Understanding the Rationale for Updating a Function's Comment

Haroon Malik, Istehad Chowdhury, Hsiao-Ming Tsou, Zhen Ming Jiang, Ahmed E. Hassan
Software Analysis and Intelligence Lab (SAIL)
School of Computing, Queen's University, Kingston, Ontario, Canada.
{malik,istehad,tsou,zmjiang,ahmed}@cs.queensu.ca

Abstract

Up-to-date comments are critical for the successful evolution of a software application. When modifying a function, developers may update the comment associated with the function or may not update it. For example, comments associated with a complex function are likely to be updated more often when the function is modified to prevent the code and the comments from drifting apart. Nevertheless, the rationale behind updating a comment has never been studied.

In this paper, we present a large empirical study to better understand the rationale for updating comments. We recover the code change history for four large open source projects (GCC: a compiler, FreeBSD: an operation system, PostgreSQL: a database management system, and GCluster: a clustering framework) with an average code history of 10 years. Using the Random Forests algorithm, we investigate the rationale for updating comments along three dimensions: characteristics of the changed function, characteristics of the change itself and time and code ownership characteristics. Our case study shows that we can predict with an accuracy of 80%; the likelihood of updating the comment associated with a modified function. We perform a sensitivity analysis to determine the most important attributes. Our analysis shows that the percentage of changed call dependencies and control statements, the age of the modified function and the number of co-changed functions which depend on it are the most important attributes in determining the likelihood of updating comments.

1. Introduction

The task of commenting one's source code is often neglected, though the merits of writing consistent comments are well-known [1]. Reading code is a fundamental task during software engineering and code is read more often than it is written [2]. Elshoff and Marcotty states that the comments, as well as the structure of the source code are good aids for understanding programs and therefore reduce maintenance costs [3]. This finding was confirmed by the studies of Ted Tenny [4]. But, as the example of Lakhotia shows, sometimes programmers do not care whether someone else might want to understand their source-code [5]. Developers spend more than half of their time trying to understand code. The lack of comments as well as the existence of outdated comments is counter-productive.

We view comments in similar capacity as [6] with code comments being of two types: *header or outer comments* and *non-header or inner comments*. *Header comments* are comments written before the definition of a function; whereas *non-header comments* are all other comments residing in the body of a function. Developers usually use header comments to describe the purpose of a function, and to document its parameters and interfaces. Non-header comments are usually used to document algorithms and low-level design decisions. In this paper, we sought to investigate the rationale for updating both of these types of comments when their associated function is modified.

We conducted an empirical study using the development history of the four large open source projects, listed in Table 6. We recovered all code changes for these four projects using the C-REX evolutionary extractor [15]. We then investigated whether we could predict the likelihood of updating a comment when its associated function is modified along three dimensions: characteristics of the changed function, characteristics of the change and the time and code-ownership characteristics. Along each dimension, we investigated several attributes and defined measures to quantify these attributes. For example, for the characteristics of the changed function dimension, we used the number of changed dependencies to capture the level of coupling [11]. Our results show that:

1. We can predict with 80% accuracy the likelihood of updating the comment for a modified function.
2. Percentage of changed dependencies and control statements, age of the modified function, and the

number of co-changed functions which depend on it are the most important attributes in determining the likelihood of updating comments.
3. Our findings are consistent across four large open source projects from different domains: GCC (compilers), FreeBSD (operating systems), PostgreSQL (databases), and GCluster (cluster).

Organization of the Paper

The organization of the paper is as follow. Section 2 presents related work. Section 3 discusses the three dimensions and presents the attributes along these dimensions. This section also highlights the intuition behind our work. Section 4 presents the Random Forests algorithm and motivates its use for studying the rationale for updating comments. Section 5 presents our case study and details the results of the various experiments conducted in our study. Finally, Section 7 concludes the paper and identifies avenues for future work.

2. Related Work

Jiang and Hassan studied the evolution of comments in PostgreSQL [6]. They investigated the addition and deletion of header and non-header comments over time. In contrast to their work, we do not restrict ourselves to studying additions and deletions. We also study changes to comments and seek to uncover the rationale for such changes.

Lawrie *et al.* [8] employed information retrieval techniques to measure how comments relate to the source code and assume that comments impact the code quality of a software system. Marcus and Poshyvanyk [7] defined metrics for measuring the conceptual cohesion of classes by incorporating the presence (or absence) of comments.

Ying *et al.* [9] investigated the use of a particular type of comment, the Eclipse task comments. These are special comments which start with *// TODO* and are commonly used by developers using the Eclipse IDE. They argued that task comments tend to depend on the context of the surrounding code and that it is difficult to infer the scope of a task comment. This often holds for comments in general and, therefore, has an impact on our work. Ying et al. mentioned a few reasons and patterns that lead to an insertion of a comment task (for example as pointers to change requests) but they did not study whether some building blocks of a program (e.g., if-statements) are more likely to be commented. Fluri *et al.* investigated the co-evolution of source code and its associated comments [16]. They presented a technique to map source code entities to comments in the code and to extract comment changes over the history of a software project. Their study focused on the ratio between the source code and comments over the history of project and the entities that are most likely to be commented e.g., classes, methods, and control statements. They did not investigate the rationale for comment changes. We believe that a good understanding of the rationale for updating comments is important to better understand the factors affecting the evolution of large software systems. Versioning systems such as CVS and subversion do not provide features for fine-grained source code change analysis nor for tracking comments. In fact, they are not capable of providing a mechanism for distinguishing comments from source code without the use of additional analysis tools. For our work, we use the C-REX tool [15] to map historical code changes to the appropriate code entities, e.g., functions, or comment blocks, and to link comments to the their corresponding entities.

3. Dimensions used to predict the likelihood of updating a comment

We investigate the rationale for updating a comment once its associated function is modified along three dimensions: 1) characteristics of the changed function, 2) characteristics of the change itself, and the 3) change time and code-ownership characteristics.

Table 1: Number of attributes per dimension

	Dimensions	Attr
1	Characteristics of the modified function	10
2	Characteristics of the change	10
3	Change time and code-ownership characteristics	9
	Total attributes across all dimensions	**29**

In this section, we explain each dimension and the various attributes, which we chose to consider within each dimension. Moreover, we explain our motivation for picking particular attributes. Table 1 lists the number of attributes considered along each dimension.

3.1 Characteristics of the modified function

We explore this dimension based on our strong belief and intuition that modifications to complex function are trickier and more likely to introduce integration bugs [12]. Therefore, such modifications must be properly commented to avoid the introduction of bugs. Table 2 describes the various attributes along this dimension.

Table 2: Characteristics of the modified function

Attribute Name	Explanation and Rationale
Length of the function name	This attribute reflects to some extent the documentation maturity of a project. Longer function names are probably a good indication of the documentation habits of the developers working on the project. Longer names are good indicators of the team's focus on maintaining understandable and readable code.
Number of dependencies	The number of dependencies is an indicator of the importance of a function. We expect functions with a large number of dependencies to have their comments updated more frequently.
Number of control statements	The number of control statements is a proxy for the Mccabe complexity metric [12]. We hypothesize that more complex code will lead to higher frequency of comment-updates.
Number of inner comments	The amount of inner comment keywords may imply numerous things: the complexity of the function or the tendency of the development team to comment function's in general.
Number of outer comments	Unlike inner comments, the outer comments (comments included in function's header) are usually there to describe the role of a function and its interface, rather than to describe the inner-workings of the function.
Has inner comments, Has outer comments, Has comments	These are binary versions of the above metrics. We use these simpler metrics to determine if a simpler model is possible or if the actual count of tokens is important.
Number of parameters	The number of parameters passed into the function is likely to indicate the complexity of using a function. Therefore, we would expect that modifying such complex functions is more likely to lead to updating their comments.
Number of string literals	The number of string literals (i.e., constant strings) in the function is an indicator of functions that are likely interacting with the user or with the environment. Changes to such functions are likely to require updating the comments since the changes may lead to changes in the interaction of the software system with its surrounding environment.

3.2 Characteristics of the change

This dimension seeks to understand the comment update phenomena from the point of view of the change itself rather than the current state of the modified function. This dimension has attributes related to the actual change such as the number of changed control statements and the percentage of changed dependencies. The intuition is that more extensive and complex changes will increase the probability that a comment will get updated.

Table 3: Characteristics of the change

Attribute Name	Explanation and Rationale
Number of changed (i.e. added or removed) dependencies	If function A calls functions B, then we say there is a dependency between A and B. It is natural to think that there would be an inner comment associated with a dependency call within the function. If one is added or removed, a comment may also be added or removed.
Number of changed control statements	The logic of considering this attribute is the same as the number of added or removed dependencies in the change list. A change in flow within the function ought to introduce a change in the comments associated with the function.
Number of changed strings	A string change can be associated with a locally significant modification, thus comments may change.
Percentage of changed dependencies, Percentage of changed control statements, Percentage of changed strings	These metrics mirror the above three metrics but try to capture the significance of a change. For example, a single dependency change for a function with one dependency is significant relative to the same change for a function with 20 dependencies.
Did return value change?	If the return type of a function changes, we would expect that the header comments should change.
Percentage of past comments updates	The percentage of times entity had its comments updated when it was changed. We expect that functions, which tend to have their comments updated when they are modified, will follow the same pattern for future changes.
Number of co-changed functions	This attribute represents the magnitude of the transaction itself. A large transaction may be an indicator of a large scale change which may lead to a high probability of comment updates for the functions in the transaction.
Is the change a bug fix?	Bug fix changes more likely to instigate comment updates; due to clarification notes, to explain the bug fixes?

Every change is associated with a change-list which records all the functions that changed with the current function. This dimension also covers attributes about the transaction associated with the change, such as the total number of co-changed functions. Table 3 lists the various attributes in this dimension.

3.3 Time and code-ownership characteristics

This dimension tracks information about the time of the change, such as the day of the week, how long it has been since the function has been last modified and who made the change.

Table 4: Time and code-ownership characteristics

Attribute Name	Explanation and Rationale
Year	Do comment-update habits change over time? When a software application is new, there is a higher tendency to update comments than when the application matures.
Weekday	This attribute records the day the change was committed into the source control system. Do developers commenting habits change based on the day of the week? Recent research shows that developers are more likely to introduce bugs on Friday [14], could this be due to changes on Fridays where the comments were not updated?
Day or Night	'DAY' if between 8:00 AM and 10:00 PM, 'Night' otherwise. This attribute signifies the time of the change. Are developers less likely to update the comments for the late night changes?
Month, Quarter	This attribute signifies the month or quarter in the year of the transaction. Are developers less likely keep comments up-to-date during summer months or around the holiday times?
Age of the function in days	Are more mature functions likely to have their comments updated when they are modified, or do most comment updates occur when functions are young
Days since last change	The number of days since the last time a function was changed. The current team may have limited knowledge of a function that has not changed for a long time; Therefore, we expect that the developers would spend more time trying to understand the code before the change. We would hope that the developers would then go ahead and update the comments based on their gained understanding.
Initial developer	The name of the developer who first committed this function. The developer's coding, design, and commenting styles are likely to play an important role in the how comments in that functions are updated in future.
Current developer	The name of the developer who committed the most recent change to this function. We choose this attribute based on similar rationale as that of Initial developer.
Same as initial developer	'YES' if current developer is the same as initial one, 'NO' otherwise. Differences in developers' styles may encourage changes in comments too.

Table 4 lists various attributes along this dimension. The motivation towards selecting these attributes is to see if time has any impact on a developer tendency to update a comment. Are developers more likely to update a comment on Fridays over other weekdays? Similar observations are noted by Zimmermann *et al.* in relation to the likelihood of introducing bugs on Fridays [14]. Are developers more likely to update comments over the weekends than during the weekdays? These are few of the questions which we expect to answer based on this dimension. This dimension also covers attributes which highlight the relation of a function with developers, such as whether the change was done by the same author who created the function. We expect that there would be an overhead and complexity associated with modifying a function that was not written by the developer performing the change. The developer performing the change may end up documenting their new understanding or at least clarifying the current documentation. We also expect that the likelihood of comment update depends on the developer performing the change. For example, novice developer may be more likely to update comments. After all, Khoshgoftaar *et al.* shows that the experience of a developer contributes to their tendency to introduce bugs [22].

4. Random Forests

The purpose of our study is to understand the rationale for updating a comment when its associated function is modified. In our study, we choose to use a large number of attributes to predict whether a change to a function will lead to its comment being updated or not. We model our study as a classification problem where each code change to a function can fall into one of the two classes: comment updated (YES) or not updated (NO).

There exist several machine learning techniques, such as Support Vector Machines (SVM) and neural networks, which can solve this classification problem. However, we chose to use a techniques based on decision trees since decision trees produce explainable models. These explainable models are essential in helping understand the comment update phenomena and to find out the important attributes in determining the likelihood of updating a function's comment. However, instead of using basic decision tree algorithms such as C4.5 [18], we used an advanced decision tree algorithm called Random Forests [17].

The Random Forests algorithm outperforms basic decision tree and other advanced machine learning algorithms in prediction accuracy. Moreover, the Random Forests is more resistant to noise in the data. This is an important advantage. We expect that the data used in our study to have noise due to its massive size and the length of the studied time period (over 40 years of change history). The Random Forests algorithm requires a limited number of configuration parameters and produces robust and stable models [19]. Finally, often the prediction accuracy of basic decision tree algorithms suffers when many of the attributes are correlated. Given the large number of attributes in our study, we need an algorithm that does not suffer from correlated attributes. Fortunately, the Random Forests algorithm deals well with correlated attributes while maintaining a high accuracy in its prediction. In contrast to simple decision tree algorithms, the Random Forests algorithm builds a large number of basic decision trees (40 trees in our case study). Each node in each tree is split using a random subset of all the attributes to ensure that all the trees have low correlation between them. We use the default random subset value which is the square root of all the studied attributes. The trees are built using 2/3 of the available data using sampling with replacement. The 1/3 of the remaining data is called the Out Of Bag (OOB) data and is used to test the prediction accuracy of the created forest. The use of bootstrapping and random selection of attributes at each node greatly improves the accuracy of tree based classifiers [20].

In our case study we use the OOB data to measure the accuracy of the created forest. Each sample in the OOB is pushed down all the trees in the forest and the final class of the sample is decided by aggregating the votes (i.e., predicted class) of each tree. One major benefit of using this technique is that we can adjust the votes accordingly based on the skewness in the data. Basic decision trees are known to perform badly with highly skewed data since the tree always changes to predict the dominant class. To overcome this problem in a Random Forests, we can assign weights to votes to offset the data skew. For example, in our analysis of the PostgreSQL project, the ratio of comment updated (YES) to comment not updated (NO) is 1:1.6 as seen in Table 6. Therefore we assign the weights 16:10 for the YES and NO classes.

Table 5: Confusion matrix

True Class	Classified As	
	YES	NO
YES	a	b
NO	c	d

To measure the accuracy of the prediction produced by the Random Forests algorithm, we calculate the overall, YES, and NO misclassification rates. We desire the lowest overall and per-class misclassification rates. The rates are defined using the confusion matrix, shown in Table 5. The YES and NO represents the two classes: Comment updated and Comment not updated. "True Class" column represent the actual number of comment updated/not updated. Whereas a, b, c & d under "Classified As" column represent arbitrary values of correctly or misclassified instances by predictor against true class. For example, if there are 100 instances of an attribute for which comment has been updated (True class: YES), the classifier may correctly predict 90 instances (a=90) and may predict 10 incorrectly classified instances (b=10) for that class. We further explain how we derived the misclassification rate with the help of Table 5.

- YES misclassification rate: This captures the performance of the forests for updating a comment. It is defined as: b/(a+b).
- NO misclassifications rate: This captures the performance for not updating a comment. It is defined as: c/(c+d).
- Overall misclassification rate: This captures the overall performance of the forests for both classes (YES and NO). It is defined as: (b+c)/(a+b+c+d).

Sensitivity Analysis

Another benefit of using the Random Forests is that we can perform sensitivity analysis on the attributes to determine the most important attributes in the created forests. To perform the sensitivity analysis for a particular attribute, the value of the attribute is randomly changed in all the samples in the OOB data. Samples are then re-classified. Thereafter, we measure the change in misclassification rate. If an attribute is not important then we expect that the misclassification rate will not change much. Otherwise the misclassification rate would increase relative to the importance of an attribute. In our case study, we created ten Random Forests of 40 trees each (400 trees in total) for each set of attributes; we then measured the average misclassification rates. The average misclassification rates for our case study are reported in Table 9. To determine the most important attributes in all of the ten forests, we calculated the attribute importance for each forest then combined the importance ranking for each forest to reach the overall importance values shown as weights in our case study. The importance weight are calculated as follows:

1) For each forest, each attribute is given a point from 1 to 10 relative to its rank in that forest: 10 for most important, 1 for the least most important and 0 if higher than ten.
2) For each attribute, we sum its points across all ten forests and we divide this sum by the maximum number of points which the attribute can get. The maximum of points is 100: 10 (for highest rank) X 10 (for ten forests). We multiply the resulting value by 100 to get a value between 0 and 100. We use this weight metric to measure the most important attributes. We only show weights higher than 50.

Table 6: Summary of Studied Systems

Studied System		Date		Comments		Studied		
Name	*Type*	*Start*	*End*	*Updated*	*Not updated*	*Functions*	*Change lists*	*Lang.*
PostgreSQL	DBMS	July 1996	Feb 2008	8,817	14,251	31,000	9,705	C
FreeBSD	OS	June 1993	Aug 2005	30,188	7,768	27,935	20,108	C
GCluster	Cluster Platform	June 2004	Feb 2008	4,488	984	15,539	1,890	C
GCC	Lang. Compiler	Aug 1997	Oct 2005	16,025	7,735	22,460	13,125	C

5. Case Study

We perform an extensive case study to substantiate that the selected attributes and the heuristics pertaining to the attributes are applicable in predicting comment update across varied software systems. Table 6 classifies the projects used in our case study. The average history of the projects is around 10 years, with PostgreSQL and FreeBSD having the largest historical life span of 12 years and Cluster having the smallest historical span of 4 years. The average number of functions in a project is around 24,234. PostgreSQL project has the largest the number functions (31,000 functions), whereas the GCluster project takes the last position with 15,539 functions. The average number of files for a project is around 1,112 files. In our case study, we conducted five experiments. Three experiments (one for each of the three dimensions) are detailed in Section 3. The fourth experiment studies all the attributes for each project. In the fifth experiment, we combined all the attributes for the three dimensions for all the projects taken together. For each experiment, we created decision trees using the Random Forests algorithm then performed sensitivity analysis to determine the most important attributes in improving the accuracy of our predictions.

5.1. Exp. #1: Characteristics of modified function

Our first experiment examines attributes along the characteristics of the modified function. Table 2 lists the detailed attributes in this dimension. The most important attributes for predicting a comment updated are shown in Table 7. Function characteristics such as the total number of comments, number of inner comments, number of string literals, and number of control statements are very influential in predicting the likelihood of updating a comment. For PostgreSQL, GCluster, and GCC, the total number of comments (inner and/or overall comments) in the function is the most important attribute in deciding the likelihood of updating a comment. We closely examine a few of the decision trees in the forest to rationalize our results. After examining the decision trees, we find that the frequency of comment updates is higher in functions that have a large number of comments. In other words, well-commented functions remain well-commented. This finding analogously correlates to the phenomenon that buggier functions tend to remain buggier [10]. For FreeBSD, the top attributes vary from the other projects. The number of dependencies is an indicator of the coupling of a function [11]. The higher the number of dependencies, the more coupled the function is. Similarly, the number of control statements such as conditional statements, looping statements, and switch statements are indicators of the cyclomatic complexity [12][13].

Table 7: Top attributes for the characteristics of the modified function

		Attribute Name	Weight
PostgreSQL	1	Total number of comments	98
	2	Number of inner comments	88
	3	Number of dependencies	81
	4	Number of control statements	57
	5	Number of string literals	55
FreeBSD	1	Number of string literals	89
	2	Number of dependencies	87
	3	Number of parameters	86
	4	Number of control statements	70
	5	Length of function name	67
GCluster	1	Number of inner comments	98
	2	Total number of comments	88
	3	Number of string literals	81
	4	Has inner comments	57
	5	Length of function name	52
GCC	1	Total number of comments	93
	2	Number of inner comments	79
	3	Number of control statements	79
	4	Number of string literals	72
	5	Number of dependencies	58

The fact that the coupling and complexity of a function have a major impact on comment update makes intuitive sense. However, we were at first surprised to see that the number of strings literals and the length of the function name show up as the top attributes for FreeBSD. A closer analysis reveals that this is due to the nature of FreeBSD functions. In FreeBSD, functions can be grouped into two groups: internal and external functions. External functions are OS API functions that are exported and are used by other applications that run on top of FreeBSD. These functions tend to have lengthy names and are well documented in contrast to other functions. As for the appearance of the string literals in the top attributes, this is due to the fact that the code for device drivers in FreeBSD tends to have a large number of string literals and tends to be complex. Hence, there is a tendency to update comments whenever changes are done to the device drivers' functions.

For this dimension, the Random Forests algorithm has an average overall misclassification rate of around 33%, as shown in Table 9. The overall misclassification rate outperforms random guessing. Although the overall misclassification rate is rather consistent among projects, the misclassification rates for YES and NO classes vary considerably. For example, in case of PostgreSQL, the misclassification rate is 18% for the YES class and 43% for the NO class. We believe that these results are promising given the limited information (only attributes related to the modified function) used in this dimension.

5.2. Exp. #2: Characteristic of the change

We re-ran our analysis based on the attributes defined in Table 3. Our overall misclassification rate improved slightly (3% for PostgreSQL and GCluster and 6% for the GCC). Moreover, unlike the case of previous dimension, our classifier performs almost equally well in predicting the YES and NO classes. Table 8 lists the most important attributes for each project. As expected, changes to dependencies and to the control structures are two of the most important attributes. Bug fix and the number of co-changed functions are also very important attributes. These findings conform to our hypothesis; the magnitude of a change transaction and modifications to fix bugs are more likely to instigate comment updates.

The top attributes in the change information dimension are more consistent throughout the projects in comparison to characteristics of the modified function dimensions. We do not find any surprises in the top attributes. The commonality between the projects strengthens the generality of our findings in this dimension. The characteristic of the change dimension is the most influential dimension out of the three studied dimensions. Attributes in this dimension produce the smallest overall misclassification rate relative to the other two dimensions.

Table 8: Top attributes for the characteristics of the change

		Attribute Name	Weight
PostgreSQL	1	Number of changed dependencies	100
	2	Percentage of changed dependencies	86
	3	Number of co-changed functions	78
	4	Is this a bug fix	70
	5	Percentage of control statements changed	56
FreeBSD	1	Number of changed dependencies	99
	2	Percentage of changed dependencies	85
	3	Number of co-changed functions	71
	4	Number of changed control statements	69
	5	Percentage of past comments updated	60
GCluster	1	Number of changed dependencies	100
	2	Number of changed control statements	87
	3	Percentage of changed control statements	83
	4	Number of co-changed functions	60
	5	Percentage of changed dependencies	56
GCC	1	Number of changed dependencies	100
	2	Percentage of changed dependencies	88
	3	Number of changed control statements	80
	4	Number of co-changed functions	72
	5	Is this a bug fix	57

5.3. Exp. #3: Change time and code-ownership

This experiment is conducted to examine how the change time and code-ownership information help explain the comment update phenomena. We re-ran our experiment using the attributes outlined in Table 4. The results of the most important attributes are summarized in Table 10. The table shows that the developer who has modified the function, the age of the function, the number of days since the function was changed, and the weekday and month of the change are very influential factors in predicting whether the comment of their associated functions will be updated or not. For all the projects, these same five attributes repeatedly and consistently bubble up as the important attributes. Recent studies reveal that particular developers are more prone to introducing bugs than others [11]. Influenced by this information, we expected to observe similar phenomenon in regard to comment change. We anticipated that certain developers update comments more frequently than others.

Table 9: Misclassification rates for all dimensions across all projects (lower values are better)

Dimension	PostgreSQL			FreeBSD			GCluster			GCC		
	O-All	*Yes*	*No*	*O-All*	*Yes*	*No*	*O-All*	*Yes*	*No*	*O-All*	*Yes*	*No*
Function Characteristics	33.61	18.00	42.90	33.37	34.80	25.30	37.97	36.50	28.60	33.37	51.40	24.20
Change Characteristics	30.62	25.83	32.98	27.74	27.20	29.83	34.73	33.43	33.84	27.73	24.75	33.78
Time and code Ownership Characteristics	36.52	23.76	44.38	32.74	33.18	31.01	41.97	45.25	38.11	38.24	37.80	39.00
All attributes	24.01	17.21	28.22	21.42	21.52	21.01	27.60	27.87	26.32	24.89	22.61	29.59

As per our anticipation, the developer who committed the change (current author) is one of the most important attributes. It is surprising that whether the current author is the same as the initial author is not considered as an important attribute. We thought that developers are more likely to update comments when they are modifying someone else's code rather than when modifying their own code. The basis of our assumption is that one knows their code well-enough and might lack the motivation to update comments. However, there is no such finding to support our intuition.

Table 10: Top attributes for the change time and code-ownership characteristics

		Attribute Name	Weight
PostgreSQL	1	Current author	98
	2	Days since last change	82
	3	Weekday	78
	4	Month	68
	5	Age of the function	67
FreeBSD	1	Days since last change	86
	2	Current author	85
	3	Weekday	83
	4	Age of the function	82
	5	Initial author	56
GCluster	1	Days since last change	90
	2	Month	79
	3	Age of the function	78
	4	Current author	69
	5	Weekday	67
GCC	1	Current author	97
	2	Age of the function	88
	3	Weekday	75
	4	Month	70
	5	Days since last change	55

Our analysis reveals that the weekday and month are important attributes in predicting comment changes, as opposed to the quarter or year of the change. The study by Zimmerman *et al.* [14], show that on certain weekdays, namely on Fridays, developers tend to produce buggier codes. On that note, we thought that developers may be reluctant to update comment on certain weekdays due to laziness or time pressure to wrap-up a week's work. We stipulate that this can be one of the many reasons why the weekday is considered as an important attribute. Days without change (indicating project phases) and age of the function in days (indicating maturity of a function) are considered as the most influential factors in predicting the likelihood of updating a comment. We expected these attributes to have some influence but never expected them to be as influential as (and sometimes more important than) the author and time of change related attributes. Unfortunately, the misclassification rates for time and code-ownership dimension are worse than that for the characteristics of the modified function and the characteristic of the change dimensions.

5.4. Exp. #4: All attributes

In our fourth experiment, we combine all the attributes from all three dimensions to derive the best prediction model for a project. The misclassification rates achieved by combining all the attributes are statistically better than the results produced using other dimensions separately. This is clearly visible from the contrast presented in Table 9. This observation suggests that best results can be achieved by considering all dimensions rather than one dimension separately. Table 11, summarizes the top attributes for each project. We note that most of the top five attributes are from the function and change characteristic dimensions. This observation implies that the attributes in these two dimensions are more influential than attributes in the time of change and code-ownership dimension. The change in dependency, current number of comments in the function, and changes in the control statements are the most important attributes across most projects. The only exception is FreeBSD where the day of the week and the age of function are regarded as more important. Nevertheless, these results show a general trend across all projects that we studied instead of specific trends per project.

Table 11: Top attributes for each project combined

		Attribute Name	Weight
PostgreSQL	1	Percentage of changed dependencies	90
	2	Number of inner comments	77
	3	Total number of comments	73
	4	Number of changed dependencies	70
	5	Percentage of control statements changed	54
FreeBSD	1	Percentage of changed dependencies	85
	2	Weekday	64
	3	Number of inner comments	64
	4	Number of changed dependencies	63
	5	Age of the function	56
GCluster	1	Number of inner comments	97
	2	Number of changed dependencies	90
	3	Total number of comments	76
	4	Number of changed dependencies	69
	5	Number of changed control statements	59
GCC	1	Number of changed dependencies	87
	2	Number of inner comments	85
	3	Percentage of changed dependencies	84
	4	Total number of comments	81
	5	Number of changed functions in the change-list	49

5.5. Exp. #5: All projects

In our final experiment, we combine all the attributes from all projects as shown in Table 12 and also include the project name as an attribute in the data set. We then rebuild our classifiers using Random Forests and perform sensitivity analysis on the attributes to determine the most important attributes across all projects instead of on a project basis. This type of analysis can help us determine whether the comment update patterns are specific to a project or if the patterns are more general. Had the comment update patterns been project-specific, the newly added attribute for the project name would have bubbled up as one of the important attributes. This did not occur in our experiment. Hence, our findings are project independent. Moreover, the performance of our classifier improves when combining the data from all projects. This is another sign of the generality of our findings across projects. The classifier has an overall misclassification rate of 20%, a YES misclassification rate of 17%, and a NO misclassification rate of 28%. Looking at Table 12, we note that the percentage of changed dependencies is the most important attribute with the age of the function as the second most important, and the number of control statements changed and the number of changed dependencies as the top third and fourth most important attributes.

Table 12: Top Attributes for all project data

	Name of attribute	Weight
1	Percentage of changed dependencies	92
2	Age of the function	78
3	Number of changed control statements	65
4	Number of changed dependencies	63

5.6 Limitations

Most commonly used source control systems track source code as text instead of tracking it structurally as source code. Therefore, to perform our study, we used an evolutionary extractor [15] to process the historical data stored in CVS and represent it in a historical database for our analysis. The tool, C-REX, uses a set of heuristics to parse source code and to link comments to the appropriate functions. In previous work [21], we verified the high accuracy of the used heuristics. These heuristics are able to handle code with bad syntax using robust parsing techniques. Nevertheless, it is possible that these heuristics may fail sometimes since the code in the source control may have very bad syntax. However, we believe that these errors are minimal and would not statistically affect the results of our analysis. Large projects tend to have a large number of general maintenance changes such as changes to update the copyright and to indent the code. The C-REX tool uses heuristics to identify these types of changes by examining the change message attached to the change. Studying the likelihood of updating comments for these types of changes would be of little value, so we exclude these types of changes from our study. We have used different projects spanning across different disciplines, e.g., database management systems, operating systems, file management systems, clustering frameworks and compilers. Still we cannot claim that these projects are representative of all types of software projects. This is due to the fact that every type of development project has its own development processes and habits. As a result, our findings may not generalize to all types of software development projects. Furthermore, the open source nature of the studied projects and the used programming language, C, may limit the generality of our findings.

6. Conclusion and Future work

Correct and up-to-date comments aid developers in understanding the source code; wrong or outdated comments mislead developers and cause the introduction of bugs. Thus, it is important that managers monitor code comments over time. In this paper, we attempt to better understand the phenomena of updating comments. We examined the evolution of four large open source projects along several dimensions and identified the contributing factors in the likelihood of a comment being updated when its associated function is changed. We motivate our dimensions and explain the various attributes in these dimensions. We used a Random Forests classifier to understand the importance of the various attributes along the various dimensions.

Our case study shows that the characteristic of the change is the most influential dimension in explaining the comment update phenomena. Our findings are consistent across projects with the performance of our classifier improving when combining data from all the projects. The percentage of changed call dependencies and control statements, age of the modified function and the number of co-changed functions are the most important attributes in determining the likelihood of updating comments. An interesting extension of our work is to closely study the cases where our classifier predicted that a function comment should be updated and it was not updated. We would like to determine if bugs were discovered later in these functions due to out–of-date comments.

7. Acknowledgements

We thank Dr. David B. Skillicorn and Hanady Abdulsalam from Queen's University for their advice and help on using the Random Forests algorithm.

8. References

[1] M. L. V. D. Vanter. The documentary structure of source code. Information and Software Technology, 44(13), pages. 767–782, October 2002.

[2] A. Goldberg. Programmer as reader. IEEE Software, 4(5), pages. 62–70, 1987.

[3] J. L. Elshoff and M. Marcotty. Improving computer program readability to aid modification. Communications of the ACM, 25(8), pages. 512–521, 1982.

[4] T. Tenny. Program readability: Procedures versus comments. IEEE Trans. Software Eng., 14(9), pages. 1271–1279, 1988.

[5] A. Lakhotia. Understanding someone else's code: Analysis and experience. Journal of Systems and Software, 23(3), pages. 269–275, 2003.

[6] Z. M. Jiang and A. E. Hassan. Examining the Evolution of code comments in postgresql. In Proc. Int'l Workshop Mining Software Repositories, pages 179–180, 2006.

[7] A. Marcus and J. I. Maletic. Recovering documentation-to-source code traceability links using latent semantic indexing. In Proc. Int'l Conf. Software Eng., pages. 125–135, 2003.

[8] D. J. Lawrie, H. Feild, and D. Binkley. Leveraged quality assessment using information retrieval techniques. In Proc. Int'l Conf. Program Comprehension, June 2006.

[9] A. T. T. Ying, J. L. Wright, and S. Abram. Source code that talks: an exploration of eclipse task comments and their implication to repository mining. In Proc. Int'l Workshop Mining Software Repositories, pages. 1–5, 2005.

[10] J. Ratzinger, M. Pinzger and H. Gall. EQ-Mine: Predicting short-term defects for software evolution, Proceedings of the Fundamental Approaches to Software Engineering, pages. 12–26.

[11] N. Nachiappan, and T. Ball, Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study, First International Symposium on Empirical Software Engineering and Measurement, pages. 364-373, 2007.

[12] T. J. McCabe & Watson, H. Arthur. Software Complexity, Crosstalk, Journal of Defense Software Engineering, 2005.

[13] J. M. Bucknal. Algorithm for Masses, Technical report on Cyclomatic Complexity, http://www.boyet.com/Articles/CyclomaticComplexity .html, Last visited: 15 March 2008.

[14] J. Sliwerski, T. Zimmermann and A. Zeller. Don't Program on Fridays! How to Locate Fix-Inducing Changes, Proceedings of the 7th Workshop Software Reengineering, Bad Honnef, Germany, May 2005.

[15] A. E. Hassan. Mining Software Repositories to Assist Developers and Support Managers, PhD. Thesis, University of Waterloo, 2004.

[16] B. Fluri, M. Wursch and H. Gall. Do Code and Comments Co-Evolve? On the Relation between Source Code and Comment Changes. WCRE 2007, Pages. 70-79, 2007.

[17] L. Breiman. Random forests, Machine Learning, 45 (2001) (1), pages. 5–32, 2001.

[18] J. Ross Quinlan. C4.5: programs for machine learning, Morgan Kaufmann Publishers Inc., San Francisco, CA, 1993.

[19] R. Diaz-Uriarte, S. Alvarez de Andres. Variable selection from random forests: application to gene expression data.TR009, 2005.

[20] L. Breiman, Bagging Predictors, Machine Learning, 26, pages. 123-140, 1996.

[21] A. E. Hassan, Zhen Ming Jiang, and Richard C. Holt. Source versus Object Code Extraction for Recovering Software Architecture, Proceedings of WCRE 2005: Working Conference on Reverse Engineering, Pittsburgh (Carnegie Mellon), USA, November, pages. 8-11, 2005.

[22] T. M. Khoshgoftaar, E. B. Allen,W. D. Jones, and J. P. Hudepohl. Data Mining for Predictors of Software Quality. International Journal of Software Engineering and Knowledge Engineering, 9(5), 1999.