The Integrity of Source Code Commenting : Benchmark Dataset and Empirical Analysis

Maksuda Islam (maksudalima321@gmail.com)
Islamic University of Technology
Mohammad Safayat Hossen
Islamic University of Technology
Ahsanul Haque
Islamic University of Technology
Md. Nazmul Haque
Islamic University of Technology
Lutfun Nahar Lota
Islamic University of Technology

Research Article

Keywords: Code-Comment Consistency, Coherence, Bilingual Evaluation Understudy, Krippendorff's Alpha

Posted Date: November 3rd, 2022

DOI: https://doi.org/10.21203/rs.3.rs-2207069/v1

The Integrity of Source Code Commenting: Benchmark Dataset and Empirical Analysis

Maksuda Islam[1*†], Mohammad Safayat Hossen[1†], Ahsanul Haque[1†], Md. Nazmul Haque[1] and Lutfun Nahar Lota[1]

[1*]Computer Science and Engineering Department, Islamic University of Technology, Gazipur, Dhaka, Bangladesh.

*Corresponding author(s). E-mail(s): maksudalima321@gmail.com;
Contributing authors: safayathossen@iut-dhaka.edu; Hoque.talha@gmail.com; nazmul.haque@iut-dhaka.edu; lota@iut-dhaka.edu;
[†]These authors contributed equally to this work.

Abstract

Code comments are a vital software feature for program cognition & software maintainability. For a long time, researchers have been trying to find ways to ensure the consistency of code-comment. While doing that, two of the raised problems have been dataset scarcity and language dependency. To address both problems in this paper, we worked on a dataset creation made using C# projects; there are no annotated datasets yet on C#. 9,310 code-comment pairs of different C# projects were extracted from a data pool. 4,922 code-comment pairs were annotated after removing NULL, constructor, and variable. Both method-comment and class-comment were considered in this study. We employed two evaluation metrics for the dataset, one is Krippendorff's Alpha which showed 95.67% similarity among the rating of 3 annotators for all the pairs & other is Bilingual Evaluation Understudy (BLEU) to validate our human-curated dataset. A modified model from a previous study is also proposed, which obtained 96.2% using the performance metric AUC-ROC after fitting the model to our annotated 4,922 code-comment pairs.

Keywords: Code-Comment Consistency, Coherence, Bilingual Evaluation Understudy, Krippendorff's Alpha

1 Introduction

Comments on codes are necessary for software systems. It helps developers to understand the code for modification or reuse. Commenting on code is used not only to give the design's implementation details but also to attach the intent behind the design. Compared to source code, comments are more straightforward, descriptive, and simple to understand [1, 2]. Comments are the primary documentation source for a software system. Comments supplied to the code hugely help to understand the source code throughout the development and maintenance phases, [2] therefore, reducing maintenance costs. It provides developers with a significant resource to maintain and improve software applications.

For regular coding, debugging, and maintenance tasks, developers spend extensive time navigating and exploring existing code [3, 4]. If they could understand the code itself by reading the description which is largely known as the "comment" written with the code, this extensive amount of spent time could be saved. One way to eliminate all these blockages could be using the auto comment generator. Generating auto comment also comes with a problem. Since the software is an ever-evolving process, the system is constantly evaluated. Due to multiple programmers working on the same codebase, tracking the changes and descriptions of written functions and classes are often challenging. Developers usually comment with a code fragment that provides insightful information about a software system for managing software evolution and maintenance. Nowadays, there are many automation tools for comment generation, but comments are more legible when written by humans because they are written in natural language [1, 2]. We need comments to enhance the understandability of the source code. The reader gets misled when there are inconsistent comments in the source code. This can confuse and do more harm than good.

Researchers have been working to mitigate the problem of inconsistent code and comments for many years. Along with other different solutions, one crucial solution that came forward was computing whether code and comment are consistent with each other at any point in the software lifecycle. To do so, a well-standard dataset is needed. Although there is an insufficiency of the existing dataset, several works of literature have proposed various datasets. However, most of the datasets are made with projects written using Java Programming Language. Nowadays, along with the Java Programming language, C# is becoming popular because of its enriched libraries, third-party software, a large number of communities, and so on. According to the recent survey of 2022 [1], C# is the tenth most popular programming language. C# is highly integrated with the .Net framework which has become one of Microsoft's most successful programming languages and is widely adopted by

[1]https://survey.stackoverflow.co/2022/#section-most-loved-dreaded-and-wanted-programming-scripting-and-markup-languages

professional software developers. However, from our thorough investigation of the existing literature, there is no dataset of C# programming language considering the coherence of code and their corresponding comment.

Java, an object-oriented programming language, supports more separation of concerns and good segregation of concepts; as a result, it produces a good number of classes [5]. Most of the previous work on code-comment consistency is done in Java language which does not mention working on class modules. Instead, they worked on method and comment pair only [6–8]. C# is also an object-oriented programming language that requires writing a good number of classes. So, computing the coherence of both class comment and method comment should be considered because both class comment and method comment aid C# program and project understanding.

To solve the aforementioned issues, in this paper, we described the complete process of creating a dataset that contains an annotation of 4922 code-comment pairs of C# projects having both classes and methods. Our study is focused on determining whether the lead comment of a method effectively describes the overall method or not. In the same way, whether the lead comment of a class effectively describes the general intent of the class as well as implementation details, for example, parameters type and returned values, etc. In-line comments are not considered in this work. Therefore, if a lead comment of a method or a lead comment of a class describes the intent of the method or class and its primary implementation detail, then that comment and the source code are coherent. To evaluate the human annotation process, we used two evaluation metrics. Krippendorff's Alpha & BLEU. First, Krippendorff's Alpha assesses the rating similarity among 3 annotators. Second, BLEU checks how much our final combined annotation matches the lexical similarity between code and comment. Our primary contribution to this paper is the proposed set of rules to determine whether or not the lead comment of the class is coherent with it, the annotated benchmark dataset, which holds 4922 code-comment pairs of C# projects.
To list down the contribution of this work is as follows:

- Took the class-comment pair into consideration and proposed a set of rules to annotate them
- Showed the qualitative analysis of the dataset through two evaluation metric
- Annotated class-comment and method-comment pair for C# programming language
- Enhanced topic modeling-based model to automatically classify code-comment consistency
- Discussed possible threats to the work

The paper is structured as follows. In Section 2, we discussed related work, and background studies is presented in Section 3. We discussed about the methodology of our dataset creation in section 4. In section 5, we discussed the model training detail & results. In section 6, we validated the possible threats that could be raised by readers. The conclusion and future of the paper are in section 7. In section 8, the declarations and statements of the paper's authors are mentioned.

2 Related Works

Commenting is the most crucial aspect of making source code comprehensible. Because comments are more direct and detailed, developers like them. More information regarding the implementation of the source code may be found in the comments. Source code changes as program development progresses. However, occasionally developers fail to update comments to reflect progress. Even though the majority of developers understand the need for software documentation, release deadlines, and other time constraints may result in the absence of comments[2]. Consequently, there were contradictory comments. A remark, for example, may contain information that is irrelevant or conflicting with the source code[9]. As a result, the costs associated with reviewing, updating, and testing source code will rise[10, 11]. All of these difficulties contribute to comment gaps and inconsistencies in source code.

Recent research for better software maintenance has yielded techniques for measuring the relationship between code and comment[12] and highlighted the need to maintain consistency between the two. Several methods for identifying ancient document comments have been published[10, 12–14]. Because document comments are well-structured for analysis, they were able to identify out-of-date document comments with high precision and recall[14]. For methods that only looked at block/line comments, the detection was done at the function/method level[10, 12]. For methods that looked at both, the detection was done at the topic level[10, 13].

A study created a technique for spotting obsolete comments during the evolution of code using a machine learning-based approach[15]. To find possible comment changes, use the source code change and the relationship between the code and the comment before and after the source code change. They were able to detect changes in 64 different properties of the software, as well as the state of comments and how they related to the code before and after the changes, using machine learning techniques.

Typically, the developer who writes the code decides whether or not to include comments. How developers comment on code determines both the quality of the comments and the readability of the code. To better understand this term, the practice of adding comments to code in various programming languages is investigated[16]. The practice of commenting on code has evolved over time as a result of the development of natural programming languages and the evolution of ecosystems.

Researchers are seeking a long-term solution for the developers in order to comprehend this inconsistency. A method is based on information retrieval for locating traceability links between source code and free-text documents. In two case studies, probabilistic and vector space information retrieval strategies were used to track C++ source code on manual pages and Java code on functional requirements[17]. In another paper, a method is proposed for detecting code comment inconsistencies in locking and calling mechanisms. They were only interested in suggestions regarding programmers' assumptions and needs[10]. Another paper testing Javadoc's comments was published[11]. The authors analyzed method properties with null values and corresponding exceptions. They utilized Natural Language Processing to identify inconsistencies involving @param tags in Javadoc comments and the null parameter exception statement in codes and comments. Their efforts consist solely of commenting on null references and throwing exceptions.

In 2020, a way to figure out how close comments are to their source code was made using topic modeling[8]. The goal of this research is to find a way to judge the consistency of source code comments that are based on topic modeling. Since comments must match the source code to which they refer, a model was made to judge how consistent they are and, by extension, how good they are. For each of the projects that were looked at, all of the source code and software class comments were taken out. In 2021, another paper on the subject of our research came out[7]. This article describes a way to automatically pull out the most important ideas from a code sample and its accompanying comment and figure out how well they fit together. Using a benchmark dataset of 2,800 pairs of Java code and comments, we showed that our method was better than the current best Support Vector Machine on the vector space model in a number of ways.

In another study, the results of a manual analysis of how well the comments and implementations of 3,636 methods from three Java open-source software systems match up are shown[6]. The evaluations that resulted were compiled into a dataset that was made publicly available thereafter. This paper also tries to find links between coherence and lexical information provided in source code and method lead comments, which is another important contribution.

All the above literature gives us a perfect picture that the problem of inconsistency between code and its comments exists and there is a lot to work on. One of the major drawbacks of all work is that they are done for Java programming language mostly. So, the first problem that should be addressed is the scarcity of datasets in any different programming language other than java. In our work, we aim to address that adequacy and work towards minimizing it.

3 Background Studies

Comments are the foremost source of code documentation. Due to this, their requirement is consistent with the written code. Computing code comment consistency is essential since a written method can be complex enough for another developer to be understood. Complexity may occur for different reasons, for example, there can be dependencies among the functions. The project can be one of the open-source libraries. Our primary focus is to reduce the language dependency of this research topic by providing a labeled dataset for another programming language and by proposing a language-independent model. With that being the goal, a few terminologies and facts need to be addressed first. Those facts & terminologies are discussed in this section.

3.1 Importance of Checking the Consistency of Code-Comments over Generating Consistent Comments

Previously many studies have been conducted on generating consistent comments, one widely known example of that is Javadocs which works as a documentation tool for java language. Generating consistent comments by analyzing the code is indeed very large progress, although one mentionable lacking of it is not considering the evolution process of a software project.

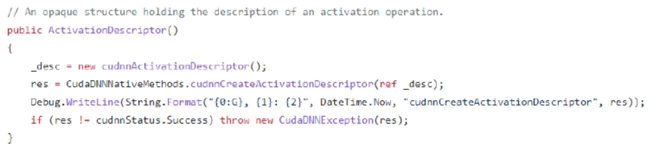
It is a common scenario in the software industry that the comment which was consistent with the previously written code is not inconsistent with the modified code. So, detecting the current state of whether the code is consistent is equally important if not more than generating consistent comments.

3.2 Importance of Topological Order of comment

Comments can be written on top of the code, inside the code, and at the end of the code. Usually, comments written between class and method explain or describe the fragment of that code. Whereas, comments written on top of the code generally describe the whole class or method. So, following the topological order of the comment is essential.

3.3 Lead Comment

Lead comment refers to the corresponding comment which is written just above the code.



```
// An opaque structure holding the description of an activation operation.
public ActivationDescriptor()
{
    _desc = new cudnnActivationDescriptor();
    res = CudaDNNNativeMethods.cudnnCreateActivationDescriptor(ref _desc);
    Debug.WriteLine(String.Format("{0:G}, {1}: {2}", DateTime.Now, "cudnnCreateActivationDescriptor", res));
    if (res != cudnnStatus.Success) throw new CudaDNNException(res);
}
```

Fig. 1: Snippet of method code-comment pair from GitHub of the project ManagedCUDA

In Figure 1, a comment was written on top of the method or class which explains briefly the corresponding code known as the lead comment.

3.4 Coherent and Incoherent

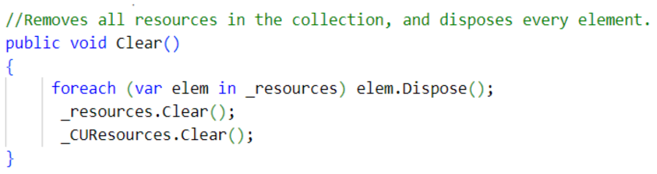
When a code snippet's meaning and respective comment are aligned correctly, the code-comment pair is recognized as Coherent pair. On the contrary, when the meaning of a code snippet and respective comment is not appropriately aligned is labeled as Incoherent paid.

3.5 Bilingual Evaluation Understudy (BLEU)

A value for comparing a candidate sentence of text to one or more reference sentences of text. Complete match outcomes in a score of 1, and complete inconsistency results in a score of 0.

This metric is widely used to evaluate machine translation results [18, 19].BLEU scores have been employed as one of the evaluation metrics in recent source code summarizing research [20, 21]. A function to compare a candidate sentence to one or more reference sentences is offered by NLTK.

A BLEU score compares a tokenized predicted sentence to a tokenized reference sentence. The result is determined by comparing the predicted sentence's average precision to the reference sentence. In our scenario the reference text is code, and the candidate text is a comment. For a better understanding, an example from our dataset is discussed below.



```
//Removes all resources in the collection, and disposes every element.
public void Clear()
{
    foreach (var elem in _resources) elem.Dispose();
    _resources.Clear();
    _CUResources.Clear();
}
```

Fig. 2: Implementation of the method 'Clear' and it's corresponding lead comment

In the code-comment pair shown in Figure 2, the reference text was the code part which is the void method, Clear, implementation and as candidate text, the comment 'Removes all resources in the collection and disposes every element' were used. After applying BLEU on them it gave a score of 0.516593 which means the candidate text(comment) is a 51.6593% percent match of the reference text(method). A fixed threshold value of 51%-100% to consider a pair as a coherent pair, due to that, using the BLEU Score, this pair Figure 2 was found to be a coherent one.

3.6 Topic Modeling

The goal of Topic Modeling is to find topics in a given textual document that represents it as a cluster of words. It does not use tags and training data. Clusters of words are often used together based on the word frequencies and word co-occurrence frequencies in one or more documents.

3.7 Latent Dirichlet Allocation(LDA)

Latent Dirichlet allocation is an often-used algorithm for topic modeling. It is used to classify text in a document to a specific topic. We used LDA on two different corpora of Code and Comments in the work.

4 Methodology of Dataset Creation

In this section, the methodology of the study is briefly discussed. The complete workflow is represented using the figure 3. This section is divided into two subsections, which are 'Data Generation', 4.1, and 'Validation & Evaluation of the Dataset' ,4.2. The data generation subsection is further divided into multiple subsections.

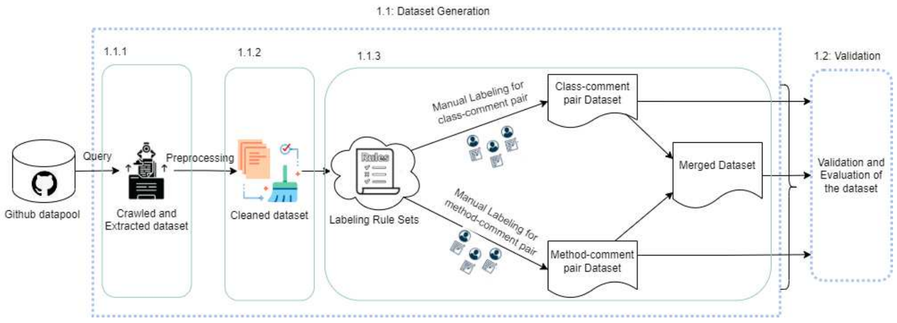

1.1: Dataset Generation
1.1.1
1.1.2
1.1.3
1.2: Validation
Github datapool
Query
Crawled and Extracted dataset
Preprocessing
Cleaned dataset
Rules
Labeling Rule Sets
Manual Labeling for class-comment pair
Manual Labeling for method-comment pair
Class-comment pair Dataset
Method-comment pair Dataset
Merged Dataset
Validation and Evaluation of the dataset

Fig. 3: Complete workflow

4.1 Dataset Generation

The dataset generation process consists of three parts: Dataset crawling and extraction, Dataset Cleaning, and Data Annotation. Each part is described in the next three subsections.

4.1.1 Dataset Crawling and Extraction

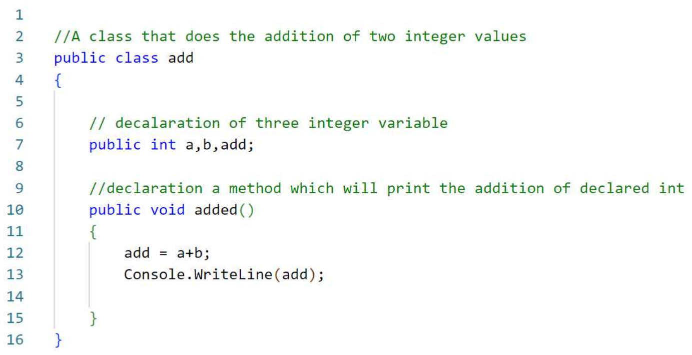
To collect a set of naturally written comments by developers, we used the extracted code-comment pair by Gelman et al.[22]. This datapool of code-comment pairs holds extracted pairs from GitHub projects for five different programming languages. C# is only considered in the scope of this work. To pull the code-comment pair, the author of [22] chose the GitHub repository based on having a redistributable license and at least 10 stars. Table 1 represents the list of the C# projects from the datapool, selected for this research. To verify the projects, first, the datapool was explored, and later from that datapool, we found the list of C# projects from the given filename. 10 projects were chosen for annotating the code-comment pair from the datapool, considering the highest number of stars. This study was restricted to making allowances for comments in an exact topological order, which is comments being on top of class and method. The renowned term for this type of comment is 'lead comment'. The selected C# projects were manually examined from the GitHub repository to ensure the comments' topological order. All the code-comment pair for these projects was then pulled from the data pool.

As aforementioned, two types of code-comment pairs were considered for the dataset. The code can be either a class or a method, and the comment is placed at the very top of that class or method.

- **Class Level Code-Comment Pair Dataset:** Object-Oriented Programming (OOP) classes are modeled after real-world entities. A user can designate a class as a blueprint or prototype from which further objects can be generated. Class declarations simply require the term class and the class identifier (name). Comments written immediately on top of class explaining the intent of the entire class are considered a lead comment. The lead comment of a class and the class itself are considered as a pair.

Table 1: Selected projects & description

ID	Project Name	Project Description	No of Stars
1	ManagedCUDA	NVidia's CUDA integration in .net applications programmed using the programming language C#	331
2	RoboSharp	RoboSharp is a .NET wrapper for the awesome Robocopy windows application	174
3	hpack	Header Compression for HTTP/2 written in C#	11
4	intense	Controls, templates, and tools for building Universal Windows Platform apps for Windows 10	90
5	Potato	Free gaming server remote control (RCON) software	20
6	EmojiVS	EmojiVS is a Visual Studio 2013 and 2015 extension to display GitHub emojis in the editor	84
7	xunit-performance	Provides xUnit plugins for creating performance tests	186
8	SimpleAuth	Simple Auth embeds authentication into the API so one doesn't need to deal with it	158
9	BDInfo	BDInfo collects video and audio technical information from unencrypted Blu-ray movie discs	18
10	Gitter	Developed as a standalone repository management tool for Windows with powerful GUI	24

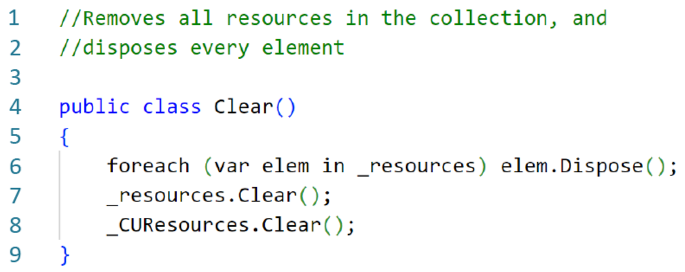


```
 1
 2  //A class that does the addition of two integer values
 3  public class add
 4  {
 5
 6      // decalaration of three integer variable
 7      public int a,b,add;
 8
 9      //declaration a method which will print the addition of declared int
10      public void added()
11      {
12          add = a+b;
13          Console.WriteLine(add);
14
15      }
16  }
```

Fig. 4: The implementation of the class 'add' and its corresponding lead comment

The Comments we considered here is the comment which is just above the class. In Figure 4, we can see that there is a class declaration named 'codeComment' in line number 3 and just above that in line number 2 the statement is the comment which is describing the class 'codeComment'.

- **Method Level Code-Comment Pair dataset:** A method is a block of code that is only executed when called. A method can receive parameters or data. Also known as functions, specific action-carrying methods are also known as functions. Using the method's name followed by parentheses to define a method.



```
1   //Removes all resources in the collection, and
2   //disposes every element
3
4   public class Clear()
5   {
6       foreach (var elem in _resources) elem.Dispose();
7       _resources.Clear();
8       _CUResources.Clear();
9   }
```

Fig. 5: Lead comment and the implementation of the method 'Clear'

The Comment we considered here is the comment which is just on the top of the method. In Figure 5, we can see that there is a method named 'Clear', and just above that, the statement is the comment which is describing the method. From Figure 5 an example of method-comment pair can also be seen written within the 'add' class. Line 9 has the lead comment of the method added and lines 10-15 has the code of the method.

4.1.2 Dataset Cleaning

Data cleaning must be done in the beginning to eliminate data noise and guarantee the correctness of the quality evaluation model. In this study, the code and comments contain a variety of identifiers and signs like punctuation marks, semicolons, brackets, and meaningless words with higher frequency. Hence, the data-cleaning procedure of this paper contains purifying special text characters, segmentation of words, etc. We removed stopwords for the English language, and words like 'public', 'class', and 'static' since these words add no extra value other than their multiple appearances in the dataset due to being a keyword of C# programming language. Symbols like '/////', '/n', '/t' were also removed from the dataset.

Table 2 shows the number of pairs distributed as class-comment pair and method-comment pair. As seen, Before and After data cleaning the number of pairs was drastically reduced. Naturally, the method-comment pair is greater than the class-comment pair as multiple methods and its lead comment can be written within one class. At first from the selected projects, 9310 pairs of code and comments were taken. Since it was extracted using Doxyzen, which

Table 2: Amount of distributed pairs

State of the Process	Class-Comment Pair	Method-Comment Pair	Total
Before Data Cleaning	1649	5432	9310 (Including Null)
After Removing Null	1171	5910	7081
After Data Cleaning	**1093**	**3829**	**4922**

often fails to extract code or comment, as a result, the space stays as null space. So we removed the Null pairs from the dataset. Table 2 also shows how much data are remaining after removing the null pairs. Among 9310 pairs 23.51% pairs had missing values, hence, tagged as Null.

The primary content of this study is the method and class-level code comment pairs. So, After discarding the Null pairs, we discard the code comment pairs with any inline comments with their corresponding commented code or constructors and their short description of a class. This marks the last step of data cleaning, after which the total number of code-comment pairs becomes 4922 as seen in Table 2.

4.1.3 Data Annotation

Annotation is a procedure for appending details to a document at some level, a word or phrase, section, or the entire manuscript [23]. This information is known as "metadata". In this research work, the initial investigation is whether the code snippet is based on classes or methods. If the code snippet is class-based, the rules for the class will be applied. If the code snippet is method-based, the rules of the method will be applied. If the code comment pair passes the rules, it will be annotated as Coherent; otherwise, it will be annotated as Incoherent.

To annotate the dataset, a human baseline approach was employed. Using the code-comment pairs of the selected projects, three annotators annotated those pairs with the defined rules for methods and classes differently. Those three annotators are Software Engineering graduates, who work in the software industry and contributed to different open-source projects. Detecting the relation between the code and the comment is a subjective task. With a substantial amount of experience and educational background, the annotators were entrusted with the annotation work. Two experts from the software industry were also present to check and ensure the quality of the set of rules and annotations.

Two sets of rules were utilized to annotate the data: Rules for method-comment and Rules for class-comment for annotating method-comment pairs already exist [6]. Table 3 contains the rules for annotating methods that were previously established and proposed by Corazza et al. aforementioned in the section 2. This set of rules is a good approach to letting others know about the intent & design implementation of the code, insights behind implementation decisions, etc. However, there are restrictions in this rule set. The rules can describe the behavior of a method and are generated exclusively for the Java programming language. This previously well-established set of rules has another setback, it is only given for method level and ignores the existence of class-level code, despite Java being an object-oriented programming language.

To annotate our data, we need two distinct rule sets, as we are focusing on both the method and class levels. The rules used to annotate classes are listed in Table 4. This set of rules was reviewed by two experts. The annotation procedure was identical to that for pairings of method code and comments. The rules specify the presence of the reason for the class declaration, implementation details of the class, information regarding the inherited class, polymorphism, and information regarding the declared constructor in the class.

Table 3: Established rules for annotating method code-comment pair[6]

ID	**Rules for Methods**
1	The comment of the method describes the intent of the source code of this method.
2	The intent described in the lead comment of the method corresponds to the actual implementation of this method.
3	The comment of the method describes all the expected behaviors of the actual implementation of this method.
4	If the comment of a method provides implementation details (e.g., names and types of input parameters according to JavaDoc), this information is aligned with the implementation of this method.
5	If the comment of the method provides details about input parameters, their intended use is properly described.

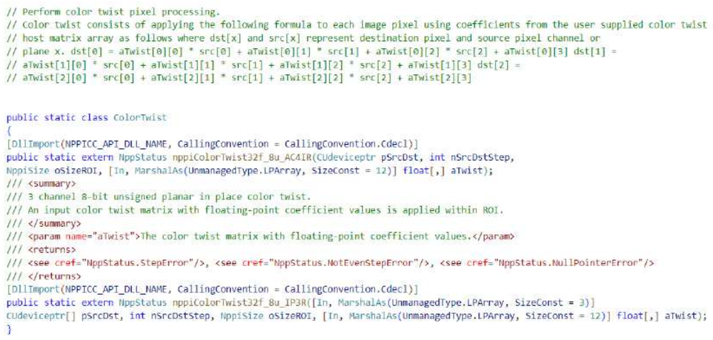
Using the annotation process while keeping the rules sets for both method level and class, we have annotated the data. If the code-comment pair passes any 3 rules or for both code-comment and method-comment pairs if it passes the first 2 rules that pair is considered a coherent pair, otherwise it is labeled as incoherent.

In Figure 6, we've extracted a portion of code from our dataset. The code snippet describes a class and its corresponding topological comments, but according to our research, we have only considered the class's lead comment. As the code snippet was a very long class, a portion of it has been truncated to facilitate its use as an image. This pair has been annotated by three different

Table 4: Proposed rules for annotating class code-comment pair

ID	Rules for Classes
1	Lead comment for the class expresses the purpose of the declaration of that class.
2	The lead comment for the class gives the idea of the constructor of that class along with the parameters used in the constructor according to the alignment to the implementation of the class.
3	Comment of a class delivers the basic implementation information—for example, names and types of input parameters. The information is in orientation with the implementation of the class.
4	The lead comment contains information about the inheritance properties of the class.
5	The lead comment contains information about the polymorphic methods of the class.

annotators as a coherent code-comment pair. To associate this with the proposed set of rules for the class, the lead comment here describes the purpose of the class declaration and provides the basic implementation details which co-relates to rules no 1, and 3 and partially with rule no 2.

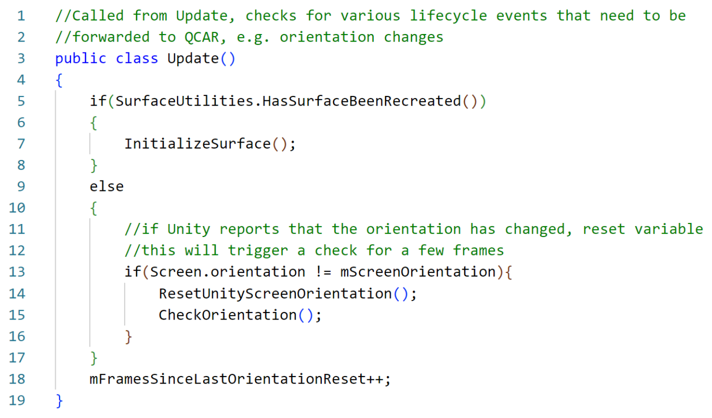


```
// Perform color twist pixel processing.
// Color twist consists of applying the following formula to each image pixel using coefficients from the user supplied color twist
// host matrix array as follows where dst[x] and src[x] represent destination pixel and source pixel channel or
// plane x. dst[0] = aTwist[0][0] * src[0] + aTwist[0][1] * src[1] + aTwist[0][2] * src[2] + aTwist[0][3] dst[1] =
// aTwist[1][0] * src[0] + aTwist[1][1] * src[1] + aTwist[1][2] * src[2] + aTwist[1][3] dst[2] =
// aTwist[2][0] * src[0] + aTwist[2][1] * src[1] + aTwist[2][2] * src[2] + aTwist[2][3]


public static class ColorTwist
{
[DllImport(NPPICC_API_DLL_NAME, CallingConvention = CallingConvention.Cdecl)]
public static extern NppStatus nppiColorTwist32f_8u_AC4IR(CUdeviceptr pSrcDst, int nSrcDstStep,
NppiSize oSizeROI, [In, MarshalAs(UnmanagedType.LPArray, SizeConst = 12)] float[,] aTwist);
/// <summary>
/// 3 channel 8-bit unsigned planar in place color twist.
/// An input color twist matrix with floating-point coefficient values is applied within ROI.
/// </summary>
/// <param name="aTwist">The color twist matrix with floating-point coefficient values.</param>
/// <returns>
/// <see cref="NppStatus.StepError"/>, <see cref="NppStatus.NotEvenStepError"/>, <see cref="NppStatus.NullPointerError"/>
/// </returns>
[DllImport(NPPICC_API_DLL_NAME, CallingConvention = CallingConvention.Cdecl)]
public static extern NppStatus nppiColorTwist32f_8u_IP3R([In, MarshalAs(UnmanagedType.LPArray, SizeConst = 3)]
CUdeviceptr[] pSrcDst, int nSrcDstStep, NppiSize oSizeROI, [In, MarshalAs(UnmanagedType.LPArray, SizeConst = 12)] float[,] aTwist);
}
```

Fig. 6: Implementation of the class 'ColorTwist' and its corresponding lead comment

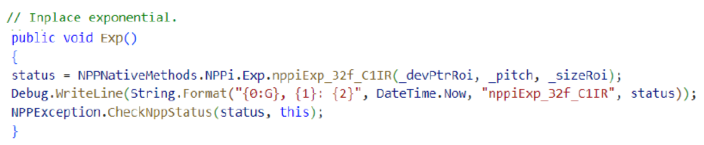
From Figure 7, a snippet of our annotated coherent code-comment pair. Code is the 'Update' method and lead comment is the comment written on top of the code. This pair was annotated as Coherent for satisfying rule no. 1, 2 but rule no. 4 & 5 is invalid for this method as this is a void method.



```
//Called from Update, checks for various lifecycle events that need to be
//forwarded to QCAR, e.g. orientation changes
public class Update()
{
    if(SurfaceUtilities.HasSurfaceBeenRecreated())
    {
        InitializeSurface();
    }
    else
    {
        //if Unity reports that the orientation has changed, reset variable
        //this will trigger a check for a few frames
        if(Screen.orientation != mScreenOrientation){
            ResetUnityScreenOrientation();
            CheckOrientation();
        }
    }
    mFramesSinceLastOrientationReset++;
}
```

Fig. 7: Implementation of the method ‘Update’ and its corresponding lead comment

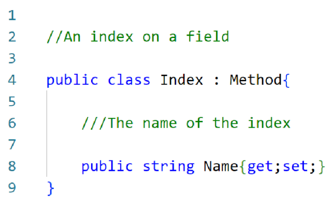
Figure 8 shows an example of an incoherent lead comment and void method. The reason behind labeling this pair as incoherent is this pair does not abide by any of the rules.



```
// Inplace exponential.
public void Exp()
{
status = NPPNativeMethods.NPPi.Exp.nppiExp_32f_C1IR(_devPtrRoi, _pitch, _sizeRoi);
Debug.WriteLine(String.Format("{0:G}, {1}: {2}", DateTime.Now, "nppiExp_32f_C1IR", status));
NPPException.CheckNppStatus(status, this);
}
```

Fig. 8: Implementation of the method ‘Exp’ and its corresponding Lead Comment

Figure 9 is an example of an incoherent lead comment and class. The reason behind labeling this pair as incoherent is this pair also does not satisfy any of the proposed rules to be a coherent code(class)-comment pair.



```

//An index on a field

public class Index : Method{

    ///The name of the index

    public string Name{get;set;}
}
```

Fig. 9: Incoherent lead comment and the implementation of the class ‘Index’

3 annotators, all of them having at least 1-year experience in the software industry, separately annotated the code-comment pairs checking both sets of rules. After annotating the dataset, Table 5 reveals that there are 1093 code-comment pairs for class and 3829 code-comment pairs for a method out of a total of 4922 code-comment pairs, of that 76.08% are coherent and 23.92% are Incoherent.

Table 5: Description of the coherent and incoherent code-comment Pairs

Dataset	**Coherent**	**Incoherent**	**Total**
Class-comment pair dataset	1021 (93.41%)	72 (6.59%)	1093
Method-comment pair dataset	2719 (71.00%)	1110 (29.00%)	3829
Merged dataset	3745 (76.08%)	1177 (23.92%)	4922

4.2 Validation and Evaluation of the Dataset

Since the three annotators annotated the dataset separately, it was essential to check how much the annotation for each data point of all three different annotators matched. To ensure that, we used Simpledorff - Krippendorff's Alpha [24]. Krippendorff's Alpha is a well-known inter-annotator reliability metric. It is devised to calculate the like-mindedness among observers, coders, reviewers, or raters, drawing distinctions among generally unformed spectacles or allocating computable weights to them. Values of Krippendorff's Alpha(α) range from 0 to 1, where 0 is perfect disagreement, and 1 is excellent agreement. Krippendorff suggests that it is conventional to require $\alpha \geq .800$. Where indecisive findings are still acceptable, $\alpha \geq .667$ is the lowest conceivable limit[25]. There is a 95.76% similarity in our dataset between the rating of 3 annotators for all the code-comment pairs. The percentage of similarity shows how much the rating matched among the raters. Upon investigation, it was found that the dissimilarity occurred for one rater with different labeling for a few small-size method-comment pairs than the other two raters.

Besides Krippendorff's Alpha, the bilingual evaluation understudy (BLEU) score is also used as an evaluation metric for each code-comment pair to check the validity and compare our annotated dataset. Many notable recent works have used the BLEU score as their primary evaluation metric. The BLEU score estimation includes a concise sentence that disciplines brief predictions. Therefore, the two pieces balance, awarding the model for foretelling solely n-grams incorporated in the reference sentence and rewarding the model for making prolonged predictions. BLEU-1 was employed, implying the BLEU score estimation includes 1 gram and every smaller n-gram. Agreeing with Hu et al.[20], this in-work smoothing was also unused to settle the insufficiency of higher-order n-gram overlapping.

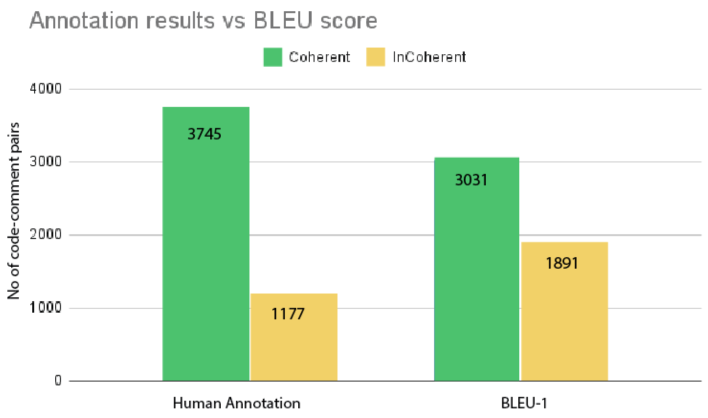


Fig. 10: Human curated annotation vs BLEU score

From Figure 10, it is visible that the BLEU labeled 714 pairs as Incoherent whereas in our human-curated dataset the annotators annotated those as consistent ones. The hypothesis is BLEU does not assume the definition of words, while it is permissible for a human to employ a different word with similar meaning. The BLUE score also ignores paraphrasing.

5 Model Training Detail & Result

In previous studies, different kinds of approaches have been proposed to detect code comment consistency. The topic modeling approach is employed because it analyzes text data to select cluster words for a set of documents. Among previously published models that employed a topic modeling approach along with an ensemble machine learning technique one caught our attention because of the use of a multi-model technique which was adapted with our modifications[7].

5.1 Model Training

Our modifications include adding natural language processing-based preprocessing techniques like lemmatization, removal of stop words, etc. Another modification of our is employing logistics regression to infuse the extracted features from the multi-model random forest. As a cross-validator stratified k-fold was applied.

Along with the modification the overall process of the method can be summarised as follows:

- **Preprocessing of code-comment pairs:** The newlines, tabs and special characters, Extra whitespaces, and stop words were removed at first. Words were also split based on camel cases. Tokenized pairs were turned into a bag of word tokens. Later on, lemmatization was applied.
- **Feature vectors from code-comment:** We used Count Vectorizer to turn the tokenized code-comment into an index vector. Then index vectors of code/comment were separated from each other and passed into two identical LDA models for training corpora separately. For n number of topics, both LDA models provided different probability distributions. Afterward, the achieved probability distribution for a code-comment pair is concatenated to produce a feature vector.
- **Multiple classifier:** Extracted features of n number of topics were fit into a random forest(RF) classifier. The RF model delivers a consistency score or probability score of being consistent for every pair of code-comment using the 'proba' function. Since the topic numbers for both corpora of comment code might vary, it was necessary to obtain multiple numbers of topics to find the informative and compelling features. Due to necessity, 20 different feature sets were extracted, as seen in fig 7, by changing the number of topics in the LDA every time. These extracted features were fitted into 20 different random forest classifiers.
- **Infusing with logistic regression:** Each random forest model produces a consistency score for a code-comment pair. These consistency scores originated from 20 different random forests required to be fused. Here, Logistic Regression combines the consistency scores supplied by 20 other random forests and predicts the result.

5.2 Result Analysis

After using the modified model on our prepared dataset, the performance score for multiple evaluation metrics for three datasets can be observed from Table 6. In this table, there are accuracy, precision, recall, and AUC-ROC scores of three separate datasets. One of the primary reasons behind this good result is that the code section has a high amount of docstrings written within it. So when the model matches code and comments based on appearance and occurrence, they can find similar words present in the comment in the docstring, if not in the code.

Fitting the model on our annotated 4922 code-comment pairs of the merged dataset, we got the value 96.2% which is better than the previous model which gave 93.1% using the performance metric AUC-ROC. The value difference is shown in Table 7.

Table 6: Performance of multiple evaluation metrics for 3 datasets

Dataset	**Accuracy**	**Precision**	**Recall**	**AUC-ROC**
Class-Comment	0.973	0.990	0.980	0.919
Method-Comment	0.974	0.975	0.989	0.963
Merged	0.973	0.981	0.983	0.962

Table 7: Comparison between ours and a state-of-the-art model on our proposed dataset(merged)

Model	**AUC-ROC**	**Accuracy**	**Precision**	**Recall**
Rabbi et al.[7]	0.931	0.96	0.95	0.96
Our Proposed	**0.962**	**0.973**	**0.981**	**0.983**

6 Threats to Validate

One of the external threats of this work can be using selected systems and the programming language. To validate this external threat, we have used checklist-based annotation work, so that the dataset can be easily extendable for other languages. The dataset proposal of our work can be seen as an elongation work of Anna Corazza et al. work, they proposed a checklist-based public benchmark dataset for the Java programming language. As far as the model is concerned, it is an enhanced version of a previous study.

7 Conclusion & Future Work

The AI community is now moving towards Data Centric AI. Due to this, building a suitable dataset is becoming far more acceptable than building complex models. We have presented a dataset of an analysis on the coherence between comment of methods and lead comments of class and their corresponding implementations. Human annotation and BLUE-1 label were different for 14.5% pairs among the total dataset. 95.76% similarity is found among the annotation of 3 annotators using the inter-annotator reliability metric, Krippendorff's Alpha. A description of the process, problems, and mitigating procedures while creating the dataset is also delivered. A step forward in this forthcoming research direction could be making it language-agnostic. Furthermore, the topological order of the comment should be analyzed more closely. More ways other than topic modeling can be explored. Even though the BLEU score has been used before to evaluate these kinds of datasets earlier, none of the code or comments was machine-generated. Exploring different metrics to evaluate this kind of dataset should be a priority. Using logistic regression in the ensemble model had the upper hand.

8 Statements and Declarations

8.1 Author's Contribution

Maksuda Islam, Mohammad Safayat Hossen, and Ahsanul Haque did the research work and wrote the main manuscript text. Md. Nazmul Haque and Lutfun Nahar Lota supervised the whole work and reviewed the manuscript.

8.2 Funding Declaration

The authors did not receive any financial or non-financial support/funding from any organization for the submitted work.

8.3 Conflict of Interest

The authors have no conflicts of interest to declare that are relevant to the content of this article.

8.4 Data Availability

The dataset generated during and analyzed during the current research study is available in the 'code-comment-consistency' repository, https://github.com/kima063/code-comment-consistency.

References

[1] Tenny, T.: Program readability: Procedures versus comments. IEEE Transactions on Software Engineering **14**(9), 1271–1279 (1988)

[2] Woodfield, S.N., Dunsmore, H.E., Shen, V.Y.: The effect of modularization and comments on program comprehension. In: Proceedings of the 5th International Conference on Software Engineering, pp. 215–223 (1981)

[3] Ko, A.J., Myers, B.A., Coblenz, M.J., Aung, H.H.: An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. IEEE Transactions on software engineering **32**(12), 971–987 (2006)

[4] Piorkowski, D., Henley, A.Z., Nabi, T., Fleming, S.D., Scaffidi, C., Burnett, M.: Foraging and navigations, fundamentally: developers' predictions of value and cost. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 97–108 (2016)

[5] Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification. Addison-Wesley Professional, ??? (2000)

[6] Corazza, A., Maggio, V., Scanniello, G.: Coherence of comments and method implementations: a dataset and an empirical investigation. Software Quality Journal **26**(2), 751–777 (2018)

[7] Rabbi, F., Haque, M.N., Kadir, M.E., Siddik, M.S., Kabir, A.: An ensemble approach to detect code comment inconsistencies using topic modeling. In: SEKE, pp. 392–395 (2020)

[8] Iammarino, M., Aversano, L., Bernardi, M.L., Cimitile, M.: A topic modeling approach to evaluate the comments consistency to source code. In: 2020 International Joint Conference on Neural Networks (IJCNN), pp. 1–8 (2020). IEEE

[9] Salviulo, F., Scanniello, G.: Dealing with identifiers and comments in source code comprehension and maintenance: Results from an ethnographically-informed study with students and professionals. In: Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering, pp. 1–10 (2014)

[10] Tan, L., Yuan, D., Krishna, G., Zhou, Y.: /* icomment: Bugs or bad comments?*. In: Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles, pp. 145–158 (2007)

[11] Tan, S.H., Marinov, D., Tan, L., Leavens, G.T.: @ tcomment: Testing javadoc comments to detect comment-code inconsistencies. In: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, pp. 260–269 (2012). IEEE

[12] Ibrahim, W.M., Bettenburg, N., Adams, B., Hassan, A.E.: On the relationship between comment update practices and software bugs. Journal of Systems and Software **85**(10), 2293–2304 (2012)

[13] de Souza, S.C.B., Anquetil, N., de Oliveira, K.M.: A study of the documentation essential to software maintenance. In: Proceedings of the 23rd Annual International Conference on Design of Communication: Documenting & Designing for Pervasive Information, pp. 68–75 (2005)

[14] Khamis, N., Witte, R., Rilling, J.: Automatic quality assessment of source code comments: the javadocminer. In: International Conference on Application of Natural Language to Information Systems, pp. 68–79 (2010). Springer

[15] Liu, Z., Chen, H., Chen, X., Luo, X., Zhou, F.: Automatic detection of outdated comments during code changes. In: 2018 IEEE 42nd Annual

Computer Software and Applications Conference (COMPSAC), vol. 1, pp. 154–163 (2018). IEEE

[16] Rani, P., Panichella, S., Leuenberger, M., Ghafari, M., Nierstrasz, O.: What do class comments tell us? an investigation of comment evolution and practices in pharo smalltalk. Empirical Software Engineering **26**(6), 1–49 (2021)

[17] Lucia, D., *et al.*: Information retrieval models for recovering traceability links between code and documentation. In: Proceedings 2000 International Conference on Software Maintenance, pp. 40–49 (2000). IEEE

[18] Papineni, K., Roukos, S., Ward, T., Zhu, W.-J.: Bleu: a method for automatic evaluation of machine translation. In: Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, pp. 311–318 (2002)

[19] Callison-Burch, C., Osborne, M., Koehn, P.: Re-evaluating the role of bleu in machine translation research. In: 11th Conference of the European Chapter of the Association for Computational Linguistics, pp. 249–256 (2006)

[20] Hu, X., Li, G., Xia, X., Lo, D., Lu, S., Jin, Z.: Summarizing source code with transferred api knowledge (2018)

[21] Iyer, S., Konstas, I., Cheung, A., Zettlemoyer, L.: Summarizing source code using a neural attention model. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pp. 2073–2083 (2016)

[22] Gelman, B., Obayomi, B., Moore, J., Slater, D.: Source code analysis dataset. Data in brief **27**, 104712 (2019)

[23] Petrillo, M., Baycroft, J.: Introduction to manual annotation. Fairview research, 1–7 (2010)

[24] Krippendorff, K.: Computing krippendorff's alpha-reliability (2011)

[25] Ford, J.M.: Content analysis: An introduction to its methodology. Personnel Psychology **57**(4), 1110 (2004)