FSS - Exercise Software Evolution

Team members:
Blum Michael - 17-717-232
Degkwitz Dimitri - 13-928-734

Project Repository:
https://github.com/degenwitz/FFS_SoftEvo

1) Complexity Hotspots

1.1) Repository of interest

Throughout this exercise we are working with the source code of the IPFS implementation written in the programming language Go.

1.2) Granularity

We focus our attention and analysis on **source code files** (ending in .go) only, discarding any additional files from the repository (e.g. README.md files or other tools).

1.3) List of files

There are a total of 407 files at the starting point of our timeframe (commit 20eabf87ad834dd1b36decc8d47aa07a2adb1ecf). This includes all .go files from the whole repository. Until the release of the final version there were 4890 commits and the end count of files accumulates to 283.

1.4) Complexity metrics

Our analysis takes **lines of code** as the metric for complexity. Other metrics were considered and partially derived (i.e. cyclomatic complexity), but for the scope of this assignment we limited ourselves to the easily accessible metric of lines of code (loc).

1.5) Timeframe

Our considered **timeframe** starts at the 1st of October 2017 and ends at the 30th of September 2021 (release of version 0.10.0). We chose this time frame since this was the first time IPFS got international attention outside of tech-enthusiasts. The conflict in spain concerning catalan independence lead the spanish government to block political websites. Some of them, mainly the Catalan Pirate Party, used IPFL to circumvent the block. This marks a change in the use-cases of IPFL, as it was for the first time important for real world changes. By using this date we hope to start with a phase were the developers found new motivations to do their best work on this project.

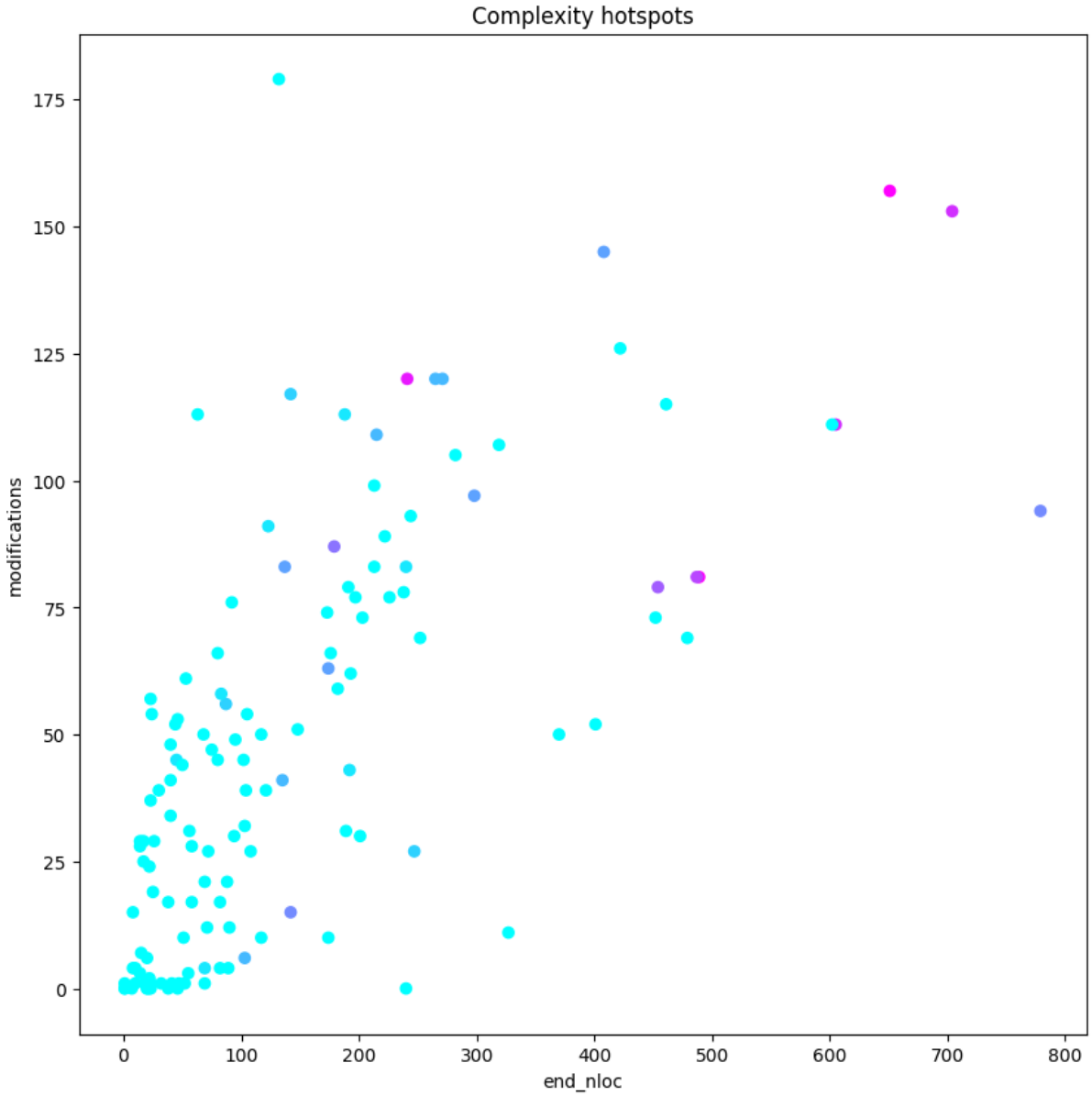
1.6) Complexity measurement

For the determination of potential hotspots we calculate a weighted complexity measurement for each file as follows:
*complexity = loc + added_loc * 3 + amount_of_modifications * 20*
We opted for this measurement in order to merge the base complexity of the file (loc) together with metrics about its modifications (added loc and amount of modifications). The weights were chosen based on educated guesses after inspection of the data.

file name	**calculated complexity**
core/corehttp/gateway_handler.go	4388
cmd/ipfs/daemon.go	4256
core/commands/add.go	3530
core/corehttp/gateway_test.go	3299
core/coreapi/unixfs.go	3175
core/commands/swarm.go	2953
core/commands/get.go	2857
core/commands/dag/dag.go	2854
core/coreunix/add.go	2738
core/commands/config.go	2727

1.7) Visualization of hotspots

The following graphic illustrates the complexity of the source code files. The x-axis represents the lines of code of the file, the y-axis models the amount of commits modifying this file and the color is a linear indicator of the amount of lines added over the timeframe (light-blue means few lines added, purple/pink mens lots of lines added).



1.8) Analysis of hotspots

Based on the visualized metrics above, we decided to perform our analysis on the following six candidate hotspots:

We have decided to perform a manual analysis on six of the ten files displayed above based on interesting characteristics:

file name	**manual analysis**
core/corehttp/gateway_handler.go	We can see that the gateway handler in the past has not been changed much. This indicates that it was

	mostly usable and working as intended and was not expanded. In the last years we see that a lot of code was added without removing nearly as many, indicating an increase of responsibility for this class. The developers should maybe consider extracting the new responsibilities to a new class.
cmd/ipfs/daemon.go	This daemon file is fairly large (900+ loc) and its functionality is crucial for the application. It contains many variable initializations, which very changed frequently over the course of the development.
core/commands/swarm.go	This file contains many commands used by other part of the application. It has been extended multiple times.
core/commands/dag/dag.go	Another command file that handles dag objects. We assume that dag objects are crucial for IPFS and these methods therefore are highly maintained.
core/coreunix/add.go	The add function contains class contains a lot of constant strings that are displayed to the user. It is more common to put these kinds of information in a separate file, to keep code clean and readable. this might also explain why add is mainly plagued by a constant adding and removing of lines, as the descriptions get more refined and new functionalities are added, that need to be explained.
core/commands/config.go	The fact that IPFS is growing as a service will lead to a higher demand of support for different users. It is to be expected that the designers are hearing this demand and are expanding the possibilities of different configurations. This can easily happen passively in an organization, since many services come with new configurations. This can lead to many different people changing the config.go file, when they need new functionality. As we can see, the file mainly shrink in big chunks. This indicates that there is a person keeping their eye on the config.go class and cleaning it up regularly.

2) Temporal / Logical Coupling

2.1) Selection of coupled entities

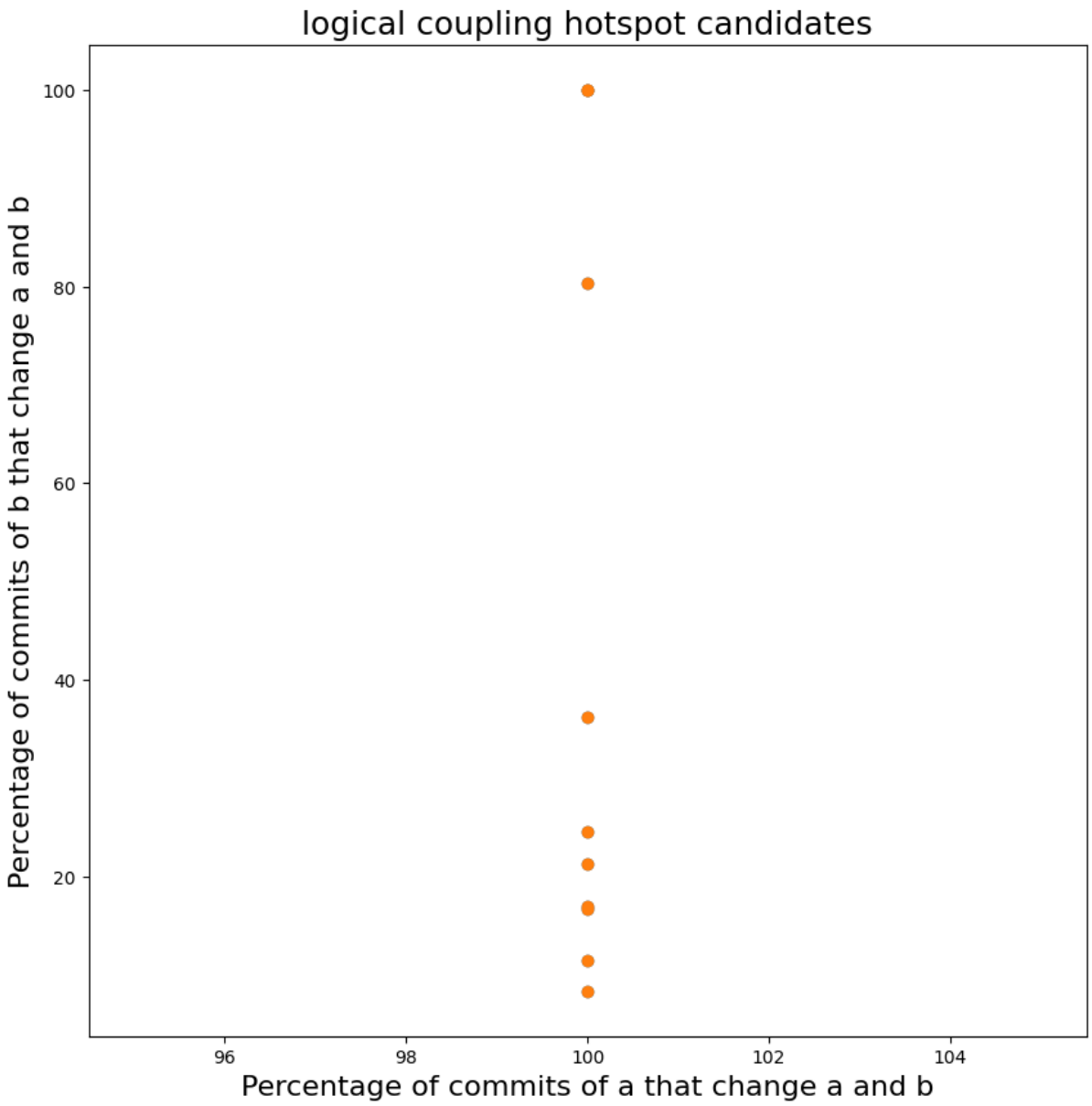
In order to figure out which files are logically coupled, we measure which files are committed together (e.g. if a commit changes file A and B, we assume that A and B are logically coupled). We do not investigate temporal coupling in our measurement.

The implementation of our algorithm resulted in the following 6 files that have the highest logical coupling:

File B	File A	commits % of commits of A that also change B	commits % of commits of B that also change A
test/bench/bench_cli_ipfs_add/main.go	'test/bench/offline_add/main.go	100%	100%
test/bench/offline_add/main.go	est/bench/bench_cli_ipfs_add/main.go	100%	100%
repo/fsrepo/fsrepo.go	repo/fsrepo/misc.go	100%	36%
fuse/ipns/ipns_test.go	repo/fsrepo/fsrepo.go	100%	12%
core/corerepo/gc.go	core/corehttp/logs.go	100%	17%
core/core.go	plugin/ipld.go	100%	8%

2.2) Visualization of candidate sets

For each pair of files we measured the relative coupling from the point of each file, because the coupling is bidirectional (e.g. all commits changing file A might also change file B, but there might be commits that change file B without changing file A).



2.3) Analysis of coupled files

Many of the coupled files are test-cases and are coupled with the classes they are testing. Generally speaking this is to be expected, they are made to test functionalities of other classes and often small modifications to the class under tests are needed to ensure testability. What is more worrying is the opposite, a class that is coupled with its test. This means that instead of writing tests that check functionality, tests need to be changed when the code changes to match the new code.
The Core of IPFS seems to the coupled a lot. This is very typical, since the core is often the oldest part of an implementation and was written when standards in the organization were lower. It is time-consuming to decouple the core and often people who implemented it are no longer around, making this process harder. But it is necessary, since it will become a problem somewhere down the road.

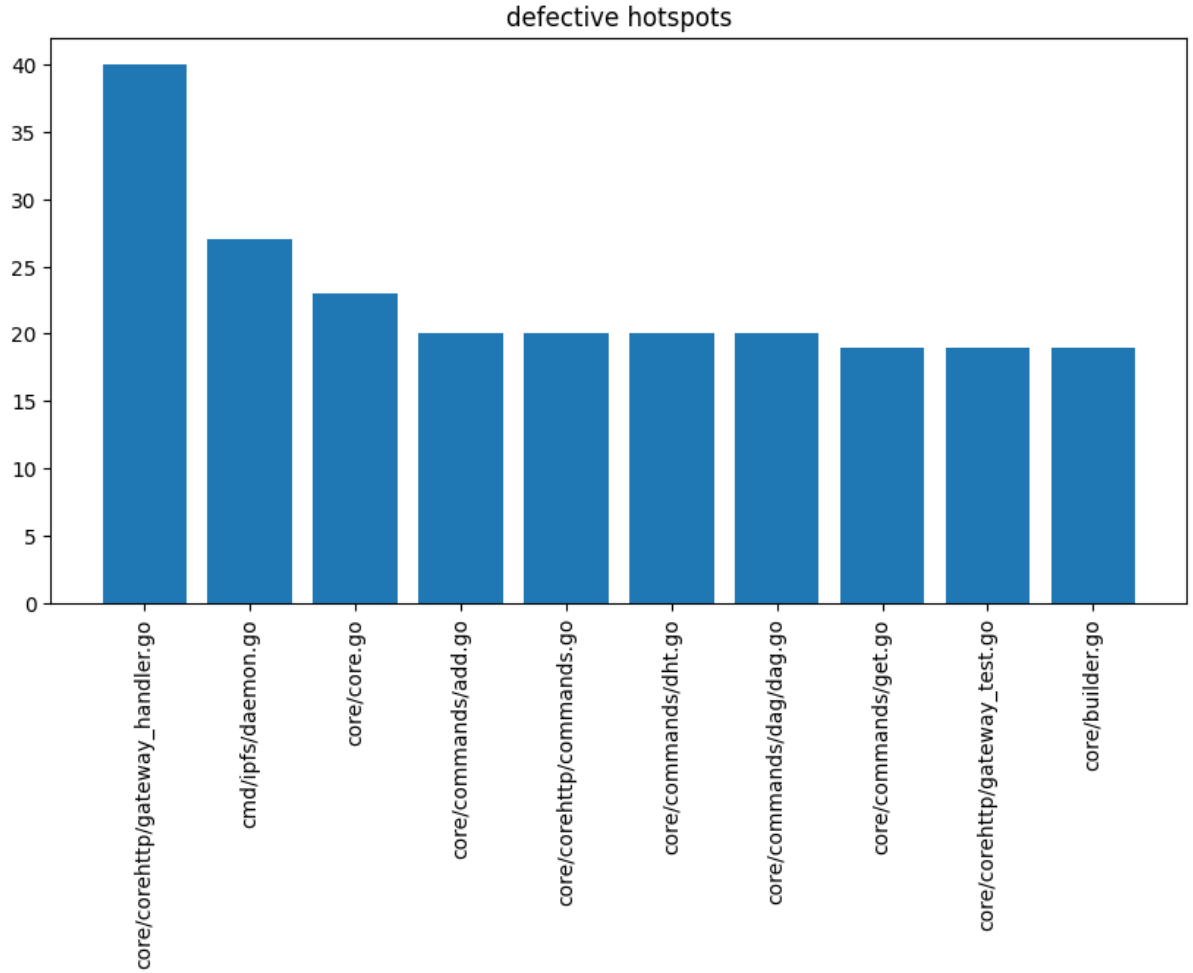
3) Defective Hotspots

3.1) Methodology

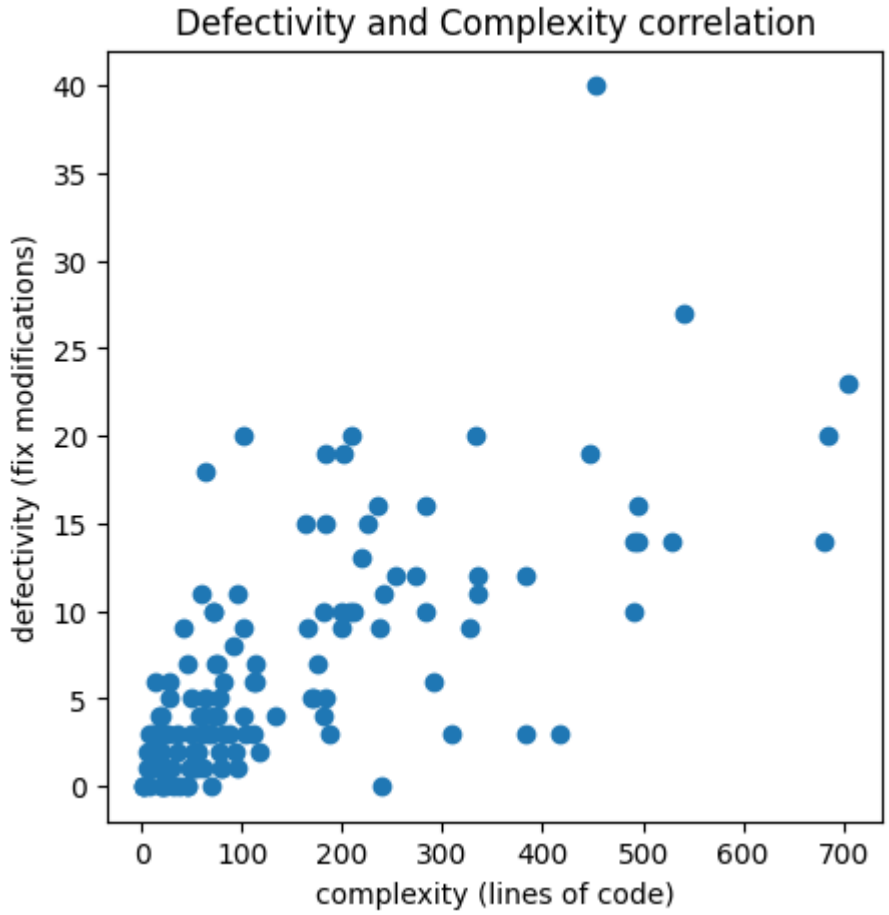
In our approach to detect files that are most defective, we analyzed all commit messages in our defined timeframe. We noticed that there is a common practice in place for the developers of the repository in tagging bug-fixes with a *fix:* keyword. Based on this observation we performed a basic text analysis looking for this specific or related keywords and consequently labeled them as a fix commit if one of the keywords is found.

3.2) Detection of defective hotspots

The results of our methodology resulted in the following ranked list of files that can be considered as hotspots for implementation errors:



3.3) Correlation of complexity and Defectivity



3.4) Interpretation of Results

The results mirror our expectations about code quality: With increasing complexity of a file, the defectivity increases as well. In this sense there is a clear correlation. Of course there are exceptions, but it is noteworthy that every file with at least 300 lines of code has been fixed at least four times.

We are pleasantly surprised by the results, given that we used rather trivial metrics. We conclude that these basic measurements are good enough to perform repository analysis. Nevertheless, we would opt for more complex metrics for a more serious and insightful analysis.