Analysis of Software Design Principles under Complex Network Theory

Juan Pablo Royo Sales & Francesc Roy Campderrós

Universitat Politècnica de Catalunya

January 20, 2021

Contents

1 Introduction

One of the most well known Software Design principles in **Software Engineering** is High Cohesion (High Cohesion) and Low Coupling (Low Coupling), which is well described here [You79].

As this two principles states a *robust Software* should be design with Low Coupling between their modules and High Cohesion inside it.

In other words, a Software that fullfil this characteristics should be very connected in their minimum Functional Units (FU) (Functions inside same file, Methods inside a class, etc), and with few connections between their coarse grained FU (a.k.a Modules or Packages).

In this work, we are going to formulate some hypothesis which we believe it can been empirically proved and shown the relationship between these principles and how to measure them with **Complex Network Theory (Complex Network Theory)**. At the same time, we are going to analysis different kinds of software of different sizes and build under different language paradigms to see if the tool set that Complex Network Theory provides are suitable for the general case.

2 Preliminaries

In this section we are going to describe how and why the different Language Paradigms are selected and what is the criterion for selection of different Software solutions to be evaluated.

On the other hand as well, we are going to formulate some hypothesis that are going to guide our work to see if our assumptions can empirically been proved using Complex Network Theory.

2.1 Context

We have selected the most important 2 main Language Paradigm to conduct the analysis: Functional Programming (Functional Programming) and Object Oriented Programming (Object Oriented Programming).

The reasons behind this decision are basically the following:

- **95%** of Software in the Industry are built with one of these 2 Paradigms according to the last results of this well-known survey [Inc20].
- Due to the intrinsic nature of each of those Paradigms we have some hypothesis that we are going to describe later that can lead to different conclusion and Metrics
- If we can deduce some Software Design properties analyzing these 2 Paradigms we can generalize for the rest because they are quite different in nature and covers almost the whole Industry.
- We also believe that Software Principles should apply indistinguishably the Paradigm.

On the other hand the selection of the programs to be analyzed are the following:

- Most of the software are Open Source or Free software that can be download publicly either from [Inc21] or from [Cen21].
- Software that are marked as **PRIVATE** are Big Projects from Privates Companies that don't want to reveal the Sources Code and Names for Commercial reasons.
- In the case of **PRIVATE** Functional Programming Solution, it belongs to a Company one of the authors of the current work is working right now.
- In the case of **PRIVATE** Object Oriented Programming Solution, it belongs to a Company one of the authors of the current work worked in the past.
- In both cases, taken anonymous data for conducting this analysis has been agreed with legal representatives of those Companies.

On the last hand we are going to use *Haskell* Programs for analyzing Functional Programming and *Java* Programs for Object Oriented Programming. We believe that right now both are the most representative ones in their Paradigm fields.

2.2 Hypothesis

In this work we are trying to prove the following **Hypothesis** that we consider can be proved using Complex Network Theory (Complex Network Theory).

Hypothesis 1. *Given any Software Program Solution, its Modularity Metric should be between the* 1*st Quartile and* 3*rd Quartile, according to the average of the Network Metrics that we have been identified in this work, to be considered as a well Software Designed Solution.*

Hypothesis 2. *Any Object Oriented Programming Program have a better modularity in terms of Complex Network Theory Metric rather than Functional Programming Programs.*

Hypothesis 3. *The more Lines of Code (LoC) a Program have, the better Modularity it presents.*

Hypothesis 4. *If the Software follows the principle design of High Cohesion and Low Coupling, the Degree Distribution (Degree Distribution) of the Generated Graph should follow a power-law like.*

3 Results

In this section first we are going to describe the **Experiments** conducted and after that we have obtained after running the different experiments; this is what we call **Metrics** subsection.

3.1 Experiments

In this section we are going to described how the experiment have been set up in order to prepare the graphs for taking the desired metrics, that could allow us to explain and verify the proposed hypothesis.

3.1.1 Preliminaries

In order to determine if a Software fullfil the 2 Software Design Principles that we want to analyze, High Cohesion and Low Coupling, we need to extract the Call Dependency Graph (CDG) from the programs that we want to analyze.

In CDG, **nodes** are Functional Units: *Functions* in the case of Functional Programming and *Methods* in the case of Object Oriented Programming. An **edge** is when from inside a FU another FU is called or invoked.

Therefore, we need to build this Call Dependency Graph in order to establish how the different Modules are interconnected in order to measure Low Coupling and High Cohesion.

3.1.2 Building Graphs from Programs

In order to achieve the desired Call Dependency Graph we are going to use some specific tooling for each Paradigm. The following has been used on each case:

- **Functional Programming**: *function-call-graph* [dus19] is a Program that given a *Haskell* Source Code it outputs a *DOT* file with the Call Dependency Graph
- **Object Oriented Programming**: *java-callgraph* [gou18] is a Program that given a *Java* Compiled Jar it outputs a in *stdout* the Call Dependency Graph. In order to use only *DOT* files we have converted this output into *DOT* using a script that is under `code/script_java.sh`

All the resulting graphs are under their respective folders as you can see here A.

3.1.3 R Scripts

We have only one *R Script* where we concentrate all the logic to obtain metrics and plots.

Basically, the most important function is the following one:

```
run_metrics <- function(file){
  graphName <- get_graph_name(file)
  adj_matrix <- read.dot(file)
  colnames(adj_matrix) <- c(1:length(adj_matrix[,1]))
  rownames(adj_matrix) <- c(1:length(adj_matrix[1,]))
  graph <- graph_from_adjacency_matrix(adjmatrix = adj_matrix,
   ↪  c("undirected"))

  table <- data.frame()
  E = length(E(graph))
  N = length(V(graph))
  k = 2 * E / N
  delta = 2 * E / (N * (N - 1))
  MGD = average.path.length(graph)
  DIAM = diameter(graph)

  communities <- walktrap.community(graph)

  modul <- modularity(communities)

  count_communities <- max(communities$membership)

  mean_cc_coef <-
   ↪  sum(sna::closeness(adj_matrix,gmode="graph",cmode="suminvundir"))/N

  mean_lc_coef <- transitivity(graph, type = "average")

  graph_d = degree.distribution(graph)
  plot(graph_d, log = "xy", ylab="Prob.", xlab="Dependency
   ↪  Call", main = paste("Program: ", graphName))
  return(c(N, E, round(k, 2), round(delta, 4),round(MGD,
   ↪  4),DIAM, round(modul, 4), count_communities,
   ↪  round(mean_cc_coef,4), round(mean_lc_coef,4)))
}
```

Listing 1: Extracted from source code code/Script.R

As it can be appreciated on highlighted lines, we are building the graph from the *DOT* files, and the rest of the `run_metrics` function is the well-known metrics that we have explored before.

On the other hand we've also built an *R Script* that can be found in `code/modularity_vs_linesC` that plots the Modularity vs. Lines of Code, which is going to be useful for the analysis of hypothesis 3.

Lately we have also built `code/degree_distribution_fit.R` that is going to be useful for analyzing hypothesis 4. This last script is based on the Analysis of Degree Distribution that we have conducted on Lab 2.

3.1.4 Testing Programs

We have selected randomly 2 programs 1 Functional Programming (Functional Programming) program and other Object Oriented Programming (Object Oriented Programming) in order to test against the metrics obtained and check if certain **hypothesis** holds or not.

3.2 Metrics

Regarding the metrics we are gathering the following ones:

- **N**: Number of nodes. In our particular set up is the number of Functions or Methods and give us an idea of the big of the program. This can be also check by B where we have how many Lines of Code each program has.
- **E**: Number of vertices or the amount of calls that are taking place in the program
- **K**: Mean Degree Distribution. This will allow us to see how is the connectivity of the Graphs
- **MGD**: Average Path Length.
- **Diameter**
- **Modularity**: This is a quite important metric. Although there is a work [ZWXY18] that is proposing a better metric for modularity to measure High Cohesion in a program, we believe that already explored Modularity metric is enough and it covers Low Coupling as well.
- **Communities**: We believe that the number of communities should be similar of the number of modules, although it is out of the scope of this work exploring that and we think it could be an interesting topic to discuss in a future work.
- **Mean Clustering Coefficient (Mean CC Coef.)** and **Transitivity (Mean LC Coef.)**: We also believe that this 2 metrics are useful

for measuring High Cohesion and Low Coupling alongside with **Modularity** because it is going to help us to understand if there is some concentration of calls in one particular Functional Units.

3.2.1 FP Programs

Table 1: FP Programs Metrics 1

Program	N	E	K	Delta	MGD	Diameter
aeson	373	1167	6.26	0.0168	3.2450	9
amazonka	739	2366	6.40	0.0087	3.3202	8
async	60	120	4.00	0.0678	2.2458	6
attoparsec	61	180	5.90	0.0984	2.3978	5
beam	852	2215	5.20	0.0061	4.4898	11
cabal	2294	9115	7.95	0.0035	3.6017	10
co-log	97	159	3.28	0.0341	3.8937	9
conduit	457	875	3.83	0.0084	3.4084	9
containers	61	125	4.10	0.0683	3.3486	8
criterion	71	143	4.03	0.0575	3.3851	6
cryptol	1803	6540	7.25	0.0040	3.4149	9
cryptonite	292	652	4.47	0.0153	3.9468	9
dhall	707	2100	5.94	0.0084	3.6749	10
free	148	328	4.43	0.0302	5.0257	11
haskoin	569	1252	4.40	0.0077	5.0333	12
hedgehog	567	1383	4.88	0.0086	4.0136	12
helm	66	97	2.94	0.0452	3.4014	6
hlint	266	624	4.69	0.0177	3.1839	10
lens	1118	3908	6.99	0.0063	2.7743	7
liquid	2568	7742	6.03	0.0023	3.3709	12
megaparsec	107	191	3.57	0.0337	4.4812	9
mios	169	397	4.70	0.0280	3.8088	10
optparse	174	464	5.33	0.0308	3.0912	7
pandoc	3640	15951	8.76	0.0024	3.3057	9
pipes	100	250	5.00	0.0505	2.2760	3
postgresql	501	1198	4.78	0.0096	3.8283	8
protolude	106	193	3.64	0.0347	4.0385	9
QuickCheck	264	699	5.30	0.0201	3.1726	8
reflex	222	425	3.83	0.0173	3.4672	12
relude	209	267	2.56	0.0123	6.0423	13
servant	237	445	3.76	0.0159	4.5035	11
snap	220	481	4.37	0.0200	3.8888	10

Table 1: FP Programs Metrics 1

Program	N	E	K	Delta	MGD	Diameter
stm	70	171	4.89	0.0708	2.4576	5
summoner	194	458	4.72	0.0245	4.1549	10
text	105	170	3.24	0.0311	2.1471	4
vector	399	2953 1	4.80	0.0372	2.2036	6
yesod	367	748	4.08	0.0111	3.9486	12
PRIVATE	1088	2239	4.12	0.0038	4.7322	13

Table 2: FP Programs Metrics 2

Program	Modularity	Communities	Mean CC Coef	Mean LC Coef
aeson	0.4826	14	0.3521	0.3367
amazonka	0.4875	36	0.3273	0.2297
async	0.3403	9	0.4868	0.4141
attoparsec	0.2458	16	0.4757	0.2906
beam	0.6363	37	0.2488	0.2455
cabal	0.4939	96	0.2981	0.1915
co-log	0.6203	11	0.3144	0.4594
conduit	0.4621	35	0.3211	0.3221
containers	0.4138	5	0.3692	0.3520
criterion	0.6054	6	0.3516	0.3147
cryptol	0.5320	79	0.3169	0.2060
cryptonite	0.5825	25	0.2886	0.2790
dhall	0.5832	29	0.2975	0.2485
free	0.5749	15	0.2570	0.4042
haskoin	0.7069	24	0.2270	0.2604
hedgehog	0.5792	44	0.2500	0.2757
helm	0.5655	4	0.3585	0.4360
hlint	0.4633	40	0.3589	0.3124
lens	0.4207	24	0.3884	0.4447
liquid	0.3388	19	0.3229	0.2057
megaparsec	0.6245	6	0.2847	0.2359
mios	0.5310	16	0.3081	0.2211
optparse	0.4466	8	0.3665	0.2248
pandoc	0.4614	22	0.3248	0.2379
pipes	0.3251	6	0.4704	0.6419
postgresql	0.6063	16	0.3060	0.5070
protolude	0.5651	8	0.3174	0.3897
QuickCheck	0.4640	19	0.3605	0.3677

Table 2: FP Programs Metrics 2

Program	Modularity	Communities	Mean CC Coef	Mean LC Coef
reflex	0.5238	22	0.3469	0.4267
relude	0.7759	14	0.2051	0.3334
servant	0.6871	20	0.2463	0.3390
snap	0.5939	19	0.2894	0.2925
stm	0.3033	9	0.4538	0.3540
summoner	0.5757	17	0.2866	0.1518
text	0.3191	17	0.4863	0.5178
vector	0.1458	17	0.4825	0.4041
yesod	0.5858	13	0.2932	0.2731
PRIVATE	0.6461	63	0.2323	0.1480

3.2.2 FP Programs - Summary

Table 3: FP Programs - Summary Metrics 1

Type	N	E	K	Delta	MGD	Diameter
Min.	18.0	18	2.000	0.00230	2.147	3.000
1st Qu.	105.5	192	3.915	0.00850	3.214	7.500
Median	237.0	481	4.690	0.01770	3.419	9.000
Mean	547.7	1764	5.036	0.02786	3.593	8.872
3rd Qu.	568.0	1742	5.615	0.03440	3.981	10.500
Max.	3640.0	15951	14.800	0.11760	6.042	13.000

Table 4: FP Programs - Summary Metrics 2

Type	Modularity	Communities	Mean CC Coef	Mean LC Coef
Min.	0.1458	4.00	0.2051	0.1480
1st Qu.	0.4540	10.00	0.2876	0.2369
Median	0.5320	17.00	0.3174	0.3124
Mean	0.5097	38.08	0.3315	0.3201
3rd Qu.	0.5898	32.00	0.3597	0.3969
Max.	0.7759	419.00	0.4868	0.6419

3.2.3 OOP Programs

Table 5: OOP Programs Metrics 1

Program	N	E	K	Delta	MGD	Diameter
akka	291	290	1.99	0.0069	5.0948	12
commons-cli-1.	133	178	2.68	0.0203	4.9707	13
commons-codec-	443	599	2.70	0.0061	6.0180	16
commons-csv-1.	168	177	2.11	0.0126	2.8243	8
commons-email-	182	202	2.22	0.0123	4.7634	14
disruptor-3.4.	457	632	2.77	0.0061	6.4273	16
ftpserver-core	1280	2319	3.62	0.0028	5.9379	17
grpc-core-1.34	3717	6544	3.52	0.0009	7.3362	24
guava	11786	18082	3.07	0.0003	7.4965	28
hbase-client-2	1724	1661	1.93	0.0011	8.4558	20
hsqldb-2.4.1	10394	22726	4.37	0.0004	6.6143	29
jackson-databi	1637	2718	3.32	0.0020	6.9562	25
javax.servlet-	440	387	1.76	0.0040	7.8849	19
jedis-3.4.1	4413	6625	3.00	0.0007	7.1696	25
jersey-core-1.	1409	2046	2.90	0.0021	6.4969	19
jetty-7.0.0.pr	1881	2947	3.13	0.0017	6.7668	24
joda-time-2.10	3901	8201	4.20	0.0011	6.3395	22
jsch-0.1.54	1314	2296	3.49	0.0027	5.5298	16
jsoup-1.13.1	1623	3558	4.38	0.0027	6.2448	17
junit-4.13.1	1602	2186	2.73	0.0017	7.0309	20
mail-1.4.7	1068	1474	2.76	0.0026	9.4969	29
mariadb-java-c	2319	3678	3.17	0.0014	8.3089	24
mongo-java-dri	9147	16481	3.60	0.0004	6.8356	27
mx4j-3.0.2	887	1192	2.69	0.0030	6.4340	18
org.eclipse.jg	12059	23175	3.84	0.0003	6.3935	22
pdfbox-2.0.22	8372	16353	3.91	0.0005	6.0460	28
poi-4.1.2	15971	26910	3.37	0.0002	7.5505	28
postgresql-42.	3164	4740	3.00	0.0009	6.8974	20
resteasy-jaxrs	4289	6818	3.18	0.0007	6.2748	18
runtime-3.10.0	441	547	2.48	0.0056	5.2854	13
slf4j-api-1.7.	180	182	2.02	0.0113	4.8468	11
spring-securit	1501	2314	3.08	0.0021	6.5614	22
spring-web-5.3	7219	11261	3.12	0.0004	7.5364	34
tomcat-embed-c	5386	8158	3.03	0.0006	7.3081	26
zookeeper-3.6.	5284	9651	3.65	0.0007	6.3153	20
PRIVATE	339	529	3.12	0.0092	5.4404	13

Table 6: OOP Programs Metrics 2

Program	Modularity	Communities	Mean CC Coef	Mean LC Coef
akka	0.8347	39	0.0748	0
commons-cli-	0.6542	20	0.1658	0
commons-code	0.7835	67	0.0805	0
commons-csv-	0.8219	29	0.0425	0
commons-emai	0.7564	30	0.0893	0
disruptor-3.	0.6508	99	0.1152	0
ftpserver-co	0.6670	213	0.1232	0
grpc-core-1.	0.6385	779	0.1039	0
guava	0.7583	2025	0.0754	0
hbase-client	0.9013	419	0.0264	0
hsqldb-2.4.1	0.6779	1254	0.1260	0
jackson-data	0.7138	228	0.1261	0
javax.servle	0.8836	111	0.0348	0
jedis-3.4.1	0.7270	586	0.1225	0
jersey-core-	0.8157	191	0.0776	0
jetty-7.0.0.	0.7034	286	0.1097	0
joda-time-2.	0.7110	461	0.1260	0
jsch-0.1.54	0.6709	203	0.1223	0
jsoup-1.13.1	0.7439	204	0.1290	0
junit-4.13.1	0.7672	254	0.0888	0
mail-1.4.7	0.7154	195	0.0872	0
mariadb-java	0.7519	244	0.0965	0
mongo-java-d	0.6193	1731	0.1066	0
mx4j-3.0.2	0.7091	106	0.1061	0
org.eclipse.	0.6403	1404	0.1293	0
pdfbox-2.0.2	0.6890	863	0.1311	0
poi-4.1.2	0.7582	2181	0.0958	0
postgresql-4	0.7695	506	0.0790	0
resteasy-jax	0.6456	888	0.0898	0
runtime-3.10	0.7279	62	0.1229	0
slf4j-api-1.	0.8531	32	0.0670	0
spring-secur	0.6879	216	0.1068	0
spring-web-5	0.6956	1315	0.0871	0
tomcat-embed	0.7786	684	0.0907	0
zookeeper-3.	0.6624	962	0.1052	0
PRIVATE	0.6676	40	0.1739	0

3.2.4 OOP Programs - Summary

Table 7: OOP Programs - Summary Metrics 1

Type	N	E	K	Delta	MGD	Diameter
Min.	133	177	1.760	0.000200	2.824	8.00
1st Qu.	672	912	2.715	0.000700	6.032	16.50
Median	1637	2718	3.120	0.001700	6.434	20.00
Mean	3456	5984	3.094	0.003403	6.489	20.46
3rd Qu.	4351	8149	3.505	0.003500	7.100	24.50
Max.	15971	26910	4.380	0.020300	9.497	34.00

Table 8: OOP Programs - Summary Metrics 2

Type	Modularity	Communities	Mean CC Coef	Mean LC Coef
Min.	0.6193	20.0	0.02640	0
1st Qu.	0.6693	108.5	0.08715	0
Median	0.7110	244.0	0.10610	0
Mean	0.7256	512.0	0.10227	0
3rd Qu.	0.7628	731.5	0.12305	0
Max.	0.9013	2181.0	0.17390	0

3.2.5 Testing Programs

Table 9: Testing Programs Metrics 1

Program	N	E	K	Delta	MGD	Diameter
gogol	339	695	4.10	0.0121	4.9149	10
google-client	942	1316	2.79	0.0030	5.5469	14

Table 10: Testing Programs Metrics 2

Program	Modularity	Communities	Mean CC Coef	Mean LC Coef
gogol	0.6553	24	0.2359	0.1892
google-client	0.7413	178	0.0880	0.0000

3.2.6 Modularity

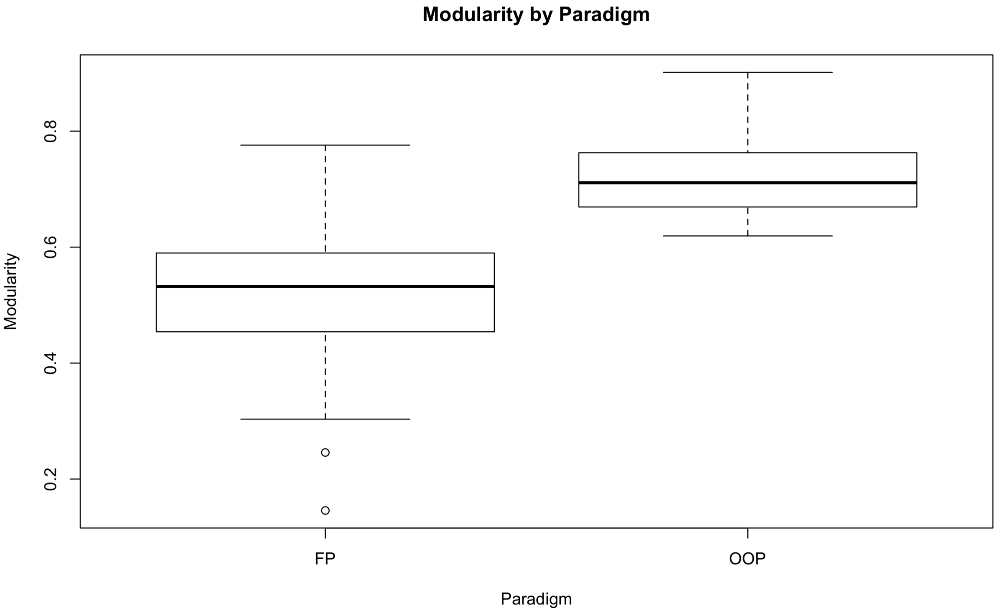


Figure 1: Modularity Metric Comparing both Paradigms

3.2.7 Modularity VS logarithm of lines of code

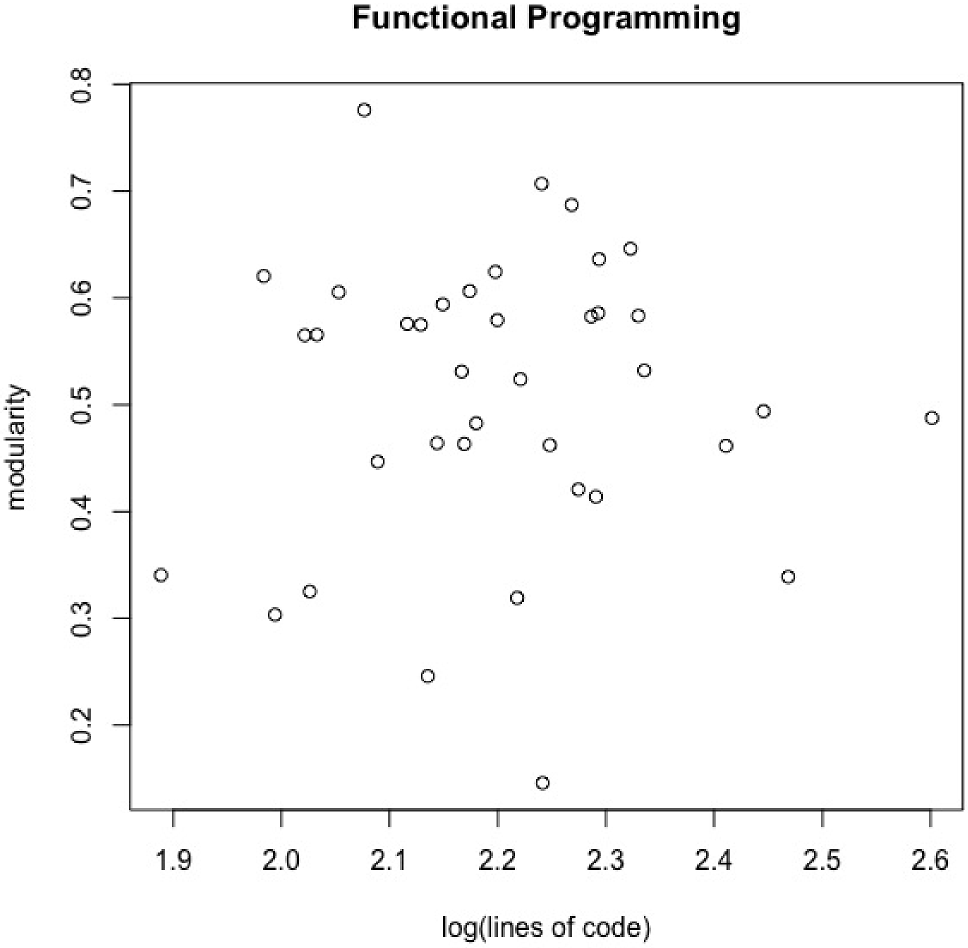


Figure 2: Modularity VS log(Lines of Code) in FP

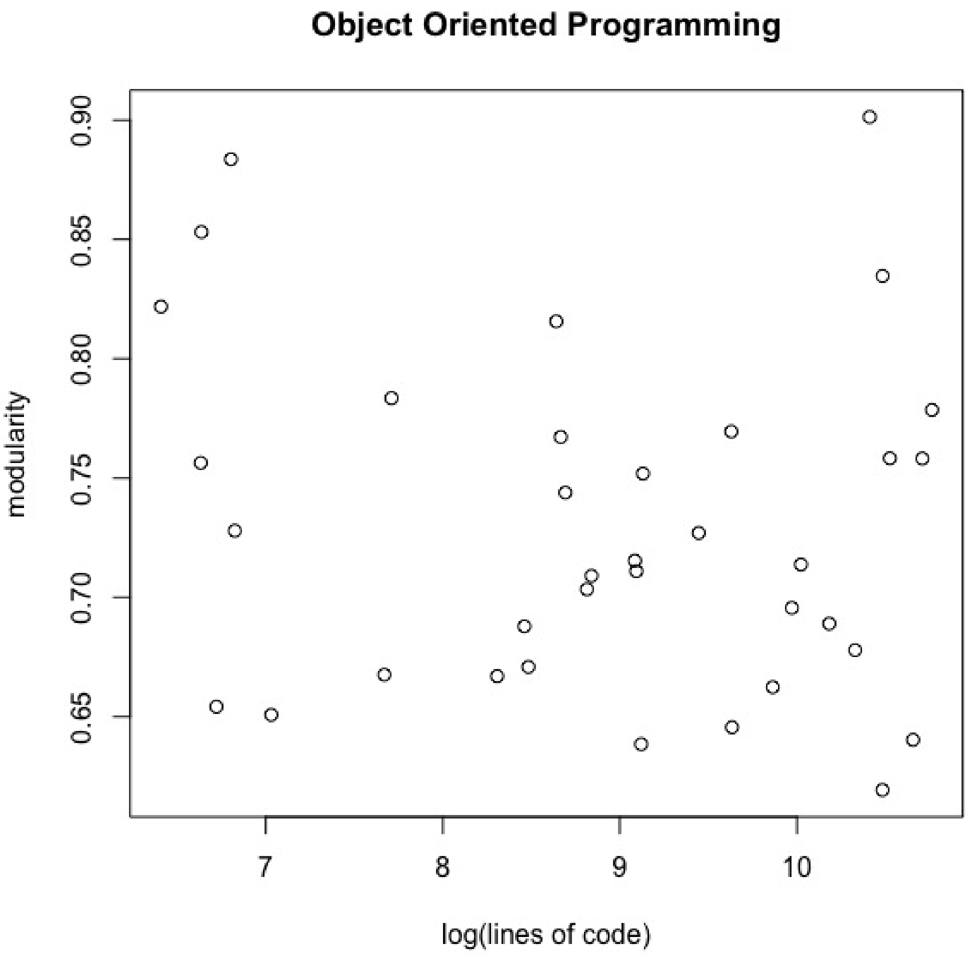


Figure 3: Modularity VS log(Lines of Code) in OOP

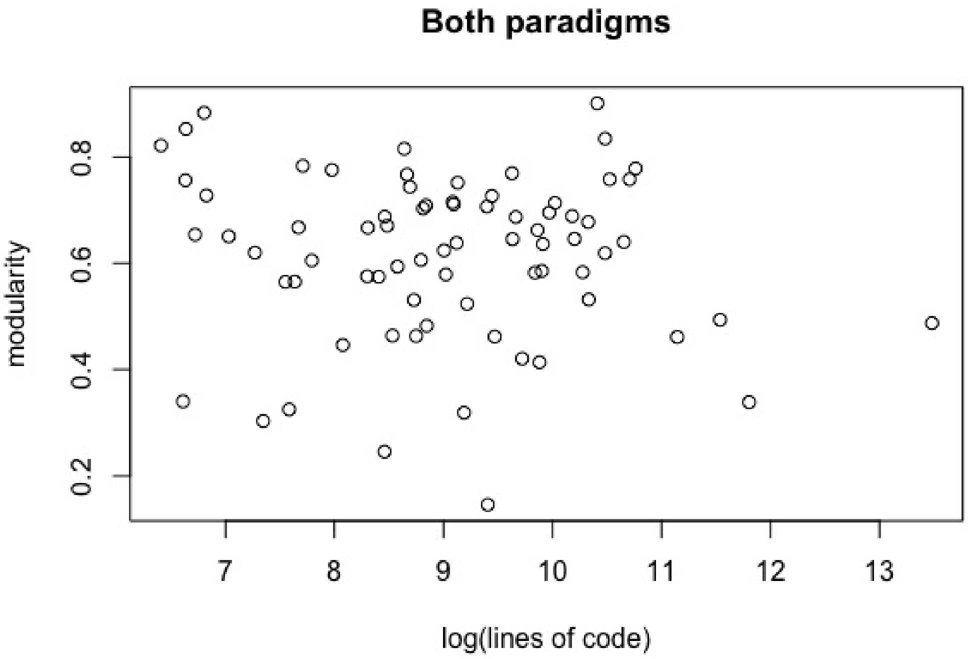


Figure 4: Modularity VS log(Lines of Code) in both paradigms together

3.2.8 Degree Distribution - FP

Table 11: FP Programs - Degree Distribution AIC Values

Program	Disp Poisson	Disp Geom	Zeta 2	Zeta	Zeta Truncated
aeson	1863.67550	0.00000	452.7808249	265.700378	265.584673
amazonka	3616.77605	0.00000	837.9114064	471.041171	468.946567
async	117.32660	0.00000	57.9771735	45.393422	47.431194
attoparsec	201.87476	0.00000	62.3974727	37.698004	39.563825
beam	2991.16891	0.00000	643.2831784	378.697487	378.551372
cabal	15395.53601	0.00000	3433.9869964	1812.260423	1790.841833
co-log	151.95216	0.00000	36.8119509	31.385101	33.434477
conduit	2569.95419	100.73831	22.7817231	0.000000	1.900681
containers	99.07037	0.00000	50.5854332	38.684562	40.727983
criterion	21.21021	0.00000	83.0163027	64.800536	66.762371
cryptol	11695.39923	0.00000	2346.8928801	1255.279725	1243.297796
cryptonite	629.52088	0.00000	221.8455212	151.598973	153.112739
dhall	2852.00284	0.00000	799.2920305	476.202745	474.806324
free	179.18579	0.00000	157.6723429	114.489806	116.181877

Table 11: FP Programs - Degree Distribution AIC Values

Program	Disp Poisson	Disp Geom	Zeta 2	Zeta	Zeta Truncated
haskoin	859.82342	0.00000	580.1156711	416.604701	417.333653
hedgehog	1369.78069	0.00000	414.3882331	276.144396	276.983555
helm	130.05235	0.00000	8.2776028	8.595222	10.714165
hlint	925.12005	0.00000	226.2607732	151.972058	153.412732
lens	15008.87852	0.00000	725.6466415	246.678077	243.899534
liquid	20225.32515	0.00000	2341.8676728	1273.728286	1265.482522
megaparsec	175.20452	0.00000	58.6959541	47.188012	49.192447
mios	183.79848	0.00000	194.2119753	137.095338	138.626423
optparse	487.33560	0.00000	210.9939556	138.873027	140.179431
pandoc	38849.09113	0.00000	5122.5021209	2432.889804	2395.127239
pipes	443.08691	0.00000	53.4193641	29.916636	31.814254
postgresql	1993.14352	0.00000	498.3831077	337.969862	338.637945
PRIVATE	2183.28655	0.00000	865.8009952	629.735886	630.170281
protolude	282.15330	0.00000	17.5198441	10.981453	13.017689
QuickCheck	895.58710	0.00000	320.8080028	211.359256	212.284991
reflex	640.20474	0.00000	147.4878994	112.081878	113.896163
relude	403.01353	20.92190	0.0000000	1.975481	4.008396
servant	477.06345	0.00000	109.9038212	81.117733	82.985295
snap	503.40964	0.00000	179.8183309	126.887624	128.531432
stm	185.14582	0.00000	66.8080777	45.693600	47.618021
summoner	169.18845	0.00000	245.7948243	174.595515	175.964707
text	485.77068	16.19161	0.3142174	0.000000	2.066109
vector	8101.09215	0.00000	870.8989597	359.858763	349.192728
yesod	900.84725	0.00000	271.9407233	198.458638	199.9
92093					

3.2.9 Degree Distribution - OOP

Table 12: OOP Programs - Degree Distribution AIC Values

Program	Disp Poisson	Disp Geom	Zeta 2	Zeta	Zeta Truncated
akka	849.6612	145.357181	20.536579	0.0000000	2.0277241
commons-cli	182.6214	0.000000	17.833938	18.3940841	20.4455174
commons-cod	645.1480	0.000000	60.343310	57.0577817	59.0402410
commons-csv	228.1798	3.582231	0.000000	0.2724416	2.3199915
commons-ema	351.5918	34.202642	0.174698	0.0000000	2.0437682
disruptor-3	783.4840	0.000000	31.659939	28.4734438	30.4552226
ftpserver-c	5580.8989	454.037696	33.448005	0.0000000	1.8283031

Table 12: OOP Programs - Degree Distribution AIC Values

Program	Disp Poisson	Disp Geom	Zeta 2	Zeta	Zeta Truncated
grpc-core-1	10835.3838	91.507957	199.586897	0.0000000	1.0972223
guava	33415.1211	783.739138	225.652920	0.0000000	0.9292341
hbase-clien	3936.7613	567.812973	111.768218	0.0000000	2.0037783
hsqldb-2.4.	60827.4576	3821.349841	769.117288	1.2167069	0.0000000
jackson-dat	6424.7809	477.549079	25.591008	0.0000000	1.8459512
javax.servl	648.7272	41.188374	28.799780	0.0000000	2.0181284
jedis-3.4.1	24909.2180	1073.532091	25.987669	0.0000000	1.7579590
jersey-core	2361.8926	0.000000	171.982172	143.2274926	145.0695584
jetty-7.0.0	4951.7062	213.122136	34.849186	0.0000000	1.8033043
joda-time-2	14344.2295	0.000000	626.523775	150.6451182	150.2312534
jsch-0.1.54	5288.0113	463.490503	25.704087	0.0000000	1.8569702
jsoup-1.13.	7396.7715	234.743571	173.210436	0.0000000	1.1219573
junit-4.13.	2547.7632	0.000000	164.405825	148.4537670	150.3365146
mail-1.4.7	2207.8988	43.976252	4.568415	0.0000000	1.9457354
mariadb-jav	5300.1980	0.000000	76.180868	5.4932517	7.1466141
mongo-java-	35497.2118	1675.279358	388.717083	0.0000000	0.3647674
mx4j-3.0.2	1979.2480	106.195916	0.000000	0.5395893	2.5146249
org.eclipse	55229.8134	2812.484850	646.961320	0.5943077	0.0000000
pdfbox-2.0.	48218.2095	3269.391476	338.721141	0.0000000	0.5533084
poi-4.1.2	64266.0959	4206.922778	338.744236	0.0000000	0.5289241
postgresql-	8332.5738	308.784490	40.648056	0.0000000	1.7415784
PRIVATE	971.4943	81.811881	1.688681	0.0000000	1.9977647
resteasy-ja	14213.8959	827.432359	68.805525	0.0000000	1.6086073
runtime-3.1	988.5741	59.818409	0.000000	1.8914142	3.9009734
slf4j-api-1	164.9488	0.000000	35.481064	36.5668486	38.6104244
spring-secu	3276.5518	0.000000	51.629722	14.4958942	16.3027900
spring-web-	20990.6186	1020.625763	120.106592	0.0000000	1.3590405
tomcat-embe	18388.7792	1354.552719	35.217930	0.0000000	1.6975627
zookeeper-3	24309.6424	1862.468736	153.995372	0.0000000	1.2761295

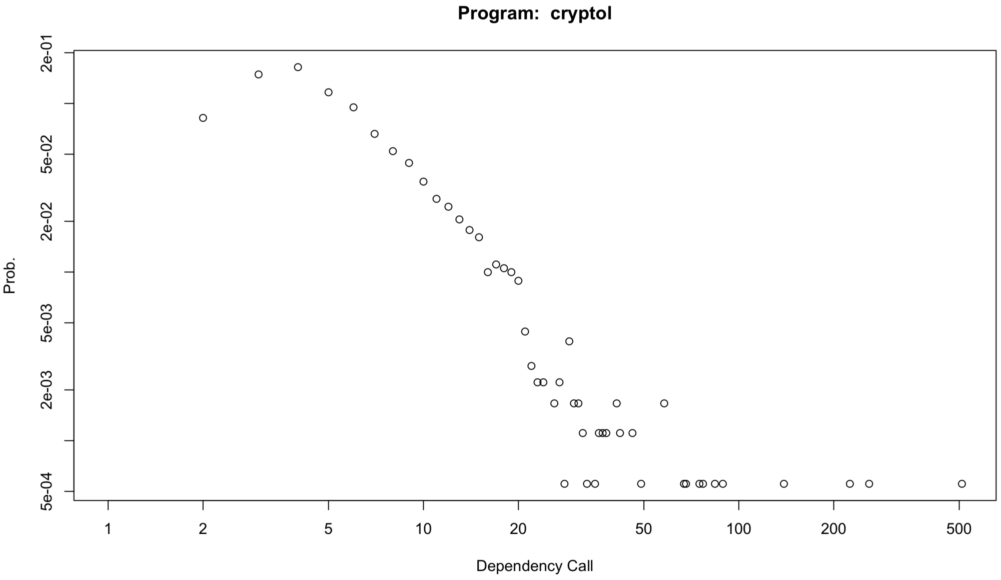


Figure 5: Example of FP Degree Distribution

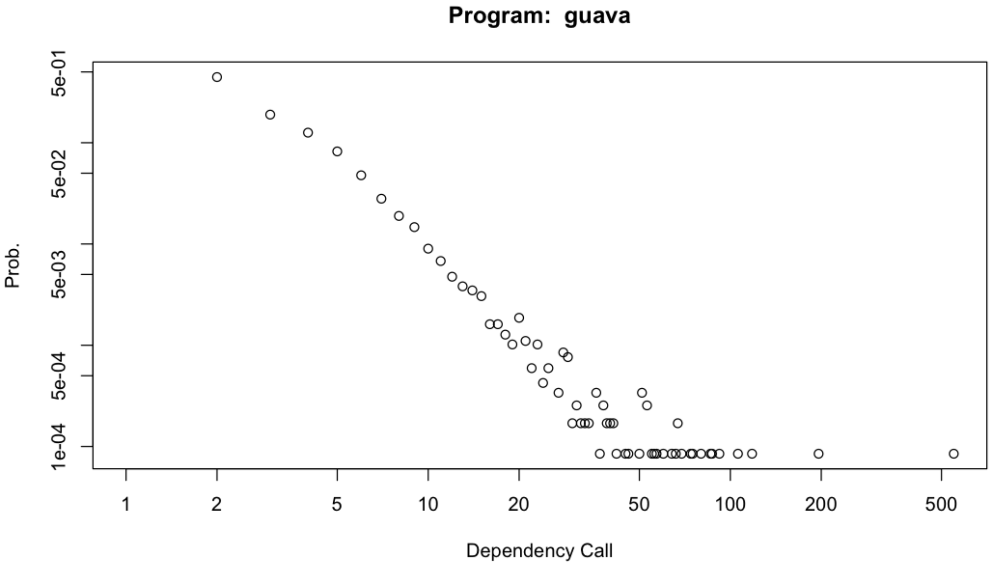


Figure 6: Example of OOP Degree Distribution

4 Discussion and Analysis

Here we are going to discuss the results obtained in our experiments.

4.1 Hypothesis 1

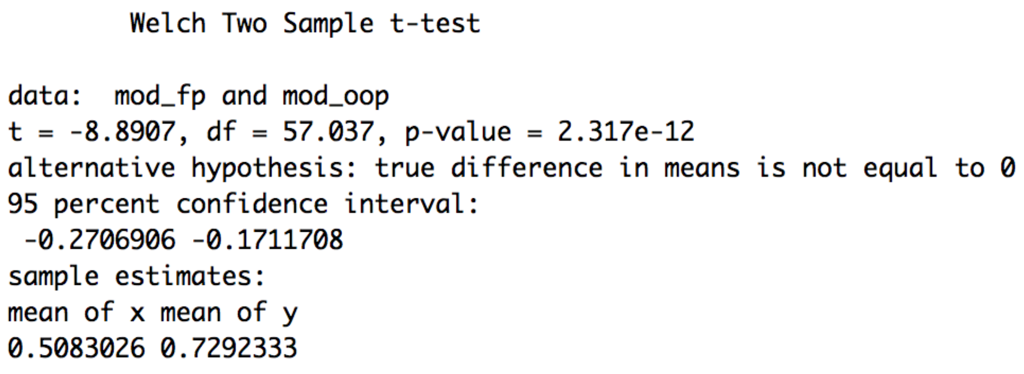
Regarding the first hypothesis 1, as we can see in 3 and 7, and taking the test programs 9 which the first one is Functional Programming and the other is Object Oriented Programming, we can see that in the case of Functional Programming Modularity is slightly above the *3rd Quartile* with 0.6553, so we can consider that this Hypothesis hold for this program because it is even better modularize. In the case of Object Oriented Programming also the modularity holds the hypothesis with 0.7413.

We are aware that for having a more accurate test, we should have tested with a lot of programs, but due to the difficulty of generating the graphs and the time involved it was not possible to achieve that goal which can be lead to future works.

4.2 Hypothesis 2

Functional Programming average **modularity** is 0.5097 whereas Object Oriented Programming average **modularity** is 0.7256. This is an increase of 42%. The lower value in Object Oriented Programming is 0.6193 and the lower in Functional Programming (maybe an outlier) is 0.1458.

We do an statiscal t-test to accept or reject our hypothesis. The null hypothesis is that both paradigms have almost equal average modularity. The alternative one is that one is bigger than the other:



```
        Welch Two Sample t-test

data:  mod_fp and mod_oop
t = -8.8907, df = 57.037, p-value = 2.317e-12
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -0.2706906 -0.1711708
sample estimates:
mean of x mean of y
0.5083026 0.7292333
```

Figure 7: T test output (executed in R environment).

As we can see we accept our hypothesis as p-value is lower than 0.05.

4.3 Hypothesis 3

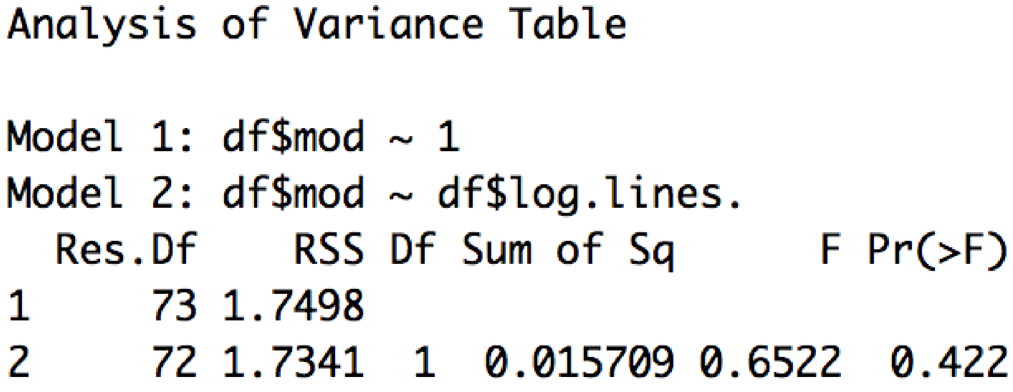
As we can see in this figure 2 and 3 we have to reject our Hypothesis as we can't see that there exist a clear relationship between this 2 variables. It is enough to check visually the images to make this statement.

But if we want to be more rigorous we will perform a statistical test to confirm it. Here the Null model is that this two variables are independent. So our alternative model that we propose is a linear model like this:

$$M = a + b \times LOC \tag{1a}$$

Where M is **Modularity** and LOC is Lines of Code.

So we perform an ANOVA Variance Analysis (ANOVA) test between this 2 models and the results are displayed in the following picture:



```
Analysis of Variance Table

Model 1: df$mod ~ 1
Model 2: df$mod ~ df$log.lines.
  Res.Df    RSS Df Sum of Sq      F Pr(>F)
1     73 1.7498
2     72 1.7341  1  0.015709 0.6522  0.422
```

Figure 8: ANOVA test output (executed in R environment).

As we can see we reject our hypothesis as p-value is greater than 0.05.

Our hypothesis was based on the intuition that when a program becomes bigger in terms of lines of code it has to be more modularized in order to be able to maintain it as developers.

4.4 Hypothesis 4

Regarding hypothesis 4, and after running the Akaike Information Criterion analysis according to what have studied on Lab 2, we can see that for Functional Programming the hypothesis cannot be accepted, because according to the results in 11, Displaced Geometric Model is the most suitable distribution that fits the data.

On the other hand and regarding Object Oriented Programming, we have been able to prove using Akaike Information Criterion technique, that almost all the programs fits into a Zeta Model, i.e. **power-law** like 12.

Moreover, this is another metric that supports our assumption about the better modularization of Object Oriented Programming related that Functional Programming.

5 Conclusions

We started this project inspired by [You79] where they used a refined modularity metric in order to asses the quality of Low Coupling and High cohesion.

We have focused our study not only in this mentioned aspect but also in:

- Trying to see other typical properties that good-design-software has, as power-law-like degree distribution in its Call dependency graph.
- Capturing differences between different software paradigms in terms of modularity.
- Trying to find some relationship between the 2 metrics of a software program (lines of code and modularity).

In order to do this project, a dataset of programs was needed and the collection of it it was the most tedious aspect in the project. This collection has been made by hand in different online repositories mentioned in the References.
For instance in the case of Java programs, to generate the Call dependency graph, a jar file was needed and it is not common to find them online.
An alternative was downloading the source files a packaging them but this is not a scalable method if you want to have a considerable amount of observations in your dataset as a lot of problems usually appear during compilation and packaging to should be solved meticulously.

Possibly a good idea could be develop a tool that can generate the CDG from the source files.

This is the reason why other hypothesis that were proposed at the fist moment as finding the relationship between modularity and number of code refactors in the code (counted with number of commits) or finding the relationship between modularity and number of contributors in a project were discarded (Github provides this 2 metrics for each project, i.e, number of commits and number of contributors).

References

[Cen21] Maven Central. Maven. https://repo1.maven.org, 2021.

[dus19] dustinnorwood. Haskell function callgraph. https://github.com/dustinnorwood/function-call-graph, 2019.

[gou18] gousiosg. Java callgraph. https://github.com/gousiosg/java-callgraph, 2018.

[Inc20] Stack Exchange Inc. 2020 development survey. https://insights.stackoverflow.com/survey/2020, 2020.

[Inc21] GitHub Inc. Github. https://github.com, 2021.

[You79] E. Yourdon. *Structured Design Fundamentals of a Discipline of-Computer Program and Systems Design.* Prentice-Hall Inc, Reno, 1979.

[ZWXY18] Jie Zhang, Jiajing Wu, Yongxiang Xia, and Fanghua Ye. Measuring cohesion of software systems using weighted directed complex networks. pages 1–5, 05 2018.

A Organization

- **code**: Under this folder you are going to find *R Scripts* code for conducting this analysis.
- **fp_graphs**: *DOT* files that contains the Dependency Graph Representation of each Functional Programming Program
- **oop_graphs**: *DOT* files that contains the Dependency Graph Representation of each Object Oriented Programming Program
- **report**: This report in Latex and PDF format.

B Programs Details - Line of Codes

B.1 FP Programs

Table 13: FP Analyzed Programs

Program	LoC
aeson	6948
amazonka	715531
async	743
attoparsec	4718
beam	20151
cabal	102525
co-log	1436
conduit	12963
containers	19556
criterion	2421
cryptol	30740
cryptonite	18763
dhall	29058
free	4472
fused-effects	4145
ghcid	1664
haskoin	12066
hedgehog	8277
helm	2071
hlint	6306
lens	16691
liquid	133740
megaparsec	8144
mios	6178
mtl	932
optparse	3220
pandoc	69179
pipes	1969
postgresql	6596
protolude	1901
quickcheck	5077
reflex	10062
relude	2913
servant	15725

Table 13: FP Analyzed Programs

Program	LoC
snap	5310
stm	1550
summoner	4025
text	9783
vector	12166
yesod	19971
PRIVATE PROGRAM	26975

B.2 OOP Analytzed Programs

Table 14: OOP Analyzed Programs

Program	LoC
akka-actor_2.10-2.3.9	35702
commons-cli-1.4	830
commons-codec-1.10	2231
commons-csv-1.8	607
commons-email-1.4	760
disruptor-3.4.2	1131
ftpserver-core-1.0.6	4052
grpc-core-1.34.1	9142
guava-28.1-jre	37216
hbase-client-2.4.0	33209
hsqldb-2.4.1	30578
jackson-databind-2.12.0	22516
javax.servlet-api-4.0.1	901
jedis-3.4.1	12640
jersey-core-1.19.4	5656
jetty-7.0.0.pre5	6727
joda-time-2.10.6	8891
jsch-0.1.54	4833
jsoup-1.13.1	5955
junit-4.13.1	5803
mail-1.4.7	8817
mariadb-java-client-2.7.1	9229
mongo-java-driver-3.12.7	35676
mx4j-3.0.2	6902
org.eclipse.jgit	42417

Table 14: OOP Analyzed Programs

Program	LoC
pdfbox-2.0.22	26413
poi-4.1.2	44691
postgresql-42.2.18	15199
resteasy-jaxrs-3.14.0.Final	15267
runtime-3.10.0-v20140318-2214	921
slf4j-api-1.7.30	763
spring-security-core-5.4.2	4726
spring-web-5.3.2	21393
tomcat-embed-core-10.0.0	47185
zookeeper-3.6.2	19207
PRIVATE PROGRAM	2143