On Evaluating the Efficiency of Software Feature Development Using Algebraic Manifolds

Jay Kothari, Dmitriy Bespalov, Spiros Mancoridis, and Ali Shokoufandeh
Department of Computer Science
Drexel University
{jayk, dmitriy.bespalov, spiros, ashokouf}@drexel.edu

Abstract

*Managers are often unable to explain objectively why or when effort was misplaced during the development process. In this paper, we present a formal technique to depict the expended effort during the life-cycle of a software feature using **feature development manifolds (FDMs)**. Using the FDMs we can compute the preferred development path for a given feature. This development path includes the versions of a software feature that contributed to the final version of the feature in a positive way. The preferred development path excludes versions of the software feature that should have been skipped. Once the preferred development path is computed the amount of wasted effort can be quantified using the metric that we have developed. We demonstrate the effectiveness of our approach to compute wasted software feature development by applying our technique to two large open source software systems, Gaim and Firefox.*

1 Introduction

Managers are often unable to provide objective explanations to justify why or when effort was misplaced during the development process. They are also unable to determine formally which versions of each software feature contributed to wasted development efforts, and to what degree the effort was wasteful. The goal of our work is to provide managers a way to recognize and quantify wasteful effort that occurred during the software development process by analyzing the evolution of a software system's features.

We chose to concentrate our work at the granularity of features since they effectively characterize an application, they represent a common ground between developers and end-users, and because managers can easily describe the progress made between versions of a system in terms of the changes made to its features. From the perspective of an end-user, software can be viewed as a collection of capabilities, or features. For example, some of the end-user features of a word processor would include those functionalities that the user could invoke, such as `Print`, `Save`, and `Spell-Check`. Developers are often presented with requests to maintain, modify, or add features to an application. Managers can summarize the efforts of their teams by describing the changes made to a software's features. Following our definition of a feature, we can model a feature's architecture as a call graph, which can be extracted automatically by profiling their executions and representing them.

In this paper, we present a formal technique to recognize and quantify the wasted effort that occurred during the development of a software system by studying the evolution of its features. As part of our technique we describe an approach to embed the evolution of a software feature into Euclidean space to allow for analyses of its evolution.

In order to embed the development history of a feature's versions into Euclidean space, we must first build a similarity matrix that compares the versions of a feature. In previous work [15, 14], we employed a technique to measure the similarity of two different features. In this work, we use the same technique to determine how much a feature has changed in successive versions. We compute this pairwise similarity measure for all the versions of a feature and retain it in a feature version similarity matrix. The feature similarity matrix is in turn used to build the embedding of the feature's evolution, which we refer to as the feature development manifold (FDM).

From the FDM we can evaluate a feature's evolution in terms of expended effort. We hypothesize that good design decisions in the initial versions of a feature will provide for a regular development pattern. Efficient development occurs when incremental versions of a software feature contribute to the final version in a positive way. When an FDM determines that the amount of effort that failed to contribute to the final version was insignificant, the development can be characterized as being efficient.

We can also determine if the development of a feature was not necessarily designed well from the beginning, but evolved in a regular pattern. This would imply that the each

version of the feature contributed to the final, evolved, version.

Lastly, using an FDM we can recognize inefficient development, where versions of the feature, not only did not contribute a significant amount to the final version, but the development process was hampered due to that particular version of the feature. For example, an intermediate version of a feature makes significant changes to the feature but in the next version, those changes are rolled back to a previous version.

To recognize and quantify the wasted effort during the development of a feature, we have developed several metrics that analyze the FDMs. The metrics we describe compute the expended effort in developing a feature, the minimal effort required to obtain the final version of the feature given the initial version, and the preferred development path given the versions of the feature that are present (*i.e.*, those versions that provided satisfactory contributions to the final version of the feature).

The remainder of the paper is organized as follows. We first present an overview of our technique using a running example in Section 2. In Section 3 we describe the algorithms and technique in detail. We apply our technique to two prominent open source software systems, Firefox[1] and Gaim[2] in Section 4. Section 5 presents related work. Lastly, we conclude the paper in Section 6 with a summary of our findings, and a plan for future work.

2 Overview

In this section we present an overview of the technique we employ for computing the optimal development path of a feature and recognizing the wasted effort in its development. The first step in assessing the development of a feature is to obtain several versions of the application we are interested in examining, and delineate its features for each version.

For each version of a software system, we must execute and profile each feature to obtain a representation of the feature. Consider the Firefox web-browser and its `Open-File` feature. For each version of the Firefox, we execute the `Open-File` feature under the supervision of dynamic analysis profiling tool to obtain a representation of its architecture as a call graph.

Once the call graphs for each version of `Open-File` are obtained, we compute the pairwise similarity of its different versions by comparing their respective call graphs. Table 1 depicts the similarity matrix for 8 versions of `Open-File`. Each row and column represent a single version of the feature, and each cell represents the pairwise similarity of the call graphs of those two versions.

Using the similarity matrix and an embedding algorithm, we can visualize the similarity of the features in Euclidean

[1] `http://www.getfirefox.com`

[2] `http://sourceforge.net/projects/gaim`

	v1	v2	v3	v4	v5	v6	v7	v8
v1	1	0.88	0.81	0.76	0.75	0.75	0.71	0.7
v2		1	0.84	0.79	0.78	0.78	0.73	0.71
v3			1	0.92	0.91	0.9	0.84	0.82
v4				1	0.98	0.98	0.87	0.89
v5					1	0.99	0.87	0.91
v6						1	0.89	0.92
v7							1	0.91
v8								1

Table 1: Similarity matrix depicting the call graph similarity between 8 versions of the `Open-File` feature of Firefox.

space. The original similarity matrix, shown for feature `Open-File` in Table 1, represents a feature version interaction graph (FVIG). The FVIG fails to satisfy the triangle inequality, making it difficult to perform extensive analysis on the relationship between the features. Using an embedding algorithm [6, 11, 4] however, we can present the FVIG in a low-dimensional Euclidean space (*e.g.*, $\Re^2$) that can be visualized and analyzed (*e.g.*, compute distance between a version and a line segment connecting two versions).

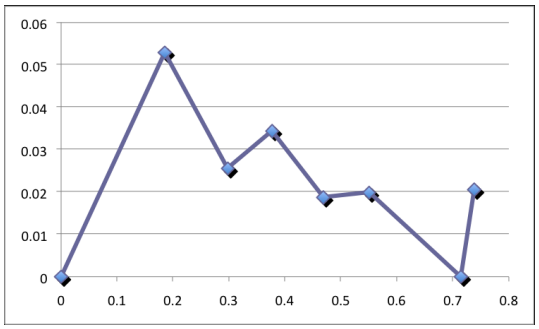

Figure 1: FDM of the `Open-File` feature of Firefox. The points represent each version of the `Open-File` feature. The edges depict the order in which the versions were created. The length of each edge represents the similarity of the two consecutive versions.

We refer to the embedding of the FVIG as well as the edges connecting consecutive versions as a feature development manifold (FDM). Figure 1 depicts the FDM of the `Open-File` feature. Each of the 8 versions in the similarity matrix is represented as a point in this FDM. The distance between any two points in the FDM is representative of the similarity between the call graphs of the versions represented by those points. The actual development path is represented by the solid edges between the points, beginning with the leftmost point (*i.e.*, the initial version) in Figure 1. These edges depict the actual development history of the feature. For example, if we begin with the leftmost point, and follow the solid edges, we will traverse the actual development path of the feature. The length of the path that we traverse is proportional to the effort expended during the development of the feature. The FDM is a representation of Euclidean space,

and the versions of features lie in that space. When examining a manifold it is important to note that the x and y axes represent a coordinate system, similarly to a map, and that the axes are not normalized to be similar to one another.

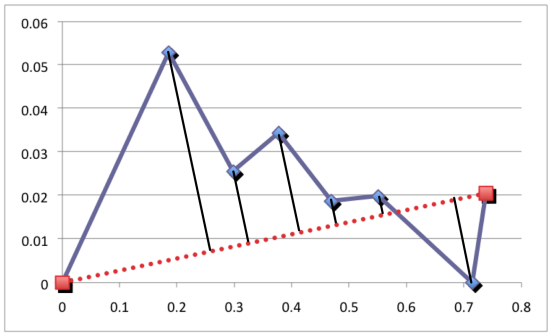

Figure 2: The dotted edges between the points depict the potential shortest path that could lead the development from the initial version to the final version of feature `Open-File`. The distance between each point and the shortest development path is the deviation of each version from the shortest path. These distances are used to compute the preferred development path.

Also using the similarity matrix, we can compute the shortest development path between the first and last version of a feature. The shortest development path indicates that given an initial version of a feature, and an evolved version of a feature (*i.e.*, the final version), if we follow that path in developing the feature, we would expend the minimal effort required to complete the feature's evolution. The basis for computing the shortest path is the similarity matrix for the feature. The shortest development path includes those versions of the feature that expend the minimal effort to go from the initial version to the final version. Figure 2 shows the shortest development path using a dotted line.

The shortest path is an intermediate result used to calculate the preferred development path, which indicates the versions of the feature that should have been avoided, and the development quality of the feature. To make these calculations, we first determine how deviant each version of the feature is from the shortest development path. Since we have embedded the FVIG into Euclidean space we are now able to make a direct comparison between the shortest development path, and the actual development path. By computing the magnitude of the normal vectors (as depicted in Figure 2 with the solid lines between the shortest path and each point not on the shortest path) we can determine how far from optimal each version of the feature was. Using statistical methods, we can estimate which versions should be included on the preferred development path and which versions are outliers and should not be included.

Figure 3 depicts the preferred development path for `Open-File` as the dashed line. Note that only two of the intermediate versions of the feature were included on the preferred path. The contributions of the other intermediate versions toward the final version of the feature were negative. In the case of `Open-File` there are 4 versions that are considered wasteful effort. Lastly, using the ratio of the length of the actual development path, and the preferred development path, we can evaluate the quality of the development of the feature. In this case, the development quality, despite having 4 wasteful versions, is above average with a value of 0.86 (on a scale of 0 to 1, where 1 denotes perfect development cycle).

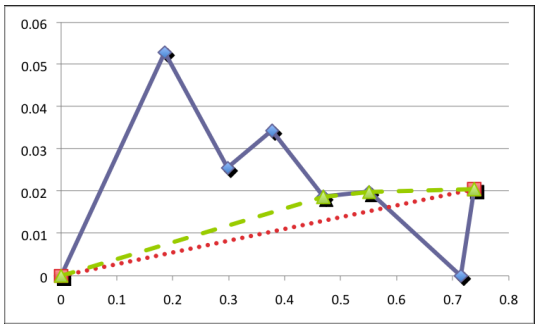

Figure 3: The dashed line depicts the preferred development path of the `Open-File` feature, avoiding those versions that did not contribute to the final version of the feature in a positive way.

3 Technique

To further clarify our technique, we present an overview of the algorithms that we use for recognizing wasted effort during the development process of software features in this section.

3.1 Feature Extraction and Representation

We begin by obtaining the versions of the application that we are going to analyze. For each version of the software, we independently identify features using release notes, documentation, use cases, or the built-in help system of the software.

Using a set of test cases designed to exercise the features of the program, under the supervision of a dynamic analysis tool, we obtain call graphs for each version of a feature. The dynamic analysis tool records the objects, functions, and variables that were exercised during the execution of the software undergoing analysis and retains this information in a call graph. For each version of the application, we obtain the call graph of every feature that was executed. If we are interested in obtaining the FDMs of specific features, we need to profile only those features for all versions of the application.

3.2 Feature Version Similarity

Once all of the feature versions have been profiled we can compare how much the feature has changed between versions. To determine how much a feature has changed between versions of the application, we compute the feature version similarity. We compute the pairwise similarity between the call graphs for a given feature and retain those values in a similarity matrix. Since call graphs are a direct representation of a feature's implementation, we consider the similarity between two call graphs to be equivalent to the similarity of the versions of the feature being represented by those call graphs.

We define the similarity of the two versions as the cardinality of the maximal clique in the computed association graph [18] of two call graphs. We refer to this measure as the association graph matching similarity measure (AGM). As formulated by Pelillo *et al.* [18], we use a formal approach for matching hierarchical structures to determine the AGM. We construct an association graph with maximal cliques that are in one-to-one correspondence with maximal subtree isomorphisms. We then formulate the clique problem as a continuous quadratic program to determine the similarity [16]. The program is then solved utilizing replicator equations, as described by Pellilo *et al.* We use this approach by building an association graph on two call graphs. Then we determine the cardinality of the maximal clique within the computed association graph and use it as the measure of similarity between the two call graphs.

The rationale behind using this measure is that if two versions of a feature have not changed significantly, their call graphs should be similar. Therefore, the dynamic call graphs that are created during the execution of two similar versions of a feature should have several vertices (functions) and edges (function call relations) in common. On the other hand, as code evolves, we begin to notice differences in the call graphs of the features affected by those code changes. Since call graphs accurately depict the implementation of software features [17, 8], measuring the similarity between the call graphs effectively measures the similarity of the underlying code.

As previously stated, we retain these pairwise similarity measures for the versions of features in a similarity matrix, one for each feature. Each row and column of a similarity matrices represent a single version of a feature. A similarity matrix represents a feature version interaction graph (FVIG), whose vertices correspond to a single version of a feature of the application, and whose weighted edges correspond to the degree of similarity between the different versions of the same feature that are incident to the edges [21, 22].

3.3 Building the Shortest Development Path

Using the similarity matrix, which represents the FVIG, we compute the shortest potential development path. To do this, we use the shortest path algorithm [13] (*i.e.*, Dijkstra's algorithm) on the original similarity matrix data. Since the relationship between feature versions in the original space does not satisfy the triangle inequality, it is possible that the shortest path between the initial version of a feature, and the final version, includes intermediate versions. In fact, this is the case in many of the features that we examined. Figure 2 depicts such a situation where an intermediate version of the `(AIM) Set-Away-Message` feature is included in the shortest development path. Again, it is important to note, that for correctness, this shortest path is computed in the original space, not the manifold space, which allows for the shortest path to include intermediate versions.

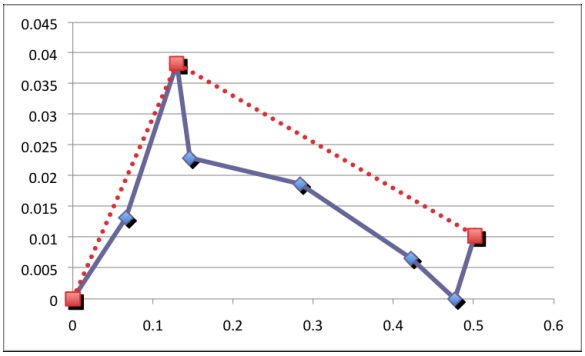

Figure 4: Feature development manifold of the `(AIM) Set-Away-Message` feature of Gaim. The points represent each version of the software system. The solid edges between the points depict the actual development path taken. The dotted edges between the points depict the potential shortest path that could lead the development from the initial version to the final version of the feature.

3.4 Building Feature Development Manifolds

Once we have a shortest development path, we want to represent it in a way that can be visualized, and use it to further assess the development of a feature. To do this we embed the FVIG into Euclidean space by building a geometrical representation of each feature's development. We investigated three embedding techniques to obtain manifolds: principal components analysis [11], multidimensional scaling [4] and spherical coding [6]. All of these approaches effectively built manifolds that could be easily analyzed and used for further examination of a feature's development. A manifold is a series of piece-wise connected trajectories where points correspond to versions of a feature, and the edges between

the points correspond to the development path taken. Embedding enables us to measure the distance between a version and the optimal development path.

After examining embedding distortions (i.e., observed error caused by reduction in dimensionality), we concluded that all three methods yield comparable results in terms of preservation of the original distance measures, and that it was sufficient to use any of them. We chose to use spherical coding approach, since it produces manifolds in $\Re^2$ and the resulting manifolds are simpler to interpret and visualize [6]. It is important to note that all of the computations related to our method that occur in the embedded space (*i.e.* $\Re^2$) are subject to error due to the distortion that is introduced by embedding process. However, we found that the embedding distortion, which can be measured as relative difference between the original and Euclidean distance values for all pairs of versions within a feature, was relatively low – less than 10% on average across all features.

3.5 Building the Preferred Development Path

Now that we have a manifold we can compute the preferred development path. We compute the normal distance (deviation) of each incremental version of the feature, to the shortest development path in Euclidean space. Once these deviation values are computed, they are normalized by the length of the shortest development path which is also computed in the manifold space. Finally, the preferred development path is constructed by selecting versions that are not considered to be statistical outliers in terms of their deviations from the shortest path. In other words, a version is added to preferred development path, if the normalized distance between this version and the shortest path is less than a wasted effort threshold value. Versions that are not included in the preferred development path are identified as wasted effort versions (*i.e.*, effort to develop these versions is said to be wasteful).

The wasted effort threshold value is computed by examining the distribution of normalized deviation values for all versions of features within in the system. In order to prune statistical outliers (*i.e.*, versions that contributed to the wasted development effort) from the preferred development path, we chose $1 \times \sigma$ or $2 \times \sigma$ (where σ denotes standard deviation) as the wasted effort threshold.

Table 2 shows the number of wasteful versions for Gaim and Firefox for $1 \times \sigma$ to $5 \times \sigma$. The greater the deviation allowed, the more lax the criteria for a version to be wasteful. Figure 5 presents the preferred development path for a sample feature of Firefox that was constructed using wasted effort thresholds set to $1 \times \sigma$ and $2 \times \sigma$.

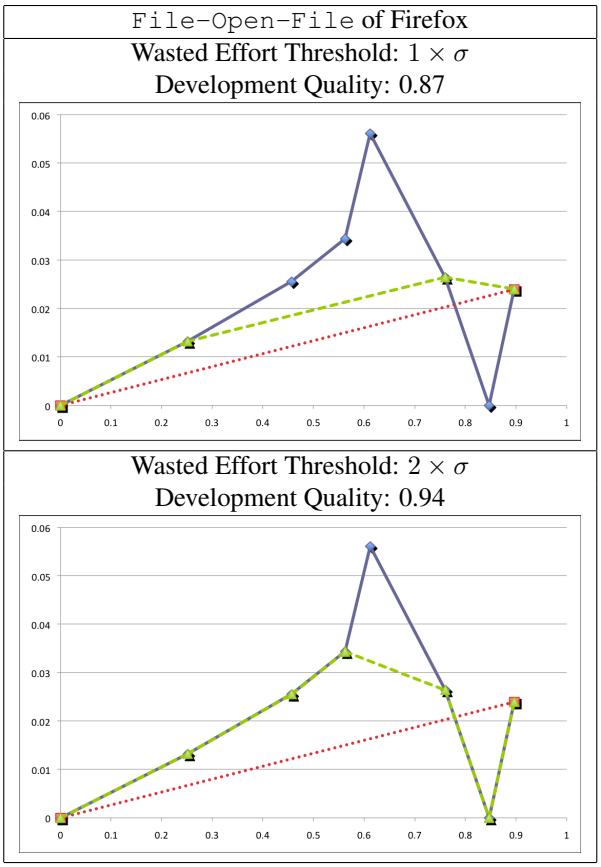

Figure 5: Actual (solid line), shortest (dotted line) and preferred (dashed line) development paths under varying wasted effort thresholds. Development quality values are also provided.

3.6 Quality of Development Process

Once the preferred development paths are constructed, we measure the development quality of each feature. This quality is defined as the ratio of the sum of distances between consecutive versions of the preferred development path normalized by the sum of distances between consecutive versions of the actual development path. The original FVIG distance values are used in this computation. Values of development quality range between 0 and 1, where a development quality value of 1 implies that no effort was wasted during the development. Figure 5 shows the preferred development paths along with measures of development quality associated with these paths.

4 Case Study

Using the technique we described in Sections 2 and 3, we analyze the development of two prominent open-source systems: Firefox and Gaim. Gaim is an Internet chat application supporting multiple protocols and Firefox is a web-browser based on the Mozilla engine.

Gaim		Firefox	
$\sigma = 0.075$		$\sigma = 0.02$	
Wasted Effort Threshold	Wasted Effort Versions Count	Wasted Effort Threshold	Wasted Effort Versions Count
$1 \times \sigma$	31	$1 \times \sigma$	26
$2 \times \sigma$	15	$2 \times \sigma$	11
$3 \times \sigma$	7	$3 \times \sigma$	4
$4 \times \sigma$	3	$4 \times \sigma$	1
$5 \times \sigma$	0	$5 \times \sigma$	0
Total versions: 144		Total versions: 72	

Table 2: Wasted effort version counts for Gaim and Firefox.

We began by obtaining stable versions of the applications from the respective repositories. Consulting the release documentation, we identified end-user features for each version of the system, and executed those features under the supervision of a dynamic analysis tool in order to obtain call graphs.

Once we obtained call graphs for each version of a feature, we computed the pairwise similarity between versions of the feature. This process was repeated for each feature of the system. The pairwise similarities were stored in a similarity matrix that was used to compute the shortest development path metric as well as the feature development manifold. Using the feature development manifold and the shortest development path we were then able to assess the quality of development and determine those versions of the software that were wasteful as described in Section 3.

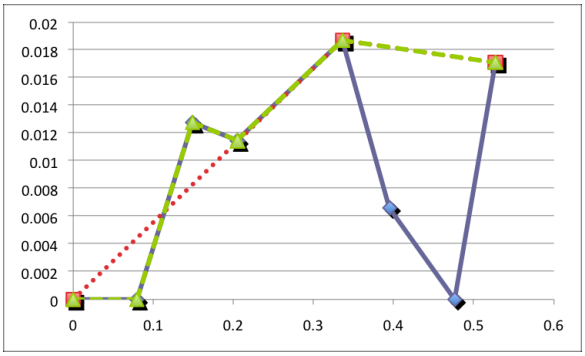

Figure 6: Feature development manifold for `Bookmark` Feature. Development paths are as follows: actual (solid line), shortest (dotted line) and preferred (dashed line) development paths.

4.1 Firefox

The first software system that we applied our technique to was the Mozilla-based web-browser, Firefox. We obtained 8 versions of the Firefox application from its source code repository, each version separated by approximately 3 to 6 months. Each version consisted of approximately 50 features. For each version of the application, we listed all of its features, and obtained call graphs of their execution. The call graphs were then used to construct a feature version similarity matrix for each feature, which in turn was used to construct a feature development manifold for each feature.

Once the feature version similarity matrices and feature development manifolds have been computed, we can assess the quality of the software's development. Table 3 lists the number of wasted versions for $1 \times \sigma$ and $2 \times \sigma$ thresholds, as well as the quality of development for a sample of Firefox's features. For both thresholds the `Go-To`, `Open-In-New-Window`, and `Bookmark-Open` had the most number of versions that did not provide satisfactory contribution to the final version of the feature. `Go-To` had the worst development Quality with 0.67. Features related to clicking links and bookmarks, and `File-Open-Location` had the best development with the fewest number of wasteful versions.

Figure 6 depicts the actual development path (solid edges), shortest development path (dotted edges), and preferred development path (dashed edges) for the `Bookmark` feature of Firefox. The `Bookmark` feature exhibited good development quality with 0.9 ($1 \times \sigma$ threshold) and 1.0 ($2 \times \sigma$ threshold) for the development quality metric, and 2 ($1 \times \sigma$ threshold) and 0 ($2 \times \sigma$ threshold) wasteful versions.

It is important to note that, when computing the shortest development path using the original similarity matrix, that 2 versions actually fall on this path. In other words, these versions were actually very well developed and provided significant contribution to the ultimate version of the feature. The versions that failed to be on the shortest path, did not deviate greatly from it, and in fact all but 1 version, in the case of a $2 \times \sigma$ threshold, were chosen to be on the preferred development path.

In other words, the `Bookmark` feature of Firefox was well developed. The architecture of the feature had not changed dramatically between the first and last version, and the incremental changes made to the feature contributed to the final, evolved, version of the feature. The one statistical outlier that was not included on the development path proved to be a change that was not useful, and therefore not contributory in the final version. Examining the manifolds, we can see that the version following the one that was not included, was quickly corrected to be more similar to the last version

Firefox				
Feature Name	Outliers Threshold: $1 \times \sigma$		Outliers Threshold: $2 \times \sigma$	
	Wasted Versions	Development Quality	Wasted Versions	Development Quality
ft1: File-Open-Location	1	0.97	0	1.0
ft11: LClick-Link	2	0.92	1	0.94
ft14: Bookmark	2	0.9	0	1.0
ft15: Bookmark-Link	1	0.95	0	1.0
ft3: Open-In-New-Window	4	0.76	2	0.87
ft4: Go-To	5	0.67	3	0.78
ft5: File-Open-File	4	0.86	1	0.94
ft6: Bookmark-Open	4	0.73	3	0.83
ft7: Bookmark-Click	3	0.93	1	0.97

Table 3: Wasted versions and Development Quality data for Firefox.

that was included on the preferred development path.

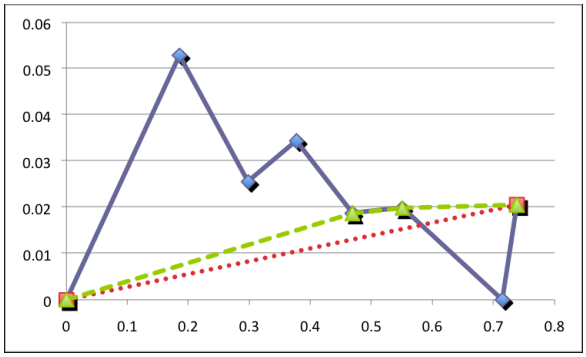

Figure 7: Feature development manifold of `Open-In-New-Window` Feature. Development paths are as follows: actual (solid line), shortest (dotted line) and preferred (dashed line) development paths.

Unlike the `Bookmark` feature, which showed a good development quality, the `Open-In-New-Window` feature had a relatively poor development quality of 0.76 ($(1 \times \sigma$ threshold). For a $1 \times \sigma$ threshold, 4 of the versions of the feature were considered wasteful. None of the intermediate versions of the feature were used on the shortest development path, and only 2 where used on the preferred development path. In other words, 4 of the 6 intermediate versions of the feature were wasteful. The development paths for the `Open-In-New-Window` feature can be seen in Figure 7.

Overall, the development of Firefox can be considered efficient. Although there are a several versions exhibiting wasted effort, as can be seen in Table 2, the deviations of those versions from the preferred path of development was not significant.

4.2 Gaim

The second software system that we applied our technique to was the open-source instant messaging client, Gaim which supports multiple protocols and platforms. For the purposes of this case study we only include features for AIM, MSN, and YAHOO. Similar to Firefox, we obtained 8 versions of Gaim from the source code repository listed on its website, consisting of approximately 40 features each. We profiled each feature in each version of the application to obtain the call graphs of those features. Once the call graphs were obtained, we computed similarity matrices for each feature depicting the relationship between all of its versions. The similarity matrices were used to build feature development manifolds.

Using the similarity matrices and their respective feature development manifolds we can assess the development quality of the software system's features. Table 4 lists the quality measures for a sample of features from the Gaim software system for $1 \times \sigma$ and $2 \times \sigma$ thresholds. The number of wasteful versions is listed for each threshold as well as the development quality metric. The features of Gaim, regardless of threshold, seem to have either a very high development quality (*e.g.*, few of the versions of the feature were wasteful and the majority were included on the preferred development path) or they exhibit a very poor development quality (*e.g.*, more than half of the intermediate versions were considered wasteful and the development quality metrics were low in magnitude).

For example, the `(AIM) Set-Away-Message`, `(MSN) Login`, `(MSN) Send-Message`, and `(AIM) Login` features all exhibit excellent development quality at 0.9 or greater and very few wasteful versions, if any. On the other hand the `(AIM) Send-File`, `(AIM) Receive-File`, and (AIM) Direct-Message features showed poor development quality with values of 0.64 to 0.66 and over two-thirds of intermediate versions being wasteful.

Figure 8 depicts the actual development path (solid edges), shortest development path (dotted edges), and preferred development path (dashed edges) for the `(AIM) Add-Buddy` feature of Gaim. The `Add-Buddy` feature

Gaim				
	Outliers Threshold: $1 \times \sigma$		Outliers Threshold: $2 \times \sigma$	
Feature Name	Wasted Versions	Development Quality	Wasted Versions	Development Quality
ft10: (AIM) Set-Away-Message	0	1.0	0	1.0
ft12: (AIM) Add-Buddy	3	0.7	2	0.88
ft15: (MSN) Login	0	1.0	0	1.0
ft16: (MSN) Send-Message	2	0.94	0	1.0
ft17: (MSN) Add-Buddy	2	0.59	1	0.88
ft2: (AIM) Login	0	1.0	0	1.0
ft3: (AIM) Send-Message	0	1.0	0	1.0
ft33: (YAHOO) Login	0	1.0	0	1.0
ft34: (YAHOO) Receive-Message	3	0.88	0	1.0
ft35: (YAHOO) Send-Message	2	0.76	1	0.93
ft36: (YAHOO) Add-Buddy	4	0.51	1	0.82
ft4: (AIM) Receive-Message	0	1.0	0	1.0
ft5: (AIM) Send-File	4	0.64	3	0.75
ft6: (AIM) Receive-File	4	0.66	2	0.87
ft7: (AIM) Direct-Message	5	0.64	3	0.89
ft8: View-Log	0	1.0	0	1.0
ft9: (AIM) Get-Info	1	0.8	1	0.8

Table 4: Wasted versions and Development Quality data for Gaim.

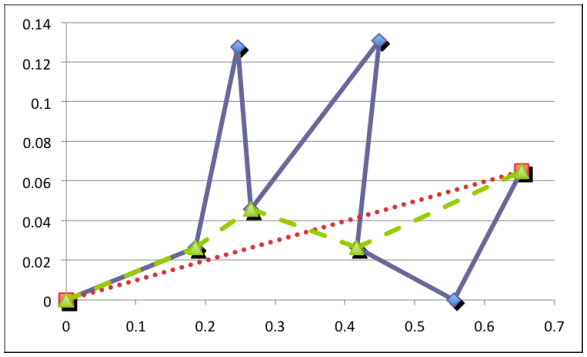

Figure 8: Feature development manifold of `(AIM) Add-Buddy` Feature. Development paths are as follows: actual (solid line), shortest (dotted line) and preferred (dashed line) development paths.

exhibited very poor development quality with 0.51 ($1 \times \sigma$ threshold) and for the development quality metric, and 4 ($1 \times \sigma$ threshold) wasteful versions. Allowing the threshold to $2 \times \sigma$ improves the development quality to 0.82.

It is easy to justify why in the case of Gaim, a low development quality does not reflect poorly on the developers, but is actually a side effect of the style of development. The features of Gaim are built by reverse engineering the actual protocols they support. In the case of the `Add-Buddy` feature, the Gaim developers were likely attempting to adjust to a change in the protocol of AIM. Examining the feature development manifold, it is very that at two instances, the developers made changes to the feature that were nearly entirely dropped in consecutive versions. This would be consistent with the rationale of implementing a reverse engineered protocol and fining flaws with it.

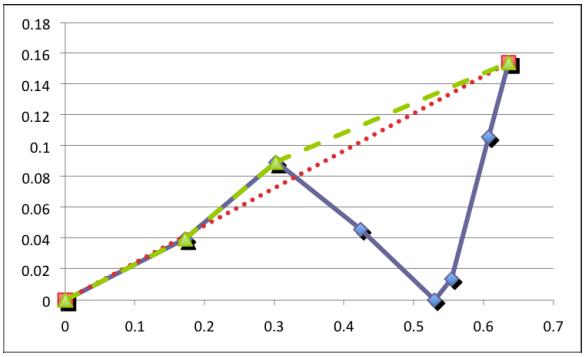

Figure 9: Feature Development manifold of `(AIM) Receive-File` Feature. Development paths are as follows: actual (solid line), shortest (dotted line) and preferred (dashed line) development paths.

The `(AIM) Receive-file` feature of Gaim shown in Figure 9, also shows poor development quality at 0.66 and 4 wasted intermediate versions at a $1 \times \sigma$ threshold. Increasing to a $2 \times \sigma$ threshold improves the development quality to 0.87, but there are still 2 wasted intermediate versions of the feature. In the case of this particular feature, one can see upon examining the feature development manifold, that a great deal of effort was placed in the second half of the feature's life-cycle, but the code changes made were more wasteful than contributory to the final version of the feature.

In other words, versions 4, 5, 6, and 7 of the `(AIM) Receive-file` feature should have not been released. There was a significant effort expended in those versions, but that effort was misplaced. Based on the FDM, we can see that although some of the effort made in the development contributed to the final version, there was a great deal of effort that did not.

Overall, the Gaim system exhibits a lower development quality than Firefox. It is important to note that this does not necessarily reflect negatively on the quality of the software, but rather on its ability to keep up with the changes in the various protocols.

5 Related Work

The goal of our work is to assess the quality of a feature's development, and recognize those versions of a feature that contributed to its evolution in a positive way.

The first step in our approach is to profile all versions of the features we are analyzing. To do so, we execute them under the supervision of a dynamic analysis tool to obtain call graphs for each feature version. Many approaches to feature profiling follow the model set forth by Wilde and Scully [23] in which they use test cases to find code that is associated to a feature. Similarly, Wong *et al.* [24] identify sets of *invoking* and *excluding* of test cases for a feature that will respectively invoke or not invoke that feature. Both sets of test cases are executed, and the two execution slices obtained are then compared to isolate the section of code pertaining to the feature. Alternatively, it is possible to use static analysis to associate features to source code or to use a combination of static and dynamic analyses to find the source code implementing the features of an application [7, 8].

Once call graphs for every version of each feature in the software system have been obtained, the similarity of the different versions of the features can be compared. We use a technique developed by Pelillo *et al.* [18] for matching hierarchically organized graphs to compare call graphs and measure the pairwise similarity between versions of features. An alternative approach for comparing feature versions in a software system based on the information obtained used dynamic analysis is presented by Antoniol and his colleagues [2] where they create models of features using both static and dynamic analyses and compare them using model transformations. Further alternatives to comparing features include using latent semantic analysis and vectors of semantic terms to characterize features, and comparing those characterizations [19].

Using pairwise similarities between the versions of a software feature, we were then able to compute embedding depicting the evolution of the feature. Fischer and Gall also use embedding techniques to depict feature evolution. In their work, they apply multidimensional scaling to depict the proximity of problem and modification reports. Their goal is to visualize the logical coupling of seemingly unrelated files over time to illustrate feature evolution, and potential architecture deterioration.

Ultimately, we use the embeddings to evaluate the actual cost (in terms of expended effort) Several other techniques exist for the estimation of cost in software development [5, 9, 10]. Briand *et al.* discuss the merits of four different cost estimation techniques: ordinary least squares regression, stepwise ANOVA, CART, and analogy. They found that of those approaches, the differences in their effectiveness was minimal, except for the analogy based approach which proved to be the least accurate.

Kemerer [12] presents a comparison of the four most popular algorithmic models to estimate software costs. These are SLIM, COCOMO, Function Points, and ESTIMACS. The SLIM method of estimation uses a source lines of code estimate for a project's overall size, then employs Raleigh curve modeling to produce effort estimates. The manager using the tool can influence the shape of the curve to accommodate his interests. COCOMO [3], the Constructive Cost Model, is a model that predicts the effort and duration of a project based on inputs that are related to cost drivers, size, and other criteria that affect productivity. Function points [1] was designed as an improved alternative to estimating source lines of code. The metric captures the number of input transaction types and unique reports. It is also able to provide estimates from the requirements definition document, a significant advantage over other techniques. The last major approach is ESTIMACS [20], a commercial tool that uses function point-like metrics to estimate the cost of development.

Unlike these approaches, our technique examines, from the source level, how the effort has been displaced based on the change of the architecture of features. We employ a formal algebraic technique that is objective, and does not rely on source code metrics or estimation, but rather on an evaluation of the actual contents of a source code repository. When assessing the development quality, we can adjust the threshold parameters if a manager would be more lenient or strict appropriately.

6 Conclusions and Future Work

In this work, we present a formal approach to evaluating the efficiency of software development at the feature level. Our work contributes to the state-of-the-art by creating a technique that allows engineers and managers to evaluate the development of a feature by:

- characterizing the similarity of different versions of a given feature based on the similarity of their underlying implementation (i.e., call-graphs);
- computing the shortest development path of a feature based on its underlying implementation;

- determining versions of a feature that contributed to its evolution in a positive way;
- recognizing those versions that reduced the overall development quality implying they should have been skipped; and
- formally quantifying the efficiency of a feature's development.

We applied our technique to the features of Gaim and Firefox to identify versions that were wasteful in the overall development of their features, as well as evaluate the development quality of each feature. The advantage of our approach over other alternatives is that the results are derived directly from the underlying architecture of features. We assess the evolution of a feature's architecture to obtain objective evaluation of development quality using a formal algebraic technique.

Our plan is to continue working on several aspects of this paper. We would also like to apply this technique to corporate projects that follow a strict development policy, with the explicit goal of efficient development. Open-source development is not necessarily concerned with meeting budget guidelines, or deadlines whereas corporations are trying to minimize the wasted effort in their overall development.

References

[1] A. Albrecht and J. Gaffney Jr. Software function, source lines of code, and development effort prediction: A software science validation. *IEEE Transactions on Software Engineering*, 9(6):639–647, 1983.

[2] G. Antoniol and Y. Gueheneuc. Feature Identification: A Novel Approach and a Case Study. *Proceedings IEEE International Conference on Software Maintenance (ICSM 2005)*, pages 357–366.

[3] B. Boehm, B. Clark, E. Horowitz, C. Westland, R. Madachy, and R. Selby. Cost models for future software life cycle processes: COCOMO 2.0. *Annals of Software Engineering*, 1(1):57–94, 1995.

[4] I. Borg and P. Groenen. *Modern Multidimensional Scaling: Theory and Applications*. Springer-Verlag New York, 1997.

[5] L. Briand, K. El Emam, D. Surmann, I. Wieczorek, and K. Maxwell. An assessment and comparison of common software cost estimation modeling techniques. *Proceedings of the 21st international conference on Software engineering*, pages 313–322, 1999.

[6] M. F. Demirci, A. Shokoufandeh, Y. Keselman, S. Dickinson, and L. Bretzner. Many-to-many feature matching using spherical coding of directed graphs. In *The 8th European Conference on Computer Vision*, pages 322–335, 2004.

[7] T. Eisenbarth, R. Koschke, and D. Simon. Aiding program comprehension by static and dynamic feature analysis. In *Proceedings of The 17th IEEE International Conference on Software Maintenance (ICSM 2001)*, pages 602–611, 2001.

[8] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 29(3):210–224, 2003.

[9] R. Fairley. Recent Advances In Software Estimation Techniques. *Software Engineering, 1992. International Conference on*, pages 382–391, 1992.

[10] Q. Hu, R. Plant, and D. Hertz. Software cost estimation using economic production models. *Journal of Management Information Systems*, 15(1):143–163, 1998.

[11] I. T. Jolliffe. *Principal Component Analysis, Series: Springer Series in Statistics*. Springer, 2002.

[12] C. Kemerer. An empirical validation of software cost estimation models. *Communications of the ACM*, 30(5):416–429, 1987.

[13] D. Knuth. A Generalization of Dijkstra's Algorithm. *Information Processing Letters*, 6(1):1–5, 1977.

[14] J. Kothari, T. Denton, S. Mancoridis, and A. Shokoufandeh. On Computing the Canonical Features of Software Systems. *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE 2006)-Volume 00*, pages 93–102, 2006.

[15] J. Kothari, T. Denton, A. Shokoufandeh, and S. Mancoridis. Reducing Program Comprehension Effort in Evolving Software by Recognizing Feature Implementation Convergence. *Proceedings of the 15th IEEE International Conference on Program Comprehension*, pages 17–26, 2007.

[16] T. Motzkin and E. Straus. Maxima for graphs and a new proof of theorem of turan. *Canadian Journal of Mathematics*, 17:533–540, 1965.

[17] G. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: bridging the gap between source and high-level models. *Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, pages 18–28, 1995.

[18] M. Pelillo, K. Siddiqi, and S. W. Zucker. Matching hierarchical structures using association graphs. *Lecture Notes in Computer Science*, 1407, 1998.

[19] D. Poshyvanyk, Y. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich. Combining Probabilistic Ranking and Latent Semantic Indexing for Feature Identification. *Proceedings of 14th IEEE International Conference on Program Comprehension (ICPC'06), Athens, Greece*, pages 137–148, 2006.

[20] H. Rubin. Macro-Estimation of Software Development Parameters: The ESTIMACS System. *SOFTFAIR Conference on Software Development Tools, Techniques and Alternatives, Arlington, IEEE Press, New York, NY*, pages 4–16, 1983.

[21] M. Salah and S. Mancoridis. A hierarchy of dynamic software views: from object-interactions to feature-interacions. In *Proceedings of The 20th IEEE International Conference on Software Maintenance (ICSM 2004)*, 2004.

[22] M. Salah, S. Mancoridis, G. Antoniol, and M. D. Penta. Towards employing use-cases and dynamic analysis to comprehend mozilla. In *Proceedings of the 21st International Conference on Software Maintenance (ICSM 2005, Budapest,September 25-30)*, pages 639–642, 2005.

[23] N. Wilde and M. Scully. Software Reconnaissance: Mapping Program Features to Code. *Journal of Software Maintenance: Research and Practice*, 7(1):49–62, 1995.

[24] W. Wong, S. Gokhale, J. Horgan, and K. Trivedi. Locating program features using execution slices. *Proceedings of the 1999 IEEE Symposium on Application-Specific Systems and Software Engineering and Technology*, page 194, 1999.