Analysis and Detection of Bad Smells

Pooja Asher
North Carolina State University
pmasher@ncsu.edu

Zeal Ganatra
North Carolina State University
zganatr@ncsu.edu

Pratik Mukherjee
North Carolina State University
pmukher@ncsu.edu

Ashwin Oke
North Carolina State University
aboke@ncsu.edu

Sandeep Samdaria
North Carolina State University
ssamdar@ncsu.edu

ABSTRACT

Software engineering deals with all the aspects of software development, right from requirements gathering to software maintenance. An important aspect of software engineering also is software project management. It is important to follow good practices in that regard as well, to shape a project into a well engineered software project. In this project, we have analyzed three different project teams for the existence of any “bad smells”. For this project, bad smells source from different GitHub software development management elements such as data from GitHub issues, milestones, commits, labels, commit patterns and communication patterns. In the end, we have come up with a simple detector for an early bad smell warning.

1. INTRODUCTION

Engineering a software often involves collaborative work and often involves assimilation of different approaches and working methodologies of the team members. Over the period of time issues, hindrances and bottlenecks are encountered in the development process which needs to be adequately by the team. Many of these arise due to the suboptimal programming practices of either the team as a whole or an individual. Sometimes a lack of a coherent, integrated working methodologies prevents a seamless and systematic development which in turn gives rise to blockades. Often inexperienced developers are embroiled in programming or implementing the solutions and tend to overlook the various facets and fineries of a streamlined approach. This leads to patterns deviates considerably from the dogmas of software engineering and affects the performance of the team adversely. These “anti-patterns” or “bad-smells” leads to escalation of software costs and development duration. Almost all of them remains undetected and unseen until the very flag end and any corrective changes made doesn’t enable any significant improvement. Therefore it becomes imperative that for cost-intensive operating on strict timelines that such patterns are avoided or identified and dealt with accordingly.
In the course we have been working in teams in similar projects - to identify user problems dealing with softwares, analyzing the causes, implementing solutions and testing them. All of these were being done under strict deadlines, often overlapping individual schedules and therefore demanded a systematic and integrated approach in each one of our deliverables. Expectedly as learners, there has been deviations from the established principles of development life cycles. Nevertheless such mistakes calls for to be studied and learnt from. We therefore aim to study three software projects developed as a part of the course including one of ours, identify the “smelly” patterns baselined on certain metrics and software features. A careful study of these would give an overture to the anomalies and the detrimental practices often committed as a part of software development and would assist in avoiding them in future and taking appropriate corrective measures.

2. METHODOLOGY

2.1 Collection

2.1.1 What we collected

We collected a wealth of information from the Github repository of each of the three groups. The data was collected in the same format that is used by the Github to save the information about individual repository.

2.1.2 How we collected

Data is extracted from each of the three groups’ repository. We used Github’s public API for this purpose. We initially generated a ‘token’ using which in the client program gave a call to the API exposed by Github to download data from the Github repository link specified in the program. The data was returned in JSON format which was than preprocessed and saved in Sqlite database.

As suggested we used the given program gitable.py client program with a few modifications for saving data in Sqlite database.

2.2 Anonymization

For all the three groups analyzed, both the users and the Groups are anonymized. Each Group is renamed to “Group” followed by a number, for example “Group1”. We assigned the Groups in a random order. This number do not signify anything. Similarly, each user in the respective groups are also renamed to “User” followed by a number for example “User1”. Again this number is randomly generated and do not signify anything.

2.3 Database Schema

The Databases created from the data extracted from Github for all the three groups consist of five tables namely comment, commits, event, issue, labels, and milestone.

Issue Table:

- Id
- Name
- CreateTime
- ClosedTime
- Milestone_Id
- Assignee_Id
- Label_Str
- Labels_Count

Milestone Table:

- Id
- Title
- Description
- Created_At
- Due_At
- Closed_At
- User
- Identifier

Event Table:

- IssueID
- Time
- Action
- Label
- User
- Milestone
- Identifier

Comment Table:

- IssueID
- User
- Createtime
- Updatetime
- Text
- Identifier

Commits Table:

- Id
- Time
- Sha
- User
- Message

Labels Table:

- Id
- Name

2.4 Feature Detection and Results

We classified the data into 6 broad categories and identified 16 features. These features were identified from the data collected by our git analyzer tool.

2.4.1 **Issues**

Issues are an essential git tool for ongoing communication. They keep track of tasks. Bugs and status of each part of the project. Here we analyse various metrics involving issues to gauge the efficiency of a project group while working together and also contributions of members on github.

2.4.1.1 Open Issues

Open issues after project completion is a big red flag that the project issues haven't been closed either because certain parts of the project that were expected have not been completed, or there are errors that have been left unfixed. Another thing that open issues show are that members are not communicating the completion of an issue using github, which may show lack of communication in general.

Group	Number of open issues
Group 1	0
Group 2	5
Group 3	0

Table 1: Number of open issues

Group 1 and Group 3 both had zero open issues at the end of the projects which ideal. The 5 issues group 2 had, is a problem since it shows some of the tasks assigned were not completed, or github wasn't updated after their completion.

2.4.1.2 Issues without milestones

Milestones show the goals set for the project. Issues connect closely with milestones and essentially can be regarded as tasks required to complete milestones. Hence Issues should be attached to their respective milestones, so as to keep track of the 'tasks' and 'bugs' for achieving the milestone.

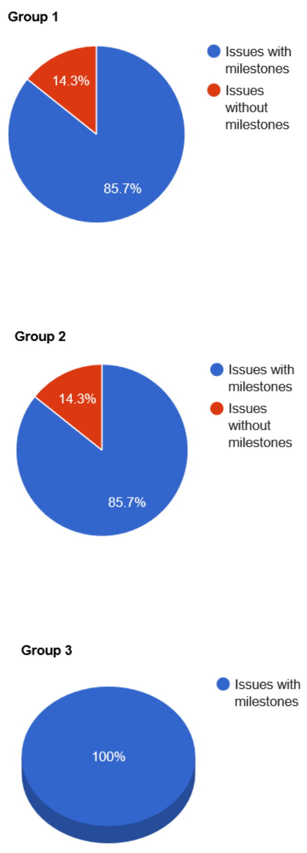
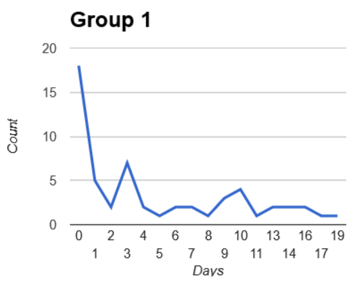


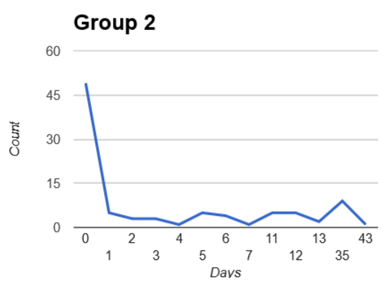
Fig 1. Distribution of issues v/s milestones

Both group one and group two have about 86% of all their issues attached to given milestones. Group 3 has every issue assigned to a particular way, which is a good way of keeping track of the project and dividing tasks. The issues without milestones might be miscellaneous which is plausible, or may not have been assigned properly, which points to poor division of tasks.

2.4.1.3 Issues with unusual duration of time

Issues have a fixed opening and closing date and can be subsequently analyzed for their duration. Issues could essentially take more or less time to complete depending on the task or error. But unusually long or short time indicate on bad smells, since it shows the issues for the milestones were not well designed to begin with, or did not complete as anticipated due to some issues in the implementation.





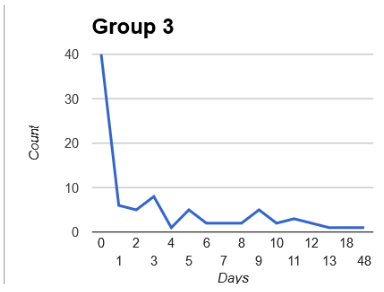


Fig 2. Issues with unusual duration of time

Graphical representation of the number of issues per number of days of issue duration, shows the distribution of duration as required. All the groups have a large number of issues on the left of the graph, i.e. with a very short duration. This indicates that issues were made on small tasks that were completed within a day or two. For long duration group 1 and group 3 have very few issues trailing over a long duration. For group 2, there was a larger number of issues with a long duration. Spikes in the number of issues for a large or very short duration indicate poor design or planning.

2.4.1.4 Issues without assignees

Issues are new tasks or errors in some tasks. Github allows issues to be assigned to a user. This is a good way to keep track of which user is responsible for which task and also a good starting point to assign divided tasks to users.

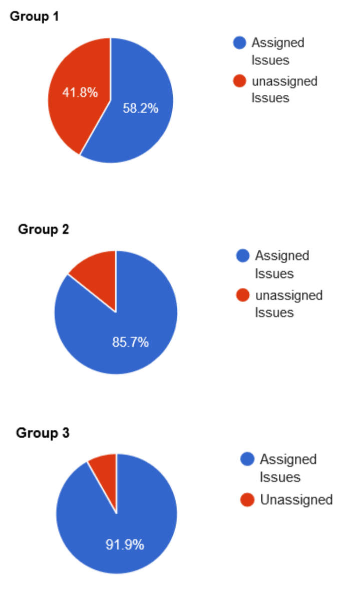


Fig 3. Issues with assignees

The charts show percentages of issues with and without assignees. Group 1 had a high number of issues without assignees which is a bad indicator. Group 2 and group 3 had relatively lesser issues without assignees. Ideally all issues should have a person responsible for it.

2.4.1.5 Issues assigned to each user

Issues assigned to users are basically a measure of how much work a user is assigned and how much work he gets done in terms of number of issues.

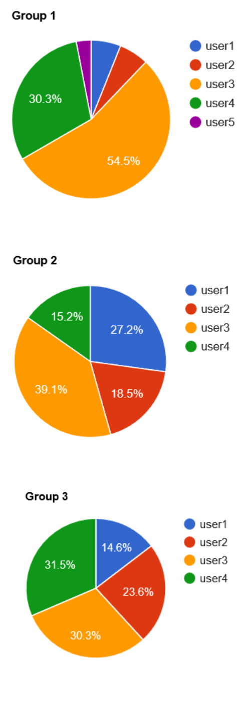
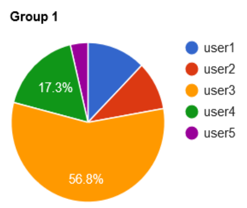


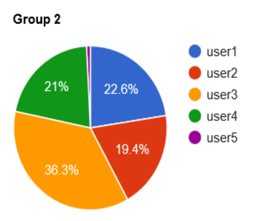
Fig 4. Issues assigned to each user.

These pie charts depict distribution of work using the assignees for users. Group 1 has an unusually large number of issues assigned to user 3. A number of users are given very few issues assigned and this is poor division of work and distribution by issues. Group 2 has approximately equal percentage of work , though user 3 has a large percentage comparatively. Group 3 has near ideal division and approximately 25% of issues each, with the exception of user 1.

2.4.1.6 Issues posted by each user

Issues are added to create and assign new tasks. It may also be added when there are issues or bugs found in the project. Both of these show initiative and planning on the part of the users and hence the number of issues per user poses as an important analysis.





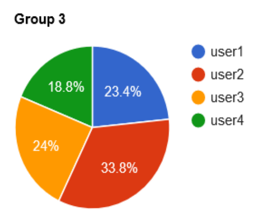
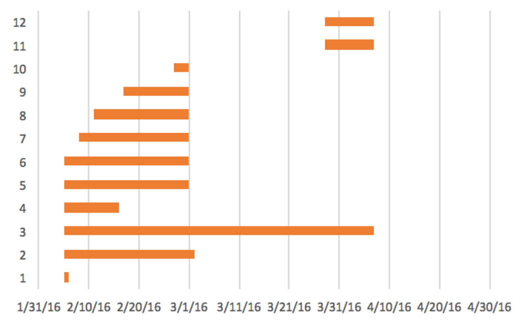


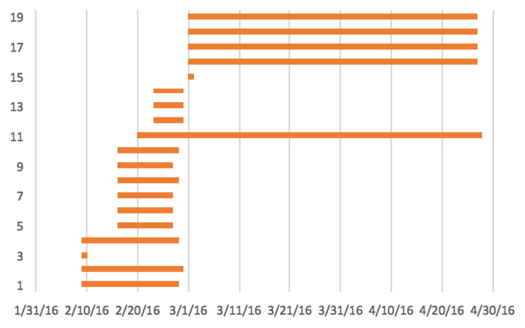
Fig 5. Issues posted by each user.

2.4.2 **Milestones**

2.4.2.1 Milestone timeline

Milestones help in organizing, tracking the issues and the progress related to the deliverable. Every milestone is associated with creation and closing date. In typical cases the duration varies from 5 days to 20 days or more depending upon the size of the deliverable. An optimal duration indicates a well defined system in place for tracking and managing the progress and the related issues.





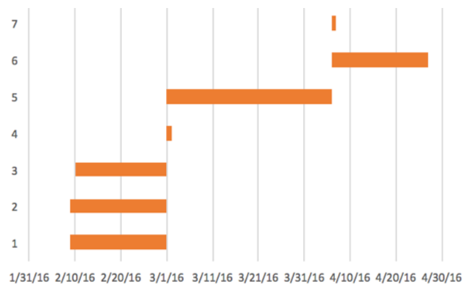


Fig 6. Milestone timeline for groups.

Group 1 had 12 milestones in total. Out of which one milestone was opened for just a day indicating low planning. One milestone (Milestone 4) was open for an unusually large duration - 62 days way beyond 2 SDs away from the mean (Mean: 18.91 and SD: 15.33). This might be due to anticipatory creation of milestones for futuristic deliverables. Such an anomaly might also occur because of inadequate breakdown of a deliverable into smaller milestones and inadequate modularity for the concerned deliverable.

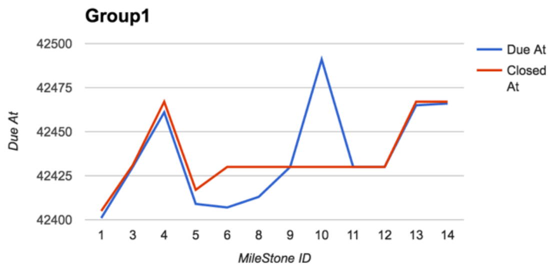
Group 2 had two milestones of only a day's duration again indicating less planning of the related issues and the progress. On the other hand, a single milestone lay open for duration 68 days - beyond the usual range of values (mean: 23.26 and SD: 22.17). From the graph it is evident that, most of them had a due date and closed date towards the last date of the month when the deliverable as a whole was due. This indicates an inadequate modularization of the tasks for a deliverable.

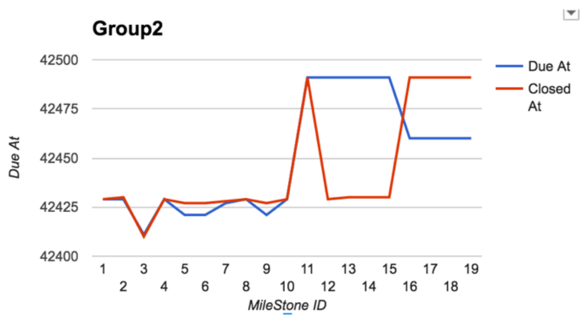
Group 3 also had two milestones that were opened for 1 day. However, it didn't have any milestones of unusually large duration.

Overall all the three groups had a good management of the milestones that they created even though all the groups had few milestones with short duration. Group 3, like groups 1 and 2 had milestones that were all closed on the same date. There was overlapping instead of being sequential again indicating towards an improper tracking mechanism

2.4.2.2 No. of milestones completed after due date

Besides the created at and the closed at attributes, milestone has an associated due at attribute associated with it that gives an estimated deadline of the required implementation to be made for the corresponding deliverable. In most cases it is an anticipated one that gives a prelude to the effort needed for the implementation. Projecting the actual completion date against the due at date provides an insight to the quality of the planning of the work. If for most of the milestones the closed at dates are before the associated due dates then it indicates a well planned and a carefully executed implementation. On the contrary if for most of the milestones, the closed at dates lie way beyond the due dates then it indicates poor planning as regards to the deliverables and a non adherence of the timelines - mostly due to hurried or rushed work.





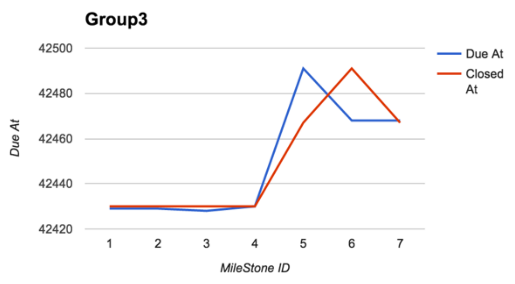


Fig 7. Milestone closed and due date.

Out of the 12 milestones. Group 1 had closed only 3 on the same date as on which they were due. 8 of the milestones were closed much beyond the due date. This indicates that the anticipated deadlines for the deliverables were either inadequate or sufficient time was not allotted to the milestones as was required. It might be also due to missed deadlines because of rushed work at the last moment. One milestone had been closed 61 days before it was due. This mostly indicates an invalid milestone that was erroneously created.

Group 2 on the other had a mixed bag of results. Three of the milestones closed on the same date as due. Only one milestone had closed a day ahead of when it was due. The rest of them had an unusually large deviation in between their due and closed days. 4 of the milestones were closed almost 61 days before their due date and the remaining four were closed about a month after their due date.

Group 3 too had most of their milestones closed after the due date with 2 closed before their due dates.

The overall picture leads to several possibilities. Since most of the teams had milestones with missed deadlines it might indicate hurried work at the last moment just before the deliverable instead of incremental development. It might be so that inadequate time was allotted to each milestone - mostly due to lack of insight or estimation about the timelines. The other probable explanation for such an anomaly is the lack of update of the milestone status upon completion as they might have been all updated just before the monthly deadlines.

2.4.2.3 Closure of issues after milestone closure

Milestones as we know act as placeholder for issues. They are a representative of a certain deliverable and all the issues related to the deliverable are linked to its corresponding milestones A milestone there can only be closed after all the issues mapped to it have been resolved and closed. Presence of issues that were closed after the date on which the corresponding milestone was closed indicates an inefficient tracking system where either the issues weren't updated and closed in a timely manner or the milestones weren't used efficiently to keep track of the issues related to the implementation of the corresponding deliverable.

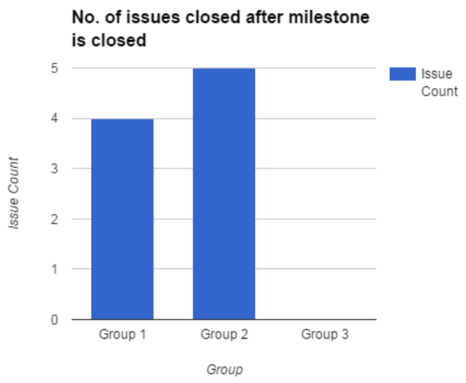


Fig 8. No. of issue closed after milestone closure.

The graph depicts the number of issues per group that were closed after the dates of the corresponding milestones. Group 3 had no such issues indicating the closure of the issues were done in a timely fashion and was in sync with the closure of the corresponding milestones. Group 1 and 2 however had 4 and 5 issues respectively that were closed after the due date indicating that these issues weren't kept track of and were not in sync with the milestone they were mapped to initially.

2.4.3 **Commits**

Git commits can be regarded as smallest units of work delivery in a software project. Commits are code check-ins that are added to the software to enhance a functionality or to add a new functionality to it entirely. Often, each commit addresses a work unit (which is a technical spec) which this particular code segment is trying to solve. As a healthy software development practice, it is ensured that commits are tracked with these work units in some ways, either manually or via a software system in place. Work units can be anything such as Github issues or an entirely different external issue management system such as JIRA. In this regard, distribution of code commits also signify important behaviours of a software project as individual code commits can be directly linked to individual work deliverables. Below, we look at a couple of feature extractors related to distribution of code commits per contributors and as the software project ages.

2.4.3.1 Distribution of commits per contributor

This feature extractor plots the distribution of commits over all the contributors in a project team. Ideally, we would like every project team's distribution curve to be pseudo uniform (approaching uniform) which would signify that all the team members are equal contributors. If we have a non-uniform distribution here, it means that we may culminate a heavy dependency on one of our top contributors on the project, which is not always a good thing to have. It also leads to a lot of skewness in the productivities of the team contributors. Following is the distribution curve for the three groups:

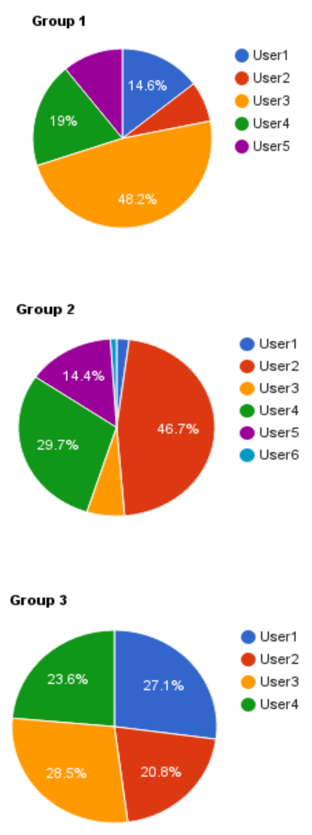
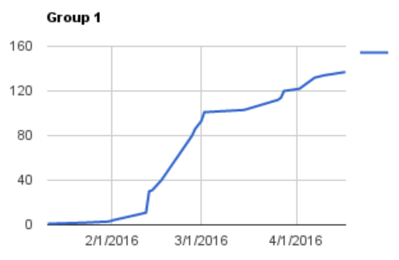


Fig 9. Distribution of commits.

2.4.3.2 Non-uniform distribution of commits over time

It is the age of agile software development today which emphasizes to go for practices like continuous integration and continuous delivery. In such a setup, it is desirable to have nice, regularly flowing commits on each day of the implementation stage of software development. This also ensures that principles like continuous integration are well followed. This feature extractor helps us look into the code commit timeline for any software project. We would expect this curve to be uniform in the development stage. This would mean that the team is actually building a complex software by committing code and merging it with the mainline in short intervals. Following is the distribution for the three groups:



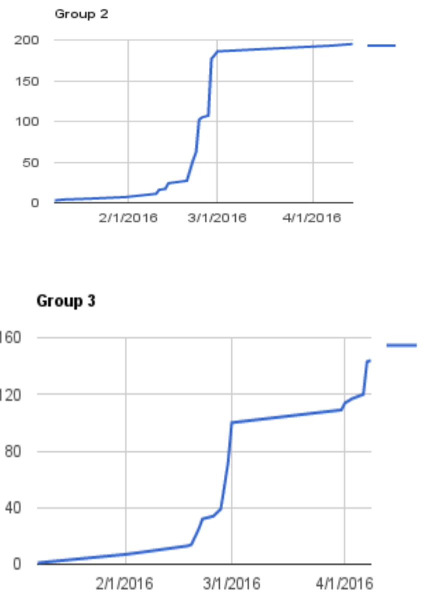


Fig 10. Distribution of commits

2.4.3.3 Commit timeline for each user

This feature extractor is an amalgamation of the above two extractors. This plots the commit timeline for each member in a team. Following are the plots for one of the groups :

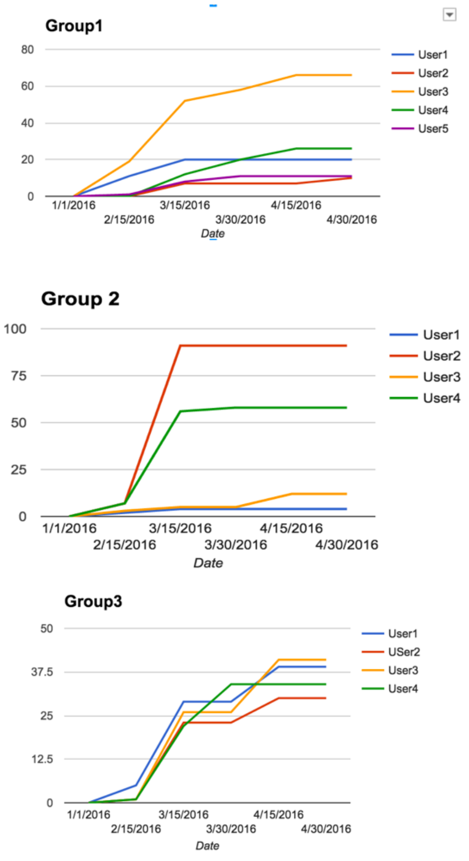


Fig 10.Commit timeline for each user

As we can see in Group1, user2, user4, user5 were absent for the first phase of the project. In Group2, only user 3 has worked after mid of the project. In Group3, everyone has contributed equally throughout the project.

2.4.4 Comments

During the software development the users can communicate via comments on issues to discuss various aspects of the issues. The discussion on the issue via comments can range from the implementation details to minute details of the issue. The comments can be used for multi-purpose and can serve as document, clears any dis-ambiguity related to the implementation.

2.4.4.1 User comment distribution

Users comment distribution on issues can help to identify the effective communication within a group. This feature shows the communication using comments between the groups within a project. Equal distribution of comments among the users depicts a strong communication within a team.

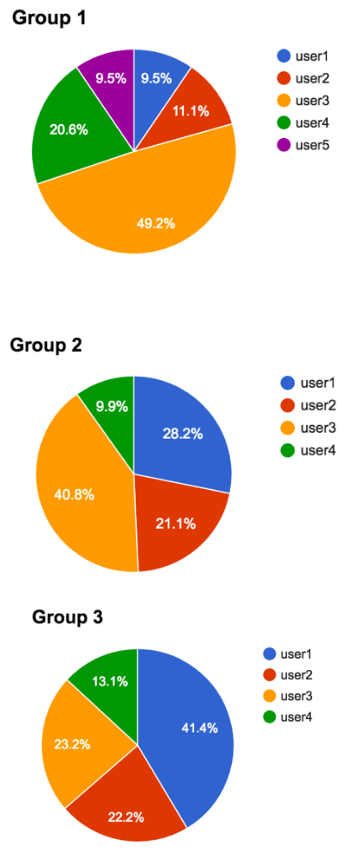


Fig 11. User comment distribution.

We can see that among all the 3 group, there was a user who has more than 40% comments driving the communication within the groups. This could potentially mean that the particular user can dominate the group and is driving the project goals, which could lead to a disastrous effect in long-term.

2.4.4.2 No. of comments on issues

The total no. of comments shows the active participation within the group. This metric also depicts the options being discussed within the group. The users can decide upon the scope of the issue by discussing in the comment.

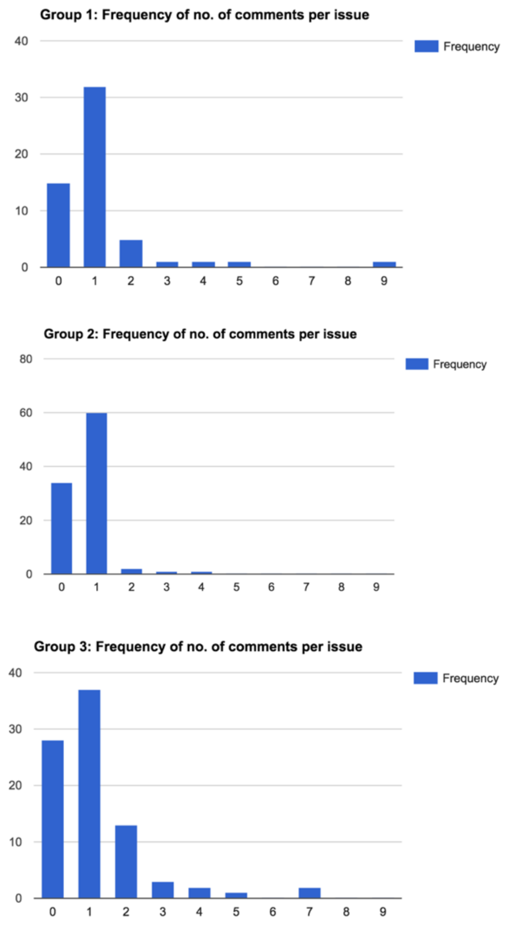
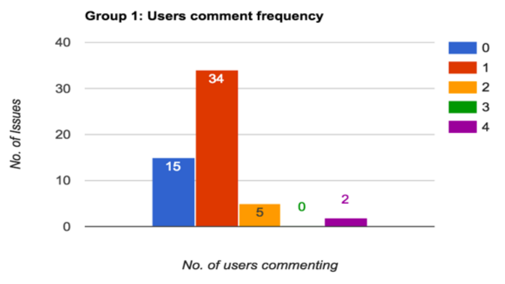


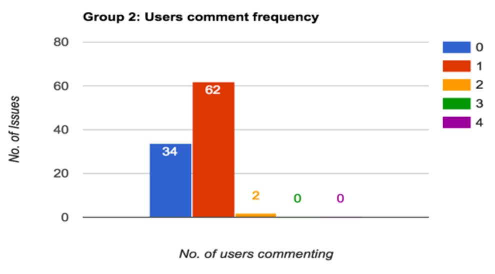
Fig 12. Comment frequency

Total no. of comment count frequency is relatively low for Group 2 as compared to Group 1 and Group 3. For group 1, there were issues which had 9 comments as well, which shows that particular issue was discussed thoroughly for that issue.

2.4.4.3 Distinct User participation per issue

On issues, there can be multiple comments, but the no. of user participating in the comments depicts the participation of individuals in discussing the options related to the issues. Higher the user participants higher the team cohesive is.





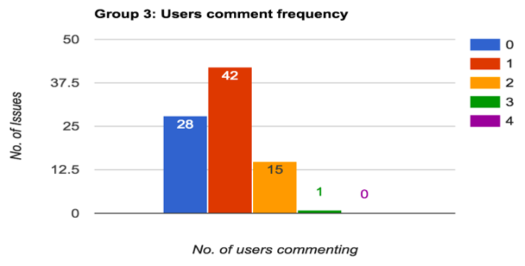


Fig 12. User participation.

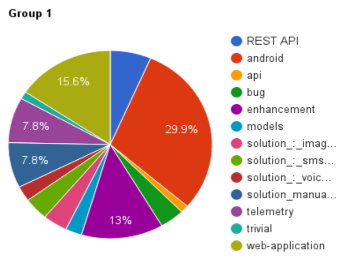
Group 1 and Group 3 have at least 10% of the issues where there is more than 1 user participation. Group 2 has most of the issues with only one user participation, which shows that user participation via comments is not effective.

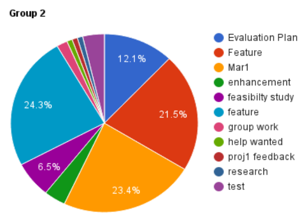
2.4.5 Labels

Labels can be applied to issues and pull requests in GitHub to signify priority, category, or any other information that contributes to organization of issues. Labels are effective tagging mechanisms that let us relate an issue to different domains that it deals with or different components of a software it addresses.

2.4.5.1 No. of issues per label

Using this feature extractor, we collect the number of issues that are tagged along the different labels in the project. This feature extractor can let us know about most featured labels and a rough distribution of issues vs labels.





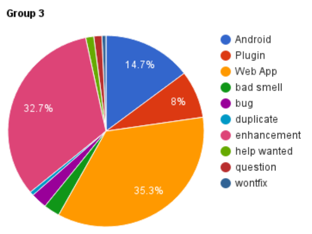


Fig 13.Label distribution

2.4.5.2 Labels without issues:

As mentioned in the above feature extractor, labels are effective tagging mechanisms. However, at the end of a software project, the project team may find out about a few labels that weren't used at all, meaning there were no issues associated with it. This extractor lets us identify such labels which do not have any issues. We have a potential for a bad smell here if we find a lot of unattended labels in a software project. It goes to say that the collection of labels for that project was not properly utilized for issue addressing.

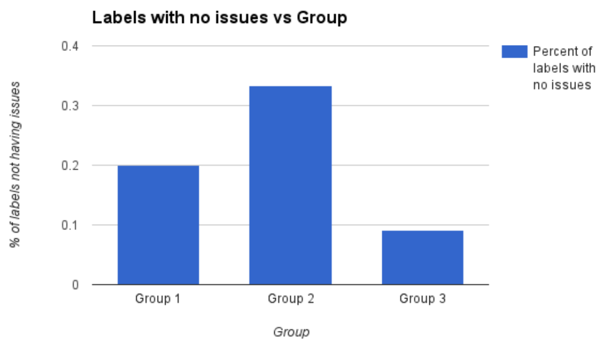
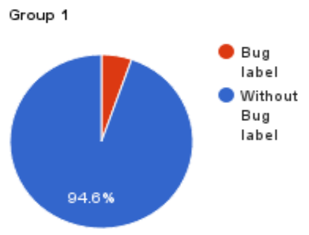
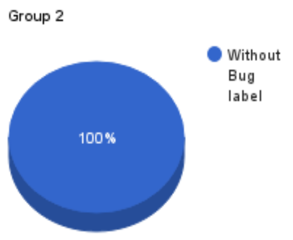


Fig 14. Label without issues.

2.4.5.3 Bug Label Usage

Every software development project undergoes an extensive testing phase, be there any software development methodology being used, waterfall or agile. Tagging every bug with the label 'bug' implicitly gives us a list of all bugs that were reported, if the development team follows this healthy practice. The team just has to look/search for all those issues labelled with 'bug' for a list of software bugs.





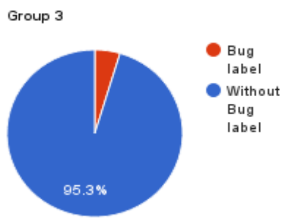


Fig 15. Bug label usage

2.4.6 Branches

Number of branches indicate parallel development. More the number of branches more scope of experimentation and parallel development.

According to the data collected, Group1 has 3 branches, Group2 has 1 branch and Group3 has 5 branches, indicating maximum parallel execution in Group3.

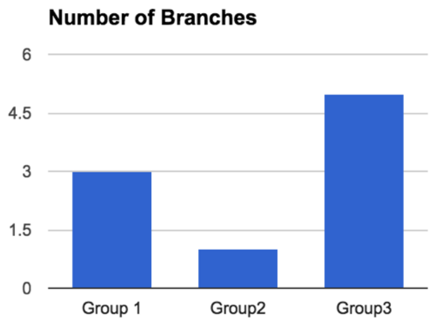


Fig 16. Number of branches.

2.5 Bad Smells Detected

2.5.1 Disproportionate distribution of work

Software development is collaborative work and a balanced - if not uniform - division of work is imperative towards optimal development. In a team expertise in technologies and skill set varies. Individuals well versed with the well versed with the designated programming language and the associated technologies is expected to handle a larger section of the required implementation. Having said that members with less familiarity should be involved in contribution to smaller deliverables providing a foundation to be built upon. Lack of a balanced division of work leads to increased dependency and may stall the project in case of absence of the concerned team member.

- Issues assigned to each user: In the course of the implementation of a deliverable, bottlenecks crop up and needs to be handled by the concerned team member responsible for implementing the deliverable. Unusually large or small number of issues being handled by a team member indicates that issues were not proportionately handled implying that work was not uniformly handled.
- Unusual no. of commits by contributors: Commits are used for version control and as per god software engineering principle frequent commits should be made by every contributor for every implementation. Unusually high number of commits per user indicates that a larger section of the responsibility is handled by only one user whereas number of commits below the threshold level indicates and a less proportion of work handled by the concerned team member.

2.5.2 Improper milestone usage

Milestones act as a representative of any deliverables of the project. They help in efficient management of any modules or sub-modules of the project. During implementation of the project deliverable, there might be issues or bottlenecks that crop up and needs to be subsequently resolved. Often these issues demand the collaboration between two or more than two team members. Therefore, tracking the issues, managing them and communication of any concerns is extremely important for long duration and large team projects. All of these are taken care of by milestones. They act as a placeholder for all the issues related to a deliverable. The issues can be mapped to the milestones and henceforth kept track of. Milestone usage, especially one that is synced with the issues that intend to manage becomes especially important for communication and coordination between the team as a whole. Milestones should be easily discern out the deliverable and reflect the parallelism and the flow of the implementation. They must be in sync with the issues they track of to enable a smooth communication. Most importantly, each of them should have an optimum duration and must be delivered within their deadline to ensure that the project as a whole can be completed in a timely manner. Improper usage and management of milestones is a "bad smell" and therefore can be detrimental for the team communication and the coordination and the completion of the project as a whole.

- Milestone timeline: Each milestone should be opened for an optimal duration and should reflect the sequential flow of the project. Milestones that had been opened for unusual large or short duration indicates inadequate tracking of the issues and improper milestones usage.
- Closure of issues after milestone closure: Milestones act as a container for the issues related to the deliverable they represent. Therefore, the issues should be resolved and closed before the milestone can be closed. On the other hand presence of issues that were closed after the milestone was closed implies a issue tracking mechanism that is not in sync and a poor usage of the milestones that was not in sync with the issues that it was meant to track.

2.5.3 Inadequate Planning

Software project planning is task, which is performed before the production of software actually starts. As a part of planning for effective management of the project accurate estimation of various measures like software size estimation, effort estimation, time estimation, cost estimation is a must. Lack of planning may lead to projects not being able to meet deadlines or projects getting stuck on issues that might make the project completely dis-functional. In some extreme cases it might lead to scrapping of the project that may cause a loss of large amounts of investment. Hence presence of "Inadequate planning" is a bad smell in the projects that must be avoided. The metrics extracted that indicate this bad smell are as listed below.

1. Non-uniform distribution of commits over time: If the group has to rush in development at the end to meet deadline, or the project is completed long before deadline, this indicates improper time estimation for project.
2. Milestone timeline: Milestone are a way of planning the project. Having milestones at regular intervals of time indicate proper planning where proper time and effort estimation is done before hand.
3. No. of milestones before or after due date: Closing milestones before/after milestone due date indicate that time and effort estimation was inappropriate. Presence of more than 5% milestones that have not stick to deadline, indicate poor planning.
4. Issues with unusual long duration of time: Issues that are open for too long indicate procrastination of work. Such issues may lead to a point which might make the project non-functional. We consider issues open for more than 5 days to be too long.
5. Open Issues: Open Issues at the end of the project indicate, the lack of time solve all the issues encountered. This is a indicator of poor time estimation hence inadequate planning.
6. Issues attached without labels: Issues that are not labelled may not be claimed, or have high chances of remaining unclaimed and hence not solved for a long time.

2.5.4 Improper use of Labels

The usage of labels helps in better organization of the issues. As we cited earlier in the report, labels serve as effective tagging mechanisms that let us relate an issue to different domains that it deals with or different components of a software it addresses. There is a potential for a bad smell here. Many a times, a team may realise that a few labels have not been used properly in terms of addressing issues to labels. For this bad smell detector, the following extractors will help us identify whether a bad smell exists or not:

- Labels without issues : This extractor will help us flag a bad smell, if there exists any. This is because if we find a label which has no issues associated with it, such a label contributes to improper usage of labels. We will use a logical metric *labelNoIssuesBadSmell* to come up with a metric to measure the bad smell caused due to it. Following are the values this metric will take. If the percentage of labels with no issues is 0 - *labelNoIssuesBadSmell = 0* (Absent). If the percentage of labels with no issues is in the range (0,10]- *labelNoIssuesBadSmell = 1* (Low). If the percentage of labels with no issues is in the range (10,25] - *labelNoIssuesBadSmell = 2* (Medium). If the percentage of labels with no issues is greater than 25% - *labelNoIssuesBadSmell = 3* (High)
- ‘Bug’ label usage : Every software project has an extensive testing phase during which various bugs are reported. It is a good software development practice to utilize the ‘Bug’ label (provided by Github by default) by tagging each bug to the label. We will use a logical boolean flag *bugLabelBadSmell* to mark if the ‘Bug’ label has been used in the project. *bugLabelBadSmell* takes the value 3 if the ‘Bug’ label was used, 0 otherwise. We will take a simple average of the above two quantities to come up with a cumulative bad smell metric for this detector.

2.5.5 Absence of contributor over a phase of project

Many a times, there are some contributors that do not enter into the project development till a very later stage of the project, while there might be some that leave the project before it is finished. This is a bad smell as it is unequal work distribution between contributors and may result in one person shouldering all the responsibilities of the project. For this we use the following metrics.

- Commit timeline for each user: This metrics indicate the time of when each user contributed to the project. Any user that has started commit 10 days after the project or has stopped committing 10 days before the project development ends, indicates this user as absent during a phase of project.

2.6 Bad Smell Detector Results

After defining the bad smell detectors and identifying those parameters for those detectors, we came up with a scoring system to score each of the parameters within a bad smell. The scores were assigned on a scale of 0 to 3. The interpretation is as follows:

a) 0: No Violation
b) 1: Low Violation
c) 2: Medium Violation
d) 3: High Violation.

Higher the score, higher the severity of the bad smell. Using this metric, we calculated the bad smell score and tabulated the results.

2.6.1 Disproportionate distribution of work

Group	Commit Distribution	Issues assigned to user	Bad smell score
Group 1	2	1	3
Group 2	1	1	2
Group 3	1	2	3

Table 2: Bad Smell for disproportionate distribution of work

2.6.2 Improper milestone usage

Group	Milestone timeline	Milestone closure before issue	Bad smell score
Group 1	1	2	3
Group 2	2	2	4
Group 3	1	0	1

Table 3: Improper milestone usage

2.6.3 Inadequate planning

This bad smell can be detected for each group by measuring the presence of bad smell indicator metrics. The following table contains only the numbers of the smell metrics against group.

Group	Non-uniform distribution of commits	Issues with unusual long duration	Open Issues	Issues attached without labels	Distinct user participation per issue	Bad smell score
Group 1	0	1	0	3	1	5
Group 2	1	3	3	1	2	10
Group 3	2	2	0	2	1	7

Table 4: Inadequate planning

2.6.4 Improper use of Labels

Group	*labelNoIssues BadSmell*	*bugLabel BadSmell*	Bad smell score
Group 1	2	0	2
Group 2	3	3	6
Group 3	1	0	1

Table 5: Improper use of labels

2.6.5 Absence of contributor over a phase of project

This bad smell can be calculated by determining the number of users that exit early or enter late in the project.

Group	*Commit timeline for each user*	Bad smell score
Group 1	1	1
Group 2	3	3
Group 3	0	0

Table 6: Absence of contributor over a phase of project

2.7 Early Warning

Commit Timeline: We use the trend of commits to flag a bad smell. During the development of the project, it is ideal that the commits are made often and made in chunks, instead of a single commit. If the team's commit is spiking within a particular duration, then it means the team is in rush and if there is a long pause between commits, then the team is not working according to the plan. The number of commits should increase linearly with time

Also, user's contributions within the team should be fairly equal. In a team individual user's commit should be approximately between 25% ± 12.5% i.e. in the range of 12.5% to 37.5%. If a user's contribution is less than 12.5%, then it implies that the user hasn't contributed much and if it's more than 37.5%, then it implies that the particular user is doing much of the work with no proper work distribution.

Since, we have to detect the smoke as early as possible, we use the project's two week (from Feb 16th to Feb 22nd and Feb 23rd to Feb 29th) of user commit history. Using the first week as the baseline, we can compare the second week and identify the individual user bad practice and the overall group bad smell.

Algorithm to calculate bad smell for individual User

1. Create a timeline for individual user's total no. of weekly commit for a project.
2. Calculate the contribution of individual user by calculating the average commit.
3. Check if the average commit is within the threshold (12.5% - 37.5%). If not, then raise a Bad Smell Flag.

Algorithm to calculate bad smell for the entire Group

1. Calculate the mean of the weekly commits for each group.
2. Bad Smell for the entire group is detected if the mean of the current week is greater than twice the mean of the previous week (i.e. not linear)

2.8 Early Warning Results

Project 1 Week 1

Mean commits: 6.2

User	Commits	Average Commits	Bad Smell
User 1	11	0.35	FALSE
User 2	0	0	TRUE
User 3	19	0.61	TRUE
User 4	0	0	TRUE
User 5	1	0.04	TRUE

Table 7 Project 1 Week 1 Total Commits per User

Project 1 Week 2

Mean: 10.6

User	Commits	Average Commits	Bad Smell
User 1	9	0.17	FALSE
User 2	7	0.13	FALSE
User 3	20	0.37	FALSE
User 4	10	0.20	FALSE
User 5	7	0.13	FALSE

Table 8 Project 1 Week 2 Total Commits per User

Project 2 Week 1

Mean: 4.75

User	Commits	Average Commits	Bad Smell
User 1	2	0.11	TRUE
User 2	7	0.37	FALSE
User 3	3	0.16	FALSE
User 4	7	0.37	FALSE

Table 9 Project 2 Week 1 Total Commits per User

Project 2 Week 2

Mean: 33

User	Commits	Average Commits	Bad Smell
User 1	0	0	TRUE
User 2	84	0.64	TRUE
User 3	0	0	TRUE
User 4	48	0.36	FALSE

Table 10 Project 2 Week 2 Total Commits per User

Project 3 Week 1

Mean: 1.75

User	Commits	Average Commits	Bad Smell
User 1	5	0.71	TRUE
User 2	0	0	TRUE
User 3	1	0.14	FALSE
User 4	1	0.14	FALSE

Table 11 Project 3 Week 1 Total Commits per User

Project 3 Week 2

Mean: 16.25

User	Commits	Average Commits	Bad Smell
User 1	17	0.26	FALSE
User 2	19	0.29	FALSE
User 3	13	0.2	FALSE
User 4	16	0.25	FALSE

Table 12 Project 3 Week 2 Total Commits per User

Group	Total Bad Smell	2 * Team Members	Average bad smell
Group 1	4	10	0.4
Group 2	4	8	0.5
Group 3	2	8	0.25

Table 13 Average Bad Smell Flag for group

In table 13, the average bad smell for group 3 is less which indicates that the group had proper plan in place and had equal work distribution among the group. Whereas Group 1 and Group 2 had high bad-smell per user.

Group	Week 1 Mean	Week 2 Mean
Group 1	6.2	10.6
Group 2	4.75	33
Group 3	1.75	16.25

Table 14: Weekly mean commit

In table 14, we can see that Group 1's 1st week commit mean is 6.2 and 2nd week commit mean is 10.6. Similarly, for Group 2 the week 1 and 2 mean commit is 4.75 and 33 respectively. And Group 3's mean commit is 1.75 and 16.25. We can see that Group 2 and Group 3 didn't anticipate the amount of work earlier and rushed in Week 2.

3. CONCLUSION

As a part of this project, we extracted project management data for three different project groups in this course. We came up with a set of different feature extractors concerning with various collaborative software development management elements such as GitHub issues, milestones, commits, labels, commit patterns and communication patterns. The next step was to come up with a few bad smell detectors, which we could detect and measure using our set of feature extractors. In the end, we came up with an early detector warning for a bad smell based on a contributor commit pattern. We plotted graphs for these features supported by numerical data.

The most important aspect of this project was the scale of analysis that one can do with public GitHub data for a software project. Using this analysis, one can really get to know the areas of improvement for proper collaborative software development. GitHub public API provides such a powerful source for this analysis.