Mobile and Desktop Release Cycles

Kendall Bailey
Oregon State University

Hongyan Yi
Oregon State University

1. INTRODUCTION

It has been believed by the software community that mobile application release cycles are shorter than desktop release cycles. The rapid rise of mobile technology has changed expectations from consumers and added pressure to developers to take advantage of new technology and quickly push it to the market. This fast-paced business model gels with agile and extreme programming methodologies.

Companies know that mobile applications now represent a large segment of the market and often provide both mobile and desktop versions of their software to reach the most consumers possible. The growing group of applications that belong to both mobile and desktop applications have not yet been looked at as single distinct group. In this paper we analyze some of the properties of these application "siblings."

Siblings: We define **sibling applications** as applications that have the same name and are released by the same company for different platforms. For this research sibling applications refers to a mobile application (iOS or Android) and desktop application (OSX, Windows Vista/7/8, or Linux) pairing. An example of sibling applications would be OmniGraffle, which maintains both a iOS and OSX application of the same name with different version histories, though often version histories of sibling applications are not separated.

TODO: Change from speculative to concrete when done In this paper we our goal is to collect the data from 12 desktop applications, 12 mobile applications, and 12 sibling applications. If we maintain our minimum number of releases at 3 releases, this will give us at least 288 releases to use for analysis.

Our analysis focuses on the length of release cycles of the different types of applications and the contents of the releases.

TODO: Overview of results

2. PROCEDURE

We collected the release notes or change logs of suitable applications that consist of a healthy mix of mobile applications, desktop applications, or mobile applications with a desktop applications sibling. Suitable applications will have:

1. A version history length of at least 3 releases
2. The date of each release
3. Release notes that mention bugs, features, enhancements, etc.

To collect the data we grab text from any release histories found on the application developers' website or application distributors' documentation.

As the granularity of change, we chose release histories because it allowed for more similar analysis of statements in both open source and proprietary applications. Often, open source applications allow users to view the open and closed issues across the history of the applications. The bugs/features/enhancements are reported by and communicated to users with high technical skills. However because releases can span many commits, release notes are often more general than the notes from each commit.

Proprietary applications tend to be more vague in their descriptions of the contents of each release to prevent lost of customer confidence and to protect their intellectual property. Proprietary applications also appeal to a wider audience who have a collectively lower level of technological understanding which also affects the level of the language chosen.

In choosing our corpus we tried to not focus on a certain genre of application while still choosing popular applications from the top sellers/downloads lists provided by distributors. We also tried to maintain a balance of Android and iOS applications. On the occasion that an application had release notes for multiple platforms and each platform had differentiated release notes, only one platform's application was chosen to add to the corpus. Table 1 shows the contents of our corpus and how we classified each application.

Using the data from our corpus, we first analyzed the release cycles of the different categories of applications. This was done using each release in the corpus listed in Table 1.

Mobile Applications		
Afterlight$^{\bullet}$	**CNN**$^{\bullet}$	Clash of Clans$^{\bullet}$
Credit Karma$^{\bullet}$	Crittercisms†	**Dictionary**$^{\bullet}$
Evernote Food$^{\bullet}$	**Facebook Messenger**$^{\bullet}$	**Heads Up**$^{\bullet}$
Instagram†	**Instapaper**$^{\bullet}$	**Luminosity**$^{\bullet}$
New Words with Friends†	**Pandora**$^{\bullet}$	Plague Inc$^{\bullet}$
Run Keeper$^{\bullet}$	**Sleep Cycle Alarm**$^{\bullet}$	**Songza**$^{\bullet}$
Wish$^{\bullet}$	**Youtube**$^{\bullet}$	

Desktop Applications	
Acorn*	Aptana$^{\square\ \star\ \diamond}$
Charles$^{\square\ \star\ \diamond}$	GameMaker$^{\square}$
Kaleidoscope*	Kerbal Space Program$^{\square\ \star\ \diamond}$
NDepend$^{\square}$	**NetNewsWire***
Tableau$^{\square}$	VM Ware Fusion*
Valgrind$^{\diamond}$	**Versions***

Sibling Applications	
1Password$^{\star\ \bullet}$	**Bastion**$^{\square\ \star\ \diamond\ \bullet}$
Covenant Eyes$^{\square\ \dagger}$	**Creative Cloud**$^{\square\ \star\ \bullet}$
Dashlane$^{\star\ \square\ \bullet}$	OmniGraffle$^{\star\ \bullet}$
Raptr$^{\square\ \bullet}$	**SplashID**$^{\star\ \dagger}$
Spotify$^{\square\ \star\ \diamond\ \dagger}$	

Table 1: Our corpus divided into the three categories: Mobile, Desktop, and Sibling applications. Bolded application names indicate we classified the release notes of those applications. The superscript symbols represent the platform from which each of the release notes was taken and are as follows: $^{\bullet}$ indicates iOS applications, † indicates Android applications, $^{\square}$ indicates Windows applications, * indicates OSX applications, $^{\diamond}$ indicates Linux applications. For desktop applicaitons, if the source of the release notes did not indicate to which platform the release notes specifically related all possible platforms are marked.

Then, we classified each of the statements within the release notes to fall into one of six categories: bugs, enhancements, features, non-functional requirements, gratitude, and feedback/contact. Statements which describe two or more changes are split into atomic parts for more accurate classification. **TODO:** Add historgram results of common word usages for each category

Bugs: The bugs category describes statements in release notes that correct flaws within the application. An example of a statement belonging in the bugs category is "Fixed a crash when closing documents while there was an active text insertion point." Statements belonging to the bugs category most often contain the words "...", "...", and "..."

Enhacements: The enhancements category describes statements in release notes that build on previously introduced features or improve performance, but do not correct overt flaws like those in the bugs category. An example of a statement belonging in the enhancement category is "Improved the performance of the Stroke and Fill layer styles." Statements belonging to the enhancements category most often contain the words "...", "...", and "..."

Features: The features category describes statements in release notes that introduce new functionality or properties to an application. An example of a statement belonging in the enhancement category is "Added preference setting for prompting to save session information on close." Statements belonging to the features category most often contain the words "...", "...", and "..."

Non-Functional Requirements: The non-functional requirements category describes statements in release notes that do not fix or enhance old functionalities or introduce new functionality to the application. This category includes all changes to help documentation, licensing, EULAs, typo fixes, and translations. An example of a statement belonging in the non-functional requirements category is "We polished up our font so all the letter tiles should come back sharp." Statements belonging to the non-functional requirements category most often contain the words "...", "...", and "..."

Gratitude: The gratitude category describes statements in release notes that thank or acknowledge users of the application. This type of release note statement commonly appears in mobile application release notes and more **TODO:** infrequently | never in desktop applications. Instances of this category often appear along side instances of the feedback-/contact category. An example of a statement belonging in the advertisements category is "Thanks so much, everyone, for playing Words with Friends." Statements belonging to the advertisements category most often contain the words "...", "...", and "..."

Feedback/Contact : The feedback/contact category describes statements in release notes that solicit users to provide feedback, acknowledge changes based on user feedback, or ways to get in touch with the developers. This type of release note statement appears in mobile application release notes and **TODO:** infrequently | never in desktop applications. Some examples of statements belonging in the feedback/contact category are "keep the feedback coming!", "We've listened to your feedback and addressed several concerns with this update", and "If you have a suggestion, you can either leave us a review, or contact us through afterglow-app.com. We'd love to hear your suggestions for Afterglow version 1.3, releasing next month!". Statements belonging to the advertisements category most often contain the words "...", "...", and "..."

There is an additional category that we do not analyze that contains titles, headers, punctuation, other organizational type text, or duplications that is not considered in our analysis.

2.1 Classification of Statements

The classification of the statements will be carried out using qualitative thematic coding. Each of the classifications was done manually by the authors using the criteria detailed above. The authors validated their classifications by reaching 79% agreement on 20% of corpus. Here the corpus refers to individual statements within the release notes, which are taken from the bolded applications in Table 1. The authors took a random sample across the corpus of statements and both classified the same data to compare for agreement.

Classification was carried out in two phases, the first broke the data into the categories: bug, feature, enhancement, and miscellaneous. The second phase saw the miscellaneous category divided into the non-functional, gratitude, and feedback/contact categories based on general themes that emerged.

We implemented a tool to help us efficiently and atomically classify each statement. The tool also converted the data from its raw form, to a standardized format broken up by application and release. Once uniformly contained, the tool would display a single statement from a chosen release and allow the authors to tag it with a single classification. If multiple classifications applied to a statement, the authors would split the statement into two single-tagged lines. For example, "Bug fixes and improvements" splits into "Bug fixes and" placed in the Bug category and improvements" placed in the Enhancements category. Our motivation for single-tagged lines considered potential future work using the classifications to create dictionaries for automatic classification, which would be easier if given distinct word banks.

2.2 Storage of Statements

After classifying each statement, we stored the data in json format to maintain the most information about the data. We chose json because it is a widely used data storage format for which many programming languages have support. Additionally, if other researchers use our corpus, storing the data in json increases the ease for other researchers to handle our data.

The data is organized by version (see Figure 1). Each version contains:

- the name of the application
- a version number

  this is used to quickly corroborate contents with the original release notes
- the release date

  this allows for the calculation of release intervals
- a string representing all of the statements in the version

  a raw version of the release notes is retained in case the statements are split when our tool breaks the text into statements. We are able to look back at the context and properly classify the statement.
- the type of the application

```
{
   "VersionNumber":"1.0",
   "ReleaseDate":"10-10-2010",
   "ReleaseContents":"Fixed a crash when closing
     documents while there was an active text
     insertion point.\nImproved the performance
     of the Stroke and Fill layer styles.",
   "ApplicationName":"Example App",
   "ApplicationType":{
      "str_type":"Desktop",
      "int_type":0
   },
   "flag":false,
   "EntryList":[
      {
  "entry":"Improved the performance of
            the Stroke and Fill layer styles.",
        "classification":{
            "str_type":"Junk",
            "int_type":4
         },
         "VersionNumber":"2.6",
         "ReleaseDate":"25/09/2014",
         "ApplicationName":"Afterlight",
         "ApplicationType":{
            "str_type":"Enhancement",
            "int_type":2
         },
         "datePattern":[
            ""
         ],
         "flag":false,
         "split":false,
         "merged":false,
         "original_text":"Here is what's new in Afterlight
      },

   ]
}
```

Figure 1: An example of the structure of our json. This is the information we store for each version of an application.

  this allows for classification based on whether the application is a desktop, mobile, or sibling application
- flag

  Allows researchers to identify atypical releases for later review
- a list of statements

  The contents of the release notes divided into classified, atomic statments. These entries contain much of the information that versions do because it allows us to select individual statements and determine their origins.

We stored each of the statements with a classification of the type of statement and the text of the atomic statement. We chose to maintain the original text to leave open the poten-

3. ANALYSIS

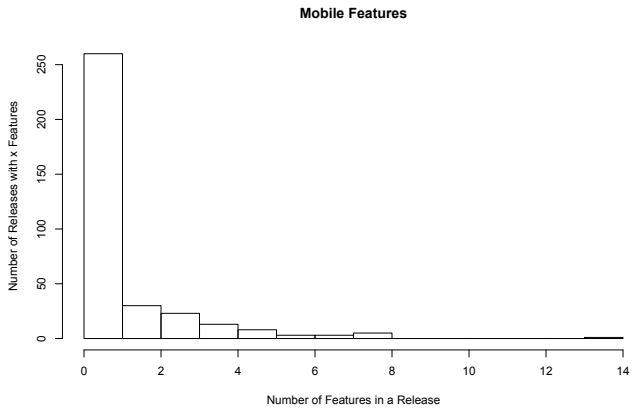

Figure 2: Mobile Features: this is a histogram of the frequency of releases containing a certain number of statements classified as features

tial for developing a dictionary of keywords or a training set for mining release notes in the future.

The flag field allows researchers to quickly identify versions that are irregular and may need to be excluded from analysis. One example of a flagged version is "If possible, please skip this update and wait for version 4.2.5. We are very sorry about this and will get it fixed ASAP!"

Finally, we developed scripts that take the information we have stored and perform statistical analysis on the certain cut of our standardized dataset to best answer our three research questions.

The data collected for all research questions fit into histograms similar to Figure 3. Figure 3 shows that the data does not follow a normal distribution, therefore, throughout the analysis we use the Wilcoxon Rank-Sum test to determine whether application categories exhibit significantly different behavior.

3.1 Release Cycles

While performing the calculations of the release cycles, we noticed instances where there would be a negative number of days between some releases. For example, some release cycle lengths Tableau are: . . . 33, 34, -68, 34, 34, 36, 33, 24, 28, 39, 28, 23, 36, -56, 20 . . . . Such patterns only occurred in desktop applications and relate to the simultaneous maintenance of multiple branches. In Tableau, the release cycles of lengths 33, 34 correspond to the 4.1.x branch, release lengths -68, 34, 34, 36, 33, 24, 28, 39, 28, 23, 36 correspond to the 5.0.x branch, and release lengths -56, 20 correspond to the 5.1.x branch.

Negative release cycles not possible and so we decided to handle simultaneously maintained branches by tracing negative release cycles backwards until we found the first positive length of the release cycle and then continued to calculate release cycle lengths down each branch. So the sequence . . . 33, 34, -68, 34, 34, 36, 33, 24, 28, 39, 28, 23, 36, -56, 20 . . . becomes . . . 33, 34, **36**, 34, 34, 36, 33, 24, 28, 39, 28, 23, 36, **3**, 20 . . . when simultaneously maintained branches are factored in.

We did not observe this behavior in mobile or sibling applications . We suspect that this is due to the nature distribution platforms for mobile applications. Where desktop applications often have all their release notes stored on a centralized website or wiki, mobile applications will either show the release notes for the current release if on the Google Play or the Microsoft App Store (though some developers append notes as opposed to replacing them each release) or up to the previous 25 releases if on the iTunes App Store. If the developers do not maintain separate release notes on their own websites it does not make sense to maintain multiple simultaneous release branches for one platform because the release notes can quickly be replaced and leave users confused if for example they have version 5.0, but the most recent release notes are shown to be 4.9.5. Multiple branches may be maintained across mobile platforms, but this is not evident because the release notes are stored across two different platforms (e.g. iTunes App Store and Google Play).

Outliers The authors looked at the outliers from each of the application categories to determine if there were any irregularities. For mobile applications, the longest cycle length appeared in the application Sleep Cycle Alarm with a length 345 days. There was no indication as to the delay, neither the release notes nor the developers twitter did not provide any insights as to why the release cycle was so long.

The sibling application OmniGraffle contained the longest release cycle of all application categories at 825 days. However, OmniGraffle appears to be the victim of missing release notes. The version number suddenly jumps from 3.2.4 to 4.1.2, it is very likely that the documentation either lapsed, was deleted, or misplaced during the 825 day period between releases. As such, the authors removed this data point from the calculations because it does not accurately represent a single release cycle. The next longest cycle length at 442 days belongs to the application Covenant Eyes, which during that cycle completed a large scale architectural change.

The desktop application with the longest cycle length is NDepend, a static analysis tool, which had a cycle length of 606 days. According to the release notes, the two year release cycle spanned the jump from .NET 1.0 to .NET 2.0 assemblies. During that time, no new Visual studio was released, so perhaps the developers did not feel the need to update NDepend.

3.2 Release Cycle Contents

To analyze the release cycle contents after classification, we added up the instances of each category (bug, feature, enhancement, non-functional, gratitude, and feedback/contact) to create a summary of the release.

After each release was summarized, we pulled the information from each summary and grouped by statement category for each application of that type into lists. For example, Desktop Application 1 has two releases containing 2 bugs and 2 features and 4 bugs and 0 features. After being

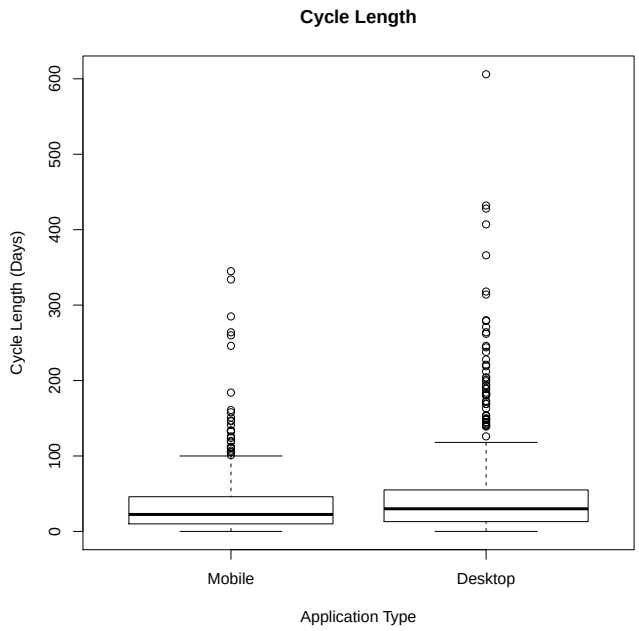

Figure 3: This box plot summarizes the information shown in Table 4.1

grouped by type desktop applications will have a list of bugs containing 2 and 4 and a list of features containing 4 and 0. From these lists we are able to calculated the statistics for what type of contents constitute a common release.

3.2.1 Linear Models

In addition analyzing the number of occurrences of each category, we also tried to generate a linear model for each of the different categories of application.

The linear model follows the following form:

$$time_{category} = \beta_0 + \beta_1 Bugs_{category} + \beta_2 Features_{category} + \beta_3 Enhancements_{category} + \beta_4 NonFunctional_{category} + Error \quad (1)$$

4. CONCLUSIONS

4.1 RQ1: Do mobile applications exhibit more frequent releases than desktop applications?

Table 2: Cycle Summary

	mean	median	std	max
Mobile Applications	35.784	22.500	43.591	345.000
Desktop Applications	51.473	30.000	69.928	606.000

Table 3: Cycle Length

Application Comparison	p-value
Mobile vs Desktop	6.35E-04

Research question one allows the authors to establish a baseline for the behavior of both only mobile applications and only desktop applications. The authors originally suspected that mobile applications have shorter release cycles than desktop applications. The data shows that this assumption is correct.

As shown in Table 4.1, we found that both the mean and median for desktop applications' cycle lengths was larger than than of mobile applications. We believe that the median more accurately represents the characteristic cycle length because the cycle lengths do not exhibit a normal distribution as mentioned in Section 3.

Given the difference in the maximum cycle lengths between the two categories of application, we expected a more substantial difference between the median cycle lengths. However, the data indicates that the difference between the two is only about a week. Table 4.1 shows that mobile and desktop applications demonstrate statistically different behavior (p-value < 0.05).

We believe that the mobile applications have shorter release cycles because they were more likely developed in an agile paradigm than the desktop applications. The oldest mobile application was first released in 2009, 8 years after the agile manifesto was published. The oldest desktop application was first published in 2004, only 3 years after the agile manifesto was published so agile programming may not have had enough time to become a common practice during the early development of the desktop applications. Additionally, during the initial development of some of the desktop applications, software was still distributed on disks or at least not as conveniently for users as automatic updates. The original need to physically update software may have created a culture of lengthened desktop release cycles.

In summary, mobile applications have distinctly shorter release cycles than desktop applications by 7.5 days.

4.2 RQ2: Do sibling applications' release cycles behave more like mobile or desktop apps?

Our second research question looks at whether the release cycles of sibling applications tend to exhibit behavior closer to desktop or mobile behavior because research question one showed that mobile and desktop release cycles displayed different behavior.

Table 4.2 adds another row to Table 4.1 for sibling applications. Table 4.2 shows that the mean and the maximum cycle length for the sibling applications lies between the mean and the maximum for desktop and mobile applications. Surprisingly, the median cycle length is lower than the mobile median cycle length. Additionally, there is no statistically significant difference between mobile and sibling applications, therefore we claim sibling applications release cycle lengths behave like mobile applications.

Sibling applications behaving like mobile applications is unexpected because almost all of the sibling applications were originally desktop applications that later had a sibling mobile application developed. The two exceptions for this include Spotify, whose mobile application was released first, and SplashID, whose sibling applications were released on

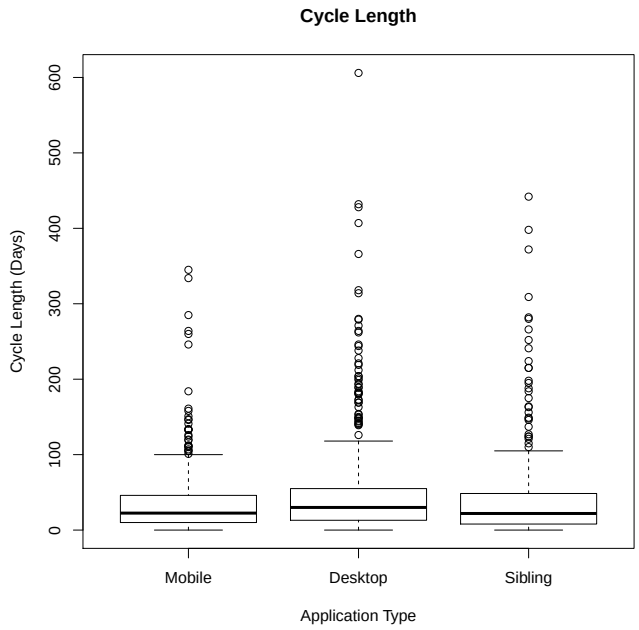

Figure 4: This box plot summarizes the information in Table 4.2

the same day. Sibling desktop applications also made up a larger portion of the data points for this category (250 desktop points vs. 62 mobile points), so we anticipated sibling applications to behave more similarly to desktop applications.

The oldest sibling application is OmniGraffle, which first released in 2005. However, unlike the desktop application category, OmniGraffle's age is an oddity among the sibling applications, with the next oldest being released 4 year later in 2009. This shows the potential hypothesis that age and release cycle length strongly correlated for future work.

In short, sibling applications exhibit release cycle lengths of mobile applications.

Table 4: Cycle Summary

mean	median	std	max
35.784	22.500	43.591	345.000
51.473	30.000	69.928	606.000
43.283	22.000	64.215	442.000

Table 5: Cycle Length

Application Comparison	p-value
Mobile vs Desktop	7.54E-03
Mobile vs Sibling	4.88E-01
Sibling vs Desktop	2.18E-03

4.3 RQ3: Are bugs/features/enhancements more common in desktop, mobile, or sibling applications?

4.3.1 Bugs

Table 6: Bug Summary

mean	median	std	max
0.899	1.000	1.043	8.000
4.467	3.000	5.179	33.000
1.464	1.000	1.590	11.000

Table 7: Bugs

Application Comparison	p-value
Mobile vs Desktop	1.38E-16
Mobile vs Sibling	1.14E-02
Sibling vs Desktop	1.92E-10

Table 4.3.1 contains a summary of the number of bugs found per release of each of the application categories. We once again use the median as our standard for comparison, which shows that desktop applications have more bugs per release than mobile or sibling applications.

After looking at the causes for the maximum number of bugs being three times that the sibling application maximum number of bugs, we found that desktop applications were more likely to include release notes from publicly release beta builds. We searched each of the application categories corpus to look at instances of the word "beta." We noticed that mobile applications mention beta features or accounts or mentions for supporting beta operating systems, but no versions are mentioned as being in beta. Likewise the only mentions of a beta version in the desktop sibling application. Many of the desktop applications mention certain versions as being in beta.

The lack of public betas could be attributed to the mobile applications public distribution that automatically checks for updates applications to prevent customers from unknowingly downloading beta applications. It may be the result of the rating system; developers may fear releasing unstable versions that could permanently damage their ratings and thus their profits. The lack of beta versions in mobile applications could be a part of a future work.

Another reason desktop applications have more bugs per release than other categories may be that longer release cycles allow developers to spend more time testing and fixing bugs.

Another influence on the disparity number of enhancements will discussed in the Threats to Validity.

4.3.2 Features

Table 8: Feature Summary

mean	median	std	max
1.092	0.000	1.831	14.000
1.492	0.000	2.326	10.000
0.611	0.000	1.204	6.000

Table 9: Features

Application Comparison	p-value
Mobile vs Desktop	2.34E-01
Mobile vs Sibling	5.45E-02
Sibling vs Desktop	3.16E-03

Table 4.3.2 contains a summary of the number of features found per release of each of the application categories. We were surprised to discover that the median value for the number of features for all categories was 0, but that the maximum values were relatively high. We also noticed that mobile and desktop applications were essentially the same, while sibling applications followed a different behavior as shown in Table 4.3.2.

We believe that the low median and the high number of maximum features in a release can be attributed to an ebb and flow of development and maintenance. When looking at an overview of the occurrences of features the non zero numbers cluster together for a few releases and then are separated my many releases with no new features.

For the mobile applications, the maximum number of features was included in the release notes for the longest mobile cycle showing that for the full year the development team had added quite a few new things. For the desktop applications, maximum number of features in a release appeared Kaleidoscope after a relatively long release cycle of about half a year.

We believe one cause of the sparse number of features could be that developing sibling application bogs down developers as they try to translate ideas across two very different platforms.

4.3.3 Enhancements

Table 10: Enhancement Summary

mean	median	std	max
1.520	1.000	1.895	16.000
4.542	3.500	5.072	28.000
1.590	1.000	2.090	12.000

Table 4.3.2 contains a summary of the number of enhancements res found per release of each of the application categories. Similarly to the bugs per release, desktop applications exhibited more enhancements per release than mobile or sibling applications. For the number of enhancements, mobile and sibling applications showed the same behavior as seen in Table 4.3.3.

Table 11: Enhancements

Application Comparison	p-value
Mobile vs Desktop	1.03E-08
Mobile vs Sibling	6.58E-01
Sibling vs Desktop	1.26E-07

	Mobile Applications
(Intercept)	23.77^{***}
	(3.46)
Bugs	1.52
	(2.57)
Features	6.95^{***}
	(1.43)
Enhancements	3.61^{*}
	(1.52)
Non-Functional	2.06
	(3.58)
R^2	0.13
Adj. R^2	0.12
Num. obs.	346

$^{***}p < 0.001$, $^{**}p < 0.01$, $^{*}p < 0.05$

Table 12: Statistical models

The largest amount of enhancements per desktop application appeared in Versions after a release cycle of 194 days. Desktop applications in general may have more enhancements per release because longer release cycles give the developers more time to test their applications. Desktop applications may also be better hiding their bugs as sometimes enhancements are used by developers to hide that bugs were present before. The authors could not know the motivation behind each enhancement, bugs may be hidden beneath some of the enhancements for all application categories.

Another influence on the disparity number of enhancements will discussed in the Threats to Validity.

4.4 Modeling Behavior

4.4.1 Threats to Validity

- A threat to the validity of RQ3 is that theres is a significant business pressure to create sibling applications on

	Desktop Applications
(Intercept)	45.12^{***}
	(10.20)
Bugs	-1.35
	(1.51)
Features	9.01^{*}
	(4.44)
Enhancements	2.45
	(2.11)
Non-Functional	-8.78
	(7.47)
R^2	0.11
Adj. R^2	0.08
Num. obs.	120

$^{***}p < 0.001$, $^{**}p < 0.01$, $^{*}p < 0.05$

Table 13: Statistical models

	Sibling Applications
(Intercept)	30.38^{***}
	(5.15)
Bugs	-2.17
	(2.16)
Features	1.74
	(3.00)
Enhancements	6.61^{***}
	(1.80)
Non-Functional	-3.42
	(2.60)
R^2	0.07
Adj. R^2	0.06
Num. obs.	239

$^{***}p < 0.001$, $^{**}p < 0.01$, $^{*}p < 0.05$

Table 14: Statistical models

the market, there must be a reason that desktop applications were unable to migrate to mobile platforms. If the reason has to do with computational power, function, or code size, then it stands to reason that more complicated applications may have more bugs, making our comparison unbalanced.

- Developers are also, whether intentionally or not, vague about the number of bugs or enhancements they address in each release. We may consider every instance of "bug fixes" or "enhancements" as 2, but in reality we will never know the exact number.
- Additionally, this process is quite time consuming and has forced us to reduce our initial goals for the number of applications down to around 12 for each category. Without research question 3 it is relatively easy to add new data to our corpus which could be done towards the end of the semester.
- We originally looked to machine learning to discover if there was an option that would automatically classify our collected data, though while being easy to implement a using the natural language processing toolkit, a classifier requires a very large training set. If we were to exceed 800 classifications, we may be able to train a naive Bayes classifier to quickly add new data, but in the time remaining in the term we may not be able to classify that much data.

5. CHALLENGES

Had this project been executed 5 to 10 years ago it would be easier to find desktop only applications. Business pressure and interests have driven most companies that are willing or organized enough to post release notes to develop sibling applications . As a result we may have to reduce the number of applications we add to the corpus so as to not unbalance our data.

Another challenge we have already overcome is the taking of release histories from iTunes, which blocks copy and paste attempts. Web scraping was unsuccessful, spoofing the iTunes user agent provides a successful and automated way to collect the data. iTunes maintains a large collection of release histories for each of the applications it sells as well as maintaining a set of application ids on each page which could be used to quickly gather data. On the other hand, the Google Play Store only maintains the current version, meaning we have to search for developers to provide the information we need.

We had not factored in web applications to our original research plan. For the time being we plan to not consider web applications as either mobile or desktop applications and therefore not a part of our study.

6. IMPLICATIONS

For team management: Our results may show a significant skew in bugs toward one classification of application. If this was the case, it may provide the stimulus for a manager to add extra members to the QA team or allow extra time for QA before releasing an application. Even within the subsets of application types there may be a certain amount of time that passes between releases that may be associated with fewer bug fixes which management can use as the length of a release cycle.

For tool builders: The data generated by our research may show a high concentration of bugs in a certain type of application. If that is the case a tool builder could look at the bugs for that type of application and determine if a tool needs to be build to help developers avoid a common bug. For example if sibling applications have a high number of bugs there may be specific pitfalls creating a mobile application from a desktop application or vice versa that a tool builder could address.

For researchers: Researchers may find our results useful when doing research on mobile applications. Also, for future work we may be able to generate a keywords dictionary or bigram dictionary based on our classification of statements that will allow them to perform more data mining on release notes. This would be similar to what Yu [5] attempted, but with keywords drawn from concrete examples.

7. RELATED WORK

Gall et al. [1] developed a product release database for telecommunication switching system that automatically generated release notes based on the source code. Like Gall et al., we plan to analyze software using the release histories provided by developers to derive information about software. However, instead of focusing on a single application, we will use information to compare applications on different platforms. In addition, the release notes we analyze are not generated based on the source code, but are disclosed by the developer.

Tsay et al. [3] paper presents experiences, including the tools, techniques and pitfalls, in mining open source release histories, and specifically shows that many factors make automating the retrieval of release history information difficult, such as the many sources of data, a lack of relevant standards and a disparity of tools used to create releases. These are the similar problems we've found in our research. The different thing from our recent research is that the projects quantity in this paper is very small, but number of versions of each project is very big. For example they collected 1579 distinct releases from 22 different open source projects, so on average 72 release versions per project. However, we focus on more types of software but less release version.

Yu [5] uses text data mining to extract information from change logs and release notes develop a mathematical model to represent software evolution based on predetermined keywords. Using the model, Yu traced the recurrences of issues across a single project. Similarly our research mines release notes to gather information about the nature of software bugs and features, but ours attempts to use the data to compare different applications. Apart from version numbers and dates, our research steers away from filtering out information solely due to predefined keywords because positive spin, marketing terms, and buzz words interfere with simple collection of data. The inconsistencies in language across applications and companies is the reason we originally looked into naive Bayes classifiers to better assess the nature of statements in release notes.

Shobe et al. [2] present an empirical study on the release naming and structure in three open source projects: Google Chrome, GNU gcc, and Subversion. The projected application requires forming a training set for a source-code change prediction model from the committed source code history are needed. Our research also mines histories, but we focus less on developing a machine learning engine and more on potentially using one as a tool to help us perform classifications.

Wright and Perry [4] discusses early results from a set of multiple semi-structured interviews with practicing release engineers and focus on why release process faults and failures occur, how organizations recover from them, and how they can be predicted, avoided or prevented in the future. Our research only seeks to look at the frequency, but Wright and Perry's interviews mentioned some root causes such as size, funding, and modularity that affect the frequency of release cycles.

8. REFERENCES

[1] Gall, H., Jazayeri, M., Kolsch, R. R., and Trausmuth, G. Software evolution observations based on product release history. In *Proceedings of the International Conference on Software Maintenance 1997*.

[2] Shobe, J. F., Karim, M. Y., Zanjani, M. B., and Kagdi, H. On mapping releases to commits in open source systems. In *Proceedings of the 22nd International Conference on Program Comprehension*, pp. 68–71.

[3] Tsay, J., Wright, H. K., and Perry, D. E. Experiences mining open source release histories. In *Proceedings of the 2011 International Conference on Software and Systems Process*.

[4] Wright, H. K., and Perry, D. E. Release engineering practices and pitfalls. In *Proceedings of the 34th International Conference on Software Engineering*, pp. 1281–1284.

[5] Yu, L. Mining change logs and release notes to understand software maintenance and evolution.