GitHub Analysis: Finding bad smells in software development using GitHub

Ashutosh Chaturvedi
Department of Computer Science
North Carolina State University
Raleigh, North Carolina, USA
achatur@ncsu.edu

Ayush Gupta
Department of Computer Science
North Carolina State University
Raleigh, North Carolina, USA
agupta25@ncsu.edu

Ankit Kumar
Department of Computer Science
North Carolina State University
Raleigh, North Carolina, USA
akumar18@ncsu.edu

Harshdeep Kaur
Department of Computer Science
North Carolina State University
Raleigh, North Carolina, USA
hkaur4@ncsu.edu

1. Abstract

Software engineering teaches good engineering practices and a key learning is to detect early code smells [1] in a software development life cycle. Since the definition of code smell is subjective, finding early bad smells is important as well as hard to achieve. This report explains the process that we have taken to reach the goal of finding such smells in GIT repositories. On the course of this objective we have defined some features matrices which we have used for data analysis of the repositories. The result of this analysis cast light on what features drive the application most and what features should be periodically taken care of to avoid bad smells.

2. Introduction

This study is aimed at extracting and analyzing the GitHub data of the teams involved in Project 1 and observe what they did to check how well they followed the Software Engineering process. This is done by detecting the “Bad Smells” in different projects. “Bad Smells”, in this case, refer to the shortcomings or the lags in the use of GitHub and the overall Software Engineering process which are determined through a combination of features extracted from the GitHub data for various repositories. Code smells are generally associated with poor design and implementation choices. The may occur due to activities performed by project members who make suboptimal choices or are under time constraint to push things in hurry. Anti-pattern, term for poor design solution is also a cause of recurring code smell.

A formal definition of code smell is subjective and different analyst can deploy different tools for detecting early code smell. The most general approach involves detecting code smell in source code to alarm project members of their presence. These approaches rely on structural information extracted from the source code. In our study we are relying on features of GIT repository to detect these code smells.

3. Data Collection

We have used the shared utility gitable.py to fetch the data from GIT repositories. Along with the data generated by utility, we have developed our own scripts [link] to fetch extra useful data for better analysis. This dataset is specific to features and help us to narrow down on the feature influence better. The python script helped us to generate JSON data files which is imported in MongoDB collections. Uploading data features in database gives us a consistent and central repository system to work with. Key steps involved are:

1. Extraction of data by using shared utility gitable.py
2. Generation of processed data from raw data
3. Upload of generated data in MongoDB
4. Feature data creation using python scripts and spreadsheets.

Collected data is present in JSON format in our GIT repository

3.1 Structure of Data

We structured the data into collections of MongoDB. Details of each collection and their respective attributes are explained below:

Repo:
List of Repositories

Repo_ID	Unique Repository ID
Repo_URL	URL of the repository

Developer:
List of Developers

Dev_ID	Unique ID of the developer
Dev_Email	Email address of the developer

Collaborators:
Mapping of developers with the Repositories

Repo_ID	Repository ID
Dev_ID	Developer ID

Commits:
List of all commits

Repo_ID	Repository ID
Dev_ID	Developer ID
Timestamp	Timestamp of the commit
Message	Message given for the commit

Issues:
List of all issues

Issue_ID	Unique Issue ID
Issue_Title	Title given to the issue
Issue_Created_A	Time when issue was created
Issue_Closed_At	Time when issue was closed
Author	Developer id of the developer who created the issue
Assignee	Developer id of the developer who was assigned the issue
Labels	List of Labels associated with the issue
Issue_Comments	Number of comments
Issue_State	State of the issue- closed/open
Milestone_ID	ID of the milestone associated with the issue

Milestones:
List of all Milestones

Repo_ID	Repository ID
Milestone_ID	Unique Milestone ID
Milestone_Title	Title of the Milestone
Created_Timestamp	Time of milestone creation
Due_Timestamp	Due date of the Milestone
Closed_Timestamp	Closed date of the Milestone
Issues	List of issues assigned to the Milestone

Comments:
List of comments per users for the issues

Issue_ID	Id of the issue
Dev_ID	Id of the developer

3.2 Information Confidentiality

To secure the privacy of the user groups and repositories under our observation, we have replaced the groups and users by simple numeric identifiers. This conversion is done after data extraction from each GIT repositories.

4. Feature Detection

The data extracted in the above step was processed to derive features from them. We applied filters and join conditions to prepare data which can represent our feature group and show better relationship between these features and code smell.

4.1 Feature Extractors

Our feature rubrics has total *13* features in it. These features have been identified as the most influential parameters guiding a project under consideration for its success or failure. These features are:

1. Milestones closed long after due date

There may be some planning issues due to which some Milestones which are created, might not be completed on time and are therefore closed sometime after the expected closing date assigned to that milestone.

2. Milestones closed long before due date

Similarly, there might be some milestones which might have been assigned too much time to be closed but were closed long before that estimated date. This also signifies a lack of proper planning as the time saved could have been assigned to some other task or milestone.

3. Milestones without issues

GIT milestones help to keep track of the ongoing activities in a project. We can distribute the activities as:

a. The ongoing activities - main milestones.
b. The future activities we plan to do - next task milestones.
c. Activities we are not sure about at the moment - issues with no milestones.

It is always helpful to keep track of this last category so as to keep clearing them and involving them in the current iteration of the development.

4. Issues created per user

To judge the participation of each project member, it is important that every member create some issues according to the work they are doing. The number of such issues created by each team member must be comparable to others (at least +-50% of the average) so as to determine if every team member has participated equally enough in the project or not.

5. Issues assigned per user

To ensure that each user has somewhat equal participation in the project, it is important that every team member is assigned with comparable number of issues. The number of issues assigned to each user must be +-50% of the total average so as to ensure equal participation.

6. Issues without milestones

All issues must have a milestone associated with them so that there is a proper planning process going on which ensures that there is proper time and resource management in the project. If there are some issues which are not assigned with a milestone, then this represents lack of proper planning.

7. Issues without labels

It is important that issues which are being created, always have a label associated with them which determines the branch of software engineering process to which that issue belongs. If there are some issues without labels, then it signifies that the process is not being followed carefully.

8. Issues without assignees

All issues must be assigned an assignee so that it is pretty clear to all the members that which person is working on a specific issue and there is no clash or confusion. Issues without assignees are ambiguous and represent lack of proper planning.

9. Comments on Issues per user

It is a good practice to close an issue with comments including keywords showing status of the issue resolution. We can also include keywords in the commit messages to automatically close issues. A fair distribution of comments on issues represent equal contribution from project members.

10. Commits per week

The project assigned was distributed over the course of eight weeks. A proper planned project development show equal distribution of commits over this time period. A project running behind schedule should show higher number of commits during last weeks of deadline.

11. Commits Per User

Every composition or update in GIT snapshot require GIT commit for persisting the changes in the repositories. A uniform distribution of commits among the group member represent equal contribution from the members. This feature represents this key idea to see if there is equal distribution and contribution from each member.

12. Same Commit Message

People often don't use the messages on commits properly to communicate the changes they have made. Users more often try to focus on committing the changes rather than telling their peers what the changes are, and this leads to inconsistency in the understanding of the current code by the different members of the team. The main observations we decided to make regarding this is the same commit message appearing multiple times. While the same commit message may appear a few times, a lot of messages being the same conveys an improper use of commit messages by the committers.

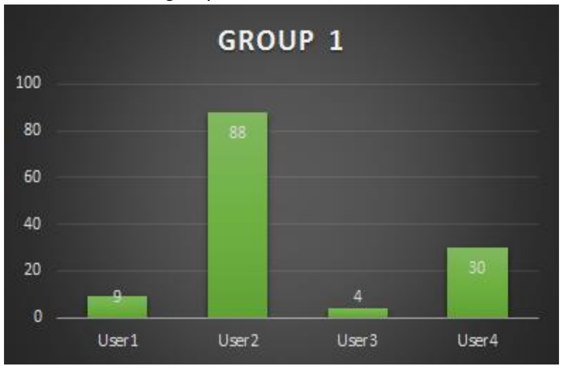
13. Issue without Comment

The issues which are being created need proper comments by the creator to make sure that the assignee understands the issue properly. Sometimes, there are no comments mentioned with the issue, which creates unnecessary doubts and lacks clarity. This should be avoided to ensure that the process is followed smoothly.

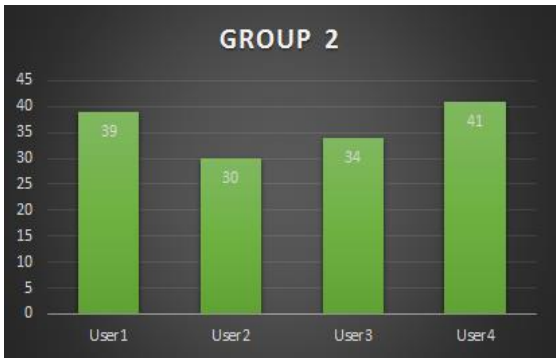
4.2 Features Analysis

We analyzed the data for every repository with respect to each features extracted and created a series of visual graphs as described below:

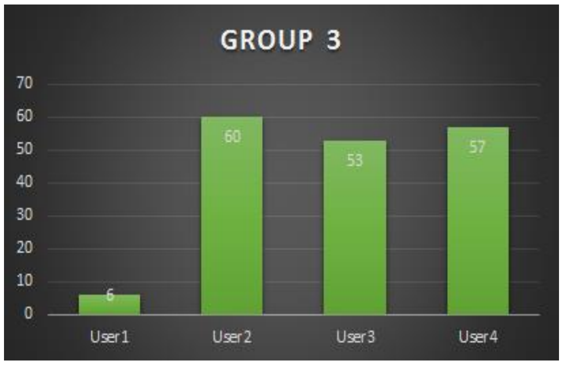
1. **Commits per user:** Each Column Graph corresponds to a group and shows the total number of commits per user in that group.



As we can clearly see that this group has a lot of inconsistencies in this department. The highest number of commits here is 88 whereas lowest is 4. This clearly indicates unequal participation where one team member is hyper-active and other is not so active.

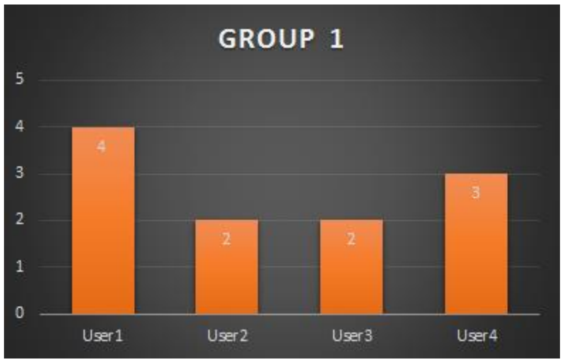


This represents a very consistent participation by each member of the group where highest number of commits is 41 and lowest is 30. So, the number of commits lies well within a range of +-50 % of the average number of commits.

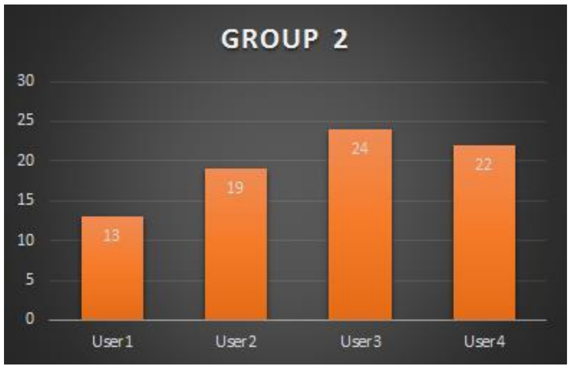


Here, we can see that while 3 members of the group (member 2, 3 and 4) are equally active and have comparable number of commits, member 1 has only 6 commits, which is much lower as compared to other people. This clearly indicates that member 1 is under-active in the project.

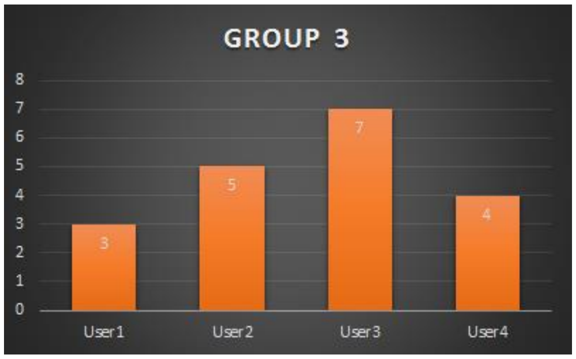
2. **Issues assigned per User:** Each Column Graph corresponds to a group and shows the total number of issues assigned per user in that group.



This data looks consistent for each of the team members as the number of issues assigned are comparable for each member.

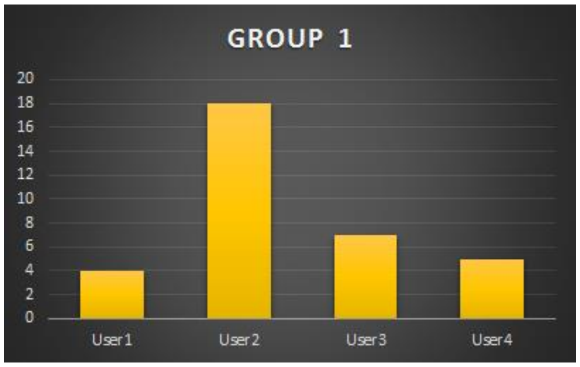


The data looks pretty consistent with all the members being assigned with a good amount of issues and well within the range of +-50% of the average number of issues per person.

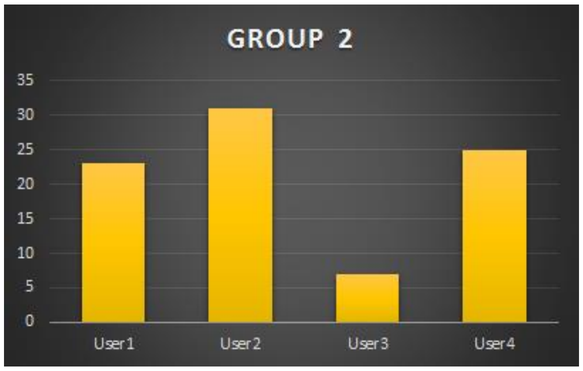


The data looks consistent and every member has been assigned with comparable number of issues. This represents equal participation.

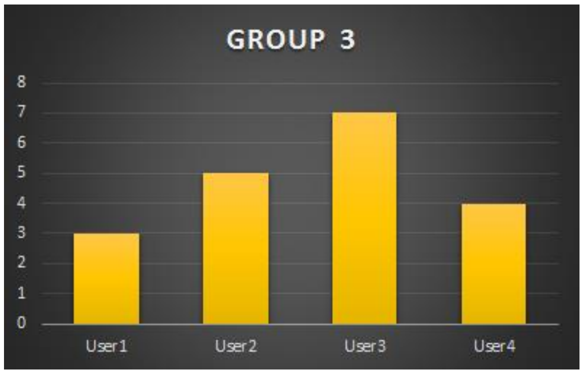
3. **Issues created per user:** Each Column Graph corresponds to a group and shows the total number of issues created per user in that group.



Here, the maximum number of issues created is 18 whereas the average number of issues created per user is around 8. So, there is an inconsistency here because 1 person in the group has created more than 50% of the average issues whereas others have created much less.

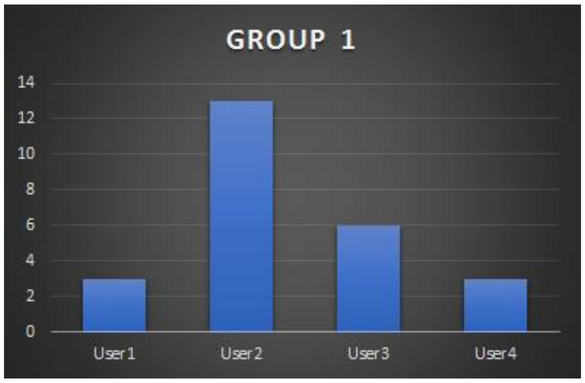


As we can see, the number of issues created by user 3 is clearly less than the other three people. So, this shows an inconsistency in terms of issues created by each member and hence the participation.

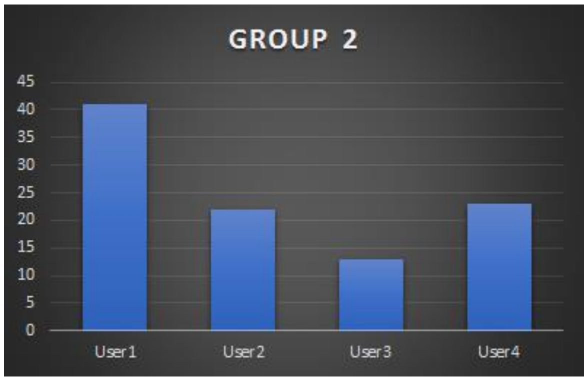


This looks a pretty consistent graph where each member has created comparable number of issues, well within the range of +-50% of the average number of issues created per user.

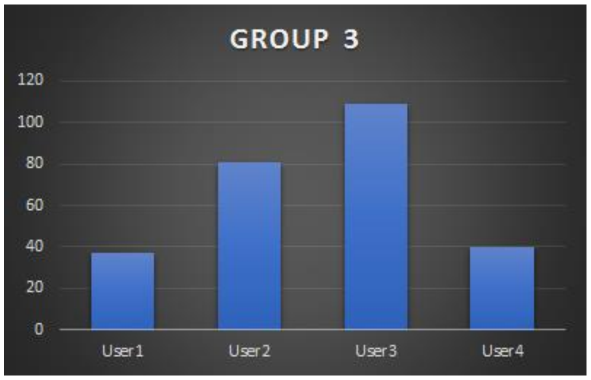
4. **Comments on issues per user:** Each Column Graph corresponds to a group and shows the total number of comments per user in that group.



Here, the average number of comments made is around 6. We can see that user 2 has made more than 50% of the average number of comments which is pretty inconsistent.

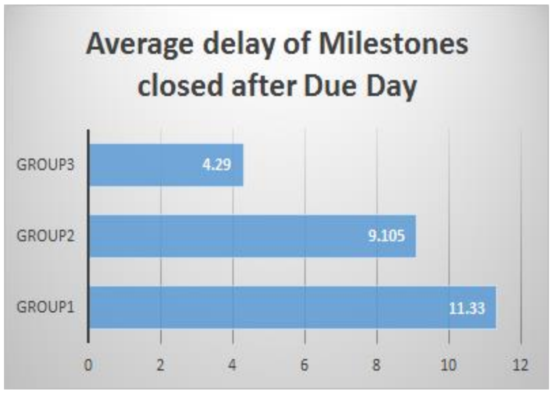


This looks like a pretty decent and consistent data as far as participation of each member is concerned in terms of comments made. The average number of comments here is 25 and all the users have made comments well within the range of +-50% of the average number of comments made.

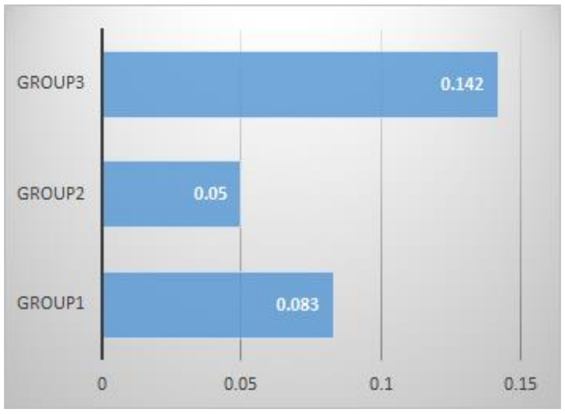


Here, the average number of comments made is around 66 per user. So, we can see from the graph that all the users are within the range of +-50% of the average number of comments which points towards consistency.

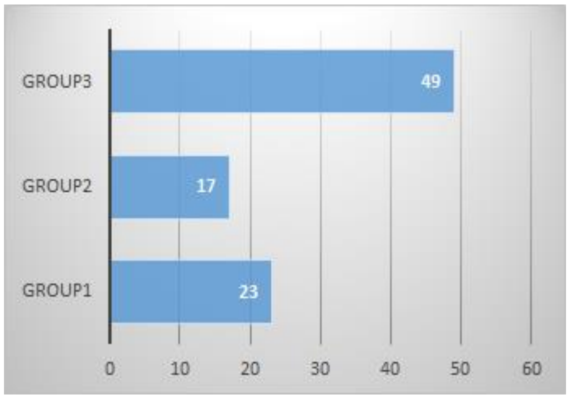
5. **Milestones closed long after due date:** Graph shows the average number of days for milestones which were closed after due date. Higher the average means milestones have been closed long after due date.



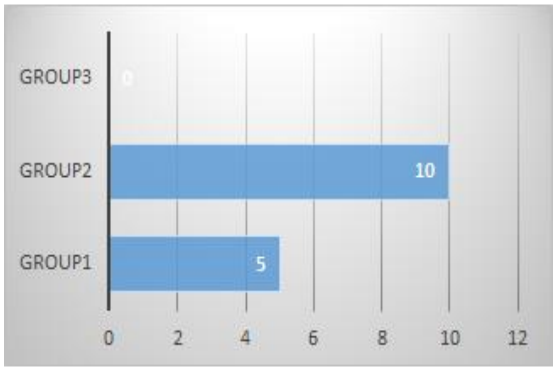
6. **Milestones closed long before due date:** Graph shows the average number of days for milestones which were closed before due date. Higher the average means milestones have been closed long before due date.



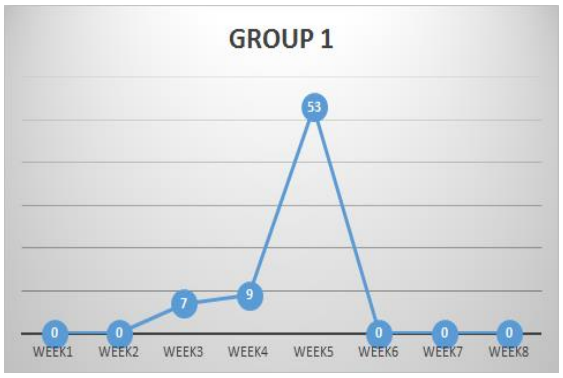
7. Same Commit Message: The below graph show the relationship between issues and the commit messages provided by the user. We have here represented the count of same message repeated by the user during git commit. Higher value of same message used show lack of clarity.



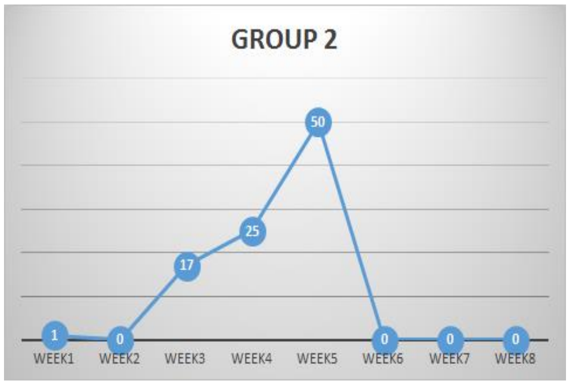
8. Milestones without issues*:* Graph shows the milestones without issues for each group. Clearly Group 3 has all their milestones assigned with issues whereas Group 2 has the most number of Milestones without issues.



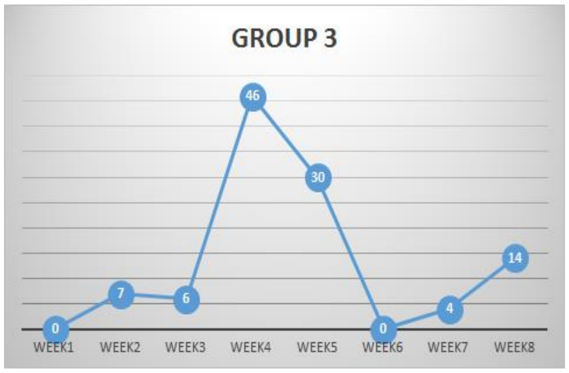
9. Commits per week: Each line graph shows the number of commits made by a group in a 8 week period.



It can be seen from the above graph that this group did not worked consistently and did majority of work in the 5th week and least amount of work in the first 2 weeks.

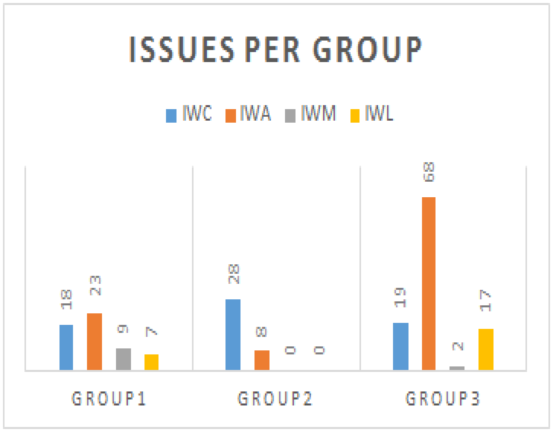


This group also did not work consistently across complete period instead worked majorly in 4th and 5th week.



This group though did not work consistently across weeks but worked in almost week means lesser number of weeks with 0 commits.

10. Issues without comments (IWC), Issues without assignees (IWA), Issues without milestones (IWM) and Issues without labels (IWL): The column graph shows the issues data for each group. The Graph shows that group 2 has least number of issues without milestones and labels whereas group has most number of such issues. Group 1 has average number of such issues.



5. Bad Smells

- *Well it doesn't have a nose... but it definitely can stink*!

The term code smell was first coined by Kent Beck in the late 90s. A bad smell provide a hint about something going wrong in development process. We can sniff these smells to track down the problem at the earliest. Developing code nose earlier in software lifecycle helps to develop an efficient.
As per Wikipedia page of Code Smell:

A 2015 empirical study of half a million code commits to 200 open source software projects found that

- most bad smells affecting a piece of code are already present since its creation, rather than being introduced later via evolutionary code changes
- although code smells are often introduced while adding new features and enhancing existing ones, refactoring activities can also add bad smells
- "Newcomers are not necessary [sic] responsible for introducing bad smells [sic], while developers with high workloads and release pressure are more prone to introducing smell instances", indicating a need for increased code inspection efforts in such stressful work situations.

Instead of code refactoring, our basis for detection of bad smell in this project is features extracted from GIT repositories of the project groups involved. Using the analysis of our features, we grouped them together to represent five code smells detectors.

1. Lazy Worker

A Lazy worker is the person in the group who is less active than the rest of the group members in terms of commits performed or issues generated. Basically, a lazy worker is not playing enough part in the project as other members of the team are. Amongst the features which we we have extracted, the features which can be combined to determine a lazy worker are:

- *Commits per user* - If a user has made significantly less commits as compared to other team members
- *Issues generated per user* - If a user has not generated as many issues as other users have.
- *Issues assigned per user* - If a user has been assigned much lesser number of issues than the other team members.
- *Number of comments per user* - If a user is not commenting as actively as other team members.

The following table and clearly shows the features detected for this bad smell amongst the different repositories we studied, and the percentage of this bad smell each of those repositories have.
(Note: ‘ciu’ stands for commits per user, ‘cou’ stands for comments per user, ‘igu’ stands for issues generated per user, ‘iau’ stands for issues assigned per user and ‘bs’ stands for bad smell as a ratio between 0 and 1)

Group ID	ciu	cou	igu	iau	bs
1	Y	Y	Y	N	0.75
2	N	Y	Y	N	0.5
3	Y	Y	Y	N	0.75

2. Hyper Worker

A Hyper worker is the person in the group who is much more active than the rest of the group members in terms of commits performed or issues generated. Basically, a hyper worker is supposed to be handling the project by himself without much involvement from rest of the team. Amongst the features which we have extracted, the features which can be combined to determine a hyper worker are:

- *Commits per user* - If a user has made significantly more number of commits as compared to other team members
- *Issues generated per user* - If a user has logged much more issues than other users have.
- *Issues assigned per user* - If a user has been assigned much more number of issues than the other team members.
- *Number of comments per user* - If a user is commenting much more than the other team members.

The following table and clearly shows the features detected for this bad smell amongst the different repositories we studied, and the percentage of this bad smell each of those repositories have.
(Note: ‘ciu’ stands for commits per user, ‘cou’ stands for comments per user, ‘igu’ stands for issues generated per user, ‘iau’ stands for issues assigned per user and ‘bs’ stands for bad smell as a ratio between 0 and 1)

Group ID	ciu	cou	igu	iau	bs
1	Y	Y	Y	N	0.75
2	N	Y	N	N	0.25
3	N	Y	Y	N	0.5

3. Improper Planning/Estimation

Improper planning or estimation is a bad smell which represents if the team has made proper plans before starting a task or not. This can be determined mainly by the following features:

- *Milestones closed long after due date* - If the milestones being created are getting closed much after the desired deadline, this shows that there has been a lack of good planning while creating that milestone and more time needed to be allotted.

- *Milestones closed long before due date* - If the milestones are being closed much before the actual deadline, then this is also a problem of improper planning.

The following table and clearly shows the features detected for this bad smell amongst the different repositories we studied, and the percentage of this bad smell each of those repositories have.
(Note: ‘cad’ stands for milestones closed long after due date, ‘cbd’ stands for milestones closed long before due date and ‘bs’ stands for bad smell as a ratio between 0 and 1)

Group ID	cad	cbd	bs
1	Y	N	0.5
2	N	N	0.0
3	N	N	0.0

4. Improper use of GitHub

Improper use of GitHub refers to not following the Software Engineering process properly. There are different phases of the SE process which need to be taken care of while making use of GitHub repository. This improper use can be defined using the following features:

- *Milestones without issues* - If there are no issues created inside a milestone, then the purpose of milestone is not served well and the meaning is ambiguous. This contributes to improper use of GitHub.
- *Issues without milestones* - If there are random issues raised and have no milestones associated to them, it is not a very appropriate use of GitHub and should be avoided.
- *Issues without labels* - All the issues raised must have some labels attached to them so as to represent which branch of Software Engineering process those issues belong to. Not assigning labels to issues is not a good practice as it does not clearly define an issue and its classification. This should be avoided.
- *Issues without assignees* - If an issue has not been assigned to an assignee, that does not make much sense and is not a very good use of GitHub. Every issue must be assigned to a user who is required to work on that issue.
- *Issues without comments* - If people don’t comment on issues, then it also doesn’t make a lot of sense because issues are meant for proper communication between team mates, and no matter how smart we humans think we are, we need clarifications and further help. Thus most of the issues should have some comment on them.
- *Same commit Message* - If many commits have the same message, it means that some of the team members have not tried to be clear about their changes. This can tend to develop much more complex understanding issues about the current code lying in the repositories and clearly shows an improper use of GitHub.

The following table and clearly shows the features detected for this bad smell amongst the different repositories we studied, and the percentage of this bad smell each of those repositories have.
(Note: ‘mwi’ stands for milestones without issues, ‘iwm’ stands for issues without milestones, ‘iwl’ stands for issues without labels, ‘iwc’ stands for issues without comments, ‘iwa’ stands for issues without assignees, ‘scm’ stands for same commit messages, and ‘bs’ stands for bad smell as a ratio between 0 and 1)

Group ID	mwi	iwm	iwl	iwc	iwa	scm	bs
1	N	N	N	N	N	N	0.0
2	Y	N	N	N	N	N	0.17
3	N	Y	Y	N	Y	Y	0.67

5. Inconsistency in work rate

Inconsistency in work rate means that the team has not been active equally throughout the duration of the project. It might be the case that the team is more active during some period of time and less or not at all active during some. This can be determined by analyzing the following features:

- *Commits per week* - By determining the number of commits a team makes on a weekly basis, we can easily determine whether a team has been active equally over the entire duration of the project or not. There might be some weeks when commits are much more than the other weeks which means that much of the project work was done in that week. On the other hand there might be weeks in which there is little or no commit activity at all and can easily lead to conclusion that those weeks were passive weeks for the team.

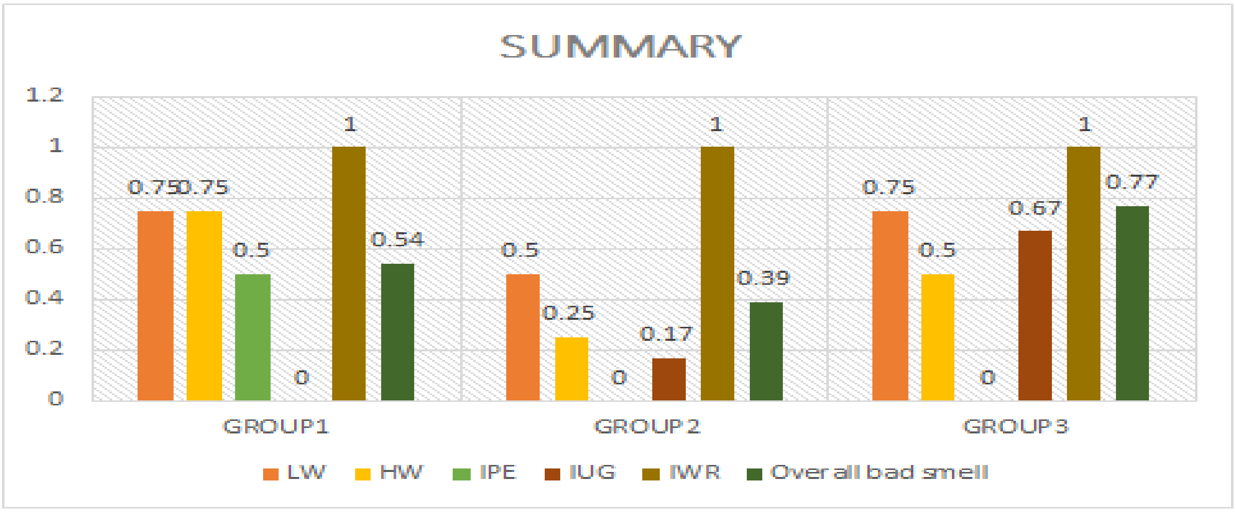
The following table and clearly shows the features detected for this bad smell amongst the different repositories we studied, and the percentage of this bad smell each of those repositories have.
(Note: ‘cw’ stands for commits per week and ‘bs’ stands for bad smell as a ratio between 0 and 1)

Group ID	cw	bs
1	1	1.0
2	1	1.0
3	1	1.0

Further we combine all these bad smells to find out what is the overall bad smell of each repo. This is basically a weighted mean of the different bad smells defined, where

Group ID	lw	hw	ipe	iug	iwr	obs
1	0.75	0.75	0.5	0	1	0.54
2	0.5	0.25	0	0.17	1	0.39
3	0.75	0.5	0	0.67	1	0.77

Note: **'lw'** stands for lazy worker, **'hw'** stands for hyper worker, **'ipe'** stands for improper planning/estimation, **'iug'** stands for improper use of github, **'iwr'** stands for inconsistency in work rate, and **'obs'** stands for overall bad smell.



the weights are the number of features under each bad smell. Result for the same can be seen in the table and the graph below.

6. Conclusion

After analyzing our results we came at a conclusion that different groups work in different ways. This leads to a development of different kind of bad smells for every group, but there certainly are bad smells. This clearly shows that the way teams work needs some kind of optimization. Further, we found out that some of our bad smell detectors were not so smart after all. While we were able to collect data on milestones closed long before due date, and issues without assignees, either the data collected was not sufficient or the way we measured outlier or unusual values was not that good, such that these did not yield good results. With this we conclude that developers generally work in a manner that they generate lot of bad smells, the most commonly occurring of these being the inconsistency in work rate and the case of a lazy worker. With more data, these bad smells could be better detected and more bad smells could be defined. But above everything, there is a dire need to deal properly with the software development process in order to reduce these bad smells and make the overall development process better.

7. References:

1. https://en.wikipedia.org/wiki/Code_smell
2. http://www.jot.fm/issues/issue_2012_08/article5.pdf
3. http://www.cs.wm.edu/~denys/pubs/ASE'13-HIST-Camera.pdf
4. https://en.wikipedia.org/wiki/Code_smell#cite_note-:0-1
5. https://sourcemaking.com/refactoring/smells