewis and Colleen in their work (Lewis, 2011).

As per the survey conducted by Dogs and Klimmer, the most commonly used methodology was XP (38.6%) followed by Feature Driven Development (FDD at 14.55%), Rational Unified Process (RUP at 1.9%) and Scrum (at 7.2%) (Dogs and Klimmer, 2004).

Williams, a pioneer in agile software development investigated the importance of pair programming. According to her, pair programming is an efficient form of defect removal before it propagates further. Collaborative programming practices like pairing improves job satisfaction amongst professionals and boosts their morale. She made remarkable contribution in the field of pair programming by reiterating its benefits in many of her outstanding publications (Williams, 2000).

Cockburn and Williams conducted a study on the costs and benefits of pair programming. They identified certain significant paths like: Satisfaction, design quality, team building, learning and problem solving which serves as a strong basis to explore pair programming further (Cockburn and Williams, 2001). Apart from the above mentioned significant paths, pairing can be explored by considering few more critical aspects to comprehend it further.

Begal and Nagappan reported the results of a large scale survey deployed at Microsoft Corporation to gain better insights into perception towards pair programming in industry. From their results, it could be implied that 64% of the respondents believe that pairing works well for them (Begel and Nagappan, 2008). There is scope for future research in terms of presenting quantitative results to prove that pair programming may lead to better prospects.

Stapel *et al*. (2010) emphasized upon the relevance of communication structure in pair programming. They derived an inference to measure the intra-pair communication in pair programming. As per their observation, developers improved their programming abilities and did not talk much about code issues. Successful team building skills were exhibited by developers (Stapel *et al*., 2010). Future work can encompass the usage of a tool to evaluate codebase and fathom its results.

Kavitha and Ahmed put forward their research findings whereby they suggested that pair programming could prove to be a useful approach to facilitate knowledge sharing among students. As demonstrated by them, students performed well in their lab examinations and expressed a sense of accomplishment during pair programming tasks (Kavitha and Ahmed, 2015). There may be basic level lab courses which demand a solo programming pedagogy. For certain advanced level lab courses, applying pairing would be suitable. Therefore, a clear demarcation is required in this context to understand when to introduce pair programming in a scholastic framework.

Sajeev and Datta analyzed pair programming behavior of programmers without prior experience in XP. They discussed the significance of certain key factors like: Whether it is better to train a pair by assigning simple tasks or complex tasks, when to enforce team building etc. As mentioned in their work, the results obtained helps in inducting programmers who are not skilled in XP (Sajeev and Datta, 2013). As a matter of fact, the finely crafted work cited above serves as a basis for undertaking pair programming research in scholastic framework. Prospective students can be trained through pair programming pedagogy before they mark their beginning in software industry.

Williams *et al*. (2002) presented anecdotal evidence from industry and statistical evidence from academia to advocate the benefits of pair programming. They stated that knowledge is passed continuously between partners and pairing helps in every phase of software development life cycle. They could interpret that due to human nature, pairs put a positive pressure on each other to deliver the best (Williams *et al*., 2002). Their viewpoints can be manifested further by deriving quantitative results which could make pair programming research more substantial.

Through empirical research, Vanhanen and Lassenius illustrated that pair programming affects project attributes like: Project productivity, design quality, effort estimation and knowledge transfer within the team (Vanhanen and Lassenius, 2005). Similarly, from an academic perspective, there is an absolute need to evaluate the above mentioned attributes and determine exact ways to capitalize upon the same.

Bernardo and Rafael elucidated the importance of the state of the art Distributed Pair Programming (DPP) from the teaching perspective. They put across a novel practice which involved combining pair programming with geographically distributed team members. It was reported that DPP promotes work and communication between teams (Estácio and Prikladnicki, 2015). The quantitative analysis performed by them can be extended further to get better insights on software code metrics and its comparison with solo programming.

Plonka *et al*. (2015) carried out a systematic inquiry on pair programming and discussed the manner in which pairing influences the strategies, challenges and benefits of driver and navigator. Their work urged developers to utilize the maximum benefits from pairing sessions by throwing more light on expert-novice constellations (Plonka *et al*., 2015). Two other possible combinations could be expert-expert and novice-novice. The selection of a pairing combination entirely depends upon the purpose as to whether it is being applied for knowledge

transfer (expert-novice), assessing the skills of individuals (novice-novice) or for accomplishing the best deliverables (expert-expert).

Coman *et al*. (2008) opined that pair programming is a formalization and enhancement of naturally occurring interactions between developers. They differentiated between experimental and observational studies on pair programming. From this study, it can be inferred that the scenario in which pair programming is advocated plays a vital role for it to be widely accepted and encouraged (Coman *et al*., 2008). From an academic perspective, it should be implemented only when there is a demand for enhancing the quality of codebase. Pair programming reinforces the idea of creating an enriching learning environment by mutual exchange of intellectual thoughts.

Some of the key advantages of pair programming could be perceived from the above mentioned literature. Additionally, many researchers have also focused upon the kind of impact pair programming creates on individuals who apply it (di Bella *et al*., 2013; Turk *et al*., 2014; Hanks *et al*., 2011; Nawahdah *et al*., 2015).

Methodology

As mentioned in the previous section I, pair programming task was introduced for post graduate students of Computer Applications in their 5th semester. This particular group of students was considered because they would be starting their software career soon. Thus they needed a practical understanding of pairing. The primary focus of this task was to compare it with solo programming which was adopted earlier for the previous batch of students but was not successful in enhancing the quality of codebase. A sincere attempt was made by the guiding faculty members to implement this pedagogy for all the lab courses of 5th semester namely: Design patterns, Free and open source software and Service Oriented Architecture (SOA). The above mentioned lab courses had theory elements to them which were taught separately by the guiding faculty members in theory sessions. Pair programming activity involved the execution of the following sequence of steps:

- Creation of possible combinations of pairs
- Conducting 20 lab sessions to complete the lab cycle of prescribed programming courses
- Each student essayed the role of a driver and navigator ten times respectively during these lab sessions. Whenever a student happened to be a driver, he/she coded. His/her partner (navigator) observed. Only driver had control over keyboard and mouse. During the next lab session, they swapped their roles
- Deriving the list of parameters which would help in examining the different aspects of pair programming approach
- Analyzing the outcome of pairing task through a dichotomous questionnaire
- Reporting the results and eventually comparing it with previously adopted solo programming

The following research questions were formulated for which answers are provided in forthcoming section:

RQ1: Which parameters are taken into consideration to ascertain that pair programming is better than solo programming?
RQ2: What are the possible combinations for creating a pair?
RQ3: How do you measure the effectiveness of pair programming strategy?

Since pair programming pedagogy was never applied in the past, seeking permission from the concerned authorities was mandatory to proceed further. It is depicted in Fig. 1.

A student who is considerably good in programming was asked to team up with a student who has average programming skills for Scholar-Naive combination (Lui and Chan, 2006).

Two individuals with sound knowledge of programming collaborate for Scholar-Scholar combination (Lui and Chan, 2006).

Naive-naive combination doesn't yield any benefits in a scholastic framework. Thus it is not dealt with in this context (Lui and Chan, 2006).

Structural and Behavioural Design Patterns aid in Loose Coupling

Design patterns form an integral part of software development and reuse. Students are required to grasp the meaning and discern the usage of design patterns prescribed in their curriculum.

Design patterns can be best explained by relating the 23 patterns to case studies. In this view, students were instructed to take up certain case studies which should be intriguing in the real world context.

Structural design patterns are composed of classes and objects to form larger, complex structures. During pair programming task, all the structural design patterns were studied, analyzed and implemented by students. Among the seven structural design patterns, two codebase were slightly complex than the rest. They are: Flyweight and Facade.

The flyweight pattern describes how to share objects. Each flyweight object has two parts: Intrinsic and extrinsic. The flyweight pattern can be used in conjunction with other objects to handle different applications. The object with intrinsic state is called the flyweight object (Gamma *et al*., 1994).

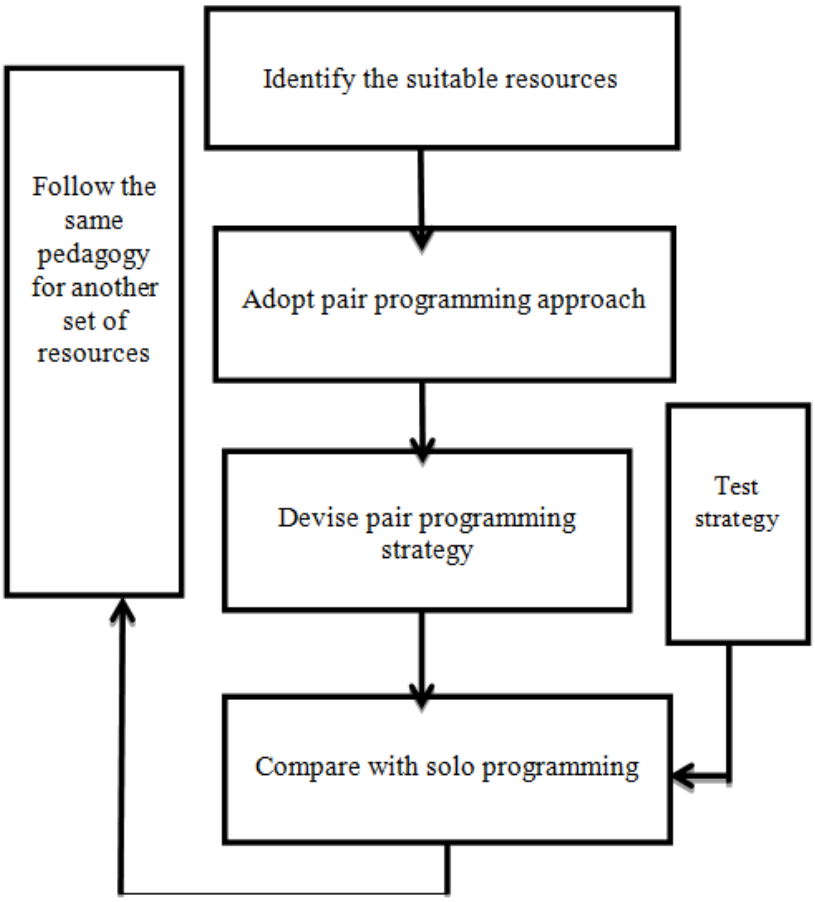

Fig. 1. Overall process of pairing

The best use of flyweight could be explained by considering an example of war strategy simulation game. This application involves the tracking of each and every unit on the battlefield. Large amount of memory is required to hold the details of individual objects.

Flyweight pattern conserves memory space by sharing single copy of intrinsic state across all objects. Flyweights are stored in factory's repository. The usage of flyweight seems to be economical wherein it can be used to decrease memory footprint and improve performance.

A simple example can be visualized for a flyweight. There may be a huge collection of objects used to draw lines. A flyweight would create one line object for each color: Green and red. If there is a drawing with 100 red and 400 green lines, only two lines are instantiated instead of 500 lines.

A facade pattern is known to provide a simple interface to a complex body of code. Facade reduces the dependencies between sub systems. The idea behind facade pattern is to hide the complexities of subsystems from the client. Facade promotes loose coupling.

As depicted in Fig. 2 and 3, facade defines an interface that makes sub systems easier to use. All the client has to do is to use facade as a first point of access to interact with any of the subsystems as per their choice (Gamma *et al*., 1994).

As part of their pairing task, a general E-commerce problem scenario was programmed by a student pair.

The series of steps are mentioned below:

- Create a SmartphoneShopee interface
- Create a Smartphone class which will implement SmartphoneShopee interface
- Create three classes that will implement SmartphoneShopee interface: Asus, Samsung and Lenovo
- Create a concrete class called Retailer
- Create a CustomerFacade class who purchases any of the three smartphones from SmartphoneShopee through Retailer

The idea here is to perceive the fact that CustomerFacade is a class which uses Retailer as the interface to purchase smartphones of their choice like: Samsung, Asus or Lenovo.

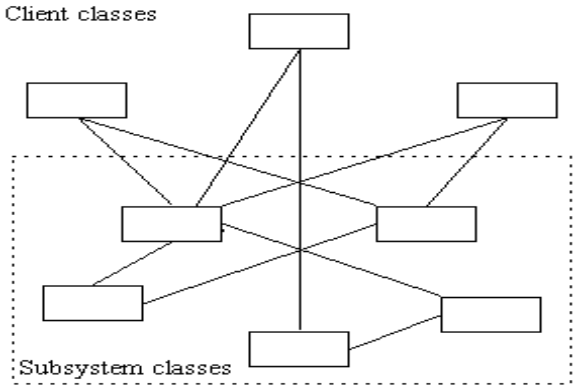

Fig. 2. Interface without a facade (http://www.cs.unc.edu/~stotts/GOF/hires/pat4efso.htm)

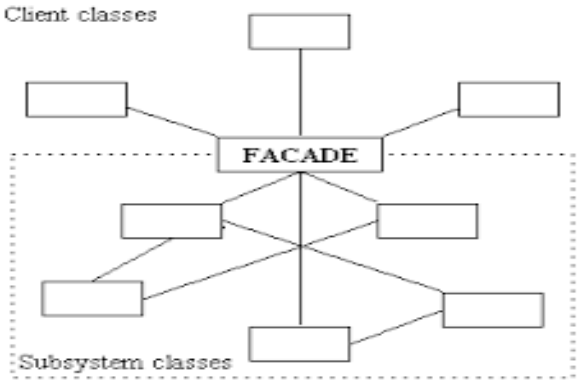

Fig. 3. Interface with a façade (http://www.cs.unc.edu/~stotts/GOF/hires/pat4efso.htm)

A small portion of the code segment is given below:

```
public class Asus implements SmartphoneShopee {
      @Override
      public void model () {
      System.out.println("Asus Zenphone6");
      }
      @Override
      public void price() {
      System.out.println("Rs 17,000");
      }
    };
public class Retailer {
     private SmartphoneShopee Asus;
   private SmartphoneShopee Samsung;
    private SmartphoneShopee Lenovo;
 public Retailer(){
   zenphone = new Asus();
   samsungj7 = new Samsung();
   lenovovibe = new Lenovo();
   }
    public void asussales(){
       asus.modelNo();
       asus.price(); }
 // Samsung sales here
{ }
  // Lenovo sales here{}};
```

From the above code fragment, it can be noted that Facade pattern contributes to a decrease in the lines of code. The code looks modularized too.

Loose coupling is a coupling mechanism in which two components are linked in such a way that the services that they offer are not dependent on each other. Facade pattern promotes loose coupling by emphasizing on functionality rather than internal details. The design will not get affected upon extending the subsystem.

Observer pattern contributes a lot towards greater code reuse and better maintainability. A clear demarcation between UI and business logic is necessary. It is quite common for UI requirements to change without prior notification similar to how business requirements change without regard to the UI. So, the separation between both makes logical sense and observer pattern is best suited for such object oriented software development (Lui and Chan, 2006).

The primary advantage of design patterns lies in the fact that they encourage reusability. Observer pattern is a behavioral pattern which maps one-to-many-dependency between objects. Whenever there is any change in any object's state, all its dependencies are notified and updated automatically. A typical observer pattern is shown below in Fig. 4.

A student pair implemented an observer pattern as described below. The outline of the case study is as follows. Kroger health mart is an online drug shop which specializes in manufacturing, marketing and ensuring abundant availability of different drugs for mass consumption. Three patients are in need of Morpheme mind plus capsules, a type of vitamin supplements. Currently they are out of stock.

All three users clicked on "notify me" button to receive alerts as soon as the capsules are available. In this scenario, all the users who clicked "notify me" are observers. The subject of their observation is vitamin supplements. So, Morpheme mind plus is observable.

A small portion of the code implemented through observer pattern is mentioned below:

```
public class User implements Observer
{
   private Observable obs = null;
   public User( Observable obs)
   {
     this. obs=obs;
   }
@ override
```

```
   public void update()
{
   purchaseMorpheme();
   unsubscribe();
}
   public void purchaseMorpheme()
   {

 System.out.println("Bought Morpheme");
}
   public void unsubscribe()
   {
     obs.removeObserver(this);
   }
};
```

User implements Observer. Morpheme can be purchased as and when it is available for sale. Users can unsubscribe from notify me depending upon their interest.

Thus, it is possible to state that observer pattern allows loose coupling by separating the logic between observer and observable to make them independent of each other. Any change done to observable will not impact observer.

The next segment focuses on another behavioral pattern namely Mediator. As shown in Fig. 5 and 6, it is possible to compare and contrast how objects would interact in the absence of mediator pattern and in its presence respectively.

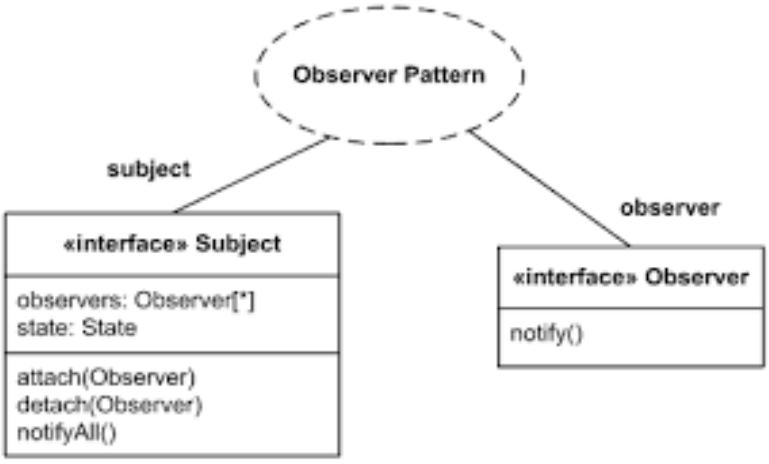

Fig. 4. An observer pattern (Gamma *et al*., 1994)

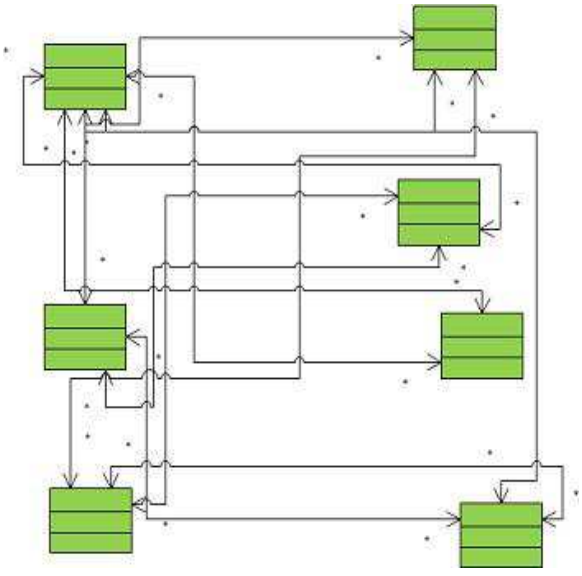

Fig. 5. Tight coupling without Mediator (http://javapapers.com/design-patterns/mediator-design-pattern/)

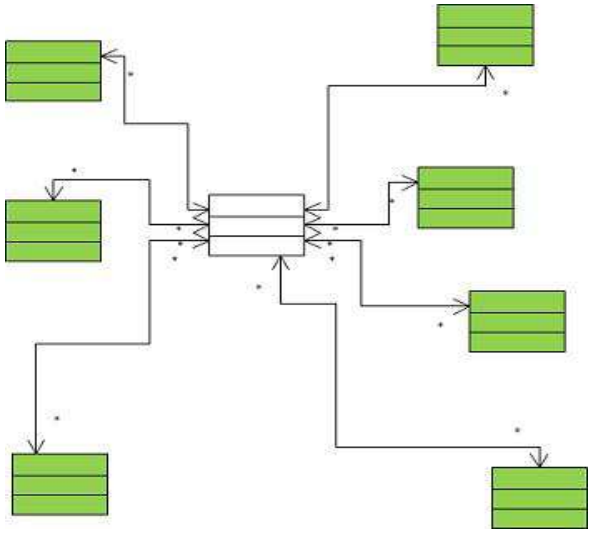

Fig. 6. Loose coupling with Mediator (http://javapapers.com/design-patterns/mediator-design-pattern/)

As part of the case study undertaken by a student pair for mediator pattern, a real time scenario was examined. Sky scanner is a B2C travel aggregator which offers a convenient platform for travellers/tourists to search compare and buy flight tickets, book hotels and hire cabs. Hotels, Airlines and travel agents have their own share of responsibilities and there is absolutely no need to interact with each other. Such travel aggregators act like a mediator to ensure the best deal for tourists. Mediator pattern is used to simplify the communication between many objects. The idea is to have a central point of access for many communicating parties. Unnecessary communication between peer objects can be eliminated through mediator pattern.

A code snippet implemented by a student pair is given below:

```
public interface Skyscanner
  {
     void deal ();
  }
public class Airline implements Skyscanner
  {
    @ override
    public void deal()
    {
      System.out.println("Airline:: deal()");
    }
  };
public class Hotel implements Skyscanner
{
  @ override
   public void deal()
  {
    System.out.println("Hotel:: deal()");
  }
};
public class Taxi implements Skyscanner
{
  @ override
   public void deal()
   {
    System.out.println("Taxi:: deal()");
  }
};
```

Colleague classes constitute an important part of Mediator pattern wherein they interact with it in situations in which they could have interacted with each other. E.g.: A tourist will get the best deal by not explicitly interacting with airline, hotels or travel agencies. In the above example, all the three deals are handled by the travel aggregator independently by essaying the role of mediator.

Impact of Code Metrics on Pairing

Normally, software code metrics are considered as the most pertinent tools for improving the quality of codebase. There is a need to have appropriate standards in place to differentiate between good, average and bad code. Code metrics indicate as to what extent certain desired software characteristics are present, which ones may be deficient and how it could be improved. Software can be maintained reasonably well if due importance is given to software code metrics evaluation and assessment.

During this pair programming effort, three predominant code metrics were evaluated in order to gain a better understanding of the developed codebase:

- Maintainability Index (MI): This software metric indicates how maintainable the source code is (Butler *et al*., 2010). MI can be calculated using (1):

$$MI = 171 - 5.2 * \log(V) \\ -0.23 * (G) - 16.2 * \log(LOC) \quad (1)$$

  V refers to Halstead volume, G refers to cyclomatic complexity and LOC refers to lines of code.
- Cyclomatic Complexity (CC): This metric indicates complexity of the software codebase. It is a quantitative measure of the independent paths in a program's source code (Butler *et al*., 2010)
  McCabe's Cyclomatic metric, $V(G)$ of a graph "G" with "n" vertices and "e" edges is given by the following formula shown in (2):

$$V(G) = e - n + 2 \quad (2)$$

- Data Abstraction Coupling (DAC): This metric measures the number of instantiations of other classes within a single class (Elish and Alshayeb, 2011)
  DAC = number of ADTs defined in a class. ADT's refer to abstract data types.

As a part of forthcoming exercise on pair programming task, students were advised to devise wider test coverage and concentrate on afferent and efferent coupling. Afferent coupling (C_a) is defined as a measure of the number of classes and interfaces from other packages that depend upon classes in the analyzed package. Efferent coupling (C_e) is a measure of outgoing dependencies or the number of classes or interfaces inside a package that depends on other types (Sato *et al*., 2007).

In general, all software artifacts have a certain degree of instability. Based on C_a and C_e it is possible to

calculate instability associated with a software artifact as shown in (3):

$$Instability = C_e / (C_e + C_a) \quad (3)$$

Consider the following code fragment which is a part of Mediator codebase.

```
Class Traveller
{
    Airline airline;
    Hotel hotel;
    Taxi taxi;
};
```

This class would have a high C_e as it depends on three types: Airline, Hotel and Taxi. C_a would depend upon the number of classes that depend upon these three classes. A software artifact is stable when Instability is close to zero. If instability is close to one, the software artifact is considered unstable.

Within the stipulated time, student pairs could understand the impact of C_a and C_e theoretically only.

Data Collection and Key Parameters

Data was collected from 80 students and 6 guiding faculty members twice. Firstly, they were asked to answer a questionnaire which was designed and validated in consultation with all guiding faculty members. From these results, shortcomings of solo programming could be determined. Secondly, to establish the authenticity of pair programming, a dichotomous questionnaire was formulated and corresponding data was collected. The questionnaire highlighted the interesting aspects of pair programming approach as mentioned below in Table 1.

Table 1. List of parameters

Sl. no	Parameter
1.	Discipline
2.	Resilient flow
3.	Interruptions
4.	Collective code ownership
5.	Higher Design quality
6.	Decrease in LOC (Lines of code)
7.	Performance
8.	Bug density
9.	Code complexity
10.	Coding skills
11.	Debugging skills
12.	Use of new tools
13.	Shortens program development time
14.	Exploring test cases to analyze programs
15.	Satisfaction

To assess the impact of pair programming on students, a questionnaire was developed which highlighted the key parameters as explained below:

- Discipline: It was observed that students maintained lot more discipline during lab sessions as compared to programming solo.
- Resilient flow: Whenever a program encountered an exceptional condition, students found it easier to deal with it in the presence of a partner.
- Interruptions: Students requested less number of breaks during lab sessions during the execution of pair programming activity.
- Collective code ownership: The onus was on both individuals whenever code worked or failed.
- Higher Design quality: Students devoted substantial amount of time towards design process (which was lacking during solo programming) when they were asked to program in pairs.
- Decrease in LOC (Lines of code): Quite often, readability suffers due to numerous lines of code. When two students work in pairs, there is a scope to decrease the lines of code by removing redundant portions of codebase.
- Performance: Our hypothesis clearly shows that students performed better during pair programming activity.
- Bug density: Bug density refers to a measure used to understand the ability of developers to err. In case of pairing, it was noticed that a student erred lesser as he was always coupled with a partner.
- Code complexity: Due to mutual exchange of intellectual thoughts, students developed codebase which were relatively easier to understand and eventually led to reasonable reduction in code complexity.
- Coding skills: As compared to solo programming, there was a remarkable improvement in students' coding skills.
- Debugging skills: Fixing bugs in a program is a far more difficult task to do than coding a new program. Students got ample scope to debug their partner's code.
- Use of new tools: Students were interested to explore new tools and also search for different free and open source softwares which was a clear indication of their inquisitiveness.
- Shortens program development time: The program development time taken by the class (which worked in pairs) to make programs fully functional was lesser.
- Exploring test cases to analyze programs: Students went a step ahead towards determining various test cases to check the correctness of programs.

- Satisfaction: Quite apparently, there was a greater amount of satisfaction found in both students as well as faculty members.

A sample dichotomous questionnaire is shown below in which some questions were specific to guiding faculty members. They are as follows:

- Was there a higher level of discipline exhibited by current batch of students who did pair programming than previous batch who programmed solo? Yes/No
- Do you feel that students requested for less number of breaks during lab sessions? (Duration of each lab session was 3 h). Yes/No
- Do you think that students devoted ample time to understand design aspect of all case studies?
- Yes/ No
- As a guiding faculty member of this lab course, do you feel that readability of codebase has been better? Yes/No
- Do you think there is a decrease in the erring facet of student pairs? Yes/No
- Do you think codebase developed by this batch of students is less complex? Yes/No
- Is there an improvement in students' coding and debugging skills since this class worked in pairs? Yes/No
- Were students interested in exploring new tools and softwares? Yes/No
- Did you notice that students showed an interest towards exploring test cases? Yes/No
- As a guiding faculty member, were you satisfied with the overall performance of student pairs? Yes/No

The following dichotomous questions were answered by students:

- Were you able to handle exceptional conditions in programs in a better way in presence of your partner? Yes/No
- Did you feel that onus of codebase was equally shared with your partner too? Yes/No
- Did you give ample importance to comprehend the design aspect of all case studies? Yes/No
- Were you able to remove redundant piece of code since you worked with your partner? Yes/No
- Do you feel your coding and debugging skills have improved after this pair programming activity? Yes/No
- Did you use new tools and softwares apart from the ones prescribed for you? Yes/No
- Did you feel the need to explore test cases to check the correctness of your programs? Yes/No
- Was there a decrease in your erring facet during pair programming? Yes/No
- Were you satisfied with this pair programming task? Yes/No
- Do you feel that there has been a considerable decrease in your program development time since you worked in pairs in this semester as compared to the previous semester when you programmed solo? Yes/No

Results and Implications

The parameter score was calculated by the formula as given below:

$$Score = \frac{Number\ of\ positive\ responses}{Total\ number\ of\ participants} \times 100 \quad (2)$$

where Score denotes the score which was calculated individually for each parameter.

From the results displayed in Fig. 7 and 8, it is a clear indication of the fact that pair programming has performed better than solo programming. As an add-on, we present results obtained from Non Commented Source code Statements (JavaNCSS) which only enhances our study. JavaNcss is a command line tool used to perform quantitative analysis of code written in java. It can be conveniently used at the command line (Tomas *et al*., 2013). The reason behind using this particular tool is that it requires code written in Java and students preferred to use Java as the programming language to implement design pattern case studies. Results are tabulated in Table 2-4 which further augments the pair programming outcome.

The lab course titled Design patterns comprised of 23 design patterns categorized as creational, structural and behavioral. An implementation of these patterns was done in Java. Only five complex codebase were reviewed and the values of predominant metrics were calculated.

As a result of the experiments conducted, following are the answers to the research questions formulated in the previous section III. RQ1: To ascertain that pair programming is better than solo programming, fifteen parameters were considered as explained in previous section VI. RQ2: There are three possible combinations of pairs. In this context, two combinations of pairs were considered. They are: (1) Scholar-Naive (2) Scholar-Scholar. RQ3: To measure the effectiveness of pair programming, JavaNCSS tool was used as explained above along with code metrics.

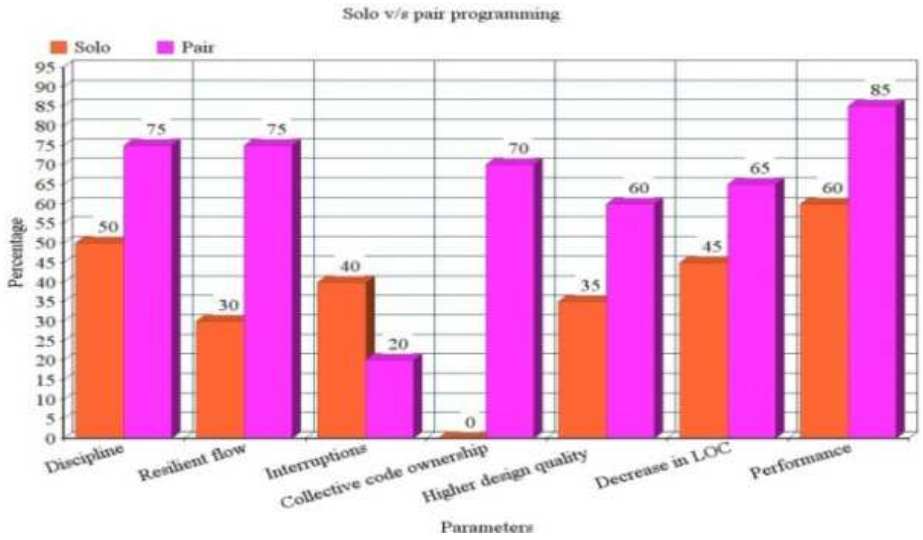

Fig. 7. Solo v/s pair programming (1/2)

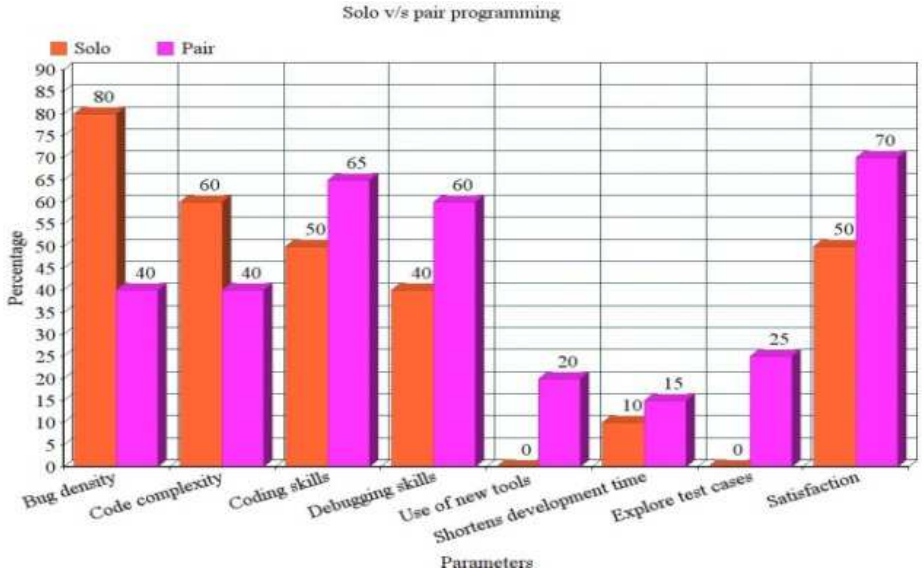

Fig. 8. Solo v/s pair programming (1/2)

Table 2. MI solo v/s pair

Pattern	MI(Solo)	MI(Pair)
Flyweight	60.25	80.23
Facade	45.56	63.25
Observer	39.26	55.80
Iterator	43.60	47.99
Mediator	67.88	79.00

Table 3. CC solo v/s pair

Pattern	CC(Solo)	CC(Pair)
Flyweight	25	18
Facade	30	22
Observer	27	21
Iterator	32	25
Mediator	30	24

Table 4. DAC solo v/s pair

Pattern	DAC(Solo)	DAC(Pair)
Flyweight	23	16
Facade	19	15
Observer	22	17
Iterator	14	10
Mediator	16	12

As observed in this study, advantages of pair programming pedagogy can be summarized as follows:

- Pair programming pedagogy was fairly successful
- Produces better quality of code

- Students can tackle programming tasks in different ways

Conclusion

In this study, a sincere attempt was made to establish that pair programming is better than solo programming for certain advanced level lab courses in postgraduate studies. All the above mentioned parameters were chosen after a systematic analysis of pair programming paradigm. The quantitative analysis discussed in this study shows that pair programming has a great potential of generating desirable software code. Key advantages identified in the study on pair programming include better quality of code, tackling programming exercises differently and a higher success rate than solo programming. Overall, pair programming can be termed as a noteworthy XP practice which can be definitely recommended for postgraduate students.

As part of future work for student pairs, cognitive complexity, a software metric used to test the quality of code will be introduced. Cognitive complexity is a far more sophisticated metric than cyclomatic complexity to track control flow in codebase.

Acknowledgment

The authors would like to express their gratitude to 5th semester MCA students and faculty members of Manipal University for participating in pair programming activity.

Author's Contributions

Smitha. R, preparation of the manuscript. Hareesha. K.S responsible for reviewing the manuscript and offering valuable suggestions. Poornima Panduranga Kundapur contributed towards organization of the manuscript and offered meaningful inputs.

Ethics

The authors confirm that this manuscript has not been published elsewhere and that no ethical issues are involved.

References

Begel, A. and N. Nagappan, 2008. Pair programming: What's in it for me? Proceedings of the 2nd ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, Oct. 09-10, ACM, New York, pp: 120-128. DOI: 10.1145/1414004.1414026

Butler, S., M. Wermelinger, Y. Yu and H. Sharp, 2010. Exploring the influence of identifier names on code quality: An empirical study. Proceedings of the 14th European Conference on Software Maintenance and Reengineering, Mar. 15-18, IEEE Xplore Press, Madrid, pp: 156-165. DOI: 10.1109/CSMR.2010.27

Cockburn, A. and L. Williams, 2001. The Costs and Benefits of Pair Programming. Extreme Programming Examined, Succi, G. and M. Marchesi (Eds.), Addison-Wesley, pp: 223-243.

Coman, I.D., A. Sillitti and G. Succi, 2008. Investigating the usefulness of pair-programming in a mature agile team. Proceedings of the Agile Processes in Software Engineering and Extreme Programming, Jun. 10-14, Springer Berlin Heidelberg, pp: 127-136. DOI: 10.1007/978-3-540-68255-4_13

di Bella, E., I. Fronza, N. Phaphoom, A. Sillitti and G. Succi *et al*., 2013. Pair programming and software defects--a large, industrial case study. IEEE Trans. Software Eng., 39: 127-136. DOI: 10.1109/TSE.2012.68

Dogs, C. and T. Klimmer, 2004. An evaluation of the usage of agile core practices. MSc Thesis, Blekinge Institute of Technology, Sweden.

Elish, K.O. and M. Alshayeb, 2011. A classification of refactoring methods based on software quality attributes. Arabian J. Sci. Eng., 3: 1253-1267. DOI: 10.1007/s13369-011-0117-x

Estácio, B.J.D.S. and R. Prikladnicki, 2015. Distributed pair programming: A systematic literature review. Inform. Software Technol., 63: 1-10. DOI: 10.1016/j.infsof.2015.02.011

Gamma, E., R. Helm, R. Johnson and J. Vlissides, 1994. Design Patterns: Elements of Reusable Object-Oriented Software. 1st Edn., Addison-Wesley, ISBN-10: 0201633612, pp: 395.

Hanks, B., S. Fitzgerald, R. McCauley, L. Murphy and C. Zander, 2011. Pair programming in education: A literature review. Comput. Sci. Educ., 21: 127-136. DOI: 10.1080/08993408.2011.579808

http://javapapers.com/design-patterns/mediator-design-pattern/

http://www.cs.unc.edu/~stotts/GOF/hires/pat4efso.htm

Kavitha, R.K. and M.S.I. Ahmed., 2015. Knowledge sharing through pair programming in learning environments: An empirical study. Educ. Inform. Technol., 20: 319-333. DOI: 10.1007/s10639-013-9285-5

Lewis, C.M., 2011. Is pair programming more effective than other forms of collaboration for young students? Comput. Sci. Educ., 21: 127-136. DOI: 10.1080/08993408.2011.579805

Lui, K.M. and K.C.C. Chan, 2006. Pair programming productivity: Novice-novice Vs. expert-expert. Int. J. Human-Comput. Stud., 64: 127-136. DOI: 10.1016/j.ijhcs.2006.04.010

Nawahdah, M., D. Taji and T. Inoue, 2015. Collaboration leads to success: A study of the effects of using pair-programming teaching technique on student performance in a Middle Eastern society. Proceedings of the IEEE International Conference on Teaching, Assessment and Learning for Engineering, Dec. 10-12, IEEE Xplore Press, pp: 16-22. DOI: 10.1109/TALE.2015.7386009

Plonka, L., H. Sharp, J. van der Linden and Y. Dittrich, 2015. Knowledge transfer in pair programming: An in-depth analysis. Int. J. Human-Comput. Stud., 73: 66-78. DOI: 10.1016/j.ijhcs.2014.09.001

Sajeev, A.S.M. and S. Datta, 2013. Introducing Programmers to Pair Programming: A Controlled Experiment. 1st Edn., Springer Berlin Heidelberg.

Sato, D., A. Goldman and F. Kon, 2007. Tracking the evolution of object-oriented quality metrics on agile projects. Proceedings of the 8th International Conference on Agile Processes in Software Engineering and Extreme Programming, Jun. 18-22, Springer Berlin Heidelberg, pp: 84-92. DOI: 10.1007/978-3-540-73101-6_12

Stapel, K., E. Knauss, K. Schneider and M. Becker, 2010. Towards understanding communication structure in pair programming. Proceedings of the International Conference on Agile Software Development, (ASD' 10), Springer, Berlin Heidelberg, pp: 117-131. DOI: 10.1007/978-3-642-13054-0_9

Tomas, P., M.J. Escalona and M. Mejias, 2013. Open source tools for measuring the internal quality of Java software products. A survey. Comput. Standards Interfaces, 36: 244-255. DOI: 10.1016/j.csi.2013.08.006

Turk, D., R. France and B. Rumpe, 2014. Assumptions underlying agile software development processes. arXiv preprint arXiv: 1409.

Vanhanen, J. and C. Lassenius, 2005. Effects of pair programming at the development team level: An experiment. International Symposium on Empirical Software Engineering, Nov. 17-18, IEEE Xplore Press, pp: 10-10. DOI: 10.1109/ISESE.2005.1541842

Williams, L., 2000. The collaborative software process. PhD Thesis, University of Utah, USA.

Williams, L., E. Wiebe, K. Yang, M. Ferzli and C. Miller, 2002. In support of pair programming in the introductory computer science course. Comput. Sci. Educ., 12: 197-212. DOI: 10.1076/csed.12.3.197.8618